2QBF: Challenges and Solutions

Valeriy Balabanov[1(✉)], Jie-Hong Roland Jiang[2], Christoph Scholl[3], Alan Mishchenko[4], and Robert K. Brayton[4]

[1] Calypto Systems Division, Mentor Graphics, Fremont, USA
balabasik@gmail.com
[2] National Taiwan University, Taipei, Taiwan
jhjiang@ntu.edu.tw
[3] University of Freiburg, Freiburg, Germany
scholl@informatik.uni-freiburg.de
[4] UC Berkeley, Berkeley, USA
{alanmi,brayton}@berkeley.edu

Abstract. 2QBF is a special form $\forall \boldsymbol{x} \exists \boldsymbol{y}.\phi$ of the quantified Boolean formula (QBF) restricted to only two quantification layers, where ϕ is a quantifier-free formula. Despite its restricted form, it provides a framework for a wide range of applications, such as artificial intelligence, graph theory, synthesis, etc. In this work, we overview two main 2QBF challenges in terms of solving and certification. We contribute several improvements to existing solving approaches and study how the corresponding approaches affect certification. We further conduct an extensive experimental comparison on both competition and application benchmarks to demonstrate strengths of the proposed methodology.

1 Introduction

Satisfiability (SAT) solving recently attracted much attention due to its numerous applications in computer science [14]. Some problems (e.g., in the domains of artificial intelligence and games), however, are beyond the reach of SAT solving alone but are naturally expressible in terms of quantified Boolean formulas (QBFs) [18]. 2QBF is a restriction of general QBF problems to just two quantification levels, i.e., to the form $\forall \boldsymbol{x} \exists \boldsymbol{y}.\phi$ or $\exists \boldsymbol{y} \forall \boldsymbol{x}.\phi$, where ϕ is a quantifier-free propositional formula. Despite this restriction many applications can be naturally expressed in 2QBF language [15,16,18]. The main goal of this study is to evaluate and improve scalability of the existing 2QBF methodology to enable QBF as a competitive framework for both existing and new applications.

Recent QBF evaluation showed that there are two main robust algorithms for 2QBF solving: search-based (i.e., QDPLL [13]) and expansion-based (i.e., CEGAR [11,12]). Both have their own strengths and weaknesses, depending on the problem domain, and both can be generally used with either CNF and/or circuit problem representation. Conjunctive normal form (CNF) is a commonly accepted format for propositional satisfiability problems. It is as well extended to QCNF and used to represent QBF problems [3]. It is not an uncommon case,

© Springer International Publishing Switzerland 2016
N. Creignou and D. Le Berre (Eds.): SAT 2016, LNCS 9710, pp. 453–469, 2016.
DOI: 10.1007/978-3-319-40970-2_28

however, that originally QBF is specified on a circuit, rather on a CNF. E.g., in [15] FPGA synthesis benchmarks are formulated on And-Inverter Graphs (AIGs), which is an efficient way to represent general Boolean networks. It is known that any Boolean circuit can be transformed into an equisatisfiable CNF formula, by the various *CNFization* procedures, e.g., by Tseitin transform [20]. The same procedure extends to the QBF context. In this work we shall provide a detailed comparison of different solving techniques over different representations, introduce several algorithmic and implementational improvements, and outline the important observations that must be taken in account by 2QBF users and developers (Sects. 4 and 5).

It is often not enough to only determine the answer to the QBF problem, but also to provide a certificate, that either could be used to prove the validity of the answer, or to be used for other application-specific purposes. The problem of finding certificates with QDPLL solvers was addressed in [4]. In the later sections we are going to show that certificates for 2QBFs have a very restricted form compared to general QBFs, and will show how they could be found using both QDPLL or CEGAR (Sect. 6).

In Sect. 7 we will evaluate all the proposed techniques on the existing competition benchmarks as well as on the application benchmarks, and show that we contribute a significant improvement over the existing 2QBF solvers. Conclusions will be drawn in Sect. 8.

2 Preliminaries

In this work we shall use commonly accepted notations from Boolean algebra and logic. A *Boolean variable* is interpreted over the binary domain $\{0, 1\}$. A *literal* is either a variable or its negation. A *clause* (resp. *cube*) is a disjunction (resp. conjunction) of literals (sometimes we might use set operations on clause/cube literals as well for convenience). A Boolean formula in conjunctive normal form (CNF) is a conjunction of clauses. A Boolean formula in disjunctive normal form (DNF) is a disjunction of cubes. Both CNF and DNF might be a subject to set operations for convenience. As a notational convention, we may also sometimes (where the context allows) omit the conjunction symbol ($\wedge$) and represent negation ($\neg$) by an overline.

For a Boolean formula $\phi(x_1, \ldots, x_i, \ldots, x_n)$, we say its positive (resp. negative) cofactor with respect to variable x_i, denoted $\phi|_{x_i}$ (resp. $\phi|_{\overline{x}_i}$), is the formula $\phi(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ (resp. $\phi(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$). The cofactor definition also extends to a cube of literals $\alpha = l_1 \wedge \cdots \wedge l_m$, with the following recursive definition $\phi|_\alpha = (\phi|_{\alpha \setminus l_m})|_{l_m}$, where $\phi|_{l_m}$ is a positive (resp. negative) cofactor with respect to variable $var(l_m)$ if l_m is a positive (resp. negative) literal. If the context allows we may as well drop the vertical line and simply write ϕ_α. We say that formula ϕ is satisfiable if there is an assignment to its variables that evaluates ϕ to true. We say ϕ is unsatisfiable otherwise. We denote $ON(\phi)$ the *onset* of ϕ, i.e., the set of assignments under which ϕ evaluates to true. We define an *unsatisfiable core* of an unsatisfiable CNF formula ϕ as an arbitrary unsatisfiable subset of clauses of ϕ.

A *quantified Boolean formula* (QBF) Φ over universal variables $\boldsymbol{x} = \{x_{11}, \ldots, x_{ki_k}\}$ and existential variables $\boldsymbol{y} = \{y_{11}, \ldots, y_{kj_k}\}$ in *prenex form* is of the form $\Phi = \forall x_{11} \ldots x_{1i_1} \exists y_{11} \ldots y_{1j_1} \ldots \forall x_{k1} \ldots x_{ki_k} \exists y_{k1} \ldots y_{kj_k}.\phi$ where the quantification part is called the *prefix*, denoted Φ_{pfx}, and ϕ, a quantifier-free formula in terms of variables $\boldsymbol{x}$ and $\boldsymbol{y}$, is called the *matrix*, denoted Φ_{mtx}. Further Φ is said to be a k-QBF, or a QBF with k quantification levels. 2QBF (resp. 3QBF) is a special case of QBF with $k = 2$ (resp. $k = 3$).

We say that 2QBF $\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi$ is false if there is an assignment $\alpha_{\boldsymbol{x}}$ to $\boldsymbol{x}$ variables (also referred to as the winning strategy for the universal player or a constant Herbrand-functions countermodel) such that $\Phi_{\text{mtx}} \mid_{\alpha_{\boldsymbol{x}}}$ (which is a function of $\boldsymbol{y}$ variables) is unsatisfiable. We say Φ is true otherwise (i.e., a winning strategy for the universal player does not exist). Alternatively, 2QBF $\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi$ is true if and only if there exists a set of the so-called Skolem functions $S_{\boldsymbol{y}}(\boldsymbol{x})$ that renders ϕ to a tautology (i.e., $\phi(\boldsymbol{x}, S_{\boldsymbol{y}}(\boldsymbol{x})) \equiv 1$). For more general 3QBF $\Phi = \exists \boldsymbol{w} \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y})$, it is true if and only if there exists a set of constant functions $S_{\boldsymbol{w}}$, and a set of functions $S_{\boldsymbol{y}}(\boldsymbol{x})$ (i.e., depending on $\boldsymbol{x}$), such that $\phi(S_{\boldsymbol{w}}, \boldsymbol{x}, S_{\boldsymbol{y}})$ is a tautology. Here $S_{\boldsymbol{w}}$ and $S_{\boldsymbol{y}}$ form the Skolem-functions model, certifying the validity of Φ. There are other forms of certificates (e.g., Q-resolution). For more details on general QBF solving and certification please refer to [4,5,8,13].

3 Overview of Prior Work

Among other possibilities, there are two main approaches to QBF solving: search based and expansion based. The former is referred to the QDPLL style search algorithm, and the latter to the counterexample guided abstraction refinement (CEGAR). Recent 2QBF evaluation showed that CEGAR solvers are generally more robust than QDPLL solvers. In this work, we focus on the CEGAR-based approach, and will provide an intuition later how it is more beneficial than QDPLL.

Figure 1 outlines the generic CEGAR-based algorithm `Cegar2QBF` for solving an arbitrary 2QBF formula in the prenex form, introduced in [11]. In Line 1, two SAT solving managers, i.e., *synthesis manager* `synMan` (to guess a candidate winning move of the universal player) and *verification manager* `verMan` (to verify if the guessed move of the universal player is indeed winning), are initialized. Initially `synMan` contains variables $\boldsymbol{x}$ and `verMan` contains variables $\boldsymbol{x}$ and $\boldsymbol{y}$. In Line 3, we search for a candidate winning move $\alpha_{\boldsymbol{x}}$. If all candidates have been blocked, the 2QBF is determined to be true in Line 4. In Line 5, a counterexample to the candidate winning move is searched, which renders ϕ true (i.e., it disproves the candidate winning move $\alpha_{\boldsymbol{x}}$). Note that if CNF is used as an underlying data structure for `verMan`, then $\alpha_{\boldsymbol{x}}$ can be easily passed to the SAT solver via its assumptions interface (which is commonly available in modern SAT solvers, e.g., in MINISAT [7]). If no counterexample can be found, the QBF is determined to be false in Line 6. Otherwise, cofactor $\phi|_{\alpha_{\boldsymbol{y}}}$ is performed in Line 7, and block all the known wrong candidates $\boldsymbol{x}'$, such that $\phi|_{\alpha_{\boldsymbol{y}}}(\boldsymbol{x}') = 1$, in Line 8.

Algorithm **Cegar2QBF**
input: a QBF $\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi$
output: True or False
begin
01 `synMan`$[\boldsymbol{x}] := 1$; `verMan`$[\boldsymbol{x}, \boldsymbol{y}] := \phi$;
02 **while** True
03 $\alpha_{\boldsymbol{x}} :=$ SatSolve(`synMan`);
04 **if** $\alpha_{\boldsymbol{x}} = \emptyset$ **then return** True;
05 $\alpha_{\boldsymbol{y}} :=$ SatSolve(`verMan`, $\alpha_{\boldsymbol{x}}$);
06 **if** $\alpha_{\boldsymbol{y}} = \emptyset$ **then return** False;
07 `negCof` $:= \neg\phi|_{\alpha_{\boldsymbol{y}}}$;
08 `synMan` $:=$ `synMan` $\wedge$ `negCof`;
end

Fig. 1. CEGAR algorithm for generic 2QBF solving.

The above CEGAR algorithm differs from QDPLL-based algorithms [9,13] in that in QDPLL `negCof` is simply substituted with $\neg\alpha'_{\boldsymbol{x}}$, where $\alpha'_{\boldsymbol{x}}$ is obtained from $\alpha_{\boldsymbol{x}}$ by a single minimal hitting set generalization, to block the failed candidate within `synMan`. The strength of `Cegar2QBF` is in that besides $\alpha'_{\boldsymbol{x}}$ it potentially blocks several other hitting set assignments.

Algorithm `Cegar2QBF` may be applied to an arbitrary 2QBF in prenex form regardless of the representation of its matrix. Some QBF solvers use And-Inverter graphs (AIGs) as an underlying matrix structure [15,17]. On the other hand, as CNF has been proven to be an efficient data structure for SAT solving (e.g., due to an efficient representation of learnt information in the form of learnt clauses [7]), CNF is also the most commonly accepted QBF matrix representation format [3]. We therefore distinguish between *CNF* and *circuit* 2QBF solvers, depending on the matrix input format that they accept. Both CNF and circuit representations have their advantages and disadvantages, which will be discussed in later sections.

Below we mention the idea of Qesto [12] for efficient implementation of CEGAR-based 2QBF solving on CNF matrix. Note that `verMan` is initialized to ϕ in Line 1 of Fig. 1 and is never changed, while `synMan` is constantly changed by conjunction with `negCof` in Line 8. If the matrix is already represented in CNF, then the main complication of the algorithm is the CNFization of `negCof` prior to conjunction with `synMan`. The approach of [11] suggested to use Tseitin transform [20] to perform a syntactic negation, at the cost of introducing fresh variables. This approach, however, suffers from variable blow up within `synMan` after a large number of iterations. In Qesto, the variable blow-up problem is overcome by efficient representation of a larger number of cofactors within `synMan` as follows. Consider a matrix $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where each C_i is split into existential literals C_{ei} and universal literals C_{ui}. Notice that regardless of the

specifics of the assignment $\alpha_{\boldsymbol{y}}$, `negCof` always takes the form $\neg C_{uj_1} \vee \neg C_{uj_2} \vee \cdots \vee \neg C_{uj_k}$. By defining $d_i \equiv \neg C_{ui}$ for each $i \in [1..n]$, one can conveniently represent `negCof` as a clause $(d_{j_1} \vee d_{j_2} \vee \cdots \vee d_{j_k})$. Consequently, at most n fresh variables need to be introduced, independent of the number of iterations of the while-loop in Fig. 1. Under this scenario, the algorithm in Fig. 1 can be modified to initialize the synthesis manager by

$$\texttt{synMan}[\boldsymbol{x}, \boldsymbol{d}] := \bigwedge\nolimits_{i \in [1..n]} (d_i \equiv \neg C_{ui}).$$

Qesto was experimentally shown to be superior to others [12]. In the next section we will examine the advantages and disadvantages of CNF CEGAR-based 2QBF solving, and introduce a heuristics, inspired by Qesto, to improve existing circuit CEGAR-based 2QBF algorithms.

4 Heuristics for CNF and Circuit 2QBF Solving

In this section we introduce several implementation improvements to the Qesto algorithm and also show how the Qesto CNF ideas can be lifted to the circuit 2QBF solving.

4.1 Improvements for CNF-based 2QBF Solving

The following three observations could be used to enhance the performance of the Qesto implementation of the `Cegar2QBF` algorithm:

1. In case $C_{uj_1} = x_i$ (i.e., C_{j_1} is a universally unit clause), there is no need to introduce a fresh variable for this clause, but rather just using x_i itself in computing `negCof`. Note that this condition may occur quite often if CNF was obtained through the Tseitin transform. For example AND-gate $c = a \wedge b$, where c is existential and both a and b are universal, will be transformed into clauses $(\overline{c} \vee a)(\overline{c} \vee b)(c \vee \overline{a} \vee \overline{b})$, therefore producing two universally unit clauses.
2. In case $C_{uj_1} = C_{uj_2}$, there is no need to introduce two definition variables d_1 and d_2, but just one. Similar to the previous case, this may happen after Tseitin transform, e.g., if the universally quantified primary input had several fanouts in the circuit.
3. In case $C_{uj_1} \subseteq C_{uj_2}$ and $\texttt{negCof} = d_{uj_1} \vee d_{uj_2} \vee \cdots$, we can simply drop C_{uj_1} from `negCof` because of the logical implication $d_{uj_1} \rightarrow d_{uj_2}$. This often may be seen after existential variable elimination, which is a common preprocessing technique in both QBF and SAT solving. For example given QBF

   $$\forall ab \exists cde \,.\, (\overline{c} \vee a)_1 (c \vee \overline{a})_2 (d \vee \overline{b} \vee \overline{c})_3 (\overline{d} \vee b)_4 (\overline{d} \vee c)_5 (a \vee e)_6 (b \vee \overline{e})_7,$$

   after elimination of variable c we get QBF

   $$\forall ab \exists de \,.\, (d \vee \overline{b} \vee \overline{a})_3 (\overline{d} \vee b)_4 (\overline{d} \vee a)_5 (\overline{a} \vee e)_6 (\overline{b} \vee \overline{e})_7,$$

   where clause $C_{u3} = (\overline{b} \vee \overline{a})$ now subsumes both $C_{u6} = \overline{a}$ and $C_{u7} = \overline{b}$ (note that C_{u3} does not trigger any of the first two heuristics).

The first two enhancements are simpler, and intend to decrease the number of definitions added to the `synMan` at the initialization step. As a consequence, underlying SAT solver has to deal with less variables, and hopefully find the solution faster. On the other hand the third enhancement does not change the number of definitions, but rather simplifies the blocking constraint `negCof` by observing that some literals could be effectively dropped from the underlying blocking clause. As learned clause size is an important criteria in SAT solving, we speculate that this could also potentially speed up the solving process. All the three refinements are to be evaluated in Sect. 7.

4.2 Improvements for AIG-based 2QBF Solving

The main idea behind Qesto could be formulated in a different manner: the cofactors computed in Line 7 of algorithm `Cegar2QBF` on Fig. 1 have a lot in common. If certain universal subclause C_{ui} is present in several cofactors then the Tseitin definition for this clause could be reused multiple times when performing the CNFization of the complemented cofactors. This idea could be efficiently lifted to the circuit solvers. Following the notion of *structural hashing* of nodes in And-Inverter graphs (AIGs), we propose algorithm `AigShare2Qbf` as sketched in Fig. 2 for circuit-based 2QBF solving.

The core CEGAR procedure of algorithm `AigShare2Qbf` is the same as that of `Cegar2QBF`. Furthermore `Cegar2QBF` may use AIGs as the underlying data structure of its verification manager and the negated cofactors as well. The only difference is the *cofactor hashing* heuristics in lines 8 and 9 of Fig. 2.

Algorithm **AigShare2Qbf**
input: a QBF $\Phi = \forall X \exists Y.\phi$
output: True or False
begin
01 `synMan`$[X] := 1$; `verMan`$[X, Y] := \phi$; `aigMan` $:= \emptyset$;
02 **while** True
03 $\alpha_X :=$ SatSolve(`synMan`);
04 **if** $\alpha_X = \emptyset$ **then return** True;
05 $\alpha_Y :=$ SatSolve(`verMan`, α_X);
06 **if** $\alpha_Y = \emptyset$ **then return** False;
07 `negCof` := AIG($\neg\phi|_{\alpha_Y}$);
08 `newAnd` := NotHashed(`negCof`,`aigMan`);
09 Hash(`aigMan`, `newAnd`);
10 `synMan` := `synMan` $\wedge$ CNF(`newAnd`);
end

Fig. 2. CEGAR algorithm for AIG-based 2QBF solving with cofactor sharing heuristics.

More specifically, in Line 8 we extract a subset `newAnd` of AND gates from `negCof` that have not been hashed previously. Then in Line 9 we hash them and add them to the AIG manager `aigMan`. In Line 10 we add CNFized `newAnd` gates to `synMan`. For the AND gates in Line 8 which have been hashed previously, `synMan` already contains clauses for the corresponding definitions. As to be seen from experiments, cofactor sharing heuristic gives a significant improvement on the benchmarks where a large number of iterations are needed for algorithm `Cegar2QBF` to converge.

5 CNF Versus Circuit Solvers

In this section we compare the CNF and circuit 2QBF solvers. We outline the strengths of one over the other, and address the question in which applications which one should be used.

Given a non-CNF formula ϕ (e.g., represented as an AIG) one could *CNFize* it (i.e., transform it to an equisatisfiable CNF formula, for example by using the Tseitin transform [20]) to get ϕ_{CNF}, and then run the QESTO algorithm introduced in Sect. 3. However, this approach could be inefficient as the following example suggests.

Example 1. Consider the following simple true 2QBF formula $\Phi = \forall abc\ \exists d\ .\ \phi$, where ϕ is given as a *nested XOR tree* circuit $\phi = a \oplus b \oplus c \oplus d$. Assume that the synthesis manager in algorithm `Cegar2QBF` of Fig. 1 comes up with a candidate $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}\overline{c}$ (i.e., assigns $a = b = c = 0$), and the verification manager returns a counterexample $\alpha_{\boldsymbol{y}} = d$ (i.e., assigns $d = 1$). Note that in this case $\texttt{negCof}_1 = a \oplus b \oplus c$. After the second iteration with, e.g., $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}c$ and $\alpha_{\boldsymbol{y}} = \overline{d}$, we have $\texttt{negCof}_2 = \neg(a \oplus b \oplus c)$, thus blocking all the universal candidate assignments, and conclude that Φ is true. The same number of iterations would be required for an arbitrary large nested XOR tree.

In contrast, consider the same 2QBF, but CNFized by Tseitin transform prior to solving, introducing definitions $x = a \oplus b$ and $y = x \oplus c$, and resulting into QBF

$$
\begin{aligned}
\Phi_{CNF} =& \forall abc\ \exists d\ \exists xy\ .\ \phi_{CNF},\ \text{where} \\
\phi_{CNF} =& (x \vee \overline{a} \vee b)(x \vee a \vee \overline{b})(\overline{x} \vee a \vee b)(\overline{x} \vee \overline{a} \vee \overline{b}) \\
& \wedge (y \vee \overline{x} \vee c)(y \vee x \vee \overline{c})(\overline{y} \vee x \vee c)(\overline{y} \vee \overline{x} \vee \overline{c}) \\
& \wedge (y \vee d)(\overline{y} \vee \overline{d}).
\end{aligned}
$$

Suppose that the synthesis manager in algorithm `Cegar2QBF` guesses the same candidate $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}\overline{c}$, and the verification manager replies with a counterexample $\alpha_{\boldsymbol{y}} = d\overline{x}\overline{y}$. In this case we compute $\texttt{negCof}_1 = (a \oplus b) \vee c$. After the second iteration with $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}c$ and $\alpha_{\boldsymbol{y}} = \overline{d}\overline{x}y$, we have $\texttt{negCof}_2 = (a \oplus b) \vee \neg c$, thus resulting in formula $a \oplus b$ in the synthesis manager. Without any conclusion, we have to proceed with further iterations. In fact, for a general nested XOR tree the computation may require an exponential number of iterations to terminate.

The rationale behind Example 1 is as follows. Whenever the counterexample (Line 5 in Fig. 1) is computed under the CNF representation, it fixes a value assignment to the auxiliary variables (i.e., the intermediate variables introduced during CNFization). The negated cofactor (Line 7 in Fig. 1) in this case will only block X assignments that respect the auxiliary variable assignment. This phenomenon is summarized in Proposition 1.

Proposition 1. *Given a 2QBF $\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi$, with ϕ represented as a circuit, let $\alpha_{\boldsymbol{y}}$ be an assignment to variables $\boldsymbol{y}$ and $\alpha_{\boldsymbol{x}_1}$ and $\alpha_{\boldsymbol{x}_2}$ be two assignments to variables $\boldsymbol{x}$ and let $g = G(\boldsymbol{x}, \boldsymbol{y})$ be such an intermediate gate in the circuit of ϕ, such that*

$$\phi|_{\alpha_{\boldsymbol{y}}\alpha_{\boldsymbol{x}_1}} \ \wedge \ \phi|_{\alpha_{\boldsymbol{y}}\alpha_{\boldsymbol{x}_2}} \ \wedge \ G|_{\alpha_{\boldsymbol{y}}\alpha_{\boldsymbol{x}_1}} \oplus G|_{\alpha_{\boldsymbol{y}}\alpha_{\boldsymbol{x}_2}}.$$

(That is, the output of ϕ evaluates to the same value under the two input assignments, but there is an internal gate disagreement for the two assignments.) Then $\alpha_{\boldsymbol{x}_1}$ and $\alpha_{\boldsymbol{x}_2}$ will be blocked within the same *iteration computing counterexample $\alpha_{\boldsymbol{y}}$ in Line 5 of Fig. 1, if algorithm* `Cegar2QBF` *is applied to Φ. On the other hand, $\alpha_{\boldsymbol{x}_1}$ and $\alpha_{\boldsymbol{x}_2}$ will not be blocked within the same iteration, if* `Cegar2QBF` *is applied to a CNFized version (by Tseitin transform) of Φ.*

Example 1 and Proposition 1 show that flattening the circuit structure into CNF affects the 2QBF solving process more than it does for propositional SAT. In theory one could avoid cofactoring on Tseitin variables, and modify the algorithm `Cegar2QBF` accordingly to eliminate the problem described in Proposition 1. In practice, however, CNF based QBF solvers can not easily distinguish auxiliary variables from the original primary inputs quantified in the same quantification layer. It is therefore advised to use CNF for underlying SAT queries (e.g., for efficient clause learning), while cofactoring on circuit level instead of CNF. As will be confirmed experimentally later, the gained reduction in number of iterations needed for completion of algorithm `Cegar2QBF` even overcomes the benefits of efficient cofactor representation in QESTO algorithm applied after circuit CNFization.

Please note that in circuit 2QBF solvers the SAT queries in Line 3 of algorithm Cegar2QBF (Fig. 1) are made to a CNF-based SAT solver as well. This choice is caused by a specific "incremental" nature of the underlying SAT calls: Please recall that synthesis manager `synMan` is updated by iterative conjunction with `negCof` in Line 8 of Fig. 1, i.e., in each iteration clauses from the CNFized `negCof` are added to manager `synMan`. Therefore it will be highly benefitial to use the learned information from the previous solving iterations in the later ones.

Despite the ineffectiveness of CNF 2QBF solvers compare to their circuit counterparts, we speculate that there is a better encoding of a circuit 2QBF problem into QCNF, as is described below. Given circuit 2QBF formula $\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi(\boldsymbol{x}, \boldsymbol{y})$, instead of computing its CNFized version $\Phi_{CNF} = \forall \boldsymbol{x} \ \exists \boldsymbol{y} \ \exists \boldsymbol{t} \ . \ \phi_{CNF}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{t})$ we obtain it's negation as $\Phi' = \neg\Phi = \exists \boldsymbol{x} \forall \boldsymbol{y}.\phi'(\boldsymbol{x}, \boldsymbol{y})$, with $\phi' = \neg\phi$. Now the translation to an equisatisfiable CNF can be done as following: $\Phi'_{CNF} = \exists \boldsymbol{x} \ \forall \boldsymbol{y} \ \exists \boldsymbol{t} \ . \ \phi'_{CNF}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{t})$. By construction, the truth of Φ could be determined as an inverse of Φ'_{CNF}. Further any model (resp. countermodel) for Φ'_{CNF} could be mapped to corresponding model (resp. countermodel) for Φ.

Note that Φ'_{CNF} above is a 3QBF formula, rather than 2QBF. Despite the increase in the number of quantifier alternations (which in general correlates with the formula complexity), Φ'_{CNF} may be much easier to solve compare to Φ_{CNF}. The intuition here is that innermost quantification level in Φ'_{CNF} contains only Tseitin variables. Values for these variables shall be uniquely determined upon the assignments to variables $\boldsymbol{x}$ and $\boldsymbol{y}$.

Generally CEGAR based algorithm for 3QBF is slightly more complex than for 2QBF, but let us briefly outline how the above procedure applies to the earlier Example 1. First compute Φ'_{CNF} in the same way as it was done for Φ_{CNF}:

$$\begin{aligned}\Phi'_{CNF} =& \exists abc\ \forall d\ \exists xy\ .\ \phi'_{CNF},\ \text{where}\\ \phi'_{CNF} =& (x \vee \overline{a} \vee b)(x \vee a \vee \overline{b})(\overline{x} \vee a \vee b)(\overline{x} \vee \overline{a} \vee \overline{b})\\ & \wedge (y \vee \overline{x} \vee c)(y \vee x \vee \overline{c})(\overline{y} \vee x \vee c)(\overline{y} \vee \overline{x} \vee \overline{c})\\ & \wedge (\overline{y} \vee d)(y \vee \overline{d}).\end{aligned}$$

It is worthy to mention that the negation on circuit level is a very cheap operation, and as one can see Φ'_{CNF} differs from Φ_{CNF} only by the last two clauses. Let us now examine candidate solution $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}\overline{c}$, and the (potential) counterexample to this solution $\alpha_{\boldsymbol{y}} = d$. First observe that values of x and y are uniquely determined to be $x = y = 0$. Second, under the completed assignment ϕ' evaluates to false. So at this point no further computations are required to conclude that $\alpha_{\boldsymbol{y}} = d$ is a counterexample to candidate $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}\overline{c}$. To block this candidate we basically add formula $\texttt{negCof}_1 = \neg\phi'_{CNF}|_d$ to the outermost existential solver, and rename variables x, y to x_1, y_1. After the second iteration with $\alpha_{\boldsymbol{x}} = \overline{a}\overline{b}c$ and $\alpha_{\boldsymbol{y}} = \overline{d}$ we shall again conclude invalidity of $\alpha_{\boldsymbol{x}}$ and block it by formula $\texttt{negCof}_2 = \neg\phi'_{CNF}|_{\overline{d}}$, where variables x, y are renamed to x_2, y_2. Intuitively $\texttt{negCof}_1(a, b, x_1, y_1)$ and $\texttt{negCof}_2(a, b, x_2, y_2)$ are the CNFized versions of circuit cofactors $\phi|_d$ and $\phi|_{\overline{d}}$. One could verify that $\texttt{negCof}_1(a, b, x_1, y_1) \wedge \texttt{negCof}_2(a, b, x_2, y_2)$ is unsatisfiable. Generally speaking, solving 3QBF Φ'_{CNF} should not take more CEGAR iterations than circuit 2QBF Φ. On the other hand each iteration could be more inefficient due to the naive cofactoring and variable renaming. In circuit solving this can be done more efficiently using e.g., cofactor sharing heuristics introduced in the previous section. This phenomenon is to be examined in the experimental section.

6 Certificate Generation for 2QBF

As mentioned previously, recent evaluation on QBF solvers suggested the superiority of CEGAR-based QBF solvers. In contrast to search-based approaches (e.g., DepQBF [13]), however, there exists no methodology to certify their answer with semantic winning strategies in a closed form (e.g., Skolem-functions for true QBFs, which are essential for many QBF applications). CEGAR-based QBF solver RAReQS [10], can produce partial winning moves for both existential and universal players at each turn of an abstraction-refinement game [10]. One straightforward use of this ability is that RAReQS returns the winning

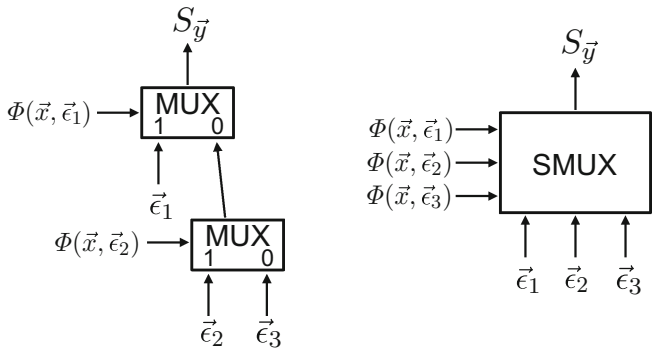

Fig. 3. Multiplexer construction [on the left], SMUX cell [on the right].

assignment to the outermost existential (resp. universal) variables for true (resp. false) QBFs upon completion. In this section we describe how to construct Skolem/Herbrand functions for 2QBFs, based on the partial winning-move information deduced from CEGAR-based QBF solvers. We present several heuristics to enhance the construction procedure as well as the produced certificate quality.

6.1 Construction Procedure

Consider a true 2QBF formula Φ. Assume that the CEGAR 2QBF solver needs three, say, refinement iterations to prove its validity. It means that three candidate solutions for the universal player were found, leading to existential counterexamples $\boldsymbol{\epsilon}_1$, $\boldsymbol{\epsilon}_2$, and $\boldsymbol{\epsilon}_3$. Let Φ_{cof} be the formula refined by the three corresponding cofactors as shown below.

$$\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi(\boldsymbol{x},\boldsymbol{y}) \qquad \Phi_{cof} = \forall \boldsymbol{x}.\{\phi(\boldsymbol{x},\boldsymbol{\epsilon}_1) \vee \phi(\boldsymbol{x},\boldsymbol{\epsilon}_2) \vee \phi(\boldsymbol{x},\boldsymbol{\epsilon}_3)\}$$

Now the search for a candidate solution fails, i.e., the universal player does not find a candidate solution which falsifies all cofactors generated so far. Consequently Φ_{cof} is true, which is determined by an unsatisfiable SAT call $\neg\Phi_{cof}$ (which is propositional as it has only existentially quantified variables $\boldsymbol{x}$).

Effectively, validity of Φ_{cof} says that an arbitrary assignment to $\boldsymbol{x}$ is included in $ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_1))$, $ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_2))$, or $ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_3))$. This information in fact is sufficient to get Skolem functions $S_{\boldsymbol{y}}(\boldsymbol{x})$ for any assignment $\boldsymbol{\alpha}$ to $\boldsymbol{x}$, by the following steps:

1. For all $\boldsymbol{\alpha} \in ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_1))$, define $S_{\boldsymbol{y}}(\boldsymbol{\alpha}) = \boldsymbol{\epsilon}_1$.
2. For all $\boldsymbol{\alpha} \in ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_2)) \setminus ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_1))$, define $S_{\boldsymbol{y}}(\boldsymbol{\alpha}) = \boldsymbol{\epsilon}_2$.
3. For all $\boldsymbol{\alpha} \in ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_3)) \setminus (ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_1)) \cup ON(\phi(\boldsymbol{x},\boldsymbol{\epsilon}_2)))$, define $S_{\boldsymbol{y}}(\boldsymbol{\alpha}) = \boldsymbol{\epsilon}_3$.

The above computation of the Skolem functions is visualized by a multiplexer construction as shown on the left of Fig. 1. We abbreviate the multiplexer

construction by a cell "SMUX" which means that we have a series of multiplexers defining a prioritization in case that the sets $ON(\Phi(\boldsymbol{x}, \boldsymbol{\epsilon}_i))$ overlap. SMUX cell is shown on the right of Fig. 1. By the following proposition we ensure the soundness of returned Skolem functions Figs. 3 and 5 .

Proposition 2. *The functions $S_{\boldsymbol{y}}(\boldsymbol{x})$ constructed by the above procedure form a valid model for Φ.*

Proof. Since Φ_{cof} is true, for every assignment $\boldsymbol{\alpha}$ to $\boldsymbol{x}$, some $\phi(\boldsymbol{\alpha}, \boldsymbol{\epsilon}_i))$, $i \in [1..3]$, must be true. Our construction ensures that $S_{\boldsymbol{y}}(\boldsymbol{\alpha}) = \boldsymbol{\epsilon}_i$, i.e., that $\phi(\boldsymbol{\alpha}, S_{\boldsymbol{y}}(\boldsymbol{\alpha}))$ evaluates to true. By definition, the functions $S_{\boldsymbol{y}}$ form a valid set of Skolem functions. □

The proposed procedure can be easily extended to true 2QBFs with an arbitrary number of refinement steps, just by replacing three cofactors from the above example with the cofactors returned by the solver. Further, the method can be extended for true 3QBFs as follows. Suppose we are given a true 3QBF $\Phi = \exists \boldsymbol{w} \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y})$, and a winning move (assignment) $\boldsymbol{\beta}$ for $\boldsymbol{w}$ variables (which is returned upon completion of RAReQS as a by-product of solving process). Because the 2QBF $\Phi = \forall \boldsymbol{x} \exists \boldsymbol{y}.\phi(\boldsymbol{\beta}, \boldsymbol{x}, \boldsymbol{y})$ must be true, Skolem functions $S_{\boldsymbol{y}}$ can be extracted using the previous method for 2QBFs. The complete Skolem model now consists of $\{S_{\boldsymbol{w}} = \boldsymbol{\beta}, S_{\boldsymbol{y}}\}$. Although to achieve efficient implementation of this extension is slightly more sophisticated than the 2QBF case, the essential idea is as described above.

6.2 Certificate Optimization

Below we propose three optimization techniques in order to minimize the certificates returned by the proposed Skolem-function construction procedure.

1. Observe that any reordering of the multiplexers in a SMUX cell still maintains a valid set of Skolem functions. In our implementation, we allow an option to use cofactors either in a forward or backward order with respect to the derivation sequence of the corresponding winning moves. The backward order turns out to be empirically superior.
2. It was observed that cofactors often share identical clauses. Therefore we implement a hashing procedure that detects and substitutes repeating clauses.
3. Although each next counterexample returned by CEGAR QBF solving approach covers at least one new (so-far unblocked) universal winning move candidate, in practice it happens that older cofactors are fully covered by newer ones. The problem of identifying redundant cofactors can be done using the so-called group minimal unsatisfiability subset extraction (group MUS, or GMUS). The GMUS framework partitions the clauses of a CNF formula into groups (cofactors in our case), and returns the minimal subset of them, which is still unsatisfiable (which is clearly a requirement for our extracted Skolem functions to be sound). More information on GMUS extraction can be found at [19].

7 Experimental Results

We performed experimental evaluation of various 2QBF solving and certification techniques both on the competition as well as application benchmarks. The experiments are divided into the solver performance testing and the certificate quality testing. To test solvers' performance, we used two sets of benchmarks: 2QBF track of QBFEVAL'10 [2] QBF competition formulas and FPGA technolgy mapping benchmarks from [15]. For certificate generation, due to the few number of true 2QBFs in QBFEVAL'10, we used the application track benchmarks of QBF Gallery 2014 [1].[1]

7.1 2QBF Solving

2QBF Track of QBFEVAL'10 [2] . We used CEGAR-based QBF solvers RAReQs [11] and Qesto [12] for comparison. In this section we refer to Qesto as to the tool from [12], rather than to the algorithm itself. Please note that according to [12], tool Qesto dominates other 2QBF solvers, including both QDPLL-based and circuit-based solvers. Since the source code for Qesto is not publicly available, we have reimplemented its simplified 2QBF version (further referred to as Mini2qbf) on top of the Minisat SAT solver [7]) to test the enhancements proposed in Sect. 4.

The benchmark suit contains 200 QBFs in prenex CNF from the 2QBF track of QBFEVAL'10. Preprocessor Bloqqer [6] was used to preprocess the formulas, and 135 formulas solved directly by Bloqqer were excluded from further experiments. All CNF-based solvers, namely RAReQS, Qesto, and Mini2qbf, ran on the preprocessed benchmarks. To compare against the circuit 2QBF solvers, we implemented a tool minicnf2blif to extract circuits from unpreprocessed CNF formulas, and then ran the existing circuit 2QBF solver (command "&qbf") embedded into the synthesis tool ABC [15] under two settings: without cofactor sharing (referred to as ABC-) and with cofactor sharing (referred to as ABC+). A third version, called PreABC+ is similar to ABC+, with the only difference that the QBF preprocessor Bloqqer is used for preprocessing the QBFs where minicnf2blif did not find any circuit structure (after preprocessing the resulting CNF formulas are then just translated to product-of-sums circuits and then solved by ABC+). All the solvers were limited by the 4GB memory and 1200 second time limit.

Table 1 summarizes the results. A cactus plot of time performance is shown in Fig. 4. The extraction time of minicnf2blif was negligible compared to solving, and we omitted reporting it. From Table 1 and Fig. 4, we see that the circuit-based 2QBF solver PreABC+ outperforms existing CNF based solvers. On the 17 benchmarks where no circuit structure was found, PreABC+ was on average 3 times slower compared to Qesto. The remaining 48 benchmarks were found highly structural. On these ABC+ was on an average of 28 times faster than

[1] All the tools, benchmarks, and experimental results can be found at https://www.dropbox.com/s/xyfb2i9xl1pvrv3/sat16_tools_2qbf.zip?dl=0.

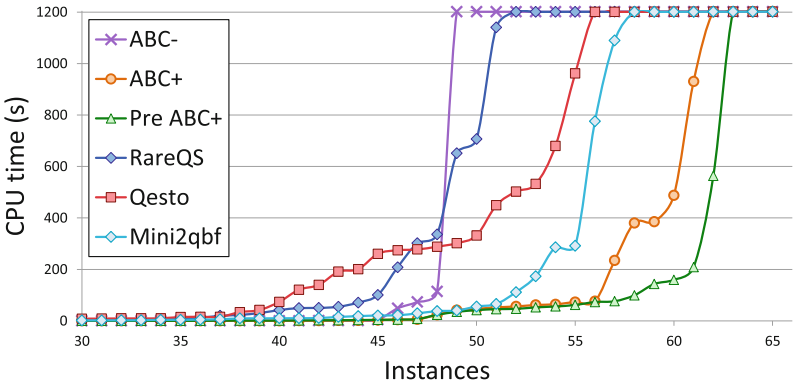

Fig. 4. Cactus plot of solvers performance on QBFEVAL'10 2QBF benchmark set.

Table 1. Statistics for 2QBF track from QBFEVAL'10.

	RAReQS	Qesto	Mini2qbf	ABC-	ABC+	PreABC+
Solved	50	55	57	48	61	62
Time, s	20658	17842	12725	20677	7748	4124
Iterations	NA	7.26M	NA	46.0K	368K	153K

Qesto. ABC+ in comparison to ABC- was on average 30 % faster; however if we only consider problems solved within more than 100 iterations (or alternatively solved in about more than 1 second) cofactor sharing gives about an order of magnitude speed up. This phenomenon is well explained by the fact that within first few iterations cofactor sharing occurs rarely, while on the larger scale AIG nodes from the new cofactors are found to be previously hashed practically all the time. On the other hand we can also see that our reimplementation of Qesto algorithm performs quite well too. If we switch off the last heuristic from Sect. 4, Mini2qbf solves 2 instances less and is of similar performance to Qesto (the first two heuristics from Sect. 4 are hardcoded so we cannot switch them on or off).

FPGA Mapping Benchmarks From [15] . We picked 100 (50 SAT and 50 UNSAT) 2QBF benchmarks from an FPGA mapping application [15]. For the comparison with CNF solvers we encoded these benchmarks into both 2QBF and negated 3QBF forms (introduced in Sect. 4). Original AIG circuits have 50 primary inputs and 250 AIG nodes on average. After CNFization by ABC's internal engine resulting QCNFs on average have 93 variables and 240 clauses.

For the circuit solving we used ABC to solve original problems, and RAReQS and Qesto for the corresponding CNF problems. Further we also used RAReQS for the negated 3QBF problems. The Bloqqer preprocessor was found to degrade the performance of CNF QBF solvers significantly on this benchmark set, therefore we do not include it in this set of experiments.

Table 2. Statistics for FPGA mapping benchmark set.

	RAReQS	Qesto	Bloqqer+RAReQS+3QBF	ABC
Solved	22	44	100	100
Time, s	96.9K	75.0K	63.5	64.3
Iterations	NA	6.46M	NA	1241

Solving statistics are shown in Table 2. As one can observe from Table 2 CNF 2QBF solvers required several orders of magnitude more iterations, and significantly larger solving time.

The cumulated number of iterations shown in Table 2 confirms that in 2QBF solving CNF representation is good for carrying out SAT queries, but cofactoring should be done on the circuit and not on the CNF level. On the other hand an alternative, equally efficient to circuit 2QBF solving, way is to use complemented 3QBF encoding presented in Sect. 4.

7.2 Certificate Derivation

We patched RAReQS solver to emit countermoves associated with the CEGAR QBF solving process. Proposed in Sect. 6 algorithm was implemented into a tool CegarSkolem. In order to test the unsat core optimization from Sect. 6 we used Haifa-HLMUC group MUS extractor [19]. Experimental setup consisted of QBFs from "QBFLIB" and "Applications" tracks taken from QBF Gallery 2014 [1]. We used DepQBF QBF solver [13] and ResQu [4] to compare CegarSkolem against existing Q-resolution based model computation in search-based QBF-solvers framework. An additional limitation of 1 Gb was imposed on Q-resolution proofs produced by DepQBF and moves information emitted by RAReQS.

We selected 29 and 137 true 3QBFs either solved by DepQBF or RAReQS, for experiments from "QBFLIB" and "Applications" tracks, respectively. The number of true 2QBF formulas was insufficient so we decided to focus only on 3QBFs instead. Table 3 shows the solving and certification time statistics (rightmost "#cert" and "#mincert" columns stand for CegarSkolem w/o and w/ optimization heuristics, respectively). Note that as certification requires additional effort, both DepQBF and RareQs were not able to solve some of the instances that they could solve w/o certification. As we could see in general RAReQS solved more instances, but DepQBF has much smaller runtime-per-instance. On contrary, even with optimizations, CegarSkolem constructed Skolem functions much quicker than ResQu, which is explained by large overhead in the size of Q-refutations produced by DepQBF, in comparison to (relatively small) number of existential counterexamples emitted by RAReQS.

Figure 2 compares certificates quality in terms of numbers of AIG nodes. X-axis in figures corresponds to certificates produced by ResQu, and Y-axis to those by CegarSkolem and CegarSkolemMin. We omit the details of

Table 3. Solving and certification statistics.

	DepQBF+ResQu				RAReQS+CegarSkolem					
	#solved	time, s	#cert	time, s	#solved	time, s	#cert	time, s	#mincert	time, s
QBFLIB [29]	15	424.1	14	943.2	19	2710.6	19	49.5	19	67.5
Appl [137]	94	457.9	86	5162.8	102	4351.6	97	533.8	97	589.0

the impact of various optimizations in CegarSkolemMin, but as one can see from Table 3 the computational overhead they introduce is small anyway. The certificate sizes, on the other hand, are reduced much in some cases. Another observation to make is that certificates for DepQBF and RAReQS are quite scattered across the figures. This means that for some benchmarks there exist simple Skolem-functions found by ResQu but not found by Cegar-Skolem and vice-versa.

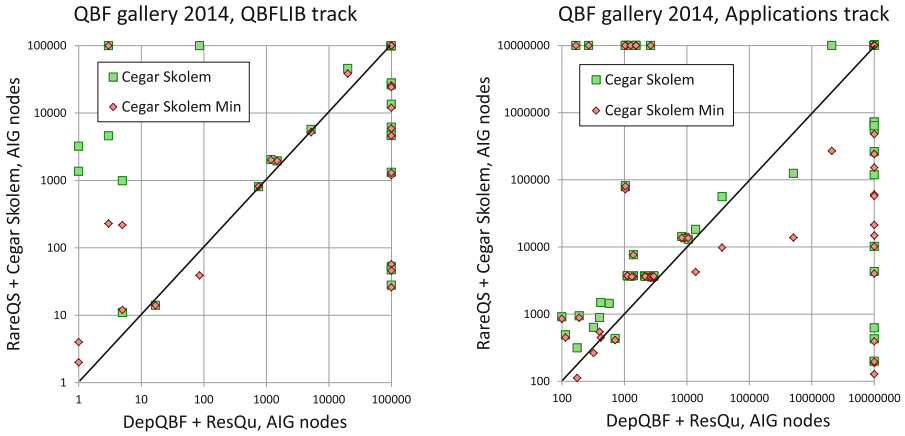

Fig. 5. Comparison of Skolem functions AIG sizes.

8 Conclusions and Future Work

In this work we overviewed different methods of solving and certifying 2QBF formulas. Based on our experience we introduced several improvements both to the CNF and circuit based solving techniques. We as well performed an extensive comparison to the state of the art algorithms. Experiments showed that the proposed solving improvements outperform existing 2QBF solvers, as well as the introduced approach for semantic certificate generation. In conclusion we summarize that circuit CEGAR-based 2QBF solvers generally scale better compare to the CNF based solvers. On the other hand, CNF preprocessing may be one of the levers, infeasible to the circuit solvers, that could lift the off-the-shelf CNF QBF solvers to a competitive level, given that circuit problem is properly encoded into CNF (e.g., with the negated 3QBF encoding).

Acknowledgments. This work was partly supported by NSF/NSA grant "Enhanced equivalence checking in cryptoanalytic applications" at University of California, Berkeley.

References

1. QBF Gallery 2014. http://qbf.satisfiability.org/gallery/
2. QBF solver evaluation portal. http://www.qbflib.org/qbfeval/
3. QDIMACS: Standard QBF input format. http://www.qbflib.org/qdimacs
4. Balabanov, V., Jiang, J.-H.R.: Unified QBF certification and its applications. Formal Meth. Syst. Des. **41**, 45–65 (2012)
5. Benedetti, M.: sKizzo: a suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
6. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 101–115. Springer, Heidelberg (2011)
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: an efficient QBF solver. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 201–213. Springer, Heidelberg (2004)
9. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. J. Artif. Intell. Res. (JAIR) **26**, 371–416 (2006)
10. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012)
11. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 230–244. Springer, Heidelberg (2011)
12. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Proceedings International Joint Conference on Artificial Intelligence (IJCAI), pp. 325–331 (2015)
13. Lonsing, F., Biere, A.: DepQBF: a dependency-aware QBF solver (system description). J. Satisfiability Boolean Model. Comput. **7**, 71–76 (2010)
14. Marques-Silva, J.P., Sakallah, K.A.: Boolean satisfiability in electronic design automation. In: Proceednigs Design Automation Conference (DAC), pp. 675–680 (2000)
15. Mishchenko, A., Brayton, R.K., Feng, W., Greene, J.W.: Technology mapping into general programmable cells. In: Proceedings International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 70–73 (2015)
16. Mneimneh, M., Sakallah, K.A.: Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 411–425. Springer, Heidelberg (2004)
17. Pigorsch, F., Scholl, C.: Exploiting structure in an AIG based QBF solver. In: Proceedings Design, Automation and Test in Europe (DATE), pp. 1596–1601 (2009)
18. Remshagen, A., Truemper, K.: An effective algorithm for the futile questioning problem. J. Autom. Reasoning **34**(1), 31–47 (2005)

19. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)
20. Tseitin, G.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic (1970)