Template-based circuit understanding

Adrià Gascón*, Pramod Subramanyan†, Bruno Dutertre*, Ashish Tiwari* Dejan Jovanović*, Sharad Malik†
*SRI International
adria.gascon@sri.com, bruno@csl.sri.com, tiwari@csl.sri.com, dejan.jovanovic@sri.com

†Princeton University
psubrama@princeton.edu, sharad@princeton.edu

***Abstract*—When verifying or reverse-engineering digital circuits, one often wants to identify and understand small components in a larger system. A possible approach is to show that the sub-circuit under investigation is functionally equivalent to a reference implementation. In many cases, this task is difficult as one may not have full information about the mapping between input and output of the two circuits, or because the equivalence depends on settings of control inputs.**

We propose a template-based approach that automates this process. It extracts a functional description for a low-level combinational circuit by showing it to be equivalent to a reference implementation, while synthesizing an appropriate mapping of input and output signals and setting of control signals. The method relies on solving an exists/forall problem using an SMT solver, and on a pruning technique based on signature computation.

I. Introduction

Digital circuits are designed and implemented in a top-down fashion, typically using computer-aided design (CAD) tools that provide several levels of abstraction. Hence, a variety of components must be understood separately to derive the high-level functionality of the whole system. However, after the original high-level description is mapped to a low-level digital circuit—i.e., a flattened netlist—most of the modularity that made the original description understandable is lost. For this reason, an unavoidable task in reverse-engineering of industrial size digital circuits is to extract subcircuits of the original design to verify them independently. This task is referred as the functional block identification step in [1]. Techniques that tackle this problem include structural, functional, and mixed approaches such as

1) FSM extraction [2]
2) Functional aggregation and matching [3]
3) Word identification and propagation [4]
4) Identification of repeated structures [5]

After identification of a component $\mathcal{C}$ by these methods, an important step is understanding $\mathcal{C}$'s functionality. Ideally, we would like a systematic way of obtaining a reasonable approximation of the high-level description of $\mathcal{C}$ in some Hardware Description Language (HDL). A possible approach is to try matching the function computed by $\mathcal{C}$ against a library of predefined components. However, this option is typically too strict in practice. A source of difficulty is that the mapping between the inputs of $\mathcal{C}$ and the component to be matched is usually unknown. *Permutation-Independent Equivalence Checking* (PIEC) addresses this problem [6], [7], [8]. It has been applied in the context of technique 2) above. Once the wires in a flattened netlist have been grouped into *unordered words*, and a combinational subcircuit $\mathcal{C}$ operating on those words has been extracted, $\mathcal{C}$ is checked for equivalence with known library components *modulo a permutation* of such words that *is determined* by the matching algorithm.

Even with an equivalence checking algorithm that synthesizes a suitable input/output permutation, another practical difficulty may be that suitable library components are not available. Because of optimization steps applied in the flattening process, and the specifics of the design, $\mathcal{C}$ does not necessarily have a standard functionality. For example, our benchmark includes a subcircuit automatically obtained from a real design by unfolding an FSM found using a technique in the first category above. The circuit has 170 wires and 120 components from a circuit synthesis library, and it has 30 inputs out of which six are control signals. The circuit implements a 22-bit up counter modulo $(2^{20} + 2^{21})$ with synchronous reset and hold. This design can be described in fewer than ten lines of VHDL, but it is not reasonable to assume that we have a predefined reference circuit that exactly matches its functionality, even modulo a permutation of inputs and outputs. This motivates the need for more flexible functional matching algorithms that enables reverse engineering without prior low-level knowledge of the circuit under investigation.

Ideally, we would like to synthesize a suitable permutation of the inputs and the corresponding VHDL code for $\mathcal{C}$. Unfortunately that is not possible in practice. Instead, we solve a more constrained version of this synthesis problem: the combinational circuit $\mathcal{C}$ is checked for equivalence against a template spanning a finite, but possibly huge, family of high-level circuit descriptions. More specifically, our goal is to describe the functionality of a *combinational* circuit $\mathcal{C}$ using word-level operations such as concatenation, extraction, shifting, and rotation, as well as arithmetic functions on words such as addition, subtraction, multiplication, modulo, and the usual arithmetic comparison functions for signed and unsigned integers.

Our solution is inspired by recent progress in the area of program synthesis. Synthesizing a program from an abstract specification is not achievable in practice, but *template-based* synthesis is much more practical [9]. In this approach, the designer provides a *template* that captures the shape of the intended solution(s) together with the specification. A synthesis algorithm fills in the details. This general idea has

The research presented in this paper has been partially supported by the National Science Foundation under grant CCF-1423296.

ISBN: 978-0-9835678-4-4.

been successfully applied to several domains. For example, imperative programs can be obtained from a given sketch, as long as their intended behavior is also provided [10]; efficient bitvector manipulations can be synthesized from naïve implementations [11]; agent behavior in distributed algorithms can be synthesized from a description of a global goal [12]; circuits can be repaired given a specification of their intended behavior [13]; deobfuscated code can be obtained using similar ideas [14]. Although all these applications rely on template-based synthesis, different synthesis algorithms are used in different domains.

In our setting, the functional specification is the circuit $\mathcal{C}$ itself, and our goal is to generate a high-level description of its functionality by instantiating a user-provided template that operates at the word level. The template is a convenient way of integrating user knowledge to reduce the search space. Our approach automatically synthesizes both an input/output permutation and a set of Boolean conditions on some inputs, under which the circuit $\mathcal{C}$ is equivalent to a high-level description. We call this problem *Permutation-Independent Conditional Equivalence Checking* (PICEC).

Our approach to solving PICEC relies on (i) a set of syntactic transformations similar to the ones used in [15], (ii) an efficient implementation of validity checking for Boolean formulas over the theory of bitvectors with two levels of quantification, i.e., $\exists\forall$ QF_BV formulas, and (iii) the use of distinguishing signatures to handle the search for suitable input and output permutations.

We evaluated our techniques on a set of reverse-engineering benchmarks that were generated by synthesizing a variety of circuits described in high-level (behavioral) Verilog using the Synopsys Design Compiler (DC). All our benchmarks and circuits, both as high-level Verilog and as flattened netlist, are available at [16]. Our results indicate that our functional matching approach can be very effective in practice for any task that requires getting a precise understanding of the high-level functionality of a digital system.

In Section II, we introduce some notation and precisely state the Permutation Independent Conditional Equivalence Checking problem. In Section III, we briefly present our approach for solving the synthesis problem, including the preprocessing techniques. In Section IV, we review previous work on the use of output and input distinguishing signatures for solving PIEC and show how we used it in our context. In Sections V and VI, we present our experimental results and future lines of research.

II. Template-based circuit understanding

As mentioned in the previous section, our goal is to extract a high-level understanding of the behavior of a given combinational circuit $\mathcal{C}$. More specifically, we would like to raise the level of abstraction of the description of the functionality of $\mathcal{C}$ from bits and standard logical gates to a variety of word-level manipulation operations and arithmetic functions. Motivated by this goal, we first formulate a generic *Permutation Independent Conditional Equivalence Checking* (PICEC, pronounced "pieces") problem, then present a refined PICEC problem. Finally, we show how it can be solved using an exists-forall solver.

Let I and O be disjoint sets of variables ranging over some domain D. Intuitively, I and O correspond to the inputs and outputs of our circuit $\mathcal{C}$. In all our experiments, D is the Boolean domain, but the PICEC problem can be defined for arbitrary domains. Given a set V of variable ranging over D, by $\texttt{Words}_\texttt{k}(V)$ we denote the set of words over V of length k. We simply refer to $\texttt{Words}(V)$ when k is clear from the context or irrelevant.

Since our goal is to raise the level of abstraction of the description of the functionality of $\mathcal{C}$ from bits to words, a key challenge is *finding the right words* from the sets I and O. To do so, our procedure must consider all possible functions that produce a word of a certain size from I and O, so-called extraction functions. An *extraction function* is a function that maps a set V of variables to a word in $\texttt{Words}_k(V)$, for some positive constant k.

We are now ready to define our problem precisely. As commented in Section I, our goal is to provide a *flexible* procedure for checking whether a circuit exhibits a certain behavior, that is, checking whether a circuit may compute a certain function under conditions *to be determined* and for some selection of its inputs and outputs that is also *to be determined*. The *Generic* PICEC Problem captures this idea. The goal of the following definition is to provide the reader with a high-level intuition of the goal of our formalization. As comented above, this general definition is then refined to the formulation of the problem being addressed in this work, which is presented in Definition 3 below.

Definition 1 (Generic PICEC): Given a quantifier-free formula $\mathcal{C}(I, O)$ (over free variables I and O), and a function $\phi : \texttt{Words}(D) \times \texttt{Words}(D) \mapsto \texttt{Words}(D)$, the *PICEC problem* seeks to find
(a) a partition $I_C \cup I_D$ of I into control variables I_C and data variables I_D,
(b) a satisfiable formula $\psi(I_C)$ with free variables in I_C,
(c) extraction functions $\texttt{ex}_1, \texttt{ex}_2$ on I_D, and
(d) an extraction function $\texttt{ex}_3$ on O,
such that the sentence

$$\forall I, O \ : \ \mathcal{C}(I, O) \Rightarrow \\ (\psi(I_C) \Rightarrow (\texttt{ex}_3(O) = \phi(\texttt{ex}_1(I_D), \texttt{ex}_2(I_D))))$$

is valid in the theory of the underlying domain $\texttt{Words}(D)$.

Intuitively, a solution of the generic PICEC problem shows that, under the condition $\psi(I_C)$, the circuit $\mathcal{C}$ behaves like the function ϕ on some suitably identified input and output words. Solving the generic PICEC problem amounts to synthesizing the parts (a)–(d) in Definition 1.

Example 1: Consider a circuit $\mathcal{C}(I, O)$ with set of binary inputs $I = \{i_1, i_2, i_3, i_4, c\}$ and a single output o. Assume that $\mathcal{C}$ implements the following function

$$f(i_1, i_2, i_3, i_4, c) = \begin{cases} i_1 i_2 > i_3 i_4, & \text{if } c = 0 \\ i_1 i_1 \geq i_3 i_4, & \text{otherwise.} \end{cases}$$

and consider the function $\phi(w_1, w_2) = (w_1 \geq w_2)$. An *interesting solution* of this PICES instance consists of: (a) the partition $I = \{c\} \cup \{i_1, i_2, i_3, i_4\}$, (b) the Boolean formula $\psi(\{c\}) = c$, (c) extraction functions $\texttt{ex}_1(I \setminus \{c\}) =$

$i_1 i_1, \mathtt{ex}_2(I \setminus \{c\}) = i_3 i_4$, and (d) the extraction function $\mathtt{ex}_3(\{o\}) = o$.

We have defined ϕ as a binary function to keep the presentation simple, but the definition generalizes to functions of any arity. Furthermore, in practice, we do not have just one function ϕ that we are "searching for" in a circuit $\mathcal{C}$, but a whole set $\phi_1, \ldots, \phi_m$ of functions. In this case, we want to share the partition synthesized in Part (a) across all the m functions, but synthesize different Parts (b)–(d) for the m different functions. This extension corresponds to the *Generic m-PICEC problem* defined as follows:

Definition 2 (Generic m-PICEC): Given a quantifier-free formula $\mathcal{C}(I, O)$ (over free variables I and O), and given m functions $\phi_1, \ldots, \phi_m$ each with signature $\mathtt{Words}(D) \times \mathtt{Words}(D) \mapsto \mathtt{Words}(D)$, the *generic m-PICEC problem* seeks to find
(a) a partition $I_C \cup I_D$ of I into control variables I_C and data variables I_D,
(b) m satisfiable formulas $\psi_i(I_C)$ with free variables in I_C,
(c) $2m$ extraction functions $\mathtt{ex}_{i,1}, \mathtt{ex}_{i,2}$ on I_D, and
(d) m extraction functions $\mathtt{ex}_{i,3}$ on O,
(where $i \in \{1, \ldots, m\}$ in Items (b)–(d)) such that the sentence

$$\forall I, O \ : \ \mathcal{C}(I, O) \Rightarrow \Big(\bigwedge_i \psi_i(I_C) \Rightarrow (\mathtt{ex}_{i,3}(O) = \phi_i(\mathtt{ex}_{i,1}(I_D), \mathtt{ex}_{i,2}(I_D)))\Big)$$

is valid in the theory of the underlying domain D.

Note that a solution to the m-PICEC problem does not necessarily specify a total mapping between inputs and outputs values of $\mathcal{C}$, but only a mapping under the condition $\bigvee_i \psi_i(I_C)$, and hence the first C in PICEC. This flexibility is very helpful in a reverse-engineering process to *incrementally* understand the high-level functionality of the circuit.

The generic m-PICEC problem raises two issues. First, its synthesis search space (that is, the state space of the synthesis parameters in Parts (a)–(d) above) is huge. More importantly, it does not provide a way of integrating user-provided knowledge to reduce the synthesis search space. In particular, the user might have some knowledge about which variables form *unordered words*. The user may also wish to put constraints on the different extraction functions used for different choices of i (saying that some of them have to be the same extraction function). This is typically the case in practice, for example, when trying to understand an ALU-like circuit.

The user's knowledge of the circuit is captured in a *template*. A template T for a circuit $\mathcal{C}(I, O)$ is an 8-tuple

$$\langle O_T, \{S_1, \ldots, S_n, I_C\}, p, \{\phi_1, \ldots, \phi_m\}, \mathit{arg}_1, \mathit{arg}_2, \mathit{perm}_1, \mathit{perm}_2 \rangle$$

where $O_T \subseteq O$ is a subset of output variables, $I = (I_C \cup \bigcup_{i=1}^{n}(S_i))$ is a partition of the input variables, $p \geq 1$ is a natural number, the ϕ_i's are binary functions over words as before, and $\mathit{arg}_1, \mathit{arg}_2 : \overline{m} \mapsto \overline{n}$ and $\mathit{perm}_1, \mathit{perm}_2 : \overline{m} \mapsto \overline{p}$ are mappings. Here by $\overline{m}$ we denote the set $\{1, \ldots, m\}$.

Intuitively, O_T represents the subset of outputs of T explained in the template, the partition of I captures knowledge on how input words and control inputs are grouped. The problem is to correctly order the wires within those words by synthesizing p input permutations $\sigma_1, \ldots, \sigma_p$. The ϕ_i's are functions that the circuit is expected to implement *under some conditions on the inputs in I_C*. The template specifies that the input to ϕ_i are the two sets of $S_{\mathit{arg}_1(i)}$ and $S_{\mathit{arg}_2(i)}$ and that these sets of wires must be ordered according to permutations $\sigma_{\mathit{perm}_1(i)}$ and $\sigma_{\mathit{perm}_2(i)}$, respectively.

We are now ready to define the m-PICEC problem.

Definition 3 (m-PICEC problem): Given a quantifier-free formula $\mathcal{C}(I, O)$ (over free variables I and O), and given a template T as defined above, the *m-PICEC problem* seeks to find
(a) $p + 1$ permutations $\theta, \sigma_1, \ldots, \sigma_p$ and
(b) m satisfiable formulas $\psi_i(I_C)$ with free variables in I_C,
such that the sentence

$$\forall I, O : \mathcal{C}(I, O) \Rightarrow \bigwedge_i (\psi_i(I_C) \Rightarrow Eq_i) \qquad (1)$$

is valid in the theory of the underlying domain D, where Eq_i stands for

$$(\theta(O_T) = \phi_i(\sigma_{\mathit{perm}_1(i)}(S_{\mathit{arg}_1(i)}), \sigma_{\mathit{perm}_2(i)}(S_{\mathit{arg}_2(i)})))$$

Since encoding the "satisfiable" condition in Part (b) is tricky, we assume that the formula $\psi_i(I_C)$ denotes *an assignment* to the variables in I_C. Then, it immediately follows that the m-PICEC problem reduces to checking validity of the following exists-forall *synthesis constraint*:

$$\exists \psi_1, \ldots, \psi_m, \sigma_1, \ldots, \sigma_p, \theta : \forall I, O : \Phi \qquad (2)$$

where Φ is the matrix (quantifier-free part) of Formula (1).

Example 2: In practice, we specify the templates using an extension of the Yices language [17], as illustrated in Figure 1. In this example, we wish to determine whether a circuit $\mathcal{C}$ behaves as an adder under some condition and as a comparator under another condition. The corresponding formal template is given by

$$O_T := \mathtt{outputs}$$
$$\{S_1 := \mathtt{inputsA}, S_2 := \mathtt{inputsB}, I_C := \mathtt{control}\}$$
$$p := 2$$
$$\{\phi_1 := \mathtt{bv\text{-}add}, \phi_2 := \mathtt{bv\text{-}slt\text{-}int}\}$$

with arg_1, arg_2, perm_1, and perm_2 defined by $\mathit{arg}_1(i) = 1, \mathit{arg}_2(i) = 2, \mathit{perm}_1(i) = 1, \mathit{perm}_2(i) = 2$ for $i = 1, 2$. The function `bv-slt-int` is a signed less-than operator that returns 1 or 0. Let $\prec$ denote the less-than relation on signed integers encoded in two's complement representation. The synthesis constraint is satisfiable if there exist two permutations p and q, and bitvector constants $v1$ and $v2$, such that, for all possible values of `inputsA` and `inputsB`, (1) whenever `control` $= v1$, then $\mathcal{C}$ outputs $p(\mathtt{inputsA}) + q(\mathtt{inputsB})$ and (2) whenever `control` $= v2$, then $\mathcal{C}$ outputs 1 if $p(\mathtt{inputsA}) \prec q(\mathtt{inputsB})$ and 0 otherwise.

Since we are dealing either with combinational circuits or unfolding of sequential circuits, the relation $\mathcal{C}(I, O)$ can be represented as a Boolean formula. Then Formula 2 belongs to the logic of fixed-sized bit vectors with two levels of quantification $\exists$ and $\forall$. In the following section, we describe our solver, whose implementation is based on the Yices [18] SMT-solver. Our solver applies some general preprocessing

```
(and
    (=>
        (value v1 control)
            (=
               outputs
                   (bv-add
                       (permute p inputsA)
                       (permute q inputsB)
    )   )   )

    (=>
        (value v2 control)
            (= outputs
                (ite
                    (bv-slt
                        (permute p inputsA)
                        (permute q inputsB)
                    )
                    (mk-bv 32 1)
                    (mk-bv 32 0)
)   )   )   )     )
```

Fig. 1. Example of a user-defined template

techniques also used in [15]. In particular equality resolution is very effective in our setting due to the restricted form of our templates.

III. Solving the ∃∀ problem

The synthesis problem reduces to solving Boolean formulas over the theory of bit-vectors with two levels of quantification, commonly called the ∃∀ QF_BV fragment. Formulas in this fragment have the general form

$$(\exists \vec{x})\,(E(\vec{x}) \wedge (\forall \vec{y}) A(\vec{x}, \vec{y}))$$

Such formulas can be reduced to quantified Boolean formulas and delegated to a general QBF solver (e.g., [19]). Instead, we opt for reasoning at the higher level of bit-vectors and relying on a counterexample-refinement loop, similar to the approach used in 2QBF solvers (e.g., [20], [21]).

This loop is sketched in Figure 2. Given an ∃∀ formula, as above, the procedure is a game between two (quantifier-free) bit-vector solvers. The first solver generates candidate solutions for the existential variables $\vec{x} \mapsto \vec{a}$ by solving E. If there are no solutions to E, then the ∃∀ formula is unsatisfiable. Otherwise, the second solver checks whether the candidate solution $\vec{a}$ is correct, by trying to refute A modulo the assignment $\vec{x} \mapsto \vec{a}$. If the latter formula can not be refuted, then $\vec{a}$ is a solution to the ∃∀ problem. Otherwise, the second solver produces a refutation counterexample $\vec{b}$. This counterexample $\vec{b}$ eliminates $\vec{a}$ from the set of candidates for the existential variables. But $\vec{b}$ can eliminate more candidates than $\vec{a}$: all good candidates must satisfy $A[\vec{y}/\vec{b}]$. This new assertion (on the variables $\vec{x}$) is then added to the first solver's context and the loop proceeds. It is easy to see that this procedure terminates as the variables $\vec{x}$ have a finite domain. It is worth noting that the more general procedure for deciding quantified bit-vectors and uninterpreted functions in the Z3 SMT solver [15] reduces to our procedure when used on the ∃∀ QF_BV fragment.

A. Formula Simplification

The ∃∀ procedure is complete but may be very slow to terminate. High-level preprocessing and simplifications of the ∃∀ formula are essential to make it practical.

loop
  $\langle sat_x, \vec{x} \mapsto \vec{a} \rangle \leftarrow$ SMT-SOLVE(E)
  if not sat_x **then**
    return unsat
  $\langle sat_y, \vec{y} \mapsto \vec{b} \rangle \leftarrow$ SMT-SOLVE$(\neg A[\vec{x}/\vec{a}])$
  if not sat_y **then**
    return $\langle \textbf{sat}, \vec{x} \mapsto \vec{a} \rangle$
  $E \leftarrow E \wedge A[\vec{y}/\vec{b}]$

Fig. 2. Main loop for solving $(\exists \vec{x})\,(E(\vec{x}) \wedge (\forall \vec{y})\ A(\vec{x}, \vec{y}))$.

For reducing the scope of quantifier we distribute quantifiers over compatible Boolean operators (this is known as *miniscoping*):

$$\begin{array}{rcl} (\exists \vec{x})A \vee B & \Leftrightarrow & (\exists \vec{x})A \ \vee \ (\exists \vec{x})B \\ (\forall \vec{x})A \wedge B & \Leftrightarrow & (\forall \vec{x})A \ \wedge \ (\forall \vec{x})B \end{array}$$

The first simplification decomposes an ∃∀ problem into smaller subproblems, while the second simplification reduces candidate checking to several smaller checks.

It is common for our ∃∀ problems to contain subformulas of the following form:

$$(\exists \vec{x})(E(\vec{x}) \wedge (\forall \vec{y})(\bigwedge_{i \in S} (y_i = x_{j_i}) \Rightarrow B(\vec{y})))$$

where S is a subset of indexes in $1..|\vec{y}|$ and $j_i \in 1..|\vec{x}|$, for every $i \in S$. A naïve application of our procedure does not work well on such problems. To illustrate a worst case scenario, let us assume that B is unsatisfiable. In such a case, each iteration of our procedure will pick a fresh candidate assignment $\vec{x} \mapsto \vec{a}$, then refute the universal subformula with a counterexample $\vec{y} \mapsto \vec{b}$. Since we must have that $b_i = a_{j_i}$, for every $i \in S$, and B evaluates to false under $\vec{b}$, the counterexample instantiation yields the weakest possible explanation $\bigvee_{i \in S}(x_{j_i} \neq a_{j_i})$, which (essentially) eliminates only the current candidate for $\vec{x}$.[1]

To preserve the connections that equalities introduce over quantifiers, we perform *equality resolution*. We detect equalities of the form $(y_i = t_i)$ in the antecedents of universal subformulas, then solve out the variables y_i. In our previous examples, this simplifies the problem to the equivalent formula

$$(\exists \vec{x})(E(\vec{x}) \wedge (\forall \vec{y})B')$$

where B' is the result of solving out the variables y_i from B. On this simplified formula, the procedure now has a chance to eliminate more than one candidate at each iteration.

In addition to these formula simplifications, we reduce the solver's search space by relying on *distinguished signatures*, a technique originally proposed for solving PIEC.

IV. Distinguishing signatures

Functional equivalence checking of circuits is a central problem in logic synthesis and verification. Roughly speaking, it consists of determining whether two given circuits $\mathcal{C}_1(I_1, O_1)$, $\mathcal{C}_2(I_2, O_2)$ implement the same function. Without loss of generality, we can assume that $n = |I_1| = |I_2|$ and $m = |O_1| = |O_2|$.

[1] It may eliminate more than one candidate if S is a strict subset of $1..|\vec{y}|$.

In our setting, $\mathcal{C}_1$ and $\mathcal{C}_2$ are combinational circuits represented as multi-output Boolean functions $f = (f^1, \ldots, f^m)$ and $g = (g^1, \ldots, g^m)$, respectively. The combinational equivalence checking problem consists in deciding whether the sentence $\forall 1 \leq i \leq m : \forall x_1, \ldots, x_n : f^{\theta(i)}(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) = g^i(x_1, \ldots, x_n)$ is valid, where σ and θ are input and output correspondence between $\mathcal{C}_1$ and $\mathcal{C}_2$; σ is the input permtation and θ is the output permutation.

In the PIEC problem, the mappings σ and θ are not known and we must synthesize them. We can then formulate the problem as checking validity of the formula

$$\exists \theta, \sigma : \forall 1 \leq i \leq m, x_1, \ldots, x_n : \quad f^{\theta(i)}(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) = g^i(x_1, \ldots, x_n) \tag{3}$$

This problem has been widely studied [22], [23], [8]. To reduce the size of the search space for θ, a common approach is to assign an abstract *signature* to every output of $\mathcal{C}_1$ and $\mathcal{C}_2$, with two key properties. First, functions with different signatures cannot be equivalent. Second, the signature of a function f_i (or g_i) is invariant under any permutation of the input variables $x_1, \ldots, x_n$. If g^i and f^j have different signatures, we can reduce the search to output permutations that satisfy $\theta(i) \neq j$. This method extends to the input signals: one can also assign signatures to every input x_i to eliminate a priori some invalid input permutations σ.

More concretely, let $\mathcal{B}_n$ be the set of all single-output Boolean functions with n input variables, and let $\mathcal{D}$ be an ordered set. An input signature is a function $s_{in} : \{x_1, \ldots, x_n\} \times (\mathcal{B}_n)^m \to \mathcal{D}$, such that the equality

$$s_{in}(x_i, f^1(x_1, \ldots, x_n), \ldots, f^m(x_1, \ldots, x_n)) = s_{in}(x_{\sigma(i)}, f^{\theta(1)}(x_{\sigma(1)}, \ldots, x_{\sigma(n)}), \ldots, f^{\theta(m)}(x_{\sigma(1)}, \ldots, x_{\sigma(n)}))$$

holds for every input permutation σ and output permutation θ. Similarly, an output signature s_{out} is a function $s_{out} : \mathcal{B}_n \to \mathcal{D}$ such that $s_{out}(f(x, \ldots, x_n)) = s_{out}(f(x_{\sigma(1)}, \ldots, x_{\sigma(n)}))$ holds for any input permutation σ.

Consider Formula (3) above and assume that, for some inputs x_i and x_j, we have $s_{in}(x_i, f^1, \ldots, f^n) \neq s_{in}(x_j, g^1, \ldots, g^n)$. Then, since equal signatures are a necessary condition for i to be mapped to j by the input permutation σ, it follows that $\sigma(i) \neq j$. The case of output permutations and an output signature s_{out} is analogous. We can collect all disequality constraints derived from input signatures in a formula $C_{in}(\sigma)$ and all disequalities derived from output signatures in $C_{out}(\theta)$. Then, Formula 3 is equivalent to

$$\exists \theta, \sigma \ : \ C_{in}(\sigma) \wedge C_{out}(\theta) \wedge \forall 1 \leq i \leq m, x_1, \ldots, x_n \ : \ f^{\theta(i)}(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) = g^i(x_1, \ldots, x_n) \tag{4}$$

We apply a similar idea to our synthesis constraint (Formula (2)) from Section II.

A variety of signatures have been presented in the literature, many of them derived from a Reduced Ordered Binary Decision Binary Diagrams (ROBDDs) representation of Boolean functions. For a detailed presentation of a variety of signatures, their applications, and limitations, the reader is referred to [24], [25]. In this paper we focus on two signatures that do not rely on ROBDDs and are thus more scalable.

A. The in_dep and out_dep Signatures

Given a Boolean formula $f(x_1, \ldots, x_n)$ and a variable x_i, we say that f essentially depends on x_i, denoted by $f \preceq x_i$, if there exists an Boolean tuple $(\alpha_1, \ldots, \alpha_n)$ such that $f(\alpha_1, \ldots, \alpha_i, \ldots, \alpha_n) \neq f(\alpha_1, \ldots, \bar{\alpha_i}, \ldots, \alpha_n)$. Consider a circuit defined by m functions $f_1, \ldots, f_m$; we define the *input dependence set of* x and the *output dependence set of* f_i as follows:

$$\begin{aligned} in\_dep\_set(x, f^1, \ldots, f^n) &= \{f^j \ : \ f^j \preceq x\} \\ out\_dep\_set(f_i) &= \{x \ : \ f_i \preceq x\} \end{aligned}$$

Then, we define the two following signatures:

(a) $in\_dep(x, f_1, \ldots, f_n) = |in\_dep\_set(x, f_1, \ldots, f_n)|$

(b) $out\_dep(f) = |out\_dep\_set(f)|$

We must adapt these signatures to take templates into account. We want to produce a formula $C(\theta) \wedge C(\sigma_1) \wedge \ldots \wedge C(\sigma_n)$ that, as in the case of Formula 4, can be added to our synthesis constraint while preserving validity.

Recall that we defined a template as a tuple

$$\langle O_T = \{o_1, \ldots, o_l\}, \{S_1, \ldots, S_n, C\}, p, \{\phi_1, \ldots, \phi_m\}, arg_1, arg_2, perm_1, perm_2 \rangle.$$

The inputs and outputs of functions $\phi_1, \ldots, \phi_m$ are all bit vectors. We can then interpret each ϕ_i as a multi-output Boolean function, and we denote by ϕ_i^k the k-th bit of ϕ_i's output. We define our template-version of in_dep and out_dep, which we denote by in_dep_T and out_dep_T, for every input x_i and output o^k as follows

(c) $in\_dep_T(x_i, T) = |\bigcup_{j=1}^{m} in\_dep\_set(x_i, \phi_j(S_{arg_1(j)}, S_{arg_2(j)}))|$

(d) $out\_dep_T(o^k) = |\bigcup_{j=1}^{m} out\_dep\_set(\phi_j^k((S_{arg_1(j)}, S_{arg_2(j)})))|$

To see how to take advantage of this definition of signatures, consider a combinational circuit $\mathcal{C}$ and its representation as single-output Boolean functions $f_{\mathcal{C}}^1, \ldots, f_{\mathcal{C}}^m$ with input variables $x_1, \ldots, x_n$. Consider a template T for $\mathcal{C}$. The key observation is that fixing the value of variables in C cannot cause $in\_dep(x, f_{\mathcal{C}}^1, \ldots, f_{\mathcal{C}}^n)$ or $out\_dep(f_{\mathcal{C}}^i)$ to increase. Let j be any index in $\{1, \ldots, m\}$, and let x and y be input variables in $S_{arg_1(j)}$. Then we have that

$$(in\_dep(x_{k_1}, f_{\mathcal{C}}^1, \ldots, f_{\mathcal{C}}^n) > in\_dep_T(x_{k_2}, T)) \Rightarrow \sigma_{perm_1(j)}(x) \neq y$$

An analogous implication holds for $x, y \in S_{arg_2(j)}$. Similarly, for the output permutation θ, if $f_{\mathcal{C}}^i$ corresponds to a variable in $o_i \in O_T$ then, for any $k \in \{1, \ldots, l\}$, we have

$$(out\_dep(f_{\mathcal{C}}^i) > out\_dep_T(o^k)) \Rightarrow \theta(o_i) \neq o_k$$

The definition of essential dependence at the beginning of this section directly gives us a procedure to precompute input and output signatures of $\mathcal{C}$ and T by means of a quadratic number of calls to an SMT solver. Other signatures based on model counting are known to be very effective but they require circuits to be represented using ROBDDs. Our signature computation scales better than these BDD-based

approaches but we did not put emphasis on efficient signature computation in our investigations. Other symbolic approaches might be more effective.

In summary, we have an effective approach to produce a conjunction of constraints $C_{out}(\theta) \wedge C_{in}(\sigma_1) \wedge \ldots \wedge C_{in}(\sigma_n)$ that eliminates irrelevant permutations, and such that the formula

$$\exists \psi_1, \ldots, \psi_m, \sigma_1, \ldots, \sigma_n, \theta : \\ \forall I, O : C_{out}(\theta) \wedge C_{in}(\sigma_1) \wedge \ldots \wedge C_{in}(\sigma_n) \wedge \Phi$$

is equivalent to our synthesis constraint from section II.

In our implementation, we encode a permutation σ using a quadratic number of Boolean variables $\sigma_{i,j}$ such that $\sigma(i) = j \Leftrightarrow \sigma_{i,j}$. With this encoding, the formulas $C_{out}(\theta)$ and $C_{in}(\sigma)$ are simply conjunctions of literals.

Let us remark that, since the values of the signatures can be computed *independently* in the template and the circuit, we do not report on computation time needed for signature computation in our examples. Nevertheless, using a naive implementation based on calling an SMT solver, we can compute the signatures for all our examples in the order of minutes.

V. Experimental evaluation

We have evaluated our techniques on a set of reverse engineering benchmarks. These are flattened Verilog netlists that contain components such as ALUs, multipliers, shifters and counters. The benchmars were derived from various sources including the ISCAS'85 benchmarks, an ALU from an academic processor implementation, and synthetic examples. The flattened netlists were generated by synthesizing high-level (behavioral) Verilog using the Synopsys Design Compiler (DC). All the circuits, both in high-level Verilog and flattened netlists form are available at [16].

Our benchmarks exemplify the situation where a reverse engineer tries to understand the high-level functionality of a flattened design with limited information about the operation that it may perform and no detailed knowledge of its input/output buses. In less restrictive cases, the reverse engineer is given the grouping of the inputs into *unordered* words, and in some cases no information at all is known. The output of Synopsys DC is an optimized flattened Verilog netlist and our goal is to identify and extract the high-level modules contained within this flat netlist using template-based matching. Our toolchain reads these flattened Verilog netlists and the templates against which they are to be matched. It then encodes the matching problem as satisfiability queries to be solved by a backend solver. Currently, we can generate queries for the SMT solvers Yices and Z3, and QBF instances in the Q-Dimacs format.

Our template library contains modules such as adders, subtracters, shifters, multipliers, and counters of varying bitwidths. Each netlist was matched against a subset of these templates. We ensured that each netlist was matched against an approximately equal number of satisfying (matching) and unsatisfying (not matching) instances. The total number of instances is 40, of which an equal number are satisfiable and unsatisfiable. We believe these instances are a challenging yet realistic set of benchmarks relevant to the reverse engineering/logic deobfuscation problem. We have made the QBF, Yices and SMT2 instances generated by these matching problems available at [16]. The solver binaries used in our experiments are also available at this location.

We evaluated the performance of the following solvers: Yices [18] and Z3 [26], the QBF solvers RAReQS [27], DepQBF [28] and sKizzo [29], and a variant of the algorithm in Figure 2 (and also Algorithm 1 in [21]) that is somewhat similar to the algorithm presented in [13] that operates on a Boolean circuit representation. We refer as Cir-CEGAR to this variant in the rest of this paper. Since the encoding of our instances is written using the Yices language, we converted our instances to (1) the QDIMACS format used by the QBF solvers using the Yices standard bitblasting procedure, (2) SMTLIB-2 format using a simple syntactic transformation (basically a renaming of bitvector operations), and (3) QDIMACS format with a distinguished special literal equivalent to the validity of the whole formula, as required by Cir-CEGAR. Transformations (1) and (3) were performed after the simplification steps presented in Section III. To assess the effectiveness of Cir-CEGAR, we also produced benchmarks from the original formula, without applying the preprocessing steps. Empirical results are presented at the end of this section.

We modified Yices to incorporate the $\exists\forall$ solver algorithm from Section III. We refer to this modified version as Yices_EF in the results. Cir-CEGAR was implemented using Minisat v2.2 as the underlying SAT solver. When testing the QBF solvers, we first simplified the QBF-instances using Bloqqer [30]. The solvers we used include Z3 v4.3.2, for Linux x64 nightly build downloaded on 2014-05-14, RAReQS v1.1, DepQBF v3.0 and sKizzo v0.8.2. We executed the solvers on a cluster with Intel Xeon E31230 and E5645 processors with a one-hour timeout.

The QBF solvers did not work well on our benchmarks. RAReQS solved only three instances, while DepQBF and sKizzo did not solve any. Therefore, we omit results from these three solvers in the rest of this section.

A. Results

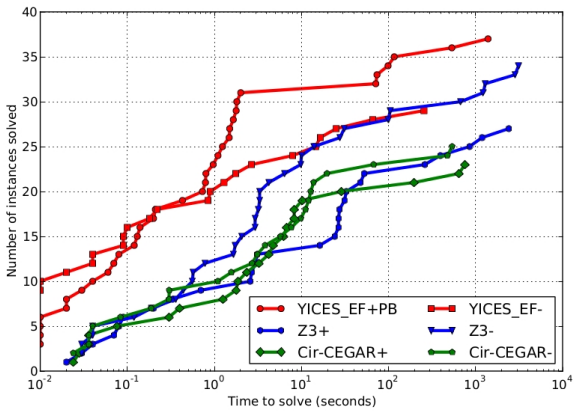

Fig. 3. Comparison of solver performance: Yices, Z3, Cir-CEGAR

Figure 3 shows the number of instances solved by a solver (y-axis) given a particular time limit (x-axis). The encoding of

permutations has a significant impact on solver performance. Our default encoding is explained in the previous section. A permutation σ is defined by a quadratic number of Boolean variables $\sigma_{i,j}$ and constraints such that $\sigma(i) = j \Leftrightarrow \sigma_{i,j}$. This *positive encoding* is denoted by the suffix '+' in the plot. We also experimented with a *negative encoding* (denoted by the suffix '-' in the graph). In the negative encoding, the polarity of the Boolean variables is reversed, that is, we have $\sigma(i) = j \Leftrightarrow \bar{\sigma}_{i,j}$ for each i and j in σ's domain.

The choice of encoding is significant as it interacts with the decision heuristics employed by the SMT solvers. By default, Yices uses negative branching with phase caching [31]. With this heuristic, each time Yices makes a decision on the value of a Boolean variable $\sigma_{i,j}$, it gives preference to the value $false$. This leads to poor performance on benchmarks that use the positive encoding, as setting $\sigma_{i,j}$ to $false$ triggers no unit propagations. After noticing this issue, we changed Yices's branching heuristic to use "positive-branching" (i.e., prefer $true$ over $false$). This is denoted by the suffix PB in the graph. With this setting, Yices solves 37 instances within the time limit. It performs worse with the negative encoding (and the default branching heuristics), solving only 29 instances. In its default configuration, Z3 has better results with the negative encoding. It solves 33 instances with this encoding but only 27 instances with the positive encoding. Cir-CEGAR is not as sensitive to the encoding as Yices and Z3.

Figure 4 shows the benefit of signatures. On the x-axis of each graph we show the time to solve the instance without signatures, while the y-axis is the time to solve the instance with signatures. Most points on these graphs are below the diagonal, showing that adding signatures is a gain in most cases. Many instances cannot be solved within our 3600 s timeout without signatures, but can be solved when signatures are added. The few outliers are instances in which the solver "gets lucky" even without signatures, which happens mostly on satisfiable instances.

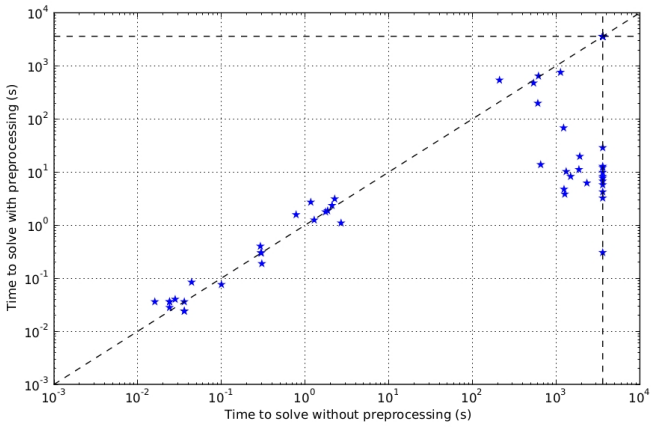

Fig. 5. Improvement in solver performance due to preprocessing. Results are for Cir-CEGAR.

Figure 5 shows the impact of formula simplification presented in Section III on Cir-CEGAR. The x-axis shows the number of seconds taken by the solver when preprocessing is *not* performed on the QBF instances while the y-axis shows the time taken by the solver when preprocessing *is* performed. As before, instances which failed to finish are represented with a value of 3600 seconds. We see that a number of such instances are present on the vertical line with $x = 3600s$. These are instances solved with preprocessing but not when preprocessing was omitted. The behavior is quite interesting. Either preprocessing has little effect on solver performance (the points close to the diagonal) or it has a huge effect (the points where $x > 10^3$ and $y < 10^2$).

VI. Conclusion and Further Work

We have presented the Permutation Independent Conditional Equivalence Checking problem (PICEC) as a method for synthesizing high-level functional descriptions of combinational circuits. PICEC extends permutation independence equivalence checking by considering control signals and conditional matching. We solve the problem using a template-based approach. A template can be seen as describing a (usually very large) family of possible high-level descriptions. Our procedure automatically instantiates the template to match the circuit under investigation. Templates encode partial knowledge about the circuit provided by the user.

PICEC can be reduced to solving formulas in the logic of fixed-sized bit vectors with two levels of quantification $\exists$ and $\forall$— that is, $\exists\forall QF\_BV$. We have implemented a solver for this class of problems using the Yices SMT solver. We have shown that distinguishing signatures are effective to prune the solver search space and lead to significant performance improvement.

We have evaluated this approach on a set of realistic reverse-engineering benchmarks, using different solvers and permutation encodings. Our benchmarks are available to the community in four formats: Yices language, SMT2, QDIMACS, and the QDIMACS format with a special top literal used in in Cir-CEGAR.

An interesting line of further research is in exploring more complex signatures, and efficient algorithms to compute their values. We also plan to investigate whether our pruning approach based on signatures can be included as part of the interaction between the two solvers in the algorithm of Section III.

References

[1] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *HOST*. IEEE, 2012, pp. 83–88.

[2] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *ISCAS*. IEEE, 2010, pp. 2610–2613.

[3] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *DATE*, E. Macii, Ed. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 1277–1280.

[4] W. Li, A. Gascón, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "Wordrev: Finding word-level structures in a sea of bit-level gates," in *HOST*. IEEE, 2013, pp. 67–74.

[5] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[6] J. Mohnke, P. Molitor, and S. Malik, "Establishing latch correspondence for sequential circuits using distinguishing signatures," *Integration*, vol. 27, no. 1, pp. 33–46, 1999.

[7] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification," in *ICCD*. IEEE Computer Society, 1992, pp. 452–458.

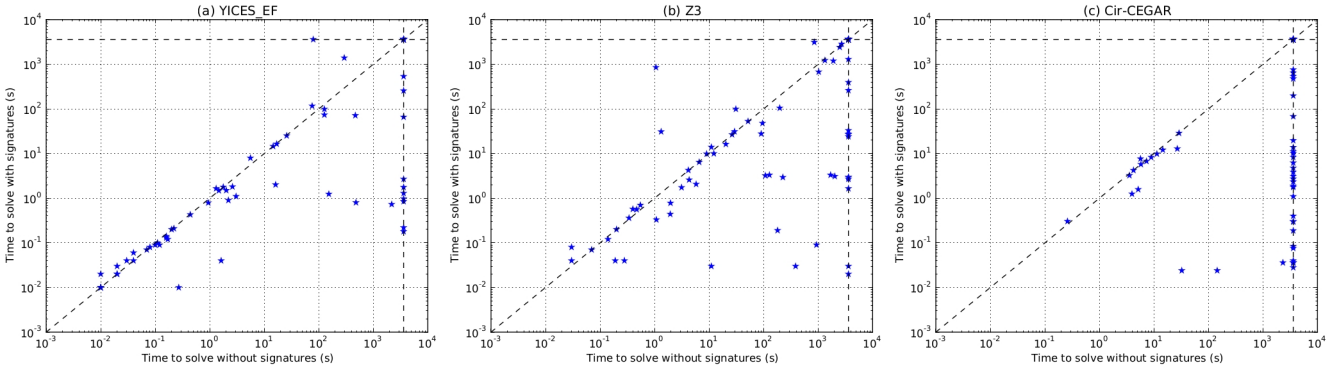

Fig. 4. Improvement in solver performance with signatures.

[8] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," *Formal Methods in System Design*, vol. 10, no. 2/3, pp. 137–148, 1997.

[9] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–17.

[10] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013.

[11] M. W. Hall and D. A. Padua, Eds., *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 2011.

[12] A. Gascón and A. Tiwari, "A synthesized Algorithm for Interactive Consistency," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, J. M. Badger and K. Y. Rozier, Eds., vol. 8430. Springer, 2014, pp. 270–284.

[13] M. Fujita, S. Jo, S. Ono, and T. Matsumoto, "Partial synthesis through sampling with and without specification," in *ICCAD*, J. Henkel, Ed. IEEE/ACM, 2013, pp. 787–794.

[14] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE (1)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 215–224.

[15] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura, "Efficiently solving quantified bit-vector formulas," *Formal Methods in System Design*, vol. 42, no. 1, pp. 3–23, 2013.

[16] "Online repository of benchmarks and experimental results," https://bitbucket.org/spramod/fmcad14-experiments, 2014.

[17] B. Dutertre, "Yices 2 Manual," Computer Science Laboratory, SRI International, Tech. Rep., 2014, available at http://yices.csl.sri.com.

[18] ——, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.

[19] A. Biere, "Resolve and expand," in *Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 59–70.

[20] D. P. Ranjan, D. Tang, and S. Malik, "A Comparative Study of 2QBF Algorithms," in *SAT*, 2004.

[21] M. Janota and J. P. M. Silva, "Abstraction-Based Algorithm for 2QBF," in *SAT*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 230–244.

[22] *Proceedings 1991 IEEE International Conference on Computer Design: VLSI in Computer & Processors, ICCD '92, Cambridge, MA, USA, October 11-14, 1992*. IEEE Computer Society, 1992.

[23] J. Mohnke and S. Malik, "Permutation and phase independent boolean comparison," *Integration*, vol. 16, no. 2, pp. 109–129, 1993.

[24] J. Mohnke, P. Molitor, and S. Malik, "Application of BDDs in Boolean matching techniques for formal logic combinational verification," *STTT*, vol. 3, no. 2, pp. 207–216, 2001.

[25] ——, "Limits of using signatures for Permutation Independent Boolean Comparison," *Formal Methods in System Design*, vol. 21, no. 2, pp. 167–191, 2002.

[26] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[27] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke, "Solving QBF with Counterexample Guided Refinement," in *SAT*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 114–128.

[28] F. Lonsing and A. Biere, "DepQBF: A Dependency-aware QBF Solver," *JSAT*, vol. 7, no. 2-3, pp. 71–76, 2010.

[29] M. Benedetti, "sKizzo: A Suite to Evaluate and Certify QBFs," in *CADE*, ser. Lecture Notes in Computer Science, R. Nieuwenhuis, Ed., vol. 3632. Springer, 2005, pp. 369–376.

[30] A. Biere, F. Lonsing, and M. Seidl, "Blocked Clause Elimination for QBF," in *CADE*, ser. Lecture Notes in Computer Science, N. Bjørner and V. Sofronie-Stokkermans, Eds., vol. 6803. Springer, 2011, pp. 101–115.

[31] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing–SAT 2007*. Springer, 2007, pp. 294–299.