Formal Verification Using Quantified Boolean Formulas (QBF)

William Klieber

CMU-CS-14-117
May 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Edmund M. Clarke, chair
Randal E. Bryant
Jeannette M. Wing
João P. Marques-Silva (University College Dublin)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Semiconductor Research Corporation, by a Google Research Award, and by Portugal Fundação para a Ciência e a Tecnologia (FCT) grant ATTEST (CMU-PT/ELE/0009/2009).

Keywords: Formal verification, formal methods, model checking, QBF, SAT, non-prenex, non-clausal, DPLL, CEGAR

To my parents.

Abstract

Many problems in formal verification of digital hardware circuits and other finite-state systems are naturally expressed in the language of quantified boolean formulas (QBF). This thesis presents advancements in techniques for QBF solvers that enable verification of larger and more complex systems.

Most of the existing work on QBF solvers has required that formulas be converted into *prenex conjunctive normal form (CNF)*. However, the Tseitin transformation (which is used to convert a formula to CNF) is asymmetric between the two quantifier types. In particular, it introduces new existentially quantified variables, but not universally quantified variables. It turns out that this makes it harder for QBF solvers to detect when a formula has become true (but not when a formula has become false) under an assignment.

We present a technique using *ghost variables* that handles non-CNF QBF formulas using a symmetric alternative to the Tseitin transformation. We introduce *sequent learning*, a reformulation and generalization of *clause/cube learning*. With sequent learning, we can handle non-prenex formulas. Whereas clause/cube learning only allows learning when the whole input formula becomes true or false, sequent learning allows us to learn when a quantified subformula becomes true or false.

Almost all QBF research so far has focused on *closed* formulas, i.e., formulas without any free variables. Closed QBF formulas evaluate to either `true` or `false`. Sequent learning lets us extend existing QBF techniques to handle *open* formulas, which contain free variables. A solution to an open QBF formula is a quantifier-free formula that is logically equivalent to the original formula. For example, a solution to the open QBF formula $\exists x.\ (x \wedge y) \vee z$ is the formula $y \vee z$.

The final part of this thesis discusses an approach to QBF that uses Counterexample-Guided Abstraction Refinement (CEGAR) to partially expand quantifier blocks. The approach recursively solves QBF instances with multiple quantifier alternations. Experimental results show that the recursive CEGAR-based approach outperforms existing types of solvers on many publicly-available benchmark families. In addition, we present a method of combining the CEGAR technique with a DPLL-based solver and show that it improves the DPLL solver on many benchmarks.

Acknowledgments

First, I would like to thank my advisor, Edmund Clarke, for his guidance, support, and encouragement throughout my graduate career. I am grateful to Ed for the freedom he allowed me in exploring my interests in static analysis and satisfiability solvers. Ed taught me a great deal about how to conduct research and how to communicate ideas effectively. I am grateful to my thesis committee members, Randy Bryant, Jeannette Wing, and João Marques-Silva, for their invaluable advice and feedback.

It was a great pleasure to collaborate with Mikoláš Janota on much of the work described in this thesis. I would also like to thank my fellow graduate students in Ed Clarke's group: Samir Sapra, Soonho Kong, Anvesh Komuravelli, Sicun Gao, Himanshu Jain, and Qinsi Wang. I'd also like to thank Ed's many postdocs and visitors, especially Paolo Zuliani. I am grateful to Sagar Chaki and Arie Gurfinkel for their insightful feedback on my static analysis work.

I would like to thank Prof. Kwangkeun Yi for inviting me to collaborate with him for a summer at Seoul National University. I learned a great deal from him about research in static analysis and abstract interpretation. I also greatly enjoyed experiencing Korean culture, and I even acquired a taste for kimchi during my stay.

I would like to thank Dr. Steve Miller, who I worked for during my internship at Rockwell Collins, for his encouragement and support.

I would like to thank Robert C. Seacord for his understanding in my postponement of my employment starting date at SEI as my thesis defense timeframe slowly shifted from January to August 2013.

Finally, I owe a great debt of gratitude to my family for their love and encouragement.

Contents

Chapter 1

Introduction

Many problems in formal verification (among other areas) are naturally expressed in the language of Quantified Boolean Formulas (QBF). QBF is an extension of propositional logic in which boolean variables can be quantified. Syntactically, we consider QBF formulas described by the following grammar:

$$\begin{array}{rcl} x & ::= & \textsf{boolean variable} \\ Q & ::= & \exists \mid \forall \\ \phi & ::= & x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \textsf{true} \mid \textsf{false} \\ \Phi & ::= & \phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid Qx.\, \Phi \end{array}$$

Note that in the above grammar, quantifiers cannot occur within the scope of a negation; this requirement is imposed to simplify matters. Also, we require that all occurrences of a variable have the same binding (if any). For example, $x \wedge \exists x.\,(y \wedge x)$ is disallowed, because the first occurence of x is free but the last occurrence is bound by an existential quantifier.

A QBF instance is *closed* iff every occurrence of every variable is bound by a quantifier. If not all variables are bound by quantifiers, the QBF formula is said to be *open*. In chapters 2 and 4, we will consider open QBF; in all other chapters, we will only consider closed instances.

A *literal* is a variable or its negation. For a literal ℓ, $\mathsf{var}(\ell)$ denotes the variable in ℓ, i.e., if x is a variable, then $\mathsf{var}(\neg x) = \mathsf{var}(x) = x$.

A *clause* is a disjunction of literals. A boolean formula in *conjunctive normal form (CNF)* is a conjunction of clauses.

A *cube* is a conjunction of literals. A boolean formula in *disjunctive normal form (DNF)* is a disjunction of cubes.

Assignments. In this thesis document, when we use the term "assignment", we mean a mapping of variables to boolean values. For example, the assignment $\{(x, \mathsf{true})\}$ maps the variable x to the boolean value true.

A *total* assignment to a set of variables V maps every variable in V to a boolean value. A *partial* assignment to V maps a subset of the variables in V to boolean values. For example, if $V = \{x, y\}$, then the assignment $\{(x, \mathsf{true})\}$ is a partial assignment to V, and the assignment $\{(x, \mathsf{true}), (y, \mathsf{false})\}$ is a total assignment to V.

For convenience, we identify an assignment 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 the context of a QBF formula such as $\forall x. \exists y. \phi$, where the quantifier of a variable y occurs inside the scope of the quantifier of a variable 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. We say that a variable is **outermost** under an assignment π iff it is not downstream of any unassigned variables. For example, for the QBF formula $\exists e_1. \forall u_2. \exists e_3. \phi$, under the assignment $\pi = \{e_1\}$, the variable u_2 is outermost (because e_1, the only variable upstream of u_2, is assigned under π), but e_3 is not outermost (because it is downstream of the unassigned variable u_2).

The terms *downstream*, *upstream*, and *outermost* can also be applied to literals. We say that a literal ℓ_1 is downstream (or resp. upstream) of a literal ℓ_2 iff $\mathsf{var}(\ell_1)$ is downstream (or resp. upstream) of $\mathsf{var}(\ell_2)$. A literal ℓ is outermost if $\mathsf{var}(\ell)$ is outermost.

Prenex Form. A formula is in **prenex form** iff it has the form $(Q_1X_1...Q_nX_n.\,\phi)$, where ϕ is quantifier-free. We say that $Q_1X_1...Q_nX_n$ is the **quantifier prefix** and that ϕ is the **matrix**.

Definition 1.1 (Substitution). Let Φ be a formula, and let π be a partial assignment, i.e., an assignment that maps some of the variables in Φ to boolean values (true, false). Recall that we require that all occurrences of a variable have the same binding (if any). I.e., unlike first-order logic, we do not allow a variable in Φ to be quantified more than once, and we do not allow a variable to have both free occurrences and bound occurrences. With this in mind, 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). Formally:

$$x|\pi = \begin{cases} \mathsf{true} & \text{if } (x, \mathsf{true}) \in \pi \\ \mathsf{false} & \text{if } (x, \mathsf{false}) \in \pi \\ x & \text{otherwise} \end{cases} \qquad \begin{aligned} (\neg\Phi)|\pi &= \neg(\Phi|\pi) \\ (\Phi_1 \wedge \Phi_2)|\pi &= (\Phi_1|\pi) \wedge (\Phi_2|\pi) \\ (\Phi_1 \vee \Phi_2)|\pi &= (\Phi_1|\pi) \vee (\Phi_2|\pi) \\ (r \ ?\ \psi_1 \ :\ \psi_2)|\pi &= r|\pi \ ?\ \psi_1|\pi \ :\ \psi_2|\pi \end{aligned}$$

$$(Qx.\,\Phi)|\pi = \begin{cases} \Phi|\pi & \text{if } x \in \text{vars}(\pi) \\ Qx.\,(\Phi|\pi) & \text{otherwise} \end{cases}$$

The notation "$r\ ?\ \psi_1\ :\ \psi_2$" denotes an *if-then-else* construct; it is not part of our

QBF grammar, but it will be used in Chapter 4 in quantifier-free formulas. An assignment π is said to *satisfy* a formula Φ iff $\Phi|\pi$ evaluates to true.

Semantics. Semantically, boolean quantifiers are defined as follows:

- Universal quantifier: $\forall x.\, \Phi \;=\; \Phi|\{x\} \wedge \Phi|\{\neg x\}$
- Existential quantifier: $\exists x.\, \Phi \;=\; \Phi|\{x\} \vee \Phi|\{\neg x\}$

Lemma 1.1. Given a formula Φ and an assignment π, if an existential (or respectively universal) literal ℓ is outermost in Φ under π, and $\Phi|\pi \cup \{\ell\}$ evaluates to true (resp. false), then $\Phi|\pi$ also evaluates to true (resp. false).

Example. Consider the formula $\Phi = \forall u.\exists e.\ (e \wedge u) \vee (\neg e \wedge \neg u)$ and the assignment $\pi = \{u\}$. Then e is outermost in Φ under π, and $\Phi|\pi \cup \{e\} = \mathsf{true}$, so $\Phi|\pi = \mathsf{true}$.

QBF as a Game. A closed prenex 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 $\forall$.
- On each turn of the game, the owner of an outermost unassigned variable assigns it a value.
- 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 $var(\ell)$.

The above definition of *goals* and the below definition of *optimal-strategy assignment* were chosen to ensure the following: If π is an assignment produced by both players following an optimal strategy, then $\Phi|\pi$ evaluates to the same truth value as Φ.

Definition 1.2 (Optimal-Strategy Assignment). We inductively define the set of *optimal-strategy assignments* for Φ as follows:

1. The empty assignment is an optimal-strategy assignment for Φ.
2. If π is an optimal-strategy assignment for Φ, and

   a literal ℓ is outermost in Φ under π, and

   $\Phi|\pi \cup \{\ell\}$ evaluates to the same truth value as $\Phi|\pi$,

   then $\pi \cup \{\ell\}$ is an optimal-strategy assignment for Φ.

For example:

- Consider the formula $\Phi = \exists e.\forall u.\ e \vee u$. The assignment $\{e\}$ is an optimal-strategy assignment, because Φ evaluates to true and so does $\Phi|\{e\}$. The assignment $\{\neg e\}$ is not an optimal-strategy assignment, because $\Phi|\{\neg e\}$ evaluates to false while Φ evaluates to true.
- Consider the formula $\Phi = \forall u.\exists e.\ u \wedge (\neg u \vee e)$. The assignment $\{\neg u\}$ is an optimal-strategy assignment, because Φ evaluates to false and so does $\Phi|\{\neg u\}$. The assignment $\{u\}$ is not an optimal-strategy assignment, because $\Phi|\{u\}$ evaluates to true while Φ evaluates to false.
- Consider the formula $\Phi = \forall u.\exists e.\ (e \wedge u) \vee (\neg e \wedge \neg u)$. Both $\{u\}$ and $\{\neg u\}$ are optimal-strategy assignments.

As the above examples illustrate, if the existential player can make the formula true, then he must do so in order to play an optimal strategy as defined above. This observation is stated more precisely in the following lemma:

Lemma 1.2. If π is an optimal-strategy assignment for a formula Φ, and

1. an existential literal ℓ is outermost under π, and
2. $\Phi|\pi \cup \{\ell\}$ evaluates to true, and
3. $\Phi|\pi \cup \{\neg\ell\}$ evaluates to false,

then $\pi \cup \{\ell\}$ is an optimal-strategy assignment but $\pi \cup \{\neg\ell\}$ is not.

Proof. By premises 1 and 2, and Lemma 1.1, $\Phi|\pi$ evaluates to true. The conclusion of the lemma follows from the definition of *optimal-strategy assignment*.

A similar lemma can be proved from the universal player.

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", as distinct from gate variables.

Definition 1.3 (**Winning and losing under an input assignment**)**.** Let Φ be a closed QBF formula and let π be an partial assignment to the input variables (i.e., π does not assign any gate variables).

- We say "Player $\exists$ wins Φ under π" iff $\Phi|\pi = \mathsf{true}$.
- We say "Player $\exists$ loses Φ under π" iff $\Phi|\pi = \mathsf{false}$.
- We say "Player $\forall$ wins Φ under π" iff $\Phi|\pi = \mathsf{false}$.
- We say "Player $\forall$ loses Φ under π" iff $\Phi|\pi = \mathsf{true}$.

Definition 1.4 (Disjoint Assignments)**.** Two assignments π_1 and π_2 are said to be *disjoint* iff vars(π_1) is disjoint from vars(π_2).

Quantifier Blocks. The prefix of a prenex instance is divided into *quantifier blocks*, each of which is a subsequence $\forall x_1 \ldots \forall x_n$ or respectively $\exists x_1 \ldots \exists x_n$, which we denote by $\forall X$ or respectively $\exists X$, where $X = \{x_1, ..., x_n\}$.

SAT Solvers

A QBF formula in which all variables are existentially quantified is essentially a SAT problem. Many modern QBF solvers use techniques that were first developed for SAT solvers, so it is instructive to briefly consider SAT solvers before considering QBF in full generality. Almost all modern SAT solvers use a variant of the DPLL algorithm [19]. This algorithm uses a backtracking search. It requires that the formula be tranformed into conjunctive normal form (CNF), which is done using the *Tseitin tranformation* (described in Section 2.1 on page 10).

In the DPLL algorithm, the solver picks a variable in the input formula and arbitrarily assigns it a boolean value (true or false), simplifying the formula under the assignment. This assignment is called a *decision*. The solver then performs *unit propagation*: If the formula has a *unit clause* (a clause with exactly one literal), then the solver assigns that literal true and simplifies the formula under that assignment. Unit propagation is repeated until there are no more unit clauses. If a satisfying assignment (i.e., an assignment that makes the formula true) is discovered, the solver returns true. If a falsifying assignment is discovered, the solver backtracks, undoing its decisions. High-level pseudocode of the DPLL algorithm is shown in Fig. 1.1.

For example, consider the formula $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$. Under the decision $x_3 = \mathsf{true}$, the formula becomes: $(\mathsf{true}) \wedge (x_1) \wedge (\neg x_1 \vee x_2)$. The clause (x_1) is a unit clause, so $x_1 = \mathsf{true}$ is forced by unit propagation. Then the formula simplifies to the unit clause (x_2). Unit propagation forces $x_2 = \mathsf{true}$, and the formula becomes true.

Great improvements to SAT solvers have been made beginning in the mid-1990s. GRASP [56] introduced a powerful form of conflict analysis that enables (1) non-chronological backtracking, allowing the solver to avoid considering unfruitful as-

```
function Solve($\phi$) {
    $\phi$ := Propagate($\phi$);
    if ($\phi$ = true) {return true;}
    if ($\phi$ = false) {return false;}
    x := (pick a variable in $\phi$);
    return (Solve($\phi$[x/true]) or Solve($\phi$[x/false]));
}
```

Figure 1.1: DPLL pseudocode; $\phi[x/c]$ denotes syntactic substitution of c for x in ϕ.

signments, and (2) learning of additional implied clauses, which enables the solver to discover more implied literals via unit propagation. When unit propagation forces a literal to be assigned a certain value, GRASP records the set of literals responsible. When a conflict is discovered (i.e., when the formula simplifies to false), GRASP uses this information to derive a new clause. The learned clause is logically redundant, but it enables unit propagation to be more effective. Another major breakthrough is the *two watched literals* scheme introduced by Chaff [48]. SAT solvers spend most of their time doing unit propagation, and the watched-literals scheme makes unit propagation significantly more efficient.

Chapter 2

Sequent Learning and Gate Literals in QBF

Most early QBF solvers have used conjunctive normal form (CNF). Although CNF works well for SAT solvers, it hinders the work of QBF solvers by impeding the ability to detect and learn from satisfying assignments. In fact, as we will see in Section 2.1, there is a family of formulas that are trivially satisfiable but whose CNF translations were experimentally found to require exponential time (in the number of variables) for existing CNF solvers.

Various techniques have been proposed for avoiding the drawbacks of a CNF encoding. Sabharwal et al. have developed a QBF modeling approach based a game-theoretic view of QBF [52]. Ansotegui et al. have investigated the use of *indicator variables* [1]. Lintao Zhang has investigated dual CNF-DNF representations in which a boolean formula is transformed into both a CNF formula (produced by the Tseitin transformation) and a DNF formula (produced by the Tseitin transformation on the negation of the formula) [60]. A prenex circuit-based DPLL solver with "don't care" reasoning and clause/cube learning has been developed by Goultiaeva, Iverson, and

Bacchus [33]. Non-clausal techniques using symbolic quantifier expansion (rather than DPLL) have been developed by Lonsing and Biere [43] and by Pigorsch and Scholl [50]. Non-clausal representations have also been investigated in the context of SAT solvers [22, 35, 57].

This chapter presents the *ghost variable* and *sequent learning* approach for non-CNF formulas that was presented in [40] and further developed in [39]. Experimental results are presented in the next chapter. A *dual propagation* technique similar to ghost variables was independently and contemporaneously developed in [31].

2.1 Tseitin Transformation Harmful in QBF

The Tseitin transformation [58] 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 formula:

$$\Phi_{\text{in}} = \exists e.\, \forall u.\, \underbrace{\underbrace{(e \wedge u)}_{g_1} \vee \neg e}_{g_2}$$

This formula is converted to:

$$\begin{aligned} \Phi'_{\text{in}} = \exists e.\, \forall u.\, \exists g_1.\, \exists g_2.\; & g_2 \wedge (g_1 \Leftrightarrow (e \wedge u)) \wedge \\ & (g_2 \Leftrightarrow (g_1 \vee \neg e)) \end{aligned} \tag{2.1}$$

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

$$\begin{aligned} & (g_1 \Leftrightarrow (e \wedge u)) \\ = {} & ((e \wedge u) \Rightarrow g_1) \;\wedge\; (g_1 \Rightarrow (e \wedge u)) \\ = {} & (\neg(e \wedge u) \vee g_1) \;\wedge\; (\neg g_1 \vee (e \wedge u)) \\ = {} & (\neg e \vee \neg u \vee g_1) \;\wedge\; (\neg g_1 \vee e) \wedge (\neg g_1 \vee u) \end{aligned}$$

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 2.1, if $e|\pi = \mathsf{false}$ and $g_1|\pi = \mathsf{true}$, then the existential player loses Φ'_{in} under π. This asymmetry can be harmful to QBF solvers. For example, consider the QBF

$$\forall X.\, \exists y.\; y \vee \underbrace{\psi(X)}_{g_1} \tag{2.2}$$

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 2.2 becomes:

$$\forall X.\, \exists y.\, \exists\{g_1, ..., g_n\}.\; (y \vee g_1) \wedge (\text{clauses equating gate variables})$$

Setting y to be true no longer immediately makes the matrix true. Instead, an assignment might need to include the gate variables and the universal variables X in order to satisfy the matrix. This makes it much harder to detect when the existential player has won. Experimental results [1, 60] indicate that purely CNF-based QBF solvers would, in the worst case, require time exponential in the number of variables in X to solve the CNF formula, even though the original problem (before translation to CNF) is trivial.

2.2 Ghost Variables

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 [60], which performed the Tseitin transformation twice: once on the input formula, and once on its negation.

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.[1]

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.

For clarity, it should be noted that, if an assignment π includes a ghost variable g^Q that represents a formula ϕ_g, then performing substitution under π **does not** necessarily substitute ϕ_g with the assigned value of g^Q. For example, if g_1^Q represents $x \wedge y$ and $\pi = \{g_1^Q\}$, then $(x \wedge y)|\pi$ evaluates to $(x \wedge y)$. Only variables that actually occur in a formula (as opposed to ghost variables that merely label parts of the formula) are substituted with their assigned values. In this thesis, we never consider formulas (other than single literals) in which ghost variables occur as actual variables (as opposed to mere labels).

[1]Let X_{last} be the innermost quantification block of the original formula. As an implementation optimization, ghost variables belonging to the owner of X_{last} can be considered to be at the same quantification level as X_{last} (rather than being downstream of this block).

Definition 2.1 (Consistent assignment to ghost literal). Given a quantifier type $Q \in \{\exists, \forall\}$ and an assignment π, we say a ghost literal g^Q is assigned **consistently** under π iff $g^Q|\pi =$ (the formula represented by g^Q)$|\pi$. We say a ghost literal g^Q is assigned **inconsistently** under π iff $g^Q|\pi = \neg$(the formula represented by g^Q)$|\pi$.

For example, if g_1^Q represents $x \wedge y$, then g_1^Q is assigned consistently under $\{\neg g_1^Q, \neg x\}$, while it assigned inconsistently under $\{\neg g_1^Q, x, y\}$. Under $\{\neg g_1^Q\}$, g_1^Q is not said to be either consistently or inconsistently assigned.

Definition 2.2 (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 both of the following conditions hold true:

- $\Phi|\pi = \begin{cases} \mathsf{true} & \text{if } Q \text{ is } \exists \\ \mathsf{false} & \text{if } Q \text{ is } \forall \end{cases}$
- 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 fails to win because $\Phi|\pi = \mathsf{false}$, and the universal player fails to win because the ghost variable $g^\forall$ is assigned inconsistently (since $g^\forall|\pi = \mathsf{true}$ but the formula represented by $g^\forall$ (i.e., the conjunction $e \wedge u$) evaluates to false).

Note that Definition 2.2 extends Definition 1.3 (on page 6) to cover assignments that have ghost variables. When the assignment π is restricted to being a total assignment to input variables (and no ghost variables), then the two definitions of "win" are equivalent.

Definition 2.3 (Losing under an assignment). Given a formula Φ and an assignment π, we define the phrase "Player Q **loses** Φ under π" recursively. We say "Player Q **loses** Φ under π" iff either:

1. Player Q does not win Φ under π and every input variable is assigned by π, or
2. there is an outermost unassigned input variable x such that either:
   (a) Player Q loses Φ under both $\pi \cup \{(x, \mathsf{true})\}$ and $\pi \cup \{(x, \mathsf{false})\}$, or
   (b) 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 $\Phi = \exists e.\, z \wedge e,$ where z is a free variable. Then:

- Player $\exists$ loses Φ under $\{\neg z, \neg e\}$, by subpart 1 of Definition 2.3.
- Player $\exists$ loses Φ under $\{\neg z\}$, by subpart 2(a) of Definition 2.3.
- Player $\forall$ loses Φ under $\{z\}$, by subpart 2(b) of Definition 2.3.
- Neither player can be said to lose Φ under the empty assignment.

Note that Definition 2.3 is consistent with Definition 1.3. Now let us make a few general observations about when a player loses under an arbitrary partial assignment.

Observation 2.1. If $\Phi|\pi = \mathsf{true}$, then Player $\forall$ loses Φ under π.

Observation 2.2. If $\Phi|\pi = \mathsf{false}$, then Player $\exists$ loses Φ under π.

Observation 2.3. If a ghost variable owned by Q in vars(π) is assigned inconsistently under π, then Player Q loses Φ under π.

Observation 2.4. If Player Q loses Φ under the assignment $\pi \cup \{\ell\}$, where ℓ is owned by the opponent of Q (and $\neg\ell \notin \pi$)[2], then Player Q loses Φ under π. For example, consider $\Phi = \exists e. \forall x.\ e \wedge x$. Player $\exists$ loses Φ under the assignment $\{\neg x\}$, so Observation 2.4 indicates that Player $\exists$ also loses Φ under the empty assignment.

[2]If $\neg\ell \in \pi$, then $\pi \cup \{\ell\}$ wouldn't be an assignment.

2.3 Game-State Sequents

In this section, we introduce *game-state learning*, a reformulation of clause/cube learning. For closed prenex instances, the game-state formulation is isomorphic to clause/cube learning; the differences are merely cosmetic. However, the game-state formulation is more convenient to extend to the non-prenex case and to formulas with free variables.

To motivate the definition of our sequents, we start by reviewing certain aspects of clause learning. Suppose the input formula Φ_{in} is a prenex CNF QBF whose first clause is $(e_1 \vee e_3 \vee u_4 \vee e_5)$. Under an assignment π, if all the literals in the clause are false, then clearly $\Phi_{\text{in}}|\pi$ is false. Moreover, if, under π, all the clause's existential literals are assigned false and none of the clause's universal literals are assigned true (i.e., they may either be assigned false or be unassigned), as depicted in Figure 2.1, then $\Phi_{\text{in}}|\pi$ is false, since the universal player can win by making all the universal literals in the clause false. (To show that a formula with a universal quantifier evaluates to false, we need only show that it evaluates to false under one assignment to the universally quantified variables.)

$$\exists e_1 \exists e_3 \forall u_4 \exists e_5.\ (\underbrace{e_1}_{\text{false}} \vee \underbrace{e_3}_{\text{false}} \vee \underbrace{u_4}_{\substack{\text{false or}\\\text{unassigned}}} \vee \underbrace{e_5}_{\text{false}}) \wedge \ \ldots$$

Figure 2.1: Universal player can win by making u_4 be false.

As shown in [61], when a QBF clause learning algorithm with long-distance resolution is applied to

$$\exists e_1 \exists e_3 \forall u_4 \exists e_5 \exists e_7.\ (e_1 \vee e_3 \vee u_4 \vee e_5) \wedge (e_1 \vee \neg e_3 \vee \neg u_4 \vee e_7) \wedge \ldots \tag{2.3}$$

it can yield the tautological learned clause $(e_1 \vee u_4 \vee \neg u_4 \vee e_5 \vee e_7)$. Note that this clause is tautological (i.e., always true) because it contains the complementary literals u_4 and $\neg u_4$. Although counter-intuitive, the algorithm in [61] allows this learned clause to be soundly interpreted in the same way as a non-tautological clause: Under an assignment π, if all the clause's existential literals are assigned false and none of the clause's universal literals are assigned true, then $\Phi_{\text{in}}|\pi$ is false.

Learned cubes are similar: Under an assignment π, if all the cube's universal literals are assigned true and none of the cube's existential literals are assigned false, then $\Phi_{\text{in}}|\pi$ is true. With game-state learning, we explicitly separate the "must be true" literals from the "may be either true or unassigned" literals. Instead of writing a cube $(e_1 \vee u_2 \vee \neg e_3)$, we will write a *game-state sequent*: $\langle \{u_2\}, \{e_1, \neg e_3\} \rangle \models (\forall \text{ loses } \Phi_{\text{in}})$. Now we will formally define game-state specifiers and sequents.

Definition 2.4 (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}, $\ell|\pi \neq \mathsf{false}$ (i.e., either $\ell|\pi = \mathsf{true}$ or $\mathsf{var}(\ell) \notin \text{vars}(\pi)$).

For example, $\langle \{u\}, \{e\} \rangle$ matches the assignments $\{u\}$ and $\{u, e\}$ (because both conditions in Definition 2.4 are satisfied), but does not match the empty assignment (because condition 1 fails) or $\{u, \neg e\}$ (because condition 2 fails).

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 2.5 (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 Φ be the following formula:

$$\forall u.\, \exists e.\, (e \vee \neg u) \wedge (u \vee \neg e) \wedge \overbrace{(x_1 \vee e)}^{g_3} \tag{2.4}$$

Note that sequent $\langle \{u\}, \{e\} \rangle \models (\forall \text{ loses } \Phi)$ holds true: in any assignment π that matches it, $\Phi|\pi = \mathsf{true}$. However, $\langle \{u\}, \varnothing \rangle \models (\forall \text{ loses } \Phi)$ does not hold true: it matches the assignment $\{u, \neg e\}$, under which Player $\forall$ does not lose Φ. Finally, $\langle \{g_3^\forall\}, \{e, \neg e\} \rangle \models (\forall \text{ loses } \Phi)$ holds true. Let us consider why Player $\forall$ loses Φ under the assignment $\{g_3^\forall\}$. The free variable x_1 is the outermost unassigned variable, so under Definition 2.3, 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 $\Phi|\{g_3^\forall, x_1\}$ 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 $\{g_3^\forall, \neg x_1, \neg e\}$.

With sequent learning, instead of having clause and cube databases, we maintain a sequent database. It turns out that whenever we learn a new game-state sequent for a closed prenex instance, the literals owned by the winner all go in L^{fut}, and the literals owned by the loser go in L^{now}. The relationship between game-state sequents and learned clauses/cubes (for prenex instances) is as follows.

A learned clause $(\ell_1 \vee ... \vee \ell_n)$ is equivalent to the sequent $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\exists \text{ loses } \Phi_{\text{in}})$ where L^{now} contains all the existential literals from $\{\neg \ell_1, ..., \neg \ell_n\}$, and L^{fut} contains all the universal literals from $\{\neg \ell_1, ..., \neg \ell_n\}$. Note that the literals from the clause occur negated in the sequent. For example, for the QBF in Equation 2.3, the clause $(e_1 \vee u_4 \vee \neg u_4 \vee e_5 \vee e_7)$ is equivalent to $\langle \{\neg e_1, \neg e_5, \neg e_7\}, \{u_4, \neg u_4\} \rangle \models (\exists \text{ loses } \Phi_{\text{in}})$.

Likewise, a learned cube $(\ell_1 \wedge ... \wedge \ell_n)$ is equivalent to $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\forall \text{ loses } \Phi_{\text{in}})$ where L^{now} contains all the universal literals from $\{\ell_1, ..., \ell_n\}$, and L^{fut} contains all the existential literals from $\{\ell_1, ..., \ell_n\}$. Unlike the case for clauses, the literals in a cube do not get negated for the corresponding sequent.

2.4 Algorithm

In this section, we consider only closed QBF. The top-level algorithm, shown in Figure 2.2, 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 (§ 2.4.1) 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 (which might be a ghost literal if the optimization in Footnote 1 on page 12 is used), adds it

```
 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 $\langle \varnothing,\ L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$) return $Q$;
10.    Backtrack();
11.    Propagate();
12.  }
```

Figure 2.2: Top-Level Algorithm. Details have been omitted for sake of clarity.

to π_{cur}, and performs boolean constraint propagation (BCP). BCP may add further literals to π_{cur}, as described in detail in § 2.4.4; 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* [56] 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 line 4.

In BCP, a literal owned by Q is forced by a sequent if the sequent indicates that Q needs to make ℓ true to avoid losing. Learned sequents prevent the solver from re-exploring parts of the 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).

The solver maintains a list of assigned literals in the order in which they were assigned; this list is referred to as the *trail* [20]. 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}.

2.4.1 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 2.2 (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 2.1). 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}: $\langle\{\neg g_{\text{in}}^{\exists}\}, \varnothing\rangle \models (\exists \text{ loses } \Phi_{\text{in}})$ and $\langle\{g_{\text{in}}^{\forall}\}, \varnothing\rangle \models (\forall \text{ loses } \Phi_{\text{in}})$.

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, $g_{\text{in}}^{\exists}|\pi$ must be equal to $g_{\text{in}}^{\forall}|\pi$, 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. $\langle\{g_*^Q, \neg g_i^Q\}, \varnothing\rangle \models (Q \text{ loses } \Phi_{\text{in}})$ for $i \in \{1, ..., m\}$

3. $\langle\{\neg g_*^Q, x_1, ..., x_n, g_1^Q, ..., g_m^Q\}, \varnothing\rangle \models (Q \text{ loses } \Phi_{\text{in}})$

Note that if π is an assignment such that (1) the ghost variable 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 (owned by Q), then π must match one of the above-listed initial sequents.

2.4.2 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}. For example, given the QBF $\Phi_{\text{in}} = \exists e. \forall u.\, \phi$, an initial sequent $\langle \{e, u\}, \varnothing \rangle \models (\exists \text{ loses } \Phi_{\text{in}})$ would be normalized to $\langle \{e\}, \{u\} \rangle \models (\exists \text{ loses } \Phi_{\text{in}})$. The soundness of this normalization is justified by the following inference rule:

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

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 2.4 on page 14).

2.4.3 Properties of Sequents in Database

After the initial sequents have been normalized (as described in § 2.4.2), the solver maintains the following invariant for all sequents in the sequent database, including sequents added to the database as a result of learning (§ 2.4.5). In any sequent of the form $\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (Q \text{ loses } \Phi_{\mathrm{in}})$:

1. Every literal in L^{now} is owned by Q.
2. Every literal in L^{fut} is owned by the opponent of Q.

2.4.4 Propagation

The `Propagate` procedure is similar to unit propagation for SAT solvers. Consider a sequent $\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (Q \text{ loses } \Phi_{\mathrm{in}})$ in the sequent database. If, under π_{cur},

1. there is exactly one unassigned literal ℓ in L^{now}, and
2. no literals in $L^{\mathrm{now}} \cup L^{\mathrm{fut}}$ 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}. The sequent that forced $\neg\ell$ is called the *antecedent* of $\neg\ell$. 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. Note that, in light of the property of sequents discussed in § 2.4.3, a sequent of the form $\langle L^{\mathrm{now}}, L^{\mathrm{fut}} \rangle \models (Q \text{ loses } \Phi_{\mathrm{in}})$ can force a literal only if the literal is owned by Q; it cannot force a literal owned by Q's opponent.

We employ a variant of the watched-literals rule designed for SAT solvers [48] and adapted for QBF solvers [24]. 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}.

2.4.5 Learning

In the top-level algorithm in Figure 2.2, 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 by Silva and Sakallah in [56] and adapted for QBF by Zhang and Malik in [61, 62] using long-distance resolution. It should be noted that not all modern QBF solvers use long-distance resolution. DepQBF [44] uses a slightly different learning technique developed in [28, 29] (and further developed in [45]) that avoids long-distance resolution.

We use inference rules shown in Figure 2.4 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 2.3, 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}, 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 § 2.4.4. We use the inference rules in Figure 2.4 to infer a new sequent from *seq* and

```
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[$r$], $r$);
    } until (seq.L^now $= \varnothing$ or has_good_UIP(seq));
    return seq;
}
```

Figure 2.3: Procedure for learning new sequents

Resolving on a literal r owned by Player Q:

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

Figure 2.4: Resolution-like inference rule.

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) [62], 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.

2.4.6 Justification of inference rules

The inference rule in Figure 2.4 is analogous to long-distance resolution [61] and its soundness can be proved by similar methods (e.g., [3]). We give a proof in Section 4.7.

Chapter 3

Non-Prenex Formulas

3.1 Introduction

Requiring that formulas be in prenex form simplifies the construction of a QBF solver. However, it can also be harmful, because it artificially limits the order in which variables can be branched on. For example, consider the following formula:

$$(\exists e_1.\forall u_2.\ \phi_1) \wedge (\exists e_3.\forall u_4.\ \phi_2) \tag{3.1}$$

Recall that a DPLL QBF solver can only pick an *outermost* unassigned variable to branch on. Once e_1 is assigned, both u_2 and e_3 are outermost unassigned variables in the above formula. However, if the formula is prenexed, then only one of these two can be an outermost unassigned variable. If then formula is prenexed as $(\exists e_1.\forall u_2.\exists e_3.\forall u_4.\ \phi_1 \wedge \phi_2)$, then u_2 would be an outermost variable (once e_1 is assigned). If it is prenexed as $(\exists e_1.\exists e_3.\forall u_2.\forall u_4.\ \phi_1 \wedge \phi_2)$, then e_3 would be an outermost variable. In general, in a prenex formula, it is impossible for both an existential variable and a universal variable to simultaneously be outermost unassigned variables.

How does restricting the branching order hurt QBF solvers? One way is that it frustrates decision heuristics such as VSIDS [48]. VSIDS tries to branch first on variables that have been directly relevant to recent conflicts (and therefore likely to relevant to conflicts in the near future). However, if the most relevant variables are not outermost, then the solver must first branch on the outermost variables, even if they are unlikely to be relevant to producing conflicts. After each such variable is branched on, BCP must be performed, wasting a great deal of computation time that could have been avoided if the solver were able to branch on only those variables that would actually be relevant.

Previous research in non-prenex DPLL-based QBF solving by Egly, Seidl, and Woltran [21] introduced a technique that employs dependency-directed (non-chronological) backtracking, but without learning or sharing of subformulas. Giunchiglia et al. have developed a technique for exploiting tree-like quantifier prefixes [26]. Lonsing and Biere have introduced dependency schemes that are more general than tree-like prefixes [44].

This chapter presents the non-prenex approach of [40], which allows learning to be applied to individual quantified subformulas instead of only the entire input formula. Most existing DPLL-based QBF solvers perform clause/cube learning. However, traditional clause/cube learning was designed for prenex QBF instances, and it is not optimal for (or even directly applicable to) non-prenex QBF instances. In Chapter 2, we reformulated clause/cube learning as *sequent* learning with *ghost variables*. In this chapter, we will extend it to the non-prenex case. Experimental results indicate that our approach can beat other state-of-the-art solvers on fixed-point computation instances of the type found in the `tipfixpoint` benchmark family.

3.2 Preliminaries

We label each conjunction and disjunction of the input QBF with a *gate variable*, as illustrated in Figure 3.1. In addition, quantified subformulas are also labeled with gate variables. Unlike quantifier-free subformulas, each labelled quantified subformula is only allowed to occur once in the input formula Φ_{in}; this follows from the requirement that each variable can be bound by a quantifier at most once in the input formula.

$$\exists e_{10} \Big[\underbrace{\big[\exists e_{11} \, \forall u_{21} \; \overbrace{(e_{10} \wedge e_{11} \wedge u_{21})}^{g_1} \big]}_{g_1'} \wedge \underbrace{\big[\forall u_{22} \, \exists e_{30} \; \overbrace{(e_{10} \wedge u_{22} \wedge e_{30})}^{g_2} \big]}_{g_2'} \Big]$$

Figure 3.1: Example QBF instance with gate labels.

The term "gate variable" arises from the circuit representation of a propositional formula, in which a gate variable labels a logic gate.

Let "Φ_{in}" denote the formula that the QBF solver is given as input. We impose the following restriction on Φ_{in}: Every variable in Φ_{in} must be quantified exactly once, and no variable may occur free (i.e., outside the scope of its quantifier). The variables that occur in Φ_{in} are said to be *input variables*. An *input assignment* is an assignment in which every assigned variable is an input variable (as opposed to a gate variable). We say that a gate literal g is *upstream* of an input literal y iff every variable that occurs in the subformula represented by g is upstream of y.

For non-prenex instances, we say that each quantifier-prefixed subformula (e.g., g_1' and g_2' in Figure 3.1) is a *subgame*. We say that two subgames are *independent* iff they have no unassigned variables in common. For example, in Figure 3.1, after e_{10} is assigned a value, the two subgames g_1' and g_2' become independent of each other, and the variables e_{11} and u_{22} become outermost unassigned variables.

3.3 Algorithm

An overview of the top-level solver algorithm is provided in Figure 3.2. Initially, the current assignment π_{cur} is empty. For non-prenex instances, if/when the input formula Φ_{in} (as simplified under the current assignment) becomes partitioned into independent subgames, we may temporarily target in on one such independent subgame and ignore the others; the subgame being targetted is referred to as *TargFmla*. On each iteration of the main loop, we first test to see if we know who wins *TargFmla* under the current assignment. There are two cases:

- If the winner of *TargFmla* is unknown, then we call `DecideLit`, which picks an unassigned input variable (from the first available quantifier block in the prefix of *TargFmla*) and assigns it a value in π_{cur}. If there are no more unassigned variables in the quantifier prefix of the current *TargFmla*, then we pick a new *TargFmla* from among the unassigned immediate subformulas of *TargFmla* and try again. After adding a new literal to π_{cur}, we call `Propagate` to perform boolean constraint propagation (BCP).

- If the winner is known, then we call `Learn` to learn a new game-state sequent, adding it to the database. If the new game-state sequent reveals which player wins Φ_{in} under the empty assignment, then we return with our final answer. Otherwise, we backtrack. We follow the well-known non-chronological backtracking technique, with the addition that we must also undo changes to *TargFmla* as appropriate. (That is, if we backtrack to the beginning of the k^{th} decision level, then we must restore *TargFmla* to the value that it held at the beginning of the k^{th} decision level. For this purpose, we maintain an array `UndoTarg` that maps each decision level to the value of *TargFmla* to be restored.) After backtracking, the newly-learned game-state sequent will force a literal, so we call `Propagate` to perform BCP.

```
initialize_sequent_database();
$\pi_{\text{cur}}$ := $\varnothing$;  $\mathit{TargFmla}$ := $\Phi_{\text{in}}$; Propagate();
while (true) {
  while ($\pi_{\text{cur}}$ doesn't match any database sequent for $\mathit{TargFmla}$) {
    DecideLit();   // Picks new $\mathit{TargFmla}$ if necessary.
    Propagate();
  }
  Learn();
  if (learned seq has form $\langle\varnothing,\ L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_{\text{in}})$) return $Q$;
  Backtrack();
  Propagate();
}
```

Figure 3.2: Overview of top-level solver algorithm.

3.3.1 Propagation

The `Propagate` procedure is similar to that of Section 2.4.4. Conceptually, we examine each learned game-state sequent *seq* of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_b)$ where Φ_b is a subformula of *TargFmla*. Let g_b be the gate variable that labels Φ_b. If, under the current assignment 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}, and
4. either (a) the quantifier of ℓ is within Φ_b, or

   (b) π_{cur} contains g_b^Q (if $Q = \exists$) or $\neg g_b^Q$ (if $Q = \forall$), then $\neg\ell$ is *forced* — it is added to the current assignment π_{cur}. The first three rules are the same as for the prenex case (Section 2.4.4). The 4th rule deals with subgames.

If all literals in L^{now} are true and none of the literals in L^{fut} are assigned `false`, then π_{cur} matches *seq*, so Q loses Φ_b under the current assignment. There are two subcases to consider:

1. If $\Phi_b = \textit{TargFmla}$, then we know who wins *TargFmla* under the current assignment, so we stop propagation and return to the top-level procedure (Figure 3.2).
2. If $\Phi_b \neq \textit{TargFmla}$, then the ghost variables $g_b^{\exists}$ and $g_b^{\forall}$ are forced to be false (if $Q{=}\exists$) or true (if $Q{=}\forall$).

3.3.2 Learning

In addition to the inference rule from the previous chapter, we have another inference rule that relates subgames with their parent games, shown in Figure 3.3.

3.4 Experimental Results

We implemented the ghost-variables technique from the previous chapter and the non-prenex learning technique from this chapter in a solver which we call *GhostQ*. We performed an experimental comparison to other solvers that were state-of-the-art at the time that the original version of GhostQ was finished.

We ran GhostQ on the non-CNF instances from QBFLIB on 2.66 GHz machine with a timeout of 300 seconds. For comparison we show the results for CirQit published in [33] (which were conducted on a 2.8 GHz machine with a timeout of 1200 seconds). As shown in Table 3.1, GhostQ performs better than CirQit on

Table 3.1: Comparison between GhostQ and CirQit.

Family	inst.	GhostQ		CirQit	
Seidl	*150*	**150**	(1606 s)	147	(2281 s)
assertion	*120*	**12**	(141 s)	3	(1 s)
consistency	*10*	0	(0 s)	0	(0 s)
counter	*45*	**40**	(370 s)	39	(1315 s)
dme	*11*	**11**	(13 s)	10	(15 s)
possibility	*120*	**14**	(274 s)	10	(1707 s)
ring	*20*	**18**	(28 s)	15	(60 s)
semaphore	*16*	**16**	(4 s)	16	(7 s)
Total	*492*	261	(2435 s)	240	(5389 s)

Table 3.2: Comparison between GhostQ and Qube.

Family	inst.	GhostQ		Qube	
bbox-01x	*450*	171	(133 s)	**341**	(1192 s)
bbox_design	*28*	19	(256 s)	**28**	(15 s)
bmc	*132*	43	(266 s)	**49**	(239 s)
k	*61*	**42**	(355 s)	13	(55 s)
s	*10*	10	(1 s)	10	(5 s)
tipdiam	*85*	**72**	(143 s)	60	(235 s)
tipfixpoint	*196*	**165**	(503 s)	100	(543 s)
sort_net	*53*	0	(0 s)	**19**	(176 s)
all other	*121*	9	(38 s)	**23**	(227 s)
Total	*1136*	531	(1695 s)	643	(2687 s)

Table 3.3: Comparison between GhostQ and Non-DPLL Solvers.

		Timeout 60 s			Timeout 600 s	
Family	inst.	GhostQ	Quantor	sKizzo	GhostQ	AIGsolve
bbox-01x	*450*	171	130	166	178	173
bbox_design	*28*	19	0	0	22	23
bmc	*132*	43	106	83	51	30
k	*61*	42	37	47	51	56
s	*10*	10	8	8	10	10
tipdiam	*85*	72	23	35	72	77
tipfixpoint	*196*	165	8	25	170	133
sort_net	*53*	0	27	1	0	0
all other	*121*	9	49	31	17	35
Total	*1136*	531	388	396	571	537

In Tables 1–2, we give the number of instances solved and the time needed to solve them. (Times shown do not include time spent trying to solve instances where the solver timed out.) In Table 3, we give the number of instances solved.

Resolving on a literal r owned by Player Q:

$$\frac{\begin{array}{l}\text{The quantifier type of } r \text{ is } Q\\ \text{The quantifier of } r \text{ is in } \Phi_b\\ \langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_b)\\ \langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_b)\\ r \text{ is not downstream of any } \ell \text{ such that } \ell \in L_1^{\text{fut}} \text{ and } \neg\ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})\\ r \text{ doesn't occur (positively or negated) in } (L_1^{\text{now}} \cup L_2^{\text{now}} \cup L_1^{\text{fut}} \cup L_2^{\text{fut}})\end{array}}{\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_b)}$$

Subgame Rule

$$\frac{\begin{array}{l}\text{The quantifier type of } r \text{ is } Q\\ \Phi_b \text{ is a subgame of } \Phi_a\\ \langle L_1^{\text{now}},\, L_1^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_b)\\ \langle L_2^{\text{now}} \cup \{\pm g_b\},\, L_2^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_a)\\ \text{The label of } \Phi_b \text{ is } g_b \text{, and ``} \pm g_b \text{'' denotes } g_b \text{ (if } Q = \forall \text{) or } \neg g_b \text{ (if } Q = \exists \text{)}\end{array}}{\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (Q \text{ loses } \Phi_a)}$$

Figure 3.3: Inference rules

every benchmark family except `consistency`. The `ring` and `semaphore` families consist of prenex instances. The other families are non-prenex, so our solver took advantage of its ability to perform non-prenex game-state learning. During testing of our solver, it was noted that non-prenex learning was especially helpful on the `dme` family. (The `dme` family instances were originally given in prenex form, but we pushed the quantifiers inward as a preprocessing step. The unprenexing time was about 0.8 seconds per instance and is included in our solver's total time shown in the table.)

We compared GhostQ to the state-of-the-art solvers Qube 6.6 [26], Quantor

3.0 [8], and sKizzo 0.8.2 [5]. We ran these solvers on the prenex CNF track of the QBFLIB QBFEVAL 2007 benchmarks [49] on a 2.66 GHz machine, with a time limit of 60 seconds and a memory limit of 1 GB. The results are shown in Tables 3.2 and 3.3. On these benchmarks, GhostQ outperformed the state-of-the-art CNF solvers on the `k`, `tipdiam`, and `tipfixpoint` families. We also show the results for AIGsolve published in [50], but these numbers are not directly comparable because they were obtained on a different machine and with a timeout of 600 s.

For the CNF benchmarks, we wrote a script to reverse-engineer the QDIMACS file to circuit form and convert it to our solver's input format. (This is similar to the technique in [50] and [32], but we also looked for "if-then-else" gates of the form $g = (x \,?\, y : z)$.) Of the four other solvers shown in Tables 3.2 and 3.3, Qube is the only other DPLL-based solver, so it is most similar to our solver. Our experimental results show that GhostQ does better than Qube on the `tipdiam` and `tipfixpoint` families (which concern diameter and fixpoint calculations for model checking problems on the TIP benchmarks) and on the `k` family.

The use of ghost literals can help GhostQ in two ways: (1) By treating the gate literals symmetrically instead of treating them as belonging to the existential player, we can more readily detect when the input formula is satisfied and we can learn more powerful cubes; (2) By using universal ghost literals, we have a more powerful propagation procedure for the universal input literals. (We did not perform unprenexing on any of the originally-CNF benchmarks, so our use of non-prenex learning doesn't improve performance here.) To further investigate, we turned off downward propagation of universal ghost literals; on most families the effect was negligible, but on `tipfixpoint` we solved only 149 instances instead of 165.

3.5 Future Work

Samulowitz and Bacchus [53] discovered a technique for dynamically partitioning a QBF problem into independent subproblems. It may be worthwhile to investigate whether a similar technique can allow allow dynamic unprenexing of QBF problems. This would allow our non-prenex technique to be applied to a much greater extent.

Chapter 4

Open QBF

4.1 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 chapter shows how DPLL-based closed-QBF solvers can be extended to solve QBFs with free variables. In Chapter 2, it was shown how clause/cube learning for DPLL-based QBF solvers can be reformulated in terms of *sequents* and extended to non-CNF formulas. This technique uses *ghost variables* to handle non-CNF formulas

```
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));
}
```

Figure 4.1: Naive algorithm. The notation "ite(x, ϕ_1, ϕ_2)" denotes a formula with an *if-then-else* construct that is logically equivalent to $(x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$.

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 4.1. 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, 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) [13], 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 chapter 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.

4.2 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**.

Quantifier Order. We extend the notion of *downsteam/upstream* to cover free variables. As in previous chapters, in a formula where the quantifier of a variable y occurs inside the scope of the quantifier of a variable x (e.g., $\forall x.\exists y.\,\phi$), and the quantifier type of x is different from the quantifier type of y, we say that y is **downstream** of x. 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.

4.2.1 Sequents with Free Variables

In Section 2.3, we introduced sequents that indicate if a player loses a formula Φ. Now, we will generalize sequents so that they can indicate that Φ 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 Φ and an assignment π that assigns all the input variables, we want the semantic evaluation $[\![\Phi]\!]_\pi$ to have the following properties:

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

Note that the above properties cannot be satisfied in a two-valued logic if both players lose Φ 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. Likewise, a disjunction of boolean values evaluates to true if any disjunct is true, and otherwise it evaluates to dontcare if any disjunct is dontcare. 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
dontcare	true	dontcare	true
dontcare	false	false	dontcare
dontcare	dontcare	dontcare	dontcare

For convenience in defining semantics with free variables, we assume that the formula is prepended with a dummy "quantifier" block for free variables. For example, the formula $(\exists e.\, e \wedge z)$ becomes $(\mathcal{F}z.\, \exists e.\, e \wedge z)$, where $\mathcal{F}$ denotes the dummy "quantifier" for free variables.

Definition 4.1 (Semantics with Free Variables)**.** Given an assignment π and a formula Φ, we define $[\![\Phi]\!]_\pi$ recursively. If Φ contains free variables unassigned by π, then $[\![\Phi]\!]_\pi$ is a formula in terms of these free variables.

- **Base case:** If π assigns all the input variables and a subset of the ghost variables, we define $[\![\Phi]\!]_\pi$ as follows:

$$[\![\Phi]\!]_\pi := \begin{cases} \mathsf{true} & \text{if Player } \exists \text{ wins } \Phi \text{ under } \pi \quad \text{(definition 2.2 on page 13)} \\ \mathsf{false} & \text{if Player } \forall \text{ wins } \Phi \text{ under } \pi \quad \text{(definition 2.2 on page 13)} \\ \mathsf{dontcare} & \text{if both players lose } \Phi \text{ under } \pi \quad \text{(definition 2.3 on page 13)} \end{cases}$$

- **Recursive case:** If π assigns only a proper subset of the input variables, we define $[\![\Phi]\!]_\pi$ as follows:

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

The notation "$x\ ?\ \phi_1 : \phi_2$" denotes a formula with an *if-then-else* ternary operator; the formula 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 [54].

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 4.1? 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 4.2. Given $x \in \{\mathsf{true}, \mathsf{false}, \mathsf{dontcare}\}$ and $y \in \{\mathsf{true}, \mathsf{false}, \mathsf{dontcare}\}$, we define the relation "$\overset{*}{\equiv}$" as follows: $x \overset{*}{\equiv} y$ is true iff either $x = y$ or $x = \mathsf{dontcare}$. Note that the relation is not symmetrical between x and y. (It is defined this way to simplify the proof of Lemma 4.8 on page 57.) As a truth table:

x	y	$x \overset{*}{\equiv} y$
true	true	true
true	false	false
false	true	false
false	false	true
true	dontcare	false
false	dontcare	false
dontcare	true	true
dontcare	false	true
dontcare	dontcare	true

Given two formulas ψ_1 and ψ_2, we write $\psi_1 \overset{*}{=} \psi_2$ iff the formula $\psi_1 \overset{*}{\equiv} \psi_2$ is valid (true under all assignments to its free variables). We write $\psi_1 \overset{*}{\neq} \psi_2$ iff $\psi_1 \overset{*}{=} \psi_2$ doesn't hold true. (Note that $\psi_1 = \psi_2$ implies $\psi_1 \overset{*}{=} \psi_2$, and $\psi_1 \overset{*}{\neq} \psi_2$ implies $\psi_1 \neq \psi_2$.)

Definition 4.3 (Repeat of Definition 2.4 on page 16)**.** 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}, $\ell|_\pi \neq \mathsf{false}$ (i.e., either $\ell|_\pi = \mathsf{true}$ or $\mathsf{var}(\ell) \notin \mathsf{vars}(\pi)$).

Definition 4.4 (Free Sequent)**.** Consider a QBF Φ and a propositional formula ψ that contains only variables free in Φ. The sequent "$\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi)$" means "$[\![\Phi]\!]_\pi \overset{*}{=} \psi|_\pi$ holds true for all assignments π that match $\langle L^{\text{now}}, L^{\text{fut}} \rangle$".

Definition 4.5. An assignment π is a *counterexample* to $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi)$ iff $[\![\Phi]\!]_\pi \overset{*}{\neq} \psi|_\pi$ and π matches $\langle L^{\text{now}}, L^{\text{fut}} \rangle$.

Remark. A sequent holds true iff there is no counterexample to it.

Example. Let $\Phi = \exists e.\, \overbrace{(e \wedge z)}^{g_1}$, and let *seq* be the sequent $\langle \varnothing, \varnothing \rangle \models (\Phi \Leftrightarrow z)$. The assignment $\pi_1 = \{\neg e\}$ is a counterexample to *seq* because $[\![\Phi]\!]_{\pi_1} = \mathsf{false}$ but $z|_{\pi_1} = z$. The assignment $\pi_2 = \{\neg e, g_1^{\forall}\}$ is not a counterexample, because $[\![\Phi]\!]_{\pi_2} = \mathsf{dontcare}$.

Remark. The sequent definitions in Definitions 4.4 and 2.5 are related as follows:

- "$\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\exists \text{ loses } \Phi)$" means the same as "$\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \mathsf{false})$".
- "$\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\forall \text{ loses } \Phi)$" means the same as "$\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \mathsf{true})$".

We treat a game sequent as interchangeable with the corresponding free sequent. Sequents of the form $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi)$ 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.

4.3 Algorithm

The top-level algorithm, shown in Figure 4.2, is the same as for closed QBF (Figure 2.2 on page 18) except for a minor change on line 9. As in closed QBF, when the current assignment matches a sequent in the database, the solver performs learning

```
 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.  }
```

Figure 4.2: Top-Level Algorithm. Details have been omitted for sake of clarity.

and backtracks to the earliest decision level at which the newly learned sequent will trigger a forced literal in BCP. In particular, free variables can be forced. The intuition behind forcing free variables is to prevent the solver from re-exploring parts of the 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).

In a (normalized) sequent for a closed prenex formula, all the loser's literals always belong in L^{now} and all of the winner's literals always belong in L^{fut}. That means that the L^{now} and L^{fut} components do not need to be explicitly separated. That is, it is possible to store them intermingled in a single undifferentiated set of variables, without explicitly storing which variables belong in which component, because this information can be unambiguously reconstructed. However, for formulas with free variables, it is necessary to explicitly record which literals belong in L^{now} and which in L^{fut}. If the ψ component of the sequent is a formula with free variables, then there is no "winner" or "loser" of the sequent, so the L^{now} and L^{fut} component cannot be reconstructed from a single undifferentiated set of variables.

4.3.1 Properties of Sequents in Database

After the initial sequents have been normalized (as described in § 2.4.2), the solver maintains the following invariants for all sequents in the sequent database, including sequents added to the database as a result of learning (§ 4.3.3):

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.

There is no corresponding property for sequents in which the ψ component is not a boolean constant.

4.3.2 Propagation

The `Propagate` procedure is similar to that of closed-QBF. 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} or ψ, and
4. all variables in ψ are unassigned,

then $\neg\ell$ is *forced* — it is added to the current assignment π_{cur}. The first two conditions above are the same as for closed QBF (Section 2.4.4). In the 3rd condition, we add the condition that ℓ be downstream of all unassigned literal in ψ. This, together with the new 4th condition, ensures that a quantified variable can be forced only by a sequent whose ψ component is a boolean constant. (It may be noted that, instead of making these two changes to the propagation rules, we could achieve the same effect by adding both polarities of every variable in ψ to L^{fut}. In fact, this is how the solver is actually implemented.)

It is instructive to consider how the propagation rules apply in light of the properties of sequents discussed in § 4.3.1:

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 the 4th condition of propagation fails. If there is a variable v in ψ that is left unassigned by π_{cur}, then the 3rd condition of propagation fails, since every quantified variable is downstream of free variable v.

We employ the same variant of the watched-literals rule as in Chapter 2. 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}.

4.3.3 Learning

The learning algorithm from § 2.4.5 (page 23) has been carefully formulated so that it applies unchanged to open QBF except for the addition of resolution-like rules for free variables. The new inference rules are shown in Figure 4.3. In Figure 4.4, we give an example of several successive applications of the resolution rules.

4.3.4 Justification of inference rules

Proofs of the inference rules in Figure 4.3 are provided in Section 4.7. Theorem 4.1 in Figure 4.3 is analogous to long-distance resolution [61] and can be proved by similar methods (e.g., [3]). Intuitively, if the current 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.

Theorem 4.1 (**Resolving on a literal r owned by Player Q, case 1**).

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

Theorem 4.2 (**Resolving on a literal r owned by Player Q, case 2**).

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

Theorem 4.3 (**Resolving on a variable r that is free in Φ_{in}**).

$$\begin{array}{l}
\text{Literal } r \text{ is free} \\
\langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_1) \\
\langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_2) \\
r \text{ doesn't occur (positively or negated) in } (L_1^{\text{now}} \cup L_2^{\text{now}} \cup L_1^{\text{fut}} \cup L_2^{\text{fut}}) \\
\hline
\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow (r \mathbin{?} \psi_1 : \psi_2))
\end{array}$$

Figure 4.3: Resolution-like inference rules.

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

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: $\langle\{\}, \{e_3, \neg e_3, \neg g_4^\forall, \neg g_5^\forall\}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$

Figure 4.4: Example: applications of the resolution rules.

If the current assignment π_{cur} matches the sequent in the conclusion of Theorem 4.2, 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, proven in Lemma 4.11 on page 60, free literals can be removed from L^{fut}). 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 Theorem 4.3, we don't need a condition about r not being downstream of other literals, since no free variable is downstream of any variable.

4.4 Experimental Results

We extended the existing closed-QBF solver GhostQ [40] to implement the techniques described in this paper. For comparison, we used the solvers and benchmarks from [4].[1] The benchmarks concern the automatic synthesis of a simple hardware load-balancer from a set of formal specifications. 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 was 2 GB.

There are three solvers from [4], 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 4.5 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.

One possible explanation of why GhostQ performs better than Learner-C is that Learner-C splits the open-QBF problem into multiple closed-QBF problems and uses a closed-QBF solver as an opaque black-box, so that information learned in one run of the QBF solver is not available in later runs. In contrast, GhostQ has a single

[1] The results do not exactly match the results reported in [4] 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.

learned-sequent database that is used for the entire duration of the solving process.

Figure 4.6 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., $(x \mathbin{?} y : \mathsf{false})$ is simplified to $x \wedge y$). For example, the size of the formula x is 1, the size of $\neg x$ is also 1, and the size of $x \wedge y$ is 3.

For additional experimental validation, we compared solver performance on the benchmarks from the 2010 Hardware Model Checking Competition (HWMCC'10). In particular, the solvers were tasked with finding the set of states reachable in exactly one step from the initial state. The machines, time limit, and memory limit were the same as for the load-balancer benchmarks above. GhostQ solved 570 instances (including 48 instances that Learner-C couldn't solve), while Learner-C solved 534 instances (including 12 instances that GhostQ couldn't solve). Figure 4.7 shows how many benchmarks were able to be solved within smaller per-instance time limits; this data is plotted in Figure 4.8.

We also compared GhostQ to the BDD method of NuSMV [14, 15], using the same HWMCC'10 task. NuSMV was instructed to compute one-step forward reachability by using the "`-df`" option and the specification "`SPEC AX FALSE`". These experiments were performed on a 2.66 GHz machine with a time limit of 120 seconds and a memory limit of 2 GB. GhostQ solved 564 instances (including 186 instances that NuSMV couldn't solve), while NuSMV solved 387 instances (including 9 instances that GhostQ couldn't solve).

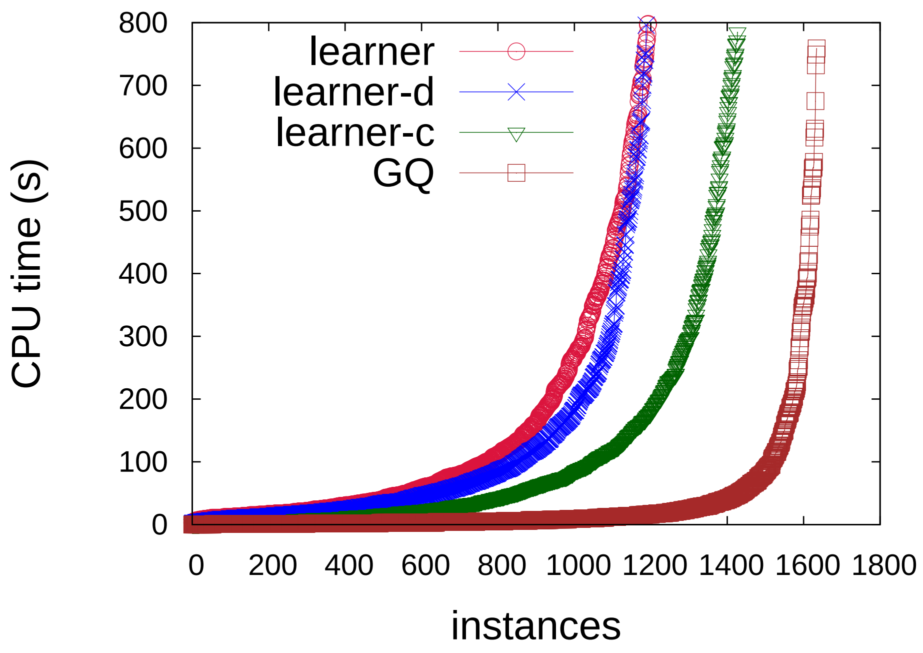

Figure 4.5: Time comparison on load-balancer benchmarks, not including instances solved by all solvers in less than 10 s.

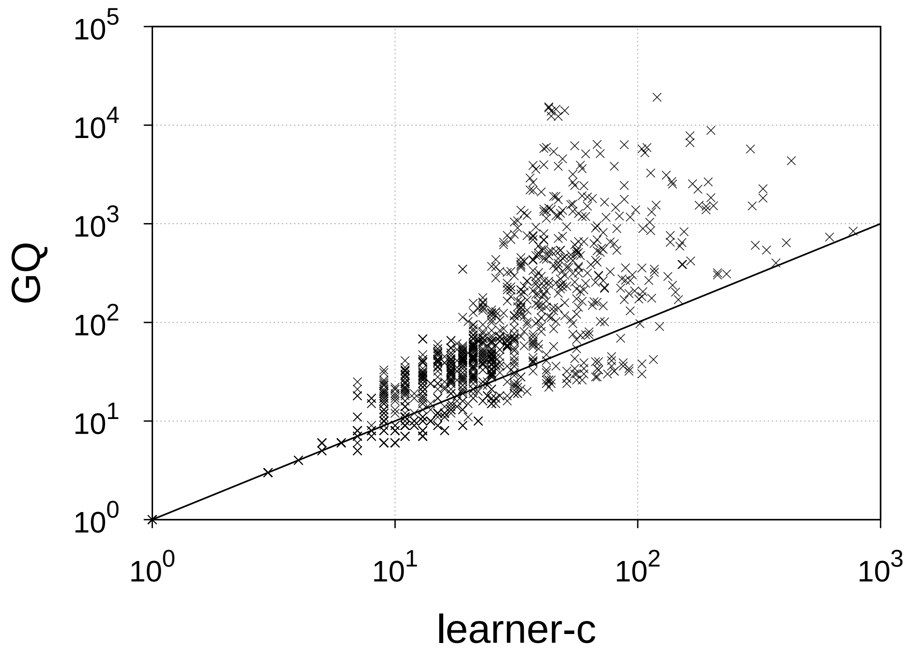

Figure 4.6: Comparison of output formula sizes on load-balancer benchmarks.

Time	Learner-C	GhostQ
3 sec	204	502
10 sec	268	547
30 sec	363	563
100 sec	458	568
300 sec	526	568
800 sec	534	570

Figure 4.7: Number of HWMCC'10 benchmarks solved

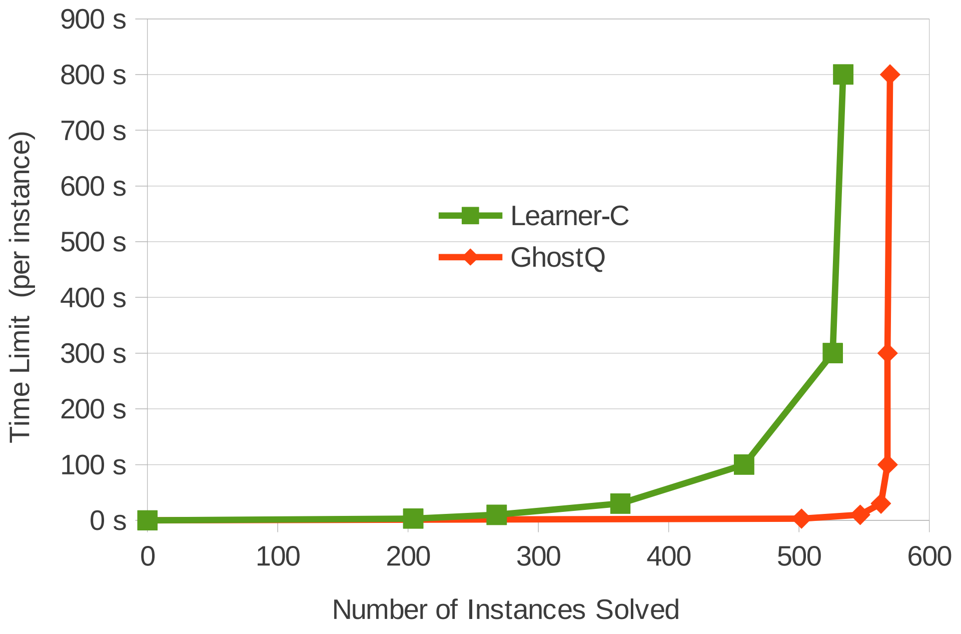

Figure 4.8: Cactus plot for HWMCC'10 benchmarks

4.5 Related Work

Ken McMillan [46] proposed a method to use SAT solvers to perform quantifier elimination on formulas of the form $\exists \vec{x}.\, \phi$, generating CNF output. This problem (i.e, given a formula $\exists \vec{x}.\, \phi$, return a logically equivalent quantifier-free CNF formula) has received attention recently. Brauer, King, and Kriener [12] designed an algorithm that combines model enumeration with prime implicant generation. Goldberg and Manolios [30] 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 [4], which uses computational learning to generate CNF, DNF, or CDNF formulas, and that of Benedetti and Mangassarian [7], which adapts sKizzo [6] for open QBF. The use of SAT solvers to build unordered BDDs [59] and OBDDs [34] has also been investigated.

4.6 Future Work

In practice, the output formulas produced by our solver 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. In many cases, converting our unordered BDD to an OBDD decreased the size of the formula. Although the BDDs produced by GhostQ are necessarily unordered due to unit propagation, as future work it may be desirable to investigate ways to reduce the size of the formula.

4.7 Soundness of Inference Rules

In this section, we shall prove that the inferences rules in Figure 4.3 on page 47 are sound. First, we need to prove several preliminary lemmas.

Lemma 4.1. Consider an assignment π to a formula Φ. If r is an outermost unassigned variable (i.e., r is not downstream of any variables unassigned by π), then

$$[\![\Phi]\!]_\pi = \begin{cases} [\![\Phi]\!]_{(\pi \cup \{r\})} \vee [\![\Phi]\!]_{(\pi \cup \{\neg r\})} & \text{if } r \text{ is existential} \\ [\![\Phi]\!]_{(\pi \cup \{r\})} \wedge [\![\Phi]\!]_{(\pi \cup \{\neg r\})} & \text{if } r \text{ is universal} \\ r \ ?\ [\![\Phi]\!]_{(\pi \cup \{r\})} \ :\ [\![\Phi]\!]_{(\pi \cup \{\neg r\})} & \text{if } r \text{ is free} \end{cases}$$

Proof. Follows from Definition 4.1 (on pages 40–41).

Example. Consider the formula $\Phi = \exists e_1 . \forall u_2 . \exists e_3 .\ (e_1 \wedge u_2) \vee e_3$ and the assignment $\pi = \{\neg u_2\}$. Then e_3 is outermost, so: $[\![\Phi]\!]_\pi = [\![\Phi]\!]_{(\pi \cup \{e_3\})} \vee [\![\Phi]\!]_{(\pi \cup \{\neg e_3\})}$.

Lemma 4.2. Consider a formula Φ and disjoint assignments π_1 and π_2, where π_2 contains only variables that are free in Φ. Then $(\Phi|_{\pi_1})|_{\pi_2} = \Phi|_{\pi_1 \cup \pi_2}$, and $([\![\Phi]\!]_{\pi_1})|_{\pi_2} = [\![\Phi]\!]_{\pi_1 \cup \pi_2}$.

Proof. By structural induction on Φ, using Definition 1.1 (on page 3) and Definition 4.1 (on pages 40–41).

Definition 4.6 (Free-Total)**.** In the context of a QBF Φ, an assignment is said to be *free-total* if it assigns all the free variables in Φ. Note that if π is free-total, then $[\![\Phi]\!]_\pi$ must evaluate to true, false, or dontcare (i.e., it cannot evaluate to a formula containing free variables).

Example. In the context of $\exists e . (z_1 \wedge z_2 \wedge e)$, the assignment $\{\neg z_1, z_2, e\}$ is free-total, whereas the assignment $\{\neg z_1, e\}$ is not free-total.

Lemma 4.3. Consider a sequent *seq* of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \psi)$ where L^{fut} has no free variables. Then there exists a counterexample (Definition 4.5 on page 42) to *seq* iff there exists a *free-total* counterexample to *seq*.

Proof of Lemma 4.3. One direction of the "iff" is trivial. For the other direction, we assume that there exists a counterexample π to *seq* and must show that there exists a free-total counterexample.

1. L^{fut} has no variables that are free in Φ. (assumption)
2. π is a counterexample to *seq* (assumption)
3. π matches $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ (step 2 and Definition 4.5 on page 42)
4. $[\![\Phi]\!]_\pi \overset{*}{\neq} \psi|\pi$ (step 2 and Definition 4.5)
5. Let V be the set of free variables that are not assigned by π.
6. There exists an assignment π_{ext} to V such that $([\![\Phi]\!]_\pi)|\pi_{\text{ext}} \overset{*}{\neq} (\psi|\pi)|\pi_{\text{ext}}$ (steps 5, 4, Definition 4.2)
7. $\pi \cup \pi_{\text{ext}}$ is free-total (steps 5, 6, Definition 4.6)
8. $[\![\Phi]\!]_{\pi\cup\pi_{\text{ext}}} \overset{*}{\neq} \psi|\pi\cup\pi_{\text{ext}}$ (step 6, Lemma 4.2)
9. $\pi \cup \pi_{\text{ext}}$ matches $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ (steps 1, 3, 6)
10. $\pi \cup \pi_{\text{ext}}$ is a counterexample to *seq* (steps 8, 9)

Lemma 4.4. Consider a QBF formula Φ and two disjoint assignments π and π_{ext}. If π_{ext} is a set of universal literals and $[\![\Phi]\!]_{(\pi \cup \pi_{\text{ext}})} \overset{*}{=} \mathsf{false}$, then $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{false}$.
If π_{ext} is a set of existential literals and $[\![\Phi]\!]_{(\pi \cup \pi_{\text{ext}})} \overset{*}{=} \mathsf{true}$, then $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{true}$.

Proof. By structural induction on Φ using Definition 4.1 (on pages 40–41).

Example. Consider the formula $\Phi = \forall u.\exists e.\ (e \wedge u) \vee (\neg e \wedge \neg u)$ and the assignments $\pi = \{u\}$ and $\pi_{\text{ext}} = \{e\}$. Then $[\![\Phi]\!]_{\pi \cup \{e\}} = \mathsf{true}$, so $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{true}$.

Lemma 4.5. Consider a a literal ℓ and an assignment π such that $\mathsf{var}(\ell) \notin \mathsf{vars}(\pi)$:

- If ℓ is existential and $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{false}$, then $[\![\Phi]\!]_{\pi \cup \{\ell\}} \overset{*}{=} \mathsf{false}$.
- If ℓ is universal and $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{true}$, then $[\![\Phi]\!]_{\pi \cup \{\ell\}} \overset{*}{=} \mathsf{true}$.

Proof. By structural induction on Φ using Definition 4.1 (on pages 40–41).

Lemma 4.6. Useful properties of "$\overset{*}{=}$" relation (Definition 4.2 on page 41):

1. If $y \overset{*}{=} z$, then $w \vee y \overset{*}{=} w \vee z$.
2. If $y \overset{*}{=} z$, then $w \wedge y \overset{*}{=} w \wedge z$.

Proof. Easily verified by truth tables. An abbreviated truth table for the first property is shown below. Entries where $y \overset{*}{=} z$ fails to hold are omitted. A dash indicates that the value is irrelevant.

w	y	z	$w \vee y$	$w \vee z$	$w \vee y \overset{*}{=} w \vee z$
true	—	—	true	true	true
false	true	true	true	true	true
false	false	false	false	false	true
false	dontcare	—	dontcare	—	true
dontcare	true	true	true	true	true
dontcare	false	false	dontcare	dontcare	true
dontcare	dontcare	—	dontcare	—	true

Lemma 4.7. Useful properties of "$\overset{*}{=}$" relation:

1. If $x = y_1 \vee y_2$, and $y_1 \overset{*}{=} z_1$, and $y_2 \overset{*}{=} z_2$, then $x \overset{*}{=} z_1 \vee z_2$.
2. If $x = y_1 \wedge y_2$, and $y_1 \overset{*}{=} z_1$, and $y_2 \overset{*}{=} z_2$, then $x \overset{*}{=} z_1 \wedge z_2$.
3. If $x = (\ell \ ? \ y_1 \ : \ y_2)$, and $y_1 \overset{*}{=} z_1$, and $y_2 \overset{*}{=} z_2$, then $x \overset{*}{=} (\ell \ ? \ z_1 \ : \ z_2)$.

Proof. Easily verified by truth tables or using Lemma 4.6.

Lemma 4.8. Suppose that $\langle L^{\text{now}} \cup \{r\},\, L^{\text{fut}} \rangle \models (P \text{ loses } \Phi)$ holds true, and $\langle L^{\text{now}},\, L^{\text{fut}} \rangle$ matches π_0, and r is owned by Player P, and r is upstream of all literals in L^{fut} not assigned by π_0, and r does not occur (positively or negatively) in $L^{\text{now}} \cup L^{\text{fut}}$ or π_0. Then, $[\![\Phi]\!]_{\pi_0} \overset{*}{=} [\![\Phi]\!]_{\pi_0 \cup \{\neg r\}}$.

Remark. This lemma is a formalization of the idea that adding a forced non-free literal (here, $\neg r$) to an assignment doesn't change the truth value that the formula evaluates to under the assignment (except in situations that we don't care about).

Proof. Follows immediately from Lemma 4.9 below (using $i = 1$ and $\pi_{\text{up}} = \varnothing$).

Lemma 4.9. Consider a QBF formula $\Phi = Q_1 x_1 ... Q_k r ... Q_n x_n .\, \phi$. (We assume that all free variables have corresponding dummy 'quantifiers', as discussed on page 40). Suppose that $\langle L^{\text{now}} \cup \{r\},\, L^{\text{fut}} \rangle \models (P \text{ loses } \Phi)$ holds true, and $\langle L^{\text{now}},\, L^{\text{fut}} \rangle$ matches π_0, and r is owned by Player P, and r is upstream of all literals in L^{fut} not assigned by π_0, and r does not occur (positively or negatively) in $L^{\text{now}} \cup L^{\text{fut}}$ or π_0. Further suppose that $i \in \{1, ..., k\}$ and π_{up} is an assignment to $\{x_1, ..., x_{i-1}\} \setminus \mathsf{vars}(\pi_0)$. Then $[\![\Phi]\!]_{\pi_0 \cup \pi_{\text{up}}} \overset{*}{=} [\![\Phi]\!]_{\pi_0 \cup \pi_{\text{up}} \cup \{\neg r\}}$. (We will prove this lemma for the case where Player P is existential; the proof is similar if P is universal.)

Proof. By induction on i, counting down from k to 1.

- Base case: $i = k$

  1. Let $\pi' = \pi_0 \cup \pi_{\text{up}}$
  2. π' assigns all variables upstream of r, so r is outermost under π'.
  3. $\pi' \cup \{r\}$ matches $\langle L^{\text{now}} \cup \{r\},\, L^{\text{fut}} \rangle$, and thus $[\![\Phi]\!]_{\pi' \cup \{r\}} \overset{*}{=} \mathsf{false}$.
  4. $[\![\Phi]\!]_{\pi'} = [\![\Phi]\!]_{\pi' \cup \{\neg r\}} \vee [\![\Phi]\!]_{\pi' \cup \{r\}}$ (Lemma 4.1)

     $\overset{*}{=} [\![\Phi]\!]_{\pi' \cup \{\neg r\}} \vee \mathsf{false}$ (step 3, Lemma 4.7)
  5. $[\![\Phi]\!]_{\pi'} \overset{*}{=} [\![\Phi]\!]_{\pi' \cup \{\neg r\}}$

- Inductive case: $i \in \{1, ..., k-1\}$

  1. π_{up} is an assignment to $\{x_1, ..., x_{i-1}\} \setminus \mathsf{vars}(\pi_0)$
  2. Let $\pi' = \pi_0 \cup \pi_{\text{up}}$
  3. Inductive hypothesis (IH): For every assignment π_{IH} to $\{x_1, ..., x_i\} \setminus \mathsf{vars}(\pi_0)$, $[\![\Phi]\!]_{\pi_0 \cup \pi_{\text{IH}}} \overset{*}{=} [\![\Phi]\!]_{\pi_0 \cup \pi_{\text{IH}} \cup \{\neg r\}}$
  4. If $x_i \in \mathsf{vars}(\pi_0)$, then $\{x_1, ..., x_{i-1}\} \setminus \mathsf{vars}(\pi_0) = \{x_1, ..., x_i\} \setminus \mathsf{vars}(\pi_0)$, so IH directly proves the desired property; go to step 10.
  5. For $\ell \in \{x_i, \neg x_i\}$, it follows from IH that $[\![\Phi]\!]_{\pi' \cup \{\ell\}} \overset{*}{=} [\![\Phi]\!]_{\pi' \cup \{\ell\} \cup \{\neg r\}}$.
  6. π' assigns all variables upstream of x_i, so x_i is outermost under π'.
  7. If x_i is existential:

$$\begin{aligned} [\![\Phi]\!]_{\pi'} &= [\![\Phi]\!]_{\pi' \cup \{x_i\}} \vee [\![\Phi]\!]_{\pi' \cup \{\neg x_i\}} && \text{(Lemma 4.1)} \\ &\overset{*}{=} [\![\Phi]\!]_{\pi' \cup \{x_i\} \cup \{\neg r\}} \vee [\![\Phi]\!]_{\pi' \cup \{\neg x_i\} \cup \{\neg r\}} && \text{(IH)} \\ &= [\![\Phi]\!]_{\pi' \cup \{\neg r\}} && \text{(Lemma 4.1)} \end{aligned}$$

  8. If x_i is universal:

$$\begin{aligned} [\![\Phi]\!]_{\pi'} &= [\![\Phi]\!]_{\pi' \cup \{x_i\}} \wedge [\![\Phi]\!]_{\pi' \cup \{\neg x_i\}} && \text{(Lemma 4.1)} \\ &\overset{*}{=} [\![\Phi]\!]_{\pi' \cup \{x_i\} \cup \{\neg r\}} \wedge [\![\Phi]\!]_{\pi' \cup \{\neg x_i\} \cup \{\neg r\}} && \text{(IH)} \\ &= [\![\Phi]\!]_{\pi' \cup \{\neg r\}} && \text{(Lemma 4.1)} \end{aligned}$$

  9. If x_i is free:

$$\begin{aligned} [\![\Phi]\!]_{\pi'} &= (x_i \; ? \; [\![\Phi]\!]_{(\pi \cup \{x_i\})} \; : \; [\![\Phi]\!]_{(\pi \cup \{\neg x_i\})}) && \text{(Lemma 4.1)} \\ &\overset{*}{=} (x_i \; ? \; [\![\Phi]\!]_{(\pi \cup \{x_i\} \cup \{\neg r\})} \; : \; [\![\Phi]\!]_{(\pi \cup \{\neg x_i\} \cup \{\neg r\})}) && \text{(IH)} \\ &= [\![\Phi]\!]_{\pi' \cup \{\neg r\}} && \text{(Lemma 4.1)} \end{aligned}$$

  10. Therefore, $[\![\Phi]\!]_{\pi'} \overset{*}{=} [\![\Phi]\!]_{\pi' \cup \{\neg r\}}$, which is what needed to be proved.

Lemma 4.10. The following inference rule is sound:

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

Intuition. Given the formula $\Phi = \forall u.\exists e.\,(u \wedge e) \vee (\neg u \wedge \neg e)$, consider the difference between the following two sequents:

1. $\langle \varnothing, \{e, \neg e\}\rangle \models (\Phi \Leftrightarrow \mathsf{true})$, which holds true, and
2. $\langle \varnothing, \varnothing\rangle \models (\Phi \Leftrightarrow \mathsf{true})$, which fails to hold true.

The first sequent means roughly "Player $\exists$ always wins if he plays e optimally". The second sequent means roughly "Player $\exists$ wins regardless of how he plays e". So Lemma 4.10 roughly says "If Player Q can't win even if he plays ℓ optimally, then he certainly doesn't win if he plays ℓ pessimally" (but this isn't an exact translation, because Lemma 4.10 deals only with one polarity of ℓ).

Proof. We consider the case where Q is existential; the universal case is similar.

1. $\langle L^{\text{now}}, L^{\text{fut}} \cup \{\ell\}\rangle \models (\exists \text{ loses } \Phi)$ (Premise 1)
2. Player $\exists$ owns ℓ, and $\neg\ell \notin L^{\text{now}}$ (Premise 2)
3. Consider an arbitrary assignment π that matches $\langle L^{\text{now}}, L^{\text{fut}}\rangle$.
4. If $\neg\ell \notin \pi$, then π matches the sequent in Premise 1, so $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{false}$; go to step 8.
5. Let $\pi' = \pi \setminus \{\neg\ell\}$, so that $\pi = \pi' \cup \{\neg\ell\}$.
6. $[\![\Phi]\!]_{\pi'} \overset{*}{=} \mathsf{false}$ (because π' matches the sequent in Premise 1)
7. $[\![\Phi]\!]_{\pi' \cup \{\neg\ell\}} \overset{*}{=} \mathsf{false}$ (Lemma 4.5 and step 6)
8. $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{false}$
9. $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\exists \text{ loses } \Phi)$ (steps 3 and 8)

Lemma 4.11. The following inference rule is sound:

$$\frac{\begin{array}{l}\langle L^{\text{now}},\, L^{\text{fut}} \cup \{\ell\}\rangle \models (\Phi \Leftrightarrow \psi) \\ \mathsf{var}(\ell) \text{ is free in } \Phi, \text{ and } \neg\ell \notin L^{\text{now}}\end{array}}{\langle L^{\text{now}},\, L^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \psi)}$$

Proof.

1. $\langle L^{\text{now}}, L^{\text{fut}} \cup \{\ell\}\rangle \models (\Phi \Leftrightarrow \psi)$ (Premise 1)
2. $\mathsf{var}(\ell)$ is free in Φ, and $\neg\ell \notin L^{\text{now}}$ (Premise 2)
3. Consider an arbitrary assignment π that matches $\langle L^{\text{now}}, L^{\text{fut}}\rangle$.
4. If $\neg\ell \notin \pi$, then π matches the sequent in Premise 1, so $[\![\Phi]\!]_\pi \overset{*}{=} \psi|\pi$; go to step 8.
5. Let $\pi' = \pi \setminus \{\neg\ell\}$, so that $\pi = \pi' \cup \{\neg\ell\}$.
6. π' matches the sequent in Premise 1, so $[\![\Phi]\!]_{\pi'} \overset{*}{=} \psi|\pi'$
7. $$\begin{aligned} [\![\Phi]\!]_\pi &= [\![\Phi]\!]_{\pi' \cup \{\neg\ell\}} && \text{(step 5)} \\ &= ([\![\Phi]\!]_{\pi'})|\{\neg\ell\} && \text{(Lemma 4.2)} \\ &\overset{*}{=} (\psi|\pi')|\{\neg\ell\} && \text{(step 6)} \\ &= \psi|(\pi' \cup \{\neg\ell\}) && \text{(Lemma 4.2)} \\ &= \psi|\pi && \text{(step 5)} \end{aligned}$$
8. $[\![\Phi]\!]_\pi \overset{*}{=} \psi|\pi$
9. $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \psi)$ (steps 3 and 8)

Definition 4.7 (Vacuous game-state, vacuously true sequent)**.** A game-state $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ is *vacuous* iff no assignment matches it. Equivalently, it is vacuous iff there exists a literal $\ell \in L^{\text{now}}$ such that $\neg\ell \in (L^{\text{now}} \cup L^{\text{fut}})$. A sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \psi)$ is *vacuously true* iff the game-state $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ is vacuous. (It is easy to confirm that a *vacuously true* sequent is true under Definition 4.4 on page 42).

Observation 4.1. If assignment π matches a game-state $\langle L^{\text{now}} \cup L^{\text{now}}_{+},\, L^{\text{fut}} \cup L^{\text{fut}}_{+}\rangle$, then, *a fortiori*, π also matches $\langle L^{\text{now}}, L^{\text{fut}}\rangle$.

Lemma 4.12. If the sequent $\langle L^{\text{now}},\, L^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \psi)$ holds true, then the sequent $\langle L^{\text{now}} \cup L^{\text{now}}_{+},\, L^{\text{fut}} \cup L^{\text{fut}}_{+}\rangle \models (\Phi \Leftrightarrow \psi)$ also holds true.

Proof. Follows from Observation 4.1.

Lemma 4.13. In the inference rules in Figure 4.3 on page 47, if either $\langle L^{\text{now}}_1, L^{\text{fut}}_1\rangle$ or $\langle L^{\text{now}}_2, L^{\text{fut}}_2\rangle$ is vacuous, then the sequent in the conclusion of the inference rule is vacuously true.

Proof. Follows from Observation 4.1.

Lemma 4.14. If $\neg\ell \in L^{\text{now}}$, then the sequent $\langle L^{\text{now}},\, L^{\text{fut}} \cup \{\ell\}\rangle \models (\Phi \Leftrightarrow \psi)$ is vacuous.

Proof. Follows from Definition 4.7.

Proof of Theorem 4.1 (page 47). We consider the case where r is owned by Player $\exists$; the case for Player $\forall$ is similar. We prove a nominally weaker version by adding the premise that all literals in L_1^{fut} are owned by Player $\forall$. (Any literals in L_1^{fut} not owned by Player $\forall$ can be removed by Lemmas 4.10 and 4.11 (unless the premise sequents are vacuously true, in which case the conclusion sequent follows by Lemma 4.13), and they can be added back to the sequent in the conclusion of the inference rule by Lemma 4.12.)

1. The quantifier type of r in Φ is existential. (premise)
2. $\langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \mathsf{false})$ (premise)
3. $\langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \mathsf{false})$ (premise)
4. r is not downstream of any ℓ such that $\ell \in L_1^{\text{fut}}$ and $\neg\ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})$ (premise)
5. r doesn't occur (positively or negated) in $(L_1^{\text{now}} \cup L_2^{\text{now}} \cup L_1^{\text{fut}} \cup L_2^{\text{fut}})$ (premise)
6. All literals in L_1^{fut} are owned by Player $\forall$. (premise)
7. Let π be an arbitrary assignment matching $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\ L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle$
8. If r is assigned by π, then $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{false}$ follows from step 2 or 3; go to step 15.
9. Let $L_{1\,\text{up}}^{\text{fut}}$ be the subset of literals in L_1^{fut} that are upstream of r
10. Let $\pi^+ = \pi \cup L_{1\,\text{up}}^{\text{fut}}$
11. π^+ matches $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\ L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle$ (steps 4, 7-10)
12. π^+ matches $\langle L_1^{\text{now}},\ L_1^{\text{fut}}\rangle$ (step 11)
13. $\pi^+ \cup \{\neg r\}$ matches $\langle L_2^{\text{now}} \cup \{\neg r\},\ L_2^{\text{fut}}\rangle$ (steps 11, 5)
14. $[\![\Phi]\!]_{\pi^+} \overset{*}{=} [\![\Phi]\!]_{\pi^+ \cup \{\neg r\}}$ (Lemma 4.8; steps 12, 2, 5, 9–10)

    $\overset{*}{=} \mathsf{false}$ (steps 3, 13)
15. $[\![\Phi]\!]_\pi \overset{*}{=} \mathsf{false}$ (Lemma 4.4; steps 6, 10, 14)
16. $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (\Phi \Leftrightarrow \mathsf{false})$ (steps 15 and 7)

Proof of Theorem 4.2. We consider the case where r is owned by Player $\exists$; the case for Player $\forall$ is similar. We prove a nominally weaker version by adding an extra premise restricting what types of literals may occur in L_1^{fut} and L_2^{fut}. (Any disallowed literals in L_1^{fut} or L_2^{fut} can be removed by Lemmas 4.10 and 4.11 (unless the premise sequents are vacuously true, in which case the conclusion sequent follows by Lemma 4.13), and they can be added back to the sequent in the conclusion of the inference rule by Lemma 4.12.)

1. The quantifier type of r in Φ is existential. (premise)
2. $\langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \mathsf{false})$ (premise)
3. $\langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi)$ (premise)
4. r is not downstream of any ℓ such that $\ell \in L_1^{\text{fut}}$ and $\neg\ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})$ (premise)
5. r doesn't occur (positively or negated) in $(L_1^{\text{now}} \cup L_2^{\text{now}} \cup L_1^{\text{fut}} \cup L_2^{\text{fut}})$ (premise)
6. All literals in L_1^{fut} are owned by Player $\forall$. No free vars occur in L_2^{fut}. (premise)
7. Let π be arbitrary free-total assignment matching $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{\neg r\}\rangle$
8. Since π is free-total, it follows that $\psi|\pi$ evaluates to true or to false (i.e., it doesn't evaluate to a formula with free variables).
9. If $\psi|\pi = \mathsf{true}$, then:
   (a) $[\![\Phi]\!]_{\pi \cup \{\neg r\}} \overset{*}{=} \psi|(\pi \cup \{\neg r\})$ (step 3)
   $= \psi|\pi$ (since ψ contains only free vars)
   $= \mathsf{true}$
   (b) $[\![\Phi]\!]_{\pi} \overset{*}{=} \mathsf{true}$ (Lemma 4.4)
10. If $\psi|\pi = \mathsf{false}$, then:
    (a) $[\![\Phi]\!]_{\pi} \overset{*}{=} \mathsf{false}$ (following the same logic as for Theorem 4.1)
11. $[\![\Phi]\!]_{\pi} \overset{*}{=} \psi|\pi$ (steps 9, 10)
12. $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{\neg r\}\rangle \models (\Phi \Leftrightarrow \psi)$ (steps 11 and 7)

Proof of Theorem 4.3.

1. Literal r is free (premise)
2. $\langle L_1^{\text{now}} \cup \{r\},\, L_1^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi_1)$ (premise)
3. $\langle L_2^{\text{now}} \cup \{\neg r\},\, L_2^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi_2)$ (premise)
4. Let π be an arbitrary assignment matching $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\ L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{r, \neg r\} \rangle$
5. r is not assigned by π (step 4; definition 2.4 (page 16))
6. $[\![\Phi]\!]_{\pi \,\cup\, \{r\}} \overset{*}{=} \psi_1|\pi$ (steps 4 and 2; definition 4.4)
7. $[\![\Phi]\!]_{\pi \,\cup\, \{\neg r\}} \overset{*}{=} \psi_2|\pi$ (steps 4 and 3; definition 4.4)
8. $[\![\Phi]\!]_{\pi} = r \ ?\ [\![\Phi]\!]_{\pi \,\cup\, \{r\}} \ :\ [\![\Phi]\!]_{\pi \,\cup\, \{\neg r\}}$ (step 5, Lemma 4.1)
9. $[\![\Phi]\!]_{\pi} \overset{*}{=} r \ ?\ \psi_1|\pi \ :\ \psi_2|\pi$ (steps 6, 7, 8)
10. $[\![\Phi]\!]_{\pi} \overset{*}{=} (r \ ?\ \psi_1 \ :\ \psi_2)|\pi$ (steps 9 and 5)
11. $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{r, \neg r\} \rangle \models (\Phi \Leftrightarrow (r \ ?\ \psi_1 : \psi_2))$ (steps 10 and 4)
12. $\langle L_1^{\text{now}} \cup L_2^{\text{now}},\, L_1^{\text{fut}} \cup L_2^{\text{fut}} \rangle \models (\Phi \Leftrightarrow (r \ ?\ \psi_1 : \psi_2))$ (Lemma 4.11 and step 11)

Chapter 5

Counterexample-Guided Abstraction Refinement (CEGAR) in QBF

5.1 Introduction

A number of approaches have been proposed for QBF, including (Q)DPLL (e.g., [27]), expansion [2, 8, 43], and Skolemization [5]. This chapter presents a new approach by M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke [36]. It employs Counterexample-Guided Abstraction Refinement (CEGAR) [17] to gradually expand the input formula. The CEGAR approach differs from existing expansion-based solvers (such as Quantor [8] and Nenofex [43]) in how the expansion is performed.

For a quantifier block of n variables, existing expansion-based solvers perform up to n expansions (one for each variable), and the formula grows exponentially with the number of expansions performed (in the worst case), for a total expansion factor of $O(2^n)$. The CEGAR approach has the same worst-case expansion factor of $O(2^n)$, but it gets there by performing up to 2^n partial expansions (one for each possible assignment to all n variables), with the formula growing only linearly with the number of partial expansions performed. In practice, often only a relatively small number of partial expansions are needed, allowing the CEGAR approach to solve instances on which existing expansion-based solvers run out of memory.

We present two different ways in which CEGAR expansion can be used. In the first approach, we use a recursive algorithm that is driven by CEGAR expansion. In the second approach, CEGAR is used as an additional learning strategy in an existing DPLL-based QBF solver.

5.2 Preliminaries

1. We write "$\bar{Q}$" to denote "$\forall$" (if Q is "$\exists$") or "$\exists$" (if Q is "$\forall$").
2. We write "$moves(X)$" to denote the set of assignments to the variables X.
3. A *winning move* for X in a closed QBF $QX.\Phi$ is an assignment $\tau \in moves(X)$ such that $\Phi|_\tau$ is true (if Q is $\exists$) or $\Phi|_\tau$ is false (if Q is $\forall$).
4. We use the notation "$\texttt{SAT}(\phi)$" to represent a call to a SAT solver on a propositional formula ϕ. The SAT solver returns a satisfying assignment for ϕ, if such exists, and returns NULL otherwise.
5. A formula is in *strictly alternating* prenex form iff no two adjacent quantifier blocks have the same quantifier type (existential or universal). In this chapter, we assume the input formula is closed and in strictly alternating prenex form.

5.3 Recursive CEGAR-based Algorithm

In previous work, a CEGAR approach was used to solve quantified boolean formulas with 2 levels of quantifiers [37]. Here we present a recursive algorithm that applies to formulas with any number of quantifier alternations. The algorithm attempts to find a winning move for the outermost quantification block or prove that none exists. In doing so, it makes recursive calls to subproblems with fewer quantifier blocks. In the base case, there is only a single quantifier block, which is handled by a SAT solver.

The basic idea is as follows. Consider a QBF instance $\exists X.\, \forall Y.\, \Phi$. Note that $\forall Y.\, \Phi$ is logically equivalent to the full expansion over all possible moves for Y:

$$(\forall Y.\, \Phi) \;\Leftrightarrow\; \bigwedge_{\mu \in \mathit{moves}(Y)} \left(\Phi|\mu\right)$$

Of course, if there are many variables in Y, then performing a full expansion is not practical. However, it turns out that, in many instances that arise in practice, only a small number of assignments (moves) need to be considered. Accordingly, we use a *partial expansion* defined below.

Definition 5.1 (Partial Expansion)**.** Let ω be a subset of $\mathit{moves}(Y)$.

The *partial expansion* of $\exists Y.\ \Phi$ over ω is the formula $\bigvee_{\mu\in\omega} \Phi|\mu$.

The *partial expansion* of $\forall Y.\ \Phi$ over ω is the formula $\bigwedge_{\mu\in\omega} \Phi|\mu$.

For example, if $\Psi = \exists x_1, x_2.\ \forall z_1, z_2.\ (x_1 \vee z_1) \wedge (x_2 \vee z_2) \wedge (\neg x_1 \vee \neg z_2)$, and $\omega = \big\{\{x_1, x_2\}, \{\neg x_1, x_2\}\big\}$, then the partial expansion of Ψ over ω is $\neg z_2 \vee z_1$, since $\Psi|\{x_1, x_2\} = \neg z_2$ and $\Psi|\{\neg x_1, x_2\} = z_1$.

Observation 5.1. A partial expansion of $QY.\Phi$ can be considered an *abstraction* of $QY.\Phi$. It represents a handicap on player Q in the sense that player Q is allowed to play only those moves in ω rather than any move in $\mathit{moves}(Y)$. Thus, if Q wins a partial expansion of $QY.\Phi$, then Q also wins $QY.\Phi$:

- $\big(\big(\bigwedge_{\mu\in\omega} \Phi|\mu\big) \Leftrightarrow \mathsf{false}\big) \;\Rightarrow\; \big(\big(\bigwedge_{\mu\in \mathit{moves}(Y)} \Phi|\mu\big) \Leftrightarrow \mathsf{false}\big)$
- $\big(\big(\bigvee_{\mu\in\omega} \Phi|\mu\big) \Leftrightarrow \mathsf{true}\big) \;\Rightarrow\; \big(\big(\bigvee_{\mu\in \mathit{moves}(Y)} \Phi|\mu\big) \Leftrightarrow \mathsf{true}\big)$

Definition 5.2 (Candidate)**.** Let Φ be a formula $QX.\,\bar{Q}Y.\,\Psi$, and let α be an arbitrary partial expansion of $\bar{Q}Y.\,\Psi$. Given an assignment $\pi \in moves(X)$, we say "π is a *candidate* with respect to α" iff Q wins $\alpha|\pi$.

Example. Let $\Phi = \exists e_1, e_2.\, \forall u.\, (e_1 \wedge u) \vee (e_2 \wedge \neg u)$, and let α be the partial expansion over $\omega = \big\{\{\neg u\}\big\}$. Then $\alpha = (e_1 \wedge \mathsf{false}) \vee (e_2 \wedge \mathsf{true}) = e_2$, so $\{\neg e_1, e_2\}$ is a candidate w.r.t. α, but $\{e_1, \neg e_2\}$ is not.

Lemma 5.1. Let Φ be a formula $QX.\,\bar{Q}Y.\,\Psi$, and let α be an arbitrary partial expansion of $\bar{Q}Y.\,\Psi$. If there is no candidate with respect to α, then $\bar{Q}$ wins Φ.

Proof. Follows from Observation 5.1.

Definition 5.3 (Counterexample)**.** Let Φ be a formula $QX.\,\bar{Q}Y.\,\Psi$, and let cand be a candidate with respect to an arbitrary partial expansion of $\bar{Q}Y.\,\Psi$. Given an assignment $\pi \in moves(Y)$, we say "π is a *counterexample* to cand with respect to Φ" iff Player $\bar{Q}$ wins $(\Phi|_{\mathit{cand}})|\pi$. We may omit saying "with respect to Φ" if the formula Φ is understood from context.

Example. If $\Phi = \exists e_1, e_2.\, \forall u.\, (e_1 \wedge u) \vee (e_2 \wedge \neg u)$ and $\mathit{cand} = \{\neg e_1, e_2\}$, then $\{u\}$ is a counterexample to cand, but $\{\neg u\}$ is not.

Lemma 5.2. Let Φ be a formula $QX.\,\bar{Q}Y.\,\Psi$, and let $\mathit{cand} \in moves(X)$ be a candidate with respect to an arbitrary partial expansion of $\bar{Q}Y.\,\Psi$. If there is no counterexample to cand with respect to Φ, then Q wins Φ.

Proof. Since Player Q wins $(\Phi|_{\mathit{cand}})|\pi$ for all $\pi \in moves(Y)$, it follows that Player Q wins $\Phi|_{\mathit{cand}}$. Furthermore, since Player Q owns all the literals in cand, it follows that Player Q wins Φ.

Lemma 5.3. Let Φ be a formula $QX.\, \bar{Q}Y.\, \Psi$,

let ω be a subset of $moves(Y)$,

let α be the partial expansion of $\bar{Q}Y.\, \Psi$ over ω,

let $\mathit{cand} \in moves(X)$ be a candidate with respect to α,

let $\mathit{cex} \in moves(Y)$ be a counterexample to cand, and

let α' be the partial expansion of $\bar{Q}Y.\, \Psi$ over $\omega \cup \{\mathit{cex}\}$.

Then cand is not a candidate w.r.t. α'.

Proof. We consider the case where Q is existential:

$$\begin{aligned}
\alpha'|\mathit{cand} &= \big(\textstyle\bigwedge_{\mu \in \omega \cup \{\mathit{cex}\}} \Phi|\mu\big)|\mathit{cand} \\
&= \Big(\big(\textstyle\bigwedge_{\mu \in \omega} \Phi|\mu\big) \wedge (\Phi|\mathit{cex})\Big)|\mathit{cand} \\
&= \big(\textstyle\bigwedge_{\mu \in \omega} \Phi|\mu\big)|\mathit{cand} \wedge (\Phi|\mathit{cex})|\mathit{cand} \\
&= \big(\textstyle\bigwedge_{\mu \in \omega} \Phi|\mu\big)|\mathit{cand} \wedge (\Phi|\mathit{cand})|\mathit{cex} \\
&= \big(\textstyle\bigwedge_{\mu \in \omega} \Phi|\mu\big)|\mathit{cand} \wedge \mathsf{false} \\
&= \mathsf{false}
\end{aligned}$$

To solve $QX.\, \bar{Q}Y.\, \Phi$, we start with a coarse abstraction and gradually refine it until we find an answer. At a high level, the algorithm is as follows:

1. Initialize ω such that $\omega \subseteq moves(Y)$. (Specifically, we use $\omega = \varnothing$.)
2. Let α be the partial expansion of $\bar{Q}Y.\, \Phi$ over ω.
3. Try to find $\mathit{cand} \in moves(X)$ such that Player Q wins $\alpha|\mathit{cand}$.
4. If no such assignment exists, we're done: Player $\bar{Q}$ wins $QX.\bar{Q}Y.\, \Phi$.
5. Try to find $\mathit{cex} \in moves(Y)$ such that Player $\bar{Q}$ wins $(\Phi|\mathit{cand})|\mathit{cex}$.
6. If no such assignment exists, we're done: Player Q wins $QX.\bar{Q}Y.\, \Phi$.
7. Let $\omega := \omega \cup \{\mathit{cex}\}$ and go back to Step 2.

This algorithm is fleshed out in Algorithm 1, and illustrated in Figure 5.1.

Algorithm 1: Basic recursive CEGAR algorithm for QBF

```
Function Solve ($QX.\bar{Q}Y.\,\Phi$)
/* Return value: A winning assignment for $X$ if there is one, NULL otherwise.
begin
    if ($Y = \varnothing$) then return ($Q{=}\exists$ ? SAT($\Phi$) : SAT($\neg\Phi$))
    $\omega := \varnothing$
    while true do
        $\alpha := \begin{cases} \exists X.\bigwedge_{\mu\in\omega}\Phi|\mu & \text{if } \bar{Q}=\forall \\ \forall X.\bigvee_{\mu\in\omega}\Phi|\mu & \text{if } \bar{Q}=\exists \end{cases}$
        cand := Solve(Prenex($\alpha$))                    // find a candidate solution
        if cand = NULL then return NULL
        Remove from cand any variables not in $X$
        cex := Solve($\bar{Q}Y.\,\Phi|$cand)                 // find a counterexample
        if cex = NULL then return cand
        $\omega := \omega \cup \{$cex$\}$
    end
end
```

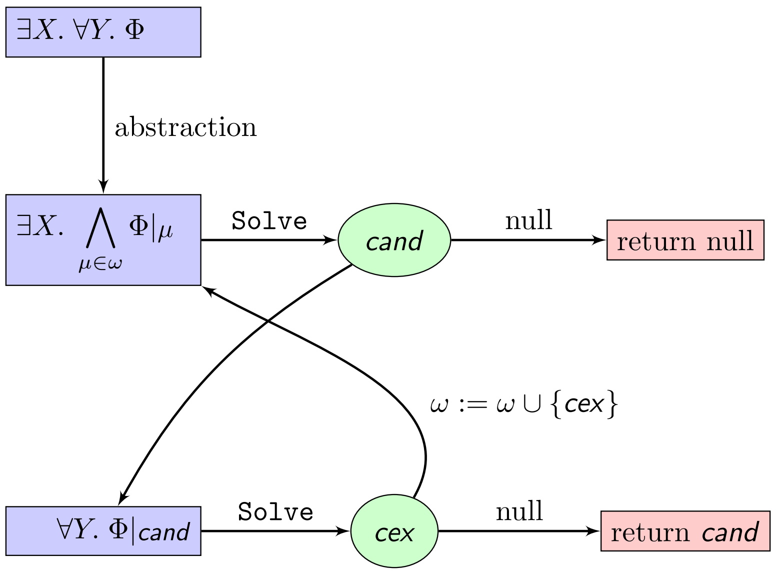
$\exists X.\,\forall Y.\,\Phi$
abstraction
$\exists X.\bigwedge_{\mu\in\omega}\Phi|\mu$
Solve
cand
null
return null
$\omega := \omega \cup \{cex\}$
$\forall Y.\,\Phi|cand$
Solve
cex
null
return cand

Figure 5.1: Flowchart for CEGAR Algorithm

5.3.1 Improving Recursive CEGAR-based Algorithm

Note that Algorithm 1 requires prenexing α. This is harmful because it loses information about dependencies among variables. Algorithm 2 avoids this prenexing by using the concept of a *multi-game*:

Definition 5.4 (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. 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 $QX.\ \Phi_i$.

Algorithm 2: RAReQS: Recursive Abstraction Refinement QBF Solver

```
Function RAReQS ($QX.\{\Phi_1, \ldots, \Phi_n\}$)
/* Return value: A winning assignment for $X$ if there is one, NULL otherwise.
begin
    if ($\Phi_i$ have no quantifiers) then return $Q{=}\exists$ ? SAT($\bigwedge_i \Phi_i$) : SAT($\neg(\bigvee_i \Phi)$)
    $\alpha := QX.\ \{\}$
    while true do
        cand := RAReQS($\alpha$)                                // find a candidate solution
        if cand = NULL then return NULL
        Remove from cand any variables not in $X$.
        for $i := 1$ to $n$ do  $cex_i$ := RAReQS($\Phi_i|_{cand}$)      // find a counterexample
        if $cex_i$ = NULL for all $i \in \{1..n\}$ then return cand
        let $l \in \{1..n\}$ be such that  $cex_l \neq$ NULL
        $\alpha :=$ Refine($\alpha$, $\Phi_l$, $cex_l$)
    end
end
Refine is defined as follows:
    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 $X_1$, and $\Psi'$ is $\Psi$ with $X_1$ replaced by $X_1'$
    Refine$\big(QX.\{\Psi_1, \ldots, \Psi_n\},\ \bar{Q}Y.\,\psi,\ \mu\big) \ =\ QX.\{\Psi_1, \ldots, \Psi_n, \psi|\mu\}$
    where $\psi$ is a propositional formula (where no duplicates are needed)
```

5.4 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.

Notation. Given an assignment π, let $\mathit{prop}(\pi)$ be the assignment produced by adding literals that would be forced in boolean constraint propagation (BCP) using the solver's sequent database. For example, if the input formula contains a subformula $(x \vee y)$ labelled by g, then $\mathit{prop}(\{x\})$ would contain x, $g^{\exists}$, and $g^{\forall}$.

To illustrate the basic idea of the CEGAR-in-DPLL technique, let Φ be a QBF of the form $\forall X.\, \exists Y.\ \phi$. Let π_{cand} be an assignment to the variables in X such that $\mathit{prop}(\pi_{\text{cand}})$ doesn't match any sequent in the solver's sequent database. Let π_{cex} be a counterexample to π_{cand}; i.e., let π_{cex} be an assignment to the variables in Y such that $\phi|\pi_{\text{cand}} \cup \pi_{\text{cex}} = \mathsf{true}$. The goal of the CEGAR learning is to produce a set of sequents such that, if these sequents are added to the sequent database, then for every assignment $\pi'_{\text{cand}} \in \mathit{moves}(X)$ to which π_{cex} is a counterexample, some sequent in the database would match $\mathit{prop}(\pi'_{\text{cand}})$. This goal is accomplished as follows:

1. Substitute the assignment π_{cex} into ϕ, yielding the formula $\phi|\pi_{\text{cex}}$.
2. Introduce ghost variables for any gates in $\phi|\pi_{\text{cex}}$ that are not already labelled by ghost variables. Add sequents that relate these ghost variables to the gates that they represent, as in Section 2.4.1 on page 20.
3. Let $g_*^{\forall}$ be the universal ghost variable that labels the formula $\phi|\pi_{\text{cex}}$.
4. Learn the new sequent $\langle \{g_*^{\forall}\}, \pi_{\text{cex}} \rangle \models (\Phi \Leftrightarrow \mathsf{true})$.

Consider an arbitrary assignment $\pi'_{\text{cand}} \in \mathit{moves}(X)$ to which π_{cex} is a counterexample. Then $\phi|\pi_{\text{cand}} \cup \pi_{\text{cex}} = \mathsf{true}$. To prove that $\mathit{prop}(\pi'_{\text{cand}})$ matches $\langle \{g_*^{\forall}\}, \pi_{\text{cex}} \rangle$, we must prove (1) $g_*^\forall \in \mathit{prop}(\pi'_{\text{cand}})$ and (2) $\mathit{prop}(\pi'_{\text{cand}})$ does not contain the negation of any literal in π_{cex}:

1. Since all the variables in the formula labelled by $g_*^\forall$ are assigned by π'_{cand}, it follows that either the variable $g_*^\forall$ or its negation must be a forced literal under π'_{cand}. And since $g_*^\forall$ labels $\phi|\pi_{\text{cex}}$, and $(\phi|\pi_{\text{cex}})|\pi_{\text{cand}} = \mathsf{true}$, it follows that the positive literal $g_*^\forall$ is forced, i.e., $g_*^\forall \in \mathit{prop}(\pi'_{\text{cand}})$.
2. A literal ℓ is forced under an assignment π only if the owner of ℓ is doomed to lose under $\pi \cup \{\neg\ell\}$. Since Player $\exists$ owns π_{cex} and wins under $\pi'_{\text{cand}} \cup \pi_{\text{cex}}$, it follows that no literals from π_{cex} appear negated in $\mathit{prop}(\pi'_{\text{cand}})$.

For example, consider the formula $\Phi = \forall X.\exists Y.\phi$ where ϕ is:

$$\phi = (\neg u_1 \vee \neg e_3) \wedge (\neg u_2 \vee \neg e_4) \wedge (u_1 \vee e_3) \wedge (u_2 \vee e_4) \tag{5.1}$$

Suppose that $\pi_{\text{cand}} = \{u_1, u_2\}$ and $\pi_{\text{cex}} = \{\neg e_3, \neg e_4\}$. Then $\phi|\pi_{\text{cex}} = u_1 \wedge u_2$. Let $g_6^\forall$ be the universal ghost variable for $u_1 \wedge u_2$. The solver learns the sequent $\langle\{g_6^\forall\}, \{\neg e_3, \neg e_4\}\rangle \models (\Phi \Leftrightarrow \mathsf{true})$, as well as sequents relating $g_6^\forall$ to the gate which it represents.

To add CEGAR learning to the DPLL-based solver GhostQ, we insert a call to a new CEGAR-learning procedure after standard DPLL learning, as shown in Algorithm 3. As shown in Algorithm 3, CEGAR learning is performed only if the last decision literal in π_{cur} 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.)

Consider a QBF $Q_1 Z_1 .... Q_n Z_n . \phi$. Suppose that the last decision literal belongs to the winner and is in the block Z_i. Then CEGAR learning would proceed as follows:

Algorithm 3: DPLL Algorithm with CEGAR Learning

```
initialize_sequent_database();
$\pi_{\text{cur}}$ := $\varnothing$; Propagate();
while (true) {
  while ($\pi_{\text{cur}}$ doesn't match any database sequent) {
    DecideLit();
    Propagate();
  }
  Learn();
  if (learned seq has form $\langle \varnothing,\ L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$) return $\psi$;
  if (last decision literal is owned by winner) {cegar_learn($\phi_{\text{in}}$);}
  Backtrack();
  Propagate();
}
```

1. Let π_c be a total assignment to the variables in Z_i. If a variable in Z_i is assigned by π_{cur}, it should have the same value in π_c; if it doesn't appear in π_{cur}, it can be assigned an arbitrary value in π_c.
2. Let *guard* be a subset of π_{cur} that assigns a subset of variables in $Z_1, ..., Z_{i-2}$.
3. Let $Z'_{i+1}, ..., Z'_n$ be fresh variables corresponding to $Z_{i+1}, ..., Z_n$, respectively.
4. Let ϕ' be the result of substituting the assignment π_{cex} into ϕ and replacing all occurrences of variables in $Z_{i+1}, ..., Z_n$ with $Z'_{i+1}, ..., Z'_n$, respectively.
5. Introduce ghost variables for any gates in ϕ' not already labelled by ghost vars. Add sequents that relate these ghost variables to the gates that they represent.
6. Let Q^* be $\bar{Q}_i$. Let $g_*^{Q^*}$ be the ghost variable that labels the formula ϕ' (or the negation of the ghost variable, if Q^* is $\exists$) .
7. Learn the new sequent $\langle guard \cup \{g_*^{Q^*}\},\ \pi_{\text{cex}} \rangle \models (Q^* \text{ loses } \Phi)$.

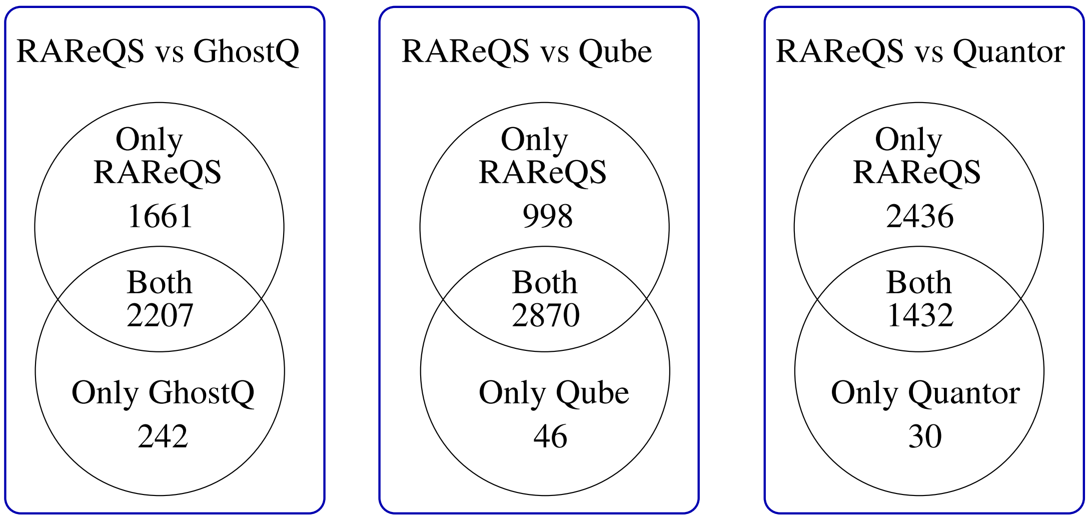

Table 5.1: Number of instances solved by RAReQS but not by a competing solver, and *vice versa*. For example, there were 1661 instances that RAReQS solved but GhostQ didn't, and 242 instances that GhostQ solved but RAReQS didn't.

5.5 Experimental Results

A prototype of the CEGAR algorithm is implemented in a solver called RAReQS (Recursive Abstraction Refinement QBF Solver). For the underlying SAT solver, `minisat 2.2` [20] is used. We compared RAReQS to other solvers on the the *formal verification* and *planning* suites of QBF-LIB [51]. Several large and hard families were sampled with 150 files (`terminator`, `tipfixpoint`, `Strategic Companies`). The solvers QuBE7.2 [25], 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. The experimental results were obtained on an Intel Xeon 5160 3GHz. The time limit was set to 800 seconds and the memory limit to 2GB.

All the instances were preprocessed by the preprocessor bloqqer [11] 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

Family	Lev.	RAReQS	GQ	GQ-C	QuBE	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 5.2: Number of instances in each benchmark family solved within 800 seconds by each solver. “Lev” indicates the number of quantifier blocks (min–max) in the family, post-bloqqer.

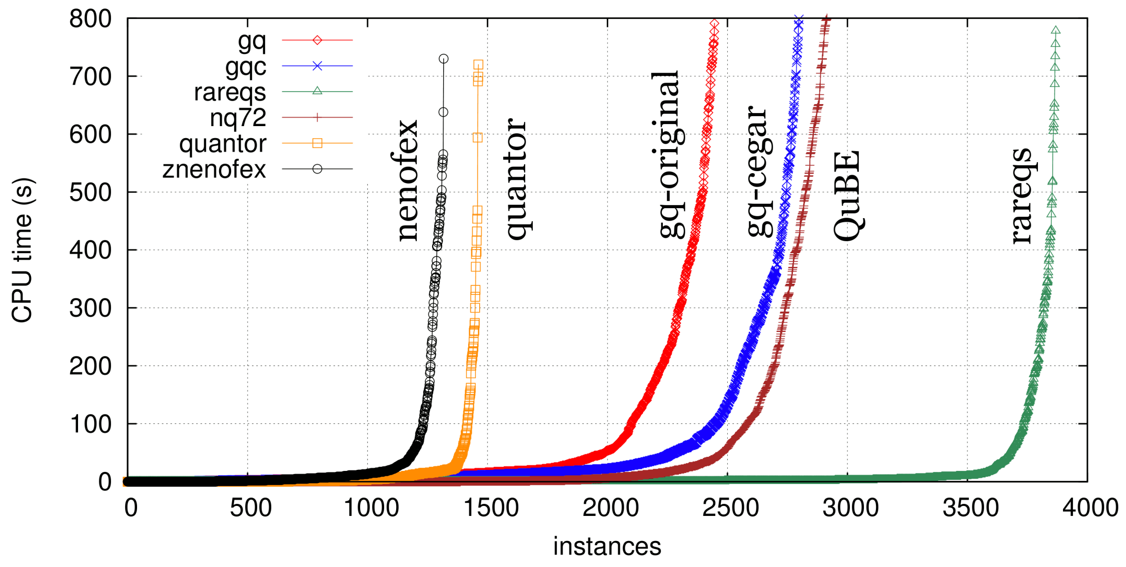

Figure 5.2: Cactus plot of the overall results

the family was considered in its unpreprocessed form.

Unlike the other solvers, GhostQ's input format is not clause-based (QDIMACS); instead, 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 5.2; number of solved instances in each family are shown in Table 5.2; a comparison of RAReQS with other solvers is presented in Table 5.1.

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 5.1 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. These results indicate that, for at least the types of practical problems represented by these benchmarks, only a relatively small number of assignments (moves) are necessary to obtain a partial expansion that is logically equivalent to the full expansion.

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.

5.6 Future Work

Although the recursive CEGAR algorithm works well for many practical problems, there are some simple problems on which it takes exponential time. For example, consider the following class of formulas, parameterized by n:

$$\forall X.\exists Y.\ (x_1 \Leftrightarrow y_1) \wedge ... \wedge (x_n \Leftrightarrow y_n) \tag{5.2}$$

where $X = \langle x_1, ..., x_n \rangle$ and $Y = \langle y_1, ..., y_n \rangle$. For this class of problems, RAReQS needs to consider all 2^n partial expansions, so it takes time exponential in n. To deal with this sort of problem, we can consider a generalization of the CEGAR technique. Instead of only considering assignments of variables to boolean constants, we can consider assignments of variables to formulas in terms of upstream variables. This would enable Formula 5.2 above to be solved with only a single counterexample (namely, by an assignment that maps y_i to x_i, for $i \in \{1, ..., n\}$).

Specifically, we would modify Algorithm 1 so that instead of adding *cex* to ω, we create a modified counterexample *cex*′ and add *cex*′ to ω. The modified counterexample cex' will map each variable of Y to a formula that may contain variables from X (but no other variables). We write "$\Phi|_{\mathit{cex}'}$" to denote the same notion of substitution as in Definition 1.1 on page 3, except that variables are mapped to formulas instead of just to boolean constants. The modified counterexample cex' should be equivalent to cex under cand, i.e., the following should hold true:

$$\left(y_i|_{\mathit{cex}'}\right)|_{\mathit{cand}} = \left(y_i|_{\mathit{cex}}\right)|_{\mathit{cand}}$$

Of course, $y_i|_{\mathit{cex}}$ is a boolean constant, so $y_i|_{\mathit{cex}} = \left(y_i|_{\mathit{cex}}\right)|_{\mathit{cand}}$. By Lemma 4.2 on page 54, $\left(y_i|_{\mathit{cex}'}\right)|_{\mathit{cand}} = y_i|_{\mathit{cex}' \cup \mathit{cand}}$

The tricky part about the above generalization of the CEGAR algorithm is devising a heuristic for generating cex' from cex. All other things being equal, it is best to generate a modified counterexample cex' that strengthens the abstraction the most.

Chapter 6

Conclusion

In this thesis, we first introduced the concepts of *ghost variables* and *game-state sequents*, which improve upon existing CNF propagation and learning techniques. The use of ghost variables eliminates the asymmetry between the two quantifier types, strengthening propagation and learning in cases where the existential player wins. The technique is especially helpful in cases where the formula has multiple alternations of conjunctions and disjunctions and the asymmetric algorithm needs to consider many assignments under which the existential player wins. If the formula is naturally written in CNF, then the symmetric technique provides no benefit. If the existential player rarely wins when employing the asymmetric algorithm, then there is little opportunity for improvement when using the symmetric algorithm.

Using game-state sequents, we have reformulated the techniques for conflict and satisfaction analysis, BCP, non-chronological backtracking, and learning. In all cases, we give a unified presentation which is applicable to both the existential and universal players, instead of using separate terminology and notation for the two players. Experiments show that our techniques work particularly well on certain benchmarks related to formal verification.

We have extended the concept of *game-state sequents* for non-prenex formulas to allow learning to be applied to quantified subformulas. This way, when we learn the truth value of a quantified subformula under an assignment, we effectively cache it so that we don't need to recompute it again. For future work, it may be worthwhile to investigate whether the ideas of dynamic partitioning [53] can be extended to allow dynamic unprenexing.

We have 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). Instead of only being able to express when the original formula evaluates to True or to False, our sequent can express when the original quantified formula evaluates to propositional formulas that contain free variables. Our extended solver GhostQ produces unordered BDDs, which have several favorable properties [18]. 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 ways to reduce the size of the output formula.

Finally, we have presented two novel techniques for using CEGAR in solving QBF problems. First, a CEGAR-driven solver RAReQS has been presented which builds an abstraction of the given formula by constructing partial expansions. This technique works well on formulas for which relatively few partial expansions are needed. For formulas that require many partial expansions, such as Equation 5.2 (page 78), the technique performs poorly. Experimentally, RAReQS has been shown to work very well on a wide variety of benchmarks. Second, CEGAR has been incorporated into a DPLL solver as an additional learning technique. While this technique does not take advantage of the full range of CEGAR learning exploited by RAReQS, it still provides a more powerful learning technique than standard clause/cube learning, and experimentally it has been shown helpful for a variety of benchmarks.

Appendix A

Encodings of Problems in QBF

A.1 Encoding Bounded Model Checking in QBF

Bounded Model Checking (BMC) [9] is the method used by most industrial-strength model checkers today. Given a finite state-transition system, a temporal logic property, and a bound k, BMC generates a propositional formula that is satisfiable if and only if the property can be disproved by a counterexample of length k. This propositional formula is then fed to a *Boolean satisfiability* (SAT) solver. If no counterexample of length k is found, then we look for longer counterexamples by incrementing the bound k. For safety properties (i.e., checking whether a "bad" state is unreachable), it can be shown that we only need to check counterexamples whose length is smaller than the *diameter* of the system — the smallest number of transitions to reach all reachable states. Alternatively, BMC can be used for bug catching (rather than full verification) by simply running it up to a given counterexample length or for a given amount of time. BMC has been observed to have several advantages over symbolic model checking with BDDs in typical industrial experience:

1. BMC finds counterexamples faster than BDD-based approaches.
2. BMC finds counterexamples of minimal length.

3. BMC uses much less memory than BDD-based approaches.
4. BMC does not require the user to select a variable ordering and does not need to perform costly dynamic reordering.

In BMC, the states of the model are represented as vectors of Booleans. For example, in a hardware circuit, the state of each flip-flop would be usually encoded as a single Boolean variable. A state transition system is encoded as follows:

- the set of initial states is specified by a propositional formula $I(s)$ that holds true iff s is an initial state;
- the transition relation is specified by a propositional formula $R(s, s')$ that holds true iff there exists a transition from s to s';
- for each atomic proposition p, there is a propositional formula $p(s)$ that holds true iff p is true in state s.

Definition A.1. A sequence of states $(s_0, ..., s_k)$ is a **valid path prefix** iff:

1. $I(s_0)$ holds true (s_0 is an initial state); and
2. $\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$ holds true (for all $i < k$ there exists a transition $s_i \rightarrow s_{i+1}$)

For simplicity, we first describe BMC for LTL safety properties of the form $\mathbf{G}\,p$, where p is an atomic proposition.

A.1.1 Safety Properties

The property $\mathbf{G}\,p$ asserts that p holds true in all reachable states (remember that LTL formulas are implicitly quantified by an outer $\mathbf{A}$ path operator.) We wish to determine whether there exists a counterexample whose length is no larger than a fixed bound k. In other words, we wish to determine whether there exists a valid path prefix $(s_0, ..., s_k)$ in which p fails for some state s_i, with $i \leq k$. Thus, we have that a sequence $(s_0, ..., s_k)$ is a counterexample to $\mathbf{G}\,p$ iff the following formula is

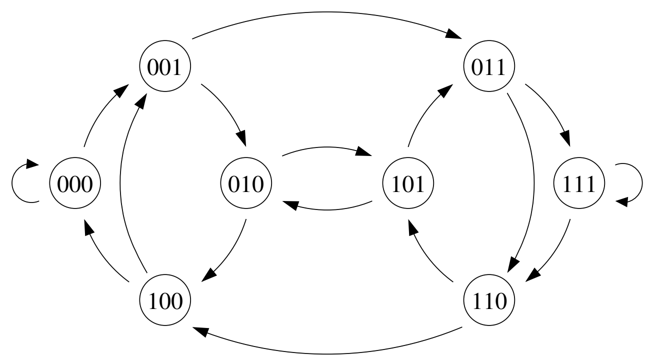

Figure A.1: State diagram for Example A.1. Each state s is labelled with $\langle s[3], s[2], s[1]\rangle$.

satisfiable:

$$\underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})}_{(s_0,\ldots,s_k)\text{ valid path prefix}} \wedge \underbrace{\bigvee_{i=0}^{k} \neg p(s_i)}_{p\text{ fails in }(s_0,\ldots,s_k)} \qquad \text{(A.1)}$$

Example A.1. We write $s[i]$ to denote bit i of the state $s = \langle s[0] \ldots s[n]\rangle$. Consider a 3-bit state transition system defined as follows and shown in Fig A.1:

$$\begin{aligned} I(s) &= \neg s[0] \wedge \neg s[1] \wedge \neg s[2] \\ R(s, s') &= (s[2] \Leftrightarrow s'[1]) \wedge (s[1] \Leftrightarrow s'[0]) \\ p(s) &= \neg s[0] \;. \end{aligned}$$

We want to model check the property $\mathbf{G}\, p$. First we try to find a counterexample of length $k = 0$. (We measure the length of a path prefix by the number of transitions between states, not the number of states; a counterexample of length 0 is a sequence of exactly one state.) Substituting into formula (A.1), we obtain:

$$I(s_0) \wedge \neg p(s_0) = (\neg s_0[0] \wedge \neg s_0[1] \wedge \neg s_0[2]) \wedge s_0[0]$$

which is clearly unsatisfiable, so no counterexample of length 0 exists. It turns out that the shortest counterexample is of length 3. In fact, for $k = 3$ we have that formula (A.1) becomes

$$\begin{aligned}(\neg s_0[0] \wedge \neg s_0[1] \wedge \neg s_0[2]) &\wedge (s_0[2] \Leftrightarrow s_1[1]) \wedge (s_0[1] \Leftrightarrow s_1[0]) \\ &\wedge (s_1[2] \Leftrightarrow s_2[1]) \wedge (s_1[1] \Leftrightarrow s_2[0]) \\ &\wedge (s_2[2] \Leftrightarrow s_3[1]) \wedge (s_2[1] \Leftrightarrow s_3[0]) \\ &\wedge (s_0[0] \vee s_1[0] \vee s_2[0] \vee s_3[0])\end{aligned}$$

which is satisfiable by the states $(s_0, s_1, s_2, s_3) = (\langle 000 \rangle, \langle 001 \rangle, \langle 011 \rangle, \langle 111 \rangle)$. Therefore, the sequence of state transitions $s_0 \to s_1 \to s_2 \to s_3$ is a counterexample to $\mathbf{G}\, p$.

In practice, the formulas obtained by expanding (A.1) can be very large. Nevertheless, BMC remains useful because modern SAT solvers can efficiently handle formulas with millions of clauses.

A.1.2 Determining the Bound

We now discuss two methods for determining the counterexample length when verifying a safety property such as $\mathbf{G}\, p$. Let d be the *diameter* of the system, i.e., the least number of steps to reach all reachable states. Alternatively, d is the least number for which the following holds: for every state s, if there exists a valid path prefix that contains s (i.e., s is reachable), then there exists a valid path prefix of length at most d that contains s. Clearly, if property p holds for all valid path prefixes of length k, where $k \geq d$, then p holds for all reachable states. So, we only need to consider counterexamples of length at most d. However, finding d is computationally hard.

Given a bound k, we can decide whether $k \geq d$ by solving a *quantified Boolean* *formula*. In particular, if every state reachable in $k+1$ steps can also be reached in up to k steps, then $k \geq d$. More formally, let $\mathit{reach}_{=n}$ and $\mathit{reach}_{\leq n}$ be the predicates defined over the state space S as follows:

$$\begin{aligned}
\mathit{reach}_{=n}(s) &= \exists s_0, ..., s_n \quad I(s_0) \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge s = s_n \\
\mathit{reach}_{\leq n}(s) &= \exists s_0, ..., s_n \quad I(s_0) \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{n} s = s_i \right)
\end{aligned}$$

The predicate $\mathit{reach}_{=n}(s)$ holds iff s is reachable in *exactly* n transitions, while $\mathit{reach}_{\leq n}$ holds iff s can be reached in *no more* than n transitions. Then, $k \geq d$ iff

$$\forall s \in S \quad \mathit{reach}_{=k+1}(s) \Rightarrow \mathit{reach}_{\leq k}(s)\ . \tag{A.2}$$

The above method of bounding the counterexample length is of limited value due to the difficulty of solving the quantified Boolean formula (A.2). Another way of using BMC to prove properties (i.e., not merely for bug-finding) is *k-induction* [55]. With k-induction, to prove a property $\mathbf{G}\, p$, one needs to a find an *invariant* q such that:

1. $q(s) \Rightarrow p(s)$, for all $s \in S$.
2. For every valid path prefix $(s_0, ..., s_k)$, $q(s_0) \wedge ... \wedge q(s_k)$ holds true.
3. For every state sequence $(s_0, ..., s_{k+1})$, if $\bigwedge_{i=0}^{k} R(s_i, s_{i+1})$ holds true then $(q(s_0) \wedge ... \wedge q(s_k)) \Rightarrow q(s_{k+1})$ holds true.

Other techniques for making BMC complete are cube enlargement [46], circuit cofactoring [23], and Craig interpolants [47].

A.1.3 BMC for General LTL Properties: Original Encoding

In this Section we present the BMC encoding for full LTL, as originally proposed by Biere *et al.* [9]. A counterexample to $\mathbf{F}\,p$ can only be an infinite path. In order to use a finite path prefix to represent an infinite path, we consider potential *back-loops* from the last state of a finite path prefix to an earlier state, as illustrated in Fig. A.2. More precisely, a valid path prefix $(s_0, ..., s_\ell, ..., s_k)$ has a *back-loop* from k to ℓ iff the transition relation R contains the pair (s_k, s_ℓ).

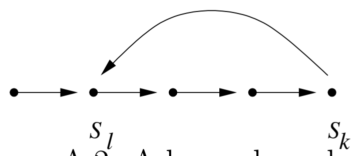

Figure A.2: A lasso-shaped path

Note that an LTL formula is false iff its negation is true. So, the problem of finding a counterexample of an LTL formula f is equivalent to the problem of finding a witness to its negation $\neg f$. In this section, we will follow this approach.

Given a state transition system M, an LTL formula f, and a bound k, we will construct a propositional formula $[\![M, f]\!]_k$ that holds true iff there exists a path prefix $(s_0, ..., s_k)$ along which f holds true. We assume that all negations in f have been pushed inward so that they occur only directly in front of atomic propositions. First we define a propositional formula $[\![M]\!]_k$ that constrains $(s_0, ..., s_k)$ to be a valid path prefix:

$$[\![M]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \ . \tag{A.3}$$

Now, we have to consider two cases, depending on whether the sequence $(s_0, ..., s_k)$ has a back-loop or not. First we consider the case without a back-loop. We introduce a bounded semantics, employing the following identities (similar to those used in the fixed-point characterizations of CTL):

- $\mathbf{F}\, f \;=\; f \vee \mathbf{X}\,\mathbf{F}\, f$
- $\mathbf{G}\, f \;=\; f \wedge \mathbf{X}\,\mathbf{G}\, f$
- $[f\;\mathbf{U}\;g] \;=\; g \vee (f \wedge \mathbf{X}\,[f\;\mathbf{U}\;g])$

Definition A.2 (Bounded Semantics without a Back-Loop)**.** Given a bound k and a finite or infinite sequence π whose first k states are $(s_0, ..., s_k)$, we say that an LTL formula f *holds true along* π *with bound* k iff $\pi \models_k^0 f$ is true, where $\pi \models_k^i f$ is defined recursively as follows for $i \in \{0, ..., k\}$:

$$
\begin{array}{lll}
\pi \models_k^i \; p & \text{iff} & \text{atomic proposition } p \text{ is true in state } s_i \\
\pi \models_k^i \; \neg p & \text{iff} & \text{atomic proposition } p \text{ is false in state } s_i \\
\pi \models_k^i \; f \vee g & \text{iff} & (\pi \models_k^i f) \text{ or } (\pi \models_k^i g) \\
\pi \models_k^i \; f \wedge g & \text{iff} & (\pi \models_k^i f) \text{ and } (\pi \models_k^i g) \\
\pi \models_k^i \; \mathbf{X}\, f & \text{iff} & i < k \text{ and } (\pi \models_k^{i+1} f) \\
\pi \models_k^i \; \mathbf{F}\, f & \text{iff} & \pi \models_k^i f \vee \mathbf{X}\,\mathbf{F}\, f \\
\pi \models_k^i \; \mathbf{G}\, f & \text{iff} & \pi \models_k^i f \wedge \mathbf{X}\,\mathbf{G}\, f \\
\pi \models_k^i \; f\;\mathbf{U}\;g & \text{iff} & \pi \models_k^i g \vee \big(f \wedge \mathbf{X}\,[f\;\mathbf{U}\;g]\big)
\end{array}
$$

Note that the recursion is well-founded, since $\pi \models_k^i \mathbf{X}\, f$ is false if $i \geq k$. This also means that formulas of the type $\mathbf{G}\, f$ do not hold true for any bound.

It is easily seen that $\pi \models_k^0 f$ implies $\pi \models f$ for any infinite path π and LTL formula f. Given a bound k, an LTL formula f, and a valid path prefix $(s_0, ..., s_k)$, we construct a propositional formula $[\![\, f \,]\!]_k^0$ that is true iff $\pi \models_k^0 f$.

Definition A.3 (Original translation of LTL formula without a loop)**.**

$$\begin{aligned}
[\![\, p \,]\!]_k^i &:= p(s_i) && \text{where } p \text{ is an atomic proposition}\\
[\![\, \neg p \,]\!]_k^i &:= \neg p(s_i) && \text{where } p \text{ is an atomic proposition}\\
[\![\, f \vee g \,]\!]_k^i &:= [\![\, f \,]\!]_k^i \vee [\![\, g \,]\!]_k^i\\
[\![\, f \wedge g \,]\!]_k^i &:= [\![\, f \,]\!]_k^i \wedge [\![\, g \,]\!]_k^i\\
[\![\, \mathbf{X}\, f \,]\!]_k^i &:= \begin{cases} [\![\, f \,]\!]_k^{i+1} & \text{if } i < k \\ \mathsf{false} & \text{otherwise} \end{cases}\\
[\![\, \mathbf{F}\, f \,]\!]_k^i &:= [\![\, f \,]\!]_k^i \vee [\![\, \mathbf{X}\,\mathbf{F}\, f \,]\!]_k^i\\
[\![\, \mathbf{G}\, f \,]\!]_k^i &:= [\![\, f \,]\!]_k^i \wedge [\![\, \mathbf{X}\,\mathbf{G}\, f \,]\!]_k^i\\
[\![\, f \;\mathbf{U}\; g \,]\!]_k^i &:= [\![\, g \,]\!]_k^i \vee \big([\![\, f \,]\!]_k^i \wedge [\![\, \mathbf{X}\,(f \;\mathbf{U}\; g) \,]\!]_k^i\big)
\end{aligned}$$

The translations for **F** and **G** are easily expanded:

$$\begin{aligned}
[\![\, \mathbf{F}\, f \,]\!]_k^i &= \bigvee_{j=i}^{k} [\![\, f \,]\!]_k^j\\
[\![\, \mathbf{G}\, f \,]\!]_k^i &= \mathsf{false}\ .
\end{aligned}$$

For $[\![\, f \;\mathbf{U}\; g \,]\!]_k^i$, we write a propositional formula that requires that g holds for some path suffix π^j (where $i \le j \le k$) and that f holds on all path suffixes in the set $\{\pi^n \mid i \le n < j\}$, as illustrated in Fig. A.3:

$$[\![\, f \;\mathbf{U}\; g \,]\!]_k^i = \bigvee_{j=i}^{k} \left([\![\, g \,]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![\, f \,]\!]_k^n \right) .$$

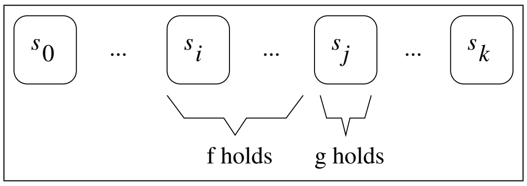

Figure A.3: Translation of $[\![\, f \,\mathbf{U}\, g \,]\!]_k^i$ for a loop-free path prefix.

Now consider a path prefix $(s_0, ..., s_k)$ with a back-loop from k to ℓ. Define an infinite lasso path π as shown in Fig. A.2: $\pi = (s_0, ..., s_{\ell-1},\ s_\ell, ...s_k,\ s_\ell, ..., s_k,\ ...)$. We construct a propositional formula ${}_\ell[\![\, f \,]\!]_k^0$ that holds iff f holds on π (in the usual LTL semantics).

Definition A.4 (Original translation of LTL formula with a loop)**.**

$$\begin{aligned}
{}_\ell[\![\, p \,]\!]_k^i &:= p(s_i) \qquad \text{where } p \text{ is an atomic proposition} \\
{}_\ell[\![\, \neg p \,]\!]_k^i &:= \neg p(s_i) \quad \text{where } p \text{ is an atomic proposition} \\
{}_\ell[\![\, f \vee g \,]\!]_k^i &:= {}_\ell[\![\, f \,]\!]_k^i \vee {}_\ell[\![\, g \,]\!]_k^i \\
{}_\ell[\![\, f \wedge g \,]\!]_k^i &:= {}_\ell[\![\, f \,]\!]_k^i \wedge {}_\ell[\![\, g \,]\!]_k^i \\
{}_\ell[\![\, \mathbf{X} f \,]\!]_k^i &:= \begin{cases} {}_\ell[\![\, f \,]\!]_k^{i+1} & \text{if } i < k \\ {}_\ell[\![\, f \,]\!]_k^{\ell} & \text{if } i = k \end{cases} \\
{}_\ell[\![\, \mathbf{G} f \,]\!]_k^i &:= \bigwedge_{j=\min(i,\ell)}^{k} {}_\ell[\![\, f \,]\!]_k^j \\
{}_\ell[\![\, \mathbf{F} f \,]\!]_k^i &:= \bigvee_{j=\min(i,\ell)}^{k} {}_\ell[\![\, f \,]\!]_k^j \\
{}_\ell[\![\, f \,\mathbf{U}\, g \,]\!]_k^i &:= \underbrace{\bigvee_{j=i}^{k} \left({}_\ell[\![\, g \,]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_\ell[\![\, f \,]\!]_k^n \right)}_{\text{Similar to loop-free case}} \vee \underbrace{\bigvee_{j=\ell}^{i-1} \left({}_\ell[\![\, g \,]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_\ell[\![\, f \,]\!]_k^n \wedge \bigwedge_{n=\ell}^{j-1} {}_\ell[\![\, f \,]\!]_k^n \right)}_{\text{See Fig. A.4}}
\end{aligned}$$

The translation for ${}_\ell[\![\, f \;\mathbf{U}\; g \,]\!]_k^i$ deserves some explanation. The translation is a disjunction of two parts. The first part is similar to the loop-free case. The second part is illustrated in Fig. A.4. It handles the case where f holds on all path suffixes from π^i through π^k, continues holding for π^ℓ through π^{j-1}, and then g holds on π^j. (Note that $\pi^{k+1} = \pi^\ell$, since π has infinite length.)

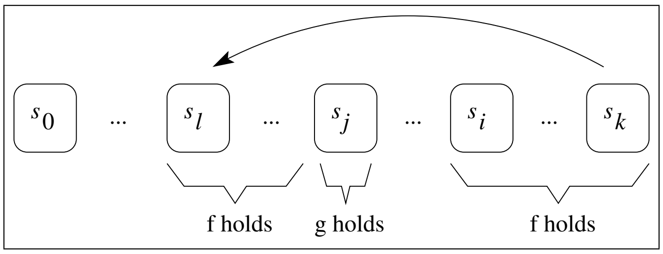

Figure A.4: Translation of ${}_\ell[\![\, f\ \mathbf{U}\ g\,]\!]_k^i$ for a path prefix with a back-loop.

Having defined the translation for paths both with and without back-loops, we are now almost ready to define the final translation into SAT. But first we need two auxiliary definitions. We define ${}_\ell L_k$ to be true iff there exists a transition from s_k to s_ℓ, and we define L_k to be true if there exists any possible back-loop in $(s_0, ..., s_k)$.

Definition A.5 (Loop Condition)**.** For $l \le k$, let ${}_\ell L_k := R(s_k, s_\ell)$, and let $L_k := \bigvee_{\ell=0}^k {}_\ell L_k$.

Now we are ready to state the final translation into SAT, which we denote by "$[\![\, M, f\,]\!]_k$":

$$[\![\, M, f\,]\!]_k \;:=\; \underbrace{[\![\, M\,]\!]_k}_{\text{valid prefix}} \wedge \Big(\underbrace{\big(\neg L_k \wedge [\![\, f\,]\!]_k^0\big)}_{\text{loop-free case}} \vee \underbrace{\bigvee_{\ell=0}^{k} \big({}_\ell L_k \wedge {}_\ell[\![\, f\,]\!]_k^0\big)}_{\text{case with loop}} \Big) \;.$$

Theorem 1. Given an LTL formula f, there exists a path π that satisfies f iff there exists a k such that $[\![\, M, f\,]\!]_k$ is satisfiable. Equivalently, $M \models \mathbf{A}\neg f$ iff $[\![\, M, f\,]\!]_k$ is unsatisfiable for all k.

A.1.4 Improved Encoding for General LTL Properties

The translations that we have given above in Definitions A.3 and A.4 are not the most efficient, although they have the benefit of being relatively straight-forward. More efficient translations are given in [10, 41, 42]; these translations have the benefit of having size linear in k (the unrolling depth) for the $\mathbf{U}$ operator, compared to size cubic in k (or quadratic in k, if certain optimizations [16] are used) for the translations in Definitions A.3 and A.4.

We use the same formula $[\![\, M \,]\!]_k$ as the original encoding (defined in Equation A.3 on page 90) to constrain the path to be a valid prefix. In addition, we define formulas for *loop constraints*, which are used to non-deterministically select at most one back-loop in the path prefix $(s_0, ..., s_k)$. We introduce $k+1$ fresh *loop selector variables*, $l_0, ..., l_k$, which determine which possible back-loop (if any) to select. If l_j is true (where $1 \leq j \leq k$), then we select a back-loop from k to j. The state s_{j-1} is constrained to be equal to the state s_k, and we consider an infinite path $\pi = (s_0, ..., s_{j-1},\ s_j, ..., s_k,\ s_j, ..., s_k,\ ...)$. If none of the loop selector variables are true, we use the bounded semantics (Definition A.2 on page 91).

We introduce auxiliary variables InLoop_0 through InLoop_k, which will be constrained so that InLoop_i is true iff position i is in the loop part of the path. In other words, InLoop_i should be true iff there exist a position $j \leq i$ such that l_j is true. To ensure that at most one of $\{l_0, ..., l_k\}$ is true, we require that l_i must not be true if there exists an earlier position $j < i$ such that l_j is true. Let $|[\mathit{LoopConstraints}]|_k$ be the conjunction of the following formulas for $i \in \{1, ..., k\}$:

$$\begin{aligned} l_0 &\Leftrightarrow \mathsf{false} \\ l_i &\Rightarrow (s_{i-1} = s_k) \\ \text{InLoop}_0 &\Leftrightarrow \mathsf{false} \\ \text{InLoop}_i &\Leftrightarrow \text{InLoop}_{i-1} \vee l_i \\ \text{InLoop}_{i-1} &\Rightarrow \neg l_i \end{aligned}$$

In Fig. A.5, we define a function $|[f]|_0$ that translates an LTL formula f into a Boolean formula that indicates whether the path prefix $(s_0, ..., s_k)$ is a witness for f. If none of the loop selector variables are true, then $|[f]|_{k+1}$ simplifies to false, in accord with the bounded semantics. If a single loop selector variable l_j is true, we consider an infinite path $\pi = (s_0, ..., s_{j-1},\ s_j, ..., s_k,\ s_j, ..., s_k,\ ...)$. Note that the infinite path suffix π^{k+1} is equal to π^j. Thus, the translation for $|[f]|_{k+1}$ simplifies to $|[f]|_j$, except in the case of the **U** operator.

For $|[f\ \mathbf{U}\ g]|_i$, we make two passes through the loop part of path prefix. On the first pass, we consider path suffixes π^i through π^k (see Fig. A.6). If f holds true for all these path suffixes, but g never holds true, then we need to make a second pass and continue checking at the start of the back-loop (π^j). If we reach position k on the second pass without g ever being true, then we know that g is never true at any position in the loop, so $f\ \mathbf{U}\ g$ is false. The auxiliary definition $\langle\langle f\ \mathbf{U}\ g \rangle\rangle$ handles the second pass. The final encoding for Kripke structure M, LTL formula f, and bound k is given by $|[M, f, k]|$:

$$|[M, f, k]| \ = \ |[M]|_k \wedge |[\mathit{LoopConstraints}]|_k \wedge |[f]|_0$$

Formula	Translation for $i \leq k$	Translation for $i = k+1$
$\|[p]\|_i$	$p(s_i)$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[p]\|_j\right)$
$\|[\neg p]\|_i$	$\neg p(s_i)$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[\neg p]\|_j\right)$
$\|[f \wedge g]\|_i$	$\|[f]\|_i \wedge \|[g]\|_i$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[f \wedge g]\|_j\right)$
$\|[f \vee g]\|_i$	$\|[f]\|_i \vee \|[g]\|_i$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[f \vee g]\|_j\right)$
$\|[\mathbf{X}\, f]\|_i$	$\|[f]\|_{i+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[\mathbf{X}\, f]\|_j\right)$
$\|[f\ \mathbf{U}\ g]\|_i$	$\|[g]\|_i \vee \left(\|[f]\|_i \wedge \|[f\ \mathbf{U}\ g]\|_{i+1}\right)$	$\bigvee_{j=1}^{k} \left(l_j \wedge \langle\langle f\ \mathbf{U}\ g\rangle\rangle_j\right)$
$\langle\langle f\ \mathbf{U}\ g\rangle\rangle_i$	$\|[g]\|_i \vee \left(\|[f]\|_i \wedge \langle\langle f\ \mathbf{U}\ g\rangle\rangle_{i+1}\right)$	false

Figure A.5: Improved BMC Translation

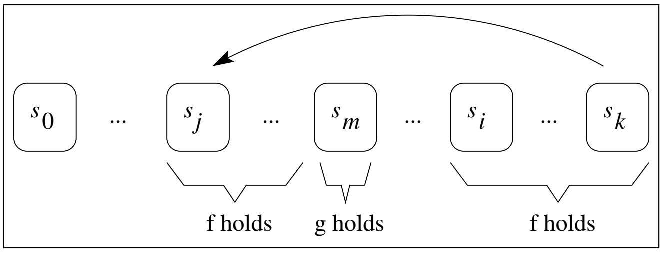

Figure A.6: Translation for a path prefix with a back-loop.

More compact translations are possible using QBF [38]. Often the transition relation is a very large formula, and Equation A.3 (on page 90) requires k copies of the transition relation. With a QBF encoding, the requirement $\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$ can be encoded using only a single copy of the transition relation, as follows:

$$\left(\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})\right) \equiv \left(\forall x, x'.\ \left(\bigvee_{i=0}^{k-1} (s_i = x) \wedge (s_{i+1} = x')\right) \Rightarrow R(x, x')\right) \quad \text{(A.4)}$$

Bibliography

[1] C. Ansótegui, C. P. Gomes, and B. Selman. The Achilles' Heel of QBF. In *AAAI 2005*. 2, 2.1

[2] A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In *FMCAD*, 2002. 5.1

[3] V. Balabanov and J.-H. R. Jiang. Unified QBF certification and its applications. *Formal Methods in System Design*, 41(1):45–65, 2012. 2.4.6, 4.3.4

[4] B. Becker, R. Ehlers, M. D. T. Lewis, and P. Marin. ALLQBF Solving by Computational Learning. In *ATVA*, 2012. 4.4, 1, 4.5

[5] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *LPAR 2004*. 3.4, 5.1

[6] M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *CADE*, 2005. 4.5

[7] M. Benedetti and H. Mangassarian. QBF-Based Formal Verification: Experience and Perspectives. *JSAT*, 2008. 4.5

[8] A. Biere. Resolve and Expand. In *SAT*, 2004. 3.4, 5.1

[9] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In *CAV*, 1999. A.1, A.1.3

[10] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science*, 2(5), 2006. A.1.4

[11] A. Biere, F. Lonsing, and M. Seidl. Blocked Clause Elimination for QBF. In *CADE*, 2011. 5.5

[12] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, 2011. 4.5

[13] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986. 4.1

[14] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An Open-Source Tool for Symbolic Model Checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. 4.4

[15] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999. 4.4

[16] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *VMCAI*, 2002. A.1.4

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003. 5.1

[18] A. Darwiche and P. Marquis. A Knowledge Compilation Map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002. 6

[19] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-

proving. *Commun. ACM*, 5(7):394–397, 1962. 1

[20] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, 2003. 2.4, 5.5

[21] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In *ECAI 2006*. 3.1

[22] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *DAC 2002*. 2

[23] M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic Model Checking using circuit cofactoring. In *International conference on Computer-aided design (ICCAD'04)*, pages 510–517, 2004. A.1.2

[24] I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In *SAT 2003*. 2.4.4

[25] E. Giunchiglia, P. Marin, and M. Narizzano. QuBE 7.0 System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7, 2010. 5.5

[26] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier structure in search based procedures for QBFs. In *DATE 2006*. 3.1, 3.4

[27] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In *IJCAR*, pages 364–369, 2001. 5.1

[28] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *J. Artif. Intell. Res. (JAIR)*, 26:371–416, 2006. 2.4.5

[29] E. Giunchiglia, M. Narizzano, A. Tacchella, et al. Learning for quantified boolean logic satisfiability. In *AAAI*, pages 649–654, 2002. 2.4.5

[30] E. Goldberg and P. Manolios. Quantifier elimination by Dependency Sequents.

In G. Cabodi and S. Singh, editors, *FMCAD*, pages 34–43. IEEE, 2012. 4.5

[31] A. Goultiaeva and F. Bacchus. Exploiting QBF Duality on a Circuit Representation. In *AAAI*, 2010. 2

[32] A. Goultiaeva and F. Bacchus. Recovering and Utilizing Partial Duality in QBF. In *Theory and Applications of Satisfiability Testing–SAT 2013*, pages 83–99. Springer, 2013. 3.4

[33] A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A Circuit-Based QBF Solver. In *SAT 2009*. 2, 3.4

[34] J. Huang and A. Darwiche. Using DPLL for Efficient OBDD Construction. In *SAT*, 2004. 4.5

[35] H. Jain, C. Bartzis, and E. M. Clarke. Satisfiability Checking of Non-clausal Formulas Using General Matings. In *SAT 2006*. 2

[36] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with Counterexample Guided Refinement. In *SAT*, pages 114–128, 2012. 5.1

[37] M. Janota and J. Marques-Silva. Abstraction-Based Algorithm for 2QBF. In *SAT*, 2011. 5.3

[38] T. Jussila and A. Biere. Compressing BMC Encodings with QBF. *Electr. Notes Theor. Comput. Sci.*, 174(3):45–56, 2007. A.1.4

[39] W. Klieber, M. Janota, J. Marques-Silva, and E. M. Clarke. Solving QBF with Free Variables. In *CP 2013*. 2

[40] W. Klieber, S. Sapra, S. Gao, and E. M. Clarke. A Non-prenex, Non-clausal QBF Solver with Game-State Learning. In *SAT*, 2010. 2, 3.1, 4.4

[41] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple Bounded LTL Model Checking. In *FMCAD*, pages 186–200, 2004. A.1.4

[42] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple Is Better: Efficient Bounded Model Checking for Past LTL. In *VMCAI*, pages 380–395, 2005. A.1.4

[43] F. Lonsing and A. Biere. Nenofex: Expanding NNF for QBF Solving. In *SAT 2008*. 2, 5.1

[44] F. Lonsing and A. Biere. Integrating dependency schemes in search-based QBF solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 158–171. Springer, 2010. 2.4.5, 3.1

[45] F. Lonsing, U. Egly, and A. Van Gelder. Efficient Clause Learning for Quantified Boolean Formulas via QBF Pseudo Unit Propagation. In *SAT 2013*, pages 100–115. Springer, 2013. 2.4.5

[46] 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. 4.5, A.1.2

[47] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer-Aided Verification (CAV'03)*, LNCS 2725, pages 1–13, 2003. A.1.2

[48] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC 2001*. 1, 2.4.4, 3.1

[49] M. Narizzano, L. Pulina, and A. Tacchella. QBFEVAL. `http://www.qbfeval.org/`. 3.4

[50] F. Pigorsch and C. Scholl. Exploiting structure in an AIG based QBF solver. In *DATE 2009*. 2, 3.4

[51] The Quantified Boolean Formulas Satisfiability Library. `http://www.qbflib.org/`. 5.5

[52] A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF

Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In *SAT 2006*. 2

[53] H. Samulowitz and F. Bacchus. Dynamically Partitioning for Solving QBF. In *SAT 2007*. 3.5, 6

[54] C. E. Shannon. The Synthesis of Two Terminal Switching Circuits. *Bell System Technical Journal*, 28:59–98, 1949. 4.1

[55] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS 1954, pages 108–125, 2000. A.1.2

[56] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996. 1, 2.4, 2.4.5

[57] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL Search. In *CP 2004*. 2

[58] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968. 2.1

[59] R. Wille, G. Fey, and R. Drechsler. Building free binary decision diagrams using SAT solvers. *Facta universitatis-series: Electronics and Energetics*, 2007. 4.5

[60] L. Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *AAAI 2006*. 2, 2.1, 2.2

[61] L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *ICCAD 2002*. 2.3, 2.3, 2.4.5, 2.4.6, 4.3.4

[62] L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *CP 2002*. 2.4.5