Chapter to appear in Handbook of Logical Thought in India, 2022

Synthesizing Skolem functions: A view from theory and practice

S. Akshay · Supratik Chakraborty

(Pre-print)

Abstract Skolem functions play a central role in logic, from helping eliminate quantifiers in first order logic formulas to providing functional implementations of relational specifications. While their existence follows from classical results in logic, less is known about how to compute them effectively and efficiently (whenever such computation is possible). The problem of computing or synthesizing Skolem functions from relational specifications, however, continues to show up in many interesting applications. Recently, a rich line of work has considered theoretical and practical aspects of the problem in a restricted setting, namely synthesis of Boolean Skolem functions from Boolean relational specifications. In this article we take an indepth look into this fascinating problem and its various implications, from general theoretical and complexity results to practical algorithms, and also draw interesting connections to the knowledge representation literature.

Keywords Boolean functional synthesis, Skolem functions, expansion-based algorithms

1 Introduction

The genesis of Skolem functions dates back to 1920, when the Norwegian mathematician, Thoralf Albert Skolem, gave a simplified proof of a landmark result in logic, now known as the *Löwenheim-Skolem* theorem. Leopold Löwenheim had already proved this theorem in 1915. However, Skolem's 1920 proof was significantly simpler and made use of a key observation that can be summarized as follows[1]. *For every first order logic formula* $\exists y\, \varphi(x, y)$*, the choice of* y *that makes* $\varphi(x, y)$ *true (if at all) depends on* x *in general. This dependence can be thought of as implicitly defining a function that gives the "right" value of* y *for every value of* x*. If* F *denotes a fresh function symbol, the second order sentence* $\exists F\, \varphi(x, F(x))$ *formalizes this dependence explicitly. Thus, the second order sentence* $\exists F\, \forall x\, \big(\exists y\, \varphi(x, y) \Rightarrow \varphi(x, F(x))\big)$ *always holds.* Since the implication trivially holds in the other direction too, we have $\exists F\, \forall x\, \big(\exists y\, \varphi(x, y) \Leftrightarrow \varphi(x, F(x))\big)$.

Indian Institute of Technology Bombay, India

[1] We assume the reader is familiar with basic notation and terminology of first order logic.

Let ξ_1 and ξ_2 denote the first order formulas $\exists y\, \varphi(x, y)$ and $\varphi(x, F(x))$ respectively referred to above. The following points are worth noting.

- While ξ_2 has one less existential quantifier than ξ_1, the signature of ξ_2 has one more function symbol than the signature of ξ_1. Thus, an existential quantifier has been traded off, so to say, for a function symbol.
- Although ξ_1 and ξ_2 are not semantically equivalent, there is an interpretation of F such that for every assignment of the free variable x, the formula ξ_1 is satisfiable iff ξ_2 is.
- Every model $\mathfrak{M}$ of $\forall x\, \xi_1$ can be augmented with an interpretation of F to yield a model $\mathfrak{M}'$ of $\forall x\, \xi_2$. Similarly, for every model $\mathfrak{M}'$ of $\forall x\, \xi_2$, restricting $\mathfrak{M}'$ to the signature of ξ_1 yields a model $\mathfrak{M}$ of $\forall x\, \xi_1$.

The process of transforming ξ_1 to ξ_2 by eliminating $\exists y$ and substituting $F(x)$ for y is an instance of Skolemization. The fresh function symbol F introduced in the process is called a Skolem function. Skolem functions play a very important role in logic – both in theoretical investigations and in practical applications. The model theory of Skolemization in first order logic is rich: for instance, the Skolem expansion of a complete theory need no longer be complete, thus inviting further characterizations of Skolem hulls and indiscernibles. The extension of Skolemization to higher order logic is problematic and challenging (but needed, for instance, in automatic theorem proving).

While it suffices in some studies to simply know that a Skolem function F exists, in other cases (see Section 3 for such examples), we require an algorithm that effectively computes $F(x)$ for every x. It turns out that obtaining such an algorithm is impossible in general, and even for the subcases where it is possible, the computational complexity is often very high. The purpose of this article is to discuss these computational challenges, and to survey some techniques for computing Skolem functions that have been proposed in recent years in the context of a significantly restricted yet practically useful logic, viz. quantified propositional logic.

Before delving further, it is important to formally define some notation and terminology. We use lower case English letters, viz. x, y, z, possibly with subscripts, to denote first order variables, and bold-faced upper case English letters, viz. $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$, to denote sequences of first order variables. We use lower case Greek letters, viz. φ, ξ, α, possibly with subscripts, to denote formulas. For a sequence $\mathbf{X}$, we use $|\mathbf{X}|$ to denote the count of variables in $\mathbf{X}$, and $x_1, \ldots x_{|X|}$ to denote the individual variables in the sequence. With abuse of notation, we also use $|\varphi|$ to denote the size of the formula φ, represented using a suitable format (viz. as a string, syntax tree, directed acyclic graph etc.), when there is no confusion. Let Q denote a quantifier in $\{\exists, \forall\}$. For notational convenience, we use $Q\mathbf{X}$ to denote the block of quantifiers $Qx_1 \ldots Qx_{|\mathbf{X}|}$. It is a standard exercise in logic to show that every well-formed first order logic formula can be transformed to a semantically equivalent *prenex normal form*, in which all quantifiers appear to the left of the quantifier-free part of the formula. Without loss of generality, let $\xi(\mathbf{X}) \equiv \exists \mathbf{Y}\; \forall \mathbf{Z}\; \exists \mathbf{U} \ldots \forall \mathbf{V}\; \exists \mathbf{W}\, \varphi(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{U}, \ldots \mathbf{V}, \mathbf{W})$ be such a formula in prenex normal form, where $\mathbf{X}$ is a sequence of free variables and φ is a quantifier-free formula. In case the leading (resp. trailing) quantifier in ξ is universal, we consider $\mathbf{Y}$ (resp. $\mathbf{W}$) to be the empty sequence. Given such a formula ξ, *Skolemization* refers to the process of transforming ξ to a new (albeit related) formula $\xi^\star$ without any existential quantifiers via the following steps: (i) for every existentially quantified variable, say a, in ξ, substitute $F_a(\mathbf{X}, \mathbf{S}_a)$ for a in the quantifier-free formula φ, where F_a is a new function symbol and $\mathbf{S}_a$ is a sequence of universally quantified variables that appear to the left of a in the quantifier prefix of ξ, and (ii) remove all existential quantifiers from ξ. The functions F_a introduced above are called *Skolem functions*. In case ξ has no free variables, i.e. $\mathbf{X}$ is empty, the Skolem functions for variables y_i in the leftmost existential quantifier block of ξ have no arguments (i.e. are nullary functions), and are also called *Skolem constants*. The sentence $\xi^\star$ is said to be in *Skolem normal form* if the quantifier-free part of $\xi^\star$ is in conjunctive normal form. For notational convenience, let $\exists\mathfrak{F}$ denote the second order quantifier block $\exists F_{y_1} \ldots \exists F_{y_{|Y|}} \cdots \exists F_{w_1} \ldots \exists F_{w_{|W|}}$ that existentially quantifies over all Skolem functions introduced above. The key guarantee of Skolemization is that the second order sentence $\exists\mathfrak{F}\,\forall\mathbf{X}\big(\xi \Leftrightarrow \xi^\star\big)$ always holds. Note that substituting Skolem functions for existentially quantified variables need not always make the quantifier-free part of ξ, i.e. φ, evaluate to true. This can happen, for example, if there are valuations of universally quantified variables for which no assignment of existentially qualified variables renders φ true. For every other valuation of universally quantified variables, the Skolem functions indeed provide the "right" values of existentially quantified variables so that φ evaluates to true.

Example 1 Consider $\xi \equiv \exists y \forall x \exists z \forall u \exists v\, \varphi(x, y, z, u, v)$. On Skolemizing, we get $\xi^\star \equiv \forall x \forall u\, \varphi(x, C_y, F_z(x), u, F_v(x, u))$, where C_y is a Skolem constant for y, and $F_z(x)$ and $F_v(x, u)$ are Skolem functions for z and v respectively.

As mentioned earlier, the focus of this article is on effective computation of Skolem functions. It is well known (see e.g. [31]) that there exist functions that cannot be computed by any halting Turing machine, or equivalently, by any algorithm. Therefore, it is interesting to ask: *Can every Skolem function be computed?* In other words, given a first order formula ξ, does there always exist a halting Turing machine that computes each Skolem function appearing in a Skolemized version of ξ? In general, such a Turing machine (or algorithm) may need to evaluate predicate and function symbols that appear in the signature of ξ as part of its computation. Therefore, the most appropriate notion of computation in our context is that of *relative computation* or *computation by oracle machines*[2]. Formally, let $\mathcal{P}_\xi$ and $\mathcal{F}_\xi$ denote the set of predicate and function symbols respectively in the signature of ξ. Given oracles for interpretations of predicate symbols in $\mathcal{P}_\xi$ and of function symbols in $\mathcal{F}_\xi$, we ask if every Skolem function F in a Skolemized version of ξ can be computed by a halting Turing machine, say M_ξ^F, with access to these oracles. Note that we require M_ξ^F to depend only on ξ and F. However, the oracles that M_ξ^F accesses can depend on specific interpretations of predicate and function symbols.

Unfortunately, it has been shown in [1] that M_ξ^F does not always exist for every ξ and F. In other words, Skolem functions cannot be effectively computed in general, even in the relative sense mentioned above [1]. In fact, it doesn't take much to hit the uncomputability frontier. As shown in [1], uncomputability arises even if we allow a single unary uninterpreted predicate in the signature. What happens if all predicates and functions are interpreted, viz. in the theory of natural numbers with multiplication and addition? It turns out that Skolem functions

---

[2] See [7] for a detailed exposition on relative computability.

cannot be computed in general in this case too [1]. The proof in this case [1] appeals to the Matiyasevich-Robinson-Davis-Putnam (MRDP) theorem [23] that equates Diophantine sets with recursively enumerable sets.

Not all hope is lost however. As shown in [1] again, Skolem functions can indeed be computed for formulas in several interesting first order theories. For example, every first order theory that is (i) decidable, (ii) has a recursively enumerable domain, and (iii) has computable interpretations of predicates and functions, admits effective computation of Skolem functions. Such theories include Presburger arithmetic, linear rational arithmetic, countable dense linear order without endpoints, theory of evaluated trees, first order theories with bounded domain etc. Whenever Skolem functions are computable, one can further ask: *Can Skolem functions be represented as terms in the underlying logical theory?* It is easy to see that a positive answer to this question implies an effective procedure for quantifier elimination. We also know that some theories, viz. Presburger logic without divisibility predicates, do not admit quantifier elimination. Therefore, there exist first order theories for which Skolem functions can be effectively computed, but are not expressible as terms in the underlying logical theory. The study of algorithmic computation of Skolem functions is therefore highly nuanced.

Given the above discussion, perhaps the simplest theories for which we can compute Skolem functions are those with bounded domains. Consider a formula ξ in such a theory where the domain $\mathcal{D}$ has κ ($\in \mathbb{N}$) elements. Since the elements of $\mathcal{D}$ can be encoded as $\lceil \log_2 \kappa \rceil$-tuples of 0's and 1's, reasoning about ξ can be reduced to reasoning about a quantified propositional formula $\widehat{\xi}$, where $|\widehat{\xi}| \leq \lceil \log_2 \kappa \rceil \cdot |\xi|$. While this reduction does not affect the computational complexity results (in terms of complexity classes) that we study later, it can have an impact on the practical performance of algorithms, especially if $\log_2 \kappa$ is large.

One may argue that over bounded domains, we can replace quantifiers by conjunctions or disjunctions and thus work only with propositional logic. This leads to an exponential blow-up in the size of the formula, which is undesirable. Hence, we are motivated to consider quantified propositional formulas directly. This is analogous to satisfiability for quantified Boolean formulas, which is a well-studied problem with dedicated techniques and implementations, even though it can be reduced to satisfiability for propositional formulas (with an exponential blow-up).

Despite the expressive limitations of *quantified propositional logic*, there are many important applications where quantified propositional formulas play an important role [57]. Furthermore, not only can we effectively compute Skolem functions for formulas in this logic, we can also represent them as Boolean functions. We therefore focus on the algorithmic computation of Skolem functions for quantified propositional logic in the remainder of the article.

2 Boolean Skolem functions, synthesis and unification

We use *Quantified Propositional Logic* (henceforth, QPL) to refer to propositional logic augmented with existential and universal quantifiers. Without loss of generality, we assume that formulas in quantified propositional logic (QPL) are given in prenex normal form. Prenex normal form sentences in this logic with the quantifier-free part expressed in conjunctive normal form (CNF) are also called *quantified Boolean formulas* (QBF).

We introduce some additional notation for clarity of exposition. Given a propositional formula φ, its *support*, denoted $\mathsf{sup}(\varphi)$, is the set of variables that appear in φ. As mentioned earlier, we use bold-faced upper case English letters to denote sequences of variables. To reduce notational clutter, we use the same letter to denote the set underlying a sequence as well, when there is no confusion. For example, we speak of a propositional formula $\varphi(\mathbf{X})$ having support $\mathbf{X}$. If $\mathbf{Y} = (y_1, \ldots y_r)$ is a sequence of variables appearing in φ, and if $\mathbf{\Psi} = (\psi_1, \ldots \psi_r)$ is a sequence of propositional formulas such that no formula ψ_i has any variable in $\mathbf{Y}$ in its support, we use $\varphi[\mathbf{Y} \mapsto \mathbf{\Psi}]$ to denote the propositional formula obtained by substituting ψ_i for each y_i in φ. If $\mathbf{Y} = (y)$ and $\mathbf{\Psi} = (\psi)$ are singleton sequences, we simply use $\varphi[y \mapsto \psi]$ to denote the propositional formula resulting from substituting ψ for y in φ.

Let $\xi(\mathbf{X}) \equiv \exists\mathbf{Y}\ \forall\mathbf{Z}\ \exists\mathbf{U} \ldots \forall\mathbf{V}\ \exists\mathbf{W}\, \varphi(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{U}, \ldots \mathbf{V}, \mathbf{W})$ be a formula in QPL, where φ is a purely propositional formula. We wish to find Skolem functions for all existentially quantified variables in ξ. Since the domain of variables is $\{\mathsf{true}, \mathsf{false}\}$, each Skolem function is a mapping from $\{\mathsf{true}, \mathsf{false}\}^k$ to $\{\mathsf{true}, \mathsf{false}\}$, for some $k > 0$. Such a Skolem function can also be viewed as defining the truth semantics of a propositional formula over k variables. We therefore represent every Skolem function, say F, in QPL by a propositional formula, say $\psi^{(F)}$, such that F gives the truth semantics of $\psi^{(F)}$. Although the distinction between F and $\psi^{(F)}$ is significant (one is a function, the other is a formula), for notational convenience, we use the formula $\psi^{(F)}$ to refer to the Skolem function F, when there is no confusion. When F is implicit from the context, we simply use ψ instead of $\psi^{(F)}$.

Although the quantifier prefix of the formula ξ mentioned above has multiple quantifier alternations, it suffices to know how to generate Skolem functions for QPL formulas with *only a single block* of existential quantifiers. To see why this is so, suppose $\mathbf{\Psi_W}$ is a sequence of propositional formulas (representing Skolem functions), one for each variable w_i in $\exists\mathbf{W}\, \varphi$. By definition of Skolem functions, we have $\exists\mathbf{W}\, \varphi \Leftrightarrow \varphi[\mathbf{W} \mapsto \mathbf{\Psi_W}]$. Let φ' denote $\exists\mathbf{W}\, \varphi$. Since $\forall\mathbf{V}\ \exists\mathbf{W}\, \varphi \Leftrightarrow \forall\mathbf{V}\, \varphi' \Leftrightarrow \neg\exists\mathbf{V}\neg\varphi'$, if $\mathbf{\Psi_V}$ represents a sequence of Skolem functions for $\mathbf{V}$ in $\exists\mathbf{V}\, \neg\varphi'$, then $\forall\mathbf{V}\ \exists\mathbf{W}\, \varphi \Leftrightarrow \neg\big(\neg\varphi'[\mathbf{V} \mapsto \mathbf{\Psi_V}]\big) \Leftrightarrow \varphi'[\mathbf{V} \mapsto \mathbf{\Psi_V}] \Leftrightarrow \big(\varphi[\mathbf{W} \mapsto \mathbf{\Psi_W}]\big)[\mathbf{V} \mapsto \mathbf{\Psi_V}]$. By repeating the above steps, it is possible to successively eliminate all quantifiers in ξ. This also yields a sequence of Skolem functions $\mathbf{\Psi_Y}$, $\mathbf{\Psi_U}$, $\ldots$ $\mathbf{\Psi_W}$ for the existentially quantified variables in $\xi \equiv \exists\mathbf{Y}\ \forall\mathbf{Z}\ \exists\mathbf{U} \ldots \forall\mathbf{V}\ \exists\mathbf{W}\, \varphi(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{U}, \ldots \mathbf{V}, \mathbf{W})$. Note that the Skolem functions in $\mathbf{\Psi_Y}$ (for variables in $\mathbf{Y}$) have only the free variables $\mathbf{X}$ as arguments. Similarly, the Skolem functions in $\mathbf{\Psi_U}$ (for variables in $\mathbf{U}$) have only the variables in $\mathbf{X}, \mathbf{Y}$ and $\mathbf{Z}$ as arguments. By substituting $\mathbf{\Psi_Y}$ for $\mathbf{Y}$ in $\mathbf{\Psi_U}$, we obtain Skolem functions for variables in $\mathbf{U}$ in terms of only $\mathbf{X}$ and $\mathbf{Z}$, i.e. universally quantified variables appearing to the left of $\mathbf{U}$ in the quantifier prefix of ξ. It is easy to see that by repeating this process, we obtain Skolem functions for every existentially quantified variable in terms of (i) free variables $\mathbf{X}$, and (ii) universally quantified variables appearing to its left in the quantifier prefix of ξ.

In light of the above discussion, it makes sense to focus only on QPL formulas of the form $\exists\mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$ or $\forall\mathbf{X}\exists\mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$ for purposes of computing Skolem functions. Interestingly, with this restriction on the quantifier prefix, the problem of computing Skolem functions can also be viewed as one of *synthesis*. We elaborate more on this connection below.

2.1 The synthesis connection

Automatically and efficiently synthesizing correct systems from logical specifications is one of the holy grails of computer science. Suppose we wish to design a system with inputs $\mathbf{X}$ and outputs $\mathbf{Y}$. To avoid notational confusion, we call $\mathbf{X}$ as *system inputs*, and $\mathbf{Y}$ as *system outputs* to distinguish them from inputs and outputs of Skolem functions/formulas. A *relational specification* $\varphi(\mathbf{X}, \mathbf{Y})$ is a logical formula that implicitly relates desired values of system outputs with values of system inputs. Thus, every model of $\varphi(\mathbf{X}, \mathbf{Y})$ gives values of $\mathbf{X}$ and $\mathbf{Y}$ that corresponds to a desired output in response to a specific input. Monadic second order logic, temporal logic and several variants of these logics [32] have been widely used to specify desirable system behaviour. In general, the specification $\varphi(\mathbf{X}, \mathbf{Y})$ may permit multiple behaviours of the system outputs in response to a given input. A correct system design is required to produce any one of these allowed behaviours. It is also possible that for some values of the system inputs $\mathbf{X}$, there are no values of the system outputs $\mathbf{Y}$ that render $\varphi(\mathbf{X}, \mathbf{Y})$ true. In such cases, the specification cannot always be satisfied, no matter how we design the system. Such specifications are also called *unrealizable*. A correct synthesis procedure generates the system outputs $\mathbf{Y}$ as a function $\mathfrak{F}$ of the system inputs $\mathbf{X}$, such that $\forall \mathbf{X} \left(\exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \varphi(\mathbf{X}, \mathfrak{F}(\mathbf{X}))\right)$. If a specification is *realizable*, $\forall \mathbf{X} \exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$ is identically true; hence the requirement for synthesis simplifies to designing $\mathfrak{F}(\mathbf{X})$ such that it renders $\forall \mathbf{X}\, \varphi(\mathbf{X}, \mathfrak{F}(\mathbf{X}))$ identically true as well. Interestingly, even if a specification is unrealizable, it may be perfectly meaningful to synthesize $\mathfrak{F}(\mathbf{X})$ such that $\forall \mathbf{X} \left(\exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \varphi(\mathbf{X}, \mathfrak{F}(\mathbf{X}))\right)$ holds. Indeed, as long as there is at least one way to generate system outputs in response to a given input such that the specification φ is satisfied, we want the system outputs generated by the synthesized system to satisfy the specification. In other cases, there are effectively no requirements on the system outputs.

Deciding realizability of a specification, and synthesizing a realizable specification are computationally hard problems in general. A relatively simpler cousin of the general synthesis problem, called *Boolean Functional Synthesis*, has recently received a lot of attention [36, 47, 38, 26, 53, 65, 2, 3, 54, 4, 52, 5, 29]. This problem is "simpler" in the sense that it concerns synthesis of Boolean functions, represented as Boolean circuits with AND, OR and NOT gates, from propositional logic specfications. Since every Boolean circuit corresponds to a propositional formula and vice versa, Boolean functional synthesis for $\varphi(\mathbf{X}, \mathbf{Y})$ with system inputs $\mathbf{X}$ and system outputs $\mathbf{Y}$ can be seen to be equivalent to computing Skolem functions for the QPL formula $\forall \mathbf{X} \exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$. Therefore, we refer to the problem of computing Skolem functions for QPL and that of Boolean functional synthesis interchangeably. For notational convenience, we use *Boolean Skolem function synthesis*, or $\mathsf{BoolSkFnSyn}$ for short, to refer to either problem in the remainder of this article. It is worth emphasizing here that Boolean Skolem function synthesis is distinct from the problem of *combinational logic synthesis and optimization* [24]. In the former, we start from a relational specification that doesn't necessarily give the system outputs explicitly as functions of system inputs, and our primary task is to synthesize these outputs as Boolean functions of system inputs. In contrast, in combinational logic synthesis and optimization, we are given system outputs as explicit Boolean functions of system inputs, and our goal is to implement these functions optimally as Boolean circuits with specified gate types (viz. NAND, NOR, XOR, etc.).

In the context of QPL, the specification $\varphi(\mathbf{X}, \mathbf{Y})$ and the Skolem functions for $\mathbf{Y}$ can be represented in several ways. Some commonly used representations include lists of clauses for propositional formulas in conjunctive normal form (CNF), Boolean circuits, reduced ordered binary decision diagrams (ROBDDs) [62], and-inverter graphs (AIGs) [40], decision lists, decision trees etc. The choice of representation has a bearing on the computational complexity of BoolSkFnSyn; hence it is important to spell out the representation clearly when discussing a solution to the problem. Interestingly, all the representations mentioned above can be translated to Boolean circuits with AND, OR and NOT gates with at most a linear blow-up. Hence, we consider Boolean circuits with AND, OR and NOT gates as a unifying representation for both relational specifications and for Skolem functions. Computational hardness (lower bound) results based on Boolean circuit representations naturally hold when the other representations are used as well. A particularly convenient form of Boolean circuits are those in which every NOT gate is immediately fed by a circuit input (labeled by a variable). Such circuits are also called *Negation Normal Form (or NNF)* circuits. For notational convenience, we treat every NOT gate fed by a circuit input labeled v in a NNF circuit as a new circuit input labeled $\neg v$. Thus, an NNF circuit can be viewed as one containing only AND and OR gates, with the circuit inputs labeled by *literals* over the set of variables, i.e. variables and their negations. It is easy to see that every Boolean circuit can be compiled to a NNF circuit that computes the same function as the original circuit, and is at most twice its size.

2.2 The unification connection

The BoolSkFnSyn problem is related to that of *Boolean unification* – a classical problem studied by George Boole [13] and Leopold Löwenheim [44] much before Alan Turing and Alonzo Church formalized the notion of computation. The interested reader is referred to an excellent (albeit, dated) survey by Martin and Nipkow [48] for details about the Boolean unification problem. For our purposes, Boolean unification may be viewed as asking the following question: *Given two Boolean functions* $F, G : \{\mathsf{true}, \mathsf{false}\}^n \rightarrow \{\mathsf{true}, \mathsf{false}\}$*, find a map* $\mathfrak{F} : \{\mathsf{true}, \mathsf{false}\}^m \rightarrow \{\mathsf{true}, \mathsf{false}\}^n$*, where* $m \geq 0$ *such that* $F\big(\mathfrak{F}(\sigma)\big) = G\big(\mathfrak{F}(\sigma)\big)$ *for all* $\sigma \in \{\mathsf{true}, \mathsf{false}\}^m$*, or report that no such map exists.* The map $\mathfrak{F}$, if it exists, is called a *unifier* of F and G. In general, there can be zero, one or multiple unifiers of F and G. A unifier $\mathfrak{F} : \{\mathsf{true}, \mathsf{false}\}^m \rightarrow \{\mathsf{true}, \mathsf{false}\}^n$ is said to be *more general* than unifier $\mathfrak{G} : \{\mathsf{true}, \mathsf{false}\}^\ell \rightarrow \{\mathsf{true}, \mathsf{false}\}^n$ if there exists a map $\mathfrak{H} : \{\mathsf{true}, \mathsf{false}\}^\ell \rightarrow \{\mathsf{true}, \mathsf{false}\}^m$ such that $\mathfrak{F}\big(\mathfrak{H}(\widehat{\sigma})\big) = \mathfrak{G}(\widehat{\sigma})$ for all $\widehat{\sigma} \in \{\mathsf{true}, \mathsf{false}\}^\ell$. A *most general unifier* of F and G is a unifier that is more general than all unifiers of F and G. By a result due to Boole [13], we know that if two Boolean functions F and G are unifiable, there exists a most general unifier of F and G.

To see the connection of Boolean unification with BoolSkFnSyn, let $\varphi(\mathbf{X}, \mathbf{Y})$ be a propositional relational specification such that (i) $|\mathbf{X}| + |\mathbf{Y}| = n$, and (ii) the truth semantics of φ is given by $F(\mathbf{X}, \mathbf{Y})$. Let $G(\mathbf{X}, \mathbf{Y})$ denote the truth semantics of $\exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$, viewed as a function of $\mathbf{X}$ and (redundantly) of $\mathbf{Y}$. A solution to the BoolSkFnSyn problem for φ yields a vector $\mathbf{\Psi}$ of propositional formulas (representing Skolem functions), one for each variable in $\mathbf{Y}$, that can be converted to a unifier of F and G as follows. Note that $\mathbf{\Psi}$ represents a mapping from $\{\mathsf{true}, \mathsf{false}\}^{|\mathbf{X}|}$ to $\{\mathsf{true}, \mathsf{false}\}^{|\mathbf{Y}|}$. Let $\mathsf{Id}_{|\mathbf{X}|}$ be the identity mapping on $\{\mathsf{true}, \mathsf{false}\}^{|\mathbf{X}|}$. The concatentation of $\mathsf{Id}_{|\mathbf{X}|}$ and $\mathbf{\Psi}$, denoted $(\mathsf{Id}_{|\mathbf{X}|}, \mathbf{\Psi})$, gives a vector of functions mapping $\{\mathsf{true}, \mathsf{false}\}^{|\mathbf{X}|}$ to $\{\mathsf{true}, \mathsf{false}\}^{|\mathbf{X}|+|\mathbf{Y}|}$, such that $F\big(\mathsf{Id}_{|\mathbf{X}|}(\sigma), \mathbf{\Psi}(\sigma)\big) = G\big(\mathsf{Id}_{|\mathbf{X}|}(\sigma), \mathbf{\Psi}(\sigma)\big)$ for all $\sigma \in \{\mathsf{true}, \mathsf{false}\}^{|\mathbf{X}|}$. The above discussion shows that given $\varphi(\mathbf{X}, \mathbf{Y})$, if F and G are chosen appropriately, then specific *unifiers* for F and G correspond to Skolem functions for $\mathbf{Y}$ in $\varphi(\mathbf{X}, \mathbf{Y})$ Indeed, if the unifier is a most general unifier, then the Skolem functions turn out to be specific instantiations of this most general unifier. Interestingly, algorithms for finding the most general unifier in Boolean unification were given by both Boole [13] and Lowenheim [44] in their early work. These and other variant algorithms for finding most general unifiers in Boolean unification were experimentally evaluated in [45]. Applications of Boolean unification have also been reported in [12, 16, 59, 46]. Unfortunately, solving BoolSkFnSyn using the Boolean unification approach turns out to be too inefficient for use in practical applications with thousands of variables and beyond.

3 Applications of Boolean Skolem function synthesis

Before delving deeper into the compuational aspects of BoolSkFnSyn, let us look at a few interesting applications of the problem. These applications provide strong motivation for developing algorithms for BoolSkFnSyn that work well in practice, despite non-trivial worst-case complexity-theoretic lower bounds.

We start with a particularly challenging application that illustrates why an efficient algorithmic solution of BoolSkFnSyn can have far-reaching implications in practice. Consider a system with a single $2n$-bit unsigned integer input $\mathbf{X}$, and two n-bit unsigned integer outputs $\mathbf{Y}_1$ and $\mathbf{Y}_2$. Suppose the relational specification is given as $F_{\mathsf{fact}}(\mathbf{X}, \mathbf{Y}_1, \mathbf{Y}_2) \equiv ((\mathbf{X} = \mathbf{Y}_1 \times_{[n]} \mathbf{Y}_2) \wedge (\mathbf{Y}_1 \neq 1) \wedge (\mathbf{Y}_2 \neq 1))$, where $\times_{[n]}$ denotes n-bit unsigned integer multiplication. This specification requires that $\mathbf{Y}_1$ and $\mathbf{Y}_2$ are non-trivial factors of $\mathbf{X}$. Note, however, that if $\mathbf{X}$ represents a prime number, there are no values of $\mathbf{Y}_1$ and $\mathbf{Y}_2$ that satisfy the specification. Technically, the specification in unrealizable. Nevertheless, we are interested in obtaining values of $\mathbf{Y}_1$ and $\mathbf{Y}_2$ that satisfy the specification, whenever possible. Significantly, the above specification can be encoded as a Boolean formula of size $\mathcal{O}(n^2)$ over the individual bits of $\mathbf{X}$, $\mathbf{Y}_1$ and $\mathbf{Y}_2$. However, if we want to express $\mathbf{Y}_1$ and $\mathbf{Y}_2$ directly as Boolean functions of $\mathbf{X}$, our task turns out to be significantly harder. In fact, there are no known polynomial-sized Boolean functions (represented as circuits of AND, OR and NOT gates) that can express individual bits of $\mathbf{Y}_1$ and $\mathbf{Y}_2$ directly in terms of the individual bits of $\mathbf{X}$. Otherwise, we could efficiently factorize products of n-bit prime numbers, rendering cryptographic systems vulnerable to attacks. This application also illustrates how relational specifications can be more natural and succinct than expressing outputs directly as functions of inputs.

As another application, we consider satisfiability checking of quantified boolean sentences (also called QBF-SAT), which is increasingly being used in diverse applications such as planning, model checking, non-monotonic reasoning, reactive synthesis, games, equivalence checking, circuit repair, program synthesis etc. An excellent survey of such applications can be found in [57]. Given the sophistication of modern QBF-SAT solvers, it is hard to rule out bugs in solver implementations. It is therefore desirable that when a QBF-SAT solver is invoked, it not only produces a "Yes"/"No" answer to the decision problem, but also a certificate that can be independently (machine-)checked to validate the correctness of the answer. Multiple notions of certificates have been used in the literature [57, 8, 50], including the use of Skolem functions for existentially quantified variables in valid QBFs, and the use of Herbrand functions[3] for universally quantified variables in unsatisfiable QBFs. In addition to their use as certificates of QBF-SAT results, Skolem function based certificates also have independent value as they can be used for other objectives, such as, to extract a feasible plan in a robotic planning problem, a replacement sub-circuit in a circuit repair problem, a program fragment in automated program synthesis, a winning strategy in a game. As discussed earlier, knowing how to synthesize Skolem functions for QBF formulas of the form $\forall \mathbf{X}\ \exists \mathbf{Y}\ \varphi(\mathbf{X}, \mathbf{Y})$ suffices to generate Skolem functions (resp. Herbrand functions) for all existentially (resp. universally) quantified variables in a QBF. This underscores the importance of the BoolSkFnSyn problem.

Talking of synthesis, recall that BoolSkFnSyn can be viewed as a simpler version of the more general reactive synthesis problem (see [25] for a survey). It turns out that several algorithmic approaches to reactive synthesis use BoolSkFnSyn as a key step (see e.g [14, 35]). Hence, a practically efficient algorithmic solution to BoolSkFnSyn benefits reactive synthesis as well.

4 Boolean Skolem function synthesis through lens of computation

Recall the definition of BoolSkFnSyn from Section 2. We are given a propositional formula $\varphi(\mathbf{X}, \mathbf{Y})$, specifying a relation between system inputs $\mathbf{X}$ and system outputs $\mathbf{Y}$. For notational convenience, we use m to denote $|\mathbf{X}|$ and n to denote $|\mathbf{Y}|$. The BoolSkFnSyn problem requires us to find a vector of propositional formulas (representing Boolean functions) $\mathbf{\Psi}(\mathbf{X}) = \big(\psi_1(\mathbf{X}), \ldots \psi_n(\mathbf{X})\big)$ such that $\forall \mathbf{X}\big(\exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y}) \Leftrightarrow \varphi(\mathbf{X}, \mathbf{\Psi}(\mathbf{X}))\big)$ is true. The formula $\psi_i(\mathbf{X})$ represents a Skolem function for y_i in φ, and $\mathbf{\Psi}(\mathbf{X})$ is called a *Skolem functon vector* for $\mathbf{Y}$ in φ. As discussed earlier, we represent all Skolem functions and propositional formulas by Boolean circuits comprised of AND, OR and NOT gates.

Example 2 Consider the relational specification $\varphi(\mathbf{X}, \mathbf{Y}) \equiv (x_1 \vee y_2) \wedge (\neg x_2 \vee \neg x_1 \vee y_1)$. A few (among many possible) Skolem function vectors for $\mathbf{Y}$ in φ are $\big(\mathsf{true}, \mathsf{true}\big)$, $\big(\mathsf{true}, \neg x_1\big)$, $\big(x_1, \neg x_1\big)$, $\big(x_2, \neg x_1\big)$, where each tuple represents $(\psi_1(\mathbf{X}), \psi_2(\mathbf{X}))$.

While a given problem instance may admit multiple Skolem function vectors, a solution to BoolSkFnSyn seeks only one such vector. Thus, there may not be a unique solution to an instance of BoolSkFnSyn.

It is not hard to see that BoolSkFnSyn can be solved in time (and space) exponential in $|\varphi|$ in the worst-case, simply by brute-force enumeration of all possible values of $\mathbf{X}$ and $\mathbf{Y}$. However, does the problem admit more efficient solutions? If $|\mathbf{Y}| = n = 1$, it turns out that there is a surprisingly efficient solution. To understand this, we need some additional notation. Let α be a propositional formula and $v \in \mathsf{sup}(\alpha)$. We use $\alpha|_v$ (resp. $\alpha|_{\neg v}$) to denote the positive (resp. negative)

---

[3] A Herbrand function for universally quantified variables in a quantified propositional sentence φ may be thought of as Skolem functions for existentially quantified variables in $\neg\varphi$.

co-factor of α with respect to v, i.e. α with v set to true (resp. false). It can now be verified that if $\varphi(\mathbf{X}, y)$ is a specification with a single system output y, then both $\varphi|_y$ and $\neg(\varphi|_{\neg y})$ serve as Skolem functions for y in φ. This technique for obtaining a Skolem function for a single system output is also called *self-substitution*, and has been used in several prior works [66, 36, 26, 38, 2, 29]. In fact, if $\beta(\mathbf{X})$ denotes $\varphi|_y$ and $\gamma(\mathbf{X})$ denotes $\varphi|_{\neg y}$, then the entire set of Skolem functions for y in φ can be parametrically represented as $\big(\neg\gamma(\mathbf{X}) \wedge \beta(\mathbf{X})\big) \vee \big((\beta(\mathbf{X}) \Leftrightarrow \gamma(\mathbf{X})) \wedge \delta(\mathbf{X})\big)$, where $\delta(\mathbf{X})$ is *any* Boolean function on $\mathbf{X}$ [66, 36].

An obvious question to ask at this point is whether the simple solution for $|\mathbf{Y}| = 1$ can be extended to the case where $|\mathbf{Y}| > 1$. Unfortunately, this turns out to be more difficult, and there are complexity-theoretic barriers along the way. Nevertheless, the underlying idea for the $|\mathbf{Y}| = 1$ case can be generalized to obtain some insights. Towards this end, let $y_1 \prec y_2 \cdots \prec y_n$ be a (arbitrary) linear ordering of the system outputs, and let $\mathbf{Y}_i^j$ denote the subsequence $(y_i, \ldots y_j)$ of $\mathbf{Y}$, for $1 \leq i \leq j \leq n$. Furthermore, let $\varphi^{(i-1)}(\mathbf{X}, \mathbf{Y}_i^n)$ denote $\exists \mathbf{Y}_1^{i-1}\, \varphi(\mathbf{X}, \mathbf{Y})$, where $\varphi^{(0)}$ is defined to be φ. For every i in 1 to n in that order, suppose we view the formula $\varphi^{(i-1)}(\mathbf{X}, y_i, \mathbf{Y}_{i+1}^n)$ as a specification with system inputs $\mathbf{X} \cup \mathbf{Y}_{i+1}^n$ and a single system output y_i. We can now apply the reasoning for synthesizing a single Skolem function, as discussed above, to obtain a Skolem function for y_i in terms of $\mathbf{X} \cup \mathbf{Y}_{i+1}^n$. Let $\psi_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)$ be such a Skolem function for y_i, i.e. $\varphi^{(i-1)}(\mathbf{X}, \psi_i, \mathbf{Y}_{i+1}^n) \Leftrightarrow \exists y_i \varphi^{(i-1)}(\mathbf{X}, y_i, \mathbf{Y}_{i+1}^n)$. Once we have computed ψ_i for $i \in \{1, \ldots n\}$ in this manner, we can substitute ψ_{i+1} through ψ_n for y_{i+1} through y_n respectively, in the definition of ψ_i to obtain a Skolem function for y_i as a function of only $\mathbf{X}$. This approach is widely used in the $\mathsf{BoolSkFnSyn}$ literature [36, 37, 38, 26, 2, 4, 29], and we follow it for the rest of our discussion. Note that this allows us to focus on synthesizing ψ_i in terms of $\mathbf{X}$ and $\mathbf{Y}_{i+1}^n$, instead of synthesizing it directly in terms of $\mathbf{X}$. Generalizing the idea of the solution when we have a single system output, it can be shown that both $\neg(\varphi^{(i-1)}|_{\neg y_i})$ and $\varphi^{(i-1)}|_{y_i}$ serve as Skolem functions for y_i (in terms of $\mathbf{X}$ and $\mathbf{Y}_{i+1}^n$). Furthermore, if $\beta_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)$ denotes $\varphi^{(i-1)}|_{y_i}$ and $\gamma_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)$ denotes $\varphi^{(i-1)}|_{\neg y_i}$, then every Skolem function for y_i can be parametrically represented as $\big(\neg\gamma_i(\mathbf{X}, \mathbf{Y}_{i+1}^n) \wedge \beta_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)\big) \vee \big((\beta_i(\mathbf{X}, \mathbf{Y}_{i+1}^n) \Leftrightarrow \gamma_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)) \wedge \delta_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)\big)$, where $\delta_i(\mathbf{X}, \mathbf{Y}_{i+1}^n)$ is *any* Boolean function on $\mathbf{X}$ and $\mathbf{Y}_{i+1}^n$.

While the above discussion may seem to imply that there is an easy way to solve $\mathsf{BoolSkFnSyn}$ in general, the difficulty in the above approach lies in computing a good linear ordering of y_i's and also in computing $\varphi^{(i-1)}$ for $1 \leq i \leq n$. Experiments, e.g, from [38, 2, 29], show that using different linear orderings affects the time taken for synthesizing functions considerably. For values of $|\mathbf{X}| = m$ and $|\mathbf{Y}| = n$ running into thousands, these issues can pose enormous scalability challenges in practice. However, the computational hurdles are not restricted to only the approach discussed above. It turns out that any other algorithmic technique to solve $\mathsf{BoolSkFnSyn}$ must also encounter scalability hurdles in the worst-case. Computational complexity theory provides the tools necessary to reason about these challenges, by allowing us to derive lower bounds on computational resources (viz. space and time) needed to solve $\mathsf{BoolSkFnSyn}$ in general. We elaborate on this in the next couple of sections.

4.1 A quick primer on the polynomial hierarchy and related complexity classes

In computational complexity theory, a *decision problem* is one that has a "Yes"/"No" answer. An example of such a problem is: *Given a propositional formula φ, is φ satisfiable?* A *function problem* generalizes a decision problem by allowing the answer to be more general than "Yes"/"No". For example, we could ask: *Given a propositional formula φ in conjunctive normal form, what is the maximum number of clauses of φ that can be simultaneously satisifed?* For a large class of function problems, an efficient solution to an appropriately defined decision version of the problem implies an efficient solution to the function problem itself. Studying the complexity of decision problems has therefore been a major focus of complexity theoretic investigations. A decision problem can also be viewed as a *language recognition* problem, where the input is presented as a finite string over the alphabet $\{0, 1\}$, and the set of all input strings that yield a "Yes" answer comprises the language L corresponding to the problem. Thus, given an input string str representing an instance of the problem, the decision problem effectively asks if $\mathsf{str} \in L$. This is equivalent to asking if the problem instance has a "Yes" answer.

The complexity class P (resp. NP) consists of the set of all languages accepted by deterministic (resp. non-deterministic) Turing machines in time that grows at most polynomially in the size of the input. The class coNP is the set of all languages, the complement of which are in NP. The polynomial hierarchy generalizes these classes by defining two inter-related sub-hierarchies – the Σ^{P}-hierarchy and the Π^{P}-hierarchy. We start by defining $\Sigma_0^{\mathsf{P}} = \Pi_0^{\mathsf{P}} = \mathsf{P}$. For every $n \in \mathbb{N} \setminus \{0\}$, we then define Σ_n^{P} and Π_n^{P} inductively as follows, where $\{0, 1\}^*$ denotes the set of all finite strings over $\{0, 1\}$, and $|\mathsf{str}|$ denotes the length of the string str.

- Σ_n^{P} consists of all languages/problems L such that there exists a language $L' \in \Pi_{n-1}^{\mathsf{P}}$ and a polynomial q such that
$$\forall x \in \{0,1\}^* \, x \in L \Leftrightarrow \exists y \in \{0,1\}^*, |y| \le q(|x|) \text{ and } (x, y) \in L'.$$
- Π_n^{P} consists of all languages/problems L such that there exists a language $L' \in \Sigma_{n-1}^{\mathsf{P}}$ and a polynomial q such that
$$\forall x \in \{0,1\}^* \, x \in L \Leftrightarrow \forall y \in \{0,1\}^*, |y| \le q(|x|) \Rightarrow (x, y) \in L'.$$

It is easy to see from the definitions that $\mathsf{NP} = \Sigma_1^{\mathsf{P}}$ and $\mathsf{coNP} = \Pi_1^{\mathsf{P}}$. The hierarchy of complexity classes defined above is known as the *Polynomial Hierarchy* (henceforth, PH). The PH is said to *collapse to level $i \in \mathbb{N}$* if $\Sigma_i^{\mathsf{P}} = \Sigma_{i+1}^{\mathsf{P}}$. Notice that if PH collapses to level 0, then $\mathsf{P} = \mathsf{NP}$. It is widely believed that PH is a strict infinite hierarchy and does not collapse to any finite level. However, this is only a conjecture; the question of whether PH indeed collapses to any finite level has remained open for decades, and is one of the outstanding open problems in computational complexity theory.

The classes in PH are also related to the notion of *oracle computation* or *relative computation*, referred to in Section 1. Recall that an oracle machine is a Turing machine with access to a "black-box" (oracle) that can provide "Yes"/"No" answers to a specific class of decision problem in a single step. If oracles are restricted to be Turing machines themselves with well-defined resource constraints, we obtain an alternative characterization of the complexity classes in PH. The interested reader is referred to [7] for details. For our purposes, it suffices to note that P^{NP} is one such complexity class obtained by considering polynomial-time Turing machines with access to an NP oracle. That is, any problem in this class can be solved by a deterministic Turing machine in polynomially many steps, if it is allowed to make at most polynomially many calls to an NP oracle. In fact, the complexity class P^{NP} can be shown to coincide with $\Sigma_2^{\mathsf{P}} \cap \Pi_2^{\mathsf{P}}$, and hence is within the second level of the polynomial hierarchy!

Just as P is the class of languages accepted by deterministic Turing machines running for at most polynomial time, PSPACE denotes the class of languages accepted by deterministic Turing machines that use atmost polynomial space. It is known that non-determinism does not add power in this case, i.e., $\mathsf{NPSPACE} = \mathsf{PSPACE}$. Also it is known that $\mathsf{PH} \subseteq \mathsf{PSPACE}$, i.e., the entire polynomial hierarchy is contained in the class PSPACE, thereby making this a very expressive class. Notice, however, that if a Turing machine can run for exponential time, then it can indeed simulate a Turing machine that is allowed to use only polynomial space. The class of languages accepted by deterministic Turing machines running for exponential time is denoted EXP, and we immediately see that $\mathsf{PSPACE} \subseteq \mathsf{EXP}$. We refer the interested reader to excellent textbooks, e.g., [7], in this area for more information about complexity classes and their relations.

4.2 Computational hardness of Boolean Skolem Function Synthesis

With the above notations, we can now present complexity-theoretic hardness results for BoolSkFnSyn. As mentioned earlier, we assume the input and output of BoolSkFnSyn are represented as Boolean circuits. It turns out that three conditional results can be shown, two of which are related to the collapse of the polynomial hierarchy defined above.

The first result is about time-complexity. Specifically, any algorithm that solves BoolSkFnSyn must take *super-polynomial* (i.e., asymptotic growth greater than that of any polynomial) time in the worst case, unless the polynomial hierarchy collapses to the first level (i.e., $\mathsf{P} = \mathsf{NP}$). Since the question of whether $\mathsf{P} = \mathsf{NP}$ has remained open for decades, with the general wisdom being $\mathsf{P} \neq \mathsf{NP}$, it is highly unlikely that all instances of BoolSkFnSyn can be solved in polynomial time. This easily follows from the observation that propositional satisfiability can be reduced to BoolSkFnSyn where we have no system inpus $\mathbf{X}$.

Next, we inquire about the space complexity of BoolSkFnSyn, and ask if it is possible to solve BoolSkFnSyn *compactly.* More precisely, do there always exist polynomial-sized Skolem functions for instances of BoolSkFnSyn, even if it takes exponential time to synthesize them? Again, the answer turns out to be negative, but with a stronger condition. It is shown in [3, 5] that unless the polynomial hierarchy collapses to the second level, there must exist instances of BoolSkFnSyn for which any algorithm must generate super-polynomial sized Skolem functions.

The above results provide conditional super-polynomial time and space lower bounds for BoolSkFnSyn. On the other hand, a trivial upper bound was mentioned earlier, namely, BoolSkFnSyn can be solved in exponential time and space. A naive exponential time algorithm would be to enumerate all possible values of system inputs $\mathbf{X}$, and for each such valuation, check by enumeration again if there exists a valuation of the system outputs $\mathbf{Y}$ that satisfies the given specification. Since we are concerned about Boolean specifications, this can be done in time exponential in $|\mathbf{X}|$ and $|\mathbf{Y}|$; of course, in doing so, it may produce Skolem functions of at most exponential size.

Given the large gap between a polynomial lower bound and an exponential upper bound, a natural question is whether this gap can be narrowed or bridged. In [3, 5], it is shown that under a stronger hypothesis, this gap can in fact be completely eliminated giving us optimal and tight (albeit conditional) complexity bounds. To understand this result, let us start by considering two unproven complexity-theoretic conjectures. The exponential-time hypothesis ETH [34] and its non-uniform variant, $\mathsf{ETH}_{\mathsf{nu}}$ [18], are unproven computational hardness conjectures that have been used to show that several classical decision, functional and parametrized NP-complete problems are unlikely to have sub-exponential algorithms. These conjectures are also widely believed to be true. Formally, $\mathsf{ETH}_{\mathsf{nu}}$ – the variant that we need – states that there is no family of algorithms (one for each input-size n) that can solve the n-variable instance of the propositional satisfiability problem (the canonical NP-complete problem) in sub-exponential time (i.e., in time that is lower than any exponential function of n, also written $2^{o(n)}$). By adapting the earlier result, one can now show that, unless the non-uniform exponential time hypothesis $\mathsf{ETH}_{\mathsf{nu}}$ fails, there exist instances of BoolSkFnSyn for which any algorithm for must generate exponential-sized Skolem functions. Notice that this immediately implies exponential time complexity as well, since generating an output of size $f(n)$ requires at least $f(n)$ time.

Summarizing, we obtain the following theorem, whose details and proof can be found in [3, 5].

Theorem 1 *1.* BoolSkFnSyn *can be solved in exponential time and space.*

2. There exists no algorithm for BoolSkFnSyn *that*
   (a) always takes polynomial time on all inputs, unless PH *collapses to level 0.*
   (b) always generates polynomial sized Skolem functions, unless PH *collapses to the second level.*
   (c) always generates sub-exponential sized Skolem functions (and takes sub-exponential time), unless the non-uniform exponential-time hypothesis fails.

Together these results imply that BoolSkFnSyn is unlikely to have polynomial-time or polynomial-space algorithms in general. Any such efficient algorithm must necessarily falsify one of the above well-regarded and intensely researched conjectures in complexity theory.

4.3 Exploiting the structure of the specification

Given a Boolean relational specification as a circuit, we now ask if there are conditions on the structure/representation of the circuit that can be exploited to efficiently synthesize Skolem functions. Indeed, this turns out to be the case, and we discuss some such cases below.

4.3.1 Unate variables

Recall that BoolSkFnSyn requires us to synthesize the entire Skolem function vector, i.e., Skolem functions for all system outputs in $\mathbf{Y}$. However, synthesizing Skolem functions for some system output variables may be easier than that for others. For example, consider the case of *unate* variables. The formula φ is said to be *positive unate* in $v \in \mathsf{sup}(\varphi)$ iff $\varphi|_{\neg v} \Rightarrow \varphi|_{v}$. Similarly, φ is said to be *negative unate* in v iff $\varphi|_{v} \Rightarrow \varphi|_{\neg v}$. Finally, φ is *unate* in v if it is either positive unate or negative unate in v. If φ is positive unate in v, it immediately follows that $\exists v\, \varphi \Leftrightarrow (\varphi|_{v} \vee \varphi|_{\neg v}) \Leftrightarrow \varphi|_{v}$. As a result, if v is a system output, the constant function true serves as a correct Skolem function for v in φ. Similarly false serves a correct Skolem function for v in φ if φ is negative unate in φ. Thus, we obtain,

Proposition 1 *If a specification $\varphi(\mathbf{X}, \mathbf{Y})$ is unate in $y_i \in \mathbf{Y}$, one can generate constant-sized Skolem functions for y_i in φ in constant time.*

Substituting a constant Skolem function for $y_i \in \mathbf{Y}$ in the specification $\varphi(\mathbf{X}, \mathbf{Y})$ and simplifiying it may, in turn, reveal that the simplified specification is unate in y_j (distinct from y_i), even if the original specification was not unate in y_j. It is therefore beneficial to iterate through this process of detecting if a specification is unate in a system output variable and substituting a constant Skolem function for the variable to simplify the specification.

Example 3 Consider the specification $\varphi \equiv (\neg x_1 \vee y_1) \wedge (x_1 \vee \neg x_2 \vee y_1 \vee \neg y_2) \wedge (\neg x_1 \vee \neg x_2 \vee y_2 \vee y_3) \wedge (x_2 \vee \neg y_3 \vee y_2)$. Applying the checks for positive and negative unateness described above, it is easy to verify that φ is only positive unate in y_1, and neither positive nor negative unate in y_2 or y_3. If we now set the Skolem function for y_1 to the constant true, the specification simplifies to $\varphi|_{y_1} \equiv (\neg x_1 \vee \neg x_2 \vee y_2 \vee y_3) \wedge (x_2 \vee \neg y_3 \vee y_2)$. Using the unateness checks again, we now find that $\varphi|_{y_1}$ is positive unate in y_2, but neither positive nor negative unate in y_3. Setting the Skolem function for y_2 to true, the specification further simplifies to $\big(\varphi|_{y_1}\big)\,|_{y_2} \equiv \mathsf{true}$. Hence, any Skolem function for y_3 suffices; in particular, we choose $y_3 \equiv \mathsf{false}$. We have thus solved the BoolSkFnSyn problem for the given specification, obtaining the constant Skolem functions $\psi_1 \equiv \psi_2 \equiv \mathsf{true}$ and $\psi_3 \equiv \mathsf{false}$.

From the definition of unateness, we can see that checking unateness can be reduced to checking (un)satisfiability of a propositional formula: φ is positive (resp. negative) unate in v iff the formula $\varphi|_{\neg v} \wedge \neg\varphi|_{v}$ (resp. $\varphi|_{v} \wedge \neg\varphi|_{\neg v}$) is unsatisfiable. A variant of this unateness check is used in [6] and other recent approaches to BoolSkFnSyn (e.g., [4, 5]). In the other direction, checking validity of an arbitrary formula φ can be reduced to checking if the formula $z \vee \varphi$ is positive unate in z, where $z \notin \mathsf{sup}(\varphi)$. Thus, unateness checking is coNP-hard, and cannot be done in polynomial time unless P = NP. However, we can have sufficient conditions for unateness that are checkable in polynomial time. For example, if v (resp. $\neg v$) is a *pure literal* in φ, i.e., the negation of the literal does not appear as the label of any leaf in a NNF circuit representation of φ, then φ is positive (resp. negative) unate in v_i. The above structural condition can clearly be checked in time linear in the size of the NNF circuit representing φ.

4.3.2 Functionally determined or implicitly defined variables

Suppose the specification φ *uniquely* defines a system output variable as a function of system input variables and other system output variables. We call such a variable *functionally determined* or *implicitly defined* in φ. For example, if $\varphi \equiv (\neg y_i \vee y_j) \wedge (\neg y_i \vee x_k) \wedge (y_i \vee \neg y_j \vee \neg x_k) \wedge \cdots$, then we can infer $(y_i \Leftrightarrow (y_j \wedge x_k))$ and hence, y_i is functionally determined (henceforth called FD) in φ. The implied functional dependencies like $(y_i \Leftrightarrow (y_j \wedge x_k))$ are called *functional definitions* of FD variables. Given a set $\mathbf{T} \subseteq \mathbf{Y}$ of FD system outputs in φ, we let $\mathsf{Fun}_{\mathbf{T}}$ denote the conjunction of functional definitions of all variables in $\mathbf{T}$. We say that $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ is an *acyclic system of functional definitions* if no variable in $\mathbf{T}$ transitively depends on itself via the functional definitions in $\mathsf{Fun}_{\mathbf{T}}$. The main observation is that for a given acyclic system $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ obtained from φ, we can simply replace each of the output variables in $\mathbf{T}$ by their functional definitions. Recall that these functional definitions are in terms of system inputs and other system outputs. Thus, once Skolem functions for all system outputs other than those in $\mathbf{T}$ are generated, we can generate Skolem functions for those in $\mathbf{T}$ simply by substituting the already generated Skolem functions in the functional definitions in $\mathsf{Fun}_{\mathbf{T}}$. This can be done in polynomial time by effectively connecting the outputs of sub-circuits representing already generated Skolem functions to corresponding inputs of sub-circuits representing functional definitions in $\mathsf{Fun}_{\mathbf{T}}$.

The above idea is remarkably simple and results in considerable simplification in practical benchmarks. The reason is that functionally determined variables occur widely in practice and are often easy to identify. For instance, specifications containing functionally determined variables arise naturally when a non-CNF Boolean formula is converted to CNF via Tseitin encoding [67], and are easily identifiable as patterns in the formula. Given the widespread use of Tseitin encoding in obtaining CNF formulas, such variables have a surprisingly large impact on benchmarks. As a result many practical tools for BoolSkFnSyn, (including [53, 5, 4, 29]) first identify and eliminate (at least some!) functionally determined variables before processing the formulas.

A note about Beth definability, as applied to quantified propositional formulas, is pertinent here. By a celebrated theorem of Beth [10], a system output y_i that is implicitly defined by a specification φ also has an expxlicit definition in terms of the system inputs and other system outputs. Such an explicit definition can indeed serve as the functional definition for y_i. However, Beth's theorem doesn't immediately give us an explicit definition of y_i; indeed, it can be computationally expensive to extract an explicit definition of y_i from φ in general. Practical tools therefore often use a range of heuristics to efficiently extract explicit definitions of implicitly defined system output variables. Fortunately, for variables introduced by Tseitin encoding, this can be done easily by matching patterns of clauses in a given CNF formula, as was illustrated in the example above. Such techniques, also called *syntactic gate extraction* (see e.g. [27]), are incomplete in general, but can be very effective in practice when reasoning about specifications containing Tseitin variables. In a recent work [60], a practically efficient, sound and complete *semantic gate extraction* technique for extracting explicit definitions of all implicitly defined variables, has been proposed. Incorporation of such techniques in Boolean Skolem function synthesis tools is likely to result in improved performance of such tools in practice.

4.3.3 Using maximal falsifiable sets of input clauses

Yet another class of specifications that admit relatively efficient synthesis in practice, follows from the work of [17]. Consider a specification $\varphi(\mathbf{X}, \mathbf{Y})$ given in CNF as a set of implicitly conjoined clauses $C = \{C_1, \ldots C_k\}$. Each clause potentially has some literals over system inputs $\mathbf{X}$, and some literals over system outputs $\mathbf{Y}$. Such a specification can of course be represented as a 3-level NNF circuit. For all $i \in \{1, \ldots k\}$, let $C_i|_{\mathbf{X}}$ denote the clause formed by taking the disjunction of all literals over $\mathbf{X}$ in C_i. Similarly, let $C_i|_{\mathbf{Y}}$ be the clause formed by disjoining all literals over $\mathbf{Y}$ in C_i. The set of *input clauses* of φ is then defined to be $S_{in} = \{C_1|_{\mathbf{X}}, \ldots C_k|_{\mathbf{X}}\}$. Similarly, the set of *output clauses* of φ is $S_{out} = \{C_1|_{\mathbf{Y}}, \ldots C_k|_{\mathbf{Y}}\}$. Note that if a clause has no system input (resp. system output) literal, then the corresponding clause in S_{in} (resp. S_{out}) is the empty clause, representing false.

Let S be a subset of clauses in S_{in}. We say S is a *maximal falsifiable subset* (MFS) of S_{in} if (i) there exists an assignment π that makes all clauses in S false, and (ii) for every set S' such that $S \subset S' \subseteq S_{in}$, there exists no assignment that makes all clauses in S' false. In a similar manner, $\widehat{S} \subseteq S_{out}$ is said to a *maximal saitsifable subset* (MSS) of S_{out} if (i) there exists an assignment π that makes all clause in $\widehat{S}$ true, and (ii) for every S'' such that $\widehat{S} \subset S'' \subseteq S_{out}$, it is not possible to find an assignment that renders all clauses in S'' true.

With the above notation, the following results follow from the work of [17].

Proposition 2 *(a) Let* MFS(S_{in}) *be the set of all* MFS *of* S_{in}. *Given* MFS(S_{in})*, the* BoolSkFnSyn *problem for* $\varphi(\mathbf{X}, \mathbf{Y})$ *can be solved in time linear in* $|\mathsf{MFS}(S_{in})| \cdot |\varphi(\mathbf{X}, \mathbf{Y})|$*, given access to an* NP*-oracle.*

(b) Let MSS(S_{out}) *be the set of all* MSS *of* S_{out}. *Given* MSS(S_{out})*, the* BoolSkFnSyn *problem for* $\varphi(\mathbf{X}, \mathbf{Y})$ *can be solved in time linear in* $|\mathsf{MSS}(S_{out})| \cdot |\varphi(\mathbf{X}, \mathbf{Y})|$*, given access to an* NP*-oracle.*

The intuition behind Proposition 2 can be informally stated as follows. For every assignment $\pi_{\mathbf{X}}$ of $\mathbf{X}$, consider the set of input clauses not satisfied by $\pi_{\mathbf{X}}$. By definition, this set is included in some MFS, say S', of S_{in}, and $\pi_{\mathbf{X}}$ satisfies all input clauses in $S_{in} \setminus S'$. Clearly, for each input clause in $S_{in} \setminus S'$, the corresponding clause in the specification φ is also satisfied by $\pi_{\mathbf{X}}$, regardless of what we assign to $\mathbf{Y}$. Therefore, if we assign values to $\mathbf{Y}$ such that all output clauses corresponding to input clauses in S' are satisfied, the overall specification is satisfied. This gives a way to solve BoolSkFnSyn by considering each MFS of S_{in} and by finding a satisfying assignment of the corresponding subset of S_{out}. To see how BoolSkFnSyn can be solved using MSS of S_{out}, let $\pi_{\mathbf{Y}}$ be an assignment of $\mathbf{Y}$ that satisfies an MSS, say S'', of S_{out}. Since S'' is an MSS, π_Y must falsify all clauses in $S_{out} \setminus S''$. Therefore, if the assignment of $\mathbf{X}$ satisfies all input clauses corresponding to output clauses in $S_{out} \setminus S''$, the overall specification φ is again satisfied. Thus, BoolSkFnSyn can be solved by considering satisfying assignments of every MSS of S_{out}.

In order to use Proposition 2 effectively, we must, of course, find ways to compute MFS(S_{in}) or MSS(S_{out}) efficiently in practice. Fortunately, finding an MFS of a given set of clauses, viz. S_{in}, is not hard. One way of doing this is by analyzing the *consensus graph* [28] of S_{in}. This is an undirected graph with a node for each clause in S_{in}, and an edge between two nodes iff the corresponding clauses have no literal ℓ that appear with opposite polarities in the two clauses. It is easy to see that two clauses of S_{in} can be falsified at the same time iff there is an edge between the corresponding nodes in the consensus graph. Thus, there is a one-to-one correspondence between the MFS of S_{in} and the maximal cliques in its consensus graph. The set of all MFS can therefore be enumerated by enumerating the maximal cliques in the consensus graph. Finding a maximal clique in a graph can be achieved by a greedy algorithm in time polynomial in the size of the graph. This yields an algorithm for enumerating all MFS of S_{in} that takes time polynomial in $|S_{in}|$ and in the number of maximal cliques in the consensus graph of S_{in} [33]. The following result, derived from [17], is an immediate consequence of the above observations.

Proposition 3 *[17] Let $\mathcal{C}$ be a class of* CNF *specifications such that the consensus graphs of input clauses of specifications in $\mathcal{C}$ have polynomially many maximal cliques. This is the case, for example, if the consensus graphs are planar or chordal. Then, the* BoolSkFnSyn *problem for class $\mathcal{C}$ of specifications is in* P^{NP} *(i.e. solvable in polynomial time by a Turing machine with access to an* NP *oracle).*

In practice, when implementing an algorithm for solving BoolSkFnSyn, a propositional satisfiability solver must be used in place of an NP-oracle. Given the significant advances made in propositional satisfiability solving over the last few decades, Proposition 3 allows us to identify a class of specifications for which BoolSkFnSyn can be solved efficiently in practice.

Unlike in the case of finding MFS, however, we do not know of any polynomial-time algorithm for finding an MSS of a given set of clauses. Indeed, finding an MSS requires solving an instance of the MaxSAT problem, which is known to be NP-complete. Therefore, Proposition 2(b) does not yield an easily identifiable class of specifications for which BoolSkFnSyn can be solved efficiently in practice.

Example 4 Consider the specification $\varphi \equiv (x_1 \vee y_1) \wedge (x_2 \vee \neg y_1 \vee \neg y_2) \wedge (x_2 \vee \neg x_3 \vee \neg y_2) \wedge (\neg x_1 \vee \neg y_1 \vee y_2)$. Clearly, $S_{in} = \{(x_1), (x_2), (x_2 \vee \neg x_3), (\neg x_1)\}$, and $S_{out} = \{(y_1), (\neg y_1 \vee \neg y_2), (\neg y_2), (\neg y_1 \vee y_2)\}$. The consensus graph of S_{in} is shown in Fig. 1.

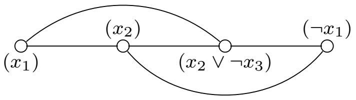

Fig. 1: Consensus graph of S_{in}

Notice that there are two maximal cliques in this graph, corresponding to two MFS of S_{in}, i.e. $\{(x_1), (x_2), (x_2 \vee \neg x_3)\}$ and $\{(x_2), (x_2 \vee \neg x_3), (\neg x_1)\}$. The corresponding subsets of S_{out} are $\{(y_1), (\neg y_1 \vee \neg y_2), (\neg y_2)\}$ and $\{(\neg y_1 \vee \neg y_2), (\neg y_2), (\neg y_1 \vee y_2)\}$, with satisfying assignments $(y_1, y_2) = (\mathsf{true}, \mathsf{false})$ and $(\mathsf{false}, \mathsf{false})$ respectively. Furthermore, the subsets of input clauses not included in the MFS are $\{(\neg x_1)\}$ and $\{(x_1)\}$ respectively. Therefore, using the idea sketched above in the intuition behind Proposition 2, we can obtain a Skolem function vector (ψ_1, ψ_2) that evaluates as follows:

$$\textit{if } (\neg x_1) \textit{ then } (\psi_1, \psi_2) = (\mathsf{true}, \mathsf{false}) \textit{ else } (\psi_1, \psi_2) = (\mathsf{false}, \mathsf{false})$$

For more details of the technique, and also to see how a Skolem function vector can be obtained from the MSS of S_{out}, the reader is referred to [17].

5 Knowledge representation for Boolean Skolem function synthesis

The representation of the relational specification $\varphi(\mathbf{X}, \mathbf{Y})$ has an important bearing on the computational complexity of solving BoolSkFnSyn. In the previous sections, we assumed that the specification is given by a NNF Boolean circuit, represented as a directed acyclic graph (DAG). It turns out that if this circuit has special structural and functional properties, BoolSkFnSyn can indeed be solved efficiently. Of course, compiling an arbitrary specification to a circuit representation with these properties isn't always easy. Given the hardness results of Section 4.2, such compilation must necessarily require super-polynomial time and space in the worst-case, unless long-standing complexity theoretic conjectures are falsified. Nevertheless, it is interesting to study normal forms of circuit-based representations of relational specifications that allow efficient synthesis of Boolean Skolem functions.

We start by considering some circuit (and related) representations of Boolean formulas that have been studied extensively in the context of hardware verification, model counting, artifical intelligence etc. Consider an NNF circuit representing a Boolean formula φ. For every node N in a DAG representation of the circuit, let $lits(N)$ (resp. $vars(N)$) denote the set of literals (resp. variables) labeling leaves that have a path from N in the DAG. Suppose for each AND-labeled node with children $c_1, \ldots c_k$ in the DAG, we have $vars(c_r) \cap vars(c_s) = \emptyset$ for all distinct $r, s \in \{1, \ldots k\}$. The circuit is then said to be in *decomposable negation normal form* or DNNF [20]. DNNF is a popular representation form used in artificial intelligence applications, and enjoys many nice properties [20]. Similarly, free/reduced ordered binary decision diagrams (collectively, BDDs) [15] is a representation form for Boolean formulas that is widely used in hardware verification, symbolic model checking etc. As shown in [20], every such BDD can be converted to DNNF in linear time [20]. In [4], a slight generalization of DNNFs, called *weak decomposable negation normal form*, or wDNNF, was introduced. In wDNNF, for each AND-labeled internal node with children $c_1, \ldots c_k$ in an NNF circuit, we have $lits(c_r) \cap \{\neg \ell \mid \ell \in lits(c_s)\} = \emptyset$ for every distinct $r, s \in \{1, \ldots k\}$. Note that every DNNF circuit is also a wDNNF circuit.

We now have the following result from [4], which says that for all the above normal forms BoolSkFnSyn is easy, i.e., solvable in polynomial time and size.

Theorem 2 ([4]) *Given an input specification $\varphi(\mathbf{X}, \mathbf{Y})$ as a* DNNF *or* wDNNF *circuit, or as a* BDD*,* BoolSkFnSyn *can be solved in time polynomial in the size of the representation. This yields a polynomial-sized Skolem function vector.*

Example 5 Consider the following Boolean formulas in NNF over the set of variables $x_1, x_2, x_3, y_1, y_2, y_3$:

$$\varphi_1 \equiv (x_1 \vee x_2) \wedge (x_3 \vee \neg y_1) \wedge (\neg y_2 \vee y_3) \quad (1)$$

$$\varphi_2 \equiv (x_1 \vee x_2) \wedge (x_2 \vee \neg y_1) \wedge (\neg y_1 \vee y_2) \quad (2)$$

$$\varphi_3 \equiv (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg y_2) \wedge (y_1 \vee y_2) \quad (3)$$

Each of these formulas is naturally represented as a 3-level NNF circuit with an AND-labeled root node having three OR-labeled children, and leaves labeled by literals as shown in Figure 2. Note that the representation of φ_1 is in DNNF, and hence also in wDNNF. However, the representation of φ_2 is not in DNNF, although it is in wDNNF. Indeed, in the circuit representing φ_2, the label $\neg y_1$ appears in a leaf reachable from two distinct children of the AND-labeled root. However, there is no literal ℓ such that a leaf labeled ℓ is reachable from one child of the AND-labeled root, and a literal labeled $\neg\ell$ is reachable from another child of the root. Hence, the requirement for wDNNF is satisfied by the representation of φ_2. Finally, the representation of φ_3 is not in wDNNF since the AND-labeled root has two distinct children such that leaves labeled y_2 and $\neg y_2$ are reachable from these children. Of course, this also means that the representation of φ_3 is not in DNNF either.

By Theorem 2, it is "easy" to synthesize Skolem functions for φ_1 and φ_2, as given in Example 5. Importantly, the above theorem only gives a sufficient, but not necessary condition for efficient Boolean Skolem function synthesis. Indeed, it turns out that even for φ_3 given in Example 5, Boolean Skolem functions can be synthesized efficiently. It is therefore interesting to ask if we can weaken the representational requirements beyond that of wDNNF, while ensuring polynomial time synthesis of Boolean Skolem functions. One easy way is to require the wDNNF condition only on literals corresponding to system outputs. This captures NNFs that are decomposable except on a set of atoms [20]. It can be seen that Theorem 2 applies in this setting as well. However, it turns out that we can go significantly beyond this, as we discuss in the next section.

5.1 A representation for efficient synthesis

Recall from the discussion in the initial part of Section 4 that if we can efficiently compute $\varphi^{(i-1)}(\mathbf{X}, \mathbf{Y}_i^n)$, i.e. $\exists y_1, \ldots y_{i-1}\, \varphi(\mathbf{X}, \mathbf{Y})$, for all $i \in \{2, \ldots n\}$, then we can solve BoolSkFnSyn efficiently. We will therefore try to arrive at a representational requirement weaker than that of wDNNF and that allows us to compute $\varphi^{(i-1)}(\mathbf{X}, \mathbf{Y}_i^n)$ for all $i \in \{2, \ldots n\}$.

Consider an NNF circuit representing the formula $\varphi(\mathbf{X}, \mathbf{Y})$. The *output-positive form* of φ, denoted $\widehat{\varphi}$, is obtained by replacing all leaves labeled $\neg y_i$ by new variables $\overline{y_i}$ in the NNF circuit representation of $\varphi(\mathbf{X}, \mathbf{Y})$. Thus, $\widehat{\varphi}$ is a formula with support $\mathbf{X} \cup \mathbf{Y} \cup \overline{\mathbf{Y}}$, where $\overline{\mathbf{Y}}$ denotes the sequence (or set, depending on the

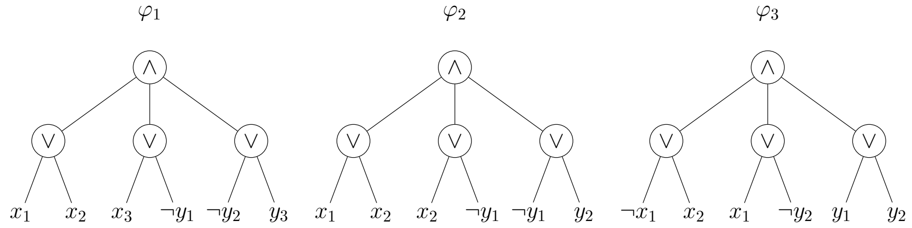

Fig. 2: NNF circuit representations of formula $\varphi_1, \varphi_2, \varphi_3$ from Example 5.

context) $(\overline{y_1}, \ldots \overline{y_n})$. It is easy to see that $\varphi(\mathbf{X},\mathbf{Y}) \Leftrightarrow (\widehat{\varphi})[\overline{\mathbf{Y}} \mapsto \neg\mathbf{Y}]$, where $\neg\mathbf{Y}$ denotes the sequence $(\neg y_1, \ldots \neg y_n)$. Since the output-positive form, represented as a NNF circuit, does not have any leaf labeled $\neg y_i$ or $\neg\overline{y_i}$ for any $i \in \{1, \ldots n\}$, it follows that $\widehat{\varphi}$ is monotone with respect to every such y_i and $\overline{y_i}$.

An immediate consequence of the above monotonicity is that we have

$$\exists y_1\, \varphi(\mathbf{X},\mathbf{Y}) \Leftrightarrow (\varphi|_{y_1} \;\vee\; \varphi|_{\neg y_1}) \Rightarrow \big(\widehat{\varphi}|_{y_1,\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n], \tag{4}$$

where we have used $\widehat{\varphi}|_{y_1,\overline{y_1}}$ to denote $(\widehat{\varphi}[y_1 \mapsto \mathsf{true}])[\overline{y_1} \mapsto \mathsf{true}]$, and $\overline{\mathbf{Y}}_2^n$ and $\neg\mathbf{Y}_2^n$ to denote the sequences $(\overline{y_2}, \ldots \overline{y_n})$ and $(\neg y_2, \ldots \neg y_n)$, respectively. In general, the converse of the above implication, i.e. $\big(\widehat{\varphi}|_{y_1,\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n] \Rightarrow (\varphi|_{y_1} \vee \varphi|_{\neg y_1})$, doesn't always hold. However, if we can ensure (for example, by imposing restrictions on the representation of φ) that the converse implication also holds, then we will have $\exists y_1\, \varphi(\mathbf{X},\mathbf{Y}) \Leftrightarrow \big(\widehat{\varphi}|_{y_1,\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$. This will immediately give us an efficient way to obtain $\exists y_1\, \varphi(\mathbf{X},\mathbf{Y})$. Specifically, we can simply set y_1 and $\overline{y_1}$ to true in $\widehat{\varphi}$, and set all other $\overline{y_i}$ to $\neg y_i$, in order to obtain $\exists y_1\, \varphi(\mathbf{X},\mathbf{Y})$. As already seen earlier, efficient existential quantification of system output variables from $\varphi(\mathbf{X},\mathbf{Y})$ directly leads to an efficient way of computing Skolem functions. Hence, it is meaningful to investigate what restrictions on the representation of φ ensure that the converse of implication (4) holds.

We start by asking: *when is implication (4) given above strict, i.e. when does its converse not hold?* Clearly, this happens iff there is an assignment π of $\mathbf{X}$ and $\mathbf{Y}_2^n$ that renders $\big(\widehat{\varphi}|_{y_1,\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ true and also simultaneously renders $\exists y_1\, \varphi(\mathbf{X},\mathbf{Y})$ false. It follows from the definitions of $\exists y_1\, \varphi(\mathbf{X},\mathbf{Y})$ and $\widehat{\varphi}(\mathbf{X},\mathbf{Y},\overline{\mathbf{Y}})$ that assignment π must cause both $\big(\widehat{\varphi}|_{y_1,\neg\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ and $\big(\widehat{\varphi}|_{\neg y_1,\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ to evaluate to false. Since φ is monotone with respect to y_1 and $\overline{y_1}$, it also follows that $\big(\widehat{\varphi}|_{\neg y_1,\neg\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ evaluates to false under assignment π. Thus, assignment π causes $\widehat{\varphi}[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ to "semantically behave like" $y_1 \wedge \overline{y_1}$.

The above discussion yields the important intuition that $\exists y_1\, \varphi(\mathbf{X},\mathbf{Y})$ is semantically equivalent to $\big(\widehat{\varphi}|_{y_1,\overline{y_1}}\big)[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ iff $\widehat{\varphi}[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ can never be made to behave like $y_1 \wedge \overline{y_1}$ under any assignment of $\mathbf{X}$ and $\mathbf{Y}_2^n$. In other words, $\forall y_1 \forall \overline{y_1}\, \big(\widehat{\varphi}[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n] \Leftrightarrow (y_1 \wedge \overline{y_1})\big)$ must be unsatisfiable. By virtue of the monotonicity properties of $\widehat{\varphi}$, the above condition simplifies to the requirement that $(\widehat{\varphi}|_{y_1,\overline{y_1}})[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n] \;\wedge\; \neg(\widehat{\varphi}|_{\neg y_1,\overline{y_1}})[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n] \;\wedge\; \neg(\widehat{\varphi}|_{y_1,\neg\overline{y_1}})[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$ is unsatisfiable. This intuition can now be inductively lifted to the general case. Towards this end, let $(\overbrace{\mathsf{true}\ldots\mathsf{true}}^{t})$ denote a sequence of t Boolean constants, each being true. For each $i \in \{1, \ldots n\}$, we now define a formula $[\widehat{\varphi}]_i$, also called the i^{th} *reduct of* φ, as follows.

$$[\widehat{\varphi}]_i \equiv \Big(\big(\widehat{\varphi}[\mathbf{Y}_1^{i-1} \mapsto (\overbrace{\mathsf{true}\ldots\mathsf{true}}^{i-1})]\big)[\overline{\mathbf{Y}}_1^{i-1} \mapsto (\overbrace{\mathsf{true}\ldots\mathsf{true}}^{i-1})]\Big)[\overline{\mathbf{Y}}_{i+1}^n \mapsto \neg\mathbf{Y}_{i+1}^n]. \tag{5}$$

Thus, we take $\widehat{\varphi}$ and set all y_j and $\overline{y_j}$ for $j \in \{1, \ldots i-1\}$ to true, and all $\overline{y_k}$ for $k \in \{i+1, \ldots n\}$ to $\neg y_k$, in order to get $[\widehat{\varphi}]_i$. The reduct $[\widehat{\varphi}]_1$ is simply defined as $\widehat{\varphi}[\overline{\mathbf{Y}}_2^n \mapsto \neg\mathbf{Y}_2^n]$. Note that the support of $[\widehat{\varphi}]_i$ includes $\overline{y_i}$ in addition to $\mathbf{X} \cup \mathbf{Y}_i^n$.

Using arguments similar to that used above, we can now show that $\exists \mathbf{Y}_1^i\, \varphi(\mathbf{X},\mathbf{Y}) \Rightarrow ([\widehat{\varphi}]_i)|_{y_i,\overline{y_i}}$. Furthermore, the converse implication holds iff $[\widehat{\varphi}]_i$ cannot be made to semantically behave like $y_i \wedge \overline{y_i}$ for any assignment of $\mathbf{X}$ and $\mathbf{Y}_{i+1}^n$, i.e. iff $([\widehat{\varphi}]_i)|_{y_i,\overline{y_i}} \;\wedge\; \neg([\widehat{\varphi}]_i)|_{\neg y_i,\overline{y_i}} \;\wedge\; \neg([\widehat{\varphi}]_i)|_{y_i,\neg\overline{y_i}}$ is unsatisfiable. Referring back to the discussion in the initial part of Section 4, it follows that if the above unsatisfiability condition holds, then both $([\widehat{\varphi}]_{i+1})|_{y_{i+1},\neg\overline{y_{i+1}}}$ and $\neg([\widehat{\varphi}]_{i+1})|_{\neg y_{i+1},\overline{y_{i+1}}}$ serve as Skolem functions for y_{i+1} (in terms of $\mathbf{X}\cup\mathbf{Y}^n_{i+2}$) in $\varphi(\mathbf{X},\mathbf{Y})$. Specifications that satisfy the above unsatisfiability condition for all reducts $[\widehat{\varphi}]_i$ are said to be in *Synthesis Negation Normal Form* or SynNNF, and the corresponding Skolem functions alluded to above are called GACKS functions, following the terminology of [4]. Note that if $\varphi(\mathbf{X},\mathbf{Y})$ is in SynNNF, then computing the GACKS functions is easy, i.e., can be done in polynomial time. Formally, we have the following definition.

Definition 1 [4] An NNF circuit representing a specification $\varphi(\mathbf{X},\mathbf{Y})$ is said to be in SynNNF with respect to the sequence $\mathbf{Y}$ of system outputs iff the formula $([\widehat{\varphi}]_i)|_{y_i,\overline{y_i}} \quad\wedge\quad \neg([\widehat{\varphi}]_i)|_{\neg y_1,\overline{y_1}} \quad\wedge\quad \neg([\widehat{\varphi}]_i)|_{y_1,\neg\overline{y_1}}$ is unsatisfiable for all $i \in \{1,\ldots n\}$

Example 6 Consider again $\varphi_3 \equiv (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg y_2) \wedge (y_1 \vee y_2)$ from Example 5, represented as the rightmost circuit in Fig. 2. We have seen that this representation is neither in wDNNF nor in DNNF. However, with respect to the sequence of system outputs (y_1, y_2), it is in SynNNF. To see this, note that $[\widehat{\varphi_3}]_1$ cannot be equivalent to $y_1 \wedge \overline{y_1}$ for any assignment of the other variables as y_1 does not occur negatively at all. Furthermore, in obtaining $[\widehat{\varphi_3}]_2$, we must assign true to y_1; hence the clause $y_1 \vee y_2$ becomes true. As a result, $[\widehat{\varphi_3}]_2$ cannot evaluate to $y_2 \wedge \overline{y_2}$ for any assignment of x_1 and x_2. Hence, we conclude that the representation of φ_3 as the rightmost circuit in Fig. 2 is in SynNNF.

Note that the definition of SynNNF makes crucial reference to a sequence (or ordering) of variables in $\mathbf{Y}$. Indeed, if we change the ordering of system output variables, say from (y_1, y_2) to (y_2, y_1) in the example of φ_3 discussed above, then φ_3 is no longer in SynNNF with respect to this new ordering. Specifically, for the assignment in which $x_1 =$ false and $y_1 =$ false, $[\widehat{\varphi_3}]_1$ becomes semantically equivalent to $y_2 \wedge \overline{y_2}$.

In [4], it is also shown that SynNNF strictly subsumes (φ_3 being an example!) previously considered normal forms including wDNNF, DNNF and BDDs. In fact, we can say more. In the following theorem, sizes and times are in terms of the number of system input and system output variables, i.e. $|\mathbf{X}| + |\mathbf{Y}|$.

Proposition 4 ([4]) *Every specification in* BDD*,* DNNF *or* wDNNF *form is either already in* SynNNF *or can be compiled in linear time to* SynNNF*. Moreover, there exist polynomial-sized* SynNNF *specifications that only admit*

(i) exponential sized BDD *representations*
(ii) super-polynomial sized wDNNF *and* DNNF *representations, unless* P = NP*.*

Finally, we come to the practical utility of SynNNF, which is formalized in the following result.

Theorem 3 ([4]) *If a relational specification $\varphi(\mathbf{X},\mathbf{Y})$ is given in* SynNNF*, the* GACKS *functions serve as polynomial sized Skolem functions for φ, and can be computed in polynomial time. Hence* BoolSkFnSyn *is solvable in polynomial time for* SynNNF *specifications.*

From Theorem 1 and Theorem 3, it follows that it is not possible to compile an arbitrary relational specifications to SynNNF in polynomial time, unless some long-standing complexity-theoretic conjectures are falsified. Such hardness results for knowledge compilation are not uncommon in Computer Science, and similar results are known for other important problems like model counting, satisfiability checking, consistency checking and the like. Nevertheless, this has motivated researchers to build compilers that work well in practice, thereby facilitating efficient solutions for important classes of problems. For example, several compilers for converting an arbitrary formula into DNNF and its variants are presented in [20, 22, 51, 43, 49]. Similarly, there are several mature tools (viz. [30, 63, 11]) that can be used to compile a propositional formula into a BDD. This approach of converting a given specification into a BDD and then generating Skolem functions is used, for instance, in [26] and also in one of the experimental pipelines reported in [5]. In [4], a compiler called C2Syn was described that converts a relational specification given in CNF directly to SynNNF. We refer the interested reader to [4] for more details of C2Syn.

To complete the discussion on SynNNF, we note that SynNNF captures a semantic requirement. This is unlike BDD, DNNF and wDNNF, all of which impose purely syntactic requirements on the structure of the representation, that can be checked in time polynomial in the size of the representation. Normal forms defined by semantic conditions are however not new, e.g., the disjoint decomposable negation normal form (dDNNF) uses a semantic condition in its definition (see [21]). The semantic condition does, however, mean that the problem of checking if a circuit is in SynNNF is not always easy.

Proposition 5 ([56]) *Checking whether a given formula is in* SynNNF *w.r.t a given ordering on the variables is* coNP*-complete. Further, checking whether it is in* SynNNF *w.r.t any ordering is in* Σ_2^{P}.

In [56], the above result was established for a more general normal form. In fact, the normal form considered in [56] not only generalizes SynNNF but also precisely characterizes polynomial time and polynomial sized Boolean Skolem function synthesis. We refer interested readers to [56] for more details regarding this form.

6 Algorithmic Paradigms for Boolean Skolem function synthesis

We have seen earlier that efficient algorithms for BoolSkFnSyn are unlikely, due to the hardness results given in Theorem 1. However, this refers to the "worst-case complexity" or efficiency for all inputs, which does not always translate to use-case hardness. Given the practical relevance of the problem, different approaches have been tried to design algorithms and build software tools that work well for real-life benchmarks. Indeed, these tools have also been shown to work well in several practical instances. In this section, we discuss in some detail one such approach, that we call the *guess-check-repair paradigm* for Boolean Skolem function synthesis. Before that, let us quickly survey other (mostly orthogonal) approaches that have been explored for algorithmic solutions to BoolSkFnSyn.

- *Proof systems and proof rules.* This approach is mostly applicable to specifications $\varphi(\mathbf{X}, \mathbf{Y})$ that are realizable, i.e. $\forall \mathbf{X} \exists \mathbf{Y} \varphi(\mathbf{X}, \mathbf{Y})$ is valid. In [47, 9, 39, 8], special proof systems for quantified Boolean formulas have been proposed, and then Skolem functions have been extracted from a proof of validity of $\forall \mathbf{X} \exists \mathbf{Y} \varphi(\mathbf{X}, \mathbf{Y})$. While this works well with short proofs of validity, there are challenges when such proofs are long or when no such proof exists, e.g. if the specification is unrealizable. As the factorization example in Section 3, it is often important and useful to synthesize Skolem functions even for unrealizable specifications.

- *Incremental determinization.* A relational specification may functionally determine some system outputs, as explained in Section 4.3.2. However, there may be other system outputs that are constrained but not completely functionally determined. In [53], a technique for incrementally determinizing such system outputs is described. The technique makes us of highly effective strategies used in modern conflict-driven clause learning (CDCL) based propositional satisfiability solvers to yield a practically efficient algorithm for Boolean Skolem function synthesis. The interested reader is referred to [58] for details of CDCL satisfiability solvers. The incremental determinization technique of [53] was further developed as a system of proof rules in [54, 52].
- *Synthesis via functional composition of circuits.* A completely different approach to BoolSkFnSyn is considered in [36, 37, 66], where iterated compositions (or substitutions) of Boolean circuits are used to synthesize Skolem functions. Given $\varphi(\mathbf{X}, \mathbf{Y})$, the basic idea here is to express one system output, say y_1, as a Skolem function in terms of other system outputs and system inputs. While techniques similar to self-substitution have been used to generate such a Skolem function in [36, 66], interpolation based techniques have been used in [37]. Once such a Skolem function is obtained, it is composed with (or substituted in) $\varphi(\mathbf{X}, \mathbf{Y})$ to effectively existentially quantify y_1 from $\varphi(\mathbf{X}, \mathbf{Y})$. This yields a simplified specification with one less system output. By repeating this process, we can eventually obtain a Skolem function for y_n in terms of only the system inputs. Subsequently, the Skolem function for y_n (in terms of only system inputs) can be substituted in the Skolem function for y_{n-1} (in terms of y_n and system inputs) to obtain a Skolem function for y_{n-1} in terms of only system inputs. By continuing this process, Skolem functions for all system outputs in terms of system inputs can be obtained. While this approach is simple to understand, it suffers from the drawback that iterated composition (or substitution) can result in an exponential blow-up in the representation of Boolean formulas. Hence, tools using this approach have been empirically found not to scale well to large benchmarks.
- ROBDD-*based techniques.* ROBDDs are widely used as compact representations of complex Boolean formulas. Researchers have therefore developed techniques for synthesizing Boolean Skolem functions from relational specifications given as ROBDDs. In [41], Kukula and Shiple presented one such technique in which a circuit that is structurally similar to the ROBDD representation of the specification is generated to implement Boolean Skolem functions. In Kuncak et al [42], a generic framework for functional synthesis with unbounded domains like integers is described. As part of their exposition, the authors of [42] also suggest using ROBDDs with *input-first ordering* of variables. This approach has been developed further in [26], where a new algorithm called *TrimSubstitute* was proposed that optimizes the application of the self-substitution technique (see Section 4) to ROBDDs with input-first variable ordering. For factored specifications, i.e, specifications that are conjunctions of sub-specifications, ideas from symbolic model checking using implicitly conjoined ROBDDs have been used to enhance the scalability of ROBDD-based synthesis further in [65]. Note that

the works of [42, 26, 65] attempt to synthesize Skolem functions directly as ROBDDs. This can be significantly more difficult than generating Skolem functions as Boolean circuits from ROBDD specifications. Indeed, we know from Proposition 4 and Theorem 3 that it is possible to generate Boolean circuits representing Skolem functions in polynomial-time from specifications given as ROBDDs. This holds regardless of the variable order used in the ROBDD representing the specification. Note, however, that the Skolem functions generated by application of Theorem 3 may not be compactly representable as ROBDDs. Interestingly, the requirement of having input-first ordering of variables when representing specifications as ROBDDs, as in the works of [42, 26, 65], may result in significantly larger ROBDDs compared to the case when there are no restrictions on the variable ordering. This may be viewed as the price that has to be paid in order to obtain the Skolem functions as ROBDDs themselves.

- *Input-output separation.* We have already discussed in Section 4.3.3 how literals in the clauses of a CNF specification can be partitioned to yield a set of input clauses and a set of output clauses. We also discussed in the same section specific conditions under which either the set of input clauses can be processed to obtain Skolem functions efficiently in practice. This idea has been developed further in [17], yielding a *back-and-forth* algorithm that alternates between processing of input clauses and output clauses to generate Skolem functions as decision lists [55]. This approach has been shown to work on some difficult classes of benchmarks, for which several other state-of-the-art techniques run out of steam.
- *Template/sketch-based techniques.* In addition to the above algorithmic techniques, template-based [64] and sketch-based [61] approaches have been developed, when we have information about the set of candidate Skolem functions. In the absence of such information, however, these techniques are not very effective.

We wish to emphasize that despite the diversity of techniques, there is no single technique that dominates others when solving BoolSkFnSyn. Furthermore, it is still largely unclear which technique would perform best for a given benchmark. This suggests the use of a portfolio solver, in which we can try multiple techniques and choose the one that best suits a given problem instance. On a related note, the knowledge representation approach presented earlier allows us to understand what input representations make the problem easy to solve, without providing an efficient technique to compile a given specification into a desired normal form. Coming up with better compilation algorithms and insights into which tool performs well on which benchmark, are part of ongoing and future work.

6.1 A guess, check and repair paradigm for synthesis

In the rest of this section, we focus on *one* specific algorithmic paradigm for solving BoolSkFnSyn, that has been developed recently in a series of papers [38, 2, 3, 5] and further augmented in [29]. Let us start by recalling that, sometimes we may get "lucky" in that the representation of the relational specification may already have structure (as explained in the previous sections) that permits efficient Boolean Skolem function synthesis. However, this raises three questions: (i) how do we get lucky? (ii) how easy is it to check if we have been lucky and (ii) what do we do when we are not lucky? Indeed, in practical applications, there is no guarantee that the representation of the relational specification has structure that makes it amenable to efficient synthesis. The *guess, check and repair* paradigm, that lies (sometimes implicitly) at the heart of several existing works on BoolSkFnSyn, address these questions very elegantly. In this section, we elucidate this generic paradigm as well as show how it is instantiated in practice. The paradigm can be broken into three key steps.

- The first step runs efficiently in practice (viz. polynomial time relative to an NP-oracle) and generates polynomial-sized guesses (or candidates) for Skolem functions. If the representation of the relational specification has desirable properties (such as those mentioned in previous sections), then these candidates are often good enough to serve as Skolem functions themselves.
- Even if the representation of the relational specification does not satisfy restrictions that guarantee correctness of the guesses made above, the guessed Skolem functions may still be correct. We must therefore check if the guessed Skolem functions can indeed serve as correct Skolem functions. As we show below, this requires a single call to an NP-oracle, practically implemented using a propositional satisfiability solver.
- Finally, if the above check results in a negative answer (i.e. not all the guessed Skolem functions are correct), we need to repair the guesses to obtain correct Skolem functions. This is the third step of the paradigm, and can be done in several ways. Given the computational hardness results, we know that in the worst case, this phase may take exponential time. However, in practice, we are often able to do much better!

The reason we call this a paradigm, rather than an algorithm, is that one can take different algorithms for solving each of the above steps and put them together to obtain an overall algorithm that solves BoolSkFnSyn. We describe each of these steps in more detail, along with some algorithms for implementing the steps, in the next three subsections.

6.1.1 Science of Guessing

It is not surprising that the initial guesses of Skolem functions play an important role in the guess-check-repair paradigm of solving BoolSkFnSyn. As mentioned earlier, if the representation of the relational specification has desirable properties (viz. being in SynNNF), then the initial guesses (viz. the GACKS functions alluded to in Section 5) already serve as correct Skolem functions without any need for further checking. Note, however, that Theorem 2 only asserts that a specification being in SynNNF is a sufficient, not necessary, condition for the GACKS functions to be correct Skolem functions. So, if GACKS functions are used as the initial guesses for Skolem function, they may work for more general specifications (that are not in SynNNF) too! This is indeed what was empirically observed in [3, 5], where GACKS functions were found to be correct Skolem functions for a large collection of benchmarks, not all of which were in SynNNF. In the works of [47, 53], coming up with good initial candidates for Skolem functions from appropriate representations of the specification (or from a proof of its realizability), has often been called *preprocessing*, or *initialization*. It turns out that this is not only a crucial step for effective Boolean Skolem function synthesis, but also has deep connections with the area of knowledge representation and compilation. Indeed, in [4], this aspect has been explored in detail, and an algorithm presented to compile a specification given in CNF to a representational form (SynNNF) where the initial guesses of Skolem functions can always be correctly made.

Another important consideration when guessing candidate Skolem functions is the kind of "errors" that are allowed in the guessed functions. For example, the work of [38, 5] requires the guessed Skolem functions to either be under-approximations or over-approximations of correct Skolem functions. Thus, the error in a candidate Skolem function is always one-sided in these approaches. While this allows for easier proofs of soundness and termination (when applied in conjunction with appropriate techniques for repair), the repair of guessed Skolem functions with one-sided error may take longer in practice. Other more recent approaches, e.g. [29], have relaxed the restriction of one-sided errors, and used machine-learning based heuristics for arriving at good initial guesses of Skolem functions, albeit with two-sided errors.

6.1.2 Checking the guess

This step involves deciding whether a guessed Skolem function vector suffices to serve as a correct Skolem function vector for the given relational specification. If the answer turns out to be in the negative, it is also useful to obtain a valuation of the system inputs $\mathbf{X}$ for which at least one of the guessed Skolem functions generates an incorrect value for the corresponding system output. It turns out that this problem can be easily reduced to checking the unsatisfiability of an appropriately constructed propositional formula, called the error formula in [38].

Given the relational specification $\varphi(\mathbf{X}, \mathbf{Y})$, suppose the vector of guessed Skolem functions for the system outputs $\mathbf{Y}$ is $\mathbf{\Psi} = (\psi_1, \ldots \psi_n)$. Following [38], the error formula for φ with respect to this guess is defined as:

$$\varepsilon_{\varphi,\mathbf{\Psi}}(\mathbf{X}, \mathbf{Y}, \mathbf{Y}') \equiv \varphi(\mathbf{X}, \mathbf{Y}') \wedge \bigwedge_{i=1}^{n} (y_i \Leftrightarrow \psi_i) \wedge \neg\varphi(\mathbf{X}, \mathbf{Y})$$

Note that the first sub-formula in $\varepsilon_{\varphi,\mathbf{\Psi}}$ has free variables from $\mathbf{Y}' = (y'_1, \ldots y'_n)$, where each y'_i is a fresh variable, not originally present in $\varphi(\mathbf{X}, \mathbf{Y})$. This sub-formula asserts that there exists some valuation of $\mathbf{Y}$ that renders $\varphi(\mathbf{X}, \mathbf{Y})$ true. This is needed in order to focus only on those assignments of $\mathbf{X}$ for which $\varphi(\mathbf{X}, \mathbf{Y})$ is satisfiable. The second sub-formula in $\varepsilon_{\varphi,\mathbf{\Psi}}$ assigns variables in $\mathbf{Y}$ to the values given by the corresponding guessed Skolem functions in $\mathbf{\Psi}$, and the third sub-formula checks if this assignment falsifies the specification φ. As proved in [38, 5], the formula $\varepsilon_{\varphi,\mathbf{\Psi}}$ is unsatisfiable iff $\mathbf{\Psi}$ is a correct Skolem function vector for the specification $\varphi(\mathbf{X}, \mathbf{Y})$.

Thus, checking if a candidate Skolem function vector suffices to serve as a correct Skolem function vector can be done using a single call to an NP-oracle. In practice, a propositional satisfiability solver is used for this purpose, and this has its own advantages. Unlike an NP-oracle that simply yields a "Yes"/"No" answer, an invokation of a propositional satisfiability solver also generates a satisfying assignment, say π, of $\varepsilon_{\varphi,\mathbf{\Psi}}(\mathbf{X}, \mathbf{Y}, \mathbf{Y}')$ in case the candidate Skolem function vector is incorrect. From the definition of $\varepsilon_{\varphi,\mathbf{\Psi}}$, it is easy to see that in such a case, the projection of π on $\mathbf{X}$ gives an assignment of system inputs for which at least one guessed Skolem function in $\mathbf{\Psi}$ generates an incorrect value for the corresponding system output. Indeed, there exists an assignment of system outputs (viz. projection of π on $\mathbf{Y}'$) that satisfies the specification φ for the above assignment of $\mathbf{X}$, and yet the values given by the guessed Skolem function vector (viz. projection of π on $\mathbf{Y}$) fail to satisfy the specification with the same assignment of $\mathbf{X}$.

6.1.3 The Art of Repairing

Finally, if the above check reports that the guessed Skolem function vector is incorrect, we need a way to repair the guess. As mentioned above, using a propositional satisfiability solver to check the satisfiability of the error formula also gives us an assignment of $\mathbf{X}, \mathbf{Y}$ and $\mathbf{Y}'$ that demonstrates why the guessed Skolem function vector $\mathbf{\Psi}$ is not correct. This information is crucial in repairing the incorrect guess. Indeed, multiple approaches have been used in the literature to repair incorrect guesses of Skolem functions.

- In [38, 3, 5], the authors use an approach called *expansion based repair*. This works when the guessed Skolem functions always have one-sided error. Intuitively, if a guessed Skolem function is an under-approximation of a correct Skolem function, the set of assignments on which it evaluates to true must be "expanded" to repair the guess. Similarly, if a guess Skolem function is an over-approximation of a correct Skolem function, the set of assignments on which it evaluates to false must be "expanded" to effect the repair. For every Skolem function in error, the repair strategy ensures that errors, if any, of the repaired Skolem function are of the same nature (i.e. under-approximation error or over-approximation error) as in the original erroneous Skolem function. Thus, the erroneous Skolem function vector monotonically approaches a correct Skolem function vector, with at least one erroneous Skolem function in the vector being changed in each iteration of repair. The actual repair is obtained by examining the satisfying assignment returned by the (un)satisfiability check of $\varepsilon_{\varphi,\mathbf{\Psi}}$ to determine which Skolem functions in $\mathbf{\Psi}$ need to be repaired. In addition, the satisfying assignment is "generalized" to obtain a set of (instead of a single) assignments of $\mathbf{X}$ for which the same expansion-based repair must be applied. This helps in reducing the number of repair iterations, since a good "generalization" may address problems that can arise with multiple valuations of $\mathbf{X}$. After each iteration of repair, the error formula is reconstructed for the repaired Skolem function vector, and its (un)satisfiability checked again. Since there are only finitely many valuations of $\mathbf{X}$ and finitely many Skolem functions to repair, it is not hard to show that expansion based repair is guaranteed to terminate with a correct Skolem function vector. However, the way in which the expansion is done crucially determines how fast and effective the repair algorithm is. The interested reader is referred to [5] for more details of expansion-based repair techniques.
- In [2], the authors use the circuit structure of the input specification to parallelize the task of repairing an incorrectly guessed Skolem function vector. While the basic approach remains one of expansion-based repair, the added benefit of parallelization shows in significantly reduced synthesis times, as demonstrated in [2].
- In a recent work [29], a new and powerful idea of repair has been used in a guess-check-repair tool for solving BoolSkFnSyn. Specifically, the authors of [29]

delve deeper into the reason why an assignment of $\mathbf{X}$ leads some candidate Skolem functions in $\boldsymbol{\Psi}$ to evaluate to the wrong values for the corresponding system outputs. Using powerful techniques based on minimal unsatisfiable core extraction, they are able to obtain significant generalizations starting from a single satisfying assignment of $\varepsilon_{\varphi,\boldsymbol{\Psi}}$. This technique has the advantage that it can repair initial guesses of Skolem functions that even have two-sided errors (i.e. the guessed Skolem function is neither an under-approximation nor an over-approximation of a correct Skolem function). As shown by an extensive set of experiments in [29], allowing two-sided errors in the initial guesses of Skolem functions chosen by means of machine learning techniques, followed by powerful unsatisfiable core based repair techniques can be very effective in synthesizing Boolean Skolem functions for a large set of benchmarks.

While we have given a high-level overview of some algorithms that implement the guess-check-repair paradigm of solving BoolSkFnSyn, there appears to be a lot of uncharted territory, and the last word on the topic of practically efficient algorithm for BoolSkFnSyn is yet to be said. Our primary focus in this article has been on the theory behind the algorithms. However, the proof of the pudding is indeed in the eating, and we strong recommend the interested reader to go through the relevant papers to see the practical performance of the ideas and algorithms sketched above.

7 Conclusion

In this article, we have explained how Skolem function synthesis lies at the heart of several lines of research. These have spanned from theoretical questions, both about existence and explicit construction of Skolem functions in the general setting of first order logic, to more practical questions about the computational hardness and efficient algorithms in simpler settings. In the simplest case of the propositional setting, we have presented a deeper insight into computational hardness issues, and also how specific properties of the representation of the specification can be exploited to design practically efficient algorithms. Finally, we have discussed a powerful paradigm, called guess-check-repair, that has been instantiated in multiple tools to obtain practically efficient strategies to solve the BoolSkFnSyn problem on a large suite of benchmarks.

Multiple lines of research emerge most naturally from the results discussed here. One immediate question is whether structural (or even functional) properties for representations of specifications can be identified for non-Boolean settings, such that they allow efficient synthesis of Skolem functions. Furthermore, can we lift the ideas and techniques for synthesis beyond Boolean specifications, to say specifications in temporal logics? Similarly, the synthesis question discussed in this article does not take into account dependency information for existentially quantified variables. Finding Skolem functions for dependency quantified Boolean formulas is an important problem, and it would be interesting to consider extensions of existing BoolSkFnSyn techniques to solve this problem. Overall, given its central importance, we hope researchers will be encouraged to pursue research on synthesis of Skolem functions for richer classes of specifications, both from theoretical and practical points of view.

References

1. Akshay S, Chakraborty S (2021) On synthesizing Skolem functions for first-order logic formulae. CoRR Identifier: 2102.07463, (https://arxiv.org/abs/2102.07463)
2. Akshay S, Chakraborty S, John AK, Shah S (2017) Towards parallel boolean functional synthesis. In: TACAS 2017 Proceedings, Part I, pp 337–353, URL `https://doi.org/10.1007/978-3-662-54577-5_19`
3. Akshay S, Chakraborty S, Goel S, Kulal S, Shah S (2018) How hard is boolean functional synthesis. In: In CAV 2018 Proceedings, URL `https://doi.org/10.1007/978-3-662-54577-5_19`
4. Akshay S, Arora J, Chakraborty S, Krishna S, Raghunathan D, Shah S (2019) Knowledge compilation for boolean functional synthesis. In: Proc. of Formal Methods in Computer Aided Design (FMCAD)
5. Akshay S, Chakraborty S, Goel S, Kulal S, Shah S (2020) Boolean functional synthesis: hardness and practical algorithms. Form Methods Syst Des DOI https://doi.org/10.1007/s10703-020-00352-2
6. Andersson G, Bjesse P, Cook B, Hanna Z (2002) A proof engine approach to solving combinational design automation problems. In: Proceedings of the 39th Annual Design Automation Conference, ACM, New York, NY, USA, DAC '02, pp 725–730, DOI 10.1145/513918.514101, URL `http://doi.acm.org/10.1145/513918.514101`
7. Arora S, Barak B (2009) Computational Complexity: A Modern Approach, 1st edn. Cambridge University Press, USA
8. Balabanov V, Jiang JHR (2012) Unified QBF certification and its applications. Form Methods Syst Des 41(1):45–65, DOI 10.1007/s10703-012-0152-6, URL `http://dx.doi.org/10.1007/s10703-012-0152-6`
9. Benedetti M (2005) sKizzo: A Suite to Evaluate and Certify QBFs. In: Proc. of CADE, Springer-Verlag, pp 369–376
10. Beth E (1953) On Padoa's method in the theory of definition. Indagationes Mathematicae (Proceedings) 56:330–339, DOI https://doi.org/10.1016/S1385-7258(53)50042-3, URL `https://www.sciencedirect.com/science/article/pii/S1385725853500423`
11. Biere A (1998) ABCD. `http://fmv.jku.at/abcd/`
12. Bockmayr A (1993) Logic Programming with Pseudo-Boolean Constraints, MIT Press, Cambridge, MA, USA, pp 327–350
13. Boole G (1847) The Mathematical Analysis of Logic. Philosophical Library, URL `https://books.google.co.in/books?id=zv4YAQAAIAAJ`
14. Brenguier R, Pérez GA, Raskin JF, Sankur O (2014) Abssynthe: abstract synthesis from succinct safety specifications. In: Proceedings 3rd Workshop on Synthesis (SYNT'14), Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol 157, pp 100–116, DOI 10.4204/EPTCS.157.11, URL `http://arxiv.org/abs/1407.5961v1`
15. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691, DOI 10.1109/TC.1986.1676819, URL `http://dx.doi.org/10.1109/TC.1986.1676819`
16. Buttner W, Simonis H (1987) Embedding boolean expressions into logic programming. Journal of Symbolic Computation 4(2):191–205

17. Chakraborty S, Fried D, Tabajara LM, Vardi MY (2018) Functional synthesis via input-output separation. In: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, pp 1–9
18. Chandrasekaran V, Srebro N, Harsha P (2008) Complexity of inference in graphical models. In: UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008, pp 70–78
19. Dao TBH, Djelloul K (2006) Solving first-order constraints in the theory of the evaluated trees. In: Proceedings of the Constraint Solving and Contraint Logic Programming 11th Annual ERCIM International Conference on Recent Advances in Constraints, Springer-Verlag, Berlin, Heidelberg, CSCLP'06, p 108–123
20. Darwiche A (2001) Decomposable negation normal form. J ACM 48(4):608–647
21. Darwiche A (2001) On the tractable counting of theory models and its application to truth maintenance and belief revision. Journal of Applied Non-Classical Logics 11(1-2):11–34
22. Darwiche A (2002) A compiler for deterministic, decomposable negation normal form. In: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI), AAAI Press, Menlo Park, California, pp 627–634
23. Davis M, Matijasevic Y, Robinson J (1976) Hilbert's tenth problem. diophantine equations: positive aspects of a negative solution. In: Proceedings of symposia in pure mathematics, vol 28, pp 323–378
24. De Micheli G (1994) Synthesis and Optimization of Digital Circuits, 1st edn. McGraw-Hill Science/Engineering/Math, USA
25. Finkbeiner B (2016) Synthesis of reactive systems. In: Esparza J, Grumberg O, Sickert S (eds) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol 45, IOS Press, pp 72–98, DOI 10.3233/978-1-61499-627-9-72, URL `https://doi.org/10.3233/978-1-61499-627-9-72`
26. Fried D, Tabajara LM, Vardi MY (2016) BDD-based boolean functional synthesis. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, pp 402–421
27. Fu Z, Malik S (2007) Extracting logic circuit structure from conjunctive normal form descriptions. In: 20th International Conference on VLSI Design (VLSI Design 2007), Sixth International Conference on Embedded Systems (ICES 2007), 6-10 January 2007, Bangalore, India, IEEE Computer Society, pp 37–42
28. Ganian R, Szeider S (2017) New width parameters for model counting. In: Theory and Applications of Satisfiability Testing – SAT 2017, Springer International Publishing, pp 38–52
29. Golia P, Roy S, Meel KS (2020) Manthan: A data-driven approach for boolean function synthesis. In: Proceedings of International Conference on Computer-Aided Verification (CAV)
30. Group BLV (2008) ABC: A system for sequential synthesis and verification
31. Hopcroft JE, Motwani R, Ullman JD (2006) Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., USA

32. Huth M, Ryan M (2004) Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, USA
33. Ignatiev A, Morgado A, Planes J, Marques-Silva J (2013) Maximal falsifiability. In: Logic for Programming, Artificial Intelligence, and Reasoning, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 439–456
34. Impagliazzo R, Paturi R (2001) On the complexity of k-SAT. J Comput Syst Sci 62(2):367–375
35. Jacobs S, Bloem R, Brenguier R, Könighofer R, Pérez GA, Raskin J, Ryzhyk L, Sankur O, Seidl M, Tentrup L, Walker A (2015) The second reactive synthesis competition (SYNTCOMP 2015). In: Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015., pp 27–57
36. Jiang JHR (2009) Quantifier elimination via functional composition. In: Proc. of CAV, Springer, pp 383–397
37. Jiang JR, Lin H, Hung W (2009) Interpolating functions from large boolean relations. In: 2009 International Conference on Computer-Aided Design, ICCAD 2009, San Jose, CA, USA, November 2-5, 2009, pp 779–784
38. John A, Shah S, Chakraborty S, Trivedi A, Akshay S (2015) Skolem functions for factored formulas. In: FMCAD, pp 73–80
39. Jussila T, Biere A, Sinz C, Kröning D, Wintersteiger C (2007) A First Step Towards a Unified Proof Checker for QBF. In: Proc. of SAT, LNCS, vol 4501, Springer, pp 201–214
40. Kuehlmann A, Paruthi V, Krohm F, Ganai MK (2002) Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans on CAD of Integrated Circuits and Systems 21(12):1377–1394, URL `http://dblp.uni-trier.de/db/journals/tcad/tcad21.html#KuehlmannPKG02`
41. Kukula JH, Shiple TR (2000) Building circuits from relations. In: Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, pp 113–123
42. Kuncak V, Mayer M, Piskac R, Suter P (2010) Complete functional synthesis. SIGPLAN Not 45(6):316–329
43. Lagniez JM, Marquis P (2017) An improved decision-DNNF compiler. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), pp 667–673
44. Löwenheim L (1910) Über die Auflösung von Gleichungen in Logischen Gebietkalkul. Math Ann 68:169–207
45. Macii E, Odasso G, Poncino M (2006) Comparing different boolean unification algorithms. In: Proc. of 32nd Asilomar Conference on Signals, Systems and Computers, pp 17–29
46. Madsen M, van de Pol J (2020) Polymorphic types and effects with boolean unification. Proceedings of the ACM on Programming Languages 4(OOPSLA)
47. Marijn Heule MS, Biere A (2014) Efficient Extraction of Skolem Functions from QRAT Proofs. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014, pp 107–114
48. Martin U, Nipkow T (1989) Boolean unification - the story so far. J Symb Comput 7(3-4):275–293, DOI 10.1016/S0747-7171(89)80013-6, URL `http://dx.doi.org/10.1016/S0747-7171(89)80013-6`
49. Muise C, McIlraith SA, Beck JC, Hsu E (2012) DSHARP: Fast d-DNNF Compilation with sharpSAT. In: Canadian Conference on Artificial Intelligence

50. Niemetz A, Preiner M, Lonsing F, Seidl M, Biere A (2012) Resolution-based certificate extraction for QBF - (tool presentation). In: Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, pp 430–435
51. Oztok U, Darwiche A (2015) A top-down compiler for sentential decision diagrams. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), pp 3141–3148
52. Rabe MN (2019) Incremental determinization for quantifier elimination and functional synthesis. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II, pp 84–94
53. Rabe MN, Seshia SA (2016) Incremental determinization. In: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings, pp 375–392, DOI 10.1007/978-3-319-40970-2_23, URL `https://doi.org/10.1007/978-3-319-40970-2_23`
54. Rabe MN, Tentrup L, Rasmussen C, Seshia SA (2018) Understanding and extending incremental determinization for 2QBF. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II, pp 256–274
55. Rivest R (1987) Learning decision lists. Machine Learning 2(3):229–246
56. Shah P, Bansal A, Akshay S, Chakraborty S (2021) A normal form characterization for efficient boolean skolem function synthesis. CoRR abs/2104.14098, URL `https://arxiv.org/abs/2104.14098`, `2104.14098`
57. Shukla A, Bierre A, Siedl M, Pulina L (2019) A survey on applications of quantified boolean formula. In: Proceedings of the Thirty-First International Conference on Tools with Artificial Intelligence (ICTAI), pp 78–84
58. Silva JPM, Lynce I, Malik S (2021) Conflict-driven clause learning sat solvers. In: Biere A, Heule M, van Maaren H, Walsch T (eds) Handbook of Satisfiability, IOS Press, chap 4, pp 131–153
59. Simonis H, Dincbas M (1987) Using an extended Prolog for digital circuit design. In: IEEE International Workshop on AI Applications to CAD Systems for Electronics, Springer International Publishing, pp 165–188
60. Slivovsky F (2020) Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I, Springer, Lecture Notes in Computer Science, vol 12224, pp 508–528
61. Solar-Lezama A, Rabbah RM, Bodík R, Ebcioglu K (2005) Programming by sketching for bit-streaming programs. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, pp 281–294
62. Somenzi F (1999) Binary decision diagrams. In: Calculational System Design, vol. 173 of NATO Science Series F, IOS Press, pp 303–366
63. Somenzi F (2008) CUDD: CU decision diagram package release 2.5.0. `http://vlsi.colorado.edu/~fabio/CUDD/`
64. Srivastava S, Gulwani S, Foster JS (2013) Template-based program verification and program synthesis. STTT 15(5-6):497–518

65. Tabajara LM, Vardi MY (2017) Factored boolean functional synthesis. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017, pp 124–131
66. Trivedi A (2003) Techniques in symbolic model checking. Master's thesis, Indian Institute of Technology Bombay, Mumbai, India
67. Tseitin GS (1968) On the complexity of derivation in propositional calculus. Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics pp 115–125