Knowledge Compilation for Boolean Functional Synthesis

S. Akshay, Jatin Arora, Supratik Chakraborty, S. Krishna, Divya Raghunathan and Shetal Shah
Indian Institute of Technology Bombay, Mumbai, India

***Abstract*—Given a Boolean formula $F(\mathbf{X}, \mathbf{Y})$, where $\mathbf{X}$ is a vector of outputs and $\mathbf{Y}$ is a vector of inputs, the Boolean functional synthesis problem requires us to compute a Skolem function vector $\Psi(\mathbf{Y})$ such that $F(\Psi(\mathbf{Y}), \mathbf{Y})$ holds whenever $\exists \mathbf{X}\, F(\mathbf{X}, \mathbf{Y})$ holds. In this paper, we investigate the relation between the representation of the specification $F(\mathbf{X}, \mathbf{Y})$ and the complexity of synthesis. We introduce a new normal form for Boolean formulas, called SynNNF, that guarantees polynomial-time synthesis and also polynomial-time existential quantification for some order of quantification of variables. We show that several normal forms studied in the knowledge compilation literature are subsumed by SynNNF, although SynNNF can be super-polynomially more succinct than them. Motivated by these results, we propose an algorithm to convert a specification in CNF to SynNNF, with the intent of solving the Boolean functional synthesis problem. Experiments with a prototype implementation show that this approach solves several benchmarks beyond the reach of state-of-the-art tools.**

I. Introduction

Boolean functional synthesis is the problem of synthesizing outputs as Boolean functions of inputs, while satisfying a declarative relational specification between inputs and outputs. Also called *Skolem function synthesis*, this problem has numerous applications including certified QBF solving, reactive control synthesis, circuit and program repair and the like. While variants of the problem have been studied since long [16], [4], there has been significant recent interest in designing practically efficient algorithms for Boolean functional synthesis. The resulting breed of algorithms [13], [21], [20], [9], [23], [10], [12], [3], [2], [14], [6], [22] have been empirically shown to work well on large collections of benchmarks. Nevertheless, there are not-so-large examples that are currently not solvable within reasonable resources by any known algorithm. To make matters worse, it is not even fully understood what properties of a Boolean relational specification or of its representation make it amenable to efficient synthesis. In this paper, we take a step towards answering this question. Specifically, we propose a new sub-class of negation normal form called SynNNF, such that every Boolean relational specification in SynNNF admits polynomial-time synthesis. Furthermore, a Boolean relational specification admits polynomial-time synthesis (by any algorithm) *if and only if* there exists a polynomial-sized *refinement* of the specification in SynNNF.

To illustrate the hardness of Boolean functional synthesis, consider the specification $F(\mathbf{X}_1, \mathbf{X}_2, \mathbf{Y}) \equiv (\mathbf{Y} = (\mathbf{X}_1 \times_{[n]} \mathbf{X}_2)) \wedge (\mathbf{X}_1 \neq 0 \cdots 01) \wedge (\mathbf{X}_2 \neq 0 \cdots 01)$, where $|\mathbf{Y}| = 2n$, $|\mathbf{X}_1| = |\mathbf{X}_2| = n$ and $\times_{[n]}$ denotes multiplication of n-bit unsigned integers. This specification asserts that $\mathbf{Y}$, viewed as a $2n$-bit unsigned integer, is the product of $\mathbf{X}_1$ and $\mathbf{X}_2$, each viewed as an n-bit unsigned integer different from 1. The specification $F(\mathbf{X}_1, \mathbf{X}_2, \mathbf{Y})$ can be easily represented as a circuit of AND, OR, NOT gates with $\mathcal{O}(n^2)$ gates. However, synthesizing $\mathbf{X}_1$ and $\mathbf{X}_2$ as functions of $\mathbf{Y}$ requires us to obtain a circuit that factorizes a $2n$-bit unsigned integer into factors different from 1, whenever possible. It is a long-standing open question whether such a circuit of size polynomial in n exists. Thus, although the relational specification is succinctly representable, the outputs expressed as functions of the inputs may not have any known succinct representation.

It was recently shown [2] that unless some long-standing complexity theoretic conjectures are falsified, Boolean functional synthesis must necessarily require super-polynomial (or even exponential) space and time. In the same work [2], it was also shown that if a specification is represented in *weak decomposable negation normal form* wDNNF, synthesis can be accomplished in time polynomial in the size of the specification. While this was a first step towards identifying a normal form with the explicit objective of polynomial-time synthesis, experimental results in [2] indicate that wDNNF doesn't really characterize specifications that admit efficient synthesis. Specifically, experiments in [2] showed that a polynomial-time algorithm intended for synthesis from wDNNF specifications ends up solving the synthesis problem for a large class of specifications *not in* wDNNF. This motivates us to ask if there exists a weaker (than wDNNF) sub-class of Boolean relational specifications that admit polynomial-time synthesis.

We answer the above question affirmatively in this paper, the polynomial dependence being quadratic in the number of outputs and the size of the specification. En route, we also show that the weaker normal form, viz. SynNNF, admits polynomial-time existential quantifier elimination of a set of variables for *some (not all)* order of quantification of variables. Applications of such quantifier elimination abound in practice, viz. image computation in symbolic model checking, synthesis of QBF certificates, computation of interpolants etc. Note that ensuring efficient quantifier elimination *for some ordering* of variables is simpler than ensuring efficient quantifier elimination *for all orderings* of variables – the latter having been addressed by normal forms like DNNF [7].

Our primary contributions can be summarized as follows:

- We present a new sub-class of negation normal form, called SynNNF, that admits polynomial-time synthesis and quantifier elimination for a set of variables.

- We show that SynNNF is super-polynomially (in some cases, exponentially) more succinct than several other sub-classes studied in the literature (viz. wDNNF, dDNNF, DNNF, FBDD, ROBDD), unless some long-standing complexity theoretic conjectures are falsified.
- We show that by suitably weakening SynNNF, we can precisely characterize the class of Boolean specifications that admit polynomial-time synthesis by a simple algorithm originally proposed in [2].
- We define a natural notion of *refinement of specifications w.r.t synthesis* and show that every specification that admits polynomial-time synthesis necessarily has a polynomial-sized refinement that is in SynNNF.
- We present a novel algorithm for compiling a Boolean relational specification in CNF to a *refined* specification in SynNNF. We call this *knowledge compilation for synthesis and quantifier elimination*.
- Finally, we present experimental results that show that synthesis by compiling to SynNNF solves a large set of benchmarks, including several benchmarks beyond the reach of existing tools.

Related Work: The literature on knowledge compilation of Boolean functions is rich and extensive [5], [7], [18], [8]. While existential quantification or *forgetting* of propositions has been studied in [15], [8], neither Boolean functional synthesis nor existential quantification for *some (not all)* ordering of variables has received attention in earlier work on knowledge compilation. Sub-classes of negation normal forms like DNNF and other variants [8] admit efficient existential quantification *for all* orders in which variables are quantified. However, if we are interested in only the result of existentially quantifying a given set of variables, these forms can be unnecessarily restrictive and exponentially larger. Recent work on Boolean functional synthesis [12], [13], [10], [22], [9], [3], [2], [6] has focused more on algorithms to directly synthesize outputs as functions of inputs. Some of these algorithms (viz. [9], [2], [6]) exploit properties of specific input representations for optimizing the synthesis process. This has led to the articulation of *sufficient* conditions on representation of specifications for efficient synthesis. For example, [14] suggested using input-first ROBDDs for efficient synthesis, and a quadratic-time algorithm for synthesis from input-first ROBDDs was presented in [9]. This result was subsequently generalized in [2], where it was shown that specifications in wDNNF (which strictly subsumes ROBDDs) suffice to give a quadratic-time algorithm for synthesis. As we show later, wDNNF can itself be generalized to SynNNF. In another line of investigation, it was shown [6] that if a CNF specification is decomposed into an *input-part* and an *output-part*, then synthesis can be achieved in time linear in the size of the CNF specification and k, where k is the smaller of the count of *maximal falsifiable subsets (MFS)* of the input-part and the count of *maximal satisfiable subsets (MSS)* of the output-part. However, this does not yield an algorithm whose running time is polynomial in the size of the representation of $F(\mathbf{X}, \mathbf{Y})$.

II. Preliminaries and notations

A Boolean formula $F(z_1, \dots z_p)$ on p variables is a mapping $F : \{0,1\}^p \to \{0,1\}$. The set of variables $\{z_1, \dots z_p\}$ is called the *support* of the formula, and denoted $\mathsf{sup}(F)$. We normally use $\mathbf{Z}$ to denote the sequence $(z_1, \dots z_p)$. For notational convenience, we will also use $\mathbf{Z}$ to denote a set of variables, when there is no confusion. A *satisfying assignment* or *model* of F is a mapping of variables in $\mathsf{sup}(F)$ to $\{0,1\}$ such that F evaluates to 1 under this assignment. If π is a model of F, we write $\pi \models F$ and use $\pi(z_i)$ to denote the value assigned to $z_i \in \mathsf{sup}(F)$ by π. If $\mathbf{Z}'$ is a subsequence of $\mathbf{Z}$, we use $\pi{\downarrow}\mathbf{Z}'$ to denote the projection of π on $\mathbf{Z}'$, i.e. $(\pi(z'_1), \dots \pi(z'_k))$, where $k = |\mathbf{Z}'|$. We use $\mathsf{form}(\pi{\downarrow}\mathbf{Z}')$ to denote the conjunction of *literals* (i.e. variables or their negation) corresponding to $\pi{\downarrow}\mathbf{Z}'$. For example, if π assigns 1 to z_1, z_3 and 0 to z_2, z_4 and $\mathbf{Z}' = (z_1, z_4)$, then $\mathsf{form}(\pi{\downarrow}\mathbf{Z}') = z_1 \wedge \neg z_4$.

1) Negation normal form (NNF): This is the class of Boolean formulas in which (i) the only operators used are conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$), and (ii) negation is applied only to variables. Every Boolean formula can be converted to a semantically equivalent NNF formula. Moreover, this conversion can be done in linear time for representations like AIGs, ROBDDs, Boolean circuits etc.

2) Unate formulas: Let $F|_{z_i=0}$ (resp. $F|_{z_i=1}$) denote the positive (resp. negative) *cofactor* of F with respect to z_i. Then, F is *positive unate* in $z_i \in \mathsf{sup}(F)$ iff $F|_{z_i=0} \Rightarrow F|_{z_i=1}$. Similarly, F is *negative unate* in z_i iff $F|_{z_i=1} \Rightarrow F|_{z_i=0}$. A *literal* ℓ is said to be *pure* in an NNF formula F iff F has at least one instance of ℓ but no instance of $\neg\ell$. If z_i (resp. $\neg z_i$) is pure in F, then F is positive (resp. negative) unate in z_i.

3) Independent support and functionally defined variables: A subsequence $\mathbf{Z}'$ of $\mathbf{Z}$ is said to be an *independent support* of F iff every pair of satisfying assignments π, π' of F that agree on the assignment of variables in $\mathbf{Z}'$ also agree on the assignment of all variables in $\mathbf{Z}$. Variables not in $\mathbf{Z}'$ are said to be *functionally defined* by the independent support. Effectively, the assignment of variables in $\mathbf{Z}'$ uniquely determine that of functionally defined variables, when satisfying F. CNF encodings of Boolean functions originally specified as circuits, ROBDDs, AIGs etc. often use Tseitin encoding [24], which introduces a large number of functionally defined variables.

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

For $1 \le i \le j \le n$, we use $\mathbf{X}_i^j$ to denote the subsequence $(x_i, x_{i+1}, \dots x_j)$. If $i \le k < j$, we sometimes use $(\mathbf{X}_i^k, \mathbf{X}_{k+1}^j)$ interchangeably with $\mathbf{X}_i^j$ for notational convenience. Let

$F^{(i-1)}(\mathbf{X}_i^n, \mathbf{Y})$ denote $\exists \mathbf{X}_1^{i-1} F(\mathbf{X}_1^{i-1}, \mathbf{X}_i^n, \mathbf{Y})$. It has been argued in [13], [9], [3], [11] that the BFnS problem for $F(\mathbf{X}, \mathbf{Y})$ can be solved by first ordering the outputs, say as $x_1 \prec x_2 \cdots \prec x_n$, and then synthesizing a function $\psi_i(\mathbf{X}_{i+1}^n, \mathbf{Y}) \equiv F^{(i-1)}(\mathbf{X}_i^n, \mathbf{Y})[x_i \mapsto 1]$ for each x_i. This ensures that $F^{(i-1)}(\psi_i, \mathbf{X}_{i+1}^n, \mathbf{Y}) \Leftrightarrow \exists x_i F^{(i-1)}(x_i, \mathbf{X}_{i+1}^n, \mathbf{Y})$. Once all such ψ_i's are obtained, one can substitute ψ_{i+1} through ψ_n for x_{i+1} through x_n respectively, in ψ_i to obtain a Skolem function for x_i as a function of $\mathbf{Y}$. The primary problem of using this approach as-is is the exponential blow-up incurred in the size of the Skolem functions.

5) DAG representations: For an NNF formula F, its DAG representation is naturally induced by the structure of F. Specifically, if F is simply a literal ℓ, its DAG representation is a leaf labeled ℓ. If F is F_1 op F_2 where op $\in \{\vee, \wedge\}$, its DAG representation is a node labeled op with two children, viz. the DAG representations of F_1 and F_2. W.l.o.g. we assume that a DAG representation of F is always in a *simplified* form, where $t \wedge 1$, $t \vee 0$, $t \wedge t$ and $t \vee t$ are replaced by t, $t \wedge 0$ is replaced by 0 and $t \vee 1$ is replaced by 1 for every node t. We use $|F|$ for the node count in the DAG representation of F.

FBDD and ROBDD are well-known representations of Boolean formulas and we skip their definitions. We briefly recall the definitions of DNNF, dDNNF and wDNNF below. Let α be the subformula represented by an internal node N (labeled by $\wedge$ or $\vee$) in a DAG representation of an NNF formula F. We use $lits(\alpha)$ to denote the set of literals labeling leaves that have a path to the node N representing α in the DAG representation of F. We also use $atoms(\alpha)$ to denote the underlying set of variables in sup(F) that appear in $lits(\alpha)$. For each $\wedge$-labeled internal node N in the DAG of F with $\alpha = \alpha_1 \wedge \ldots \wedge \alpha_k$ being the subformula represented by N, if for all distinct indices $r, s \in \{1, \ldots k\}$, $atoms(\alpha_r) \cap atoms(\alpha_s) = \emptyset$, then F is said to be in DNNF [7]. If, instead, for all distinct indices $r, s \in \{1, \ldots k\}$, $lits(\alpha_r) \cap \{\neg \ell \mid \ell \in lits(\alpha_s)\} = \emptyset$, then F is said to be in wDNNF [2]. Finally $F(\mathbf{X}, \mathbf{Y})$ is said to be in deterministic DNNF(or dDNNF) [8] if F is in DNNF and for each $\vee$-labeled internal node D in the DAG of F with $\beta = \beta_1 \vee \ldots \vee \beta_k$ being the subformula represented by D, $\beta_r \wedge \beta_s$ is a contradiction for all distinct indices r, s.

6) Positive form of input specification: Given a specification $F(\mathbf{X}, \mathbf{Y})$ in NNF, we denote by $\widehat{F}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y})$ the formula obtained by replacing every occurrence of $\neg x_i$ ($x_i \in \mathbf{X}$) in F with a fresh variable $\overline{x_i}$. This is also called the *positive form* of the specification and has been used earlier in [3]. Observe that for any F in NNF, $\widehat{F}$ is positive unate (or *monotone*) in all variables in $\mathbf{X}$ and $\overline{\mathbf{X}}$. For $i \in \{1, \ldots n\}$, we sometimes split $\mathbf{X}$ into two parts, $\mathbf{X}_1^i$ and $\mathbf{X}_{i+1}^n$, and represent $\widehat{F}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y})$ as $\widehat{F}(\mathbf{X}_1^i, \mathbf{X}_{i+1}^n, \overline{\mathbf{X}}_1^i, \overline{\mathbf{X}}_{i+1}^n, \mathbf{Y})$. For $b, c \in \{0, 1\}$, let $\mathbf{b}^i$ (resp. $\mathbf{c}^i$) denote a vector of i b's (resp. c's). For notational convenience, we use $\widehat{F}(\mathbf{b}^i, \mathbf{X}_{i+1}^n, \mathbf{c}^i, \overline{\mathbf{X}}_{i+1}^n, \mathbf{Y})$ to denote $\widehat{F}(\mathbf{X}_1^i, \mathbf{X}_{i+1}^n, \overline{\mathbf{X}}_1^i, \overline{\mathbf{X}}_{i+1}^n, \mathbf{Y})|_{\mathbf{X}_1^i = \mathbf{b}^i, \overline{\mathbf{X}}_1^i = \mathbf{c}^i}$.

III. A New Normal Form for Efficient Synthesis

In [2], it was shown that if $F(\mathbf{X}, \mathbf{Y})$ is represented as a ROBDD/FBDD or in DNNF or in wDNNF form, Skolem functions can be synthesized in time polynomial in $|F|$. In this section, we define a new normal form called SynNNF that subsumes and is more succinct than these other normal forms, and yet guarantees efficient synthesis of Skolem functions.

Definition 1. *Given a specification $F(\mathbf{X}, \mathbf{Y})$, for every $i \in \{1, \ldots n\}$ we define the i^{th}-*reduct of $\widehat{F}$, *denoted $[\widehat{F}]_i$, to be $\widehat{F}(1^{i-1}, \mathbf{X}_i^n, 1^{i-1}, \overline{\mathbf{X}}_i^n, \mathbf{Y})$. We also define $[\widehat{F}]_{n+1}$ to be $\widehat{F}(1^n, 1^n, \mathbf{Y})$.*

Note that $[\widehat{F}]_1$ is the same as $\widehat{F}$, and sup$([\widehat{F}]_i) = \mathbf{X}_i^n \cup \overline{\mathbf{X}}_i^n \cup \mathbf{Y}$ for $i \in \{1, \ldots n\}$.

Example 1. *Consider the* NNF *formula $K(x_1, x_2, y_1, y_2) = (x_1 \vee x_2) \wedge (\neg x_2 \vee y_1) \wedge (\neg y_1 \vee y_2)$. Then $\widehat{K} = ((x_1 \vee x_2) \wedge (\overline{x_2} \vee y_1) \wedge (\neg y_1 \vee y_2))$. Thus, we have $[\widehat{K}]_1 = \widehat{K}$ and $[\widehat{K}]_2 = \widehat{K}[x_1 \mapsto 1, \overline{x_1} \mapsto 1] = (\overline{x_2} \vee y_1) \wedge (\neg y_1 \vee y_2)$.*

Next, we define a useful property for the i^{th}-reduct, which will be crucial for efficient synthesis of Skolem functions.

Definition 2. *Given $F(\mathbf{X}, \mathbf{Y})$, let α_i^{jk} denote $[\widehat{F}]_i[x_i \mapsto j, \overline{x}_i \mapsto k, \overline{\mathbf{X}}_{i+1}^n \mapsto \neg \mathbf{X}_{i+1}^n]$, where $j, k \in \{0, 1\}$. We say that $[\widehat{F}]_i$ is $\wedge_i$-unrealizable if $\zeta = \alpha_i^{11} \wedge \neg \alpha_i^{10} \wedge \neg \alpha_i^{01}$ is unsatisfiable.*

Intuitively, we wish to say that there is no assignment to $\mathbf{X}_{i+1}^n$ and $\mathbf{Y}$ such that $[\widehat{F}]_i$ is equivalent to $x_i \wedge \overline{x}_i$. The formula ζ captures this semantic condition. Indeed, if an assignment makes ζ true, then it also makes $[\widehat{F}]_i$ equivalent to $x_i \wedge \overline{x}_i$ (i.e., $[\widehat{F}]_i = 1$ for $x_i, \overline{x_i}$ having values $(1, 1)$, but not for $(0, 1)$, $(1, 0)$, $(0, 0)$). Note that since $[\widehat{F}]_i$ is positive unate in x_i and $\overline{x_i}$, ζ is satisfiable iff $\zeta \wedge \neg \alpha_i^{00}$ is satisfiable; we need not conjoin $\neg \alpha_i^{00}$ in the definition of ζ. A sufficient condition for $[\widehat{F}]_i$ to be $\wedge_i$-unrealizable is that in the DAG representation of $[\widehat{F}]_i$, there is no pair of paths – one from x_i and the other from $\overline{x_i}$ – which meet for the first time at an $\wedge$-labeled node. In Example 1, $[\widehat{K}]_1$ is $\wedge_1$-unrealizable since there is no leaf labeled $\overline{x_1}$ in its DAG representation. Similarly, $[\widehat{K}]_2 = (\overline{x_2} \vee y_1) \wedge (\neg y_1 \vee y_2)$ is $\wedge_2$-unrealizable as there is no leaf labeled x_2 in the DAG representation of $[\widehat{K}]_2$ (although such a leaf exists in the DAG representation of $[\widehat{K}]_1$).

Example 2. *Let $H(x_1, x_2, y_1, y_2) = (x_1 \vee x_2 \vee y_1) \wedge (\neg x_1 \vee (\neg x_2 \wedge y_2))$. Then $\widehat{H}(\mathbf{X}, \overline{\mathbf{X}}, \mathbf{Y}) = (x_1 \vee x_2 \vee y_1) \wedge (\overline{x_1} \vee (\overline{x_2} \wedge y_2))$. Using the notation in Definition 2, $\alpha_1^{11} = 1$, $\alpha_1^{10} = \neg x_2 \wedge y_2$ and $\alpha_1^{01} = (x_2 \vee y_1)$. There is an assignment $(x_2 = 0, y_2 = 0, y_1 = 0)$ such that $(\alpha_1^{11} \wedge \neg \alpha_1^{10} \wedge \neg \alpha_1^{01})$ is satisfiable. Hence $[\widehat{H}]_1$ is not $\wedge_1$-unrealizable (equivalently, it is $\wedge_1$-realizable). However, $[\widehat{H}]_2 = \widehat{H}[x_1 \mapsto 1, \overline{x_1} \mapsto 1] = 1$; hence it is vacuously $\wedge_2$-unrealizable.*

Definition 3. *A formula $F(\mathbf{X}, \mathbf{Y})$ is said to be in synthesizable* NNF *(or SynNNF) wrt the sequence $\mathbf{X}$ if F is in* NNF, *and for all $1 \leq i \leq n$, $[\widehat{F}]_i$ is $\wedge_i$-unrealizable.*

In Examples 1, 2, K is in SynNNF, while H is not. Also neither of them are in DNNF or wDNNF. Additionally, the functions as presented do not correspond to ROBDD/FBDD representations either. We now show *three* important properties

of SynNNF which motivate our proposal of SynNNF as a normal form for synthesis and existential quantification.

1) SynNNF leads to efficient quantification and synthesis: Our first result is that existentially quantifying $\mathbf{X}$ and synthesizing $\mathbf{X}$ are easy for SynNNF.

Theorem 1. *Suppose $F(\mathbf{X},\mathbf{Y})$ is in SynNNF. Then,*

(i) $\exists \mathbf{X}_1^i F(\mathbf{X},\mathbf{Y}) \Leftrightarrow [\widehat{F}]_{i+1}[\overline{\mathbf{X}}_{i+1}^n \mapsto \neg\mathbf{X}_{i+1}^n]$ *for* $i \in \{1,\ldots,n\}$,

(ii) Skolem function vector Ψ_1^n for $\mathbf{X}_1^n$ can be computed in $\mathcal{O}(n^2\cdot|F|)$ time and $\mathcal{O}(n\cdot|F|)$ space, where $|\mathbf{X}|=n$.

Proof. The proof of Part (i) is similar to that of Theorem 2(a) in [2], and follows by induction on i. For $i=1$, $\exists \mathbf{X}_1^1 F(\mathbf{X},\mathbf{Y}) \Leftrightarrow \widehat{F}(1,\mathbf{X}_2^n,0,\neg\mathbf{X}_2^n,\mathbf{Y}) \vee \widehat{F}(0,\mathbf{X}_2^n,1,\neg\mathbf{X}_2^n,\mathbf{Y}) \Rightarrow \widehat{F}(1,\mathbf{X}_2^n,1,\neg\mathbf{X}_2^n,\mathbf{Y}) = [\widehat{F}]_2[\overline{\mathbf{X}}_2^n \mapsto \neg\mathbf{X}_2^n]$ (by positive unateness of $\widehat{F}$ in $x_1,\overline{x_1}$). Conversely, as F is in SynNNF, $[\widehat{F}]_2$ is $\wedge_2$-unrealizable, which implies that with notation as in Definition 2, $\alpha_1^{11} \Rightarrow \alpha_1^{10} \vee \alpha_1^{01}$, i.e., $\widehat{F}(1,\mathbf{X}_2^n,1,\neg\mathbf{X}_2^n,\mathbf{Y}) \Rightarrow \widehat{F}(1,\mathbf{X}_2^n,0,\neg\mathbf{X}_2^n,\mathbf{Y}) \vee \widehat{F}(0,\mathbf{X}_2^n,1,\neg\mathbf{X}_2^n,\mathbf{Y})$. This give us the proof in the reverse direction, i.e., $[\widehat{F}]_2[\overline{\mathbf{X}}_2^n \mapsto \neg\mathbf{X}_2^n] \Rightarrow \exists \mathbf{X}_1^1 F(\mathbf{X},\mathbf{Y})$.

Suppose the statement holds for $1 \le i < n$. We will show that it holds for $i+1$ as well. By inductive hypothesis and definition of existential quantification, $\exists \mathbf{X}_1^{i+1} F(\mathbf{X},\mathbf{Y}) \Leftrightarrow \exists x_{i+1}[\widehat{F}]_{i+1}[\overline{\mathbf{X}}_{i+1}^n \mapsto \neg\mathbf{X}_{i+1}^n] \Leftrightarrow [\widehat{F}]_{i+1}[x_i \mapsto 1, \overline{\mathbf{X}}_{i+1}^n \mapsto \neg\mathbf{X}_{i+1}^n] \vee [\widehat{F}]_{i+1}[x_i \mapsto 0, \overline{\mathbf{X}}_{i+1}^n \mapsto \neg\mathbf{X}_{i+1}^n]$. Again, using unateness of $[\widehat{F}]_{i+1}$ in x_{i+1} and $\overline{x_{i+1}}$ in one direction, and using the defining property of SynNNF ($\alpha_{i+1}^{11} \Rightarrow \alpha_{i+1}^{10} \vee \alpha_{i+1}^{01}$) in the other direction, we obtain $\exists \mathbf{X}_1^{i+1} F(\mathbf{X},\mathbf{Y}) \Leftrightarrow [\widehat{F}]_{i+2}[\overline{\mathbf{X}}_{i+2}^n \mapsto \neg\mathbf{X}_{i+2}^n]$.

Part(ii): For $i \in \{1,\ldots n\}$, let $\psi_i'(\mathbf{X}_{i+1}^n,\mathbf{Y})$ denote $[\widehat{F}]_i[x_i \mapsto 1, \overline{x}_i \mapsto 0, \overline{\mathbf{X}}_{i+1}^n \mapsto \neg\mathbf{X}_{i+1}^n] = \alpha_i^{10}$. Further, from n to 1, we recursively define $\psi_n(\mathbf{Y}) = \psi_n'(\mathbf{Y})$ and $\psi_i(\mathbf{Y}) = \psi_i'(\Psi_{i+1}^n(\mathbf{Y}),\mathbf{Y})$. We can now show that $\psi_i(\mathbf{Y})$ is indeed a correct Skolem function for x_i in F. Starting from n to 1, we know from the preliminaries that $F^{(n-1)}[x_n \mapsto 1]$ gives a correct Skolem function for x_n in F. From part (i) above, $F^{(n-1)} \Leftrightarrow [\widehat{F}]_n[\overline{\mathbf{X}_n^n} \mapsto \neg\mathbf{X}_n^n]$. Hence $\alpha_n^{10} = \psi_n = \psi_n'$ gives a correct Skolem function for x_n in F. For any $i \in \{1,\ldots n-1\}$, assuming that Ψ_{i+1}^n gives a correct Skolem function vector for $\mathbf{X}_{i+1}^n$ in F, the same argument shows that $\psi_i'(\psi_{i+1}^n(\mathbf{Y}),\mathbf{Y})$ is a correct Skolem function for x_i in F.

Finally, note that $|\psi_n|$ is at most $|\widehat{F}|$, which is in $\mathcal{O}(|F|)$. A DAG representation of ψ_{n-k} requires a fresh copy of $[\widehat{F}]_{n-k}$, but can re-use the DAG representations of ψ_j for $j \in \{n-k+1,\ldots n\}$ as sub-DAGs. Thus, $|\psi_{n-k}|$ is in $\mathcal{O}(k\cdot|F|)$. Hence, if we use a multi-rooted DAG to represent all Skolem functions together, we need only $\mathcal{O}(n\cdot|F|)$ nodes. The time required is in $\mathcal{O}(n^2\cdot|F|)$ since the resulting DAG has $\sum_{k=1}^n k$ edges (root of ψ_j connects to a leaf of every ψ_i for $i<j$). □

The above polynomial-time strategy based on $[\widehat{F}]_i$ was used in [2] for computing over-approximations of Skolem functions $\psi_i(\mathbf{X}_{i+1},\mathbf{Y})$ for each $x_i \in \mathbf{X}$. Specifically, it was shown that $[\widehat{F}]_i[x_i \mapsto 1, \overline{x_i} \mapsto 1]$ over-approximates $\exists \mathbf{X}_1^i F(\mathbf{X},\mathbf{Y})$ and $[\widehat{F}]_i[x_i \mapsto 1, \overline{x_i} \mapsto 0]$ over-approximates a Skolem function for x_i in F. In the remainder of this paper, we refer to the functions ψ_i used in the proof of Part (ii) above as GACKS functions (after the author names of [2]). We use Ψ_1^n to denote the GACKS (Skolem) function vector $(\psi_1,\ldots,\psi_n)$.

2) Succinctness of SynNNF: SynNNF strictly subsumes many known representations used for efficient analysis of Boolean functions. In the following theorem, sizes and times are in terms of the number of input and output variables.

Theorem 2. *(i) Every specification in ROBDD/FBDD, dDNNF, DNNF or wDNNF form is either already in SynNNF or can be compiled in linear time to SynNNF.*

(ii) There exist poly-sized SynNNF specifications that only admit

a) exponential sized FBDD representations.

b) super-polynomial sized dDNNF representations, unless P = VNP

c) super-polynomial sized wDNNF and DNNF representations, unless P = NP.

(iii) There exist poly-sized NNF-representations that only admit super-polynomial sized SynNNF representations, unless the polynomial hierarchy collapses.

In the above, VNP is the algebraic analogue of NP [25]. We omit the proof of this theorem due to lack of space. This, and all skipped proofs, can be found in the full version of the paper [1]. Note that Theorem 2(iii) implies that we cannot always hope to obtain a succinct SynNNF representation.

3) SynNNF "almost" characterizes efficient synthesis using GACKS functions: We now show that SynNNF precisely characterizes specifications that admit linear-time existential quantification of output variables strengthening Theorem 1(i). Further, a slight weakening of SynNNF condition by restricting assignments on $\mathbf{X}_{i+1}^n$ gives us a necessary and sufficient condition for poly-time synthesis using GACKS functions.

Theorem 3. *Given a relational specification $F(\mathbf{X},\mathbf{Y})$,*

(i) F is in SynNNF iff $\exists \mathbf{X}_1^i F(\mathbf{X},\mathbf{Y}) \Leftrightarrow [\widehat{F}]_{i+1}[\overline{\mathbf{X}}_{i+1}^n \mapsto \neg\mathbf{X}_{i+1}^n]$

(ii) The GACKS-function vector Ψ_1^n is a Skolem function vector for $\mathbf{X}_1^n$ in $F(\mathbf{X},\mathbf{Y})$ iff $[\widehat{F}]_i[\mathbf{X}_{i+1}^n \mapsto \Psi_{i+1}^n, \overline{\mathbf{X}_{i+1}^n} \mapsto \neg\Psi_{i+1}^n]$ is $\wedge_i$-unrealizable for all $i \in \{1\ldots n\}$.

In [13], it was shown that an *error formula* ε for Ψ_1^n, defined as $F(\mathbf{X},\mathbf{Y}) \wedge \neg F(\mathbf{X}',\mathbf{Y}) \wedge \bigwedge_{i=1}^n (x_i' \leftrightarrow \Psi_i)$ is unsatisfiable iff Ψ_1^n is a Skolem function vector for F. Therefore, an (un)satisfiability check for ε serves to check if $[\widehat{F}]_i[\mathbf{X}_{i+1}^n \mapsto \Psi_{i+1}^n]$ is $\wedge_i$-unrealizable for all $i \in \{1\ldots n\}$. Further, in [2], it was observed experimentally, that GACKS functions give correct Skolem functions, even when the specifications are not in wDNNF. This surprising behavior, which was left unexplained in [2], can now be explained using SynNNF, thanks to Theorem 3(ii).

Note that Theorem 3(ii) weakens the requirement of SynNNF since $\mathbf{X}_{i+1}^n$ are constrained to take only the values defined by Ψ_{i+1}^n. For an example of a specification not in

SynNNF for which GACKS functions are correct Skolem functions, consider again H from Example 2, which we saw was not in SynNNF. In this case, $\psi_1'(x_2, \mathbf{Y}) = [\widehat{H}]_1[x_1 \mapsto 1, \overline{x}_1 \mapsto 0, \overline{x}_2 \mapsto \neg x_2] = \neg x_2 \wedge y_2$ and $\psi_2(\mathbf{Y}) = \psi_2'(\mathbf{Y}) = [\widehat{H}]_2[x_2 \mapsto 1, \overline{x}_2 \mapsto 0] = 1$. Therefore, $\psi_1(\mathbf{Y}) = \psi_1'[x_2 \mapsto \psi_2(\mathbf{Y})] = 0$. It can be verified that $x_1 = \psi_1(\mathbf{Y}) = 0, x_2 = \psi_2(\mathbf{Y}) = 1$ is indeed a correct Skolem function vector for $\mathbf{X}$ in H. Also, H satisfies the condition of Theorem 3(ii) since $[\widehat{H}]_1[x_2 \mapsto \psi_2, \overline{x}_2 \mapsto \neg\psi_2] = \overline{x}_1 \not\Leftrightarrow (x_1 \wedge \overline{x}_1)$, and $[\widehat{H}]_2 = 1$.

IV. Refinement for Synthesis

Given a specification $F(\mathbf{X}, \mathbf{Y})$, sometimes it is easier to solve the BFnS problem for a "simpler" specification $\widetilde{\mathsf{F}}(\mathbf{X}, \mathbf{Y})$ such that a solution for $\widetilde{\mathsf{F}}$ also serves as a solution for F. While "simplifications" of this nature have been used in earlier work [13], [2], [20], [6], we formalize this notion below as one of refinement.

Definition 4. *Let $F(\mathbf{X}, \mathbf{Y})$ and $\widetilde{\mathsf{F}}(\mathbf{X}, \mathbf{Y})$ be Boolean relational specifications. We say that* $\widetilde{\mathsf{F}}$ refines F w.r.t. synthesis*, denoted $\widetilde{\mathsf{F}} \preceq_{syn} F$, iff the following conditions hold: (a) $\forall \mathbf{Y}\left(\exists \mathbf{X} F(\mathbf{X}, \mathbf{Y}) \Rightarrow \exists \mathbf{X}' \widetilde{\mathsf{F}}(\mathbf{X}', \mathbf{Y})\right)$, and (b) $\forall \mathbf{Y} \forall \mathbf{X}'\left(\left(\exists \mathbf{X} F(\mathbf{X}, \mathbf{Y}) \wedge \widetilde{\mathsf{F}}(\mathbf{X}', \mathbf{Y})\right) \Rightarrow F(\mathbf{X}', \mathbf{Y})\right)$. If the implication in condition (a) is strengthened to a bi-implication, we say that* $\widetilde{\mathsf{F}}$ strongly refines F w.r.t. synthesis*, denoted $\widetilde{\mathsf{F}} \preceq^*_{syn} F$.*

Informally, condition (a) specifies that $\widetilde{\mathsf{F}}$ doesn't restrict (and preserves, for strong refinement) the set of input valuations over which the specification F can be satisfied, and condition (b) specifies that for all such input valuations $\mathbf{Y}$, any $\mathbf{X}'$ that satisfies $\widetilde{\mathsf{F}}$ also satisfies F.

Lemma 4. *If $\widetilde{\mathsf{F}} \preceq_{syn} F$, every Skolem function vector for $\mathbf{X}$ in $\widetilde{\mathsf{F}}$ is also a Skolem function vector for $\mathbf{X}$ in F.*

We say $\widetilde{\mathsf{F}}$ *refines* F w.r.t. synthesis because the set of all Skolem function vectors for $\mathbf{X}$ in $\widetilde{\mathsf{F}}$ is a subset of that for $\mathbf{X}$ in F. Note that Definition 4 provides a direct 2QBF-SAT based check of whether $\widetilde{\mathsf{F}}$ refines F without referring to the details of how $\widetilde{\mathsf{F}}$ is obtained from F.

Example 3. *Let $G(x_1, x_2, y_1, y_2) \equiv (\neg x_1 \vee x_2 \vee y_1) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee \neg y_1) \wedge (x_2 \vee y_2)$ and $\widetilde{\mathsf{G}}(x_1, x_2, y_1, y_2) \equiv x_2 \wedge x_1$. Although $G \not\Leftrightarrow \widetilde{\mathsf{G}}$, both conditions (a) and (b) of Definition 4 are satisfied; hence $\widetilde{\mathsf{G}} \preceq_{syn} G$. In fact, $\widetilde{\mathsf{G}} \preceq^*_{syn} G$.*

The following are easy consequences of Definition 4.

Proposition 5. *1) Both $\preceq_{syn}$ and $\preceq^*_{syn}$ are reflexive and transitive relations on Boolean relational specifications.*

2) If $\bigwedge_{y_j \in \mathbf{Y}} \left(F|_{y_j=0} \Leftrightarrow F|_{y_j=1}\right)$ and $\pi \models F(\mathbf{X}, \mathbf{Y})$, then $\mathsf{form}(\pi{\downarrow}\mathbf{X}) \preceq^*_{syn} F$.

3) If $\bigwedge_{x_i \in \mathbf{X}} (F|_{x_i=0} \Leftrightarrow F|_{x_i=1})$, then $1 \preceq_{syn} F$ and $F|_{\mathbf{X}=\mathbf{a}} \preceq^*_{syn} F$, where $\mathbf{a}$ is any vector in $\{0,1\}^n$.

4) If F is positive (resp. negative) unate in $x_i \in \mathbf{X}$, then $x_i \wedge F|_{x_i=1}$ (resp. $\neg x_i \wedge F|_{x_i=0}$) $\preceq^*_{syn} F$.

5) a) Let $\widetilde{\mathsf{F}}_1 \preceq^*_{syn} F_1$ and $\widetilde{\mathsf{F}}_2 \preceq^*_{syn} F_2$. Then $(\widetilde{\mathsf{F}}_1 \vee \widetilde{\mathsf{F}}_2) \preceq^*_{syn} (F_1 \vee F_2)$.

b) Let $\widetilde{\mathsf{F}}_1 \preceq_{syn} F_1$ and $\widetilde{\mathsf{F}}_2 \preceq_{syn} F_2$. If the output supports of F_1 and F_2, and similarly of $\widetilde{\mathsf{F}}_1$ and $\widetilde{\mathsf{F}}_2$, are disjoint, then $(\widetilde{\mathsf{F}}_1 \wedge \widetilde{\mathsf{F}}_2) \preceq_{syn} (F_1 \wedge F_2)$. If, in addition, $\widetilde{\mathsf{F}}_1 \preceq^*_{syn} F_1$ and $\widetilde{\mathsf{F}}_2 \preceq^*_{syn} F_2$, then $(\widetilde{\mathsf{F}}_1 \wedge \widetilde{\mathsf{F}}_2) \preceq^*_{syn} (F_1 \wedge F_2)$.

Note that Propositions 5(2) and 5(3) effectively require $F(\mathbf{X}, \mathbf{Y})$ to be semantically (but not necessarily syntactically) independent of $\mathbf{Y}$ and $\mathbf{X}$ respectively. Interestingly, a version of Proposition 5(4) was used in a pre-processing step of BFSS [2], although the precise notion of refinement w.r.t. synthesis was not defined there.

Suppose the specification $F(\mathbf{X}, \mathbf{Y})$ uniquely defines an output variable as a function of other input and output variables. For example, if $F(\mathbf{X}, \mathbf{Y}) \equiv (\neg x_i \vee x_j) \wedge (\neg x_i \vee y_k) \wedge (x_i \vee \neg x_j \vee \neg y_k) \wedge \cdots$, then $F(\mathbf{X}, \mathbf{Y}) \Rightarrow (x_i \Leftrightarrow (x_j \wedge y_k))$. Such specifications arise naturally when a non-CNF Boolean formula is converted to CNF via Tseitin encoding [24]. Variables like x_i above are said to be *functionally determined* (henceforth called FD) in F, and implied functional dependencies like $(x_i \leftrightarrow (x_j \wedge y_k))$ are called *functional definitions* (henceforth called *f-defs*) of FD variables in F.

Let $\mathbf{T} \subseteq \mathbf{X}$ be a set of FD output variables in F, and let $\mathsf{Fun}_{\mathbf{T}}$ be the conjunction of f-defs of all variables in $\mathbf{T}$. We say that $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ is an *acyclic system of f-defs* if no variable in $\mathbf{T}$ transitively depends on itself via the functional definitions in $\mathsf{Fun}_{\mathbf{T}}$. In other words, $\mathsf{Fun}_{\mathbf{T}}$ induces an acyclic system of functional dependencies between variables in $\mathbf{T}$. For $x_i \in \mathbf{X} \setminus \mathbf{T}$, define $\theta_{F,\mathbf{T},x_i,a}$ to be the formula $\left(F(\mathbf{X}, \mathbf{Y})|_{x_i=a} \wedge \bigwedge_{x_j \in \mathbf{X} \setminus (\mathbf{T} \cup \{x_i\})} (x_j \Leftrightarrow x_j') \wedge \mathsf{Fun}_{\mathbf{T}}(\mathbf{X}', \mathbf{Y})|_{x_i'=1-a}\right) \Rightarrow F(\mathbf{X}', \mathbf{Y})|_{x_i'=1-a}$, where $a \in \{0,1\}$ and $\mathbf{X}'$ is a sequence of fresh variables $(x_1', \ldots x_n')$. Informally, $\theta_{F,\mathbf{T},x_i,a}$ asserts that if the specification F can be satisfied by setting a non-FD output x_i to a, then it can also be satisfied by setting x_i to the complement value $(1-a)$, while preserving the values of all other non-FD outputs. The FD outputs in $\mathbf{T}$ must of course be set as per the functional definitions in $\mathsf{Fun}_{\mathbf{T}}$.

Lemma 6. *Let $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ be an acyclic system of f-defs in F.*

1) If $\mathbf{X} = \mathbf{T}$, then $\mathsf{Fun}_{\mathbf{T}}(\mathbf{X}, \mathbf{Y}) \preceq_{syn} F(\mathbf{X}, \mathbf{Y})$. Furthermore, $\mathsf{Fun}_{\mathbf{T}}(\mathbf{X}, \mathbf{Y}) \wedge \exists \mathbf{X} F(\mathbf{X}, \mathbf{Y})) \preceq^*_{syn} F(\mathbf{X}, \mathbf{Y})$

2) If $\mathbf{X} \setminus \mathbf{T} \neq \emptyset$, then for every $x_i \in \mathbf{X} \setminus \mathbf{T}$, we have: If $\theta_{F,\mathbf{T},x_i,0}$ is a tautology, then $(x_i \wedge F|_{x_i=1}) \preceq^*_{syn} F$. Similarly, if $\theta_{F,\mathbf{T},x_i,1}$ is a tautology, then $(\neg x_i \wedge F|_{x_i=0}) \preceq^*_{syn} F$.

If $\mathbf{T} = \emptyset$, Lemma 6(2) simply reduces to Proposition 5(4). However, if $\mathbf{T} \neq \emptyset$ (as is often the case), Lemma 6(2) shows that $x_i \wedge F|_{x_i=1}$ (resp. $\neg x_i \wedge F|_{x_i=0}$) can refine F even if F is not positive (resp. negative) unate in x_i. As an illustration, the specification $G(x_1, x_2, y_1, y_2)$ in Example 3 is not unate in either x_1 or x_2. However, with $\mathbf{T} = \{x_1\}$ and $\mathsf{Fun}_{\mathbf{T}} \equiv (x_1 \Leftrightarrow (x_2 \vee y_1))$, we have $\theta_{F,\mathbf{T},x_2,0} \equiv 1$. Hence, $x_2 \wedge G|_{x_2=1} \equiv$

$(x_1 \wedge x_2) \preceq^*_{syn} G$. When F is refined by an application of Lemma 6(2), we say that F is refined by *pivoting on* x_i.

Lemma 7. *Let* $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ *and* $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$ *be acyclic systems of f-defs in* F*, where* $\mathbf{T}' \subseteq \mathbf{T} \subseteq \mathbf{X}$ *and* $\mathsf{Fun}_{\mathbf{T}} \equiv \mathsf{Fun}_{\mathbf{T}'} \wedge \mathsf{Fun}_{\mathbf{T}\setminus\mathbf{T}'}$*. For* $a \in \{0,1\}$*, if* $\theta_{F,\mathbf{T}',x_i,a}$ *is a tautology, then so is* $\theta_{F,\mathbf{T},x_i,a}$*.*

Lemma 7, along with Lemma 6(2), shows that if $\mathbf{T}' \subsetneq \mathbf{T} \subseteq \mathbf{X}$, the system of acyclic f-defs $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ potentially provides more opportunities for refinement compared to $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$. Hence, it is advantageous to augment the set $\mathbf{T}$ of FD outputs (and correspondingly $\mathsf{Fun}_{\mathbf{T}}$) whenever possible.

The following theorem suggests that compiling a given specification to a refined SynNNF specification (as opposed to an equivalent SynNNF specification) holds promise for Boolean functional synthesis.

Theorem 8. *For every relational specification* $F(\mathbf{X}, \mathbf{Y})$*, there exists a polynomial-sized Skolem function vector for* $\mathbf{X}$ *in* F *iff there exists a SynNNF specification* $\widetilde{\mathsf{F}}(\mathbf{X}, \mathbf{Y})$ *such that* $\widetilde{\mathsf{F}} \preceq_{syn} F$ *and* $\widetilde{\mathsf{F}}$ *is polynomial-sized in* F*.*

Theorem 8 guarantees that whenever a polynomial-sized Skolem function vector exists for a specification $F(\mathbf{X}, \mathbf{Y})$, there is also a polynomial-sized refined specification in SynNNF. It is therefore interesting to ask if we can compile $F(\mathbf{X}, \mathbf{Y})$ to a "small enough" SynNNF specification $\widetilde{\mathsf{F}}(\mathbf{X}, \mathbf{Y})$ that refines F. In the next section, we present such a compilation algorithm and results of our preliminary experiments using this algorithm. As shown in [2], there exist problem instances for which there are no polynomial-sized Skolem functions, unless the Polynomial Hierarchy (PH) collapses. Thus, any algorithm for compilation to SynNNF must incur super-polynomial blow-up (unless PH collapses). Nevertheless, as our experiments show, the compilation-based approach works reasonably well in practice, even solving benchmarks beyond the reach of existing state-of-the-art BFnS tools.

V. A Refining CNF to SynNNF Compiler

We now describe C2Syn – an algorithm that takes as input a CNF specification $F(\mathbf{X}, \mathbf{Y})$ given as a set of clauses, and outputs a DAG representation of a SynNNF specification $\widetilde{F}(\mathbf{X}, \mathbf{Y})$ that refines $F(\mathbf{X}, \mathbf{Y})$ w.r.t. synthesis. Let $\mathcal{S} = \{C_1, \ldots C_r\}$ be a set of clauses. We use $\varphi_{\mathcal{S}}$ to denote the formula $\bigwedge_{C_i \in \mathcal{S}} C_i$. Abusing notation introduced in Section II, let $atoms(C_i) = \{z \mid z \in \mathbf{X} \cup \mathbf{Y}, lits(C_i) \cap \{z, \neg z\} \neq \emptyset\}$. We define an undirected graph $G_{\mathcal{S}} = (V_{\mathcal{S}}, E_{\mathcal{S}})$, where $V_{\mathcal{S}} = \{C_1, \ldots C_r\}$ and $(C_i, C_j) \in E_{\mathcal{S}}$ iff $i \neq j$ and $atoms(C_i) \cap atoms(C_j) \cap \mathbf{X} \neq \emptyset$. Thus, there exists an edge (C_i, C_j) iff C_i and C_j share an output atom. Let $\{\mathcal{S}_1, \ldots \mathcal{S}_q\}$ be the set of maximally connected components (henceforth called MCCs) of $G_{\mathcal{S}}$. It is easy to see that $\varphi_{\mathcal{S}} \equiv \bigwedge_{k=1}^{q} \varphi_{\mathcal{S}_k}$; moreover, the output supports of $\varphi_{\mathcal{S}_k}$ for $k \in \{1, \ldots q\}$ are mutually disjoint. We use $C_i \sim_{\mathcal{S}} C_j$ to denote that clauses C_i and C_j are in the same MCC of $G_{\mathcal{S}}$. We will soon see how factoring $\varphi_{\mathcal{S}}$ based on MCCs of $G_{\mathcal{S}}$ allows us to decompose the CNF-to-SynNNF compilation problem into independent sub-problems, thanks to Proposition 5(5)b. Note that factoring based on MCCs has also been used in DSharp [18] for converting a CNF formula to dDNNF. However, unlike $G_{\mathcal{S}}$ above, the underlying graph in DSharp has an edge between every pair of clauses that shares any atom, including input variables. Thus, $G_{\mathcal{S}}$ has potentially fewer edges, and hence smaller MCCs, than the corresponding graph constructed by DSharp. Before delving into Algorithm C2Syn, we first discuss some important sub-routines used in the algorithm. Sub-routine FDRefine takes as inputs a set $\mathcal{S}$ of clauses and a (possibly empty) acyclic system of f-defs $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ in $\varphi_{\mathcal{S}}$. It returns a (possibly augmented) acyclic system of f-defs $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$ and a set of clauses $\mathcal{S}'$ such that $\varphi_{\mathcal{S}'} \preceq^*_{syn} \varphi_{\mathcal{S}}$ and $\varphi_{\mathcal{S}'} \Rightarrow \mathsf{Fun}_{\mathbf{T}'}$. Sub-routine FDRefine works by iteratively finding new FD ouptut variables and refining the specification using Lemma 6(2) whenever possible. In the pseudo-code of FDRefine (see Algorithm 1), sub-routine FindFD matches a pre-defined set of clause-patterns in $\mathcal{S}'$ to identify new FD output variables not already in $\mathbf{T}'$. The patterns currently matched correspond to CNF encodings of the input-output relation of common Boolean functions, viz. and, or, nand, nor, xor, xnor, not and identity. For example, we match the pattern $(\neg\alpha \vee \beta_1) \wedge (\neg\alpha \vee \beta_2) \wedge (\neg\beta_1 \vee \neg\beta_2 \vee \alpha)$, where α, β_1, β_2 are place-holders, to identify the functional definition $(\alpha \leftrightarrow (\beta_1 \wedge \beta_2))$. Each new FD output variable thus identified is added to $\mathbf{T}'$ and the corresponding functional definition is added to $\mathsf{Fun}_{\mathbf{T}'}$ unless this introduces a cyclic dependency among the f-defs already in $\mathsf{Fun}_{\mathbf{T}'}$. Assuming all patterns used by FindFD to determine functional dependencies are sound, the (possibly augmented) $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$ computed by FindFD is a system of acyclic f-defs in $\varphi_{\mathcal{S}'}$. In lines 6-12 of Algorithm 1, we next check if Lemma 6(2) can be applied to refine $\varphi_{\mathcal{S}'}$ by pivoting on some variable $x_i \in \mathbf{Out} \setminus \mathbf{T}'$. The refinement, if applicable, is easily done by replacing each clause $C_i \in \mathcal{S}'$ by $C_i|_{x_i=1}$ (resp. $C_i|_{x_i=0}$) and by adding the unit clause x_i (resp. $\neg x_i$) to $\mathcal{S}'$. The pivot x_i is also added to $\mathbf{T}'$ and the corresponding functional definition ($x_i \leftrightarrow 1$ or $x_i \leftrightarrow 0$ as the case may be) is added to $\mathsf{Fun}_{\mathbf{T}'}$.

In general, identifying an acyclic system of f-defs in F potentially enables refinement of F via Lemma 6(2), which

Algorithm 1: FDRefine

Input: $\mathcal{S}$: set of clauses, $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$: acyclic f-defs in $\varphi_{\mathcal{S}}$
Output: $\mathcal{S}'$: set of clauses s.t. $\varphi_{\mathcal{S}'} \preceq^*_{syn} \varphi_{\mathcal{S}}$,
$(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$: Augmented acyclic f-defs in $\varphi_{\mathcal{S}'}$

1 $\mathbf{Out} := \mathsf{sup}(\varphi_{\mathcal{S}}) \cap \mathbf{X}$;
2 $\mathcal{S}' := \mathcal{S}$; $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'}) := (\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$; /* initialization */
3 **repeat**
4 $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'}) := \text{FindFD}(\mathcal{S}', \mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$;
5 Let F be the formula $\varphi_{\mathcal{S}'}$;
6 **foreach** $x_i \in \mathbf{Out} \setminus \mathbf{T}'$ **do**
7 **if** $\theta_{F,\mathbf{T}',x_i,0}$ *is a tautology* **then**
8 $\mathcal{S}' := \mathcal{S}'|_{x_i=1} \cup \{x_i\}$; $\mathbf{T}' = \mathbf{T}' \cup \{x_i\}$;
9 $\mathsf{Fun}_{\mathbf{T}'} := \mathsf{Fun}_{\mathbf{T}'} \wedge (x_i \leftrightarrow 1)$;
10 **else if** $\theta_{F,\mathbf{T}',x_i,1}$ *is a tautology* **then**
11 $\mathcal{S}' := \mathcal{S}'|_{x_i=0} \cup \{\neg x_i\}$; $\mathbf{T}' = \mathbf{T}' \cup \{x_i\}$;
12 $\mathsf{Fun}_{\mathbf{T}'} := \mathsf{Fun}_{\mathbf{T}'} \wedge (x_i \leftrightarrow 0)$;
13 **until** *either* $\mathbf{T}'$ *or* $\mathcal{S}'$ *changes*;
14 **return** $(\mathcal{S}', \mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$;

in turn, can lead to augmenting the acyclic system of f-defs further. Therefore, the loop in lines 3-13 of Algorithm 1 is iterated until no new FD outputs or additional refinements are obtained. Once this happens, subroutine FDREFINE returns the resulting acyclic system of f-defs $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'})$ and the resulting set of refined clauses $\mathcal{S}'$.

Two other important sub-routines used in C2Syn are GETCKT and GETDEFCKT. Sub-routine GETCKT takes as input an NNF specification $G(\mathbf{X}, \mathbf{Y})$ and returns the DAG representation of $G(\mathbf{X}, \mathbf{Y})$. Sub-routine GETDEFCKT is slightly more involved. It takes as input an NNF specification $G(\mathbf{X}, \mathbf{Y})$ and a system of acyclic f-defs $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$, where $\mathbf{X} = \mathbf{T}$. It returns a DAG representation of a SynNNF specification equivalent to $\mathsf{Fun}_{\mathbf{T}} \wedge \exists \mathbf{X} G(\mathbf{X}, \mathbf{Y})$. This is accomplished as follows. Let $x_i \Leftrightarrow \mathsf{op}_i(u_1, \ldots u_{n_i})$ be the functional definition of x_i in $\mathsf{Fun}_{\mathbf{T}}$, where op_i is a Boolean function identified via clause-pattern matching in sub-routine FINDFD. For each $x_i \in \mathbf{T}$, GETDEFCKT first constructs two DAGs, $\mathcal{D}_i$ and $\mathcal{E}_i$, representing $\mathsf{op}_i(u_1, \ldots u_{n_i})$ and $\neg\mathsf{op}_i(u_1, \ldots u_{n_i})$ in NNF. Let the root nodes of these DAGs be labeled v_i and nv_i respectively. Then, GETDEFCKT constructs a DAG representing the formula $\xi(\mathbf{Z}, \mathbf{V}, \mathbf{Y}) \equiv G(\mathbf{V}, \mathbf{Y}) \wedge \bigwedge_{i=1}^{|\mathbf{X}|} ((z_i \wedge v_i) \vee (\neg z_i \wedge \neg v_i))$, where $\mathbf{Z}$ and $\mathbf{V}$ are vectors of fresh variables with $|\mathbf{Z}| = |\mathbf{V}| = |\mathbf{X}|$. For every leaf l in this DAG that is labeled v_i (resp. $\neg v_i$), sub-routine GETDEFCKT now creates an edge from the root of $\mathcal{D}_i$ (resp. $\mathcal{E}_i$) to l, rendering l a non-leaf node. However, this may result in some new leaves (coming from DAGs like $\mathcal{D}_i$ and $\mathcal{E}_i$) with labels from $\mathbf{X}$ or their negations. For every such leaf l' labeled x_j (resp. $\neg x_j$), where $x_j \in \mathbf{T}$, GETDEFCKT also creates an edge from the root of $\mathcal{D}_j$ (resp. $\mathcal{E}_j$) to l'. The above steps are repeatedly applied until all leaves have labels only from $\mathbf{Z} \cup \mathbf{Y}$ and their negations. Since $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ is an acyclic system of f-defs, the above operation is guaranteed to terminate without introducing any cycles. Finally, GETDEFCKT replaces every leaf in the resulting graph with label z_i (resp. $\neg z_i$) with x_i (resp. $\neg x_i$) and returns the resulting DAG, say $\mathcal{D}$. It is easy to see that $\mathcal{D}$ represents $\mathsf{Fun}_{\mathbf{T}} \wedge \exists \mathbf{X} G(\mathbf{X}, \mathbf{Y})$ in SynNNF.

We are now in a position to describe Algorithm C2Syn. The algorithm is recursive and takes as inputs a set $\mathcal{S}$ of clauses, a (possibly empty) system of acyclic f-defs $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$ in $\varphi_{\mathcal{S}}$, and the recursion level ℓ. Initially, C2Syn is invoked with $\mathcal{S} =$ given set of CNF clauses, $\mathbf{T} = \emptyset$, $\mathsf{Fun}_{\mathbf{T}} = 1$ and $\ell = 0$. The pseudocode of C2Syn, shown in Algorithm 2, first computes the output support $\mathbf{Out}$ of $\varphi_{\mathcal{S}}$, and then checks a few degenerate cases (lines 2-8) to determine if a refined SynNNF specification can be easily obtained. In case these checks fail, sub-routine FDREFINE is invoked to augment the set $\mathbf{T}'$ of functionally dependent outputs and their corresponding acyclic f-defs $\mathsf{Fun}_{\mathbf{T}'}$, and also to obtain a (possibly) refined set $\mathcal{S}'$ of clauses. If all outputs in $\mathbf{Out}$ get functionally determined by this, we use Lemma 6(1) to get the desired SynNNF by invoking GETDEFCKT in line 12. Otherwise, we check in lines 14-17 if Theorem 3(ii) can be applied. Recall that Theorem 3(ii) relaxes the requirements of the SynNNF definition by requiring $\wedge_i$-unrealizability only when GACKS

Algorithm 2: C2Syn

Input: $\mathcal{S}$: set of clauses, $(\mathbf{T}, \mathsf{Fun}_{\mathbf{T}})$: acyclic f-defs in $\varphi_{\mathcal{S}}$, ℓ: recursion level
Output: DAG representation of $\widetilde{\mathsf{F}}$ in SynNNF s.t. $\widetilde{\mathsf{F}} \preceq^*_{syn} \varphi_{\mathcal{S}}$

1 $\mathbf{Out} := \mathsf{sup}(\varphi_{\mathcal{S}}) \cap \mathbf{X}$;
2 **if** $\varphi_{\mathcal{S}}$ *is valid (resp. inconsistent)* **then**
3 **return** GETCKT(1) (resp. GETCKT(0));
4 **else if** $\varphi_{\mathcal{S}}$ *is semantically independent of inputs* $\mathbf{Y}$ **then**
5 Let π be a satisfying assignment of $\varphi_{\mathcal{S}}$;
6 **return** GETCKT(form($\pi{\downarrow}\mathbf{Out}$));
7 **else if** $\varphi_{\mathcal{S}}$ *is semantically independent of* $\mathbf{Out}$ **then**
8 **return** GETCKT($\varphi_{\mathcal{S}}|_{\mathbf{Out}=\mathbf{a}}$); // **a** any vector in $\{0,1\}^{|\mathbf{Out}|}$
9 **else**
10 $(\mathbf{T}', \mathsf{Fun}_{\mathbf{T}'}, \mathcal{S}')$:= FDREFINE($\mathcal{S}, \mathbf{T}, \mathsf{Fun}_{\mathbf{T}}$);
11 **if** $\mathbf{Out} \setminus \mathbf{T}' = \emptyset$ **then**
12 **return** GETDEFCKT($\varphi_{\mathcal{S}'}, \mathbf{Out}, \mathsf{Fun}_{\mathbf{Out}}$);
13 **else**
14 Let Ψ be GACKS Skolem function vector for $\mathbf{Out}$ in $\varphi_{\mathcal{S}'}$;
15 Let $\varepsilon :=$ $\varphi_{\mathcal{S}'}(\mathbf{Out}, \mathbf{Y}) \wedge \neg\varphi_{\mathcal{S}'}(\mathbf{Out}', \mathbf{Y}) \wedge \bigwedge_{x_i \in \mathbf{Out}} (x'_i \Leftrightarrow \Psi_i)$; /* error formula for Ψ, as in [13] */
16 **if** ε *is unsat* **then**
17 **return** GETDEFCKT($\varphi_{\mathcal{S}'}, \mathbf{Out}, \bigwedge_{x_i \in \mathbf{Out}} (x_i \Leftrightarrow \Psi_i)$);
18 $x :=$ CHOOSEOUTPUTVAR($\mathcal{S}', \mathbf{Out} \setminus \mathbf{T}'$);
19 Pos := $\{C_j \in \mathcal{S}' \mid x \in lits(C_j)\}$;
20 Neg := $\{C_j \in \mathcal{S}' \mid \neg x \in lits(C_j)\}$;
21 $\mathcal{S}_1 := \{C_i \in \mathcal{S}' \mid \exists C_j \in \text{Pos } (C_i \sim_{\mathcal{S}'} C_j)\}$;
22 $\mathbf{T}_1 := \mathbf{T}' \cap \mathsf{sup}(\varphi_{\mathcal{S}_1})$;
23 $\mathcal{S}_2 := \{C_i \in \mathcal{S}' \mid \exists C_j \in \text{Neg } (C_i \sim_{\mathcal{S}'} C_j)\}$;
24 $\mathbf{T}_2 := \mathbf{T}' \cap \mathsf{sup}(\varphi_{\mathcal{S}_2})$;
25 $\mathcal{S}_3 := \{C_i \in \mathcal{S}' \mid \forall C_j \in \text{Pos} \cup \text{Neg } (C_i \not\sim_{\mathcal{S}'} C_j)$;
26 $\mathbf{T}_3 := \mathbf{T}' \cap \mathsf{sup}(\varphi_{\mathcal{S}_3})$;
27 Let t_1 := root of C2Syn($\mathcal{S}_1|_{x=0}, \mathbf{T}_1, \mathsf{Fun}_{\mathbf{T}_1}|_{x=0}, \ell+1$);
28 Let t_2 := root of C2Syn($\mathcal{S}_2|_{x=1}, \mathbf{T}_2, \mathsf{Fun}_{\mathbf{T}_2}|_{x=1}, \ell+1$);
29 Let t_3 := root of C2Syn($\mathcal{S}_3, \mathbf{T}_3, \mathsf{Fun}_{\mathbf{T}_3}, \ell+1$);
30 **return** GETCKT($t_3 \wedge ((x \wedge t_2) \vee (\neg x \wedge t_1))$)

functions are substituted for the $\mathbf{X}$ variables. As discussed in Section III-3, the relaxed requirement can be checked by testing the unsatisfiability of the error formula ε for the GACKS function vector Ψ. If ε is indeed unsatisfiable, Ψ is a Skolem function vector for $\mathbf{Out}$ in $\varphi_{\mathcal{S}'}$.

If ε is satisfiable, we use a sub-routine CHOOSEOUTPUTVAR that heuristically chooses an output variable $x \in \mathbf{Out} \setminus \mathbf{T}'$ on which to branch. Currently, we use a VSIDS [17] score based heuristic, similar to that used in DSHARP [18], to rank variables in $\mathbf{Out} \setminus \mathbf{T}'$, and then choose the variable with the highest score. This allows us to represent $\varphi_{\mathcal{S}'}$ as $x_i \wedge \varphi_{\mathcal{S}'|_{x=1}} \vee \neg x_i \wedge \varphi_{\mathcal{S}'|_{x=0}}$, so that we can refine the two disjuncts independently, thanks to Proposition 5(5)a. However, this may lead to some duplicate processing of clauses. We can avoid this by factoring out the subset of clauses whose satisfiability is independent of whether x_i is set to 1 or 0. Let $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) be the subset of clauses in $\mathcal{S}'$ that are in the same MCC of $G_{\mathcal{S}'}$ as some C_j that has x (resp. $\neg x$) as a literal. Let $\mathcal{S}_3$ be the set of all clauses in $\mathcal{S}'$ that are neither in $\mathcal{S}_1$ nor $\mathcal{S}_2$. By definition of $G_{\mathcal{S}'}$, the sub-specifications $\varphi_{\mathcal{S}_1}$ and $\varphi_{\mathcal{S}_3}$ (and similarly, $\varphi_{\mathcal{S}_2}$ and $\varphi_{\mathcal{S}_3}$) do not share any output variable in their supports, and can be refined independently. This is exactly what algorithm C2Syn does in lines 19-30. The roots of the DAGs resulting from the recursive calls in lines 27, 28 and 29 are finally combined as in line 30 to yield the desired DAG representation.

Theorem 9. *For every set $\mathcal{S}$ of clauses,* C2Syn$(\mathcal{S}, \emptyset, 1, 0)$ *always terminates and returns a DAG representation of a* SynNNF *specification* $\widetilde{\mathsf{F}}$ *s.t.,* $\widetilde{\mathsf{F}} \preceq^*_{syn} \varphi_{\mathcal{S}}$. *The worst-case size of* $\widetilde{\mathsf{F}}$ *is linear in* $|\mathcal{S}|$ *and exponential in the maximum recursion level, which is bounded above by the output support of* $\varphi_{\mathcal{S}}$.

VI. Experimental results

We ran Algorithm C2Syn on a suite of CNF specifications comprised of benchmarks from the Prenex 2QBF track of QBFEVAL 2018 [19], and the `.qdimacs` version of FACTORIZATION benchmarks [2], which we will refer to as FA.QD. By Theorem 2(i), a ROBDD/FBDD specification can be compiled to an equivalent SynNNF specification in linear time. Therefore, any algorithm that compiles a CNF specification to an ROBDD can be viewed as an alternative to C2Syn for compiling a CNF specification to SynNNF (albeit without refinement). We compare the performance of C2Syn with that of a BDD compiler and two state-of-the-art boolean function synthesis tools, namely, (i) the AIG-NNF pipeline of BFSS [2] with ABC's MiniSat as the SAT solver and (ii) CADET [20], [22]. For the BDD Compiler, the `.qdimacs` input was converted to an AIG using simple Tseitin variable detection; this AIG was then simplified and ROBDDs built using dynamic variable ordering (of all input and output variables) – this is part of the BDD pipeline of BFSS [2], henceforth called BDD$^{\text{BFSS}}$. We also ran DSHARP [18] which compiles a CNF formula into dDNNF (and hence SynNNF by Theorem 2(i)), but it was successful on very few of our benchmarks; hence we do not present its performance. Each tool took as input the same `.qdimacs` file. Experiments were performed on a cluster with 20 cores and 64 GB memory per node, each core being a 2.2 GHz Intel Xeon processor running CentOS6.5. Each run was performed on a single core, with timeout of 1 hour and main memory limited to 16GB.

For C2Syn, several benchmarks were solved in the initial part of the Algorithm 2 before line 17, i.e., before any recursive calls are made. Table I presents the results for C2Syn, divided into those that succeeded at recursion level zero (Stage-I) and those that required recursions (Stage-II), as well as comparison with BDD$^{\text{BFSS}}$. Since BDDs are also in SynNNF, the total number of benchmarks in QBFEVAL which could be compiled into SynNNF (by either compiler) is a whopping 283.

Benchmarks (Total)	Compiled By C2Syn			BDD compilation	Total in SynNNF
	Stage I	Stage II	Total		
QBFEVAL (402)	103	83	186	153	**283**
FA.QD (6)	0	6	6	6	**6**

TABLE I: Compilation into SynNNF

Figure 1 (left) compares the run-times of C2Syn and BDD$^{\text{BFSS}}$: for most QBFEVAL benchmarks that were solved by both, C2Syn took less time, while for FA.QD, C2Syn took more time. There were 130 QBFEVAL benchmarks that C2Syn solved by BDD$^{\text{BFSS}}$ couldn't, whereas 98 were solved by BDD$^{\text{BFSS}}$ but not C2Syn.

We next compare C2Syn with CADET and BFSS. CADET (resp. BFSS) solved 213 (resp. 181) benchmarks in QBFEVAL and 4 (resp. 3) in FA.QD. Table II gives a comparison in terms of number of benchmarks solved by each tool but not by others. Figure 1 (middle, right) compares the run-times of C2Syn and those of CADET and BFSS, respectively. As expected, since C2Syn does complete compilation, it takes more time than CADET and marginally more than BFSS on many benchmarks, though for most of these, the time taken is less than a minute. In fact for FA.QD, C2Syn takes less time than BFSS on all benchmarks. Overall, C2Syn appears to have strengths orthogonal to BDD$^{\text{BFSS}}$, BFSS and CADET, and adds to the repertoire of state-of-the-art tools for Boolean functional synthesis.

Bench mark	C2Syn vs CADET		C2Syn vs BFSS		C2Syn \ (CADET ∪ BFSS)
	C2Syn\ CADET	CADET\ C2Syn	C2Syn\ BFSS	BFSS\ C2Syn	
QBFEVAL	78	105	83	78	**75**
FA.QD	2	0	3	0	**2**

TABLE II: Comparison Results of C2Syn

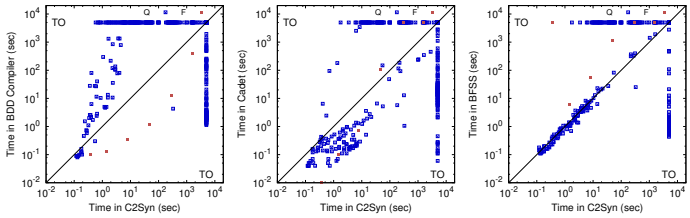

Fig. 1: Time comparisons: C2Syn vs BDD$^{\text{BFSS}}$, CADET, BFSS

To validate our experimental results, we also developed an independent approach to verify if the output of C2Syn is (i) in SynNNF and (ii) a refinement of the original specification Of the 83 QBFEVAL benchmarks that required C2Syn to go beyond recursion level 0, successfully verified 82 and ran out of memory on 1. Of the 6 factorization benchmarks on which C2Syn was successful, our verifier successfully verified 4 and ran out of time on one and out of memory on another benchmark (time limit: 2 hours, memory limit: 16GB). More details of the verification approach, as well as size comparison plots are in [1].

Finally, note that Stage-I of C2Syn subsumes some preprocessing techniques used in SAT/QBF-SAT solving, e.g., unit clause and pure literal detection, semantic unateness checks and Tseitin variable identification. Using more aggressive preprocessing could further improve the performance of our tool. We leave this for future work.

VII. Conclusion

We presented a new sub-class of NNF called SynNNF that admits quadratic-time synthesis and linear-time existential quantification of a set of variables. Our prototype compiler is able to handle several benchmarks that cannot be handled by other state-of-the-art tools. Since representations like ROBDDs, DNNF and the like are either already in or easily transformable to SynNNF, our work is widely applicable and can be used in tandem with other techniques. As future work, we intend to work on optimizing our SynNNF compiler.

REFERENCES

[1] S. Akshay, J. Arora, S. Chakraborty, S. Krishna, D. Raghunathan, and S. Shah. Knowledge compilation for boolean functional synthesis. *CoRR*, submit/2808510, 2019.

[2] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah. What's hard about Boolean functional synthesis? 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 I*, pages 251–269, 2018.

[3] S. Akshay, S. Chakraborty, A. K. John, and S. Shah. Towards Parallel Boolean Functional Synthesis. In *TACAS 2017 Proceedings, Part I*, pages 337–353, 2017.

[4] G. Boole. *The Mathematical Analysis of Logic*. Philosophical Library, 1847.

[5] M. Cadoli and F. M. Donini. A survey on knowledge compilation. *AI Commun.*, 10(3-4):137–150, 1997.

[6] S. Chakraborty, D. Fried, L. M. Tabajara, and M. Y. Vardi. Functional synthesis via input-output separation. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9, 2018.

[7] A. Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001.

[8] A. Darwiche and P. Marquis. A knowledge compilation map. *CoRR*, abs/1106.1819, 2011.

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

[10] M. Heule, M. Seidl, and A. Biere. Efficient Extraction of Skolem Functions from QRAT Proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 107–114, 2014.

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

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

[13] A. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay. Skolem functions for factored formulas. In *FMCAD*, pages 73–80, 2015.

[14] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. *SIGPLAN Not.*, 45(6):316–329, June 2010.

[15] J. Lang, P. Liberatore, and P. Marquis. Propositional independence - formula-variable independence and forgetting. *CoRR*, abs/1106.4578, 2011.

[16] L. Lowenheim. Über die Auflösung von Gleichungen in Logischen Gebietkalkul. *Math. Ann.*, 68:169–207, 1910.

[17] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[18] C. Muise, S. A. McIlraith, C. Beck, and E. Hsu. DSHARP: Fast d-DNNF Compilation with sharpSAT . In *AAAI-16 Workshop on Beyond NP*, 2016.

[19] QBFLib. QBFEval 2018. http://www.qbflib.org/qbfeval18.php.

[20] M. N. Rabe and S. A. Seshia. Incremental determinization. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 375–392, 2016.

[21] M. N. Rabe and L. Tentrup. CAQE: A certifying QBF solver. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 136–143, 2015.

[22] M. N. Rabe, L. Tentrup, C. Rasmussen, and S. A. Seshia. 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*, pages 256–274, 2018.

[23] L. M. Tabajara and M. Y. Vardi. Factored boolean functional synthesis. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 124–131, 2017.

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

[25] L. G. Valiant. Completeness classes in algebra. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 249–261, New York, NY, USA, 1979. ACM.