Counterexample Guided Knowledge Compilation for Boolean Functional Synthesis

S. Akshay(✉), Supratik Chakraborty(✉), and Sahil Jain

Indian Institute of Technology Bombay, Mumbai, India
{akshayss,supratik}@cse.iitb.ac.in, sahil.jain1.c2022@iitbombay.org

Abstract. Given a specification as a Boolean relation between inputs and outputs, Boolean functional synthesis generates a function, called a Skolem function, for each output in terms of the inputs such that the specification is satisfied. In general, there may be many possibilities for Skolem functions satisfying the same specification, and criteria to pick one or the other may vary from specification to specification.

In this paper, we develop a technique to represent the space of Skolem functions in a criteria-agnostic form that makes it possible to subsequently extract Skolem functions for different criteria. Our focus is on identifying such a form and on developing a compilation algorithm for this form. Our approach is based on a novel counter-example guided strategy for existentially quantifying a subset of variables from a specification in negation normal form. We implement this technique and compare our performance with those of other knowledge compilation approaches for Boolean functional synthesis, and show promising results.

1 Introduction

Manually designing systems that satisfy complex user-provided specifications can be notoriously tricky. *Automated synthesis* has therefore attracted significant attention of researchers over the past few decades [1–5]. In this paradigm, a user describes the desired behaviour of a system as a relational specification between its inputs and outputs, and an algorithm automatically generates an implementation, such that the specification is provably satisfied. In this paper, we focus only on systems with Boolean inputs and outputs with relational specifications given as Boolean formulas. The synthesis problem in this setting is also called *Boolean functional synthesis*. Formally, let $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ be a Boolean formula representing the specification, where $\boldsymbol{X} = (x_1, \ldots x_m)$ is a vector of Boolean inputs and $\boldsymbol{Y} = (y_1, \ldots y_n)$ a vector of Boolean outputs of the system. Boolean functional synthesis requires us to generate a vector of Boolean functions $\boldsymbol{\Psi}(\boldsymbol{X}) = \big(\psi_1(\boldsymbol{X}), \ldots \psi_n(\boldsymbol{X})\big)$ such that $\forall \boldsymbol{X}\big(\exists \boldsymbol{Y}\varphi(\boldsymbol{X}, \boldsymbol{Y}) \Leftrightarrow \varphi(\boldsymbol{X}, \boldsymbol{\Psi}(\boldsymbol{X}))\big)$.

Authors names are in alphabetical order of last names

For each $i \in \{1, \dots n\}$, the function $\psi_i(\boldsymbol{X})$ is called a Skolem function for y_i in $\varphi(\boldsymbol{X}, \boldsymbol{Y})$, and $\boldsymbol{\Psi}(\boldsymbol{X})$ is called a Skolem function vector.

There are several interesting applications of Boolean functional synthesis, including automated program synthesis, circuit repair and debugging, cryptanalysis and the like [2,6–10]. This has motivated researchers to develop novel algorithms for solving increasingly larger and more complex synthesis benchmarks [11–19]. Each such algorithm generates *a single Skolem function vector* for a given relational specification, thereby providing an implementation of the system. However, there may be many alternative function vectors that also serve as Skolem function vectors for the same specification. Some of these may yield system implementations that are more "desirable" than those obtained from other Skolem function vectors, when non-functional metrics like size of program/circuit needed for implementation, ease of understandability etc. are considered. Therefore, having a tool output a single Skolem function vector (chosen by the tool, without any user agency in the choice) can be restrictive in terms of implementation choices available to the user.

One way to address the above problem is to use a knowledge compilation approach, i.e. to compile the specification to a *special normal form* from which it is relatively easy to use downstream logic synthesis tools to generate any Skolem function vector optimizing user-specified criteria. Unfortunately, earlier work on knowledge compilation for Boolean functional synthesis [13,14,20] does not allow us to do this easily. They simply allow efficient synthesis of one (among possibly many) Skolem function vector from the compiled representation. Moreover, the user has no agency in choosing which Skolem function vector is synthesized; all choices are made implicitly deep inside heuristics of the compilation algorithms. For example, if we compile a relational specification to wDNNF [14] or SynNNF [13], the only guarantee we have is that the so-called GACKS Skolem functions (see [14]) can be efficiently synthesized from the compiled forms. But what if these functions are not the user's preferred choice of Skolem functions for an application? Unfortunately, not much can be done if we compile the specification to wDNNF or SynNNF. Similarly, the compilation approach proposed in [20] allows efficient synthesis of Skolem functions of yet another form, but even here, the user hardly has any agency in choosing which (among many alternative) Skolem function vectors is actually output. Existing algorithms therefore effectively restrict the *semantic choice* of Skolem functions with hardly any way for the user to influence this choice. Once the semantic choice has been made by the compiler, the only agency the user has is in *optimizing the implementation of this semantic choice*. We believe the inability of existing compilation approaches to allow the user semantic choice of Skolem functions is a limiting factor in practical usage of these works. In this paper, we take a first step towards remedying this problem.

The central question we ask in this paper is: *Can we compile a Boolean relational specification to a representation that does not restrict the semantic choice of Skolem functions, and yet allows easy deployment of downstream logic synthesis tools to obtain Skolem functions customized to user-provided criteria?* Our main result is an affirmative answer to this question. We also design and implement an algorithm that compiles a given specification in negation normal form to such a representation form, We emphasize that our goal in this paper is not to identify specific optimization criteria or to synthesize Skolem functions that optimize some specific criteria. Instead, we focus on developing a representation that makes it possible to use downstream logic optimization tools to synthesize Skolem functions satisfying user-provided criteria. Our experiments show that our approach is competitive performance-wise to earlier approaches that severely restrict the semantic choice of Skolem functions.

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

- We formalize the problem of symbolically and compactly representing all Skolem function vectors for a Boolean relational specification in such a way that it is amenable to downstream optimization by logic synthesis tools.
- We propose a candidate for this representation as a set of pairs of functions, one for each output, which we call the *Skolem basis vector*. We show that the Skolem basis vector is guaranteed to exist for any specification and is unique with respect to an ordering of the output variables.
- For single-output specifications, we show that the Skolem basis vector can be computed easily, as a pair of (semantically unique) Boolean functions. For multi-output specifications, we relate the problem of generating Skolem basis vector to the question of performing efficient quantification of outputs.
- We investigate two properties, namely *unateness and conflict-freeness of outputs*, that permit efficient quantification of outputs. This, in turn, allows a Skolem basis vector to be generated in polynomial time in special cases.
- We present a novel counterexample-guided algorithm for transforming a specification to one where a designated output variable is conflict-free. We call this process *rectification* of the output.
- We present an overall algorithm that takes a specification and generates a Skolem basis vector by successively rendering outputs unate or conflict-free.
- We present a tool implementing our algorithm, and report experimental results on a suite of publicly available benchmarks.

Related Work. In knowledge compilation, the general goal is to represent a problem specification in a form that allows specific questions to be answered efficiently (see e.g., [21–23]). In [22,24], representation forms for Boolean functions were proposed that allow efficient enumeration of all satisfying assignments of the function. However, this idea cannot be easily extended to enumerate Skolem functions, since the space of functions is doubly exponentially large in the number of variables. For Boolean functional synthesis, [13,20,25,26] provide normal forms and present compilers that render synthesis of a single Skolem function vector easy. However, they do not provide the user any agency in choosing the Skolem function vector. In fact, the optimizations used in [13] preclude generation of all Skolem function vectors for reasons of efficiency. In the current work, our focus is on symbolically representing the space of all Skolem function vectors, without necessarily converting the given specification to a semantically equivalent one in special normal form. Thus, the problem addressed in this paper is technically different from those addressed in [13,20,25,26]. Nevertheless, our work can be viewed as knowledge representation for all Skolem functions.

2 A Motivating Example

We start with a simple example that illustrates some of the problems we wish to address. Suppose we are designing a memoryless arbiter that must arbitrate requests from three users for a shared resource. Let the arbiter inputs be Boolean variables r_1, r_2, r_3, where r_i is true iff there is a request from user i. Let the corresponding arbiter outputs be g_1, g_2, g_3, where g_i is true iff access is granted to user i. We want the arbiter to satisfy the following properties: (a) at most one user must be granted access at a time, (b) if some user has requested access, some user must be granted access, and (c) a user should be granted access only if she has requested. The above properties can be encoded as a specification $\varphi \equiv \varphi_1 \wedge \varphi_2 \wedge \varphi_3$, where $\varphi_1 \equiv \big(g_1 \Rightarrow \neg(g_2 \vee g_3)\big) \wedge \big(g_2 \Rightarrow \neg(g_1 \vee g_3)\big) \wedge \big(g_3 \Rightarrow \neg(g_1 \vee g_2)\big)$, $\varphi_2 \equiv (r_1 \vee r_2 \vee r_3) \Rightarrow (g_1 \vee g_2 \vee g_3)$, and $\varphi_3 \equiv (g_1 \Rightarrow r_1) \wedge (g_2 \Rightarrow r_2) \wedge (g_3 \Rightarrow r_3)$.

It turns out that there are many different Skolem function vectors $\boldsymbol{\Psi} = (\psi_1, \psi_2, \psi_3)$ for the above specification, where each ψ_i gives a Skolem function for g_i. We ran two state-of-the-art Boolean functional synthesis tools, viz. Manthan2 [17] and BFSS [14], on this specification. BFSS required us to also specify a linear order of outputs (we will shortly see why), and we used $g_1 \prec g_2 \prec g_3$. Both tools solved the problem in no time, and each reported a Skolem function vector *without any room for the user to influence the choice of Skolem functions.* Specifically, the Skolem functions returned by Manthan2 can be represented by the And-Inverter Graph (AIG) shown in Fig. 1a. Here, each circle represents a two-input AND gate, and each dotted (resp. solid) edge represents a connection with (resp. without) logical negation. Thus, the Skolem functions are: $\psi_2 \equiv r_2 \wedge \neg r_1 \wedge \neg r_3$, $\psi_1 \equiv r_1 \wedge \neg r_3 \wedge \neg g_2$ and $\psi_3 \equiv r_3 \wedge \neg g_1 \wedge \neg g_2$. Running BFSS on the same specification yields Skolem functions represented by the AIG in Fig. 1c. Here, $\psi_3 \equiv r_3 \wedge \neg r_1 \wedge \neg r_2$, $\psi_2 \equiv r_2 \wedge \neg g_3$ and $\psi_1 \equiv r_1 \wedge \neg g_2 \wedge \neg g_3$.

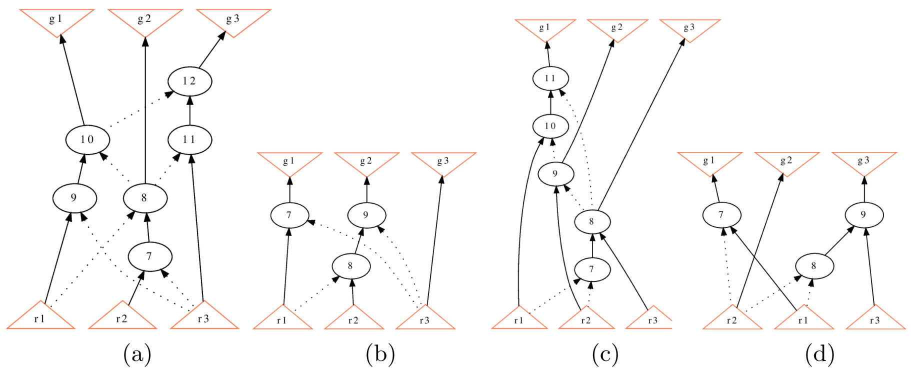

Fig. 1. Unoptimized and optimized AIGs of Skolem functions

Are the Skolem functions generated by the two tools in their simplest forms, and did they miss out some possibilities of optimization? To answer this, we used a widely used logic optimization tool, viz. abc [27], to simplify the two AIGs using commands to minimize the AND gate count and to balance lengths of paths in the AIGs. The resulting simplified AIGs are shown in Fig. 1b (obtained from Fig. 1a) and Fig. 1d (obtained from Fig. 1c). Thus, Manthan2's solution is equivalent to $\psi_3 \equiv r_3$, $\psi_2 \equiv r_2 \wedge \neg r_1 \wedge \neg r_3$, $\psi_1 \equiv r_1 \wedge \neg r_3$, while BFSS' solution is equivalent to $\psi_2 \equiv r_2$, $\psi_1 \equiv r_1 \wedge \neg r_2$, $\psi_3 \equiv r_3 \wedge \neg r_1 \wedge \neg r_2$. Note that the two solutions are semantically equivalent modulo permutaton of indices (although this wasn't obvious prior to optimization).

There are some important take-aways from this simple experiment. First, neither Manthan2 nor BFSS gave the user any agency in the semantic choice of the synthesized Skolem functions. The use of the abc tool with user-provided optimization criteria at the end simply gave us choice of implementation for the Skolem functions already determined by each tool. Significantly, there are choices of Skolem function vectors, viz. $\psi_1 \equiv r_1 \wedge (\neg r_2 \vee \neg r_3)$, $\psi_2 \equiv r_2 \wedge (\neg r_1 \vee r_3)$, $\psi_3 \equiv (\neg r_1 \wedge \neg r_2 \wedge r_3)$, that are *ignored* by both Manthan2 and BFSS (and by other tools like CADET [11]). This can lead to ignoring "better" Skolem function vectors in general. The user's criteria for desirability of Skolem functions may differ from one problem instance to another, and may be completely different from what is hard-coded in the innards of a tool like Manthan2/BFSS. For example, the new Skolem function vector considered above admits an AIG representation in which input-to-output shortest (resp. longest) path lengths are equal across all outputs. This may indeed be a desirable feature in some application where variability of output delays matters. However, there is currently no way to influence BFSS/Manthan2 to arrive at Skolem functions optimized per such criteria.

The above example also illustrates the important role played by logic optimization in obtaining efficient implementations of Skolem functions generated by state-of-the-art synthesis tools. However, using logic optimization as a post-processor can only provide a better implementation of already chosen (semantically) Skolem functions. Fortunately, more than five decades of research in logic optimization has resulted in mature (even commercial) tools that can do much more than just implementation optimization. Specifically, don't-care based optimizations [28] can search within a specified space of (semantically distinct) functions to choose one that is optimized according to a given user criteria. Such a choice involves a combined optimization across semantic and implementation choices. Given this capability of logic optimizers, and their indispensable use in synthesis flows, we posit that logic optimizers are the right engines to choose between alternative semantic choices of Skolem functions, in addition to optimizing their implementation. Of course, this requires specifying the semantic space of all (Skolem) functions in a form that can be easily processed by logic optimizers. State-of-the-art logic optimizers already allow specifying a family of functions using *on-sets* and *don't-care sets* [29]. Therefore, we propose to use this representation for representing the space of Skolem functions as well.

Before presenting the details of on-sets and don't-care sets for Skolem functions in our example, we note that Skolem functions for different outputs cannot be chosen independently in general. For example, $\psi_3 \equiv r_3$ is generated by Manthan2, and $\psi_2 \equiv r_2$ is generated by BFSS. However, there is no Skolem function vector with $\psi_2 \equiv r_2$ and $\psi_3 \equiv r_3$), since this would lead to $g_2 = g_3 = 1$ when $r_2 = r_3 = 1$. Therefore, any representation of the semantic space of all Skolem function vectors *must necessarily* take into account dependence between Skolem functions for different outputs. One way to achieve this is to impose a linear order on the outputs, and to represent the set of Skolem functions for an output in terms of Skolem functions for preceding (in the order) outputs. With this approach, the semantic space of Skolem functions for each output can be expressed by two functions: one representing the set of assignments for which every Skolem function in the represented space must evaluate to 1 (i.e. on-set), and the other representing assignments for which it is ok for a Skolem function to evaluate to either 0 or 1 (i.e. don't-care set).

The above representation is analogous to representing vector spaces using a small set of mutually orthogonal basis vectors, where every vector in the space can be expressed as a linear combination of these basis vectors. In a similar manner, let A denote the on-set of a family of Skolem functions, and B denote the don't-care set for the same family. Let $\mathsf{GenImpl}(B)$ denote the set of all *generalized implicants* of B, i.e. all formulas ν such that $\nu \Rightarrow B$. Every Skolem function in the represented space can then be obtained (modulo semantic equivalence) as $A \vee \nu$ where $\nu \in \mathsf{GenImpl}(B)$. Specifically, for our example, with $g_1 \prec g_2 \prec g_3$ of outpus (same as that given to BFSS), we have $A_1 \equiv (\neg r_3 \wedge \neg r_2 \wedge r_1)$, $B_1 \equiv (r_3 \vee r_2) \wedge r_1$, $A_2 \equiv (\neg r_3 \wedge r_2 \wedge \neg g_1)$, $B_2 \equiv r_3 \wedge r_2 \wedge \neg g_1$, $A_3 \equiv r_3 \wedge \neg g_2 \wedge \neg g_1$, $B_3 = 0$. The Karnaugh-maps shown below depict how the space of all Skolem function vectors can be visualized in terms of A_i and B_i. To obtain a specific Skolem function vector, we must place a 1 in each A_i-cell, choose a subset of the B_i cells and place 1's in those cells and 0's in the balance B_i cells. Each such choice provides a semantically distinct Skolem function vector, and every Skolem function vector corresponds to one such choice. Specifically, the Skolem function vector missed by Manthan2/BFSS can now be easily obtained by choosing the red and blue B_1 cells and the teal B_2 cell to be 1 in the Karnaugh-maps. Similarly, Manthan2's solution is obtained by choosing the blue B_1 cell and teal B_2 cell to be 1, and BFSS' solution is obtained by choosing the red B_1 cell and teal B_2 cell to be 1. Allowing a logic optimizer to optimize Skolem functions with the spaces represented by $(A_1, B_1, A_2, B_2, A_3, B_3)$ therefore makes it possible to synthesize each of these Skolem function vectors. This motivates compiling a given specification into an (A_i, B_i) pair for the Skolem functions for each output y_i.

$r_2r_3 \rightarrow$ / $r_1 \downarrow$	00	01	11	10
0	0	0	0	0
1	A_1	B_1	B_1	B_1

Space of Sk fns for g_1

$r_2r_3 \rightarrow$ / $g_1 \downarrow$	00	01	11	10
0	0	0	B_2	A_2
1	0	0	0	0

Space of Sk fns for g_2

$g_2r_3 \rightarrow$ / $g_1 \downarrow$	00	01	11	10
0	0	A_3	0	0
1	0	0	0	0

Space of Sk fns for g_3

3 Preliminaries and Notation

Let $\boldsymbol{Z} = (z_1, \ldots, z_n)$ be a vector of Boolean variables. A *literal* is a variable (z_i) or its complement $(\neg z_i)$, a *clause* is a disjunction of literals and a *cube* is a conjunction of literals. For $1 \leq i \leq j \leq n$, we use $\boldsymbol{Z}_i^j$ to denote the *slice* $(z_i, \ldots z_j)$ of the vector $\boldsymbol{Z}$. An n-input Boolean function is a mapping from $\{0,1\}^n$ to $\{0,1\}$. A Boolean formula $\varphi(\boldsymbol{Z})$ is a syntactic object whose semantics is given by a mapping from $\{0,1\}^n$ to $\{0,1\}$. Thus, every Boolean formula represents a unique Boolean function, and every Boolean function can be represented by a (not necessarily unique) Boolean formula. Henceforth, we refer to Boolean formulas and Boolean functions interchangeably.

The *support* of $\varphi(\boldsymbol{Z})$, denoted $\mathsf{sup}(\varphi)$, is the set of variables in $\boldsymbol{Z}$. For ease of exposition, we will abuse notation and use $\boldsymbol{Z}$ to denote either a vector or the underlying set of elements, depending on the context. A complete (resp. partial) *assignment* π for $\boldsymbol{Z}$ is a complete (resp. partial) mapping from $\boldsymbol{Z}$ to $\{0,1\}$. The value of variable z_i assigned by π is denoted $\pi[z_i]$. A complete assignment π of $\boldsymbol{Z}$ is a *satisfying assignment* for $\varphi(\boldsymbol{Z})$ if the Boolean function represented by φ evaluates to 1 when all variables in $\mathsf{sup}(\varphi)$ are assigned values given by π. In this case, we say that $\pi \models F$. A formula $\varphi(\boldsymbol{Z})$ is *satisfiable* if it has at least one satisfying assignment; otherwise it is *unsatisfiable*. We say that two formulas on n variables are *equivalent* if they represent the same semantic mapping from $\{0,1\}^n$ to $\{0,1\}$. Given Boolean formulas φ and α with $z_i \in \mathsf{sup}(\varphi)$, we use $\varphi[z_i \mapsto \alpha]$ to denote the formula obtained by substituting α for every occurrence of z_i in φ. We use $\varphi\big|_{z_i=1}$ (resp. $\varphi\big|_{z_i=0}$) to denote the formula obtained by setting z_i to 1 (resp. 0) in the formula $\varphi(\boldsymbol{Z})$. The resulting formulas are also called positive (resp. negative) co-factors of φ w.r.t. z_i. For notational convenience, we use $\varphi\big|_{\pi}$ to denote the formula obtained by repeatedly co-factoring φ using the (possibly partial) assignment of variables given by π. As discussed in Sect. 2, we say that a function $\varphi'(\boldsymbol{Z})$ is a *generalized implicant* of $\varphi(\boldsymbol{Z})$ if $\varphi'(\boldsymbol{Z}) \Rightarrow \varphi(\boldsymbol{Z})$. This generalizes the notion of implicants used in the literature, which are restricted to be cubes. The set of all generalized implication of φ is denoted $\mathsf{GenImpl}(\varphi)$.

A Boolean formula $\varphi(\boldsymbol{Z})$ can be represented as a circuit or a Directed Acyclic Graph (DAG) consisting of $\neg$, $\wedge$ and $\vee$ gates, with literals at leaves. Further, it can be converted to a semantically equivalent formula in *Negation Normal Form (NNF)*, i.e., with no $\neg$-labelled internal nodes, in time linear in the size of the circuit. We consider formulas to be given in NNF unless mentioned otherwise, and interchangeably refer to a Boolean formula and the circuit representing it. If an NNF formula in *Conjunctive Normal Form (CNF)*, i.e., as conjunction of clauses, is unsatisfiable, then there is a subset of its clauses whose conjunction is unsatisfiable. This set is called its *unsatisfiable core*, and a *minimal unsatisfiable core* is one without any proper subset that is also an unsatisfiable core.

The *Boolean functional synthesis* problem, and notions of *Skolem functions* and *Skolem function vectors* have already been defined in Sect. 1. Let $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ be a Boolean relational specification over inputs $\boldsymbol{X}$ and outputs $\boldsymbol{Y}$. A commonly used approach, adopted by several Boolean functional synthesis algorithms [6, 14–16], works as follows. Without loss of generality, let $y_1 \prec \cdots \prec y_n$ be a linear ordering of the outputs in $\boldsymbol{Y}$. We first define a set of derived specifications $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_i^n)$ for all $i \in \{1, \dots n\}$, where $\varphi^{(i)} \Leftrightarrow \exists \boldsymbol{Y}_1^{i-1}\, \varphi(\boldsymbol{X}, \boldsymbol{Y})$. Next, for each $i \in \{1, \dots n\}$, we find a Skolem function for y_i from the derived specification $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_i^n)$, by treating y_i as the sole output and all of $\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n$ as inputs in $\varphi^{(i)}$. Let $\psi_i(\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n)$ denote the Skolem function for y_i thus obtained. Finally, we substitute the Skolem functions $\psi_{i+1}, \dots \psi_n$ for $y_{i+1}, \dots y_n$ respectively in the Skolem function ψ_i obtained above. This gives a Skolem function for y_i only in terms of $\boldsymbol{X}$. By repeating the above process for all i in decreasing order from $n-1$ to 1, we obtain a Skolem function vector for φ.

4 A New Knowledge Representation for Skolem Functions

We start with a key definition that is motivated by the desire to represent the entire space of Skolem functions arising from a specification compactly, and in a form that is easily amenable to well-established logic synthesis and optimization workflows. Recall from Sect. 2 that for a multi-output specification, Skolem functions for different outputs may be dependent on each other. Hence, the set of Skolem function vectors cannot be expressed as a Cartesian product of sets of Skolem functions for individual outputs. Instead, we impose a linear order on the outputs, and express the Skolem function for one output in terms of the inputs and other outputs that precede it in the order. Such a linear order may be automatically generated, user-provided, or even generated with guidance from the user, e.g., if the user provides a partial order on the outputs. We assume the availability of such an order $\prec$ in the definition below.

Definition 1. *Let $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ be a specification over a linearly ordered set of outputs $\boldsymbol{Y} = \{y_1, \dots, y_n\}$. We say that output y_i has a Skolem basis in φ if there exists a pair of functions (A_i, B_i) over $\boldsymbol{X} \cup \boldsymbol{Y}_{i+1}^n$ such that*

1. *$A_i \wedge B_i$ is unsatisfiable, and*
2. *any Skolem function $\psi_i(\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n)$ for y_i in the derived specification $\varphi^{(i)}$ can be written as $\psi_i \equiv A_i \vee g$ for some $g \in \mathsf{GenImpl}(B_i)$.*

We call the vector of pairs $\langle (A_i, B_i) \rangle_{1 \leq i \leq n}$ the Skolem basis vector for φ wrt $\prec$.

The Skolem basis vector can be seen as a succinct representation of the Skolem function space, i.e., the set of all Skolem function vectors of φ. A natural question that arises at this point is: *Given a specification φ and order $\prec$ of outputs, does there always exist a Skolem basis for φ wrt $\prec$?* Fortunately, as we show in this paper, the answer is a resounding "Yes". Not only that, the Skolem basis for a given φ and $\prec$ is unique upto semantic equivalence of the basis functions. It is important to note that not every set of functions can be represented using just two basis functions. This is easy to see via a counting argument: the number of sets of Boolean functions over m inputs is $2^{2^{2^m}}$. However, the number of sets that admit a Skolem basis is (loosely) upper bounded by $2^{2 \cdot 2^m}$. Skolem functions are therefore special, since we show that the space of all Skolem functions for every output in every specification always admits representation by two basis functions, regardless of the order $\prec$. Interestingly, though the definition of Skolem basis vector needs us to specify an order $\prec$ on the outputs, somewhat surprisingly, the Skolem function space itself does not depend on the order.

Proposition 1. *Suppose $\boldsymbol{\Psi}$ is a Skolem function vector for the outputs $\boldsymbol{Y}$ in terms of inputs $\boldsymbol{X}$ in φ. Then, for any order $\prec$, $\boldsymbol{\Psi}$ can be generated using the Skolem basis vector of φ wrt $\prec$, and then substituting, for each $i \in \{1, \ldots n\}$, the Skolem functions ψ_j for y_j where $i < j \leq n$, in the Skolem function for ψ_i.*

Proof Sketch: With ordering $y_1 \prec y_2 \prec \ldots y_n$, let $\langle (A_i, B_i) \rangle$ be the corresponding Skolem basis vector. The support of A_n, B_n are only the inputs $\boldsymbol{X}$, while the support of A_i, B_i (for $i > 1$) are $\boldsymbol{X} \cup \{y_{i+1}, ...y_n\}$. Let $\boldsymbol{\Psi} = (\psi_1, \ldots \psi_n)$ be an arbitrary Skolem function vector, where each ψ_i is a function of $\boldsymbol{X}$. By definition of Skolem basis, since ψ_n is a Skolem function for y_n, it can be obtained from A_n and B_n (each of which has support $\boldsymbol{X}$). Now consider ψ_i for $1 \leq i < n$. By definition of Skolem basis, every Skolem function for y_i in terms of $\boldsymbol{X} \cup \{y_{i+1}, ...y_n\}$ can be obtained from A_i and B_i. In particular, if we set y_{i+1} to ψ_{i+1} and so on until y_n to ψ_n, every Skolem function for y_i in terms of $\boldsymbol{X}$ can be obtained from A_i and B_i. $\square$

Another interesting property about Skolem basis vector is that, when it exists, it is unique. Later we will show (constructively) that it always exists and hence we would have also constructed the unique one.

Proposition 2. *For any y_i in φ, its Skolem basis, when it exists, is unique.*

Proof. Fix i. Let S be the set of all Skolem functions for y_i in $\varphi^{(i)}$. From Definition 1, we know that for all $f \in S$, $A_i \Rightarrow f$. Hence, $A_i \Rightarrow \bigwedge_{f \in S} f$. However, we also know that $A_i \in S$ (corresponds to choosing the generalized implicant 0 from $\mathsf{GenImpl}(B_i)$). Therefore, $\left(\bigwedge_{f \in S} f\right) \Rightarrow A_i$. It follows from the two implications that $A \Leftrightarrow \bigwedge_{f \in S} f$.

In a similar manner, Definition 1 implies that for all $f \in S$, $f \Rightarrow A_i \vee B_i$. Hence $\left(\bigvee_{f \in S} f\right) \Rightarrow A_i \vee B_i$. However, we know that $A_i \vee B_i \in S$ (corresponds to choosing the generalized implicant B from $\mathsf{GenImpl}(B)$). Therefore, $A_i \vee B_i \Rightarrow \bigvee_{f \in S} f$. It follows from the two implications that $B_i \Leftrightarrow \bigvee_{f \in S} f$. $\square$

Finally, we explain how our new representation of Skolem functions using a Skolem basis vector naturally lends itself to easy processing by downstream logic synthesis and optimization tools. Thus, a Skolem basis vector is not just an arbitrary way to represent the space of all Skolem function vectors; instead, it is strongly motivated by the way modern logic synthesis and optimization tools work to search the semantic space of partially specified functions (i.e. functions specified with on-sets and don't-care sets). Specifically, in logic synthesis and optimization parlance [29], A_i is the *on-set* and B_i is the *don't-care set* for Skolem functions for y_i in φ. In other words, A_i describes all assignments for which every Skolem function for y_i must evaluate to 1 while B_i describes those assignments on which a Skolem function can evaluate to either 1 or 0 without violating the requirement of being a Skolem function for y_i in φ. Thus, every semantically distinct Skolem function for y_i in φ can be obtained by choosing a distinct subset of satisfying assignments of B_i and choosing the Skolem function to evaluate on this subset of assignments in addition to those determined by A_i. Indeed, state-of-the-art logic synthesis and optimization tools (such as abc [27]) use on-sets and don't care sets expressed as Boolean functions to represent the space of all realizations of a partially specified function. The don't cares are then used to optimize the semantic and implementation choices when choosing the optimal realization of such a partially specified function, as per user provided criteria like area, gate count, delay, power consumption, balance of delays across paths etc. Indeed, the following guarantee follows rather trivially from Proposition 1.

Proposition 3. *Suppose we have access to a logic optimization tool that finds the optimal semantic and implementation choice of a partially specified function as per user criteria. Using this tool on the Skolem basis vector of φ wrt $\prec$ yields the optimal choice among all Skolem functions, where optimality of Skolem function for y_i is conditioned on the choice of Skolem functions for y_j, for $1 \leq j < i$.*

Having defined and motivated the Skolem basis vector as our new knowledge representation, in the rest of the paper we will show how it can actually be computed, *in theory and in practice.*

5 Towards Synthesizing the Skolem Basis Vector

The Single Output Case: First, we consider the case of a singleton output and show that here the existence of Skolem basis is easy to establish, and the basis is also easy to compute.

Theorem 1. *For a single-output specification $\varphi(\boldsymbol{X}, y)$, the Skolem basis for y in φ is given by $A \equiv \varphi(\boldsymbol{X}, 1) \wedge \neg\varphi(\boldsymbol{X}, 0)$ and $B \equiv \varphi(\boldsymbol{X}, 1) \leftrightarrow \varphi(\boldsymbol{X}, 0)$. Thus, in this case, the Skolem basis vector for φ can be computed in time/space linear in size of the circuit representing φ.*

Proof. Let $2^{|\boldsymbol{X}|}$ denote the set of all complete assignments π of $\boldsymbol{X}$. Define $S_1 = \{\pi \mid \pi \in 2^{|\boldsymbol{X}|},\ \pi \models \varphi(\boldsymbol{X}, 1)\}$ and $S_0 = \{\pi \mid \pi \in 2^{|\boldsymbol{X}|},\ \pi \models \varphi(\boldsymbol{X}, 0)\}$. By definition of S_0 and S_1, (with $\overline{S_i}$ denoting complement of set S_i), we have:

- $\pi \in S_1 \cup S_0$ iff $\pi \models \exists y\, \varphi(\boldsymbol{X}, y)$.
- $\pi \in S_1 \cap S_0$ iff $\pi \models \forall y\, \varphi(\boldsymbol{X}, y)$.
- $\pi \in \overline{S_1} \cap \overline{S_0}$ iff $\pi \models \forall y\, \neg\varphi(\boldsymbol{X}, y)$.
- For every $\pi \in S_1 \cap \overline{S_0}$, the only value of y that makes $\varphi(\pi, y)$ true is 1.
- For every $\pi \in S_0 \cap \overline{S_1}$, the only value of y that makes $\varphi(\pi, y)$ true is 0.

Now let $\psi(\boldsymbol{X})$ be an arbitrary Skolem function for y in $\varphi(\boldsymbol{X})$. Recall that by definition a Skolem function satisfies $\forall \boldsymbol{X}\, \big(\exists y\, \varphi(\boldsymbol{X}, y) \Leftrightarrow \varphi(\boldsymbol{X}, \psi(\boldsymbol{X}))\big)$. It then follows from the above observations that if $\pi \in S_1 \cap \overline{S_0}$, $\psi(\pi)$ must evaluate to

1. Similarly, if $\pi \in (S_1 \cap S_0) \cup (\overline{S_1} \cap \overline{S_0})$, it makes no difference whether $\psi(\pi)$ evaluates to 0 or 1. Finally, if $\pi \in S_0 \cap \overline{S_1}$, $\psi(\pi)$ must evaluate to 0. Since ψ was an arbitrary Skolem function for y in φ, we infer that the Skolem basis for $\mathsf{AllSk}(\varphi)$ is (A, B), where $A \equiv \varphi(\boldsymbol{X}, 1) \wedge \neg\varphi(\boldsymbol{X}, 0)$ represents the set $S_1 \cap \overline{S_0}$, and $B \equiv \big(\varphi(\boldsymbol{X}, 0) \Leftrightarrow \varphi(\boldsymbol{X}, 1)\big)$ represents the set $(S_1 \cap S_0) \cup (\overline{S1} \cap \overline{S_0})$. □

We next consider the multiple output case, where our strategy (as done usually for Skolem *function* synthesis) is to reduce to the one-output case above.

Multiple Outputs and Existential Quantification: When we have multiple outputs, from the definition of Skolem basis vector (Definition 1), it follows that the problem reduces to the single output case, if we can compute the derived specifications $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n)$. Unfortunately, computing $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_i^n)$ cannot always be done efficiently, even when $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and the order $\prec$ on $\boldsymbol{Y}$ are given. We compute $\varphi^{(i)}$ from a given $\varphi^{(i-1)}$, where the variable y_i to be quantified is either chosen on-the-fly (giving a dynamic computation of $\prec$) or determined as per a statically provided order. Since $\varphi^{(i+1)} \;\Leftrightarrow\; \exists \boldsymbol{Y}_1^i\, \varphi \;\;\Leftrightarrow\; \exists y_i \varphi^{(i)}$ for all $i \in \{1, \ldots n-1\}$, we first consider how a single output variable can be quantified from a derived specification.

The conceptually simplest way to compute $\exists y_i\, \varphi^{(i)}$ is as $\varphi^{(i)}\big|_{y_i=1} \vee \varphi^{(i)}\big|_{y_i=0}$. Unfortunately, this doubles the size of the circuit representation. An alternative is to find a Skolem function, say ψ_i, for y_i in $\varphi^{(i)}$, and then use $\varphi^{(i)}[y_i \mapsto \psi_i]$. This works well when ψ_i can be represented compactly. However, an NNF representation of ψ_i can be as large as that of $\varphi^{(i)}$ (e.g. if $\psi_i \equiv \varphi^{(i)}\big|_{y_i=1}$), in which case we may double the circuit size. We therefore ask *if it is possible to compute $\exists y_i\, \varphi^{(i)}$ by simply substituting a constant (not necessarily a Skolem function) for y_i in an NNF formula of almost the same size as $\varphi^{(i)}$*. It turns out that this is possible in two practically relevant cases. In other cases, we transform the circuit to permit such constant substitutions. For notational convenience, in the rest of this section, we omit i and use y and φ for y_i and $\varphi^{(i)}$.

The Case of Unates: A variable y is *positive (resp. negative) unate* in φ if $\varphi\big|_{y=0} \Rightarrow \varphi\big|_{y=1}$ (resp. $\varphi\big|_{y=1} \Rightarrow \varphi\big|_{y=0}$). A variable is *unate* in φ if it is either positive or negative unate in φ. Then, we have: easily proved.

Lemma 1. *If y is positive unate in φ, then $\exists y\; \varphi \Leftrightarrow \varphi\big|_{y=1}$. Similarly, if y is negative unate in φ, then $\exists y\; \varphi \Leftrightarrow \varphi\big|_{y=0}$.*

Proof. The proof immediately from the definition of positive and negative unateness, and from the fact that $\exists y\, \varphi \Leftrightarrow \varphi\big|_{y=0} \vee \varphi\big|_{y=1}$. □

As an example, consider $\varphi \equiv (x \wedge (y_1 \vee y_2)) \vee (\neg x \wedge \neg y_2)$. Here, y_1 is positive unate in φ, but y_2 is not unate in φ. However, y_2 is negative unate in $\varphi\big|_{y_1=1}$, which by Lemma 1 is equivalent to $\exists y_1\, \varphi$. This shows that even if a variable is not unate to begin with, it may become unate after some variables are quantified. If we use the order $y_1 \prec y_2$ in our example, both $\exists y_1\, \varphi$ and $\exists y_1 \exists y_2\, \varphi$ can be computed by substituting for y_1 and y_2 in φ. This is however not true for $y_2 \prec y_1$.

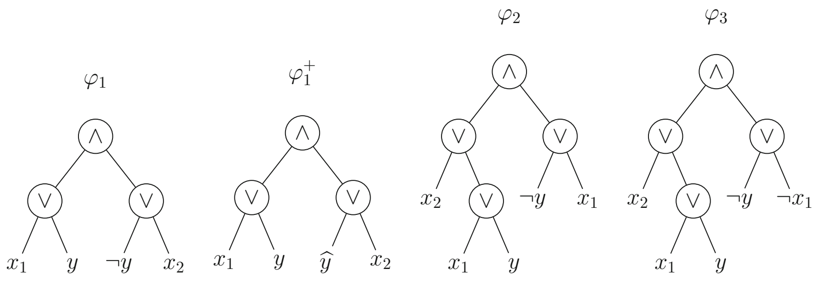

Fig. 2. NNF circuit representations of formula $\varphi_1, \varphi_1^+, \varphi_2, \varphi_3$.

In general, given a specification $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and a linear ordering $\prec$ of outputs, if each output y_i is unate in the derived specification $\varphi^{(i)} \equiv \exists \boldsymbol{Y}_1^{i-1} \varphi$, then we can apply Lemma 1, Definition 1 and Theorem 1 to synthesize the entire Skolem basis vector for φ w.r.t. $\prec$ efficiently. This also suggests a heuristic for finding a (partial) order on the outputs $\boldsymbol{Y}$. Specifically, given a derived specification $\varphi^{(i)}$, we try to find an output variable y in its support such that y is unate in $\varphi^{(i)}$. If such a variable exists, we use it as the next variable in the $\prec$ order, and obtain $\varphi^{(i+1)}$ by using Lemma 1 to compute $\exists y\, \varphi^{(i)}$. As our experiments show (see Sect. 7) and has also been observed elsewhere [14], this approach is surprisingly effective for finding Skolem functions for many benchmarks.

The Case of No Conflicts: Next, we consider another case where quantification can be achieved by substituing constants for variables.

Definition 2. *Let φ be an NNF formula, $y \in \mathsf{sup}(\varphi)$. Suppose we replace every occurence of $\neg y$ in φ by a fresh variable $\widehat{y}$ $(\widehat{y} \notin \mathsf{sup}(\varphi))$. The resulting formula is called the y-*positive form* of φ and is denoted φ^{+y}. The variable y is said to be* in conflict *in φ if there exists an assignment $\pi : \mathsf{sup}(\varphi) \setminus \{y\} \to \{0, 1\}$ such that $\varphi^{+y}\big|_\pi \Leftrightarrow y \wedge \widehat{y}$. Otherwise, we say that y is* conflict-free *in φ^{+y}.*

The assignment π in the above definition is called a *counterexample to conflict-freeness of y in φ*. It is easy to see that both y and $\widehat{y}$ are positive unate in φ^{+y}. Henceforth, we use φ^+ instead of φ^{+y} when y is clear from the context.

We illustrate conflicts and conflict-freeness in Fig. 2. The y-positive form of φ_1 is shown as φ_1^+, where $\widehat{y}$ is a fresh variable. Clearly, y is in conflict in φ_1 since $\varphi_1^+\big|_\pi \Leftrightarrow y \wedge \widehat{y}$ for $\pi : x_1 \mapsto 0, x_2 \mapsto 0$. Similarly, y is in conflict in φ_2 (as seen with $\pi : x_1 \mapsto 0, x_2 \mapsto 0$). However, y is not in conflict in φ_3 as there is no assignment π of x_1, x_2 for which $\varphi_3^+\big|_\pi \Leftrightarrow y \wedge \widehat{y}$.

Lemma 2. *If y is conflict-free in φ, then $\exists y\, \varphi \Leftrightarrow \varphi^+\big|_{y=1,\widehat{y}=1}$.*

Proof. Since y is conflict-free in φ, it follows that $\varphi^+\big|_{y=1,\widehat{y}=1} \Rightarrow \big(\varphi^+\big|_{y=1,\widehat{y}=0} \vee \varphi^+\big|_{y=0,\widehat{y=1}}$. Since all internal nodes in φ^+ are labeled by either $\wedge$ or $\vee$, it also follows that y and $\widehat{y}$ are positive unate in φ^+. Therefore, $\big(\varphi^+\big|_{y=1,\widehat{y}=0} \vee$

$\varphi^+|_{y=0,\widehat{y}=1} \Rightarrow \varphi^+|_{y=1,\widehat{y}=1}$. The proof is completed by observing that by definition $\exists y\, \varphi \Leftrightarrow \left(\varphi|_{y=0} \vee \varphi|_{y=1}\right) \Leftrightarrow \left(\varphi^+|_{y=0,\widehat{y}=1} \vee \varphi^+|_{y=1,\widehat{y}=0}\right)$. $\square$

A notion similar to conflict as defined above was used in [13,20] for defining normal forms for synthesis. The difference is that unlike in [13,20], we do not require a pre-specified subset of the support to be set to 1 in the assignment π. To identify conflicts, we define a *conflict formula* $\kappa_{\varphi,y}$ as $\left(\varphi^+|_{y=1,\widehat{y}=1} \wedge \neg\varphi^+|_{y=1,\widehat{y}=0} \wedge \neg\varphi^+|_{y=0,\widehat{y}=1}\right)$. By Definition 2, y is conflict-free in φ iff $\kappa_{\varphi,y}$ is unsatisfiable.

Proposition 4. *For $1 \leq i \leq 4$, there exist φ_i with $y_i \in \sup(\varphi_i)$ s.t., (i) y_1 is neither unate nor conflict-free in φ_1, (ii) y_2 is unate but not conflict-free in φ_2, (iii) y_3 is conflict-free but not unate in φ_3, (iv) y_4 is unate, conflict-free in φ_4.*

The formulas $\varphi_1, \varphi_2, \varphi_3$ from Fig. 2 satisfy conditions (i), (ii) and (iii) respectively. For (iv), we consider $\varphi_4 \equiv x \wedge y$, in which y is unate and conflict-free. Lemmas 1, 2 and Proposition 4 show that both unateness and conflict-freeness are independently useful, and hence combining we directly obtain:

Theorem 2. *Given $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and a linear order $\prec$ on $\boldsymbol{Y}$, if y_i is either unate or conflict-free in $\varphi^{(i)}$ for all $i \in \{1, \dots n\}$, then we can effectively synthesize the Skolem basis vector in time linear in size of φ.*

We remark that the implications of Theorem 2 go beyond what can be achieved by earlier work on normal forms for synthesis [13,20]. Indeed, there are formulas that are neither in SynNNF nor SAUNF but for which Theorem 2 applies.

Finally, unateness is a semantic property; hence if y is not unate in φ, it is not unate in every μ such that $\varphi \Leftrightarrow \mu$. However, conflict-freeness has a representational aspect. If y is in conflict in φ, we can *always* find another NNF formula μ such that (i) $\mu \Leftrightarrow \varphi$, and (ii) y is conflict-free in μ. To see why, note that if $\mu \equiv (y \wedge \varphi|_{y=1}) \vee (\neg y \wedge \varphi|_{y=0})$, i.e. Shannon expansion of φ w.r.t. y, then $\mu \Leftrightarrow \varphi$ and y is conflict-free in μ. However, taking the Shannon expansion may not always be the best way to render an output conflict-free, as it often leads to blow-up in the size of the expanded formula. In the next section, we give a counterexample guided algorithm to obtain μ from φ and y, that works much more efficiently than Shannon expansion in practice.

6 Counterexample-Guided Rectification

Recall from the previous section that if y is in conflict in $\varphi(\boldsymbol{X}, \boldsymbol{Y})$, then there exists a counterexample (assignment) $\pi : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \rightarrow \{0, 1\}$ such that $\varphi^+|_\pi \Leftrightarrow y \wedge \widehat{y}$. In this section, we discuss how we can use such counterexamples to transform $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ to a specification $\mu(\boldsymbol{X}, \boldsymbol{Y})$ such that $\mu \Leftrightarrow \varphi$ and y is conflict-free in μ. We call such a transformation *rectification* of φ w.r.t y, and the resulting formula μ is said to be *rectified* w.r.t. y.

Lemma 3. *Let π be a counterexample to conflict-freeness of y in $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and let ξ be a formula satisfying (a) $\mathsf{sup}(\xi) \subseteq \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\}$, (b) $\varphi \Rightarrow \xi$, and (c)*

$\xi|_\pi$ *is unsatisfiable. Define* $\tau \equiv \varphi \wedge \xi$ *and let* τ^+ *denotes the positive form of* τ *w.r.t.* y*. Then the following hold: (i)* $\tau \Leftrightarrow \varphi$*, (ii)* π *is not a counterexample to conflict-freeness of* y *in* τ*, and (iii) every counterexample to conflict-freeness of* y *in* τ *is also a counterexample to conflict-freeness of* y *in* φ*.*

Algorithm 1 RectifyOneOutput($\varphi(\boldsymbol{X}, \boldsymbol{Y})$, y)

```
1: $\mu := \varphi$
2: repeat
3:     $res :=$ SatSolve($\kappa_{\mu,y}$) ▷ $\kappa_{\mu,y}$ is confict formula for $y$ in $\mu$
4:     if $res$ is SAT then
5:         Let $\pi : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \to \{0,1\}$ be a satisfying assignment of $\kappa_{\mu,y}$
6:         $\xi :=$ PartialRectifier($\mu$, $\pi$)
7:         $\mu := \mu \wedge \xi$
8: until $res$ is UNSAT ▷ All counterexample to conflict-freeness of $y$ in $\mu$ are removed
9: return $\mu$
```

Proof. Since $\varphi \Rightarrow \xi$, it follows that $\tau \Leftrightarrow \varphi \wedge \xi \Leftrightarrow \varphi$. This proves claim (i) of Lemma 3. Next, note that since π is a counterexample to conflict-freeness of y in φ, we must have $\varphi^+|_\pi \Leftrightarrow (y \wedge \widehat{y})$. Since ξ does not have y in its support, it follows that $\tau^+ \Leftrightarrow \varphi^+ \wedge \xi$. Therefore, $\tau^+|_\pi \Leftrightarrow \varphi^+|_\pi \wedge \xi|_\pi \Leftrightarrow (y \wedge \widehat{y}) \wedge \xi|_\pi$. However, from the premise of Lemma 3, we know that $\xi|_\pi$ is unsatisfiable. Hence $\tau^+|_\pi$ is false. Specifically, $\tau^+|_\pi \not\Leftrightarrow (y \wedge \widehat{y})$, and hence π is not a counterexample to conflict-freeness of y in τ. This proves claim (ii) of Lemma 3. Finally, let $\pi' : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \to \{0,1\}$ be a counterexample to conflict-freeness of y in τ. By definition, $\tau^+|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$. However, $\tau^+|_{\pi'} \Leftrightarrow \varphi^+|_{\pi'} \wedge \xi|_{\pi'}$. Since all variables in support of ξ are assigned by π', we must have $\xi|_{\pi'}$ being equivalent to either 0 or 1. If $\xi|_{\pi'}$ is 0, then $\tau^+|_{\pi'}$ must also be 0, a contradiction of $\tau^+|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$. Therefore, we must have $\xi|_{\pi'}$ equivalent to 1, and hence $\varphi^+|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$ for $\tau^+|_{\pi'}$ to be equivalent to $(y \wedge \widehat{y})$. It follows that π' must be a counterexample to conflict-freeness of y in φ. This proves claim (iii) of Lemma 3. □

Henceforth, we call a formula ξ satisfying conditions (a), (b) and (c) of Lemma 3 a *partial rectifier* of φ w.r.t. y. Given π, it is easy to find a partial rectifier.

Lemma 4. *For all* $v \in \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\}$*, let* $\ell_{v,\pi}$ *denote* v *if* $\pi[v] = 1$*, and* $\neg v$ *if* $\pi[v] = 0$*. Let* ξ_π *be* $\neg\big(\bigwedge_{v \in \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\}} l_{v,\pi}\big)$*. Then* ξ_π *satisfies conditions (a), (b) and (c) of Lemma 3.*

The proof follows immediately from the observations: (i) π is the only satisfying assignment of $\neg\xi_\pi$, and (ii) $\varphi|_\pi \Leftrightarrow \big(\varphi^+[\widehat{y} \mapsto \neg y]\big)\big|_\pi \Leftrightarrow (y \wedge \widehat{y})[\widehat{y} \mapsto \neg y] \Leftrightarrow 0$. Consequently, $\neg\xi_\pi \Rightarrow \neg\varphi$. Although Lemma 4 gives a partial rectifier, it prevents only the assignment π from being a counterexample to conflict-freeness of y in

τ. Later we will see a partial rectifier that prevents many more assignments from being counterexamples. For the time being, however, we assume that we have access to a procedure PartialRectifier that takes as inputs φ and π and outputs a partial rectifier that satisfies conditions (a), (b) and (c) of Lemma 3.

The above discussion suggests a simple algorithm, shown as Algorithm RectifyOneOutput below, for rectifying a specification φ w.r.t. an output y.

The algorithm first initializes a temporary formula μ to φ. It then invokes a propositional satisfiability (SAT) solver to obtain a satisfying assignment π of the conflict formula $\kappa_{\mu,y}$ (defined in Sect. 5 just before Proposition 4). The assignment π serves as a counterexample to conflict-freeness of y in μ, and is used to obtain a partial rectifier ξ of μ w.r.t. y. The formula μ is then updated by conjoining it with ξ. Lemma 3 guarantees that this gives a specification semantically equivalent to φ, while removing π from the set of counterexamples to conflict-freeness of y in μ. By repeating the process with the updated formula μ, all counterexamples to conflict-freeness of y in μ are eventually removed.

Theorem 3. *Algorithm* RectifyOneOutput *always terminates with a formula μ s.t. $\mu \Leftrightarrow \varphi$ and y is conflict-free in μ.*

Proof. The following inductive invariants hold at end of every iteration of the loop in lines 2–8, thanks to Lemma 3: (i) $\mu \Leftrightarrow \varphi$, (ii) the set of counterexamples to conflict-freeness of y in μ has strictly fewer elements than at the start of the iteration. Since the set of counterexamples is finite (at most $2^{|\boldsymbol{X}|+|\boldsymbol{Y}|-1}$ elements), eventually this set must become empty. By definition of the conflict formula, $\kappa_{\mu,y}$ must be unsatisfiable when this happens. Hence, the algorithm eventually exits the loop in lines 2–8 and terminates. Since there are no counterexamples to conflict-freeness of y in μ on termination, y is indeed conflict-free in μ. $\square$

Rectification by Counterexample Generalization: The idea of counterexample generalization is best illustrated by an example. Consider the specification $\varphi(\boldsymbol{X}, y) \equiv \big((x_1 \wedge x_2) \vee ((x_2 \wedge x_3) \vee y)\big) \wedge \big(\neg y \vee (\neg x_3 \wedge x_4)\big)$, wherein y is in conflict. To see why this is so, consider φ^{+y} (henceforth called φ^{+}) represented as a NNF circuit in Fig. 3. Let π be an assignment that assigns 1 to x_1, x_3 and 0 to x_2, x_4. The values in red below the leaves in Fig. 3 represent this assignment. If we propagate these values upstream to the root of the circuit, we get the values/formulas shown in red adjacent to internal nodes, as shown in Fig. 3. This process is akin to *constant/symbol propagation* in symbolic simulation [30]. Note that the root of the circuit is assigned $y \wedge \widehat{y}$ by this process, indicating that $\varphi^{+}\big|_{\pi} \Leftrightarrow (y \wedge \widehat{y})$. Hence, y is in conflict in φ and π is a counterexample to conflict-freeness of y in φ.

Interestingly, the constant/symbol propagation discussed above can yield many more counterexamples beyond π. Specifically, let N denote the set of coloured nodes in the figure. Suppose we cut the circuit at the nodes in N, as shown by the dotted line in Fig. 3. Let the sub-circuit above the cut be denoted C_N. Notice that the leaf nodes of C_N are either nodes in N or leaf nodes of the original circuit corresponding to y or $\widehat{y}$. Now consider any assignment $\pi' : \{x_1, x_2, x_3, x_4\} \to \{0, 1\}$ s.t. when we propagate constants/symbols in the original circuit starting with π' at the leaves, we get the same values as in Fig. 3 at all nodes in N. This ensures that all leaves of C_N have the same constant/symbol as in Fig. 3. Therefore, further constant/symbol propagation must assign exactly the same constant/symbol/formula at every internal node of C_N as in Fig. 3. Specifically, the root node is assigned $y \wedge \widehat{y}$, implying that π' is a counterexample to conflict-freeness of y in φ.

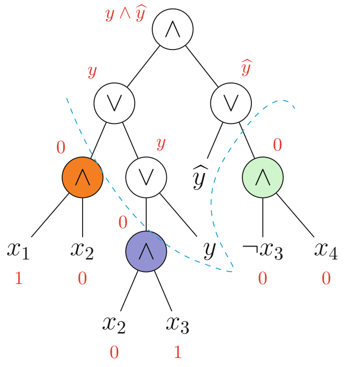

Fig. 3. Circuit representing φ^{+y}

Can we characterize all the counterexamples π' obtainable by the above method? It turns out we can do this. First, note from Fig. 3 that the sub-circuits rooted at the orange, purple and green nodes represent the Boolean formulas $x_1 \wedge x_2$, $x_2 \wedge x_3$ and $(\neg x_3 \wedge x_4)$ respectively. Hence, the set of all counterexamples π' obtained above are precisely the satisfying assignment of the formula $\beta \equiv \neg(x_1 \wedge x_2) \wedge \neg(x_2 \wedge x_3) \wedge \neg(\neg x_3 \wedge x_4)$. Notice that there are many assignments beyond π that satisfy β, e.g. $x_1x_2x_3x_4 = 0000$ or 0010 or 1000, and so on. Thus, we have truly *generalized* the counterexample π.

In general, given a specification $\varphi(\boldsymbol{X}, \boldsymbol{Y})$, an output variable y and a counterexample $\pi : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \to \{0, 1\}$ to conflict-freeness of y in φ, we first construct an NNF circuit representing φ^+. For every node n in the circuit, let φ_n^+ denote the sub-formula represented by the sub-circuit rooted at n. Next, we assign values given by π to the leaves of the circuit representing φ^+ and propagate these values to the root of the circuit. Let $v_{n,\pi}$ denote the constant/symbol/formula assigned to node n in the circuit by this process. In other words, $v_{n,\pi} \Leftrightarrow \varphi_n^+\big|_\pi$. We now choose a subset N of nodes n such that (i) $\mathsf{sup}(\varphi_n^+) \cap \{y, \widehat{y}\} = \emptyset$, (ii) $v_{n,\pi}$ is a constant, and (iii) every path from a non-y, non-$\widehat{y}$ leaf to the root passes through a node in N. Such a set N can always be found, for example, by choosing N to be the set of non-y, non-$\widehat{y}$ leaves. However, as Fig. 3 shows, N need not include only leaf nodes. Let $\beta_{\pi,N}$ denote the formula $\bigwedge_{n \in N} \left(\varphi_n^+ \Leftrightarrow v_{\pi,n}\right)$.

Lemma 5. *Every satisfying assignment of $\beta_{\pi,N}$ is a counterexample to conflict-freeness of y in φ. Moreover, $\neg\beta_{\pi,N}$ satisfies the three conditions required for a partial rectifier as specified in Lemma 3.*

Proof. Since every path from a non-y, non-$\widehat{y}$ leaf to the root passes through a node in N, we can use nodes in N and the leaves corresponding to y and $\widehat{y}$ to cut the circuit (as shown in Fig. 3). Let C_N denote the sub-circuit above this cut. Let π' be a satisfying assignment (not necessarily same as π) of $\beta_{\pi,N}$. By definition of $\beta_{\pi,N}$, constant/symbol propagation starting from π' assigns the constant value $v_{\pi,n}$ to every node $n \in N$. It follows that for all leaf nodes l of the sub-circuit C_N, $v_{\pi',l} = v_{\pi,l}$. Hence, every internal node m of C_N must also have $v_{\pi',m} = v_{\pi,m}$. In particular the root node gets assigned the same value/symbol/formula that it had when we did constant/symbol propagation starting from π. In other words, $\varphi^+\big|_{\pi'} \Leftrightarrow \varphi^+\big|_{\pi}$. However, Since π is a counterexample to conflict-freeness of y in φ, we know $\varphi^+\big|_{\pi} \Leftrightarrow (y \wedge \widehat{y})$. Therefore, $\varphi^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$ and π' is a counterexample to conflict-freeness of y in φ^+.

To see $\neg\beta_{\pi,N}$ satisfies the conditions required of a partial rectifier in Lemma 3, note that $\mathsf{sup}(\varphi_n^+) \cap \{y, \widehat{y}\} = \emptyset$. Therefore, $\mathsf{sup}(\neg\beta_{\pi,N}) \cap \{y, \widehat{y}\}$ is also empty. Next, by definiton, if an assignment $\pi' \models \beta_{\pi,N}$, every node $n \in N$ in the circuit φ^+ gets assigned the constant value $v_{\pi,n}$. Using the same argument as in the first part of the proof, we can then show that $\varphi^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$. Hence $\varphi\big|_{\pi'} \Leftrightarrow \varphi^+[\widehat{y} \mapsto \neg y]\big|_{\pi'}$ $\Leftrightarrow y \wedge \widehat{y}[\widehat{y} \mapsto \neg y] \Leftrightarrow 0$. This shows that $\beta_{\pi,N} \Rightarrow \neg\varphi$. In other words, $\varphi \Rightarrow \neg\beta_{\pi,N}$. Finally, $\beta_{\pi,N}\big|_{\pi} \Leftrightarrow \bigwedge_{n \in N} \left(\varphi_n^+\big|_{\pi} \Leftrightarrow v_{\pi,n}\right)$. However, $v_{\pi,n} \Leftrightarrow \varphi_n^+\big|_{\pi}$ by definition. Hence $\beta_{\pi,N}\big|_{\pi} \Leftrightarrow 1$ and hence $\neg\beta_{\pi,N}\big|_{\pi}$ is unsatisfiable. □

The above lemma allows us to use $\neg\beta_{\pi,N}$ as a partial rectifier of φ w.r.t. y in Algorithm RectifyOneOutput. Significantly, this eliminates in one shot all counterexamples to conflict-freeness of y in φ that are satisfying assignments of $\beta_{\pi,N}$, thereby reducing the number of iterations of the loop in Algorithm RectifyOneOutput. As seen in the example above, $\beta_{\pi,N}$ can indeed have many more satisfying assignments beyond π. We use this technique to implement the subroutine PartialRectifier in Algorithm RectifyOneOutput. Specifically, we choose the set N such that the longest path of each node $n \in N$ from a leaf of C_μ is within an empirically determined threshold (20 in our experiments).

Generalizing Using Unsatisfiable Cores: It turns out that we can generalize counterexamples even beyond what was achieved above. To see a concrete example, consider the specification $\gamma(\boldsymbol{X}, y) \equiv \varphi(\boldsymbol{X}, y) \wedge \big(\neg y \vee (x_1 \wedge x_2)\big)$, where $\varphi(\boldsymbol{X}, y)$ is the same specification considered in Fig. 3. The NNF circuit representing γ^{+y} (or γ^+ for short) is the same as that shown in Fig. 3 with an additional $\wedge$-gate that feeds the root node, and that is fed by the $\widehat{y}$ leaf and output of the orange node. The same assignment π as considered earlier serves as a counterexample to conflict-freeness of y in γ, and the same set N can be chosen to obtain the same partial rectifier $\neg\beta$, where $\beta \equiv \neg(x_1 \wedge x_2) \wedge \neg(x_2 \wedge x_3) \wedge \neg(\neg x_3 \wedge x_4)$. Note, however, that in the circuit for γ^+, if the orange and purple nodes are assigned the value 0 by constant propagation starting from an assignment π', the root node must be assigned $y \wedge \widehat{y}$, *regardless of the value assigned to the green node.* Therefore, we could have used $\beta' \equiv \neg(x_1 \wedge x_2) \wedge \neg(x_2 \wedge x_3)$, which represents a larger set of counterexamples than β. Specifically, $x_1x_2x_3x_4 = 1001$ does not satisfy β but satisfies β'. It follows that rectification using $\neg\beta'$ eliminates more counterexamples in one go than rectification using $\neg\beta$.

In general, given φ, y, π and N as in our previous discussion, let s_n be a fresh variable for every node $n \in N$, and define the formula $\rho_{\pi,N} \equiv \varphi \wedge$

$\bigwedge_{n\in N}\big((s_n \Rightarrow (\varphi_n^+ \Leftrightarrow v_{\pi,n})) \wedge s_n\big)$. Since $\varphi \Rightarrow \neg\beta_{\pi,N}$ (see Lemma 5) and since $\beta_{\pi,N} \equiv \bigwedge_{n\in N}(\varphi_n^+ \Leftrightarrow v_{\pi,n})$, it follows that $\rho_{\pi,N}$ is unsatisfiable. Assuming φ is satisfiable (otherwise the synthesis problem is itself trivial), every unsatisfiable core of $\rho_{\pi,N}$ must set a subset of the s_n variables to 1. Let $U \subseteq N$ be the set of nodes n s.t. $s_n = 1$ in a minimal unsatisfiable core of ρ. Then $\rho_{\pi,U} \equiv \varphi \wedge \bigwedge_{n\in U}\big((s_n \Rightarrow (\varphi_n^+ \Leftrightarrow v_{\pi,n})) \wedge s_n\big)$ is unsatisfiable.

Lemma 6. *Lemma 5 holds with $\beta_{\pi,N}$ replaced by $\beta_{\pi,U}$. Moreover, $\beta_{\pi,N} \Rightarrow \beta_{\pi,U}$.*

Algorithm 2 FindSkBasisVec($\varphi(\boldsymbol{X},\boldsymbol{Y})$)

```
1: $\alpha := \varphi$; $i := 1$ ▷ Assume $|\boldsymbol{Y}| = n$
2: repeat
3:     $y_i :=$ Next output variable to find Skolem basis
4:     $A_i := \alpha|_{y_i=1} \wedge \neg\alpha|_{y_i=0}$
5:     $B_i := \alpha|_{y_i=1} \Leftrightarrow \alpha|_{y_i=0}$
6:     if $y_i$ is positive unate in $\alpha$ then
7:         $\alpha := \alpha|_{y_i=1}$ ▷ Existentially quantify $y_i$
8:     else if $y_i$ is negative unate in $\alpha$ then
9:         $\alpha := \alpha|_{y_i=0}$ ▷ Existentially quantify $y_i$
10:    else
11:        $\mu :=$ RectifyOneOutput($\alpha$, $y_i$) ▷ $y_i$ is conflict-free in $\mu$
12:        $\alpha := \mu^{+y_i}|_{y_i=1,\widehat{y_i}=1}$ ▷ Existentially quantify $y_i$
13:    $i := i + 1$
14: until all outputs processed
15: return $\langle (A_i, B_i)\rangle_{1\le i\le n}$
```

Overall Algorithm: We are now present Algorithm FindSkBasisVec. The algorithm initializes a running specification α to φ. It then repeatedly chooses the next output y_i for whose Skolem functions a Skolem basis needs to be computed. The choice of y_i can be as per a static order, or as determined on-the-fly heuristically. The algorithm then finds Skolem basis (A_i, B_i) using Theorem 1 by treating y_i as the sole output in the specification α. It next updates the running specification α by existentially quantifying y_i from α. In order to do this, it first checks if y_i is unate in α, and if so, substitutes an appropriate constant for y_i in α to quantify it out. Otherwise, the algorithm invokes Algorithm Rectify-OneOutput. Thanks to Theorem 3, we can effectively and efficiently quantify y_i from α by setting $y_i = 1$ and $\widehat{y_i} = 1$ in the positive form of the formula μ returned by RectifyOneOutput. Once all outputs are processed, the algorithm outputs the vector of (A_i, B_i) pairs computed as the Skolem basis vector.

Theorem 4. *Algorithm* FindSkBasisVec *terminates with a Skolem basis vector for the specification $\varphi(\boldsymbol{X},\boldsymbol{Y})$.*

Proof. The proof of termination follows immediately from Theorem 3. The proof of correctness follows from Definition 1, Theorems 1, 3, and Lemmas 1, 2. □

Though we developed rectification as a technique for rendering a variable conflict free with the objective of generating Skolem basis vectors, it can be independently used to compile a Boolean formula to a form that allows efficient quantifier elimination. However, a performance evaluation of rectification versus other quantification techniques in such applications is beyond the scope of this paper.

7 Implementation and Experiments

We implemented the above algorithms in C++ using the abc package [27] and ran our tool on a set of 602 Boolean functional synthesis benchmarks (also used in [12,14]). We used an Intel(R) Xeon(R) CPU E5-2660 v2@2.20GHz machine with 40 cores in single-threaded mode (multiple cores used only to run experiments in parallel). We set an overall timeout of 3600 seconds, within which the timeout for unate-check was 1000 seconds.

Detailed Analysis of Our Results. We did an ablation study to understand which part of our approach was most successful in compiling the benchmarks. Our results are summarized in Fig. 4. Here, "Total solves" denotes the number (out of 602) benchmarks for which Algorithm FindSkBasisVec completed within the timeout. "PAR2 score" is a widely used weighted performance score, computed as sum of time taken (in seconds) for each solved instances and double of timeouts (3600 s)s) for each unsolved instance. For benchmarks that were rectified, *for each application of rectification, we verified (using a SAT solver) that the rectified circuit was semantically equivalent* to the original. The time for this verification is included when computing PAR2 scores. In row 3, we note the "Average time" taken (including for verification), in seconds, over all solved instances. In rows 4, 5 and 6, we count, respectively, the number of solved benchmarks, where (i) all variables were unate (ii) some but not all were unate and (iii) no variables were unate (these add up to row 1). In row 7, we list the number of solved benchmarks for which there was at least one conflict, i.e., a call to the rectification algorithm was needed. Row 8 lists the solved benchmarks with at least one output that was not unate but no outputs having conflicts. The other rows are self-explanatory.

		DO	SO	CDO	CSO
1	Total Solves	287	298	299	308
2	PAR2 Scores	3839.56	3672.65	3696.90	3565.01
3	Average time	151.28	74.29	146.94	95.24
4	allUnates	98	98	98	98
5	someUnates	146	157	151	160
6	noUnates	43	43	50	50
7	fixedConflicts	71	19	73	21
8	noConflicts	118	181	128	189
9	fixedConflicts ∩ someUnates	68	16	69	17
10	noConflicts ∩ someUnates	78	141	82	143
11	fixedConflicts ∩ noUnates	3	3	4	4
12	noConflicts ∩ noUnates	40	40	46	46

Fig. 4. Table of results

Order Dependence. Since a Skolem basis vector depends on the ordering of outputs, we considered two order variants. In the first, we considered a heuristically determined static order (denoted SO), taken as is from [14]. Then, we tried a heuristic dynamic order (denoted DO): after each output variable is processed, the next is obtained on-the-fly by applying the heuristic from [14].

Conflict Optimization in Calculating Skolem Basis Vector. We found several problem instances where the specification is not realizable, i.e., there exist input values for which no output values can make the specification true. For such instances, it is reasonable to restrict the computation of Skolem basis vector to a set F of Skolem functions, such that for any Skolem function $\psi \notin F$, there exists $\psi' \in F$ such that ψ and ψ' differ only on the space of input assignments for which no assignment of outputs would satisfy the specification. It turns out that this can be easily encoded in Algorithm 1 by modifying the conflict formula $\kappa_{\mu,y}$ to $\kappa_{\mu,y} \wedge \varphi(\boldsymbol{X}, \boldsymbol{Y'})$, where $\boldsymbol{Y'}$ is a fresh set of variables. Doing this, along with the static/dynamic ordering gives us the "CSO" and "CDO" columns in Fig. 4.

Observations. With either SO or DO, without conflict optimization, we are able to *compute Skolem basis vectors for 299 of 602 benchmarks* (286 were solved by both, 1 by only DO and 12 by only SO). Interestingly, the static order (SO) had fewer conflicts compared to the dynamic order (DO), when we had to rectify more often. Further, in the presence of conflict optimization, we are able to compute Skolem basis vectors for 309 out of 602 benchmarks. Note is that even though the PAR2 score is large, the average time taken is less than 2.5 min, including time taken for verification. In other words, *when we are able to compute Skolem basis vectors, we are able to do so in remarkably short duration.*

Comparison with Other Tools/Approaches. There are no existing tools that synthesize a represention of the space of all Skolem function vectors. Knowledge compilation tools e.g., C2Syn [13], NNF2SDD [25,31] come closest as they try to obtain a single circuit that is semantically equivalent to the original and is in a normal form: the SynNNF form for C2Syn and the SDD form for NNF2SDD. Skolem functions hence could be potential alternative approaches. In practice, C2Syn does refinement (see [13]) operations for performance boosting, thereby restricting the space of Skolem function vectors. Even with this optimization for C2Syn it can compile only 218 (out of 602) benchmarks, while NNF2SDD compiles only 142 to SDD on the same computing platform.

An apples-to-apples performance comparison of Boolean functional synthesis tools (that synthesize a single Skolem function vector) with our tool (that computes Skolem basis vectors for all Skolem function vectors) is not possible, since two different problems are being solved. Nevertheless, to understand the performance penalty incurred in computing a representation of all Skolem function vectors, we observe from [12] that with a 7200 s s timeout and using a more powerful cluster, Manthan [12] (resp. BFSS [14]) could synthesize a single Skolem function vector for ∼356 (resp. 247) out of the same 602 benchmarks. In comparison, with 3600 s s timeout, we are able to compute Skolem basis vector for

$\sim$ 300 benchmarks. In [17], an improved and highly engineered tool Manthan2 was developed, which could synthesize a single Skolem function vector for 502 benchmarks within 7200 s.s. Interestingly, *we are able to compute Skolem basis vectors for 22 benchmarks (out of which 13 have non-unate variables), for which even Manthan2* [17] *fails to synthesize a single Skolem function vector.*

8 Conclusion

In this work, we have introduced a representation for the space of Skolem functions, using the notion of Skolem basis vector. Our representation itself is criteria-agnostic, but allows the use of other existing techniques to optimize Skolem functions wrt different criteria. We develop a compilation algorithm that uses a combination unate and conflict-detection along with generalized counter-example guided approach to synthesize the Skolem basis vector. Our next step would be to identify specific problem contexts and optimization criteria and integrate our approach with the state-of-the-art logic synthesis tools to synthesize specific Skolem functions satisfying the given criteria.

References

1. Jacobs, S., et al.: 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 (2015)
2. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 1362–1369 (2017)
3. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. Int. J. Softw. Tools Technol. Transf. **7**(2), 118–128 (2005). https://doi.org/10.1007/s10009-004-0179-0
4. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT **15**(5–6), 497–518 (2013)
5. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)
6. Balabanov, V., Jiang, J.-H.R.: Resolution proofs and skolem functions in QBF evaluation and applications. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 149–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_12
7. Balabanov, V., Jiang, J.-H.R.: Unified QBF certification and its applications. Form. Methods Syst. Des. **41**(1), 45–65 (2012)
8. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 430–435. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_33
9. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. SIGPLAN Not. **45**(6), 316–329 (2010)
10. Jo, S., Matsumoto, T., Fujita, M.: Sat-based automatic rectification and debugging of combinational circuits with lut insertions. In: Proceedings of the 2012 IEEE 21st Asian Test Symposium, ATS 2012, pp. 19–24. IEEE Computer Society (2012)

11. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 375–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_23
12. Golia, P., Roy, S., Meel, K.S.: Manthan: a data-driven approach for Boolean function synthesis. Comput. Aided Verificat. **12225**, 611–633 (2020)
13. Akshay, S., Arora, J., Chakraborty, S., Krishna, S., Raghunathan, D., Shah, S.: Knowledge compilation for boolean functional synthesis. In: Proceedings of of Formal Methods in Computer Aided Design (FMCAD), 2019
14. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Boolean functional synthesis: hardness and practical algorithms. Formal Methods Syst. Des. **57**(1), 53–86 (2021)
15. Akshay, S., Chakraborty, S., John, A.K., Shah, S.: Towards parallel boolean functional synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 337–353. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_19
16. John, A., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: FMCAD, pp. 73–80 (2015)
17. Golia, P., Slivovsky, F., Roy, S., Meel, K.S.: Engineering an efficient Boolean functional synthesis engine. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, 1–4 November 2021, pp. 1–9. IEEE (2021)
18. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_22
19. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 124–131 (2017)
20. Shah, P., Bansal, A., Akshay, S., Chakraborty, S.: A normal form characterization for efficient boolean skolem function synthesis. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, 29 June–2 July 2021, pp. 1–13. IEEE (2021)
21. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Int. Res. **17**(1), 229–264 (2002)
22. Darwiche, A.: Decomposable negation normal form. J. ACM **48**(4), 608–647 (2001)
23. Darwiche, A.: Tractable Boolean and arithmetic circuits. In: Hitzler, P., Sarker, M.K. (eds.) Neuro-Symbolic Artificial Intelligence: The State of the Art, vol. 342 of Frontiers in Artificial Intelligence and Applications, pp. 146–172. IOS Press (2021)
24. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. J. Appl. Non-Classical Logics **11**(1–2), 11–34 (2001)
25. Shi, W., Shih, A., Darwiche, A., Choi, A.: On tractable representations of binary neural networks. In: Calvanese, D., Erdem, E., Thielscher, M. (eds.) Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, 12–18 September 2020, pp. 882–892 (2020)
26. Somenzi, F.: Binary decision diagrams. In: Calculational System Design, vol. 173 of NATO Science Series F, pp. 303–366. IOS Press (1999)
27. Abc: A system for sequential synthesis and verification
28. Mishchenko, A., Brayton, R.K., Jiang, J.H.R., Jang, S.: Scalable don't-care-based logic optimization and resynthesis. ACM Trans. Reconfigurable Technol. Syst. **4**(4), 34:1-34:23 (2011)

29. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, Boston (1994)
30. Bryant, R.E.: Symbolic simulation-techniques and applications. In: 27th ACM/IEEE Design Automation Conference, pp. 517–521 (1990)
31. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, 16–22 July 2011, pp. 819–826. IJCAI/AAAI (2011)