Solving QBF with Free Variables

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

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

Abstract. An *open* quantified boolean formula (QBF) is a QBF that contains free (unquantified) variables. A solution to such a QBF is a quantifier-free formula that is logically equivalent to the given QBF. Although most recent QBF research has focused on closed QBF, there are a number of interesting applications that require one to consider formulas with free variables. This article shows how clause/cube learning for DPLL-based closed-QBF solvers can be extended to solve QBFs with free variables. We do this by introducing *sequents* that generalize clauses and cubes and allow learning facts of the form "under a certain class of assignments, the input formula is logically equivalent to a certain quantifier-free formula".

1 Introduction

In recent years, significant effort has been invested in developing efficient solvers for Quantified Boolean Formulas (QBFs). So far this effort has been almost exclusively directed at solving closed formulas — formulas where each variable is either existentially or universally quantified. However, in a number of interesting applications (such as symbolic model checking and automatic synthesis of a boolean reactive system from a formal specification), one needs to consider *open* formulas, i.e., formulas with free (unquantified) variables. A solution to such a QBF is a formula equivalent to the given one but containing no quantifiers and using only those variables that appear free in the given formula. For example, a solution to the open QBF formula $\exists x.\ (x \wedge y) \vee z$ is the formula $y \vee z$.

This article shows how DPLL-based closed-QBF solvers can be extended to solve QBFs with free variables. In [14], it was shown how clause/cube learning for DPLL-based QBF solvers can be reformulated in terms of *sequents* and extended to non-CNF, non-prenex formulas. This technique uses *ghost variables* to handle non-CNF formulas in a manner that is symmetric between the existential and universal quantifiers. We show that this sequent-based technique can be naturally extended to handle QBFs with free variables.

A naïve way to recursively solve an open QBF Φ is shown in Figure 1. Roughly, we Shannon-expand on the free variables until we're left with only closed-QBF problems, which are then handed to a closed-QBF solver. As an example, consider the formula $(\exists x.\ x \wedge y)$, with one free variable, y. Substituting y with true in Φ yields $(\exists x.\ x)$; this formula is given to a closed-QBF solver,

```
function solve($\Phi$) {
    if ($\Phi$ has no free variables) {return closed_qbf_solve($\Phi$);}
    x := (a free variable in $\Phi$);
    return ite(x, solve($\Phi$ with x substituted with True),
                  solve($\Phi$ with x substituted with False));
}
```

Fig. 1. Naive algorithm. The notation "`ite(x,` ϕ_1`,` ϕ_2`)`" denotes a formula with an *if-then-else* construct that is logically equivalent to $(x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$.

which yields true. Substituting y with false in Φ immediately yields false. So, our final answer is the formula (y ? true : false), which simplifies to y. In general, if the free variables are always branched on in the same order, then the algorithm effectively builds an ordered binary decision diagram (OBDD) [7], assuming that the `ite` function is memoized and performs appropriate simplification.

The above-described naïve algorithm suffers from many inefficiencies. In terms of branching behavior, it is similar to the DPLL algorithm, but it lacks non-chronological bracktracking and an equivalent of clause learning. The main contribution of this paper is to show how an existing closed-QBF algorithm can be modified to directly handle formulas with free variables by extending the existing techniques for non-chronological backtracking and clause/cube/sequent learning.

2 Preliminaries

Grammar. We consider prenex formulas of the form $Q_1X_1...Q_nX_n.\,\phi$, where $Q_i \in \{\exists, \forall\}$ and ϕ is quantifier-free and represented as a DAG. The logical connectives allowed in ϕ are conjunction, disjunction, and negation. We say that $Q_1X_1...Q_nX_n$ is the **quantifier prefix** and that ϕ is the **matrix**.

Assignments. Let π be a partial assignment of boolean values to variables. For convenience, we identify π with the set of literals made true by π. For example, we identify the assignment $\{(e_1, \mathsf{true}), (u_2, \mathsf{false})\}$ with the set $\{e_1, \neg u_2\}$. We write "vars(π)" to denote the set of variables assigned by π.

Quantifier Order. In a formula such as $\forall x.\exists y.\,\phi$, where the quantifier of y occurs inside the scope of the quantifier of x, and the quantifier type of x is different from the quantifier type of y, we say that y is **downstream** of x. Likewise, we say that x is **upstream** of y. All quantified variables in a formula are considered downstream of all free variables in the formula. In the context of an assignment π, we say that a variable is an **outermost** unassigned variable iff it is not downstream of any variables unassigned by π.

QBF as a Game. A closed QBF formula Φ can be viewed as a game between an existential player (Player $\exists$) and a universal player (Player $\forall$):

- Existentially quantified variables are *owned* by Player $\exists$.
- Universally quantified variables are *owned* by Player $\forall$.
- Players assign variables in quantification order (starting with outermost).
- The *goal* of Player $\exists$ is to make Φ be true.
- The *goal* of Player $\forall$ is to make Φ be false.
- A player *owns* a literal ℓ if the player owns $var(\ell)$.

If both players make the best moves possible, then the existential player will win iff the formula is true, and the universal player will win if the formula is false.

Substitution. Given a partial assignment π, we define "$\Phi|\pi$" to be the result of the following: For every assigned variable x, we replace all occurrences of x in Φ with the assigned value of x (and delete the quantifier of x, if any).

Gate variables. We label each conjunction and disjunction with a *gate variable*. If a formula ϕ is labelled by a gate variable g, then $\neg\phi$ is labelled by $\neg g$. The variables originally in the formulas are called "input variables", in distinction to gate variables.

2.1 Tseitin Transformation's Undesired Effects in QBF

The Tseitin transformation [20] is the usual way of converting a formula into CNF. In the Tseitin transformation, all the gate variables (i.e., Tseitin variables) are existentially quantified in the innermost quantification block and clauses are added to equate each gate variable with the subformula that it represents. For example, consider the following formula:

$$\Phi_{\text{in}} \;:=\; \exists e.\, \forall u.\; \underbrace{(e \land u)}_{g_1} \lor \underbrace{(\neg e \land \neg u)}_{g_2}$$

This formula is converted to:

$$\Phi'_{\text{in}} \;=\; \exists e.\, \forall u.\, \exists \boldsymbol{g}.\; (g_1 \lor g_2) \land (g_1 \Leftrightarrow (e \land u)) \land (g_2 \Leftrightarrow (\neg e \land \neg u)) \tag{1}$$

The biconditionals defining the gate variables are converted to clauses as follows:

$$(g_1 \Leftrightarrow (e \land u)) \;=\; (\neg e \lor \neg u \lor g_1) \;\land\; (\neg g_1 \lor e) \land (\neg g_1 \lor u)$$

Note that the Tseitin transformation is asymmetric between the existential and universal players: In the resulting CNF formula, the gate variables are existentially quantified, so the existential player (but not the universal player) loses if a gate variable is assigned inconsistently with the subformula that it represents. For example, in Equation 1, if $e|\pi = \mathsf{false}$ and $g_1|\pi = \mathsf{true}$, then the existential player loses $\Phi'_{\text{in}}|\pi$. This asymmetry can be harmful to QBF solvers. For example, consider the QBF

$$\forall \boldsymbol{x}.\, \exists y.\; y \lor \underbrace{\psi(\boldsymbol{x})}_{g_1} \tag{2}$$

This formula is trivially true. A winning move for the existential player is to make y be true, which immediately makes the matrix of the formula true, regardless of ψ. Under the Tseitin transformation, Equation 2 becomes:

$$\forall \boldsymbol{x}.\, \exists y.\, \exists \boldsymbol{g}.\ (y \vee g_1) \wedge (\text{clauses equating gate variables})$$

Setting y to be true no longer immediately makes the matrix true. Instead, for each assignment of universal variables $\boldsymbol{x}$, the QBF solver must actually find a satisfying assignment to the gate variables. This makes it much harder to detect when the existential player has won. Experimental results [1,22] indicate that purely CNF-based QBF solvers would, in the worst case, require time exponential in the size of ψ to solve the CNF formula, even though the original problem (before translation to CNF) is trivial.

3 Ghost Variables and Sequents

We employ *ghost variables* to provide a modification of the Tseitin transformation that is symmetric between the two players. The idea of using a symmetric transformation was first explored in [22], which performed the Tseitin transformation twice: once on the input formula, and once on its negation. Similar ideas have been used to handle non-prenex formulas in [14] and to handle "don't care" propagation in [12].

For each gate variable g, we introduce two *ghost variables*: an existentially quantified variable $g^{\exists}$ and a universally quantified variable $g^{\forall}$. We say that $g^{\exists}$ and $g^{\forall}$ *represent* the formula labeled by g. Ghost variables are considered to be downstream of all input variables.

We now introduce a semantics with ghost variables for the game formulation of QBF. As in the Tseitin transformation, the existential player should lose if an existential ghost variable $g^{\exists}$ is assigned a different value than the subformula that it represents. Additionally, the universal player should lose if an universal ghost variable $g^{\forall}$ is assigned a different value than the subformula that it represents.

In this paper, we never consider formulas (other than single literals) in which ghost variables occur as actual variables. In particular, if Φ is the input formula to the QBF solver, then in a substitution $\Phi|\pi$, ghost variables in π have no effect.

Definition 1 (Consistent assignment to ghost literal). Given a quantifier type $Q \in \{\exists, \forall\}$ and an assignment π, we say that a ghost literal g^{Q} is assigned **consistently** under π iff $g^{Q}|\pi =$ (the formula represented by g^{Q})$|\pi$.

Definition 2 (Winning under a total assignment). Given a formula Φ, a quantifier type $Q \in \{\exists, \forall\}$, and an assignment π to all the input variables and a subset of the ghost variables, we say "Player Q **wins** Φ under π" iff:

- $\Phi|\pi = \mathsf{true}$ if Q is $\exists$, and
- $\Phi|\pi = \mathsf{false}$ if Q is $\forall$, and
- Every ghost variable owned by Q in vars(π) is assigned consistently. (Intuitively, a winning player's ghost variables must "respect the encoding").

For example, if $\Phi = \exists e. \forall u. (e \wedge u)$ and g labels $(e \wedge u)$ then neither player wins Φ under $\{\neg e, u, g^{\forall}, \neg g^{\exists}\}$. The existential player loses because $\Phi|\pi = \mathsf{false}$, and the universal player loses because $g^{\forall}|\pi \neq$ (the formula represented by $g^{\forall}$)$|\pi$.

Definition 3 (Losing under a total assignment). Given a formula Φ and an assignment π that assigns all the input variables, we say "Player Q **loses** Φ under π" iff Player Q does not win Φ under π.

Definition 4 (Losing under a partial assignment). Given a formula Φ, an assignment π, and an outermost unassigned input variable x, we say "Player Q **loses** Φ under π" iff either:

- Player Q loses Φ under both $\pi \cup \{(x, \mathsf{true})\}$ and $\pi \cup \{(x, \mathsf{false})\}$, or
- Q's opponent owns x and Player Q loses Φ under either $\pi \cup \{(x, \mathsf{true})\}$ or $\pi \cup \{(x, \mathsf{false})\}$.

For example, consider a formula $\Phi = \exists e. x \wedge e$, where x is a free variable. The existential player loses Φ under $\{\neg x\}$ and under $\{\neg e\}$. Neither player can be said to lose Φ under the empty assignment, because the value of Φ depends on the free variable x. Now let us make a few general observations about when a player loses under an arbitrary partial assignment.

Observation 1. If $\Phi|\pi = \mathsf{true}$, then Player $\forall$ loses Φ under π.

Observation 2. If $\Phi|\pi = \mathsf{false}$, then Player $\exists$ loses Φ under π.

Observation 3. If a ghost variable owned by Q in vars(π) is assigned inconsistently under π, then Player Q loses Φ under π.

Observation 4. If the opponent of Q owns a literal ℓ that is unassigned under π, and Q loses Φ under $\pi \cup \{\ell\}$, then Q loses Φ under π.

Definition 5 (Game-State Specifier, Match). A **game-state specifier** is a pair $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ consisting of two sets of literals, L^{now} and L^{fut}. We say that $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ **matches** an assignment π iff:

1. for every literal ℓ in L^{now}, $\ell|\pi = \mathsf{true}$, and
2. for every literal ℓ in L^{fut}, either $\ell|\pi = \mathsf{true}$ or $\ell \notin \text{vars}(\pi)$.

For example, $\langle \{u\}, \{e\} \rangle$ matches the assignments $\{u\}$ and $\{u, e\}$, but does not match $\{\}$ or $\{u, \neg e\}$. Note that, for any literal ℓ, if $\{\ell, \neg \ell\} \subseteq L^{\text{fut}}$, then $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ matches an assignment π only if π doesn't assign ℓ. The intuition behind the names "L^{now}" and "L^{fut}" is as follows: Under the game formulation of QBF, the assignment π can be thought of as a state of the game, and π matches $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ iff every literal in L^{now} is already true in the game and, for every literal ℓ in L^{fut}, it is possible that ℓ can be true in a future state of the game.

Definition 6 (Game Sequent). The sequent "$\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi)$" means "Player Q loses Φ under all assignments that match $\langle L^{\text{now}}, L^{\text{fut}} \rangle$."

As an example, let $\varPhi$ be the following formula:

$$\forall u.\, \exists e.\, (e \vee \neg u) \wedge (u \vee \neg e) \wedge \overbrace{(x_1 \vee e)}^{g_3}$$

Note that sequent $\langle \{u\}, \{e\} \rangle \models (\forall \text{ loses } \varPhi)$ holds true: in any assignment π that matches it, $\varPhi|\pi = \mathsf{true}$. However, $\langle \{u\}, \varnothing \rangle \models (\forall \text{ loses } \varPhi)$ does not hold true: it matches the assignment $\{u, \neg e\}$, under which Player $\forall$ does not lose $\varPhi$. Finally, $\langle \{g_3^\forall\}, \{e, \neg e\} \rangle \models (\forall \text{ loses } \varPhi)$ holds true. Let us consider why Player $\forall$ loses $\varPhi$ under the assignment $\{g_3^\forall\}$. The free variable x_1 is the outermost unassigned variable, so under Definition 4, Player $\forall$ loses under $\{g_3^\forall\}$ iff Player $\forall$ loses under both $\{g_3^\forall, x_1\}$ and $\{g_3^\forall, \neg x_1\}$. Under $\{g_3^\forall, x_1\}$, Player $\forall$ loses because $\varPhi|\{g_3^\forall, x_1\}$ evaluates to true. Under $\{g_3^\forall, \neg x_1\}$, Player $\forall$ loses because e is owned by the opponent of Player $\forall$ and $g_3^\forall$ is assigned inconsistently under $\{g_3^\forall, \neg x_1, \neg e\}$.

Note that a clause $(\ell_1 \vee ... \vee \ell_n)$ in a CNF formula $\varPhi_{\text{in}}$ is equivalent to the sequent $\langle \{\neg\ell_1, ..., \neg\ell_n\},\, \varnothing \rangle \models (\exists \text{ loses } \varPhi_{\text{in}})$. (Sequents in this form can also be considered similar to *nogoods* [19].) Likewise, a cube $(\ell_1 \wedge ... \wedge \ell_n)$ in a DNF formula $\varPhi_{\text{in}}$ is equivalent to the sequent $\langle \{\ell_1, ..., \ell_n\},\, \varnothing \rangle \models (\forall \text{ loses } \varPhi_{\text{in}})$.

3.1 Sequents with Free Variables

Above, we introduced sequents that indicate if a player loses a formula $\varPhi$. Now, we will generalize sequents so that they can indicate that $\varPhi$ evaluates to a quantifier-free formula involving the free variables. To do this, we first introduce a logical semantics for QBF with ghost variables. Given a formula $\varPhi$ and an assignment π that assigns all the input variables, we want the semantic evaluation $[\![\varPhi]\!]_\pi$ to have the following properties:

1. $[\![\varPhi]\!]_\pi = \mathsf{true}$ iff the existential player wins $\varPhi$ under π.
2. $[\![\varPhi]\!]_\pi = \mathsf{false}$ iff the universal player wins $\varPhi$ under π.

Note that the above properties cannot be satisfied in a two-valued logic if both players lose $\varPhi$ under π. So, we use a three-valued logic with a third value dontcare. We call it “don’t care” because we are interested in the outcome of the game when both players make the best possible moves, but if both players fail to win, then clearly at least one of the players failed to make the best possible moves. In our three-valued logic, a conjunction of boolean values evaluates to false if any conjunct is false, and otherwise it evaluates to dontcare if any conjunct is dontcare. Disjunction is defined analogously. The negation of dontcare is dontcare. In a truth table:

x	y	$x \wedge y$	$x \vee y$
true	dontcare	dontcare	true
false	dontcare	false	dontcare

Definition 7. Given an assignment π to all the input variables and a subset of the ghost variables, we define $[\![\Phi]\!]_\pi$ as follows:

$$[\![\Phi]\!]_\pi := \begin{cases} \mathsf{true} & \text{if Player } \exists \text{ wins } \Phi \text{ under } \pi \\ \mathsf{false} & \text{if Player } \forall \text{ wins } \Phi \text{ under } \pi \\ \mathsf{dontcare} & \text{if both players lose } \Phi \text{ under } \pi \end{cases}$$

For convenience in defining $[\![\Phi]\!]_\pi$ for a partial assignment π, we assume that the formula is prepended with a dummy "quantifier" block for free variables. For example, $(\exists e.\, e \wedge z)$ becomes $(\mathcal{F}z.\, \exists e.\, e \wedge z)$, where $\mathcal{F}$ denotes the dummy block for free variables. If Φ contains free variables unassigned by π then $[\![\Phi]\!]_\pi$ is a formula in terms of these free variables. We define $[\![\Phi]\!]_\pi$ as follows for a partial assignment π that assigns only a proper subset of the input variables:

$$\begin{aligned} [\![Qx.\, \Phi]\!]_\pi &= [\![\Phi]\!]_\pi \quad \text{if } x \in \mathrm{vars}(\pi) \\ [\![\exists x.\, \Phi]\!]_\pi &= [\![\Phi]\!]_{(\pi \cup \{x\})} \vee [\![\Phi]\!]_{(\pi \cup \{\neg x\})} \quad \text{if } x \notin \mathrm{vars}(\pi) \\ [\![\forall x.\, \Phi]\!]_\pi &= [\![\Phi]\!]_{(\pi \cup \{x\})} \wedge [\![\Phi]\!]_{(\pi \cup \{\neg x\})} \quad \text{if } x \notin \mathrm{vars}(\pi) \\ [\![\mathcal{F}x.\, \Phi]\!]_\pi &= x \ ?\ [\![\Phi]\!]_{(\pi \cup \{x\})} \ :\ [\![\Phi]\!]_{(\pi \cup \{\neg x\})} \quad \text{if } x \notin \mathrm{vars}(\pi) \end{aligned}$$

The notation "$x \ ?\ \phi_1 : \phi_2$" denotes a formula with an *if-then-else* construct that is logically equivalent to $(x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$. Note that the branching on the free variables here is similar to the Shannon expansion [17].

Remark. Do we really need to add the dummy blocks for free variables and have the rule for $[\![\mathcal{F}x.\, \Phi]\!]_\pi$ in Definition 7? Yes, because if π contains a ghost literal g^Q that represents a formula containing variables free in Φ, then it doesn't make sense to ask if g^Q is assigned consistently under π unless all the variables in the formula represented by g^Q are assigned by π.

Definition 8 (Sometimes-Dontcare). A formula ϕ is said to be **sometimes-dontcare** iff there is an assignment π under which ϕ evaluates to dontcare. For example, $(x \vee \mathsf{dontcare})$ is sometimes-dontcare, while $(x \vee (x \wedge \mathsf{dontcare}))$ is not sometimes-dontcare (because it evaluates to true if x is true and evaluates to false if x is false).

Definition 9 (Free Sequent). The sequent "$\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models \Phi \Leftrightarrow \psi$" means "for all assignments π that match $\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle$, if $[\![\Phi]\!]_\pi$ is not sometimes-dontcare, then $[\![\Phi]\!]_\pi$ is logically equivalent to $\psi|\pi$".

Remark. The sequent definitions in Definitions 9 and 6 are related as follows:
- "$\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (\exists \text{ loses } \Phi)$" means the same as "$\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (\Phi \Leftrightarrow \mathsf{false})$".
- "$\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (\forall \text{ loses } \Phi)$" means the same as "$\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (\Phi \Leftrightarrow \mathsf{true})$".

We treat a game sequent as interchangeable with the corresponding free sequent.

Sequents of the form $\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models \Phi \Leftrightarrow \psi$ extend clause/cube learning by allowing ψ to be a formula (in terms of the variables free in Φ) in addition to the constants true and false. This enables handling of formulas with free variables.

4 Algorithm

The top-level algorithm, shown in Figure 2, is based on the well-known DPLL algorithm, except that sequents are used instead of clauses. Similar to how SAT solvers maintain a *clause database* (i.e., a set of clauses whose conjunction is equisatisfiable with the original input formula Φ_{in}), our solver maintains a *sequent database*. A SAT solver's clause database is initialized to contain exactly the set of clauses produced by the Tseitin transformation of the input formula Φ_{in} into CNF. Likewise, our sequent database is initialized (§ 4.1) to contain a set of sequents analogous to the clauses produced by the Tseitin transformation.

In the loop on lines 4–7, the solver chooses an outermost unassigned literal, adds it to π_{cur}, and performs boolean constraint propagation (BCP). BCP may add further literals to π_{cur}, as described in detail in § 4.4; such literals are referred to as *forced literals*, in distinction to the literals added by `DecideLit`, which are referred to as *decision literals*. The stopping condition for the loop is when the current assignment matches a sequent already in the database. (The analogous stopping condition for a SAT solver would be when a clause is falsified.) When this stopping condition is met, the solver performs an analysis similar to that of *clause learning* [18] to learn a new sequent (line 8). If the L^{now} component of the learned sequent is empty, then the solver has reached the final answer, which it returns (line 9). Otherwise, the solver backtracks to the earliest decision level at which the newly learned sequent will trigger a forced literal in BCP. (The learning algorithm guarantees that this is possible.) The solver then performs BCP (line 11) and returns to the inner loop at line 4.

The intuition behind BCP for quantified variables is fairly straightforward; a literal owned by Q is forced by a sequent if the sequent indicates that Q need to make ℓ true to avoid losing. For free variables, the intuition is slightly different. Free variables are forced to prevent the solver from re-exploring parts of the

```
 1.  initialize_sequent_database();
 2.  $\pi_{\text{cur}}$ := $\varnothing$;  Propagate();
 3.  while (true) {
 4.    while ($\pi_{\text{cur}}$ doesn't match any database sequent) {
 5.      DecideLit();
 6.      Propagate();
 7.    }
 8.    Learn();
 9.    if (learned seq has form $\langle \varnothing,\ L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$) return $\psi$;
10.    Backtrack();
11.    Propagate();
12.  }
```

Fig. 2. Top-Level Algorithm. Details have been omitted for sake of clarity.

search space that it has already seen, so that the solver is continuously making progress in exploring the search space, thereby guaranteeing it would eventually terminate (given enough time and memory). (Actually, this intuition also applies to quantified variables.)

The solver maintains a list of assigned literals in the order in which they were assigned; this list is referred to as the *trail* [9]. Given a decision literal ℓ_d, we say that all literals that appear in the trail after ℓ_d but before any other decision literal belong to the same *decision level* as ℓ_d.

For prenex formulas without free variables, the algorithm described here is operationally very similar to standard DPLL QBF solvers, except that L^{now} and L^{fut} do not need to be explicitly separated, since L^{now} always consists exactly of all the loser's literals. However, for formulas with free variables, it is necessary to explicitly record which literals belong in L^{now} and which in L^{fut}.

4.1 Initial Sequents

We initialize the sequent database to contain a set of *initial sequents*, which correspond to the clauses produced by the Tseitin transformation of the input formula Φ_{in}. The set of initial sequents must be sufficient to ensure the loop on line 4–6 of Figure 2 (which adds unassigned literals to the current assignment until it matches a sequent in the database) operates properly. That is, for every possible total assignment π, there must be at least one sequent that matches π.

First, let us consider a total assignment π in which both players assign all their ghost variables consistently (Definition 1). In order to handle this case, we generate the following two initial sequents, where g_{in} is the label of the input formula Φ_{in}: $\langle\{\neg g^{\exists}_{\text{in}}\}, \varnothing\rangle \models (\exists \text{ loses } \Phi_{\text{in}})$ and $\langle\{g^{\forall}_{\text{in}}\}, \varnothing\rangle \models (\forall \text{ loses } \Phi_{\text{in}})$.

Since all ghost variables are assigned consistently in π, it follows that, for each gate g, $g^{\exists}|\pi$ must equal $g^{\forall}|\pi$, since both $g^{\exists}$ and $g^{\forall}$ must each be assigned the same value as the formula that g labels. In particular, $g^{\exists}_{\text{in}}|\pi$ must be equal to $g^{\forall}_{\text{in}}|\pi$, so π must match exactly one of the two above initial sequents.

Now let us consider a total assignment π in which at least one player assigns a ghost variable inconsistently. In order to handle this case, we generate a set of initial sequents for every conjunction and disjunction in Φ_{in}. Let g_* be the label of an arbitrary conjunction in Φ_{in} of the form

$$\left(x_1 \wedge ... \wedge x_n \ \wedge \ \underbrace{\phi_1}_{g_1} \wedge ... \wedge \underbrace{\phi_m}_{g_m}\right)$$

where x_1 through x_n are input literals. The following initial sequents are produced from this conjunction for each $Q \in \{\exists, \forall\}$:

1. $\langle\{g^{Q}_{*},\, \neg x_i\}, \varnothing\rangle \models (Q \text{ loses } \Phi_{\text{in}})$ for $i \in \{1, ..., n\}$
2. $\langle\{g^{Q}_{*},\, \neg g^{Q}_{i}\}, \varnothing\rangle \models (Q \text{ loses } \Phi_{\text{in}})$ for $i \in \{1, ..., m\}$
3. $\langle\{\neg g^{Q}_{*},\, x_1, ..., x_n,\, g^{Q}_{1}, ..., g^{Q}_{m}\}, \varnothing\rangle \models (Q \text{ loses } \Phi_{\text{in}})$

Now let g^Q_* denote a ghost literal such that (1) g^Q_* is inconsistently assigned under π and (2) no proper subformula of the formula represented by g^Q_* is labelled by a inconsistently-assigned ghost variable. Then π must match one of the above-listed initials sequents.

4.2 Normalization of Initial Sequents

Note that all the initial sequents have the form $\langle L^{\text{now}},\, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi)$ where $L^{\text{fut}} = \varnothing$. We normalize these sequents by moving all literals owned by Q's opponent from L^{now} to L^{fut}, in accordance with the following inference rule:

$$\frac{\begin{array}{l}\text{The opponent of } Q \text{ owns } \ell\text{, and } \neg\ell \notin L^{\text{fut}} \\ \langle L^{\text{now}} \cup \{\ell\},\, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi)\end{array}}{\langle L^{\text{now}},\, L^{\text{fut}} \cup \{\ell\}\rangle \models (Q \text{ loses } \Phi)}$$

To prove the above inference rule, we consider an arbitrary assignment π that matches $\langle L^{\text{now}},\, L^{\text{fut}} \cup \{\ell\}\rangle$, assume that the premises of inference rule hold true, and prove that Player Q loses under π:

1. π matches $\langle L^{\text{now}},\, L^{\text{fut}} \cup \{\ell\}\rangle$ (by assumption).
2. $\pi \cup \{\ell\}$ matches $\langle L^{\text{now}} \cup \{\ell\},\, L^{\text{fut}}\rangle$ (using the premise that $\neg\ell \notin L^{\text{fut}}$).
3. Q loses Φ under $\pi \cup \{\ell\}$ (by the premise $\langle L^{\text{now}} \cup \{\ell\},\, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi)$).
4. Q loses Φ under π (by Observation 4 on page 5).

4.3 Properties of Sequents in Database

After the initial sequents have been normalized (as described in § 4.2), the solver maintains the following invariants for all sequents in the sequent database, including sequents added to the database as a result of learning (§ 4.5):

1. In a sequent of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})$:
   (a) Every literal in L^{now} either is owned by Q or is free in Φ_{in}.
   (b) Every literal in L^{fut} is owned by the opponent of Q.
2. In a sequent of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$, every variable in ψ appears both positively and negatively in L^{fut} (i.e., if r occurs in ψ, then $\{r, \neg r\} \subseteq L^{\text{fut}}$). This is guaranteed by the learning algorithm in § 4.5.

4.4 Propagation

The `Propagate` procedure is similar to that of closed-QBF solvers. Consider a sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$ in the sequent database. If, under π_{cur},

1. there is exactly one unassigned literal ℓ in L^{now}, and
2. no literals in $L^{\text{now}} \cup L^{\text{fut}}$ are assigned false, and
3. ℓ is not downstream of any unassigned literals in L^{fut},

then $\neg\ell$ is *forced* — it is added to the current assignment π_{cur}. In regard to the 3rd condition, if an unassigned literal r in L^{fut} is upstream of ℓ, then r should get assigned before ℓ, and if r gets assigned false, then ℓ shouldn't get forced at all by the sequent. Propagation ensures that the solver never re-explores areas of the search space for which it already knows the answer, ensuring continuous progress and eventual termination. It is instructive to consider how the propagation rule applies in light of the properties of sequents discussed in § 4.3:

1. A sequent of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})$ can force a literal that is either owned by Q or free in Φ_{in}; it cannot force a literal owned by Q's opponent. If ℓ is owned by Q, then the reason for forcing $\neg\ell$ is intuitive: the only way for Q to avoid losing is to add $\neg\ell$ to the current assignment. If ℓ is free in Φ_{in}, then $\neg\ell$ is forced because the value of $[\![\Phi_{\text{in}}]\!]\pi_{\text{cur}} \cup \{\ell\}$ is already known and the solver shouldn't re-explore that same area of the search space.
2. A sequent of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$, where ψ contains free variables, can only force a literal that is free in Φ_{in}. Although L^{now} can contain literals owned by Player $\exists$ and Player $\forall$, such literals cannot be forced by the sequent. To prove this, we consider two cases: either there exists a variable v that occurs in ψ and is assigned by π_{cur}, or all variables that occur ψ are left unassigned by π_{cur}. If there is variable v in ψ that is assigned by π_{cur}, then π_{cur} cannot match $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$, since $\{v, \neg v\} \subseteq L^{\text{fut}}$. If there is a variable v in ψ that is left unassigned by π_{cur}, then $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$ cannot force any quantified variable, since v occurs in L^{fut} and all quantified variables are downstream of free variable v.

We employ a variant of the watched-literals rule designed for SAT solvers [16] and adapted for QBF solvers [10]. For each sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \psi)$, we watch two literals in L^{now} and one literal in L^{fut}.

4.5 Learning

In the top-level algorithm in Figure 2, the solver performs *learning* (line 8) after the current assignment π_{cur} matches a sequent in the database. The learning procedure is based on the clause learning introduced for SAT in [18] and adapted for QBF in [24]. We use inference rules shown in Figure 4 to add new sequents to the sequent database. These rules, in their L^{now} components, resemble the *resolution* rule used in SAT (i.e., from $(A \vee r) \wedge (\neg r \vee B)$ infer $A \vee B$). The learning algorithm ensures that the solver remembers the parts of the search space for which it has already found an answer. This, together with propagation, ensures that solver eventually covers all the necessary search space and terminates.

The learning procedure, shown in Figure 3, works as follows. Let *seq* be the database sequent that matches the current assignment π_{cur}. Let r be the literal in the L^{now} component of *seq* that was most recently added to π_{cur} (i.e., the latest one in the *trail*). Note that r must be a forced literal (as opposed to a decision literal), because only an outermost unassigned literal can be picked as a decision literal, but if r was outermost immediately before it added to π_{cur},

```
func Learn() {
    seq := (the database sequent that matches $\pi_{\text{cur}}$);
    do {
        $r$ := (the most recently assigned literal in seq.L^now)
        seq := Resolve(seq, antecedent[$r$]);
    } until (seq.L^now $= \varnothing$ or has_good_UIP(seq));
    return seq;
}
```

Fig. 3. Procedure for learning new sequents

Resolving on a literal r owned by Player Q (case 1):

$$\frac{\begin{array}{l}\text{The quantifier type of } r \text{ in } \Phi \text{ is } Q\\ \langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})\\ \langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})\\ r \text{ is not downstream of any } \ell \text{ such that } \ell \in L_1^{\text{fut}} \text{ and } \neg\ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})\end{array}}{\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})}$$

Resolving on a literal r owned by Player Q (case 2):

$$\frac{\begin{array}{l}\text{The quantifier type of } r \text{ in } \Phi \text{ is } Q\\ \langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})\\ \langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)\\ r \text{ is not downstream of any } \ell \text{ such that } \ell \in L_1^{\text{fut}} \text{ and } \neg\ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})\end{array}}{\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{\neg r\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)}$$

Resolving on a variable r that is free in Φ_{in}:

$$\frac{\begin{array}{l}\text{Literal } r \text{ is free}\\ \langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_1)\\ \langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_2)\end{array}}{\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{r, \neg r\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (r \mathbin{?} \psi_1 : \psi_2))}$$

Fig. 4. Resolution-like inference rules.

then no unassigned literal in the L^{fut} component of *seq* was upstream of r, so *seq* would have forced $\neg r$ in accordance with § 4.4. We use the inference rules in Figure 4 to infer a new sequent from *seq* and the *antecedent* of r (i.e., the sequent that forced r). This is referred to as *resolving* due to the similarity of the inference rules to the clause resolution rule. We stop and return the newly inferred sequent if it has a "good" unique implication point (UIP) [24], i.e., if there is a literal ℓ in the L^{now} component such that

1. Every literal in $(L^{\text{now}} \setminus \{\ell\})$ belongs to an earlier decision level than ℓ,
2. Every literal in L^{fut} upstream of ℓ belongs to a decision level earlier than ℓ.
3. If *seq* has the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})$, then the decision variable of the decision level of ℓ is not owned by the opponent of Q.

Otherwise, we resolve the sequent with the antecedent of the most recently assigned literal in its L^{now} component, and continue this process until the stopping conditions above are met or L^{now} is empty. Note that if the most recently assigned literal in L^{now} is a decision literal, then it is a good UIP.

Note that in the resolution rule for resolving on a free variable r, we add both r and $\neg r$ to L^{fut}. This is not necessary for soundness of the resolution itself. Rather, it is to ensure that the properties in § 4.3 hold true. Without these properties, a quantified variable could be forced by a sequent that is not equivalent to a clause or a cube.

Example. Below, we give several applications of the resolution rules. For brevity, we omit free variables from the L^{fut} component.

$$\exists e_3.\ \underbrace{(i_1 \wedge e_3)}_{g_5} \vee \underbrace{(i_2 \wedge \neg e_3)}_{g_4}$$

1. Start: $\langle\{\neg i_1, \neg i_2\}, \{\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \mathsf{false})$

2. Resolve $\neg i_1$ via $\langle\{i_1, \neg g_5^{\forall}\}, \{e_3\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \mathsf{true})$
   Result: $\langle\{\neg i_2, \neg g_5^{\forall}\}, \{e_3\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow i_1)$

3. Resolve $\neg i_2$ via $\langle\{i_2, \neg g_4^{\forall}\}, \{\neg e_3\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \mathsf{true})$
   Result: $\langle\{\neg g_5^{\forall}, \neg g_4^{\forall}\}, \{e_3, \neg e_3\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$

4. Resolve $\neg g_4^{\forall}$ via $\langle\{g_4^{\forall}\}, \{\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \mathsf{true})$
   Result: $\langle\{\neg g_5^{\forall}\}, \{e_3, \neg e_3, \neg g_4^{\forall}\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$

5. Resolve $\neg g_5^{\forall}$ via $\langle\{g_5^{\forall}\}, \{\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \mathsf{true})$
   Result: $\langle\{\}, \{e_3, \neg e_3, \neg g_4^{\forall}, \neg g_5^{\forall}\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$

4.6 Justification of inference rules

The first inference rule in Figure 4 is analogous to long-distance resolution [23] and can be proved by similar methods (e.g., [2]). Intuitively, if the current

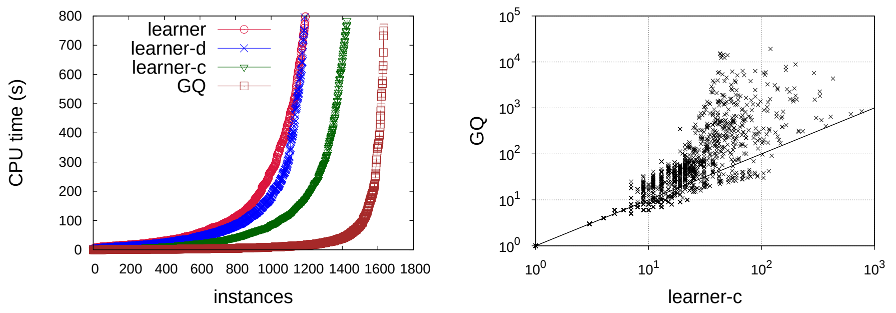

Fig. 5. Time and size comparisons, instances solved by all solvers in less than 10 s are not included in the time comparison.

assignment matches $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\ L_1^{\text{fut}} \cup L_2^{\text{fut}} \rangle$, then the opponent of Q can make Q lose Φ_{in} by assigning true to all the literals in L_1^{fut} that are upstream of r. This forces Q to assign $r = \mathsf{false}$ to avoid matching the first sequent in the premise of the inference rule, but assigning $r = \mathsf{false}$ makes the current assignment match the second sequent in the premise.

If the current assignment π_{cur} matches the sequent in the conclusion of the second inference rule, there are two possibilities. For simplicity, assume that π_{cur} assigns all free variables and that neither L_1^{fut} nor L_2^{fut} contains any free literals (since, as mentioned earlier, free literals can be removed from L^{fut} without affecting soundness of the sequent). If Q loses ψ under π_{cur}, then the situation is similar to first inference rule. If the opponent of Q loses ψ under π_{cur}, then Q can make his opponent lose Φ_{in} by assigning $r = \mathsf{false}$, thereby making the current assignment match the second sequent of the premise.

For the third inference rule, we don't need a condition about r not being downstream of other literals, since no free variable is downstream of any variable.

5 Experimental Results

We extended the existing closed-QBF solver GhostQ [14] to implement the techniques described in this paper. For comparison, we used the solvers and load-balancer benchmarks from [3].[4] The benchmarks contain multiple alternations of quantifiers and are derived from problems involving the automatic synthesis of a reactive system from a formal specification. The experimental results were obtained on Intel Xeon 5160 3-GHz machines with 4 GB of memory. The time limit was 800 seconds and the memory limit to 2 GB.

[4] The results do not exactly match the results reported in [3] because we did not preprocess the QDIMACS input files. We found that sometimes the output of the preprocessor was not logically equivalent to its input. With the unpreprocessed inputs, the output formulas produced by the learner family of solvers were always logically equivalent to the output formulas of GhostQ.

There are three solvers from [3], each with a different form of the output: CDNF (a conjunction of DNFs), CNF, and DNF. We will refer to these solvers as “Learner” (CNDF), “Learner-C” (CNF), and “Learner-D” (DNF). Figure 5 compares these three solvers with GhostQ on the “hard” benchmarks (those that not all four solvers could solve within 10 seconds). As can be seen on the figure, GhostQ solved about 1600 of these benchmarks, Learner-C solved about 1400, and Learner-D and Learner each solved about 1200. GhostQ solved 223 instances that Learner-C couldn’t solve, while Learner-C solved 16 instances that GhostQ couldn’t solve. GhostQ solved 375 instances that neither Learner-DNF nor Learner could solver, while there were only 2 instances that either Learner-DNF or Learner could solve but GhostQ couldn’t solve.

Figure 5 shows a comparison of the size of the output formulas for GhostQ and Learner-C, indicating that the GhostQ formulas are often significantly larger. The size is computed as 1 plus the number of edges in the DAG representation of the formula, not counting negations, and after certain simplifications. E.g., the size of x is 1, the size of $\neg x$ is also 1, and the size of $x \wedge y$ is 3.

6 Related Work

Ken McMillan [15] proposed a method to use SAT solvers to perform quantifier elimination on formulas of the form $\exists \boldsymbol{x}.\, \phi$, generating CNF output. This problem (i.e, given a formula $\exists \boldsymbol{x}.\, \phi$, return a logically equivalent quantifier-free CNF formula) has received attention recently. Brauer, King, and Kriener [6] designed an algorithm that combines model enumeration with prime implicant generation. Goldberg and Manolios [11] developed a method based on *dependency sequents*; experimental results show that it works very well on forward and backward reachability on the Hardware Model Checking Competition benchmarks. For QBFs with arbitrary quantifier prefixes, the only other work of which we are aware is that of Becker, Ehlers, Lewis, and Marin [3], which uses computational learning to generate CNF, DNF, or CDNF formulas, and that of Benedetti and Mangassarian [5], which adapts sKizzo [4] for open QBF. The use of SAT solvers to build unordered BDDs [21] and OBDDs [13] has also been investigated.

7 Conclusion

This paper has shown how a DPLL-based closed-QBF solver can be extended to handle free variables. The main novelty of this work consists of generalizing clauses/cubes (and the methods involving them), yielding sequents that can include a formula in terms of the free variables. Our extended solver GhostQ produces unordered BDDs, which have several favorable properties [8]. However, in practice, the formulas tended to fairly large in comparison to equivalent CNF representations. Unordered BDDs can often be larger than equivalent OBDDs, since logically equivalent subformulas can have multiple distinct representations in an unordered BDD, unlike in an OBDD. Although our BDDs are necessarily unordered due to unit propagation, in future work it may be desirable to investigate techniques aimed at reducing the size of the output formula.

References

1. C. Ansótegui, C. P. Gomes, and B. Selman. The Achilles' Heel of QBF. In *AAAI 2005*.
2. V. Balabanov and J.-H. R. Jiang. Unified QBF certification and its applications. *Formal Methods in System Design*, 41(1):45–65, 2012.
3. B. Becker, R. Ehlers, M. D. T. Lewis, and P. Marin. ALLQBF Solving by Computational Learning. In *ATVA*, 2012.
4. M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *CADE*, 2005.
5. M. Benedetti and H. Mangassarian. QBF-Based Formal Verification: Experience and Perspectives. *JSAT*, 2008.
6. J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, 2011.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
8. A. Darwiche and P. Marquis. A Knowledge Compilation Map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002.
9. N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, pages 502–518, 2003.
10. I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In *SAT 2003*.
11. E. Goldberg and P. Manolios. Quantifier elimination by Dependency Sequents. In G. Cabodi and S. Singh, editors, *FMCAD*, pages 34–43. IEEE, 2012.
12. A. Goultiaeva and F. Bacchus. Exploiting QBF Duality on a Circuit Representation. In *AAAI*, 2010.
13. J. Huang and A. Darwiche. Using DPLL for Efficient OBDD Construction. In *SAT*, 2004.
14. W. Klieber, S. Sapra, S. Gao, and E. M. Clarke. A Non-prenex, Non-clausal QBF Solver with Game-State Learning. In *SAT*, 2010.
15. K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.
16. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC 2001*.
17. C. E. Shannon. The Synthesis of Two Terminal Switching Circuits. *Bell System Technical Journal*, 28:59–98, 1949.
18. J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.
19. R. M. Stallman and G. J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artif. Intell.*, 9(2):135–196, 1977.
20. G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.
21. R. Wille, G. Fey, and R. Drechsler. Building free binary decision diagrams using SAT solvers. *Facta universitatis-series: Electronics and Energetics*, 2007.
22. L. Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *AAAI 2006*.
23. L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *ICCAD 2002*.
24. L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *CP 2002*.