A Normal Form Characterization for Efficient Boolean Skolem Function Synthesis

Preey Shah, Aman Bansal, S. Akshay and Supratik Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology Bombay, Mumbai, India.
Email: {preeyshah, aman0456b}@gmail.com; {akshayss, supratik}@cse.iitb.ac.in

Abstract—Boolean Skolem function synthesis concerns synthesizing outputs as Boolean functions of inputs such that a relational specification between inputs and outputs is satisfied. This problem, also known as Boolean functional synthesis, has several applications, including design of safe controllers for autonomous systems, certified QBF solving, cryptanalysis etc. Recently, complexity theoretic hardness results have been shown for the problem, although several algorithms proposed in the literature are known to work well in practice. This dichotomy between theoretical hardness and practical efficacy has motivated research on normal forms of specification representation that guarantee efficient synthesis, thus partially explaining the efficacy of some of these algorithms.

In this paper we go one step further and ask if there exists a normal form representation of the specification that precisely characterizes “efficient” synthesis. We present a normal form called SAUNF that answers this question affirmatively. Specifically, a specification is polynomial time synthesizable iff it can be compiled to SAUNF in polynomial time. Additionally, a specification admits a polynomial-sized functional solution iff there exists a semantically equivalent polynomial-sized SAUNF representation. SAUNF is exponentially more succinct than well-established normal forms like BDDs and DNNFs, used in the context of AI problems, and strictly subsumes other more recently proposed forms like SynNNF. It enjoys compositional properties that are similar to those of DNNF. Thus, SAUNF provides the right trade-off in knowledge representation for Boolean functional synthesis.

I. INTRODUCTION

The history of Skolem functions can be traced all the way back to the 1920’s when Thoralf Skolem provided a simplified proof of the celebrated Löwenheim Skolem theorem in first order logic. A key step in the proof showed that any first order logic formula can be converted into Skolem normal form, that has no existential quantifiers, while preserving satisfiability. This process, called Skolemization, involves replacing existentially quantified variables by terms constructed out of new function symbols, called Skolem functions. Skolemization has been an immensely influential technique in logic, and is now used routinely in many applications, viz. automated theorem proving. While it suffices for some applications to merely know that desired Skolem functions exist, others require us to efficiently synthesize such Skolem functions.

Algorithmic synthesis of Skolem functions has been studied extensively in the Boolean setting. Given disjoint sequences of Boolean variables \( I = (i_1, \ldots, i_n) \) and \( X = (x_1, \ldots, x_m) \), representing inputs and outputs respectively of a system, and given a Boolean formula \( \varphi(X, I) \) specifying a desired relation between the system inputs and outputs, the **Boolean Skolem function synthesis** (BfNS) problem asks us to synthesize a sequence of formulas \( \Psi(I) = (\psi_1(I), \ldots, \psi_m(I)) \) that can be substituted for \( X \) to satisfy the specification, i.e., \( \forall I (\varphi(\Psi(I), I) \iff \exists X \varphi(X, I)) \). The formulas in \( \Psi \) indeed represent Boolean Skolem functions for \( X \) in \( \exists X \varphi(X, I) \).

The above problem, also referred to as **Boolean functional synthesis** in the literature, has several applications; we will just mention two here. Skolem functions (and their counterparts, called Herbrand functions) can be thought of as “certificates” that help us independently verify the results of satisfiability checking for Quantified Boolean Formulas, as done in [1]. QBF-satisfiability solving is used today in diverse applications, from planning to program repair to reactive synthesis and the like [2]. Having certificates not only helps in verifying correctness of QBF-satisfiability results, but also has other benefits like providing a feasible plan in a planning problem. Yet another application of Skolem functions is motivated by cryptanalysis. Consider a system with a single 2\( n \)-bit unsigned integer input \( I \), and two \( n \)-bit unsigned integer outputs \( X_1 \) and \( X_2 \). Suppose the relational specification is given as \( \varphi_{rel}(X_1, X_2, I) \equiv ((I = X_1 \times_{[n]} X_2) \land (X_1 \neq 1) \land (X_2 \neq 1)) \), where \( \times_{[n]} \) denotes \( n \)-bit unsigned integer multiplication and \( I \) denotes an \( n \)-bit representation of the integer \( 1 \). This specification can be represented as a Boolean formula of size \( O(n^2) \) over the variables in \( I, X_1 \) and \( X_2 \). Finding Skolem functions for \( X_1 \) and \( X_2 \) in terms of \( I \) effectively asks us to solve the \( (n\text{-bit}) \) factorization problem. Note that if \( I \) represents a prime number, there are no values of \( X_1, X_2 \) that satisfy the specification. Hence the specification is technically unrealizable; yet, it is of significant interest (e.g. in cryptanalysis) to synthesize Skolem functions for \( X_1, X_2 \) that can be evaluated efficiently. Note that it is an open question whether there are polynomial time algorithms or even non-uniform polynomial sized circuits for integer factorization.

Given its significance, the Boolean Skolem function synthesis problem has received considerable attention over the last two decades, with a lot of work focussed towards design...
of practically efficient algorithms [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. These algorithms, using techniques ranging from CEGAR to decision tree learning, empirically work well on large collections of benchmarks, but fail remarkably for some small benchmarks. Further, and somewhat surprisingly, each tool seems to work well on a different set of benchmarks, often incomparable across tools. What is common among the approaches, however, is that it is difficult to predict reliably the set of benchmarks, i.e., the class of formulas, on which a particular algorithm will be efficient (other than simple cases or heuristic guesses). On a related note, a theoretical study undertaken in [12] showed that Boolean Skolem function synthesis requires super-polynomial space and time unless some well-regarded complexity-theoretic conjectures are falsified. In fact, [12] also showed that under some weaker assumptions, there cannot exist even sub-exponential algorithms for this problem.

This leads to a curious dichotomy of theoretical worst-case hardness vs practical (sometimes unreasonable) efficiency. To resolve this dichotomy, researchers have searched for structure in the input specification that can result in provably efficient synthesis. It turns out that the representation used for input specification and Skolem functions indeed has a bearing on the complexity of synthesis. For example, if the specification is given as a ROBDD [14] with input-first variable ordering (see [7] for details), there exists a polynomial-time algorithm that generates Skolem functions as ROBDDs [7]. In [15], a new normal form for specifications, called SynNNF, was proposed, which ensures polynomial-time synthesis, assuming both the specification and Skolem functions are represented as arbitrary Boolean circuits. However, these earlier studies only provide sufficient but not necessary conditions for efficient Skolem function synthesis. Significantly, it is not the case that every class of specifications that admit efficient Skolem function synthesis can be efficiently compiled to ROBDDs with input-first variable ordering, or even to SynNNF. Indeed, [15] gives (counter-)examples of specifications that are not in SynNNF but admit efficient Skolem function synthesis.

In this paper, we address the above dichotomy, by presenting a normal form for Boolean circuits, called SAUNF (acronym for Subset-And-Unrealizable Normal Form), that characterizes polynomial-time and polynomial-sized Boolean Skolem function synthesis. By a characterization, we mean that for every class $C$ of circuits (i) Skolem functions can be synthesized in polynomial-time for specifications represented by circuits in $C$ if these circuits can be compiled to semantically equivalent ones in SAUNF in polynomial-time, and (ii) specifications represented by circuits in $C$ admit polynomial-sized Skolem functions if they can be compiled into polynomial-sized semantically equivalent circuits in SAUNF. This notion is made precise later in Section III. We also explore the proposed normal form in depth, and present several interesting results. Our main contributions are the following.

- We present a polynomial-time algorithm to synthesize polynomial-sized Skolem functions from specifications in SAUNF.
- We study compositional properties of SAUNF including disjunction and conjunction operations.
- We show that checking membership in SAUNF is Co-NP hard and is in the second level of the polynomial hierarchy.
- We present a novel algorithm for compiling a Boolean relational specification in CNF to SAUNF.

Finally, we show an interesting application of SAUNF. Specifically, we show that in the context of the $n$-bit factorization problem mentioned earlier, there exist polynomial-sized SAUNF circuits relating specific bits of the input $I$ to the outputs $X_1$ and $X_2$. While this does not solve the $n$-bit factorization problem, it is worth noting that some of these bit relations are known to require exponentially large ROBDDs [17], and sub-exponential sized circuits using normal forms like DNNF, dDNNF, SynNNF are not yet known.

Normal forms for Boolean functions, or knowledge representation in general, have been investigated extensively over the last few decades [13], [19], [20], [21]. While a problem compiled to a normal form may allow the problem to be solved efficiently, compilation to the normal form may not always be easy. For instance, DDNF allows polynomial-time model counting, but converting to dDNNF cannot always be done in polynomial time unless $P = \#P$. Despite the worst-case complexity of the compilation process, research in normal forms offers several benefits, such as better understanding of compositionality and other structural properties, explanations for practical performance of algorithms (e.g., on benchmarks in a normal form that permits efficient analysis) etc. Furthermore, the study of normal forms also feeds into research on normal form compilers, that have significant practical use. For example, multiple dDNNF compilers have been developed since the introduction of dDNNF as a useful normal form. We also point out that different types of normal forms have been studied earlier. For example, syntactic or purely structural normal forms like CNF, DNF, DDNF allow efficient membership checking, while semantic normal forms like dDNNF require propositional satisfiability checks to determine membership. The proposed normal form (SAUNF) falls in the latter category, but like dDNNF, is worth studying for the good properties it exhibits.

The remainder of this paper is organized as follows. We start with preliminaries in Section II and problem statement in Section III. In Section IV, we introduce SAUNF and compare it with other normal forms in Section V. In Section VI, we explain how Skolem functions can be efficiently computed from SAUNF specifications, and discuss compositionality properties in Section VII. Next, in Section VIII, we describe an algorithm to compile a CNF formula to SAUNF. Finally, we show applications to $n$-bit factorization in Section IX and conclude in Section X.
variables. We use \( \{ \top \} \) to denote the sequence of variables \( V \). We use \( \top \) and \( \bot \) to represent the Boolean constants true and false respectively. A Boolean formula \( \varphi \) over \( V \) is defined by the grammar:

\[
\varphi :: = \neg \varphi \mid (\varphi \land \varphi) \mid \varphi \lor \varphi \mid \top \mid \bot \mid v_1 \mid \cdots \mid v_r .
\]

We write \( \varphi(V) \) to denote that the formula \( \varphi \) is defined over the sequence of variables \( V \). Special cases of formulas include clauses or disjunctions of literals, and cubes or conjunctions of literals. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses. Similarly, it is in disjunctive normal form (DNF) if it is a disjunction of cubes. A Boolean function \( f(V) \) is a mapping \( \{ \top, \bot \}^{|V|} \to \{ \top, \bot \} \). The semantics of the Boolean formula \( \varphi(V) \) is given by a Boolean function \( \varphi(V) : \{ \top, \bot \}^{|V|} \to \{ \top, \bot \} \). It is easy to see that every Boolean function \( f \) corresponds to at least one Boolean formula \( \varphi \) such that \( \varphi = f \).

Let \( U \) be a sub-sequence of \( V \) (this includes the possibility \( U = V \)), and let \( V \setminus U \) denote the sequence obtained by removing from \( V \) all variables present in \( U \). An assignment of \( U \) is a mapping \( \sigma : \text{set}(U) \to \{ \top, \bot \} \). We use \( \varphi[\sigma] \) to denote the Boolean function \( \{ \top, \bot \}^{|V \setminus U|} \to \{ \top, \bot \} \) obtained by substituting \( \sigma(v) \) for every variable \( v \in \text{set}(U) \) in \( \varphi \). We say that the formula \( \varphi(V) \) reduces to the formula \( \varphi'(V \setminus U) \) under the assignment \( \sigma \) of \( U \) if \( \varphi[\sigma] = \varphi'[\sigma] \). We say that \( \sigma \) satisfies \( \varphi \) if \( \varphi[\sigma] \) always evaluates to \( \top \).

We choose to represent both Boolean functions and Boolean formulas (modulo semantic equivalence) by Boolean circuits. For purposes of this paper, a Boolean circuit (or simply a circuit) is a rooted directed acyclic graph (DAG) \( G \) in which nodes with incoming edges, also called internal nodes, are labeled by \( \lor, \land \), and \( \neg \) operators, and nodes with no incoming edges, also called leaves, are labeled either by variables in \( V \) or by constants in \( \{ \top, \bot \} \). Every internal node labeled \( \land \) or \( \lor \) has incoming edges from exactly two children, while every internal node labeled \( \neg \) has an incoming edge from exactly one child. In order to ensure that a circuit doesn’t have superfluous nodes, we require all nodes in a circuit to be descendants of the root. The size of a circuit \( G \), denoted \( |G| \), is the number of nodes in \( G \). A circuit \( G \) represents a Boolean formula \( \varphi_G \) (alternatively, a Boolean function \( \varphi[G] \), if \( G \) is used to represent a Boolean function) defined as follows: (i) if \( G \) consists of a single leaf labeled \( \lambda \), then \( \varphi_G = \lambda \); (ii) if the root of \( G \) is labeled \( \lor \), \( \land \) or \( \neg \), then \( \varphi_G = \varphi_G' \lor \varphi_G'' \lor \neg \varphi_G' \lor \neg \varphi_G'' \); (iii) if the root of \( G \) is labeled \( \neg \) and if the sub-circuit rooted at its (only) child \( \lambda \), then \( \varphi_G = \neg \varphi_H \).

A Boolean formula is said to be in negation normal form (or NNF) if the application of \( \neg \) is restricted to only the variables. Motivated by this, a circuit in which every \( \neg \) labeled node has a leaf labeled by a variable as its child is said to be an NNF circuit. It is well-known that every Boolean formula is semantically equivalent to a formula in NNF. Since we wish to reason about Boolean formulas/functions modulo semantic equivalence, it suffices to restrict our attention to NNF circuits. For convenience of exposition, all circuits in the remainder of the paper are assumed to be in NNF, unless stated otherwise.

It is easy to see that an arbitrary Boolean circuit \( G \) can be converted to an NNF circuit \( G' \) such that \( \varphi_G = \varphi_G' \), and \( |\varphi_G'| \leq 2 \times |\varphi_G| \). For notational convenience, we treat a \( \neg \) labeled node with a child labeled \( \lambda \) in a NNF circuit, as a new leaf labeled \( \neg \lambda \). Thus, an NNF circuit can be viewed as a rooted DAG with \( \land \) and \( \lor \)-labeled internal nodes and leaves labeled by literals over \( V \). Note that popular representations of Boolean functions, viz. lists of (implicitly conjointed) clauses, lists of (implicitly disjoined) cubes, and-inverter graphs [22], ROBDDs [14], DNNF/dDNNF circuits [19], [16] etc. can all be translated to NNF circuits in linear time. Figure 1 shows an example of an NNF circuit. In this figure, the annotations \( G \), \( G_1 \) and \( G_2 \) represent the (sub-)circuits rooted at the nodes adjacent to the annotations. The leaves are designated \( L_0 \) through \( L_{15} \) from left to right. As is the case in this figure, multiple leaves of a circuit may have the same literal label.

Let \( L \) be a subset of leaves of circuit \( G \). We say \( L \) is literal-consistent in \( G \) if every leaf in \( L \) is labeled by the same literal. For a literal-consistent set \( L \) of leaves in \( G \), and for \( b \in \{ \top, \bot \} \), we use \( G \mid _{L,b} \) to denote the circuit obtained by relabeling each leaf of \( G \) in the set \( L \) with \( b \). For a literal \( \ell \) over \( V \), we use the term \( \ell \)-leaves of \( G \) to denote the set of all leaves of \( G \) labeled \( \ell \). For a set of distinct literals \( \{ \ell_1, \ldots, \ell_t \} \) and (possibly same) labels \( b_1, \ldots, b_t \), we abuse notation and use \( G \mid _{\ell_1=b_1, \ldots, \ell_t=b_t} \) to denote the circuit obtained by relabeling all \( \ell_j \)-leaves of \( G \) by \( b_j \), for all \( j \in \{ 1, \ldots, r \} \). Note that since \( \ell \) and \( \ell \) are different literals, the notation \( G \mid _{\ell=b, \ell=b} \) is meaningful (and useful), and represents the circuit obtained by re-labeling all \( \ell \)-leaves and \( \ell \)-leaves of \( G \) by \( b \).

Let \( I = \langle i_1, \ldots, i_n \rangle \) and \( X = \langle x_1, \ldots, x_m \rangle \) be disjoint sequences of Boolean variables representing inputs and outputs, respectively, of a hypothetical system. For clarity of exposition, we use “system inputs” to refer to \( I \), and “system outputs” to refer to \( X \). Consider a circuit \( G \) with leaves labeled by \( \text{lits}(X) \) and \( \text{lits}(I) \). The formula \( \varphi_G(I,X) \) represents a relational specification over the system inputs \( I \) and system outputs \( X \). Given \( G \), the Boolean Skolem Function Synthesis or BFnS problem requires us to find a sequence of Boolean formulas \( \Psi(I) = \langle \psi_1(I), \ldots, \psi_m(I) \rangle \) such that \( \forall I \) \( \varphi_G(I,X) \Leftrightarrow \exists X \varphi_G(X,I) \). As seen earlier, this is an important problem with diverse applications. We call \( \psi_j(I) \) a Skolem function\(^2\) for \( x_j \) in \( \varphi_G(X,I) \), and the sequence (or vector) of all such Skolem functions for \( x_1, \ldots, x_m \) a Skolem function vector for \( X \) in \( \varphi_G(X,I) \). Since we have chosen to represent all Boolean formulas and functions as circuits, we require each \( \psi_j(I) \) to be presented as a circuit.

\(^2\)Technically, \( \psi_j[I] \) is the Boolean Skolem function for \( x_j \) in \( \varphi_G \). However, since we represent both Boolean functions and formulas as circuits, we use \( \psi_j[I] \) and \( \psi_j[I] \) interchangeably for Boolean Skolem functions, to keep the notation simple.
Example 1. Let $X = (x_1, x_2)$ and $I = (i)$. Let $G$ be the circuit shown in Figure 7. Then $\varphi_G(X, I)$ is a relational specification over $I$ and $X$, and one (of possibly many) Skolem function vectors for $X$ in $\varphi_G$ is $\Psi(G(I)) = (\psi_1(I), \psi_2(I))$, where $\psi_1(I) = \neg i = \psi_2(I)$. Indeed, it can be verified that $\forall I (\varphi_G(\Psi(I), I) \iff \exists X \varphi_G(X, I))$.

III. PROBLEM STATEMENT

Earlier work [12] has established (conditional) time and space lower bounds for BFNS; therefore it is unlikely that efficient algorithms exist for solving this problem in general. Yet, several recent works [12], [7], [5], [23], [13], [24] have shown that BFNS indeed admits practically efficient solutions for several non-trivial benchmarks. This motivates us to ask the following question, where we are interested in Boolean circuit representations of relational specifications and Skolem functions.

Does there exist a class, say $C^*$, of circuits such that the following hold?

P0: For every circuit $G$, there is a semantically equivalent circuit $G^* \in C^*$, i.e. $[\varphi_G] = [\varphi_{G^*}]$. In other words, $C^*$ is not semantically constraining.

P1: BFNS is solvable in polynomial-time for the class $C^*$.

P2: For every circuit $C$ of circuits,

P2a: BFNS is solvable in polynomial-time for the class $C$ iff circuits in $C$ can be compiled to semantically equivalent ones in $C^*$ in polynomial-time.

P2b: Relational specifications represented by circuits in $C$ admit polynomial-sized Skolem function vectors iff circuits in $C$ admit polynomial-sized semantically equivalent circuits in $C^*$.

We answer the above question positively in this paper, effectively providing a circuit normal form characterization of efficient Boolean Skolem function synthesis. In light of our characterization, the hardness results of [12] translate to the hardness of computing $G^* \in C^*$ such that $[\varphi_{G^*}] = [\varphi_G]$.

IV. A NORMAL FORM FOR SYNTHESIS

Let $G$ be a circuit with leaves labeled by $\text{lits}(I)$ and $\text{lits}(X)$. Let $\ell$ be a literal labeling a leaf of $G$, and let $v, w$ be the underlying variable of $\ell$. Throughout this section, we assume that $w, w'$ are fresh variables not in $I$ or $X$.

Definition 1. We say that $\ell$ is $\land$-realizable in $G$ iff there is an assignment $\sigma : (\text{set}(I) \cup \text{set}(X)) \rightarrow \{\bot, \top\}$ such that $[G]_{\ell=w, \neg \ell=w'}^{\sigma} = [w \land w']$. Furthermore, we say that $\ell$ is $\land$-unrealizable in $G$ iff it is not $\land$-realizable in $G$.

Intuitively, $\ell$ is $\land$-realizable in $G$ if $\varphi_G$ reduces to $w \land w'$ under some assignment of variables other than $v$ and $w$, after $\ell$ and $\neg \ell$ are replaced by $w$ and $w'$ respectively in the leaves of $G$. It is easy to see that if $\ell$ is $\land$-realizable (resp. $\land$-unrealizable) in $G$, then so is $\neg \ell$.

Example 2. Consider circuit $G$ in Figure 7 and let $G_1$ and $G_2$ denote the sub-circuits rooted at the left and right child, respectively of the root node. Then $x_1$ is $\land$-realizable in $G_2$ and $G$ (use $\sigma(i) = \sigma(x_2) = \bot$) but is $\land$-unrealizable in $G_1$.

We now extend the notion of $\land$-(un)realizability to that of sets of literal-consistent leaves. If $S$ is the set of all $\ell$-leaves of $G$, the notion of $S$ being $\land$-realizable (resp. $\land$-unrealizable) in $G$ naturally coincides with that of literal $\ell$ being $\land$-realizable (resp. $\land$-unrealizable) in $G$. However, if $S$ does not contain all $\ell$-leaves of $G$, we must specify what to do with leaves labeled $\ell$ but not in $S$. The following definition does exactly that.

Definition 2. Let $S$ be a literal-consistent set of leaves of $G$, and let $\ell$ be the literal labeling each leaf in $S$. Let $S'$ be the set of all $\ell$-leaves of $G$. We say that $S$ is $\land$-realizable (resp. $\land$-unrealizable) in $G$ if $\ell$ is $\land$-realizable (resp. $\land$-unrealizable) in $G \mid S \cap S'$. Thus, all $\ell$-leaves that are not in $S$ must be labeled $\bot$ before we check whether $\ell$ is $\land$-realizable in the resulting circuit.

Example 3. Referring back to Figure 7, we wish to check the $\land$-(un)realizability of $S = \{L_3\}$ in $G$. The literal labeling $L_3$ is $x_1$ and the set of all $x_1$-leaves is $S' = \{L_3, L_{10}\}$. Hence $S' \setminus S = \{L_{10}\}$. To check the $\land$-(un)realizability of $S$, we relabel $L_{10}$ with $\bot$, and $L_3$ with a fresh variable $w$. Additionally, all leaves labeled $\neg x_1$, i.e. $L_1$ and $L_{14}$ are re-labeled with a fresh variable $w'$.

Let $G'$ denote the resulting circuit. We now ask if there is an assignment $\sigma : \{i, x_2\} \rightarrow \{\bot, \top\}$ such that $[\varphi_{G'}]_{\sigma} = [w \land w']$. From the circuit structure of $G'$, we can see that there is only one leaf, viz. $L_3$, labeled $w$. Hence, in order to have $[\varphi_{G'}]_{\sigma} = [w \land w']$, the assignment $\sigma$ must not mask the value of $L_3$ from “propagating” up to the root of $G'$. This implies that $L_2$ must be labeled $\top$, i.e. $\sigma(i) = \bot$, and $\sigma(x_2) = \neg \sigma(i) = \top$. With this $\sigma$, it is now easy to verify that $[\varphi_{G'}]_{\sigma} = [w] \neq [w \land w']$. Hence, there is no assignment of $x_2$ and $i$ that renders the formula represented by $G'$ semantically equivalent to $w \land w'$. It follows that $S = \{L_3\}$ is $\land$-unrealizable in the circuit shown in Figure 7. A similar exercise shows that $S = \{L_{10}\}$ is $\land$-realizable in the same circuit (use $\sigma(i) = \sigma(x_2) = \bot$).

Finally, we use the above definitions to introduce a new normal form for circuits that precisely characterizes efficient Boolean Skolem Function Synthesis. We show in subsequent sections that this normal form defines a class $C^*$ of circuits that satisfies properties P0, P1 and P2 described in Section III.
4) For each \( j \in \{2 \ldots k\} \), \( S_j \) is \( \land \)-unrealizable in \( G \mid \{S_1 \cup \ldots \cup S_j \} \). It can similarly be verified that \( \{L_1\} \) is \( \land \)-unrealizable in \( G \mid \{L_1\} \). Finally, the function represented by \( G \mid \{L_1\} \) is \( \land \)-unrealizable in \( G \mid \{L_1\} \). Since \( \{L_1\} \) is \( \land \)-unrealizable in \( G \mid \{L_1\} \), and hence is independent of \( X \). Therefore, the circuit \( G \) is not in SAUNF w.r.t. \( \{x_1, x_2\} \) and \( S' = \{\{L_1\}, \{L_2\}, \{L_3\}, \{L_7\}\} \). However, \( G \) is in SAUNF w.r.t. \( \{x_1, x_2\} \) and \( S = \{\{L_3\}, \{L_7\}, \{L_5\}, \{L_1\}\} \). Hence, \( G \) is in SAUNF w.r.t. \( \{x_1, x_2\} \) and \( S' = \{\{L_1\}, \{L_2\}, \{L_3\}, \{L_7\}\} \). Since we have seen earlier that \( \{L_10\} \) is \( \land \)-realizable in \( G \).

V. Relation with other normal forms

Several normal forms for Boolean circuits studied in the literature, viz. ROBDD [14], FDD [17], DNNF [19], \( \lambda \)DNNF [16], wDNNF [12], SynNNF [15], admit efficient Boolean Skolem function synthesis, and satisfy properties P0 and P1 in our problem statement (see Section III). However, none of these are known to satisfy property P2 in our problem statement, thereby failing to provide a characterization of efficient Boolean Skolem function synthesis. In contrast, as we show in this paper, the class of SAUNF circuits satisfies all the properties mentioned in our problem statement.

Among the various alternative normal forms, we discuss SynNNF [15] first. We say that a circuit normal form (or class of circuits) \( \mathcal{N}_1 \) is exponentially more succinct than another normal form \( \mathcal{N}_2 \) if (i) for every circuit \( G_2 \in \mathcal{N}_2 \), there exists a circuit \( G_1 \in \mathcal{N}_1 \) such that \( |G_1| \leq |G_2| \) and \( \varphi_{G_1} = \varphi_{G_2} \), and (ii) there is a circuit \( G_1 \in \mathcal{N}_1 \) such that every circuit \( G_2 \in \mathcal{N}_2 \) with \( \varphi_{G_1} = \varphi_{G_2} \) has \( |G_2| \) \( \Omega \) \( |G_1| \). The notion of super-polynomial succinctness is similarly defined. The authors of [15] showed a conditional succinctness result for SynNNF, namely SynNNF is super-polynomially more succinct than DNNF [19] and dDNNF [16], unless some longstanding complexity theoretic conjectures are falsified. We show the following stronger result for SAUNF.

**Lemma 1.** SAUNF is unconditionally exponentially more succinct than DNNF and dDNNF.

**Proof.** We use a result from [25] to prove the lemma. In Proposition 11 of [25], a family of Boolean functions \( \{JS_r \mid r \geq 2\} \) is defined. The formula \( JS_r \), defined on \( \mathcal{O}(r^2) \) variables, asserts that for every triple \( (v_j, v_k, v_l) \) of variables in a carefully constructed set \( A_r \) of triples, at least one of \( v_j, v_k \) or \( v_l \) must be false. It is shown in [25] that \( |A_r| \in \mathcal{O}(r^2) \). Therefore, a CNF formula representing \( JS_r \) has \( \mathcal{O}(r^2) \) clauses, with each clause having three negated variables as literals. Since no non-negated variables appear as literals in the formula, a circuit representation of the CNF formula cannot have any literal-consistent subset of leaves that is \( \land \)-realizable. This implies that \( JS_r \) can be represented in SAUNF in size \( \mathcal{O}(r^2) \). It is also shown in [25] that any DNNF (and hence also dDNNF) representation of \( JS_r \) requires size \( 2^{\Omega(r^2)} \). Therefore, SAUNF is unconditionally exponentially more succinct compared to DNNF and dDNNF.

Next, we show that SynNNF is, in fact, a special case of SAUNF. Towards this end, we recall the definition of SynNNF.
from [15], re-cast in our terminology.

**Definition 4.** A circuit $G$ with leaves labeled by $\text{lits}(I)$ and $\text{lits}(X)$ is in SynNNF w.r.t. $X$ iff the following hold:

- $x_i$ is $\land$-unrealizable in $G$.
- For $2 \leq i \leq |X|$, $x_i$ is $\land$-unrealizable in $G$.

The following lemma shows that SAUNF strictly subsumes SynNNF.

**Lemma 2.** Every circuit $G$ that is in SynNNF w.r.t. $X$ is also a SAUNF circuit w.r.t. $\text{set}(X)$ and a sequence $S$ of $2 \cdot |X|$ $\land$-unrealizable subsets of leaves. However, there exist SAUNF circuits that are not in SynNNF.

**Proof.** Suppose a circuit $G$ is in SynNNF w.r.t. $X$, and let $|X| = r$. We define a sequence of $2r$ literal-consistent subsets of leaves of $G$ as follows. For each $j \in \{1, \ldots, r\}$, we define $S_{2j-1}$ to be the set of $x_i$-leaves of $G$, and $S_{2j}$ to be the set of $\neg x_i$-leaves of $G$. It can now be seen from Definition 4 and Definition 3 that $G$ is in SAUNF w.r.t. $\text{set}(X)$ and the sequence $S = (S_1, \ldots, S_{2n})$ of subsets of literal-consistent leaves. This proves the first part of the lemma.

To show the second part, we must demonstrate a circuit that is in SAUNF but not in SynNNF for any permutation of the sequence of system outputs $X$. We claim that the circuit $G$ in Figure 1 already shown to be in SAUNF, suffices for this purpose. This is because Definition 4 entails that for $G$ to be in SynNNF, at least one literal over $X$ must be $\land$-unrealizable in $G$. However, none of $x_1, \neg x_1, x_2, \neg x_2$ are $\land$-unrealizable in the circuit $G$ in Figure 1. Specifically, $\llbracket \varphi_G \rrbracket_{x_1 = \top, \neg x_1 = \top} = \llbracket w \land w' \rrbracket$ when $\sigma(i) = \sigma(x_2) = \bot$, and $\llbracket \varphi_G \rrbracket_{x_1 = \top, \neg x_1 = \top} = \llbracket w \land w' \rrbracket$ when $\sigma(i) = \sigma'(x_1) = \top$. Therefore, the circuit in Figure 1 is not in SynNNF w.r.t. any permutation of $X$. □

It has been shown in [15] that every DNNF, dDNNF and wDNNF circuit is also in SynNNF, and SynNNF is super-polynomially more succinct than wDNNF unless $P = \text{NP}$. Furthermore, every ROBDD and FDD can be converted to a DNNF (hence SynNNF) circuit with at most linear blowup in size, although SynNNF can be exponentially more succinct than ROBDD or FDD [19], [15]. By virtue of Lemma 2 we now have the following result.

**Corollary 1.** All subsumption and (conditional) succinctness results for SynNNF circuits hold for SAUNF circuits as well.

Since every Boolean specification can be represented as a SynNNF circuit [15], it also follows from Lemma 2 that property P0 in our problem statement (see Section III) holds for the class of SAUNF circuits.

VI. EFFICIENT SYNTHESIS OF SKOLEM FUNCTIONS FROM SAUNF SPECIFICATIONS

We now show how a Skolem function vector can be efficiently computed if the relational specification is given as a SAUNF circuit. Informally, the process involves transforming a given SAUNF circuit $G$ with leaves labeled by $\text{lits}(I)$ and $\text{lits}(X)$ to a semantically different but related circuit $H$ with leaves labeled by $\text{lits}(I)$, $\text{lits}(X)$ and $\text{lits}(X')$, where $X'$ is a sequence of fresh system outputs, also called auxiliary outputs. The transformation is done in a way such that a Skolem function vector for $(X, X')$ in $H$ can be found efficiently, and a projection of this Skolem function vector on the first $|X|$ components directly yields a Skolem function vector for $X$ in $G$. To formalize this notion, we begin with a few definitions.

**Definition 5.** (Equisynthesizable Under Projection) Let $G$ be a circuit representing a relational specification over system inputs $I$ and systems outputs $X$. Let $H$ be another circuit representing a relational specification over $I$ and $(X, X')$, where $X'$ is a fresh sequence of system outputs or auxiliary outputs. We say that $G$ is equisynthesizable to $H$ under projection, denoted $G \rightsquigarrow H$, iff the following hold:

- $\forall I : \forall X (\varphi_G(X, I) \Rightarrow \exists X' \varphi_H(X, X', I))$
- $\forall I : \forall X' (\varphi_H(X, X', I) \Rightarrow \varphi_G(X, I))$

It follows from Definition 5 that $\rightsquigarrow$ defines a transitive relation on circuits representing relational specifications. The following lemma is an easy consequence of Definition 5.

**Lemma 3.** If $G \rightsquigarrow H$ holds and $(\Psi(I), \Psi'(I))$ is a Skolem function vector for $(X, X')$ in $\varphi_H(X, X', I)$, then $(\Psi(I), \Psi'(I))$ is a Skolem function vector for $X$ in $\varphi_G(X, I)$.

**Proof.** Since $\forall I (\varphi_G(I, I) \Rightarrow \exists X \varphi_H(X, I))$ holds trivially, we only show below that $\forall I (\exists X \varphi_G(X, I) \Rightarrow \varphi_G(\Psi(I), I))$.

From Definition 5 and from the definition of Skolem functions, we have $\forall I (\exists X \varphi_G(X, I) \Rightarrow \exists X' \varphi_H(X, X', I) \Rightarrow \varphi_H(\Psi(I), \Psi'(I), I) \Rightarrow \varphi_G(\Psi(I), I))$. □

A. Role of auxiliary outputs

We now investigate how auxiliary outputs can be introduced in a principled manner, so that they help in generating Skolem functions. We start with a circuit $G$ with leaves labeled by $\text{lits}(I)$ and $\text{lits}(X)$. Let $G_1$ and $G_2$ be two sub-circuits of $G$ such that $G_1$ is not a sub-circuit of $G_2$ and vice versa. For a fresh auxiliary variable $p \notin \text{set}(X) \cup \text{set}(I)$ for $j \in \{1, 2\}$, define a circuit transformation $\tau_{j,p}$ that replaces the sub-circuit $G_j$ in $G$ with the circuit representing $\varphi_{G_j \land p}$. The definition of the circuit transformation $\tau_{j,p}$ is similar.

**Lemma 4.** If $\varphi_{G_1} \land \varphi_{G_2}$ is unsatisfiable, then $G \rightsquigarrow \tau_{1,p}(\tau_{2,p}(G))$.

**Proof.** Let $H$ denote the circuit $\tau_{1,p}(\tau_{2,p}(G))$, and let $H_1$ and $H_2$ denote the newly introduced sub-circuits representing $p \land \varphi_{G_1}$ and $\neg p \land \varphi_{G_2}$ respectively in $H$. We show below that the conditions for $G \rightsquigarrow H$ (see Definition 5) are satisfied.

Let $\sigma : \text{set}(X) \cup \text{set}(I) \rightarrow \{\bot, \top\}$ be an assignment for which $\llbracket \varphi_G \rrbracket$ evaluates to $\top$. We have four cases to analyze depending on what $\llbracket \varphi_{G_1} \rrbracket$ and $\llbracket \varphi_{G_2} \rrbracket$ evaluate to under $\sigma$.

- $\llbracket \varphi_{G_1} \rrbracket = \bot = \llbracket \varphi_{G_2} \rrbracket$: Then for any assignment to $p$, $\llbracket \varphi_{H_1} \rrbracket$ and $\llbracket \varphi_{H_2} \rrbracket$ also evaluate to $\bot$, and hence $\llbracket \varphi_H \rrbracket$ evaluates to the same value, viz. $\top$, as $\llbracket \varphi_G \rrbracket$. 

• If \([\varphi_{G_1} \models \top, \varphi_{G_2} \models \bot]\), then with \(p\) assigned \(\top\), \([\varphi_{H_1}] \models \top\) and \([\varphi_{H_2}] \models \bot\), and hence \([\varphi_H]\) evaluates to the same value, viz. \(\top\), as \([\varphi_G]\).

• By a similar argument, if \([\varphi_{G_1} \models \bot, \varphi_{G_2} \models \top]\), assigning \(p\) to \(\bot\) causes \([\varphi_H]\) to evaluate to \(\bot\).

• The case of \([\varphi_{G_1} = \varphi_{G_2} \models \top]\) doesn’t arise since \(\varphi_{G_1} \land \varphi_{G_2}\) is unsatisfiable.

This shows that \(\forall v \forall X \varphi_G(X, I) \Rightarrow \exists p \varphi_{H}(X, p, I)\).

Consider any assignment \(\sigma': \text{set}(X) \cup \{p\} \cup \text{set}(I) \rightarrow \{\bot, \top\}\) that renders \([\varphi_H] = \top\). Let \(\sigma \models \text{set}(X) \cup \text{set}(I) \rightarrow \{\bot, \top\}\) be the projection of \(\sigma'\) on \(\text{set}(X) \cup \text{set}(I)\). Note that \(\sigma'\) necessarily assigns one of \(p\) or \(\neg p\) to \(\bot\), while still rendering \([\varphi_H] = \top\). Therefore, since all internal gates in \(H\) (i.e. \(\land\) and \(\lor\) gates) are monotone, \(\sigma\) must render \([\varphi_{H}] = \top\) as well. However, \([\varphi_{H}] = \top\) by definition. Hence, \(\sigma\) satisfies \([\varphi_G](X, I)\). This shows that \(\forall v \forall X \forall p \left( \varphi_{H}(X, p, I) \Rightarrow \varphi_{G}(X, I) \right)\).

The argument in the above proof can be given in general form to prove the following.

**Lemma 5.** Let \(G_1 = \{G_{1,1}, \ldots, G_{1,s}\}\) and \(G_2 = \{G_{2,1}, \ldots, G_{2,t}\}\) be two sets of sub-circuits of \(G\) such that (a) there are no distinct \(G_{i,j}\) and \(G_{j,i}\) where one is a sub-circuit of the other, and (b) every \(G_{i,j}\) and \(G_{j,i}\). Let \(\tau^G_{G_1}\) (resp. \(\tau^G_{G_2}\)) denote the circuit transformation that replaces every sub-circuit \(G_{k,i} \in G_k\) with a subcircuit representing \(\varphi_{G_{1,i}} \land p\) (resp. \(\varphi_{G_{2,i}} \land \neg p\)), where \(p\) is a fresh variable. Then \(G \equiv \tau^G_{G_1}(\tau^G_{G_2}(G))\).

A particularly easy application of Lemma 5 is obtained by choosing any literal \(\ell\) that labels leaves of \(G\), and by choosing \(G_1\) and \(G_2\) to be subsets of \(\ell\)-leaves and \(\neg \ell\)-leaves, respectively. Note that if \(L\) is a subset of \(\ell\)-leaves of \(G\), then \(\tau^G_L\) gives the same circuit as \(G\mid_{L \cup \ell}\).

For the following theorem, consider a relational specifica-

**Theorem 1.** Suppose \(S \subseteq S_{\ell}\) is \(\land\)-unrealizable in \(G\). For a fresh auxiliary variable \(p \notin \text{set}(X) \cup \text{set}(I), \text{let } E\) denote the circuit \(G \mid_{S_{\ell}\setminus(p \land \neg \ell)\cup \{p\}}\) and let \(H\) denote the circuit \(E \mid_{S_{\ell} \cup \{p, \neg p\} \cup \{\ell\}}\). Note that the literals labeling leaves of \(G\) are from \(\text{lits}(X) \cap \text{lits}(I)\), those labeling leaves of \(E\) are from \(\text{lits}(X) \setminus \text{lits}(I)\), \(\text{lits}(I)\) and \(\{p, \neg p\}\), while the literals labeling leaves of \(H\) are from \(\text{lits}(X \setminus \{v_{\ell}\}) \cup \{p, \neg p\}\). Then the following statements hold.

1) \(\exists v \exists \varphi_G \iff \exists p \varphi_H \iff \exists v \exists \varphi_{G|_I} \land \neg \ell\)

2) If \(\Psi_H(I)\) is a Skolem function vector for \((X \setminus \{v_{\ell}\}, p)\) in \(H\), then the projection of \(\Psi_H\) on \(X \setminus \{v_{\ell}\}\) augmented with the Skolem function \(\varphi_{E|_{\ell} \land \neg \ell} = \{\Psi_H(I), I\}\) for \(\ell\) gives a Skolem function vector \(\Psi_G(I)\) for \(X\) in \(G\).

**Proof.** By Lemma 5, we have \(G \equiv E\). It then follows from Definition 5 that \(\exists v \exists \varphi_G \iff \exists v \exists \varphi_E\). Furthermore, since \(S\) is \(\land\)-unrealizable in \(G\), it follows from the definition of \(E\) and from Definition 2 that \(\ell\) is \(\land\)-unrealizable in \(E\). From this, we will now show that \(\exists v \exists \varphi_E\) is equivalent to \(\varphi_{E|_{\ell} \land \neg \ell}\). In one direction, we observe that \(\exists v \exists \varphi_E\) is always equivalent to \(\varphi_{E|_{\ell} \land \neg \ell}\). This, in turn, logically implies \(\varphi_{E|_{\ell} \land \neg \ell}\) as all internal gates in an NNF circuit are monotone. In the other direction, \(\varphi_{E|_{\ell} \land \neg \ell}\) \(\land\) \(\neg \varphi_{E|_{\ell} \land \neg \ell}\) \(\lor\) \(\varphi_{E|_{\ell} \land \neg \ell}\) is unsatisfiable if \(\ell\) is \(\land\)-unrealizable in \(E\) (follows from definition of \(\land\)-unrealizability). Therefore, we have \(\exists v \exists \varphi_E \iff \varphi_{E|_{\ell} \land \neg \ell} \iff \varphi_H\). Hence, \(\exists v \exists \varphi_E \iff \varphi_H\), a Skolem function vector for \((X \setminus \{v_{\ell}\}, p)\) in \(E\) is obtained from \(\Psi_H(I)\). The Skolem function for \(\ell\) in \(E\) is then given by \(\varphi_{E|_{\ell} \land \neg \ell}\). To see why this works, note that \(\varphi_E\) with \(\Psi_H(I)\) substituted for \((X \setminus \{v_{\ell}\}, p)\) represents a specification with a single system output \(\ell\) and system inputs \(I\). Let us call this \(\phi(\ell, I)\). Then, \(\phi(T, I)\) serves as a Skolem function, say \(\psi_{\ell}(I)\), for \(\ell\) in \(I\), i.e., \(\exists \phi(\ell, I) \iff \phi_{\psi_{\ell}(I)}\). Indeed, suppose for some \(I, \psi_{\ell}(I) = \phi(T, I) = \top\). Conversely, if \(\psi_{\ell}(I) = \phi(T, I) = \top\), then we have \(\forall \phi(\ell, I) = \top\), i.e., all in all, we have \(\exists \phi(\ell, I) \iff \phi_{\psi_{\ell}(I)}\). The above method of obtaining a Skolem function for a single system output is also called self-substitution (12, 1, 2).

The second part of the theorem now follows from the observation that \(G \equiv E\).

**B. Generating Skolem functions from SAUNF circuits**

Theorem suggests an efficient algorithm for generating a Skolem function vector from a specification given as a SAUNF circuit. Algorithm presents the pseudo-code of algorithm SkGen. The purpose of sub-routines used in SkGen is explained in the comments.

We illustrate the running of SkGen by considering its execution on the circuit \(G\) shown in Fig. Here, \(X = (x_1, x_2)\) and \(I = (i)\). As discussed earlier, we use \(L_0\) through \(L_{15}\) to denote the leaves of the circuit \(G\) in left-to-right order. We have also seen earlier that \(G\) is in SAUNF for the sequence of subsets of leaves \(\{S_1, S_2, S_3, S_4\}\), where \(S_1 = \{L_3\}, S_2 = \{L_7\}, S_3 = \{L_5\}\) and \(S_4 = \{L_1\}\).

As algorithm SkGen proceeds, labels of different leaves of \(G\) need to be updated. For notational convenience, we use \(G^{(r)}, H^{(r)}\) and \(E^{(r)}\) to refer to the circuits \(G, H\) and \(E\) in the \(r\)th level of recursion of SkGen. Table shows how \(G^{(r)}, H^{(r)}\) and \(E^{(r)}\) are obtained by replacing the labels of suitable leaves of \(G\). Each entry in this table lists which leaf labels of \(G\) must be updated, where \(L_{(i,j,k)} : f\) denotes an update of the label of each leaf in \(\{L_1, L_2, L_3\}\) by \(f\). All leaves whose label updates are not specified are assumed to have the same labels as in \(G\). It can be verified that \(G\) with leaf labels updated as in the table entry corresponding to \(H^{(4)}\) simplifies to \(\top\) by constant propagation. Hence \(H^{(4)}\) is semantically independent of \(\{x_1, x_2\}\). This is not a coincidence, but is guaranteed by the
Algorithm 1: SkGen(G, S, r)

Input: G: Relational spec in SAUNF;
S = (S_1, S_2, ..., S_K): Sequence of ∧- unrealizable subsets of lits(X)-labeled leaves of C; r: Recursion level
Output: Ψ_G(I): Skolem function vector for C

1. if r = k + 1 then
   2. Ψ_G(I) := GetAnyFuncVec(|X|, I);
      // Returns an |X|-dim vector of (arbitrary) functions of I
3. else
   4. ℓ := Literal label of leaves in S_r;
   5. p_r := newOutputVar(); // p_r is auxiliary output variable added at recursion level r
   6. E := GetCkt(G, S_r, ℓ, p_r);
      // Replace all ℓ-labeled leaves of G other than those in S_r by ℓ ∨ p_r and replace all ¬ℓ-labeled leaves by ℓ ∨ ¬p_r
   7. S := GetNewSeq(S_r, ℓ, p_r);
      // Replace ℓ by p_r and ¬ℓ by ¬p_r in all elements (leaves) of S for j > r
   8. H := CPropSimp(E |= ℓ, ℓ = ⊤);
      // CPropSimp propagates constants and eliminates gates with constant outputs
   9. Ψ_H(I) := SkGen(H, S, r + 1);
10. ψ_E |ψ_H(I) := Ψ_H(I) \ {ψ_H(p_r)};
11. Ψ_G(I) := (ψ_E |ψ_H(I), Ψ_G(I));
12. return Ψ_G(I);

| r=1 | G(i) | E(i) | H(i) |
|-----|------|------|------|
| r=2 | Same as in H^{i-1} | L_{(1,14)} := p_1, L_{(3)} := \neg p_1, L_{(10)} := \neg p_1, L_{(14)} := \neg p_1, L_{(10)} := \neg p_1 |
| r=3 | Same as in H^{i-2} | L_{(1,14)} := p_1, L_{(3)} := \neg p_1, L_{(10)} := \neg p_1, L_{(14)} := \neg p_1, L_{(10)} := \neg p_1 |
| r=4 | Same as in H^{i-3} | L_{(1,14)} := p_1, L_{(3)}, L_{(10)} := \neg p_1, L_{(14)} := \neg p_1, L_{(10)} := \neg p_1 |

TABLE I: Run of Algorithm 1 on Fig. 1

As the recursive calls return, we obtain E^{(3)}(p_3 = f_3(i), p_4 = f_4(i), p_1 = \top, i) as Skolem function for p_1 in E^{(4)}. Call this function f_1(i). Next, we get f_2(i) = E^{(3)}(p_1 = f_1(i), p_3 = f_3(i), p_2 = \top, i) as Skolem function for p_2 in E^{(5)}. Continuing further, we obtain f_{x_2(i)} = E^{(3)}(p_1 = f_1(i), p_2 = f_2(i), x_2 = \top, i) as Skolem function for b in E^{(5)}, and f_{x_1(i)} = E^{(3)}(x_2 = f_{x_2(i)}, p_1 = f_1(i), x_1 = \top, i) as Skolem function for x_1 in E^{(4)}. The final return gives (f_{x_1(i)}, f_{x_2(i)}) as a Skolem function vector for (x_1, x_2) in G. Note that different choices of f_3(i), f_4(i) yield different Skolem function vectors of G, all of which are correct.

Theorem 2. Suppose Algorithm SkGen is invoked with a circuit G as input, that is in SAUNF w.r.t. set(X) and a sequence S of subsets of leaves. Assuming the vector returned in line 2 of the algorithm can be constructed in time O(|S|^2 · |G|), the algorithm returns a Skolem function vector of size in O(|S|^2 · |G|) in time O(|S|^2 · |G|).

Proof. We show that Algorithm SkGen generates a Skolem function vector by an inductive application of Theorem 1. Specifically, for each level i of recursion, if the Skolem function Ψ_H returned in line 9 by the i + 1** recursion call of SkGen is correct for H, Theorem 1 ensures that the Skolem function Ψ_G computed in lines 10 and 11 of the i**th recursive call is correct for G.

To see why the terminating case of this recursion yields correct Skolem functions, note that when the recursion level is k+1 (lines 1-2 of Algorithm 1), by Definition 3 the function represented by G is semantically independent of X. Hence any Skolem function vector for X suffices in line 2 of Algorithm 1.

Algorithm SkGen has exactly k + 1 recursive calls, and in each of the first k calls, the steps in lines 4, 5, 6, 7 and 8 take time linear in |G| and generate circuits E and H that are of size in O(|G|). Indeed, the circuit H in each recursion level ≤ k is simply G with the literal labels of some of its leaves replaced by other literals or by Boolean constants (possibly followed by simplification via constant propagation). Therefore, |H| ≤ |G| in each recursion level ≤ k. The circuit E is similarly obtained by replacing some leaves of G with Boolean constants, other literals or conjunctions of two literals. Therefore, |E| ≤ 2 × |G| in each recursion level ≤ k. In the k + 1** recursive call, line 2 is executed, and as discussed above, we restrict it to take time in O(|S|^2 · |G|). This also ensures that the size of the Skolem function vector returned in line 2 is in O(|S|^2 · |G|).

Once the recursive calls start returning, lines 10 and 11 of Algorithm SkGen are executed. Note that in line 10, the Skolem function for ℓ is obtained by feeding into the inputs of circuit E (as obtained in the current level of recursion) the outputs of Skolem functions computed in later (or higher) levels of the recursion. We have already seen above that |E| is at most 2 × |G| in each recursion level ≤ k. A Skolem function computed at recursion level j can potentially feed into E at all recursion levels in \{1, ..., j - 1\}. Therefore, a maximum of \sum_{j=2}^{\infty} (j - 1) ∈ O(|S|^2) connections may need to be created between the output of a Skolem function generated at some recursion level and the input of E at a lower level of recursion. Therefore, constructing the entire Skolem function vector at recursion level 1 requires time (and, hence space) in...
We now claim that the circuit, say $R$, representing $(\varphi_F \lor \varphi_G) \land \varphi_{G'}$ is in SAUNF w.r.t. set($X$) and the sequence $S$ of subsets of leaves of $F$ described above. To see why this is so, observe that all subsets $S_i \in S$ are mutually disjoint and contain leaves labeled by lits($X$). Hence the first 2 conditions of Definition 3 are satisfied. To see why conditions 3 and 4 of Definition 3 are satisfied, recall from Definition 2 that when checking $\land$-unrealizability of any $S_i \in S$, all leaves of the sub-circuit $G$ (of the circuit $R$) that are labeled by the same literal as leaves in $S_i$, must be re-labeled $\bot$. This, coupled with the fact that $S_i$ is $\land$-unrealizable in $F$, ensures that $S_i$ is $\land$-unrealizable in $R$ as well. Finally, as we will see in Section VII (see Lemma 2), $\varphi_{F[l_\downarrow \uparrow, \cdot \downarrow]} \land \varphi_{G'} \land \varphi_{G''}$. Since $\varphi_G'$ is semantically independent of $X$ by definition, condition 5 of Definition 3 is satisfied for $R$. Hence, $R$ is in SAUNF w.r.t. set($X$) and the sequence $S = (S_1, S_2, \ldots, S_{2m})$.

The proof of Theorem 5 gives the following corollary.

**Corollary 2.** Given $G$ and a Skolem function vector $\Psi$ for $X$ in $\varphi_G(X, I)$, a SAUNF circuit $H$ semantically equivalent to $G$ can be constructed in $O(|G| + |\Psi|)$ time. Furthermore, $|H| \in O(|G| + |\Psi|)$.

Finally, Corollary 2 and Theorem 2 immediately yield the following theorem.

**Theorem 4.** For every class $C$ of circuits representing relational specifications,

1) Boolean Skolem function synthesis can be solved in polynomial-time for $C$ if every circuit in $C$ can be compiled to a semantically equivalent SAUNF circuit in polynomial-time.

2) A Skolem function vector of polynomial size exists for every specification in $C$ if every circuit in $C$ can be compiled to a polynomial-sized semantically equivalent SAUNF circuit.

Thus, SAUNF satisfies properties P2a and P2b of Section III. In other words, SAUNF truly characterizes efficient Boolean functional synthesis. Note that Theorem 4 is significantly stronger than sufficient conditions for efficient synthesis given in [12], [13].

**VII. OPERATIONS ON SAUNF**

In this section, we discuss the application of basic operations like conjunction, disjunction and existential quantification of variables on formulas represented by SAUNF circuits, and also examine the complexity of checking if a given circuit is in SAUNF. Throughout this section, we assume that all specifications (circuits) are over system inputs set($I$) and system outputs set($X$) unless otherwise specified. To reduce notational clutter, given circuits $G$ and $H$, we abuse notation and use $G \lor H$ (resp. $G \land H$) to denote the circuit consisting
of an \( \lor \)- (resp. \( \land \))-labeled root node with the roots of \( G \) and \( H \) as its children.

**Lemma 6.** Suppose \( G \) is in SAUNF w.r.t set(\( X \)) and a sequence \( S^G \), and \( H \) is in SAUNF w.r.t. set(\( X \)) and \( S^H \). Then the circuit \( G \lor H \) is in SAUNF w.r.t. set(\( X \)) and \(( S^G, S^H )\).

**Proof.** The proof follows from the claim that \(( S^G, S^H )\) is a sequence of subsets of leaves of \( G \lor H \) that satisfies the conditions of Definition 3. To see why this is so, note that by Definition 2 when considering a subset, say \( S^G_i \), of \( \ell \)-labeled leaves in \( S^G \), all \( \ell \)-labeled leaves of \( H \) must be re-labeled \( \bot \). Hence, \( H \) can only contribute \( \neg \ell \), and can, at worst, combine with \( \ell \) contributed by \( G \) at the \( \lor \)-labeled root of \( G \lor H \). Since the set of \( \ell \)-labeled leaves in \( S^G_i \) are already \( \land \)-unrealizable in \( G \), we find that \( S^G_i \) is \( \land \)-unrealizable in \( G \lor H \) as well. By repeating this argument, we find that conditions 1, 2, 3, and 4 of Definition 3 are satisfied by the circuit \( G \lor H \). To see why condition 5 is also satisfied, observe that since \( G \) (resp. \( H \)) is in SAUNF w.r.t. set(\( X \)) and \( S^G \) (resp. \( S^H \)), when all subsets of leaves in \(( S^G, S^H )\) are re-labeled to \( \top \), the circuit \( G \lor H \) represents the disjunction of two formulas, each of which is semantically independent of \( H \). Hence, \( G \lor H \) is in SAUNF w.r.t set(\( X \)) and \(( S^G, S^H )\).

Significantly, Lemma 6 does not require any assumptions on the relation between ordering of subsets in \( S^G \) and \( S^H \). Other popular normal forms do not enjoy this property. For example, disjoining two ROBDDs constructed with different ordering of variables does not always yield an ROBDD in polynomial-time. Similarly, combining two SynNNF circuits with an \( \lor \) gate may not result in a SynNNF circuit unless the ordering of output variables in both circuits are the same. This shows that disjunction is more efficiently computable in SAUNF than in ROBDDs or even in SynNNF.

Next, we observe that existential quantification of all system outputs is easy for SAUNF representations.

**Lemma 7.** Suppose \( G \) is in SAUNF w.r.t set(\( X \)) and a sequence \( S \). Let \( L \) be the set of all leaves of \( G \) that are labeled by a literal over \( X \). Then \( \exists X \ G \leftrightarrow G \mid L \lor \top \).

**Proof.** Follows from Theorem 11 and Definition 5.

Next, we move to the more difficult case of conjunction.

**Lemma 8.** Suppose \( G \) is in SAUNF w.r.t set(\( X \)) and a sequence \( S^G \), and \( H \) is in SAUNF w.r.t. set(\( X \)) and sequence \( S^H \). If there is no literal \( \ell \) over \( X \) such that \( G \) has an \( \ell \)-labeled leaf and \( H \) has a \( \neg \ell \)-labeled leaf, the circuit \( G \land H \) is in SAUNF w.r.t. set(\( X \)) and \(( S^G, S^H )\). Otherwise, a SAUNF circuit semi-equivalent to \( G \land H \) cannot be constructed in time polynomial in \(|G|, |H|\) unless \( P = NP \). Further, such a circuit cannot have size polynomial in \(|G|, |H|\) unless \( \Pi^2_1 = \Sigma^2_1 \) (i.e. unless the polynomial hierarchy collapses to the second level).

**Proof.** If there is no literal \( \ell \) over \( X \) such that \( G \) has an \( \ell \)-labeled leaf and \( H \) has a \( \neg \ell \)-labeled leaf, it is easy to see that a leaf of \( G \) and a leaf of \( H \) cannot participate together to make any literal in \( \text{lits}(X) \) \( \land \)-realizable in the circuit \( G \land H \). Since \( G \) is in SAUNF w.r.t. set(\( X \)) and \( S^G \), and \( H \) is in SAUNF w.r.t. set(\( X \)) and \( S^H \), it then follows that \( G \land H \) is also in SAUNF w.r.t. set(\( X \)) and the sequence \(( S^G, S^H )\) (or alternatively, \(( S^H, S^G )\)).

The proof for the remainder of the lemma is more intricate. For this part of the discussion, we consider circuits \( G \) in which the labels of all leaves are considered to be from \( \text{lits}(X) \). In other words \( I \) is assumed to be empty. Let \( G^- \) denote the circuit \( G^- = x_1 \lor ... \lor x_m \), where \( X' = (x'_1, ..., x'_n) \) is a sequence of fresh variables. This is sometimes called the positive form of the circuit.

**Claim 1.** For every circuit \( G \), the circuit \( G^- \) is in SAUNF w.r.t. set(\( X \)) \cup set(\( X' \)) for any sequence of literal-consistent subsets of leaves.

**Proof.** There is no label of a leaf of \( G^- \) whose negation is also the label of some other leaf of \( G^- \) (since there are no negated output literals at all in the labels of leaves). This eliminates the possibility of a subset of literal-consistent leaves being \( \land \)-realizable in \( G^- \).

Further, given \( G \), let \( G^+ \) be the circuit representing the formula \( \wedge_{i=1}^n (x'_i \land \neg x_i) \lor (\neg x'_i \land x_i) \) where \( x'_i \) are the variables introduced in \( G^- \). Note that \( \varphi_{G^+} \leftrightarrow \wedge_{i=1}^n (x'_i \leftrightarrow \neg x_i) \).

**Claim 2.** For every circuit \( G \), the circuit \( G^- \) is in SAUNF w.r.t. set(\( X \)) \cup set(\( X' \)) and the sequence of literal-consistent leaves \( S = (S_1, ..., S_{2n}, S'_{2n}, ..., S'_{2n}) \), where \( S_{2n-1} \) (resp. \( S'_{2n-1} \)) is the set of all leaves labeled \( x_i \) (resp. \( x'_i \)), and \( S_2 \) (resp. \( S'_{2} \)) is the set of all leaves labeled \( \neg x_i \) (resp. \( \neg x'_i \)) in \( G^+ \).

**Proof.** It is easy to see that for the sequence \((S_1, S_2, ..., S_{2n})\), the first 4 conditions of Definition 3 are satisfied. Moreover, \( \varphi_{G^+} |_{S_1 \bot, S_2 \bot, ..., S_{2n} \bot} \equiv \top \). Therefore, condition 5 of Definition 3 is also satisfied, and \( G^- \) is in SAUNF w.r.t. set(\( X \)) \cup set(\( X' \)) and \((S_1, ..., S_{2n})\), and hence also w.r.t. the sequence \( S = (S_1, ..., S_{2n}, S'_1, ..., S'_{2n}) \).

It is easy to see that \( \varphi_{G^-} \land \varphi_{G^+} \) is equisatisfiable to \( \varphi_{G} \). Now, consider an arbitrary instance of the Boolean satisfiability (SAT) problem, i.e., given a Boolean circuit \( G \) over \( X = (x_1, ..., x_m) \), we must determine if \( \varphi_{G} \) is satisfiable. We interpret \( \varphi_{G} \) as a relational specification over system outputs \( X \), with the system inputs \( I \) being absent.

By definition, \(|G^-|\) and \(|G^+|\) are in \( \mathcal{O}(|G|) \). Using Claims 1 and 2, each of these circuits is also in \( \text{SAUNF w.r.t. set}(X) \cup set(X') \) and an appropriate sequence of subsets of leaves. Suppose there exists a polynomial-time algorithm \( A \) that takes two SAUNF circuits as inputs and produces a SAUNF circuit semantically equivalent to the conjunction of the formulas represented by the two circuits. We use algorithm \( A \) to obtain a circuit \( G' \) that is semantically equivalent to \( \varphi_{G^-} \land \varphi_{G^+} \). Clearly, \(|G'|\) must have size polynomial in \(|G^-|\) and \(|G^+|\), and therefore polynomial in \(|G|\). Since every SAUNF circuit yields Skolem functions for all outputs in time polynomial in the size of the circuit (see Theorem 2), we can compute a Skolem function for every output of \( G \) in time polynomial in \(|G|\).
Since there are no inputs, each of these Skolem functions must simplify to a Boolean constant. From the definition of Skolem functions, we also know that \( \hat{G} \) is satisfiable iff the Skolem functions obtained above (Boolean constants for variables in \( \text{set}(X) \cup \text{set}(X') \)) cause \( \hat{G} \) to evaluate to \( \top \). In other words, we can determine if \( \varphi_G \), and hence \( \varphi_G^- \land \varphi_G^+ \), is satisfiable in time polynomial in \( |G| \). Since \( \varphi_G^- \land \varphi_G^+ \) is equisatisfiable to \( \varphi_G \), this effectively solves the Boolean satisfiability problem in polynomial time. Therefore, algorithm \( \phi \) cannot run in polynomial time unless \( P = \text{NP} \).

Suppose for every two SAUNF circuits, there exists a polynomial sized SAUNF circuit that is semantically equivalent to the conjunction of the formulas represented by the two circuits. Let \( \hat{G} \) be the circuit obtained in this manner for \([\varphi_G^- \land \varphi_G^+]\). By Theorem 3, Skolem functions synthesized from \( G \) must have size polynomial in \( |G| \). By the same argument as above, it now follows that Boolean satisfiability must be in \( P/\text{Poly} \). This implies that \( \text{NP} \subseteq P/\text{Poly} \). By Karp-Lipton Theorem, however, we know that this implies that the polynomial hierarchy collapses to the second level, i.e. \( \Pi_2^P = \Sigma_2^P \).

Finally, we ask how difficult it is to check if a given circuit \( G \) is in \( \text{SAUNF} \).

**Theorem 5.** 1) Given \( G \) and a sequence \( S \) of literal-consistent disjoint subsets of leaves, checking if \( G \) is in \( \text{SAUNF} \) w.r.t. \( \text{set}(X) \) and \( S \) is \( \text{coNP} \)-complete.

2) Given \( G \), checking if \( G \) is in \( \text{SAUNF} \) w.r.t. \( \text{set}(X) \) and some (unspecified) sequence of subsets of leaves is \( \text{coNP} \)-hard and in \( \Sigma_2^P \).

**Proof.** First, we show that identifying whether a given circuit is in \( \text{SAUNF} \) for a given sequence of subsets of leaves is in \( \text{coNP} \). This is equivalent to asking whether the complement problem, i.e. if the given circuit is \( \not{\text{in}} \) \( \text{SAUNF} \) for the given sequence of subsets of leaves, is in \( \text{NP} \). We will define a non-deterministic polynomial-time Turing machine \( M \) that solves this complement problem. The machine \( M \) first checks whether the input circuit (say \( G \)) is in NNF. This can be done by checking if each internal node of \( G \) is labeled either \( \lor \) or \( \land \), and if all negations (if any) are on the labels of leaves. Clearly, this check can be done in time polynomial in the size of \( G \). If the circuit is found to be not in NNF, the machine \( M \) accepts, since \( G \) cannot be in \( \text{SAUNF} \) in this case. Otherwise (i.e. if \( G \) is in NNF), the machine \( M \) non-deterministically chooses a subset \( S_i \) of literal-consistent leaves in the given sequence \( S \) and executes the following operations.

Suppose the literal labeling leaves in the subset \( S_i \) is \( \ell_i \). The machine \( M \) does the following: (a) it constructs \( G^\dagger = G |_{S_1 \cap \ldots \cap S_{i-1} \cap \ell_i} \), (b) sets all leaves of \( G^\dagger \) that are not in \( S_i \) but are labeled \( \ell_i \) to \( \perp \), (c) replaces all remaining labels \( \ell \) (resp. \( \neg \ell \)) on leaves by \( w \) (resp. \( w' \)), (d) guesses an assignment \( \sigma \) to all variables other than \( w \) and \( w' \) labeling leaves in the resulting circuit, and (e) checks if the resulting circuit represents the Boolean function \( w \land w' \) for the assignment \( \sigma \). Note that after step (d), the resulting circuit represents a function of only \( w \) and \( w' \). Hence the check in step (e) can be performed by setting \( (w, w') \) to each of \((\top, \top), (\top, \perp), (\perp, \top), (\perp, \perp)\) and checking if the resulting circuit evaluates to \( \top, \perp, \perp \) and \( \perp \) respectively. Clearly, all the steps above can be done in time polynomial in \( |G| \).

Next, we show that the problem is \( \text{co-NP} \)-hard. We reduce the problem of checking if a propositional formula represented by a CNF circuit \( G \) is unsatisfiable to identifying whether an appropriately constructed circuit is in \( \text{SAUNF} \) for a specific sequence of subsets of literal-consistent leaves. For this, we consider the specification \( G \land \ell \land \neg x \), where \( x \) is the sole output of the specification, and the inputs are the variables labeling leaves of \( G \). Since there is only one output variable, there are only two (equivalent) orderings of subsets of leaves labeled by output literals. It is easy to see that \( x \) (equivalently, \( \neg x \)) is \( \land\)-realizable if and only if \( G \) is satisfiable. Hence, identifying whether a problem is in \( \text{SAUNF} \) for a given sequence of subsets of leaves is \( \text{coNP} \)-hard.

To prove the second part of the theorem, we show that checking whether a given \( G \) is in \( \text{SAUNF} \) w.r.t. some (unspecified) sequence of subsets of output literal-consistent leaves can be solved by a non-deterministic polynomial-time Turing machine \( M \) with access to an \( \text{NP} \) oracle, i.e. the problem is in \( \text{NP}^{\text{NP}} \). Given a specification \( G \) with system inputs \( I \) and system outputs \( O \) the machine \( M \) does the following:

(i) It guesses a sequence \( S \) of disjoint subsets of literal-consistent leaves.

(ii) It then reduces the problem of deciding whether \( G \) is not in \( \text{SAUNF} \) w.r.t. \( \text{set}(X) \) and the sequence \( S \) to checking the satisfiability of an appropriately constructed propositional formula \( \varphi \). This reduction is similar to what we discussed above in the proof of part (1).

(iii) Finally, it feeds \( \varphi \) to the \( \text{NP} \) oracle and accepts if and only if the \( \text{NP} \) oracle rejects.

Therefore, \( \text{M} \) accepts if and only if there is a sequence \( S \) of subsets of output literal-consistent leaves for which \( G \) is in \( \text{SAUNF} \) w.r.t. \( \text{set}(X) \) and \( S \). Hence, we have proved that our problem is contained in \( \text{NP}^{\text{NP}} \), or equivalently in \( \Sigma_2^P \). The proof that the problem is \( \text{coNP} \)-hard uses arguments similar to the earlier \( \text{coNP} \) hardness proof.

**VIII. CONVERSION TO SAUNF**

We now present an algorithm for compiling a circuit \( G \) representing a CNF formula over \( X \) and \( I \) to a semantically equivalent circuit in \( \text{SAUNF} \). Algorithm GetSaunf (see Algorithm 2) takes \( G \) as input and produces a circuit \( F \) and
Algorithm 2: GetSAUNF(G)

Input: G: Circuit representing a CNF formula
Output: F: SAUNF ckt semantically equivalent to G
S: Sequence of literal-consistent subsets of leaves of F
1 if \([G] \text{ is semantically independent of } X\) then
2 \(\ell := \text{ChooseLiteral}(G)\); // Gives a literal over \(X\) labeling a leaf in \(G\)
3 \(U := \text{GetSubset}(G, \ell)\); // \(U\) is an \(\land\)-unrealizable subset of \(\ell\)-labeled leaves in \(G\)
4 \(G' := \text{CktFromClausesWithLeaves}(U)\); // \(G'\) represents conjunction of clauses (in the CNF formula represented by \(G\)) containing leaves in \(U\)
5 \(D := G|_{U:T}\)
6 if \([D] \text{ is semantically independent of } \ell\) then
7 \((G_1, U_1) := \text{GetSAUNF}(D)\);
8 \(\ell \supseteq \text{ChooseLiteral}(G_1)\); // \(\ell\) is a literal over \(X\) labeling a leaf in \(G_1\)
9 \((G_2, U_2) := \text{GetSAUNF}(D|_{|\ell\land\top}\})\);
10 \(F := G' \land ((\ell \land G_1) \lor (\neg \ell \land G_2))\);
11 \(U_\ell := \{\ell\text{-leaf of }F\text{ in sub-circuit for } (\ell \land G_1)\}\);
12 \(U_{\ell\land\top} := \{\neg \ell\text{-leaf of }F\text{ in sub-circuit for } (\neg \ell \land G_2)\}\);
13 return \((F, (U_\ell, U_{\ell\land\top}, U_1, U_2))\);

The algorithm would be correct if it had returned \(U_{\ell\land\top}\) containing leaves of \(G'\) is \(\land\)-unrealizable as it was already \(\land\)-unrealizable in the circuit \(G\).

For \(U\) to be \(\land\)-realizable in \(F|_{U:T}\), there must be an assignment \(\sigma: \text{set}(X) \setminus \{v_1\} \cup \text{set}(I) \rightarrow \{\bot, \top\}\) such that \([\varphi_G\sigma]\) evaluates to \(\ell\) in \([\varphi_{G_1}\sigma]\) to \(\bot\) and \([\varphi_{G_2}\sigma]\) to \(\top\). However, this is a contradiction, since \(U\) is \(\land\)-unrealizable in \(G\). In \(F|_{U_{\ell\land\top}:U:T}\), all \(\ell\)-leaves of \(F\) have been re-labeled to \(T\), and therefore we can set the only \(\neg\ell\)-leaf (in set \(U_{\ell\land\top}\)) to \(T\). Finally, \(F|_{U_{\ell\land\top}:U:T} = G_1 \lor G_2\), which is in SAUNF assuming the recursive calls return correct representations and using Lemma 6. The base condition is when \(G\) is independent of \(X\), for which it already returns the correct value.

Note that the sequence of subsets returned in line 15 of Algorithm 3 \((U_1, U_{\ell\land\top}, U_1, U_2)\), which corresponds to the sequence of setting leaves to \(T\) as discussed above. The algorithm would be correct if it had returned \((U_1, U_{\ell\land\top}, U_1, U_2)\) as the sequence of subsets of leaves as well.

In the worst-case, Algorithm 3 \(\text{GetSAUNF}\) can reduce to brute-force Shannon expansion if \(U\) computed in line 4 of the algorithm always returns the empty set of leaves. In this case, both the running time and the size of \(F\) can grow exponentially with \(|G|\).

It remains to discuss the sub-routine \(\text{GetSubset}\). The pseudo-code for this sub-routine is shown in Algorithm 3. Let \(v_\ell\) denote the underlying variable of the literal \(\ell\). \(\text{CurS}\) is a subset of clauses containing \(\ell\) such that there exists an assignment \(\sigma: \text{set}(X) \setminus \{v_1\} \cup \text{set}(I) \rightarrow \{\bot, \top\}\) for which all (and only) these clauses of the underlying CNF formula evaluate to \(\ell\), and \(\ell\) is \(\land\)-realizable in \(G\) under \(\sigma\) (see Definition 1). Such an assignment \(\sigma\) can be obtained by effectively finding a satisfying assignment of \(\varphi_D|_{\ell=T, \neg \ell=T} \land \neg \varphi_D|_{\ell=T, \neg \ell=\bot} \land \neg \varphi_D|_{\ell=\bot, \neg \ell=T} \land \neg \varphi_D|_{\ell=\bot, \neg \ell=\bot}\). To ensure that \(\text{CurS}\) in the current iteration does not include any such set obtained in previous iterations of the loop, we conjoin \(D\) with all clauses in \(\text{CurS}\) after dropping \(\ell\). The sets \(\text{CurS}\) obtained in each iteration of the repeat-unti loop are collected in \(\text{AllS}\). Finally when \(\ell\) becomes \(\land\)-unrealizable in circuit \(D\), we obtain a satisfiable minimal hitting set (set cover) \(\text{HitS}\) of \(\text{AllS}\), i.e. a subset of clauses that is jointly satisfiable with \(\ell\) set to \(\bot\) and that includes a clause from every set in \(\text{AllS}\). Once \(\text{HitS}\) is obtained, we exclude all \(\ell\)-leaves that appear in the clauses in \(\text{HitS}\) to obtain a (maximal) \(\land\)-unrealizable subset of \(\ell\)-leaves, as required. Let us now formalize the correctness and complexity for this algorithm.

Lemma 9. Algorithm GetSAUNF returns a SAUNF circuit \(F\) semantically equivalent to the input circuit \(G\), with a worst-case running time exponential in \(|G|\), and the worst-case size of \(F\) also exponential in \(|G|\).

Proof. Since the circuit \(G'\) represents a conjunction of a subset of clauses, each of which contains the literal \(\ell\), the circuit \(D = G|_{U:T}\) obtained in line 6 of Algorithm 3 \(\text{GetSAUNF}\) simply represents the conjunction of all remaining clauses. In the circuit \(F\) constructed in line 11 of Algorithm 2, the set \(U\) containing only the leaf \(\ell\) is \(\land\)-unrealizable as it meets up \(\neg \ell\) at an \(\lor\) gate. In \(F|_{U_{\ell}\land\top}\), we can show that the set \(U\) of \(\ell\)-leaves of \(G'\) is \(\land\)-unrealizable as it was already \(\land\)-unrealizable in the circuit \(G\).

To understand how subroutine \(\text{GetSAUNF}\) works, let \(v_\ell\) denote the underlying variable of the literal \(\ell\). We use \(D\) to represent the current view of the circuit (formula) from which we wish to extract the \(\land\)-unrealizable subset of leaves. We also use \(\text{CurS}\) to denote a subset of clauses containing \(\ell\) such that there exists an assignment \(\sigma: \text{set}(X) \setminus \{v_1\} \cup \text{set}(I) \rightarrow \{\bot, \top\}\) for which all (and only) these clauses of the underlying CNF formula evaluate to \(\ell\), and \(\ell\) is \(\land\)-realizable in \(G\) under \(\sigma\) (see Definition 1). Such an assignment \(\sigma\) can be obtained by effectively finding a satisfying assignment of \(\varphi_D|_{\ell=T, \neg \ell=T} \land \neg \varphi_D|_{\ell=T, \neg \ell=\bot} \land \neg \varphi_D|_{\ell=\bot, \neg \ell=T} \land \neg \varphi_D|_{\ell=\bot, \neg \ell=\bot}\). To ensure that \(\text{CurS}\) in the current iteration does not include any such set obtained in previous iterations of the loop, we conjoin \(D\) with all clauses in \(\text{CurS}\) after dropping \(\ell\). The sets \(\text{CurS}\)
Algorithm 3: GetSubset($G, \ell$)

Input: $G$: circuit representing CNF formula, $\ell$: Literal

Output: $T$: $\land \neg$-unrealizable subset of $\ell$-leaves in $G$

1. $\text{AllS} := \text{CurrS} := \emptyset$;
2. $D := G$;
3. repeat
   4. $\sigma := \text{GetAssignment}(D, \ell)$; // Assignment of vars except $v_\ell$ for which $\ell$ is $\land \neg$-realizable in $D$
   5. $\text{CurrS} := \text{GetClausesEvaluatingToL}(G, \sigma, \ell)$; // Set of all clauses (of CNF formula represented by $G$) containing $\ell$ that do not become $\top$ under $\sigma$
   6. $\text{AllS} := \text{AllS} \cup \{\text{CurrS}\}$;
   7. $D := D \land \text{DisjoinWithoutLit}(\text{CurrS}, \ell)$; // DisjoinWithoutLit($\text{CurrS}, \ell$) gives disjunction of clauses in $\text{CurrS}$ after dropping $\ell$
4. until $\ell$ is $\land \neg$-unrealizable in $D$
5. $\text{HitS} := \text{SatisfiableHittingSet}(\text{AllS})$;
6. $T := \text{Set of all } \ell$-leaves of $G$ that don’t feed into any clause in $\text{HitS}$;
7. return $T$;

and therefore $\varphi_D|_{\ell \leftarrow \top} \land \neg \varphi_D|_{\ell \leftarrow \bot} \land \neg \varphi_D|_{\ell \leftarrow \bot} \land \neg \varphi_D|_{\ell \leftarrow \bot}$. To ensure that $\text{CurrS}$ in the current iteration does not include any such set obtained in previous iterations of the loop, we conjoin $D$ with the clause returned by DisjoinWithoutLit($\text{CurrS}, \ell$). All sets $\text{CurrS}$ obtained as the repeat-until loop iterates are collected in $\text{AllS}$. Finally when $\ell$ becomes $\land \neg$-unrealizable in circuit $D$, we obtain a satisfiable minimal hitting set (set cover) $\text{HitS}$ of $\text{AllS}$, i.e. a subset of clauses that is jointly satisfiable with $\ell$ set to $\bot$ and that includes a clause from every set in $\text{AllS}$. Given $\text{AllS}$, finding a minimal $\text{HitS}$ can be reduced to a MaxSAT problem. Once $\text{HitS}$ is as obtained, we exclude all $\ell$-leaves that appear in the clauses in $\text{HitS}$ to obtain a (maximal) $\land \neg$-unrealizable subset of $\ell$-leaves. With this, we can state the correctness and complexity of Algorithm 3.

Lemma 10. Algorithm GetSubset returns a $\land \neg$-unrealizable subset of $\ell$-leaves of $G$, and takes worst-case time exponential in $|G|$.

Proof. Suppose Algorithm GetSubset returned an $\land \neg$-realizable subset of $\ell$-leaves of $G$. Let $\sigma$ be the corresponding assignment of set($X\setminus\{v_\ell\}$) intersect set($I$). The set $S$ of clauses of $\varphi_G$ that do not become $\top$ under $\sigma$ must then be either equal to or a superset of $\text{CurrS}$ in some iteration of the repeat-until loop of lines 3-7. Therefore, $\text{HitS}$ must include some clause from $S$. This implies that $T$ contains at least one $\ell$-labeled leaf that feeds into a clause in $\text{HitS}$ - a contradiction.

The worst-case running time is dominated by the product of the number of times the repeat-until loop of lines 3-8 iterates and the time required to obtain $\sigma$ (line 4) and check the loop termination condition (line 8). The count of loop iterations can be as high as the count of all subsets of $\ell$-labeled leaves. This is exponential in $|G|$ in the worst-case. Computing $\sigma$ and checking the loop termination condition also require time exponential in $|G|$ in the worst-case. Hence, the worst-case running time of Algorithm GetSubset is exponential in $|G|$.

Note that we can modify the loop termination condition in Algorithm GetSubset by incorporating a timeout. In case a timeout happens, we conservatively return $\emptyset$ as $T$. This reduces the worst-case running time, providing a tradeoff between running time and precision of computation.

IX. SOME INTERESTING APPLICATIONS

In Section I, we described the $n$-bit factorization problem – a problem of immense interest in cryptanalysis. We now show some interesting partial results using SAUNF circuits. We start with a relational specification $R$ over inputs $I$ and outputs $(X, Y)$ defined by $(X \times [n] Y = I)$ where $X$ denotes $n$-bit unsigned integer multiplication. This specification evaluates to true iff the given product relation holds, where $X, Y$ are $n$-bit output vectors and $I$ is a $2n$ bit input vector. For $1 \leq l \leq j \leq 2n$, we define a parametrized specification $R[l, j]$ over $X, Y$ and $I$ that evaluates to true if and only if the bits from position $l$ to $j$ of $X \times [n] Y$ match the corresponding bits of $I$.

Let 1 denote an $n$-bit representation of the integer 1. If $R[1, 2n] \land (X \neq 1) \land (Y \neq 1)$ can be represented as a polynomial (in $n$) sized SAUNF circuit, our results show that Skolem functions of size polynomial in $n$ can be obtained for $n$-bit factorization with non-trivial (i.e. $\neq 1$) factors. This would have serious ramifications for cryptanalysis. While we are not close to achieving such a result, our initial studies show some interesting results in trying to represent $R[l, j]$ in SAUNF. Note that it is already known from [17] that representing $R[n, n]$ requires exponentially large ROBDDs, and sub-exponential representations using DNNF, dDNNF, wDNNF or SynNNF are not known. With SAUNF circuits however, we obtain a significant improvement.

Theorem 6. For $l \leq j \leq 2n, j - l < n, R[l, j]$ is representable by a polynomial (in $n$) sized SAUNF circuit.

Proof. By Theorem 3 a polynomial-sized SAUNF circuit can be generated from a polynomial-sized Skolem function vector. Therefore, we focus on obtaining a polynomial-sized Skolem function vector for $R[l, j], l \leq j \leq 2n, j - l < n$. We use the notation $X_i$ to denote the $l^{th}$ least significant bit of $X$ and $X_{l,j}$ to denote the bit-slice of $X$ from $l$ to $j$ (both included).

Using similar notations for $Y$ and $I$, we consider two cases:

- $l \leq n$: A Skolem function vector for $(X, Y)$ is given by $\psi^x = 1$ (resp. $\bot$), if $k \neq l$ (resp. $k = l$), and $\psi^y = I_{l+k-1}$ (resp. $\bot$) if $k \leq j + 1 - l$ (resp. $k > j + 1 - l$). Intuitively, $X$ represents $2^{l-1}$ and $Y = I_{l,j}$.
- $l > n$: Following similar logic, the Skolem function vector is given by $\psi^x = 1$ (resp. $\bot$) if $k \neq n$ (resp. $k = n$),
and \( \psi^{[k]} = I_{k+n-1} \) (resp. \( \perp \)) if \( l - n < k \leq j + 1 - n \) (resp. otherwise).

Intuitively, \( X \) represents \( 2^{n-1} \) and \( Y \) represents \( 2^{l-n} \times [n] \).

Having generated a polynomial-sized Skolem function vector for \( R[l,j], j \leq l < n \), we can generate a corresponding polynomial-sized SAUNF circuit using Theorem 4.

Surprisingly, we can use Theorem 6 to also show that a restricted version of division has a polynomial sized SAUNF representation. Consider the same relational specification \( X \times [n] \ Y = I \) considered earlier. For the division problem, we treat \( (I,Y) \) as system inputs and \( X \) as system outputs, and write the relation as \( X = I \div Y \) and obtain the following theorem.

**Theorem 7.** The relation \( X = I \div Y \), with inputs \( I, Y \) restricted to odd numbers (i.e. the relation evaluates to \( \perp \) if \( I \) or \( Y \) is even), is representable as a polynomial (in \( n \)) sized SAUNF circuit.

**Proof.** For notational convenience, we use \( X, I, Y \) to denote both sequences of Boolean variables, and also the unsigned integers represented by the corresponding bit-vectors. We use \( \times \) instead of \( \times [n] \) to denote \( n \)-bit unsigned integer multiplication, and \( + \) (resp. \( - \)) to denote \( n \)-bit unsigned integer addition (resp. subtraction).

We give below a polynomial-sized Skolem function vector for division, which can be used to obtain a SAUNF form by Theorem 4.

Suppose we have inputs \( I \) and \( Y \) and we have to find \( X \) such that \( X \times Y = I \). We first show that for odd valued inputs, if \( (X \times Y) \mod 2^n = I \mod 2^n \) and if \( I \) is divisible by \( Y \) then \( X \times Y = I \). Suppose there are two values \( X^1, X^2 \) such that \( (X^1 \times Y) \mod 2^n = I \mod 2^n \) and \( (X^2 \times Y) \mod 2^n = I \mod 2^n \). Then \( (X^1 \times X^2) \times Y = 0 \mod 2^n \). However, \( Y \) is odd and therefore co-prime to \( 2^n \); hence \( X^1 \times X^2 \equiv 0 \mod 2^n \). Since, \( X^1 \times X^2 < 2^n \), we must have \( X^1 = X^2 \). Therefore, the generated \( X \) from the Skolem function vector is correct if there exists a solution that matches the least significant \( n \) bits of \( I \). Using the notation defined above, denote \( X_i \) to be the \( i \)-th least significant bit of \( X \) and \( X_{i,j} \) to denote the bit vector from \( X_i \) to \( X_j \) (both included). Now, since the inputs \( I \) and \( Y \) are odd, \( Y_1 = \top, I_1 = \top \). Therefore \( \psi^{X_{1\cdots1}} = \top \).

Now note that \( (X \times Y)_i = (X_{1,i} \times Y_{1,i}) = (X_{1,i-1} \times Y_{1,i} + X_{1,i} \times Y_0 \times (2^{-1})^i) = X_{i} \times Y_1 \oplus (X_{1,i-1} \times Y_{1,i}) \) (using the structure of multiplication), where \( \oplus \) denotes \( 1 \)-bit multiplication and \( \oplus \) denotes \( 1 \)-bit addition modulo 2. Therefore, \( I_i = X_i \oplus (X_{1,i-1} \times Y_{1,i}) \), or equivalently, \( X_i = I_i \oplus (X_{1,i-1} \times Y_{1,i}) \).

It is now easy to see that once we obtain a Skolem function for \( \psi^{X_{1\cdots1}} \) in this manner, we can recursively generate the skolem function for \( X_i \), giving the entire Skolem function vector for \( X \).

Note that while we have used a specific Skolem function vector above, once the SAUNF form is obtained, it can be used to generate other Skolem function vectors as well (from Algorithm 1).

**X. Conclusion**

In this paper, we presented a normal form for Boolean relational specifications that characterizes efficient Skolem function synthesis. This is a significantly stronger characterization than those used in earlier works. SAUNF is exponentially more succinct than DNNF, dDNNF while enjoying similar composability properties. It also strictly subsumes the recently proposed SynNNF. As future work, we plan to improve the compilation algorithm and apply it to challenging benchmarks. It would also be interesting to see if similar characterizations or normal forms exist and are efficiently computable for Skolem functions for first order logic \( [26] \), i.e., beyond the propositional case that we treated in this work.

**References**

[1] J. R. Jiang, “Quantifier elimination via functional composition,” in CAV, ser. LNCS, vol. 5643. Springer, 2009, pp. 383–397.

[2] A. Shukla, A. Bierre, M. Sidi, and L. Pulina, “A survey on applications of quantified boolean formula,” in Proc. of 31st International Conference on Tools with Artificial Intelligence (ICTAI), 2019, pp. 78–84.

[3] M. N. Rabe and L. Trentup, “CAQE: A certifying QBF solver,” in Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015, pp. 136–143.

[4] M. S. Manjun Heule and A. Biere, “Efficient Extraction of Skolem Functions from QBF Proofs,” in Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014, pp. 107–114.

[5] M. N. Rabe and S. A. Seshia, “Incremental determinization,” in SAT, ser. LNCS, vol. 9710. Springer, 2016, pp. 375–392.

[6] J.-H. R. Jiang and V. Balabanov, “Resolution proofs and Skolem functions in QBF evaluation and applications,” in Proc. of CAV. Springer, 2011, pp. 149–164.

[7] D. Fried, L. M. Tabajara, and M. Y. Vardi, “BDD-based boolean functional synthesis,” in Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, 2016, pp. 402–421.

[8] S. Akshay, S. Chakraborty, A. K. John, and S. Shah, “Towards Parallel Boolean Functional Synthesis,” in TACAS 2017 Proceedings, Part I, 2017, pp. 337–353.

[9] S. Akshay, D. Fried, L. M. Tabajara, and M. Y. Vardi, “Functional synthesis via input-output separation,” in Formal Methods in Computer Aided Design, FMCAD, 2018, pp. 1–9.

[10] A. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay, “Skolem functions via input-output separation,” in Formal Methods in Computer Aided Design, FMCAD, 2015, pp. 1–9.

[11] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah, “What’s Hard About Boolean Functional Synthesis?” in Proc. of 30th International Conference on Computer Aided Verification, Part I, 2018, pp. 251–269.

[12] P. Golia, S. Roy, and K. S. Meel, “Manthan: A data-driven approach for boolean function synthesis,” in Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II, ser. Lecture Notes in Computer Science, vol. 12225. Springer, 2020, pp. 161–169.

[13] A. Darwiche, “On the tractable counting of theory models and its application to truth maintenance and belief revision,” Journal of Applied Non-Classical Logics, vol. 11, no. 1-2, pp. 11–34, 2001.

[14] R. E. Bryant, “On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication,” IEEE Trans. Computers, vol. 40, no. 2, pp. 205–213, 1991.
[18] M. Cadoli and F. M. Donini, “A survey on knowledge compilation,” *AI Commun.*, vol. 10, no. 3-4, pp. 137–150, 1997.
[19] A. Darwiche, “Decomposable negation normal form,” *J. ACM*, vol. 48, no. 4, pp. 608–647, 2001.
[20] C. Muise, S. A. McIlraith, J. C. Beck, and E. Hsu, “DSHARP: Fast d-DNNF Compilation with sharpSAT,” in *AAAI-16 Workshop on Beyond NP*, 2016.
[21] A. Darwiche and P. Marquis, “A knowledge compilation map,” *J. Artif. Intell. Res.*, vol. 17, pp. 229–264, 2002.
[22] A. Kuehlmann, V. Parthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.
[23] M. N. Rabe, L. Tenstrup, C. Rasmussen, and S. A. Seshia, “Understanding and extending incremental determinization for 2QBF,” in *CAV (2)*, ser. LNCS, vol. 10982. Springer, 2018, pp. 256–274.
[24] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah, “Boolean functional synthesis: hardness and practical algorithms,” *Formal Methods in System Design*, pp. 1–34, 2020.
[25] S. Bova, F. Capelli, S. Mengel, and F. Slivovsky, “Knowledge compilation meets communication complexity,” in *Proc. of 25th International Joint Conference on Artificial Intelligence, IJCAI*, 2016, pp. 1008–1014.
[26] S. Akshay and S. Chakraborty, “On synthesizing skolem functions for first order logic formulae,” *CoRR* arXiv:2102.07463 [cs.LO], 2021. [Online]. Available: https://arxiv.org/abs/2102.07463