Solving QBF with Counterexample Guided Refinement

Mikoláš Janota[1], William Klieber[3], Joao Marques-Silva[1,2], and Edmund Clarke[3]

[1] IST/INESC-ID, Lisbon, Portugal
[2] University College Dublin, Ireland
[3] Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. We propose two novel approaches for using Counterexample-Guided Abstraction Refinement (CEGAR) in Quantified Boolean Formula (QBF) solvers. The first approach develops a recursive algorithm whose search is driven by CEGAR (rather than by DPLL). The second approach employs CEGAR as an additional learning technique in an existing DPLL-based QBF solver. Experimental evaluation of the implemented prototypes shows that the CEGAR-driven solver outperforms existing solvers on a number of families in the QBF-LIB and that the DPLL solver benefits from the additional type of learning. Thus this article opens two promising avenues in QBF: CEGAR-driven solvers as an alternative to existing approaches and a novel type of learning in DPLL.

1 Introduction

Quantified Boolean formulas (QBFs) [8] naturally extend the SAT problem by enabling expressing PSPACE-complete problems, which can be found in a number of areas [13]. While nonrandom SAT solving has been dominated by the DPLL procedure, it has proven to be far from a silver bullet for QBF solving. Indeed, a number of solving techniques have been proposed for QBF [12,3,4,19,15], complemented by a variety of *preprocessing techniques* [7,14,21,5].

This paper extends the family of QBF solving techniques by employing the counterexample guided abstraction refinement (CEGAR) paradigm [10]. This is done in two different ways. The first approach develops a novel algorithm, named RAReQS, that gradually *expands* the given formula into a propositional one. In contrast to the existing expansion-based solvers [1,4,19], the use of CEGAR in RAReQS enables terminating before the formula is fully expanded and thus substantially mitigates the problems with memory blowup inherent to expansion-based solvers. The second approach employs CEGAR as an *additional learning technique* in an existing DPLL-based QBF solver. At the price of higher memory consumption, this learning technique enables more aggressive pruning of the search space than the existing techniques [28]. The experimental evaluation carried out demonstrates that CEGAR-based techniques are useful for a large number of families in the QBF-LIB [25].

* This work is partially supported by FCT grants ATTEST (CMU-PT/ELE/0009/-2009) and POLARIS (PTDC/EIA-CCO/123051/2010), by SFI grant BEACON (09/IN.1/I2618), and by Semiconductor Research Corporation contract 2005TJ1366.

2 Preliminaries

Quantified Boolean formulas (QBF) are assumed, unless noted otherwise, to be in *prenex* form $Q_1 z_1 \ldots Q_n z_n . \phi$ where $Q_i \in \{\forall, \exists\}$, z_i are distinct variables, and ϕ is a propositional formula using only the variables z_i and the constants 0 (false), 1 (true). The sequence of quantifiers in a QBF is called the *prefix* and the propositional formula the *matrix*. The prefix is divided into *quantifier blocks*, each of which is a subsequence $\forall x_1 \ldots \forall x_n$ or resp. $\exists x_1 \ldots \exists x_n$, which we denote by $\forall X$ or resp. $\exists X$, where $X = \{x_1, ..., x_n\}$.

Notation. We write $\bar{Q}$ for "$\forall$" (if Q is "$\exists$") or "$\exists$" (if Q is "$\forall$").

Whenever convenient, parts of a prefix are denoted as P with possible subscripts, e.g. $P_1 \forall X P_2 . \phi$ denotes a QBF with the matrix ϕ and a prefix that contains $\forall X$. If the quantifier of a block Y occurs within the scope of the quantifier of another block X, we say that variables in X are *upstream* of variables in Y and that variables in Y are *downstream* of variables in X.

Variable assignments are represented as sets of literals. In particular, an assignment τ to the set of variables X contains exactly one of x, $\neg x$ for each $x \in X$, with the meaning that if $x \in \tau$, the variable x has the value 1 in τ and if $\neg x \in \tau$, it has the value 0.

Notation. We write $\mathcal{B}^Y$ for the set of assignments to the variables Y.

For a Boolean formula ϕ and an assignment τ we write $\phi[\tau]$ for the substitution of τ in ϕ. In practice a substitution also performs basic simplifications, e.g. $(\neg x \vee y)[\{\neg x\}] = (\neg 0 \vee y) = 1$. We extend the notion of substitution to QBF so that it first removes the quantifiers of substituted variables and then substitutes all occurrences with their assigned values. E.g., if τ is an assignment to a block X, then $(P_1 Q X P_2 .\ \phi)\,[\tau]$ results in $P_1 P_2 .\ \phi[\tau]$.

A Boolean formula in *conjunctive normal form (CNF)* is a conjunction of *clauses*, where a clause is a disjunction of *literals*, and a literal is either a variable or its complement. Whenever convenient, a CNF formula is treated as a set of clauses. For a literal l, $\mathsf{var}(l)$ denotes the variable in l, i.e. $\mathsf{var}(\neg x) = \mathsf{var}(x) = x$.

The pseudocode throughout the paper uses the function $\mathtt{SAT}(\phi)$ to represent a call to a SAT solver on a propositional formula ϕ. The function returns a satisfying assignment for ϕ, if such exists, and returns NULL otherwise.

2.1 Game-Centric View

A QBF can be seen as a a *game* between the *universal player* and the *existential player*. During the game, the existential player assigns values to the existentially quantified variables and the universal player assigns values to the universally quantified ones. A player can assign a value to a variable only if all variables upstream of it already have a value. The existential player wins if the formula evaluates to 1 and the universal player wins if it evaluates to 0.

We note that the order in which values are given to variables in the same block is unimportant. Hence, by a *move* we mean an assignment to variables in a certain block. A concept useful throughout the paper are the *winning moves*.

Definition 1 (winning move). *Consider a (nonprenex) closed QBF $QX.\Phi$ and an assignment τ to X. Then τ is called a* winning move for $QX.\Phi$ *if $Q{=}\exists$ and $\Phi[\tau]$ is true or $Q{=}\forall$ and $\Phi[\tau]$ is false.*
Notation. *We write $\mathcal{M}(QX.\,\Phi)$ to denote the set of winning moves for $QX.\,\Phi$.*

Observation 1 *Let Φ be a QBF.*
A closed QBF $\exists X.\,\Phi$ is true iff there exists a winning move for $\exists X.\,\Phi$.
A closed QBF $\forall Y.\,\Phi$ is true iff there does not *exist a winning move for $\forall Y.\,\Phi$.*

3 Recursive CEGAR-based Algorithm

Previous work on QBF shows how CEGAR can be used to solve formulas with 2 levels of quantifiers [17]. Here we generalize this approach to an arbitrary number of quantifiers by recursion. The recursion follows the prefix of the given formula starting with the most upstream variables progressing towards more downstream variables. It tries to find a winning move (Definition 1) for variables in a certain block by making recursive calls to obtain winning moves for the downstream variables. The base case of the recursion, i.e., a QBF with one quantifier, is handled by a SAT solver.

The algorithm is presented as a recursive function returning a winning move for the given formula, if such move exists. Following the CEGAR paradigm, the function builds an abstraction which provides *candidates* for the winning move. This abstraction is gradually refined as the algorithm progresses. Refinement is realized by *strengthening* the abstraction, which means reducing the set of winning moves; strengthening is achieved by applying conjunction and disjunction.

Observation 2 *Let $\Phi_1, \ldots, \Phi_n$ be QBFs with free variables in X.*
$\mathcal{M}\left(\forall X.\ (\Phi_1 \vee \cdots \vee \Phi_n)\right) \subseteq \mathcal{M}\left(\forall X.\ \Phi_i\right)$, $i \in 1..n$.
$\mathcal{M}\left(\exists X.\ (\Phi_1 \wedge \cdots \wedge \Phi_n)\right) \subseteq \mathcal{M}\left(\exists X.\ \Phi_i\right)$, $i \in 1..n$.
$\mathcal{M}(\forall X \exists Y.\ \Phi) = \mathcal{M}(\forall X.\ \bigvee_{\mu \in \mathcal{B}^Y} \Phi[\mu])$
$\mathcal{M}(\exists X \forall Q Y.\ \Phi) = \mathcal{M}(\exists X.\ \bigwedge_{\mu \in \mathcal{B}^Y} \Phi[\mu])$

The second half of the above observation gives us a recipe how to eliminate quantifiers by *expanding* them into the corresponding propositional operator. One could thus eliminate quantifiers one by one and eventually call a SAT solver if only one quantifier is left. The clear disadvantage of this approach is that the formula grows rapidly and therefore performing the expansion is often unfeasible. This is where CEGAR comes in; the algorithm expands quantifiers *carefully*, based on counterexamples that show that the current expansion is too weak. In this spirit, we define abstraction as a partial expansion of the given formula.

Definition 2 (ω-abstraction). *Let ω be a subset of $\mathcal{B}^Y$.*
The ω-abstraction of a closed QBF $\forall X \exists Y.\ \Phi$ is the formula $\forall X.\ \bigvee_{\mu \in \omega} \Phi[\mu]$.
The ω-abstraction of a closed QBF $\exists X \forall Y.\ \Phi$ is the formula $\exists X.\ \bigwedge_{\mu \in \omega} \Phi[\mu]$.

Algorithm 1: Basic recursive CEGAR algorithm for QBF

```
1  Function Solve (QX. Φ)
   input  : QX. Φ is a closed QBF in prenex form with no adjacent
            blocks with the same quantifier
   output : a winning move for QX. Φ if there is one, NULL otherwise
2  begin
3      if Φ has no quantifiers then
4          return (Q = ∃) ? SAT(φ) : SAT(¬φ)
5      end
6      ω ← ∅
7      while true do
8          α ← (Q = ∃) ? ⋀_{μ∈ω} Φ[μ] : ⋁_{μ∈ω} Φ[μ]    // build abstraction
9          τ′ ← Solve(Prenex(QX. α))     // find a candidate solution
10         if τ′ = NULL then return NULL          // no winning move
11         τ ← {l | l ∈ τ′ ∧ var(l) ∈ X}          // filter a move for X
12         μ ← Solve(Φ[τ])                        // find a counterexample
13         if μ = NULL then return τ
14         ω ← ω ∪ {μ}                                          // refine
15     end
16 end
```

Observe that any winning move for $QX\bar{Q}Y.\ \Phi$ is also a winning move for its ω-abstraction (for arbitrary ω). The reverse, however, does not hold. Hence, following the CEGAR paradigm, we first find a winning move for the abstraction and then *verify* that it is also a winning move for the given formula. Verifying that a given assignment is a winning move entails solving another QBF.

Observation 3 *An assignment τ is a winning move for a closed $QX\bar{Q}Y.\ \Phi$ iff $\bar{Q}Y.\ \Phi[\tau]$ has no winning move.*

If a winning move for the abstraction is verified to be a winning move for the given formula, the move is returned. However, if this is not the case, the abstraction is strengthened. Observation 3 tells us that if an assignment τ is *not* a winning move for $QX\bar{Q}Y.\ \Phi$, then there *is* a winning move μ for the opposing quantifier $\bar{Q}$ for the QBF $\bar{Q}Y.\ \Phi[\tau]$. We say that this move μ is a *counterexample* to τ because it serves as a witness demonstrating that τ is not a winning move for $QX\bar{Q}Y.\ \Phi$. In accordance with the concept of counterexample *guided* abstraction refinement, if a counterexample μ is found, the current ω-abstraction is strengthened by adding μ to ω.

When we put these things together, we obtain Algorithm 1. The algorithm is given a closed QBF $QX.\ \Phi.$ and returns a winning move for $QX.\ \Phi$, if such exists, and returns NULL otherwise. It is required that $QX.\ \Phi$ is in prenex form where no two adjacent blocks have the same quantifiers (the blocks are maximal). The algorithm starts with $\omega = \emptyset$; this represents an abstraction that can be

won by any candidate. In each iteration of the CEGAR loop it first solves the abstraction (line 9) and then verifies whether the move winning the abstraction is also a winning move for the given problem (line 12). These operations are realized as recursive calls. If there is no winning move for the abstraction, then there is no winning move for the given problem and the function terminates. If there is no counterexample to the move winning the abstraction, then this move is also a winning move for the given problem and the function terminates. If there is a counterexample to the move winning the abstraction, the abstraction must be refined (line 14).

The precondition of the function that the input formula must be in prenex form with no adjacent blocks with the same quantifier poses some technical difficulty. When constructed directly according to its definition (Definition 2), the abstraction does not necessarily satisfy this condition.

Consider the case for $Q = \exists$ ($Q = \forall$ is analogous). The abstraction is of the form $\exists X.\ \bigwedge_{\mu \in \omega} \Phi[\mu]$. Prenexing the abstraction generates fresh variables for each of the conjuncts $\Phi[\mu]$, interleaves them into a single prefix, and merges adjacent blocks that start with the same quantifier. Since each $\Phi[\mu]$ starts with the existential quantifier (the substitution of μ eliminated the universal variables at the top), after prenexing, the abstraction's prefix starts with $\exists X X_1 \dots X_k$ where X_i are the fresh variables for the conjuncts $\Phi[\mu]$. For this reason if a winning move for the abstraction is computed, only the assignments to the variables X are considered (line 11).

Example 1. Consider the QBF $\exists vw.\Phi$, where $\Phi = \forall u \exists xy.\ (v \vee w \vee x) \wedge (\bar{v} \vee y) \wedge (\bar{w} \vee y) \wedge (u \vee \bar{x}) \wedge (\bar{u} \vee \bar{y})$, and the candidates $\{v, w\}$ and $\{\bar{v}, \bar{w}\}$, and corresponding counterexamples $\{u\}$ and $\{\bar{u}\}$. Refinement yields the abstraction $\exists vw.\ \Phi[\{u\}] \wedge \Phi[\{\bar{u}\}]$, with the prenex form $\exists vwxyx'y'.\ (v \vee w \vee x) \wedge (\bar{v} \vee y) \wedge (\bar{w} \vee y) \wedge (\bar{y}) \wedge (v \vee w \vee x') \wedge (\bar{v} \vee y') \wedge (\bar{w} \vee y') \wedge (\bar{x}')$ with no winning move and the algorithm terminates with the return value NULL.

3.1 Improving Recursive CEGAR-based Algorithm

Algorithm 1 clearly suffers from high memory consumption since in each iteration of the loop the abstraction is increased by the size of the input formula and the number of its variables is doubled (in the worst case). Recursive calls further amplify this unfavorable behavior. For the input formula $\exists X.\ \Phi$, performing n_1 iterations with the counterexamples $\mu_1^1, \dots, \mu_{n_1}^1$ yields the abstraction $\Omega = \exists X.\ \phi[\mu_1^1] \wedge \dots \wedge \phi[\mu_{n_1}^1]$. The algorithm subsequently invokes the recursive call `Solve`(Ω) on line 9. If within this recursive call the loop iterates n_2 times, its abstraction is of the form $\exists X.\ \Omega[\mu_1^2] \vee \dots \vee \Omega[\mu_{n_2}^2]$ with the size $O(n_1 \times n_2 \times |\phi|)$. In general, if the algorithm iterates n_i times at a recursion level i, the abstraction at level k is of the size $O(n_1 \times \dots \times n_k \times |\phi|)$.

To cope with this inefficiency, we exploit the form of the formulas that the algorithm handles. In the case of the existential quantifier, the abstraction is a conjunct, and it is a disjunct in the case of the universal quantifier. For the sake of uniformity, we bridge these two forms by introducing the notion of a *multi-game* where a player tries to find a move that wins multiple formulas simultaneously.

Definition 3 (multi-game). *A* multi-game *is denoted by* $QX.\{\Phi_1, \ldots, \Phi_n\}$ *where each* Φ_i *is a prenex QBF starting with* $\bar{Q}$ *or has no quantifiers. The free variables of each* Φ_i *must be in* X *and all* Φ_i *have the same number of quantifier blocks. We refer to the formulas* Φ_i *as* subgames *and* QX *as the* top-level prefix.

A winning move *for a multi-game is an assignment to the variables* X *such that it is a winning move for each of the formulas* $QX.\ \Phi_i$.

Observe that the set of winning moves of a multi-game $QX.\{\Phi_1, \ldots, \Phi_n\}$ is the same as the set of winning moves of the QBF $\forall X.(\Phi_1 \vee \cdots \vee \Phi_n)$ for $Q = \forall$ and it is the same as $\exists X.(\Phi_1 \wedge \cdots \wedge \Phi_n)$ for $Q = \exists$. And, any QBF $QX.\ \Phi$ corresponds to a multi-game with a single subgame $QX.\{\Phi\}$

To solve multi-games we use Algorithm 2. The algorithm is given a multi-game to solve and the abstraction is again a multi-game. To determine whether the candidate τ is a winning move, it tests whether it is a winning move for the subgames in turn. If it finds a subgame Φ_i s.t. $\Phi_i[\tau]$ is won by the opponent $\bar{Q}$ by a move μ, then $\Phi_i[\mu]$ is used to strengthen the abstraction.

Since an abstraction is a multi-game, it seems natural to add $\Phi_i[\mu]$ to the set of its subgames. This, however, cannot be done right away because the formula is not in the right form. In particular, all the subgames must start with the opposite quantifier as the top-level prefix. Hence, if Φ_i is of the form $\bar{Q}YQX_1.\ \Psi_i$ and $\mu \in \mathcal{B}^Y$, then $\Phi_i[\mu] = QX_1.\ \Psi_i[\mu]$. To bring the formula into the right form, we introduce fresh variables for the variables X_1 and move them into the top-level prefix. More precisely, the function $\texttt{Refine}(\alpha, \Phi_l, \mu_l)$ is defined as follows (observe that the subgames remain in prenex form).

$\texttt{Refine}\big(QX.\{\Psi_1, \ldots, \Psi_n\},\ \bar{Q}YQX_1.\Psi,\ \mu\big) \ :=\ QXX_1'.\{\Psi_1, \ldots, \Psi_n, \Psi'[\mu]\}$
where X_1' *are fresh duplicates of the variables* X_1 *and* Ψ' *is* Ψ *with* X_1 *replaced by* X_1'

$\texttt{Refine}\big(QX.\{\Psi_1, \ldots, \Psi_n\},\ \bar{Q}Y.\psi,\ \mu\big) \ :=\ QX.\{\Psi_1, \ldots, \Psi_n, \psi[\mu]\}$
where ψ *is a propositional formula (where no duplicates are needed)*

Similarly to Algorithm 1, after the refinement, the abstraction's top-level prefix contains additional variables besides the variables X. Hence, values for these variables are filtered out if a winning move for the abstraction is found.

3.2 Properties of the Algorithms

In CEGAR loop of Algorithm 1 no candidate or counterexample repeats. Intuitively, this is because once a counterexample μ is found, the abstraction is strengthened so that in the future winning moves for the abstraction cannot be beaten by the move μ. Consequently, the loop is terminating and for a formula $QX\bar{Q}Y.\Phi$ the number of its iterations is bounded by the number of possible assignments to the variables X and Y, i.e. $\mathsf{min}(2^{|X|}, 2^{|Y|})$. In the worst case, in each iteration the abstraction grows by the size of Φ. For a multi-game $QX.\ \{\Phi_1, \ldots, \Phi_n\}$ in the CEGAR loop of Algorithm 2 no candidates repeat but counterexamples may. However, for a given $i \in 1..n$, a counterexample μ_i does not repeat. More precisely there are no two distinct iterations of the loop with the corresponding candidates and counterexamples $\tau_1, \mu_1, \tau_2, \mu_2$, such that

Algorithm 2: Recursive CEGAR algorithm for multi-games

```
1  Function RAReQS (QX.{Φ_1,...,Φ_n})
2  output: a winning move for QX.{Φ_1,...,Φ_n} if there is one; NULL otherwise
3  begin
4      if Φ_i have no quantifiers then
5          return Q = ∃ ? SAT(⋀_i Φ_i) : SAT(¬(⋁_i Φ))
6      α ← QX. {}
7      while true do
8          τ' ← RAReQS(α)                          // find a candidate solution
9          if τ' = NULL then return NULL
10         τ ← {l | l ∈ τ' ∧ var(l) ∈ X}           // filter a move for X
11         for i ← 1 to n do μ_i ← RAReQS(Φ_i[τ]) // find a counterexample
12         if μ_i = NULL for all i ∈ {1..n} then return τ
13         let l ∈ {1..n} be s.t. μ_l ≠ NULL
14         α ← Refine(α, Φ_l, μ_l)                 // refine
15     end
16 end
```

$\mu_1 = \mu_2$ and μ_1 is a winning move for both $\Phi_i[\tau_1]$ and $\Phi_i[\tau_2]$ for some i. This demonstrates termination with the upper bound for the number of iterations as $\mathsf{min}(2^{|X|}, n \times 2^{|Y|})$. In the worst case, in each iteration the abstraction grows by the maximum of the sizes of the subgames $\Phi_1, \ldots, \Phi_n$. Soundness and completeness of the algorithms 1 and 2 is a direct consequence of Observation 2.

3.3 Implementation Details

We have implemented a prototype of RAReQS in `C++`, supporting the QDIMACS format, with the underlying SAT solver `minisat 2.2` [11].

The implementation has several distinctive features. In Algorithm 2, an abstraction computed within a sub-call is forgotten once the call returns. This may lead to repetition of work and hence the solver supports maintaining these abstractions and strengthening them gradually, similarly to the way SAT solvers provide *incremental* interface. This incremental approach, however, tends to lead to unwieldy memory consumption and therefore, it is used only when the given multigame's subgames have 2 or fewer quantification blocks.

If an assignment τ is a candidate for a winning move that turns out *not* to be a winning move, the refinement guarantees that τ is not a solution to the abstraction in the future iterations of the CEGAR loop. This knowledge enables us to make the subcall for solving the abstraction more efficient by explicitly disabling τ as a winning move for the abstraction. We refer to this technique as *blocking* and it is similar to the refinement used in certain SMT solvers [24,2].

Throughout its course, the algorithm may produce a large number of new formulas, either by substitution or refinement. Since these formulas tend to be

Algorithm 3: DPLL Algorithm with CEGAR Learning

```
1.  global π_cur = ∅;
2.  function dpll_solve(Φ_in) {
3.    while (true) {
4.      while (we don't know who has a winning strategy under π_cur) {
5.        decide_lit(); propagate();
7.      }
8.      Φ_in := dpll_learn(Φ_in);
9.      if (we learned who has a winning strategy under ∅) return;
10.     if (last decision literal is owned by winner) {
11.       Φ_in := cegar_learn(Φ_in);
12.     }
13.     backtrack();
14.     propagate(); // Learned information will force a literal.
15.   }
16. }
```

simpler than the given one, they can be further simplified by standard QBF *preprocessing* techniques. The implementation uses unit propagation and *monotone* (pure) literal rule [9]. These simplifications introduce the complication that in a multi-game $QX.\{\Phi_1, \ldots, \Phi_n\}$ the individual subgames might not necessarily have the same number of quantifier levels. In such case, all games with no quantifiers are immediately put into the abstraction before to loop starts.

4 CEGAR as a learning technique in DPLL

The previous section shows that CEGAR can give rise to a complete and sound algorithm for QBF. In this section we show that CEGAR enables us to extend existing DPLL solvers with an additional learning technique. To illustrate the basic idea consider the QBF $\forall X.\,(\exists Y.\ \phi)$ and a situation when the solver assigned values to variables in X and Y such that ϕ is satisfied, i.e., the existential player won. This assignment has two disjoint parts, π_{cand} and π_{cex}, which are assignments to X and Y, respectively. Conceptually, π_{cand} corresponds the candidate assignment in RAReQS and π_{cex} to its counterexample. In this case, the CEGAR-based learning will correspond to disjoining the formula $\phi[\pi_{\text{cex}}]$ onto ϕ, resulting in $\forall X.\,(\exists Y.\ \phi) \vee \phi[\pi_{\text{cex}}]$, so that π_{cand} is avoided in the future.

The CEGAR learning in DPLL is most naturally described in the context of a non-prenex, non-clausal solver such as GhostQ [18]. Given an assignment π, such a solver will tell us that either (1) the existential player has a winning stategy under π (i.e., $\Phi_{\text{in}}[\pi]$ is true), (2) the universal player has a winning stategy under π (i.e., $\Phi_{\text{in}}[\pi]$ is false), or (3) it is not yet known which player has a winning strategy under π.

1. Let X_c be the quantifier block of the last decision literal.
   Let Q_c and Φ_c be such that $(Q_c X_c.\Phi_c)$ is a subformula of Φ_{in}.
2. Let π_c be a complete assignment for X_c created by extending the solver's current assignment with arbitrary values for the unassigned variables in X_c and removing variables in blocks other than X_c. This assignment π_c corresponds to the *counterexample* in the recursive CEGAR approach.
3. We modify Φ_{in} by:
   - substituting $(\exists X_c.\Phi_c)$ with $(\exists X_c.\Phi_c) \vee \Phi_c[\pi_c]$, if $Q_c =$ "$\exists$", or
   - substituting $(\forall X_c.\Phi_c)$ with $(\forall X_c.\Phi_c) \wedge \Phi_c[\pi_c]$, if $Q_c =$ "$\forall$".
4. All variables that are bound by a quantifier inside $\Phi_c[\pi_c]$ are renamed to preserve uniqueness of variable names.

Fig. 1. CEGAR Learning in DPLL

We modify such a solver by inserting a call to a new CEGAR-learning procedure after performing standard DPLL learning, as shown in Algorithm 3. We write "Φ_{in}" to denote the current input formula, i.e., the input formula enhanced with what the solver has learned up to now. Both standard DPLL learning and CEGAR learning are performed by modifying Φ_{in}. As shown in Algorithm 3, CEGAR learning is performed only if the last decision literal is owned by the winner. (The case where the last decision literal is owned by the losing player corresponds to the conflicts that take place *within* the underlying SAT solver in RAReQS.) The CEGAR-learning procedure is shown in Figure 1. Step 3 is justified by Observation 5 below, which in turn is justified by Observation 4.

Observation 4 Consider an arbitrary QBF $(Q_c X_c.\, \Phi_c)$, possibly containing free variables, but where each bound variable is bound by at most one quantifier. Then it follows immediately from definition of quantification that:

$$\exists X_c.\Phi_c \;=\; \bigvee_{\pi \in \mathcal{B}^{X_c}} \Phi_c[\pi] \qquad \text{and} \qquad \forall X_c.\Phi_c \;=\; \bigwedge_{\pi \in \mathcal{B}^{X_c}} \Phi_c[\pi]$$

(Recall that "$\mathcal{B}^{X_c}$" denotes the set of all assignments to X_c.)

Observation 5 Since conjunction and disjunction are idempotent,

$$\exists X_c.\Phi_c = (\exists X_c.\Phi_c) \vee \Phi_c[\pi_c], \text{ where } \pi_c \in \mathcal{B}^{X_c}$$
$$\forall X_c.\Phi_c = (\forall X_c.\Phi_c) \wedge \Phi_c[\pi_c], \text{ where } \pi_c \in \mathcal{B}^{X_c}$$

4.1 Implementation Details

We have implemented a limited version of CEGAR learning in the solver GhostQ [18]. Our implementation uses a modified version of step 3 of Figure 1. We substitute π_c into the original version of the input formula Φ_{in}, not the current version of Φ_{in}. Although substituting into the original formula instead of the

current formula potentially reduces the effectiveness of CEGAR learning (since we can't learn a refinement of a refinement), it reduces the memory consumed per refinement. Unit propagation and the Pure Literal Rule are applied to simplify the result of the substitution, among other optimizations.

Step 2 of Figure 1 extends the counterexample π_c to a complete assignment to the quantifier block X_c. This allows completely eliminating a quantifier block, which may cause two quantifier blocks of the same quantification type to become adjacent to each other. If so, the two adjacent blocks are merged together, providing greater freedom in selecting variable order.

5 Experimental Results

Our objective was to analyze the effect of CEGAR on the different families of available benchmarks. Due to do the large number of families in QBF-LIB [25], we have targeted families from *formal verification* and *planning* as two prominent applications of QBF. Several large and hard families were sampled with 150 files (`terminator`, `tipfixpoint`, `Strategic Companies`); the area of planning contains four classes for robot planning, each counting 1000 instances with similar characteristics and thus only one of these classes was selected (`Robots2D`). The solvers QuBE7.2, Quantor, and Nenofex were chosen for comparison. QuBE7.2 is a state-of-the-art DPLL-based solver; Quantor and Nenofex are expansion-based solvers (c.f. Section 6). The experimental results were obtained on an Intel Xeon 5160 3GHz, with 4GB of memory. The time limit was set to 800 seconds and the memory limit to 2GB.

All the instances were preprocessed by the preprocessor bloqqer [5] and instances solved by the preprocessor alone were excluded from further analysis. An exception was made for the family `Debug` where preprocessing turned out to be infeasible and the family was considered in its unpreprocessed form.

Unlike the other solvers, GhostQ's input format is not clause-based (QDIMACS) but it is circuit-based. To enable running GhostQ on the targeted instances, the solver was prepended with a reverse-engineering front-end. Since this front-end cannot handle bloqqer's output, GhostQ was run directly on the instances without preprocessing. The other solvers were run on the preprocessed instances (further preprocessing was disabled for QuBE7.2).

The relation between solving times and instances is presented by a cactus plot in Figure 2; number of solved instances per family are shown in Table 2; a comparison of RAReQS with other solvers is presented in Table 1. More detailed information can be found at `http://sat.inesc-id.pt/~mikolas/sat12`.

On the considered benchmarks, RAReQS solved the most instances, approximately 33% more than the second solver QuBE7.2. RAReQS also turned out to be the best solver for most of the types of the considered instances. Table 1 further shows that for each of the other solvers, there is only a small portion of instances that the other solver can solve and RAReQS cannot. Out of the 801 instances when the solver was aborted, only 50 ran out of of memory.

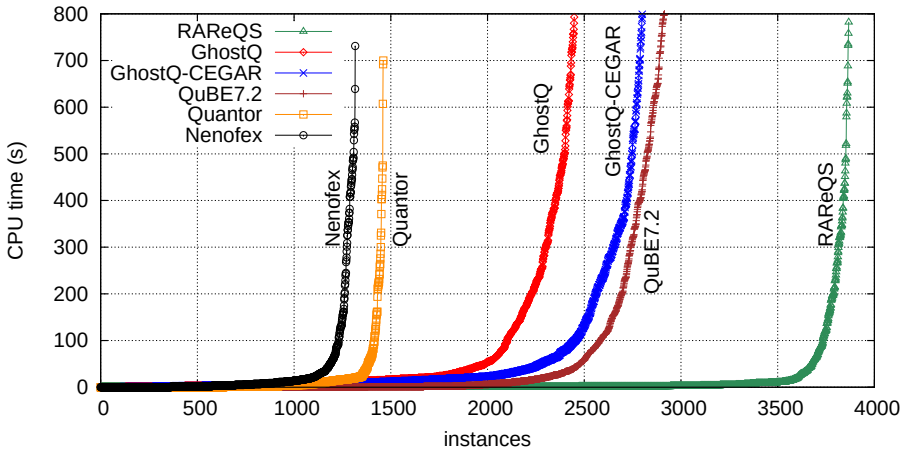

Fig. 2. Cactus plot of the overall results

	GhostQ	GhostQ-CEGAR	QuBE7.2	Quantor	Nenofex
Only RAReQS	1661	1336	998	2436	2564
Only competitor	242	269	46	30	13

Table 1. Number of instances solved by RAReQS but not by a competing solver, and *vice versa*

In several families the addition of CEGAR learning to GhostQ worsened its performance. With the exception of `Robots2D`, however, the performance was worse only slightly. Overall, GhostQ benefited from the additional CEGAR learning and in particular for certain families. A family worth noting is `irqlkeapclte`, where no instances were solved by any of the solvers except for GhostQ-CEGAR.

The usefulness of CEGAR was in particular demonstrated by the families `incrementer-encoder`, `conformant-planning`, `trafficlight-controller`, `Sorting-networks`, and `BMC` where RAReQS solved significantly more instances than the existing solvers, and GhostQ-CEGAR improved significantly over GhostQ. Most notably, for `incrementer-encoder (484)` and `RobotsD2 (700)` only one instance was not solved by RAReQS, and for `blackbox-01X-QBF (320)` and `trafficlight-controller (1459)` RAReQS solved *all* instances.

6 Related Work

CEGAR has proven useful in number of areas, most notably in model checking [10] and SMT solving [24,2]; more recently it has been applied to handle quantification in SMT [27,23]. Special cases of QBF, with limited number of quantifiers, have been targeted by CEGAR: computing vertex eccentricity [22], nonmonotonic reasoning [6,16], two-level quantification [17].

Family	Lev.	RAReQS	GhostQ	GhostQ-Cegar	QuBE7.2	Quantor	Nenofex
trafficlight-ctlr (1459)	1–287	**1459**	806	1001	1092	955	863
RobotsD2 (700)	2–2	**699**	350	271	630	0	30
incrementer-encoder (484)	3–119	**483**	285	477	284	51	27
blackbox-01X-QBF (320)	2–21	**320**	138	126	224	3	4
Strat. Comp. (samp.) (150)	1–2	**107**	12	12	**107**	18	12
BMC (85)	1–3	**73**	26	48	37	65	64
Sorting-networks (84)	1–3	**72**	24	32	45	38	38
blackbox-design (27)	5–9	**27**	**27**	**27**	18	0	0
conformant-planning (23)	1–3	**17**	7	16	5	13	12
Adder (28)	3–7	**11**	2	2	4	5	9
Lin. Bitvec. Rank. Fun. (60)	3–3	**9**	0	0	0	0	0
Ling (8)	1–3	**8**	6	**8**	**8**	**8**	**8**
Blocks (7)	3–3	**7**	6	**7**	5	**7**	**7**
fpu (6)	1–3	**6**	0	0	**6**	**6**	**6**
RankingFunctions (4)	2–2	**3**	0	0	**3**	0	0
Logn (2)	3–3	**2**	**2**	**2**	**2**	**2**	**2**
Mneimneh-Sakallah (163)	1–3	110	**148**	141	89	3	22
tipfixpoint-sample (150)	1–3	26	**128**	127	22	5	6
terminator-sample (150)	2–2	98	**109**	103	9	25	0
tipdiam (121)	1–3	55	**99**	93	54	21	14
Scholl-Becker (55)	1–29	37	**43**	40	29	32	27
evader-pursuer (15)	5–19	10	**11**	8	**11**	2	2
uclid (3)	4–6	0	**2**	**2**	0	0	0
toilet-all (136)	1–1	134	133	131	131	**135**	133
Counter (58)	1–125	30	14	11	20	**33**	15
Debug (38)	3–5	3	0	0	0	**24**	6
circuits (63)	1–3	8	4	5	5	**9**	8
Gent-Rowley (205)	7–81	52	67	67	**70**	2	0
jmc-quant (+squaring) (20)	3–9	2	0	0	**6**	0	2
irqlkeapclte (45)	2–2	0	0	**44**	0	0	0
total (4669)		**3868**	2449	2801	2916	1462	1317

Table 2. Number of instances solved within 800 seconds by each solver. "Lev" indicates the number of quantifier blocks (min–max) in the family of instances, post-bloqqer.

A SAT solver was used in [26] to guide DPLL search of a QBF solver and to cut out unsatisfiable branches. A notion of abstraction was also used in QBF preprocessing [21]. This notion, however, differs from the one used in RAReQS as it means treating universally quantified variables as existentially quantified.

An important feature of RAReQS is the expansion of the given QBF into a propositional formula, which is then solved by a SAT solver. This technique is used for preprocessing [7] but also several existing solvers tackle QBF solving in this way, most notably QUBOS [1], Quantor [4], and Nenofex [19]. Just as RAReQS uses multi-games, these solvers employ some various techniques to mitigate the blowup of the expansion (besides preprocessing). QUBOS uses *miniscoping*, Quantor *tree-like prefixes*, and Nenofex uses *negation normal form*. In these aspects, the solvers share similarities with RAReQS.

The way the expansion is carried out is significantly different. While the other solvers start the expansion from the innermost variables, RAReQS starts from the outermost variables. The main difference, however, lies in the *careful*

expansion in RAReQS. In the aforementioned solvers, once a variable is scheduled to be expanded, both of its values are considered in the expansion. In contrast, in RAReQS only a particular assignment to a block of variables chosen in the expansion and the expansion is checked whether it is sufficient or not. This is an important factor for both time and space complexity. For large formulas, the traditional expansion-based solvers are bound to generate unwieldy formulas but the use of abstraction in RAReQS enables the solver to stop before this expansion is reached. This leads to generating easier formulas for the underlying SAT solver and dramatically mitigates the problems with memory blowup.

7 Conclusions and Future Work

Applying the CEGAR paradigm, this paper develops two novel techniques for QBF solving. The first technique is a CEGAR-driven solver RAReQS and the second an additional learning technique for DPLL solvers.

In its workings, RAReQS is close to expansion-based solvers (e.g. Quantor, Nenofex) but with the important difference that the expansion is done step-by-step, driven by counterexamples. Thus, the solver builds an abstraction of the given formula by constructing a partial expansion. The downside of this approach may be that if in the end a *full* expansion is needed, then RAReQS performs the same expansion as a traditional expansion-based solver but with the overhead of intermediate tests for whether or not the expansion is already sufficient.

However, the approach has important advantages. Whenever there is no winning move for the partial expansion, then there is no winning move for the given formula. This enables RAReQS to quickly stop for formulas with no winning moves. For formulas for which there *is* a winning move, RAReQS only needs to build a strong-enough partial expansion whose winning moves are also likely to be winning moves for the given formula. The experimental results demonstrate the ability of RAReQS to avoid the inherent memory blowup of expansion solvers, and, that careful expansion outperforms a traditional DPLL-based approach on a large number of practical instances.

We have shown that abstraction-refinement as used in RAReQS is also applicable within DPLL solvers as an additional learning mechanism. This provides a more powerful learning technique than standard clause/cube learning, although it requires more memory. Experimental evaluation indicates that this type of learning is indeed useful for DPLL-based solvers.

In the future we plan to further develop our DPLL solver so that it supports the full range of CEGAR learning exploited by RAReQS and to investigate how to fine-tune this learning in order to mitigate the speed penalty for the cases where the learning provides little information over the traditional learning. This can not only be done by better engineering of the solver but also devising schemata that disable the learning once deemed too costly. In RAReQS we plan to investigate how to integrate techniques used in other solvers. In particular, more aggressive preprocessing as used in Quantor and techniques for finding commonalities in formulas used in Nenofex and dependency detection [20].

References

1. Ayari, A., Basin, D.A.: QUBOS: Deciding quantified Boolean logic using propositional satisfiability solvers. In: FMCAD (2002)
2. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: CAV (2002)
3. Benedetti, M.: Evaluating QBFs via symbolic Skolemization. In: LPAR (2004)
4. Biere, A.: Resolve and expand. In: SAT (2004)
5. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: CADE (2011)
6. Browning, B., Remshagen, A.: A SAT-based solver for Q-ALL SAT. In: ACM Southeast Regional Conference (2006)
7. Bubeck, U., Büning, H.K.: Bounded universal expansion for preprocessing QBF. In: SAT (2007)
8. Büning, H.K., Bubeck, U.: Theory of quantified boolean formulas. In: Handbook of Satisfiability. IOS Press (2009)
9. Cadoli, M., Giovanardi, A., Schaerf, M.: An algorithm to evaluate Quantified Boolean Formulae. In: National Conference on Artificial Intelligence (1998)
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5) (2003)
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT (2003)
12. Giunchiglia, E., Marin, P., Narizzano, M.: QuBE 7.0 system description. Journal on Satisfiability, Boolean Modeling and Computation 7 (2010)
13. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified boolean formulas. In: Handbook of Satisfiability. IOS Press (2009)
14. Giunchiglia, E., Marin, P., Narizzano, M.: sQueezeBF: An effective preprocessor for QBFs based on equivalence reasoning. In: SAT (2010)
15. Goultiaeva, A., Bacchus, F.: Exploiting QBF duality on a circuit representation. In: AAAI (2010)
16. Janota, M., Grigore, R., Marques-Silva, J.: Counterexample guided abstraction refinement algorithm for propositional circumscription. In: JELIA (Sep 2010)
17. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: SAT (2011)
18. Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: SAT (2010)
19. Lonsing, F., Biere, A.: Nenofex: Expanding NNF for QBF solving. In: SAT (2008)
20. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. JSAT (2010)
21. Lonsing, F., Biere, A.: Failed literal detection for QBF. In: SAT (2011)
22. Mneimneh, M.N., Sakallah, K.A.: Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In: SAT (2003)
23. Monniaux, D.: Quantifier elimination by lazy model enumeration. In: CAV (2010)
24. de Moura, L.M., Rue, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: CADE (2002)
25. The Quantified Boolean Formulas satisfiability library, http://www.qbflib.org/
26. Samulowitz, H., Bacchus, F.: Using SAT in QBF. In: CP (2005)
27. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. In: FMCAD (2010)
28. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: ICCAD (2002)