



Technisch-Naturwissenschaftliche  
Fakultät

# Extracting and Validating Skolem/Herbrand Function-Based QBF Certificates

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:

Mathias Preiner BSc

Angefertigt am:

Institut für Formale Modelle und Verifikation (FMV)

Beurteilung:

Univ.-Prof. Dr. Armin Biere

Mitwirkung:

DI Florian Lonsing

Assist.-Prof. Dr. Martina Seidl

Linz, März, 2012



# Abstract

Quantified Boolean Formulas (QBF) allow succinct representations of many real-world problems, e.g., in formal verification, artificial intelligence, or computer-aided design of integrated circuits. Hence, for many practical instances of QBF efficient decision procedures are highly requested. However, in many applications of QBF, it is not sufficient to "just" solve problems but necessary to provide additional valuable information also denoted as *certificates*. Given a problem formulated in QBF, it is possible to extract such certificates.

In this thesis, we present the tools QRPcert and CertCheck for extracting and validating Skolem/Herbrand function-based QBF certificates of (un)satisfiability, which we obtain from Q-resolution proofs and traces. We discuss the process of certificate extraction as implemented in QRPcert in detail and further describe the validation of those certificates by means of CertCheck and a SAT solver.

We applied our tools to the benchmark sets of the QBF competitions 2008 and 2010 and provide an extensive evaluation of the results. As a base for QBF certificate extraction, we used Q-resolution traces recorded by the state-of-the-art QBF solver DepQBF. The results of our experiments show that the employed extraction technique is a promising approach for a unified certification framework for QBF.

Finally, we discuss open problems and ideas for extending our tools in order to improve the certification workflow as employed in our experiments.



# Zusammenfassung

Quantifizierte Boolesche Formeln (QBF) erlauben kompakte Kodierungen von Problemen in vielen Anwendungsbereichen wie z.B., Formaler Verifikation, Künstlicher Intelligenz, sowie der Entwicklung von integrierten Schaltungen. Ein entscheidender Faktor dabei ist die Anwendung von effizienten Algorithmen für die Evaluierung von QBF. In vielen Anwendungsbereichen ist es jedoch nicht ausreichend nur die Erfüllbarkeit von Problemen zu bestimmen, sondern notwendig zusätzliche Informationen über den Beweishergang in Form von sogenannten *Zertifikaten* zur Verfügung zu stellen. Diese Zertifikate bieten nützliche Informationen über das Problem selbst und sind daher für viele Anwendungsbereiche äußerst wichtig.

Diese Arbeit beschäftigt sich mit der Extrahierung und Validierung von Skolem-/Herbrand-Funktions-basierten QBF-Zertifikaten. Es werden die wichtigsten Algorithmen in QRPCert, einem Werkzeug zur Extrahierung von QBF-Zertifikaten aus Q-Resolutionsbeweisen, im Detail vorgestellt. Darüberhinaus wird gezeigt, wie die von QRPCert extrahierten Zertifikate auf ihre Richtigkeit überprüft werden.

Diese Arbeit beinhaltet umfangreiche Experimente, die auf den Benchmark-Tests der QBF Wettbewerbe von 2008 und 2010 durchgeführt wurden. Für die Extrahierung der Zertifikate wurden Q-Resolutionsbeweise verwendet, die von DepQBF, einem Programm zur Evaluierung von QBF in konjunktiver Pränex-Normalform, erzeugt wurden. Die Ergebnisse der Experimente zeigen, dass die angewandte Technik zur Extrahierung von QBF Zertifikaten einen vielversprechenden Ansatz für die einheitliche Zertifizierung von wahren und falschen QBF bietet.

Am Ende dieser Arbeit werden einige Ideen zur Erweiterung der entwickelten Programme diskutiert, um den in den Experimenten angewendeten Zertifizierungsprozess weiter zu verbessern.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	Quantified Boolean Formulas . . . . .	3
2.2	And-Inverter Graphs . . . . .	10
2.3	Skolemization and Skolem Functions . . . . .	11
<b>3</b>	<b>QBF Certificates</b>	<b>13</b>
3.1	Related Work . . . . .	14
3.2	Skolem/Herbrand Function Extraction . . . . .	16
<b>4</b>	<b>QRPcert: Certificate Extraction</b>	<b>19</b>
4.1	Overview . . . . .	19
4.2	Input File . . . . .	21
4.3	Build Proof DAG . . . . .	23
4.4	Prepare Vertices . . . . .	23
4.5	Extract Skolem/Herbrand Functions . . . . .	25
4.6	Simplify Skolem/Herbrand Functions . . . . .	30
4.7	Construct And-Inverter Graphs . . . . .	33
<b>5</b>	<b>CertCheck: Certificate Validation</b>	<b>39</b>
5.1	Overview . . . . .	39
5.2	Certificate Validation . . . . .	42
<b>6</b>	<b>Experimental Results</b>	<b>45</b>
6.1	Overview . . . . .	45
6.2	Runtime Comparison . . . . .	50
6.3	Certificate Statistics . . . . .	53
6.4	Simplification Results . . . . .	56
6.5	Increasing the Memory Limit . . . . .	57
<b>7</b>	<b>Conclusion</b>	<b>59</b>
7.1	Future Work . . . . .	60

<b>A Appendix</b>	<b>67</b>
A.1 QRP format . . . . .	67



# Chapter 1

## Introduction

Quantified Boolean formulas (QBF) are an extension of propositional logic (SAT) and allow existential and universal quantification over variables. QBF provide compact encodings for many problems in formal verification [7, 14, 24, 31, 19, 44], planning [40, 39, 17, 38] and electronic design automation [10, 34, 49, 20]. By allowing quantification of variables, the complexity class of the satisfiability problem of QBF is raised from NP-complete (SAT) to PSPACE-complete. Due to the wide range of applicability of QBF there is high interest in developing efficient algorithms for solving many practical instances of QBF problems. The most popular approach to solve the satisfiability problem for QBF is an extension of the successful Davis-Putnam-Logemann-Loveland (DPLL) [13] procedure for SAT as employed in [51, 16, 30]. Other approaches employ techniques like Skolemization [4, 25, 26], Q-resolution with expansion [5], or circuit-based procedures [22].

Efficient evaluation techniques are crucial in applications of QBF but not the only important aspect. If a QBF solver concludes that a given problem is satisfiable (*sat*) resp. unsatisfiable (*unsat*), there is no way to verify the correctness of the result without further evidence. Hence, techniques for validating the answer of a QBF solver are highly requested.

As an example consider model checking, where an unsatisfiable problem indicates that the system to be checked is free from certain types of defects, whereas a satisfiable problem indicates that certain requested properties are not met and the system might be erroneous. Therefore, in many applications it is not sufficient that a solver returns mere *sat/unsat* answers. For example, in case of erroneous behaviour of a system the result of the solver should be the basis for providing counter-examples. As a matter of fact, it is possible to provide evidence of the correctness of a solver's result by extracting so-called *certificates* of (un)satisfiability for a solved problem. These certificates further provide valuable information to serve as a base for extracting e.g., error traces or counter-examples.

In SAT, it is relatively straightforward to extract certificates of (un)satisfiability from a SAT solver, where a certificate is either represented as a model or a resolution proof depending on whether given problem is satisfiable or unsatisfiable. Certificates of satisfiability may, e.g., be used to provide counter-examples, whereas resolution proofs may be used as input for other algorithms such as the computation of Craig interpolants in model checking [32].

In case of QBF, certificates can be beneficial in similar ways, but they are considerably harder to extract. Commonly accepted representations of certificates of (un)satisfiability are Q-resolution proofs and sets of Boolean functions that represent the truth values of the existentially quantified variables. Given a satisfiable QBF, certificates of satisfiability can be either represented by *cube resolution proofs* as in [50, 35] or as sets of so-called *Skolem functions* as in [4, 25, 26]. Certificates of unsatisfiability are usually represented by *clause resolution proofs*. In case that a given certificate is a set of Skolem functions, we gain further valuable information on the solved problem, which can be exploited, e.g., to provide counter-examples or error traces to faulty states. In [44], e.g., Skolem function-based certificates are used for localizing and correcting faults in sequential circuits. Skolem functions are directly derivable for Skolemization-based solvers as presented in [4, 25, 26]. However, the extraction of Skolem functions is not directly applicable to QBF solvers that employ the successful DPLL style procedures.

Recently, a promising approach for extracting Skolem function-based certificates of satisfiability (resp. their dual counterpart, Herbrand function-based certificates of unsatisfiability) from cube (resp. clause) resolution proofs was introduced in [2]. This technique is solver-independent and may be applied to any QBF solver that provides resolution proofs of (un)satisfiability. Hence, it is also applicable and well-suited for DPLL-based QBF solvers.

In this thesis, we present tools for extracting and validating Skolem/Herbrand function-based QBF certificates that are obtained from Q-resolution proofs, based on the approach introduced in [2]. The structure of the thesis is organized as follows. In Chapter 2, we give an overview on the preliminaries required for this thesis followed by a short summary on related work and the introduction of Skolem/Herbrand function extraction in Chapter 3. After that, in Chapters 4 and 5 we describe the tools developed for extracting and validating Skolem/Herbrand function-based QBF certificates. Chapter 6 provides an extensive evaluation of the experimental results conducted on the benchmark sets of the QBF competitions 2008 and 2010. In the last chapter, we discuss the results and provide an outlook on future improvements of our framework.

# Chapter 2

## Preliminaries

### 2.1 Quantified Boolean Formulas

Quantified Boolean formulas (QBF) are an extension of propositional logic and introduce universal and existential quantification over propositional variables. This extension provides quantified Boolean formulas with a powerful and compact representation for important application in artificial intelligence, knowledge representation, verification, and synthesis.

#### 2.1.1 Syntax

In addition to universal ( $\forall$ ) and existential ( $\exists$ ) quantifiers, quantified Boolean formulas also employ a set of logical connectives restricted to the conjunction ( $\wedge$ ), disjunction ( $\vee$ ) and negation ( $\neg$ ). Further, quantified Boolean formulas may also contain the Boolean constants true ( $\top$ ) and false ( $\perp$ ). In the following, we use  $\rightarrow$  and  $\leftrightarrow$  for denoting logical implication resp. equivalence.

**Definition 2.1** (QBF). *The set of quantified Boolean formulas is inductively defined as in [11].*

1. Propositional variables and the Boolean constants are quantified Boolean formulas.
2. If  $\Phi_1$  and  $\Phi_2$  are quantified Boolean formulas, then  $\neg\Phi_1$ ,  $\Phi_1 \wedge \Phi_2$  and  $\Phi_1 \vee \Phi_2$  are quantified Boolean formulas.
3. If  $\Phi$  is a quantified Boolean formula, then  $\exists x.\Phi$  and  $\forall x.\Phi$  are quantified Boolean formulas.
4. Only formulas given by (1) to (3) are quantified Boolean formulas.

The set of variables of a quantified Boolean formula  $\Phi$  is denoted as  $\text{var}(\Phi)$ , which is composed of the set of existentially quantified variables  $\text{var}_{\exists}(\Phi)$  and the set of universally quantified variables  $\text{var}_{\forall}(\Phi)$  of  $\Phi$ . A *literal*  $l$  is

either a variable  $x$  or its negation  $\neg x$  with  $x \in \text{var}(\Phi)$ , which we also denote as positive or negative occurrence of variable  $x$ , respectively. Variable  $x$  in formula  $\exists x.\phi$  (resp.  $\forall x.\phi$ ) is called a *quantified variable*, where  $\phi$  denotes the *scope* of  $x$ . An occurrence of variable  $x$  is called *bound* if the scope of  $\exists x$  (resp.  $\forall x$ ) includes the occurrence of  $x$ . All occurrences of variable  $x$  that are not bound are called *free* occurrences. A variable  $x$  in formula  $\Phi$  is called free (resp. bound) if there is a free (resp. bound) occurrence of  $x$  in  $\Phi$ . A formula is called *closed* if it does not contain any free variables. In the following, we consider closed quantified Boolean formulas in *prenex normal form*, i.e., a QBF in the form  $Q.\phi$ , where  $Q$  is a sequence of quantified variables denoted as *prefix* and a propositional formula  $\phi$  denoted as *matrix*.

**Definition 2.2** (Prenex Normal Form (PNF)). *Let  $\Phi$  be a QBF in prenex normal form such that:*

$$\Phi = \underbrace{Q_1 X_1 \dots Q_n X_n}_{\text{prefix}} \cdot \underbrace{\phi(x_1, x_2, \dots, x_m)}_{\text{matrix}}$$

where  $Q_i$  is a quantifier with  $Q_i \in \{\forall, \exists\}$ ,  $X_i$  is a set of variables with  $X_i \subseteq \text{var}(\Phi)$  bound by  $Q_i$  at nesting level  $i$  and  $\phi$  is a propositional formula over variables  $x_j \in \text{var}(\Phi)$ .

Note that the prefix is linearly ordered by nesting level, i.e., the variables of  $X_i$  precede the variables of  $X_{i+1}$  (resp. the variables of  $X_{i+1}$  trail the variables of  $X_i$ ). We say that  $x \prec y$  with  $x \in X_i$  and  $y \in X_{i+1}$  iff  $X_i$  precedes  $X_{i+1}$ . Nesting level 1 (resp. level  $n$ ) is called the outermost (resp. innermost) nesting level of formula  $\Phi$ . Further, note that any QBF in non-prenex form, as introduced in Definition 2.1, can be transformed into prenex normal form, as shown in [27].

The matrix of a quantified Boolean formula in PNF may be represented in several different normal forms.

**Definition 2.3** (Conjunctive Normal Form (CNF)). *A propositional formula is in conjunctive normal form if it is a conjunction of clauses  $C_1 \wedge \dots \wedge C_n$ , where a clause is a disjunction of literals, i.e.,  $C_i = l_1 \vee \dots \vee l_m$ .*

**Definition 2.4** (Disjunctive Normal Form (DNF)). *A propositional formula is in disjunctive normal form if it is a disjunction of cubes  $C_1 \vee \dots \vee C_n$ , where a cube is a conjunction of literals, i.e.,  $C_i = l_1 \wedge \dots \wedge l_m$ .*

Note that an empty clause (resp. empty cube), i.e., a clause (resp. cube) without any literals, may be represented as the empty set  $\emptyset$  and is considered to be  $\perp$  (resp.  $\top$ ). Any propositional formula can be transformed into CNF or DNF, while potentially increasing the size of the formula [27]. The transformation into CNF is done via *Tseitin's encoding* [48], whereas converting a formula into DNF is achieved by applying rewriting rules like

the distributivity law and De Morgan's laws [27]. Note that a propositional formula in CNF (resp. DNF) may also be considered as a set of clauses (resp. cubes)  $\{C_1, \dots, C_n\}$ , where a clause (resp. cube)  $C$  is a set of literals  $\{l_1, \dots, l_m\}$ . In the following, we consider quantified Boolean formulas to be given in either *prenex conjunctive normal form* or *prenex disjunctive normal form*.

**Definition 2.5** (Prenex Conjunctive Normal Form (PCNF)). *A QBF is in prenex conjunctive normal form if it is in prenex normal form and its matrix is in conjunctive normal form.*

**Definition 2.6** (Prenex Disjunctive Normal Form (PDNF)). *A QBF is in prenex disjunctive normal form if it is in prenex normal form and its matrix is in disjunctive normal form.*

*Example 2.1.* Given a propositional formula  $\phi = x \leftrightarrow y$  representing the logical equivalence of the variables  $x$  and  $y$ . We reformulate  $\phi$  into a QBF stating that for every truth value of  $x$  there exists an equivalent truth value for  $y$ . Formula 2.1 represents the resulting QBF  $\Phi_1$  in PCNF, whereas the equivalent QBF  $\Phi_2$  in PDNF is shown in Formula 2.2.

$$\Phi_1 = \forall x \exists y. (\neg x \vee y) \wedge (x \vee \neg y) \quad (2.1)$$

$$\Phi_2 = \forall x \exists y. (x \wedge y) \vee (\neg x \wedge \neg y) \quad (2.2)$$

### 2.1.2 Semantics

In contrast to the NP-complete satisfiability problem of propositional logic (also denoted as SAT), deciding the satisfiability of closed quantified Boolean formulas (also denoted as QSAT) is PSPACE-complete, as shown in [45].

**Definition 2.7** (Satisfiability of propositional logic). *Let  $\phi$ ,  $\phi'$ , and  $\phi''$  be propositional formulas and let  $\neg$ ,  $\wedge$ , and  $\vee$  be the only logical connectives used. Further, let  $\mathcal{I}$  be a given mapping from the set of propositional variables to the set of truth values  $\{\mathbb{F}, \mathbb{T}\}$ . Then,  $\mathcal{I}$  can be extended for arbitrary formulas as follows.*

1. If  $\phi = \top$ , then  $\mathcal{I}(\phi) = \mathbb{T}$ .
2. If  $\phi = \perp$ , then  $\mathcal{I}(\phi) = \mathbb{F}$ .
3. If  $\phi = v$  and  $v$  is a propositional variable, then  $\mathcal{I}(\phi) = \mathbb{T}$  iff  $\mathcal{I}(v) = \mathbb{T}$ .
4. If  $\phi = \neg\phi'$ , then  $\mathcal{I}(\phi) = \mathbb{T}$  iff  $\mathcal{I}(\phi') = \mathbb{F}$ .
5. If  $\phi = \phi' \vee \phi''$ , then  $\mathcal{I}(\phi) = \mathbb{T}$  iff  $\mathcal{I}(\phi') = \mathbb{T}$  or  $\mathcal{I}(\phi'') = \mathbb{T}$ .
6. If  $\phi = \phi' \wedge \phi''$ , then  $\mathcal{I}(\phi) = \mathbb{T}$  iff  $\mathcal{I}(\phi') = \mathbb{T}$  and  $\mathcal{I}(\phi'') = \mathbb{T}$ .

The satisfiability of quantified Boolean formulas is defined by extending Definition 2.7 as follows.

**Definition 2.8** (Satisfiability of QBF). *Let  $\Phi = Q.\phi$  be a QBF in PNF. Then*

1.  $\exists x.\phi$  is satisfiable iff  $\mathcal{I}(\phi[x/\perp]) = \mathbb{T}$  or  $\mathcal{I}(\phi[x/\top]) = \mathbb{T}$ .
2.  $\forall x.\phi$  is satisfiable iff  $\mathcal{I}(\phi[x/\perp]) = \mathbb{T}$  and  $\mathcal{I}(\phi[x/\top]) = \mathbb{T}$ .

where  $\phi[x/\perp]$  (resp.  $\phi[x/\top]$ ) denotes the substitution of all occurrences of  $x$  in  $\phi$  by  $\perp$  (resp.  $\top$ ). The substitution is also denoted as assigning the Boolean constant  $\perp$  (resp.  $\top$ ) to variable  $x$ .

In case of a satisfiable propositional formula  $\phi$ , it is sufficient to show that there exists an assignment of truth values (i.e., Boolean constants) to a subset of  $\text{var}(\phi)$  such that  $\phi$  evaluates to true. However, showing that a QBF  $\Phi$  is satisfiable requires to construct a set of Boolean functions for (a subset of)  $\text{var}_{\exists}(\Phi)$ , where a function represents the truth value of the respective variable depending on the assignments of all preceding universally quantified variables. This set of Boolean functions is denoted as a *satisfiability model*. We generalize the definition of a satisfiability model as presented in [11] as follows.

**Definition 2.9** (Satisfiability Model). *Let  $\Phi = Q.\phi$  be a closed QBF in PNF with  $\text{var}_{\exists}(\Phi) = \{y_1, \dots, y_k\}$ . Let  $f_{y_i} = f(x_1, \dots, x_n)$  for all  $x_j$  with  $x_j \prec y_i$  and  $x_j \in \text{var}_{\forall}(\Phi)$  be a Boolean function representing the truth value of variable  $y_i$ . Let  $M = \{f_{y_1}, \dots, f_{y_k}\}$  be a set of Boolean functions.  $M$  is a satisfiability model of  $\Phi$  iff  $\mathcal{I}(\phi[y_1/f_{y_1}, \dots, y_k/f_{y_k}]) = \mathbb{T}$ .*

Note that Boolean functions are constants if the corresponding existentially quantified variable is not dependent on preceding universally quantified variables. In this case, the Boolean functions are represented by the Boolean constants  $\perp$  and  $\top$ . If we extend Definition 2.9 to quantified Boolean formulas with free variables, this would also apply to the Boolean functions of the corresponding free variables, as they are considered to be existentially quantified and to precede all other bound variables (as defined in the QDIMACS<sup>1</sup> standard).

*Example 2.2.* As an example, we consider a satisfiable QBF in PCNF given in Formula 2.3 and its satisfiability model  $M = \{f_{y_1}, f_{y_2}\}$ , where  $f_{y_1} = \perp$  and  $f_{y_2} = \neg x_1 \vee \neg x_2$ .

$$\exists y_1 \forall x_1 x_2 \exists y_2. (\neg y_1 \vee x_1) \wedge (x_1 \vee \neg x_2 \vee y_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg y_2) \wedge (x_2 \vee y_2) \quad (2.3)$$

Substituting all occurrences of the existential variables  $y_1$  and  $y_2$  with their resp. Boolean function yields Formula 2.4. The resulting Formula 2.4 is true for every assignment of  $x_1$  and  $x_2$  and therefore,  $M$  is a valid satisfiability model of Formula 2.3.

$$\forall x_1 x_2. (\neg f_{y_1} \vee x_1) \wedge (x_1 \vee \neg x_2 \vee f_{y_2}) \wedge (\neg x_1 \vee \neg x_2 \vee \neg f_{y_2}) \wedge (x_2 \vee f_{y_2}) \quad (2.4)$$

Note that  $f_{y_1}$  is assigned to the Boolean constant  $\perp$  because variable  $y_1$  neither depends on  $x_1$  nor  $x_2$ .

<sup>1</sup><http://www.qbfiib.org/qdimacs.html>

### 2.1.3 Q-Resolution

Q-Resolution is a complete and sound approach for evaluating quantified Boolean formulas [12] and is an extension of the resolution rule of propositional logic.

**Definition 2.10** (Resolution). *Let  $\phi$  be a propositional formula in CNF and let  $C_1$  and  $C_2$  be two clauses of  $\phi$ . If there exists a variable  $x \in \text{var}(\phi)$  such that  $x \in C_1$  and  $\neg x \in C_2$ , resolving  $C_1$  and  $C_2$  on pivot variable  $x$  yields the resolvent  $C = \{C_1 \cup C_2\} \setminus \{x, \neg x\}$ .*

**Proposition 2.1** ([41]). *Resolvent  $C$  is a logical consequence of  $C_1$  and  $C_2$  and can be added to  $\phi$  without changing the truth value of  $\phi$ .*

Q-Resolution extends the resolution rule for propositional logic on CNF to PCNF by adding two rules: first, the pivot variable is restricted to be an existentially quantified variable. Second, all universally quantified literals that do not precede an existentially quantified literal are removed, which is called *Universal-Reduction*.

**Proposition 2.2** (Universal-Reduction [11]). *Let  $\Phi = Q.\phi$  be a QBF in PCNF and let  $C$  be a clause in  $\phi$ . Let  $y$  be the innermost existentially quantified literal in  $C$ . All universally quantified literals  $x_i \in C$  with  $y \prec x_i$  can be eliminated from  $C$  without changing the truth value of  $\Phi$ .*

**Definition 2.11** (Q-Resolution for PCNF). *Let  $\Phi = Q.\phi$  be a QBF in PCNF and let  $C_1$  and  $C_2$  be two clauses in  $\phi$ . If there exists an existentially quantified variable  $y \in \text{var}_\exists(\Phi)$  such that  $y \in C_1$  and  $\neg y \in C_2$ , we obtain a Q-resolvent by resolving  $C_1$  and  $C_2$  on pivot variable  $y$  by applying the following two steps.*

1. *Apply the universal-reduction rule on  $C_1$  and  $C_2$  and obtain the universal-reduced clauses  $C'_1$  and  $C'_2$ .*
2. *Resolve  $C'_1$  and  $C'_2$  on variable  $y$  and obtain the Q-resolvent  $\{C'_1 \cup C'_2\} \setminus \{y, \neg y\}$ . If the resulting clause is a tautology, no Q-resolvent is obtained.*

Note that it is required to apply universal-reduction in each Q-resolution step, otherwise Q-resolution would not be refutation complete [11].

For quantified Boolean formulas in PDNF, the Q-resolution rule is analogously defined to Definition 2.11 and Proposition 2.2 for PCNF.

**Proposition 2.3** (Existential-Reduction [1]). *Let  $\Phi = Q.\phi$  be a QBF in PDNF and let  $C$  be a cube in  $\phi$ . Let  $x$  be the innermost universally quantified literal in  $C$ . All existentially quantified literals  $y_i \in C$  with  $x \prec y_i$  can be eliminated from  $C$  without changing the truth value of  $\Phi$ .*

**Definition 2.12** (Q-Resolution for PDNF). *Let  $\Phi = Q.\phi$  be a QBF in PDNF and let  $C_1$  and  $C_2$  be two cubes in  $\phi$ . If there exists an universally quantified variable  $x \in \text{var}_\forall(\Phi)$  such that  $x \in C_1$  and  $\neg x \in C_2$ , we obtain a Q-resolvent by resolving  $C_1$  and  $C_2$  on pivot variable  $x$  by applying the following two steps.*

1. *Apply the existential-reduction rule on  $C_1$  and  $C_2$  and obtain the existential-reduced cubes  $C'_1$  and  $C'_2$ .*
2. *Resolve  $C'_1$  and  $C'_2$  on variable  $x$  and obtain the Q-resolvent  $\{C'_1 \cup C'_2\} \setminus \{x, \neg x\}$ . If the resulting cube is a contradiction, no Q-resolvent is obtained.*

In the following, we refer to Q-resolution for PCNF (resp. PDNF) as clause (resp. cube) resolution. Further, we use  $\text{qres}(C_1, C_2)$  for denoting clause resolution (resp. cube resolution) over  $C_1$  and  $C_2$ , which we also refer to as a clause (resp. cube) resolution step. To indicate the application of universal- (resp. existential-) reduction to a clause (resp. cube)  $C$  we use  $\text{red}(C)$ . We further use constraint for denoting a clause resp. cube.

#### 2.1.4 Q-Resolution Proofs

Both clause and cube resolution are sound and complete proof systems for evaluating quantified Boolean formulas in PCNF resp. PDNF [12, 18].

**Theorem 2.1** ([12]). *A QBF in PCNF is unsatisfiable if and only if there exists a clause resolution sequence leading to the empty clause.*

**Theorem 2.2** ([18]). *A QBF in PDNF is satisfiable if and only if there exists a cube resolution sequence leading to the empty cube.*

A clause (resp. cube) resolution sequence leading to the empty clause (resp. cube) contains all original clauses (resp. initial cubes) and Q-resolution steps that are required for the derivation of the empty clause (resp. empty cube) and is referred to as *Q-resolution proof*. As cube resolution is not directly applicable to quantified Boolean formulas in PCNF, a possible solution is to transform the formula into PDNF while risking an exponential increase in the size of the formula. However, this transformation is not required as there exists a technique to extract cube resolution proofs for satisfiable quantified Boolean formulas in PCNF [18]. Resolution proofs can be interpreted as directed acyclic graphs (DAG) and are defined similar to [2] as follows.

**Definition 2.13.** *Let  $\Pi$  be a clause (resp. cube) resolution sequence. Let  $G_\Pi = \{V_\Pi, E_\Pi\}$  be a directed acyclic graph representing  $\Pi$ .  $G_\Pi$  consists of the set of vertices  $V_\Pi$  and the set of directed edges  $E_\Pi \subseteq V_\Pi \times V_\Pi$ . A vertex  $v \in V_\Pi$  corresponds to a clause (resp. cube) in  $\Pi$ , whereas an edge  $(u, v) \in E_\Pi$  from antecedent  $u$  to vertex  $v$  indicates that  $v$  is obtained by either resolution or reduction.*



*Example 2.3.* We consider the satisfiable QBF in PCNF given as Formula 2.3 and transform it into PDNF by applying the distributive law and subsumption rule, resulting in Formula 2.5.

$$\exists y_1 \forall x_1 x_2 \exists y_2. \underbrace{(\neg y_1 \wedge y_2 \wedge \neg x_1)}_{c_1} \vee \underbrace{(\neg y_1 \wedge y_2 \wedge \neg x_2)}_{c_2} \vee \underbrace{(x_1 \wedge \neg y_2 \wedge x_2)}_{c_3} \vee \underbrace{(x_1 \wedge y_2 \wedge \neg x_2)}_{c_4} \quad (2.5)$$

A valid cube resolution proof for Formula 2.5 deriving the empty cube is, e.g.,  $\Pi = \{c_1, c_3, c_4, c_5 = \text{qres}(c_3, c_4), c_6 = \text{qres}(c_1, c_5)\}$ . The corresponding DAG of  $\Pi$  is depicted in Figure 2.1, where cubes are represented in set notation. Further, for the sake of simplicity, reduction and resolution of Q-resolution are treated separately and represented explicitly as vertices with one and two incoming edges, respectively.

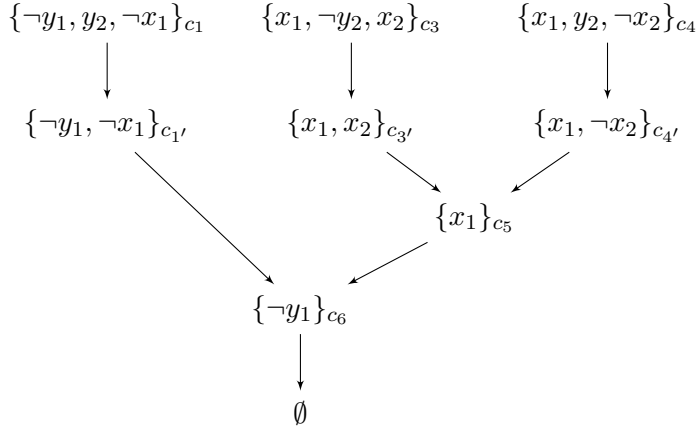


Figure 2.1: DAG of cube resolution proof for Formula 2.5.

Note that  $c_2$  of Formula 2.5 is not required to derive the empty cube and is thus not part of the resolution proof.

In the following, we treat vertices in a proof DAG and the constraints they represent equally, i.e., we use the terminology interchangeably in the context of proof DAGs.


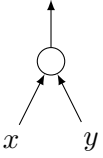
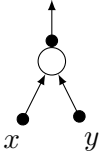
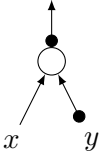
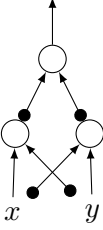
Negation	Conjunction	Disjunction	Implication	Equivalence
$\neg x$	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \leftrightarrow y$
				
$\neg x$	$x \wedge y$	$\neg(\neg x \wedge \neg y)$	$\neg(x \wedge \neg y)$	$\neg(x \wedge \neg y) \wedge$ $\neg(\neg x \wedge y)$

Table 2.1: Basic logical connectives represented as and-inverter graphs.

## 2.2 And-Inverter Graphs

And-inverter graphs (AIG) [28, 29] are directed acyclic graphs commonly used for representing combinational logic and are implemented in several logic synthesis and verification systems such as ABC [46] and MVSIS [47]. AIGs are a representation of Boolean formulas, where the set of logical connectives is restricted to conjunction ( $\wedge$ ) and negation ( $\neg$ ), which form a complete set of connectives in that all others can be expressed [15].

**Definition 2.14** (And-Inverter Graph). *An and-inverter graph is a directed acyclic graph representing primary inputs, primary outputs and two-input AND-gates as vertices with either 0 or 2 incoming edges, and 0 or more outgoing edges. A primary input is represented by a vertex with no incoming edges and can be either a Boolean variable or the Boolean constant  $\perp$ . Vertices with two incoming edges represent two-input AND-gates, whereas vertices with no outgoing edges are primary outputs. Negation is indicated by a complemented edge.*

In general, and-inverter graphs are non-canonical, i.e., there is no unique representation of a Boolean formula as an AIG. In most cases, and-inverter graphs show a high degree of redundancy, which can be reduced considerably by sharing isomorphic vertices and subgraphs. Several optimization techniques have been proposed for minimizing the overall size of and-inverter graphs such as structural and functional hashing [29, 33, 8], which can be applied during AIG construction.

Table 2.1 shows the equivalent two-input AIG representation of the most common logical connectives, where AND-gates are denoted by circles and negations by dots.

## 2.3 Skolemization and Skolem Functions

Skolemization is a technique originating from the domain of first-order logic for eliminating existential quantifiers from any first-order logic formula and is also applicable to quantified Boolean formulas [3]. By applying Skolemization to a QBF  $\Phi$  in PNF, we obtain a QBF  $\Phi_S$  in so-called *Skolem normal form*, which has the following two properties: first,  $\Phi_S$  contains no existential quantifiers and second,  $\Phi_S$  and  $\Phi$  are equisatisfiable, i.e.,  $\Phi_S$  is satisfiable if and only if  $\Phi$  is satisfiable (and vice versa). Existential quantifiers are eliminated by replacing all existentially quantified variables with so-called *Skolem functions* [43], which are defined as follows.

**Definition 2.15** (Skolem Function). *Let  $\Phi$  be a closed QBF in PNF with  $\text{var}_{\exists}(\Phi) = \{y_1, \dots, y_k\}$ . A Skolem function  $f_{y_i}$  of an existentially quantified variable  $y_i \in \text{var}_{\exists}(\Phi)$  is defined as  $f_{y_i} = f(x_1, \dots, x_n)$  over all  $x_j$  with  $x_j \prec y_i$  and  $x_j \in \text{var}_{\forall}(\Phi)$ . We further say that  $f_{y_i}$  is of arity  $n$  as it is defined over  $n$  variables. If a Skolem function is of arity zero, we replace the corresponding existentially quantified variable with a so-called Skolem constant, which in the case of QBF is either the Boolean constant  $\top$  or  $\perp$ .*

Note that Skolem functions are not in the language of QBF, where function symbols are not allowed. However, it is possible to obtain a valid QBF without function symbols by substituting each Skolem function by the formula it represents.

**Definition 2.16** (Skolemization). *Let  $\Phi = Q.\phi$  be a closed QBF in PNF. Starting with the outermost quantifier in the prefix, the Skolemization of  $\Phi$  is obtained by repeatedly applying the following transformation until all existential quantifiers are eliminated.*

$$\underbrace{\forall x_1, \dots, x_n \exists y, Q'}_Q . \phi \rightsquigarrow \underbrace{\forall x_1, \dots, x_n, Q'}_{Q \setminus \{y\}} . \phi[y / f_y(x_1, \dots, x_n)]$$

*The resulting formula  $\Phi_S$  contains only universally quantified variables and is in Skolem normal form.*

*Example 2.4.* As an example, we consider Formula 2.6 with matrix  $\phi$  defined over the variables  $x_1, x_2, x_3, y_1, y_2, y_3$ .

$$\exists y_1 \forall x_1 x_2 \exists y_2 \forall x_3 \exists y_3 . \phi(y_1, x_1, x_2, y_2, x_3, y_3) \quad (2.6)$$

We start with the outermost existential quantifier, replace  $y_1$  with a Skolem constant  $c_{y_1}$ , as  $y_1$  is not dependent on preceding universal variables, and obtain Formula 2.7.

$$\forall x_1 x_2 \exists y_2 \forall x_3 \exists y_3 . \phi(c_{y_1}, x_1, x_2, y_2, x_3, y_3) \quad (2.7)$$

We continue with the next existential quantifier and replace  $y_2$  with its Skolem function  $f_{y_2}$  resulting in Formula 2.8.

$$\forall x_1 x_2 x_3 \exists y_3. \phi(c_{y_1}, x_1, x_2, f_{y_2}(x_1, x_2), x_3, y_3) \quad (2.8)$$

Finally, we introduce  $f_{y_3}$  and obtain Formula 2.9 in Skolem normal form.

$$\forall x_1 x_2 x_3. \phi(c_{y_1}, x_1, x_2, f_{y_2}(x_1, x_2), x_3, f_{y_3}(x_1, x_2, x_3)) \quad (2.9)$$

### 2.3.1 Herbrandization and Herbrand Functions

Herbrandization is dual to Skolemization and is a technique for universal quantifier elimination. A formula  $\Phi_H$ , obtained by applying Herbrandization to formula  $\Phi$ , is in so-called *Herbrand normal form* and contains—dual to Skolem normal form—no universal quantifiers. Further, the transformation into Herbrand normal form is validity preserving, i.e.,  $\Phi_H$  is valid if and only if  $\Phi$  is valid (and vice versa). Universal quantifiers are eliminated by replacing all universally quantified variables with *Herbrand functions*, which are dually defined to Skolem functions.

*Example 2.5.* By applying Herbrandization to Formula 2.6, we obtain Formula 2.10 in Herbrand normal form.

$$\exists y_1 y_2 y_3. \phi(y_1, f_{x_1}(y_1), f_{x_2}(y_1), y_2, f_{x_3}(y_1, y_2), y_3) \quad (2.10)$$

Note that we introduced new Skolem (resp. Herbrand) functions for all existentially (resp. universally) quantified variables without further specifying the structure of the functions. In Section 3.2, we describe how we obtain Skolem (resp. Herbrand) functions for the purpose of QBF certificate extraction in more detail.

## Chapter 3

# QBF Certificates

The term *certificate* refers to any means of providing evidence of the (un)satisfiability of a given SAT resp. QBF problem. A certificate can be used to verify the correctness of a solver's result and further provides valuable information on the solution of a solved problem, which, for instance, can be exploited to provide counter-examples in model checking.

In case of SAT, a certificate of satisfiability is represented by a satisfying assignment of truth values to the variables of a formula, which can be easily extracted from a SAT solver. Verifying the satisfiability of a propositional formula is straightforward as we simply have to check whether every clause of the formula is satisfied by at least one of its literals under given satisfying assignment. In contrast, showing that a formula is unsatisfiable requires a clause resolution sequence leading to the empty clause. However, in case of SAT validating certificates of satisfiability and unsatisfiability can be done by means of efficient polynomial-time proof checkers [25].

In the context of QBF, both representation and validation of certificates is significantly more complex due to the inherently tree-like structure of QBF models. A certificate of satisfiability may be either represented as cube resolution proof or as a set of Skolem functions, which represent assignments to the existentially quantified variables with respect to their preceding universal variables. In case of unsatisfiable formulas, certification is usually done via clause resolution proofs. In addition to the verification of the correctness of the result of a QBF solver, QBF certificates also provide means to identify so-called *strategies*, which are a crucial piece of information in most game-like scenarios as well as other applications of QBF. Hence, certificates are highly requested for practical applications of QBF, e.g., in the field of formal verification, model checking, and artificial intelligence.

### 3.1 Related Work

Several approaches and tools for generating and validating certificates for QBF have been presented over the last 10 years [4, 50, 25, 35]. In most cases, certificates of satisfiability were either represented as cube resolution proofs or as sets of Skolem functions, whereas clause resolution proofs usually represented certificates of unsatisfiability.

**sKizzo/ozziKs** In [4], the Skolemization-based QBF solver **sKizzo** is instrumented to generate a so-called *inference log*, which contains all information necessary to reproduce each step performed by the solver. An inference log of **sKizzo** represents a list of instantiations of satisfiability-preserving transformations like variable assignment, variable substitution, variable elimination and clause resolution. Once an inference log is generated, it is evaluated using a reconstruction tool called **ozziKs** as follows. In case given problem is satisfiable, **ozziKs** extracts a model by applying an inductive model reconstruction procedure while reading the inference log backwards. The resulting model is represented by a set of Skolem functions and encoded into a Binary Decision Diagram [9] based certificate of satisfiability. If given problem is unsatisfiable, **ozziKs** extracts a so-called *unsatisfiable core*, i.e., a subset of the original set of clauses of the unsatisfiable input formula that is still unsatisfiable.

**Squolem/QBV** The Skolemization-based QBF solver **Squolem** [25] represents certificates of satisfiability with a set of Skolem functions, which are constructed during the solving process. In case of unsatisfiable instances, **Squolem** provides a clause resolution trace as certificate of unsatisfiability. The certificates extracted by **Squolem** are validated by a tool called **QBV** (Quantified Boolean Verifier). A certificate of satisfiability is incrementally checked by loading the model into a SAT solver and adding the negation of each clause of the input formula as assumption. If the result is unsatisfiable, given certificate of satisfiability is valid.

**yQuaffle** In [50], the search-based QBF solver **yQuaffle** is instrumented to record cube (resp. clause) resolution traces as certificates of satisfiability (resp. unsatisfiability). Each time a cube (resp. clause) is learned during the solving process, **yQuaffle** records all cubes (resp. clause) that were involved in deriving the learned cube (resp. clause). The resulting trace contains all cube (resp. clause) resolution sequences required for deriving the empty cube (resp. empty clause). The traces extracted by **yQuaffle** are validated with a verifier that implements a depth-first search and breadth-first search verification algorithm as introduced in [50].

**EBDDRES/TraceCheck** The BDD-based QBF solver EBDDRES [26, 42] supports the extraction of both certificates of satisfiability and unsatisfiability. In case of unsatisfiable instances, EBDDRES provides clause resolution traces as certificates of unsatisfiability, whereas sets of Skolem functions are extracted in case given instance is satisfiable [25]. Clause resolution traces generated by EBDDRES can either be validated by TraceCheck [42] or QBV [25]. Validation of certificates of satisfiability is done via QBV.

**CheQ** CheQ [35] is a proof-of-concept suite for extracting and validating certificates of satisfiability and unsatisfiability built on top of the search-based QBF solver QuBE. It consists of QuBE-cert, an extension of QuBE for extracting certificates, and Checker, a tool for validating the certificates generated by QuBE-cert. Certificates of satisfiability (resp. unsatisfiability) are represented with cube (resp. clause) resolution proofs, which are recorded during the solving process of QuBE-cert. The resulting certificates are validated by Checker, which checks whether each cube (resp. clause) in the cube (resp. clause) resolution proof has been derived correctly. Note that proofs generated by QuBE-cert may contain long-distance resolution [51], which is not supported by Checker and therefore cannot be validated.

Note that most of the tools described above are not maintained anymore. Further, only the QBF solvers sKizzo, Squolem and EBDDRES with their resp. tools provide sets of Skolem functions for satisfiable problems, which can be employed as strategies. None of the described QBF solvers is able to provide strategies for unsatisfiable instances.

A more recent approach is described in [21], where the circuit-based QBF solver CirQit has been extended to produce clause resolution proofs for both satisfiable and unsatisfiable instances by using a technique called *dual propagation*. Due to the circuit representation of the input formula, CirQit is able to solve the negated input formula without expensive transformation steps involved. The resulting clause resolution proofs are validated by QBV and are used to compute strategies by employing the algorithm presented in [21]. This approach allows dual treatment of certificates of (un)satisfiability but cannot exploit its full strength on formulas in PCNF.

Another recent approach for extracting certificates is presented in [2], where Skolem (resp. Herbrand) function-based certificates are extracted from cube (resp. clause) resolution proofs. A prototype called ResQu was implemented, which supports the extraction of certificates from Q-resolution proofs provided by either QuBE-cert or Squolem (in case of unsatisfiable instances). This approach enables dual treatment of certificates of (un)satisfiability and further allows the extraction of strategies from solvers that employ the successful variant of DPLL style procedures for QBF.

In [36], the state-of-the-art QBF solver DepQBF [30] was extended to provide Q-resolution traces, which in this thesis serve as a base to extract Skolem/Herbrand function-based QBF certificates. In the following, we introduce the theoretical background for extracting Skolem/Herbrand functions from Q-resolution proofs as employed in QRPcert based on the approach presented in [2].

## 3.2 Skolem/Herbrand Function Extraction

Given a Q-resolution proof of satisfiability (resp. unsatisfiability), we extract Skolem (resp. Herbrand) functions by traversing a proof DAG in topological order with the restriction that a vertex may not be processed before all of its antecedent vertices are processed. This restriction guarantees traversal sequences in the proof DAG that comply to valid Q-resolution derivations, i.e., a constraint  $c$  is not processed until all other constraints that are required for deriving  $c$  are processed. Therefore, we define a partial order relation over the vertices of a proof DAG, which considers above restriction, as follows.

**Definition 3.1** (Partial Order of Vertices). *Let  $G = (V, E)$  be a directed acyclic graph with a set of vertices  $V$  and a set of directed edges  $E$ . A node  $u_n$  is a successor of a node  $u_1$  (written as  $u_1 < u_n$ ) if there exists a sequence of edges  $(u_1, u_2), \dots, (u_{n-1}, u_n)$  with  $(u_i, u_{i+1}) \in E$  and  $1 \leq i < n$ . Node  $v$  is a direct successor of  $u$  if  $(u, v) \in E$ .*

Given the partial order relation over the vertices of a proof DAG, we obtain a valid topological order for extracting Skolem (resp. Herbrand) functions. The processing order of the vertices can be efficiently obtained by employing a depth-first search-like traversal of the proof DAG, which we describe in Chapter 4 in more detail.

*Example 3.1.* Given the proof DAG in Figure 2.1, a partial order of its vertices may be defined as the ordered set  $\{c_1, c'_1, c_3, c'_3, c_4, c'_4, c_5, c_6, \emptyset\}$ . Due to the fact that the relation  $<$  only applies to vertices on a path, the order of  $c_1$ ,  $c_3$  and  $c_4$  is not important as long as  $c_1 < c'_1 < c_6$ ,  $c_3 < c'_3 < c_5$  and  $c_4 < c'_4 < c_5$  is not violated.

As in [2], we employ a specific formula structure for constructing Skolem (resp. Herbrand) functions denoted as Right-First-And-Or (RFAO) formula, which is defined as follows.

**Definition 3.2** (RFAO construction rule [2]). *The structure of a Right-First-And-Or (RFAO) formula  $\phi$  is recursively defined by*

$$\phi : \text{clause} \mid \text{cube} \mid \text{clause} \wedge (\phi) \mid \text{cube} \vee (\phi) \quad (3.1)$$

where ” $\mid$ ” denotes a selection of given expressions.



In the following, we interpret RFAO formulas as ordered sets of constraints, which are read in the following way:

$$\begin{aligned} \phi &= \text{cube}_1 \vee (\text{clause}_1 \wedge (\text{clause}_2 \wedge (\text{cube}_2 \vee (\text{clause}_3)))) \\ &= \{\text{cube}_1, \text{clause}_1, \text{clause}_2, \text{cube}_2, \text{clause}_3\} \end{aligned} \quad (3.2)$$

We generalize the algorithm for extracting Skolem (resp. Herbrand) functions from Q-resolution proofs as introduced in [2] as follows.

**Definition 3.3** (Skolem Function Extraction). *Let  $\Phi$  be a satisfiable QBF in PCNF with  $y \in \text{var}_\exists(\Phi)$  and let  $\Pi$  be a cube resolution proof of  $\Phi$ . Let  $C_y = \{\text{red}(c) \mid c \in \Pi \wedge y \in (c \setminus \text{red}(c))\}$  and  $C_{\neg y} = \{\neg \text{red}(c) \mid c \in \Pi \wedge \neg y \in (c \setminus \text{red}(c))\}$  be a set of cubes and clauses, respectively. The RFAO formula representing the Skolem function of  $y$  is defined by the set  $F_y = C_y \cup C_{\neg y}$  ordered by  $<$ . The Skolem function is obtained by applying the construction rule of a RFAO formula to  $F_y$  as defined in Definition 3.2.*

Note that set  $C_y$  contains all existentially reduced cubes  $c' = \text{red}(c)$ , where literal  $y$  is one of the literals eliminated by applying existential reduction to cube  $c$ .  $C_{\neg y}$ , on the other hand, contains the negated form of all cubes (clauses)  $\neg c'$  with  $c' = \text{red}(c)$ , where literal  $\neg y$  is one of the literals eliminated by  $\text{red}(c)$ .

**Definition 3.4** (Herbrand Function Extraction). *Let  $\Phi$  be an unsatisfiable QBF in PCNF with  $x \in \text{var}_\forall(\Phi)$  and let  $\Pi$  be a clause resolution proof of  $\Phi$ . Let  $C_x = \{\text{red}(c) \mid c \in \Pi \wedge x \in (c \setminus \text{red}(c))\}$  and  $C_{\neg x} = \{\neg \text{red}(c) \mid c \in \Pi \wedge \neg x \in (c \setminus \text{red}(c))\}$  be a set of clauses and cubes, respectively. The RFAO formula representing the Herbrand function of  $x$  is defined by the set  $G_x = C_x \cup C_{\neg x}$  ordered by  $<$ . The Herbrand function is obtained by applying the construction rule of a RFAO formula to  $G_x$  as defined in Definition 3.2.*

Given an unsatisfiable QBF  $\Phi$  in PCNF and its clause resolution proof  $\Pi$ . The constructed set of constraints  $G_x$  for the universally quantified variable  $x \in \text{var}_\forall(\Phi)$  obtained by Definition 3.4 contains all constraints that are extracted by algorithm `Countermodel_construct` introduced in [2]. The construction of  $F_y$  in Definition 3.3 is dual to the construction of  $G_x$  in Definition 3.4. The correctness of algorithm `Countermodel_construct` is shown in [2].

*Example 3.2.* Given the proof DAG of Figure 2.1 representing a cube resolution proof of the satisfiable Formula 2.3, the Skolem functions extracted for the existentially quantified variables  $y_1$  and  $y_2$  are obtained as follows.

$$\begin{aligned} C_{y_1} &= \{\} & C_{\neg y_1} &= \{\neg \emptyset\} & F_{y_1} &= C_{y_1} \cup C_{\neg y_1} = \{\neg \emptyset\} \\ C_{y_2} &= \{c'_1, c'_4\} & C_{\neg y_2} &= \{\neg c'_3\} & F_{y_2} &= C_{y_2} \cup C_{\neg y_2} = \{c'_1, \neg c'_3, c'_4\} \end{aligned}$$

By applying the construction rule of a RFAO formula to  $F_{y_1}$  and  $F_{y_2}$ , we obtain the Skolem functions  $f_{y_1}$  and  $f_{y_2}$  as follows.

$$\begin{aligned} f_{y_1} &= \neg\top = \perp \\ f_{y_2} &= c'_1 \vee (\neg c'_3 \wedge (c'_4)) = (\neg y_1 \wedge \neg x_1) \vee ((\neg x_1 \vee \neg x_2) \wedge (x_1 \wedge \neg x_2)) \end{aligned}$$

Note that Skolem function  $f_{y_2}$  depends on the existentially quantified variable  $y_1$ . As we defined Skolem functions to be dependent on universally quantified variables only, we eliminate  $y_1$  by substituting it with its Skolem function  $f_{y_1}$ , which results in:

$$f_{y_2} = (\neg f_{y_1} \wedge \neg x_1) \vee ((\neg x_1 \vee \neg x_2) \wedge (x_1 \wedge \neg x_2)) = \neg x_1 \vee \neg x_2$$

The Skolem functions  $f_{y_1}$  and  $f_{y_2}$  represent concrete assignments to the variables  $y_1$  and  $y_2$ , such that under all possible assignments of the variables  $x_1$  and  $x_2$  Formula 2.3 is satisfiable. Table 3.1 shows that each clause of Formula 2.3 is satisfied by at least one literal (columns 3-6) under all possible assignments of  $x_1$  and  $x_2$  (columns 1 and 2).

		Clauses of Formula 2.3			
$x_1$	$x_2$	$\neg f_{y_1} \vee x_1$	$x_1 \vee \neg x_2 \vee f_{y_2}$	$\neg x_1 \vee \neg x_2 \vee \neg f_{y_2}$	$x_2 \vee f_{y_2}$
$\perp$	$\perp$	$\neg f_{y_1}$	$\neg x_2$	$\neg x_1$	$f_{y_2}$
$\perp$	$\top$	$\neg f_{y_1}$	$f_{y_2}$	$\neg x_1$	$x_2$
$\top$	$\perp$	$\neg f_{y_1}$	$x_1$	$\neg x_2$	$f_{y_2}$
$\top$	$\top$	$\neg f_{y_1}$	$x_1$	$\neg f_{y_2}$	$x_2$

Table 3.1: Formula 2.3 satisfied by Skolem functions  $f_{y_1}$  and  $f_{y_2}$ .

Based on the theory discussed above, we implemented a tool for extracting Skolem/Herbrand function-based QBF certificates. In the following chapters, we describe the implementation of certificate extraction and validation in more detail, and further provide an extensive evaluation on recent benchmark sets.

## Chapter 4

# QRPcert: Certificate Extraction

QRPcert is a tool for extracting Skolem/Herbrand function-based QBF certificates of (un)satisfiability from Q-resolution proofs and traces based on the algorithm introduced in [2]. In the following, we describe certificate extraction as implemented in QRPcert in detail.

### 4.1 Overview

QRPcert extracts Skolem (resp. Herbrand) function-based QBF certificates of satisfiability (resp. unsatisfiability) from clause (resp. cube) resolution proofs and traces. It further provides the possibility to extract a subset of Skolem (resp. Herbrand) functions for variables of interest. As input format for Q-resolution proofs and traces, QRPcert currently supports the QRP format [36], a lightweight and explicit format for representing clause (resp. cube) resolution proofs and traces. QRPcert represents Skolem (resp. Herbrand) functions as AIGs, which are simplified by common basic simplification techniques such as structural sharing of isomorphic AND-gates and Boolean constant propagation. A certificate generated by QRPcert is represented as AIG in the ASCII version of the AIGER<sup>1</sup> format.

In the following, we describe the general workflow of QRPcert and its main steps for generating certificates from Q-resolution proofs of satisfiability (resp. unsatisfiability) as depicted in Figure 4.1. Given a Q-resolution proof (resp. trace) as input file, we construct the full Q-resolution proof DAG data structure. After that, we prepare all vertices that are required for deriving the empty constraint for the traversal of the proof DAG in the extraction phase. We then construct the RFAO formulas for all required existentially (resp. universally) quantified variables by traversing the proof

---

<sup>1</sup><http://fmv.jku.at/aiger/FORMAT.aiger>

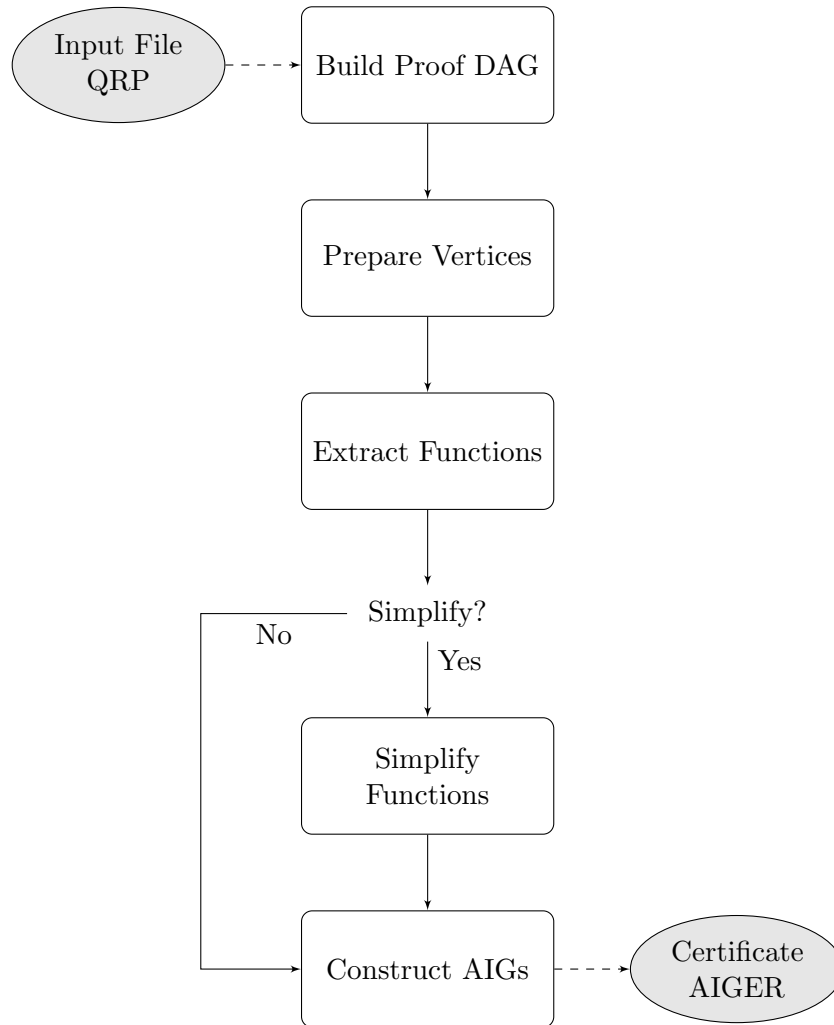


Figure 4.1: General workflow of QRPCert.

DAG in topological order as defined in Definition 3.1. In case simplification is enabled, we propagate Boolean constants derived during the extraction phase in order to simplify the extracted Skolem (resp. Herbrand) functions. Finally, we transform each Skolem (resp. Herbrand) function into an AIG and export it into ASCII AIGER format. The resulting set of AIGs represent the certificate of satisfiability (resp. unsatisfiability). In the following, we describe each individual step of the workflow in more detail.

## 4.2 Input File

QRPcert currently supports Q-resolution proofs and traces in QRP format, which is based on the QDIMACS format for QBF and the tracecheck<sup>2</sup> format for SAT. Note that a Q-resolution trace contains all Q-resolution sequences that were derived during the solving process of a QBF solver. Thus, it may contain Q-resolution sequences that are not required for deriving the empty constraint.

We introduce the QRP format with the example trace depicted in Figure 4.2b. The corresponding unsatisfiable input formula in QDIMACS format is given in Figure 4.2a. Given a file in QRP format, the header line starts with "p qrp" and specifies the number of variables and the number of steps given. In this example, the number of variables is 9 and the number of steps is 13. The header line is followed by a set of quantified sets of variables, which denotes the prefix of the input formula. A quantified set consists of a quantifier (either universal (a) or existential (e)) and a list of variables terminated by 0. For example, line "a 1 2 0" defines a universally quantified set that contains the variables 1 and 2.

The prefix definition is followed by a list of Q-resolution steps, which are uniquely identified by their step id. Each step further consists of a constraint (given as a list of literals) and its antecedents, separated by the delimiter 0. Original clauses of the input formula (resp. initial cubes) do not have any antecedents, whereas constraints obtained by resolution have exactly two antecedents and constraints obtained by reduction have exactly one antecedent. In our example, clauses 1 to 6 are original clauses of the input formula in Figure 4.2a, whereas clauses 7 to 13 were obtained by resolution. Further, clause 7 contains the literals -5, -7, and -8 and is obtained by resolving clauses 2 and 3. Note that a negative number indicates a negative occurrence of a variable.

The last line starting with "r" is the result statement, which indicates whether given Q-resolution proof is a proof of satisfiability ("r sat") or unsatisfiability ("r unsat"). In case that given file is a Q-resolution trace, the result statement indicates if the trace contains a proof of satisfiability or unsatisfiability. Note that the QRP format does not explicitly distinguish between clauses and cubes and further allows that a trace may contain both clause and cube resolution sequences. Therefore, it is important to consider only those clause (resp. cube) resolution sequences that are required for deriving the empty clause (resp. empty cube). The grammar of the QRP format is provided in the appendix A.1, a more detailed introduction is given in [36].

---

<sup>2</sup><http://fmv.jku.at/booleforce/README.tracecheck>

<pre> p cnf 9 6 a 1 2 0 e 3 0 a 4 0 e 5 6 0 a 7 0 e 8 9 0 4 5 -7 -8 0 6 -7 -8 0 -5 -6 7 0 -2 3 8 0 1 -3 -9 0 8 9 0 </pre> <p>(a) Input formula</p>	<pre> p qrp 9 13 a 1 2 0 e 3 0 a 4 0 e 5 6 0 a 7 0 e 8 9 0 1 4 5 -7 -8 0 0 2 6 -7 -8 0 0 3 -5 -6 7 0 0 4 -2 3 8 0 0 5 1 -3 -9 0 0 6 8 9 0 0 7 -5 -7 -8 0 2 3 0 8 -2 3 -5 0 7 4 0 9 4 -7 -8 0 1 7 0 10 4 -7 9 0 9 6 0 11 -2 3 0 9 4 0 12 1 -3 0 6 10 0 13 0 11 12 0 r unsat </pre> <p>(b) Clause resolution trace</p>
----------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 4.2: Clause resolution trace (b) in QRP format of input formula (a) given in QDIMACS format.

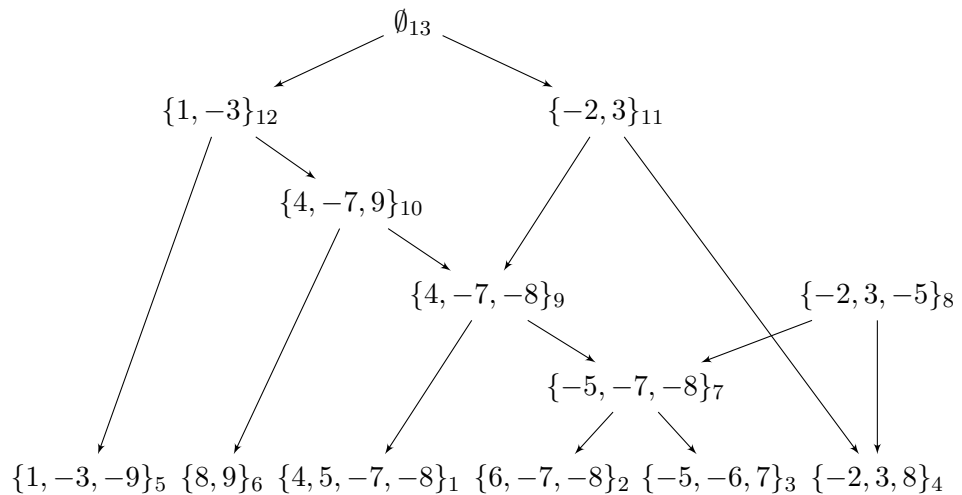


Figure 4.3: Proof DAG of trace in Figure 4.2b as represented in QRPCert.

### 4.3 Build Proof DAG

The first step is to process the input file and to build an internal representation of the set of Q-resolution steps given. QRPCert internally maintains Q-resolution steps of proofs and traces as vertices in a proof DAG similar to the representation introduced in Figure 2.1 except for two differences. First, existential- (resp. universal-) reduction of Q-resolution steps is not made explicit, i.e., we do not create an additional vertex that represents a reduction step. However, if the input file contains an explicit reduction step, we represent it in the proof DAG, accordingly. Second, the directed edges are reversed, i.e., the proof DAG starts with the empty constraint and edges point from parent vertices to its antecedents. In the following, we divide the set of vertices of a proof DAG into two sets  $V_p$  and  $V_i$ , where  $V_p$  denotes the set of vertices required for deriving the empty constraint and  $V_i$  represents the set of irrelevant vertices.

QRPCert generates a total order over all variables by assigning consecutive indices in the order of appearance in the prefix. Note that free variables are not allowed. The new index is used for sorting the literals of a constraint, which is important for AIG construction, which is discussed in Section 4.7 in more detail. Note that QRPCert internally maintains variables and vertices in arrays in order to provide fast access to their data structure via their respective indices. Therefore, variable indices as well as vertex indices are renumbered consecutively in order to avoid sparse arrays and thus, keep the memory overhead as small as possible.

Figure 4.3 depicts the proof DAG representation of the clause resolution trace in Figure 4.2b in QRPCert, where clauses are represented as sets of literals with the corresponding vertex id as subscript. For the sake of simplicity, the variables of given example are already consecutively indexed and all literals in the clauses are sorted with respect to their index.

### 4.4 Prepare Vertices

For the extraction of Skolem (resp. Herbrand) functions it is important to process the proof DAG in topological order as defined in Definition 3.1. Therefore, in [2], the proof DAG is traversed starting with the original clauses (resp. initial cubes) with the restriction that a vertex is processed only after all of its antecedents are processed. We implemented a different approach in QRPCert, where we traverse the proof DAG in depth-first search-like order starting with the empty constraint. Our condition for processing a vertex is that a vertex may not be processed before all of its parent vertices have been processed. Hence, the vertices of the proof DAG are processed in the reverse topological order defined in Definition 3.1. The advantage of the approach implemented in QRPCert is that only vertices required for deriving

```

1 function prepare_vertices ()
2 {
3   STACK s
4   VERTEX v
5   push (s, empty_vertex)
6   while not is_empty (s)
7   {
8     v ← pop (s)
9     if get_ref_cnt (v) = 0
10    {
11      foreach a in get_antecedents (v)
12        push (s, a)
13    }
14    incr_ref_cnt (v)
15  }
16 }

```

Figure 4.4: Compute the reference counters of all required vertices.

the empty constraint are processed as the Q-resolution proof is extracted on-the-fly and thus, processing Q-resolution traces does not affect the result nor the performance of the extraction process.

QRPCert maintains reference counters for vertices to determine the number of parent vertices of a vertex. We use the reference counter of a vertex to check if it has yet unvisited parent vertices that have to be processed first. As we only process vertices in  $V_p$ , it is important that we consider references from vertices in  $V_p$  only, i.e., we do not consider references from vertices in  $V_i$  to vertices in  $V_p$ . Hence, we traverse all vertices in  $V_p$  and compute their reference counters. Note that the reference counters cannot be computed during the construction of the proof DAG as the vertices can only be divided into  $V_p$  and  $V_i$  after the complete proof DAG is constructed.

Figure 4.4 describes the algorithm for computing the reference counters of all vertices in  $V_p$ . Algorithm `prepare_vertices` traverses the proof DAG in depth-first search order starting with the empty vertex `e` in order to ensure that only vertices in  $V_p$  are considered. Initially, the reference counters of all vertices are set to 0. In case we that visit vertex `v` for the first time (i.e., the reference counter of `v` is 0), we push the antecedents of `v` onto the visit stack. Further, each time we visit vertex `v`, we increment its reference counter by one. After the algorithm terminates, each vertex  $v \in V_p$  has been visited  $n$  times, where  $n$  is the number of parent vertices of  $v$ . Note that in case of the empty vertex, in order to avoid special treatment during the extraction phase its reference counter is always set to 1 even though it is not referenced by any other vertex.



Figure 4.5 illustrates the reference counters (in parenthesis) of all vertices (labeled with their resp. id) of the proof DAG in Figure 4.3 after algorithm `prepare_vertices` terminates. Note that vertex 8 is irrelevant for the proof, hence its references to vertex 7 and 4 are not counted.

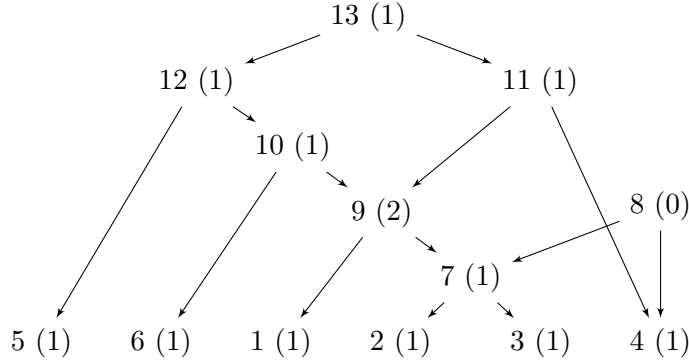


Figure 4.5: Reference counters of all vertices after `prepare_vertices`.

## 4.5 Extract Skolem/Herbrand Functions

The extraction of Skolem (resp. Herbrand) functions is based on the algorithm described in [2]. For all existentially (resp. universally) quantified variables, QRPcert constructs so-called RFAO stacks, which represent the partially ordered sets obtained by Definition 3.3 (resp. Definition 3.4) in reversed order. Unlike [2], in QRPcert we treat Skolem and Herbrand functions dually, i.e., we employ one extraction algorithm for both cases. Further, for constructing RFAO stacks we consider the vertices in  $V_p$  only.

QRPcert maintains two sets of literals  $L_s$  and  $L_r$  for representing a constraint  $C$ , where  $L_s$  is the set of literals that we obtain by  $\text{red}(C)$ . The set of reduced literals  $L_r$ , on the contrary, contains all literals of  $C$  that are eliminated via  $\text{red}(C)$ . This enables QRPcert to easily distinguish between the literals of  $C$  that are obtained by resolution and the literals that are eliminated by the subsequent application of  $\text{red}(C)$ .

*Example 4.1.* Given the proof DAG in Figure 4.3, consider the constraint of vertex 3, where we divide the literals  $\{-5, -6, 7\}$  into  $L_s = \{-5, -6\}$  and  $L_r = \{7\}$ . In contrast, vertex 11 with the literals  $\{-2, 3\}$  is divided into  $L_s = \{-2, 3\}$  and  $L_r = \{4, -7\}$  as literals 4 and  $-7$  are eliminated by universal-reduction after resolving vertices 4 and 9.

Figure 4.6 illustrates the algorithm for extracting Skolem (resp. Herbrand) functions as implemented in QRPcert. Algorithm `extract_functions` uses the previously computed reference counters of a vertex  $v$  for indicating the number of parent vertices that have to be visited before  $v$  can be processed.

```

1  function extract_functions ()
2  {
3    STACK s
4    VERTEX v
5    push (s, empty_vertex)
6
7    while not is_empty (s)
8    {
9      v ← pop (s)
10     decr_ref_cnt (v)
11
12     if get_ref_cnt (v) > 0
13       continue
14
15     vid ← get_id (v)
16     foreach l in get_reduced_literals (v)
17     {
18       var ← lit2var (l)
19       rfao ← get_rfao_stack (var)
20
21       if is_negated (l)
22         push (rfao, -vid)
23       else
24         push (rfao, vid)
25     }
26
27     foreach a in get_antecedents (v)
28       push (s, a)
29   }
30 }

```

Figure 4.6: Extract Skolem (resp. Herbrand) functions from proof DAG.

Therefore, each time vertex  $v$  is visited, one of its parent vertices has been processed and hence, we decrement the reference counter of  $v$  by one and check its state. If the reference counter is greater than 0, it indicates that there are still parent vertices of vertex  $v$  yet to be processed, and we skip  $v$  for now. In case the reference counter of  $v$  becomes 0, we know that all parent vertices have been processed and thus, we are allowed to proceed with vertex  $v$ . We then check for each literal  $l$  in  $L_r$  of vertex  $v$  (function `get_reduced_literals`) if it is either a positive or negative occurrence of variable `var` and based on that we push either the vertex id or its negation onto the RFAO stack of `var`. Note that a negative vertex id indicates that

the resp. constraint has to be negated when constructing the RFAO formula.

Finally, we push the antecedents of vertex  $v$  onto the visit stack  $\mathbf{s}$  and continue with the next vertex to be visited. Algorithm `extract_functions` terminates after all vertices in  $V_p$  have been processed. The resulting RFAO stacks of the existentially (resp. universally) quantified variables correspond to the partially ordered sets obtained by Definition 3.3 (resp. Definition 3.4), but in reversed order.

In the following, we describe the extraction process applied to the example proof DAG in Figure 4.3 step by step.

*Example 4.2.* Figure 4.7 illustrates the algorithm's processing order of the vertices, where processed vertices are denoted in black, and gray otherwise. We start with the empty vertex (vertex 13) and decrement its reference counter, which immediately becomes 0. We identify literals 1 and -2 to be universally-reduced from vertex 13 after resolving vertex 12 and 11, and push 13 and -13 onto the RFAO stacks of variables 1 and 2, respectively. After that, we continue with the first antecedent of vertex 13 (vertex 12), which we are allowed to process immediately as its single parent vertex is already processed. Vertex 12 is obtained by resolving vertices 5 and 10, and eliminating the literals 4 and -7 by applying universal-reduction to the resolvent. Hence, we push vertex id 12 and -12 onto the RFAO stacks of variables 4 and 7 respectively, and continue with vertex 10. Neither vertex 10 nor its first antecedent (vertex 6) is universally-reduced and we continue with its second antecedent (vertex 9). The reference counter of vertex 9 does not become 0 because one of its parents (vertex 11) is not yet processed and thus, for now we skip vertex 9 and continue with vertex 5, which is also not universally-reduced. In order to obtain vertex 11, the literals 4 and -7 are eliminated after resolving vertex 9 and 4 and thus, we push vertex id 11 and -11 onto the RFAO stacks of variables 4 and 7, respectively. Vertex 4 is not universally-reduced and we continue with vertex 9. The reference counter of vertex 9 becomes 0 (as all parents of vertex 9 are processed) and we continue with its first antecedent (vertex 7) as vertex 9 is not universally-reduced. Vertex 7 is also not universally-reduced and we proceed with vertex 3, where literal 7 is eliminated by universal-reduction before resolving vertices 2 and 3. Hence, we push vertex id 3 onto the RFAO stack of variable 7. After processing vertices 2 and 1, which are not universally-reduced, the algorithm terminates.

The topological order in which the vertices of the proof DAG were traversed is defined by the sequence (13, 12, 10, 6, 5, 11, 4, 9, 7, 3, 2, 1), which complies to the reversed topological order of Definition 3.1. Table 4.1 illustrates the state of the RFAO stacks of all universally quantified variables during the extraction process of the example described above. Note that steps, where the RFAO stacks of the variables are not modified, are omitted.

Step	Vertex	Reduced Literals	RFAO Stacks			
			1	2	4	7
1	13	1, -2	13	-13		
2	12	4, -7	13	-13	12	-12
6	11	4, -7	13	-13	12, 11	-12, -11
10	3	7	13	-13	12, 11	-12, -11, 3

Table 4.1: State of the RFAO stacks during the extraction process.

Given the complete RFAO stack of a variable, we are able to construct the corresponding RFAO formula, which represents the Skolem (resp. Herbrand) function of the corresponding existentially (resp. universally) quantified variable. For the construction of RFAO formulas, we consider only those literals of a constraint that are not eliminated by universal- (resp. existential-) reduction.

*Example 4.3.* Given the RFAO stacks of the universally quantified variables 1, 2, 4 and 7, we are able to construct the corresponding Herbrand functions by applying the construction rule of a RFAO formula as follows.

$$\begin{aligned}
 f_1 &= v_{13} = \perp \\
 f_2 &= \neg v_{13} = \top \\
 f_4 &= v_{11} \wedge v_{12} = (-2 \vee 3) \wedge (1 \vee -3) \\
 f_7 &= v_3 \wedge (\neg v_{11} \vee \neg v_{12}) = (-5 \vee -6) \wedge ((2 \wedge -3) \vee (-1 \wedge 3))
 \end{aligned}$$

Note that in contrast to Definition 3.4, the RFAO stacks are in reversed order. Hence, we construct the corresponding RFAO formula starting with the vertex on top of the RFAO stack.

In the following, we demonstrate what happens if we ignore the reference counter of a vertex while traversing the proof DAG, i.e., we assume that the reference counter of a vertex becomes 0 as soon as it is visited. Due to this assumption, at step (3) in Figure 4.7 vertex 9 is not skipped after visiting vertex 10. Thus, we continue with vertex 7, 3, 2 and 1 before vertex 6 is processed. After algorithm `extract_functions` terminates, the vertices of the proof DAG were processed in the order (13, 12, 10, 9, 7, 3, 2, 1, 6, 5, 11, 4). Hence, the order of the vertices on the RFAO stack of variable 7 changes to (-12, 3, -11), where  $11 < 3$  violates the partial order relation defined in Definition 3.1 as vertex 3 is not a successor of 11. Thus, the resulting Herbrand function is invalid and not a correct representation of variable 7 and the validation of the certificate does not succeed. Therefore, it is important to process the vertices in a proof DAG in a valid topological order, as otherwise the extracted Skolem (resp. Herbrand) functions may not be correct.

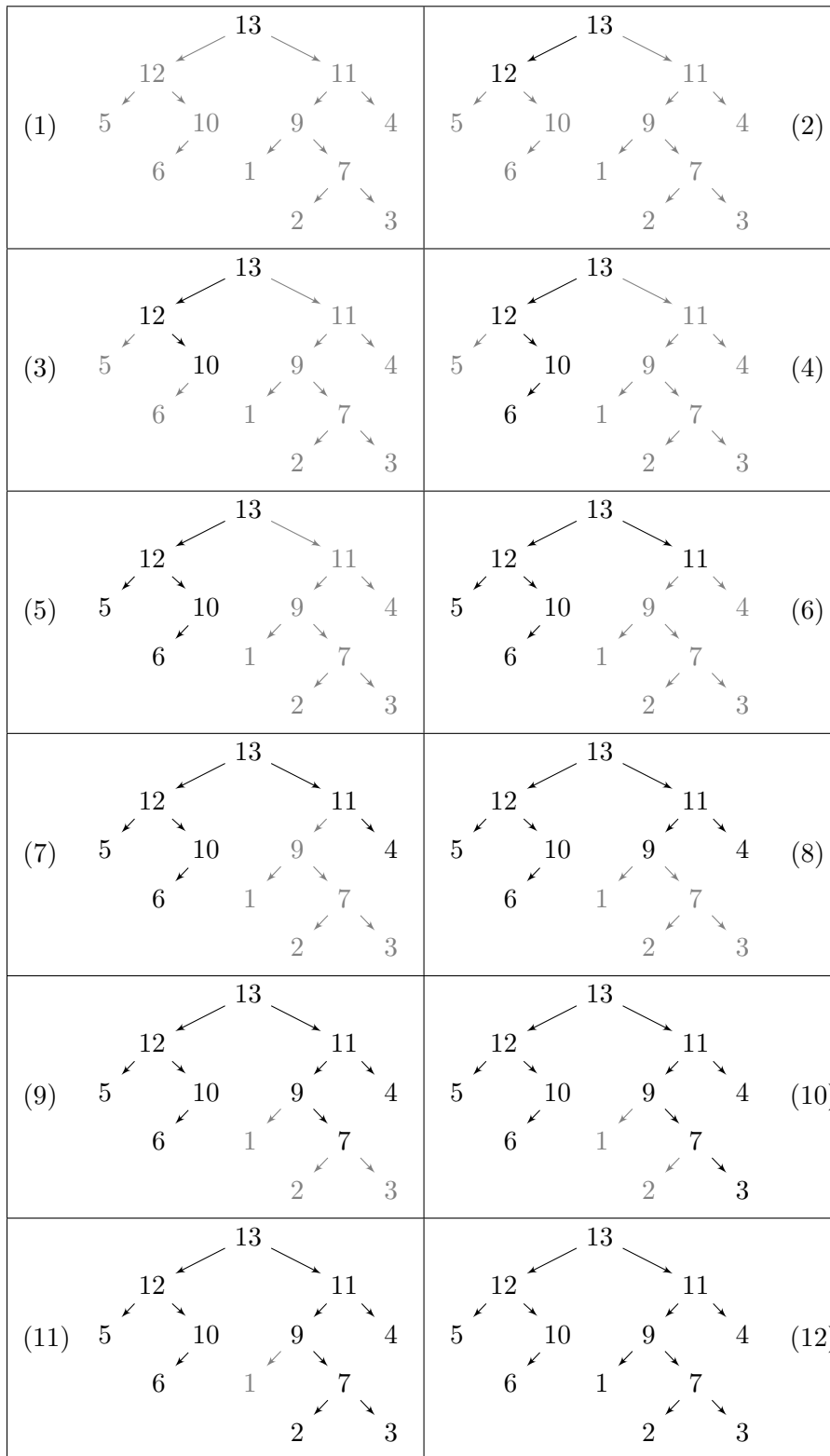


Figure 4.7: Proof DAG processing order in QRPcert.

## 4.6 Simplify Skolem/Herbrand Functions

If simplification is enabled, QRPCert tries to minimize Skolem (resp. Herbrand) functions by propagating Skolem (resp. Herbrand) constants that were derived during the extraction process.

Each subformula on the RFAO stack has either the structure  $(\text{clause} \wedge (\phi))$  or  $(\text{cube} \vee (\phi))$ , where the innermost subformula  $(\phi)$  is a constraint. Hence, there are two ways of simplifying a RFAO formula, considering that either the left part (`clause` resp. `cube`) or the right part  $(\phi)$  may be simplified to a Boolean constant.

The basic simplification workflow of algorithm `simplify_rfao` is illustrated in Figure 4.8. The algorithm performs simplification on the RFAO structure level, i.e., it exploits the specific structure of a RFAO formula for eliminating certain parts of the formula.

Given a variable `var`, we traverse the vertices on its RFAO stack from bottom to top, i.e., we start with the rightmost vertex of the RFAO formula as current vertex `v`. We simplify `v` (Figure 4.9) and obtain its truth value `val`, which is either  $\top$ ,  $\perp$  or `undef`. If function `simplify_vertex` returns `undef`, i.e., vertex `v` cannot be simplified to a constant and we skip `v`. Otherwise, we negate `val` in case vertex `v` occurs negated on the RFAO stack. We further check if `v` is the only vertex on the RFAO stack and if this is the case, we terminate the algorithm and return `val` as truth value for `var`.

If the RFAO stack contains more than one vertex, we try to simplify the formula in two phases. First, we consider the innermost subformula on the RFAO stack and simplify it with respect to its rightmost vertex (line 15-30). Second, in case the rightmost vertex of the innermost subformula cannot be simplified to a constant, we continue with the simplification of each subformula with respect to its leftmost vertex, starting with the innermost subformula (line 31-40).

The structure of the innermost subformula is defined by the next vertex of `v` (function `get_next_vertex`) i.e., the subformula left to `v` denoted by `v'`. In case `v'` is a clause, the structure of the innermost subformula is defined as  $(v' \wedge v)$ . If the truth value of vertex `v` (`val`) is  $\top$ , we eliminate `v` from the RFAO stack, as the truth value of the subformula depends on vertex `v'` only, and continue the simplification with the next vertex on the RFAO stack. Else, the truth value of the subformula depends on vertex `v` only and thus, `v'` is eliminated. We then repeat the simplification of the innermost subformula with vertex `v` and the next vertex on the RFAO stack. If vertex `v'` is a cube, the structure of the innermost subformula is defined as  $(v' \vee v)$ . Hence, if `val` is  $\top$ , we eliminate `v'` and again continue with vertex `v` in the next simplification iteration. In case `val` is  $\perp$ , `v` can be eliminated and we continue with the next vertex on the RFAO stack.

In case the truth value of the rightmost vertex is `undef`, we skip the vertex and simplify the subformula with respect to its left part, which is

```

1  function simplify_rfao (VARIABLE var)
2  {
3    rfao ← get_rfao_stack (var)
4    foreach v in rfao
5    {
6      val ← simplify_vertex (v)
7      if val = undef
8        continue
9      if is_negated (v)
10       val = ¬val
11     if get_num_vertices (rfao) = 1
12       return val
13
14     /* simplify innermost subformula */
15     if is_bottom_vertex (rfao, v)
16     {
17       v' = get_next_vertex (rfao)
18       if (is_clause (v') and val = ⊤) or
19         (is_cube (v') and val = ⊥)
20       {
21         remove_vertex (rfao, v)
22         continue with v'
23       }
24       elif (is_clause (v') and val = ⊥) or
25         (is_cube (v') and val = ⊤)
26       {
27         remove_vertex (rfao, v')
28         continue with v
29       }
30     }
31     if is_true_clause (v) or is_false_cube (v)
32     {
33       remove_vertex (rfao, v)
34     }
35     elif is_false_clause (v) or is_true_cube (v)
36     {
37       remove_all_vertices_below (rfao, v)
38       continue with v
39     }
40   }
41   return undef
42 }

```

Figure 4.8: Simplify RFAO stack of a variable var.

```

1 function simplify_vertex (VERTEX v)
2 {
3   foreach lit in get_literals (v)
4   {
5     val ← evaluate_literal (lit)
6
7     if val = undef
8     continue
9     if get_num_literals (v) = 1
10    return val
11
12    if is_clause (v) and val = ⊤
13    return ⊤
14    elif is_cube (v) and val = ⊥
15    return ⊥
16
17    remove_literal (v, lit)
18  }
19
20  if get_num_literals (v) = 0
21  {
22    if is_clause (v)
23    return ⊥ /* empty clause */
24    else
25    return ⊤ /* empty cube */
26  }
27  return undef
28 }

```

Figure 4.9: Simplify and evaluate a vertex.

now denoted by vertex  $v$ . Based on the construction rule of a RFAO formula, the structure of a subformula is either  $(v \wedge \phi)$  or  $(v \vee \phi)$ , depending on whether vertex  $v$  is a clause or a cube, respectively. Formula  $\phi$  denotes the subformula constructed from the vertices on the RFAO stack below vertex  $v$ . If  $v$  is a clause (resp. cube) evaluating to  $\top$  (resp.  $\perp$ ), we eliminate  $v$ . However, if vertex  $v$  is a clause (resp. cube) evaluating to  $\perp$  (resp.  $\top$ ), we eliminate subformula  $\phi$  (function `remove_all_vertices_below`), which results in a simplified innermost subformula with its rightmost vertex being  $v$ . We then repeat simplification and start with the first phase. In case the simplification of the RFAO stack of `var` was not successful, function `simplify_rfao` returns `undef`, which indicates that `var` is defined by a formula and not a constant.



The algorithm for simplifying vertices is illustrated in Figure 4.9. Given a vertex  $v$ , we traverse over all of its literals and evaluate each literal `lit` (function `evaluate_literal`). We obtain the truth value `val` of `lit`, which is either  $\top$ ,  $\perp$  or `undef`. If `val` is `undef`, we skip `lit` and continue with the next literal of  $v$ . Else, in case that given vertex  $v$  only contains one literal, the truth value of  $v$  depends on `lit` only and hence, we return `val`.

If vertex  $v$  contains more than one literal, we check if  $v$  evaluates to  $\top$  (resp.  $\perp$ ) under given truth value of `val`. Hence, we return  $\top$  (resp.  $\perp$ ) if  $v$  is a clause (resp. cube) and the truth value of `lit` is  $\top$  (resp.  $\perp$ ). If both cases do not apply, we can eliminate `lit` from vertex  $v$ , as `lit` is  $\perp$  (resp.  $\top$ ) and  $v$  is a clause (resp. cube). After evaluating all literals of vertex  $v$ , we finally check if  $v$  was simplified to an empty clause (resp. empty cube) and return  $\perp$  (resp.  $\top$ ) if this is the case.

Note that simplification in `QRPcert` is optional and is disabled by default due to the fact that it is time-consuming and not considered beneficial in its current implementation. For further details please refer to the experiments in Section 6.4.

*Example 4.4.* Given the RFAO stacks of variables 4 and 7 of Example 4.3, by applying simplification we obtain the following simplified Herbrand functions.

$$\begin{aligned} f_4 &= (\neg 2 \vee 3) \wedge (1 \vee -3) = 3 \wedge -3 \\ f_7 &= (-5 \vee -6) \wedge ((2 \wedge -3) \vee (\neg 1 \wedge 3)) = (-5 \vee -6) \wedge (-3 \vee 3) \end{aligned}$$

Note that simplification only propagates constants of variables and thus, subformulas like  $(-3 \vee 3)$  as in  $f_7$  are not further simplified.

## 4.7 Construct And-Inverter Graphs

A certificate generated by `QRPcert` consists of a set of AIGs (cf. Section 2.2), which represent the Skolem (resp. Herbrand) functions of the corresponding variables. Hence, we transform each RFAO formula that we obtained during the extraction phase into an AIG. For that purpose, we implemented a lightweight AIG library, which employs structural hashing of two-input AND-gates, and currently supports the ASCII AIGER format.

By default, `QRPcert` maintains all constructed AIGs in memory in order to share isomorphic AND-gates among all of them, which considerably reduces the overall number of AND-gates of a certificate (cf. Section 6.3). Optionally, in case that not enough memory is available, `QRPcert` supports incremental AIG construction, where the constructed AIGs are immediately written to the output file if they exceed a given size limit. Note that incremental AIG construction may increase the size of the resulting certificate, as isomorphic AND-gates are only shared among AIGs that are kept in memory.

```

1  function rfao_to_aig (VARIABLE var)
2  {
3    rfao ← get_rfao_stack (var)
4    v ← remove_bottom_vertex (rfao)
5    aig ← vertex_to_aig (v)
6
7    foreach v in rfao
8    {
9      if is_clause (v)
10     {
11       aig ← aig_and (vertex_to_aig (v), aig)
12     }
13     else
14     {
15       aig ← aig_and (aig_not (vertex_to_aig (v)),
16                    aig_not (aig))
17       aig ← aig_not (aig)
18     }
19   }
20   return aig
21 }

```

Figure 4.10: Construct AIG from RFAO stack of `var`.

Figure 4.10 describes the top-level algorithm for constructing AIGs from the RFAO stack of a variable. Given a variable `var`, we construct an AIG starting from the innermost subformula of the RFAO stack and initially start with the rightmost vertex `v` of the RFAO formula. We translate vertex `v` into an AIG `aig` and traverse the remaining vertices on the RFAO stack from bottom to top. The structure of each subformula depends on whether vertex `v` is a clause or a cube. Hence, in case `v` is a clause, we first translate `v` into an AIG and then create a new AND-gate with `v` and `aig` as inputs. However, if vertex `v` is a cube, we have to apply De Morgan's law in order to express the resulting subformula  $(v \vee \text{aig})$  with AND-gates and negations only. Therefore, we translate vertex `v` into an AIG and create a new AND-gate with both inputs `v` and `aig` negated. We negate the resulting `aig` and continue with the next vertex on the RFAO stack. If all vertices on the RFAO stack are processed, function `rfao_to_aig` terminates and returns the AIG representing the Skolem (resp. Herbrand) function of variable `var`.

The algorithm for constructing AIGs from vertices is depicted in Figure 4.11. Given a vertex `v`, we consider its existentially- (resp. universally-) reduced form only. Hence, we use only the literals of the set  $L_s$  of vertex `v` (function `get_static_literals`) for constructing an AIG. In case vertex `v`

```

1  function vertex_to_aig (VERTEX v)
2  {
3    aig ← init_aig ()
4
5    foreach lit in get_static_literals (v)
6    {
7      if is_clause (v)
8        lit = aig_not (lit)
9
10     if is_first_literal (lit)
11       aig ← lit
12     else
13       aig ← aig_and (lit, aig)
14   }
15
16   if is_clause (v)
17     return aig_not (aig)
18
19   return aig
20 }

```

Figure 4.11: Construct AIG from vertex  $v$ .

is a clause, we have to apply De Morgan's law and thus, we negate `lit`. If `lit` is the first literal to be processed, we initialize `aig` with `lit` and continue with the next literal. Else, we create a new AND-gate with the inputs `lit` and `aig`. After all literals are processed, we negate the resulting `aig` in case  $v$  is a clause (De Morgan) and return it. In case  $v$  is a cube, we return `aig` as-is.

The order of the literals of a vertex is important for constructing AIGs as it has an impact on the number of AND-gates shared, which we demonstrate with the following example.

*Example 4.5.* We consider the propositional formulas  $\phi_1$  and  $\phi_2$ , which are defined as follows.

$$\begin{aligned}\phi_1 &= c \vee b \vee \neg d \vee \neg e \\ \phi_2 &= \neg d \vee c \vee a \vee b\end{aligned}$$

If we construct an AIG for both formulas, we obtain two separate AIGs (Figure 4.12a) that do not share any AND-gate even though they have literals  $b$ ,  $c$  and  $\neg d$  in common. However, if we sort the literals of  $\phi_1$  and  $\phi_2$  with respect to some predefined variable order (e.g.,  $a < b < c < d < e$ ) and construct an AIG for both formulas, we obtain the AIG depicted in Figure 4.12b, where we are able to share two AND-gates among  $\phi_1$  and  $\phi_2$ .

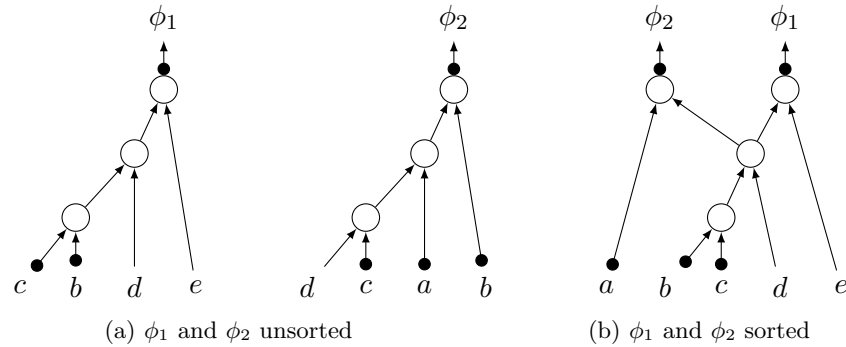


Figure 4.12: Impact of literals order on AND-gate sharing.

As illustrated in Figure 4.12, a suboptimal order of the literals may considerably affect the overall number of AND-gates shared. Hence, QRPCert sorts all literals of a vertex by their internal variable index while building the proof DAG.

After the AIGs of all Skolem (resp. Herbrand) functions are constructed, the certificate of satisfiability (resp. unsatisfiability) is complete and we write its ASCII AIGER representation to the output file. Each primary output of the certificate AIG corresponds to a Skolem (resp. Herbrand) function, whereas the primary inputs are universally (resp. existentially) quantified variables.

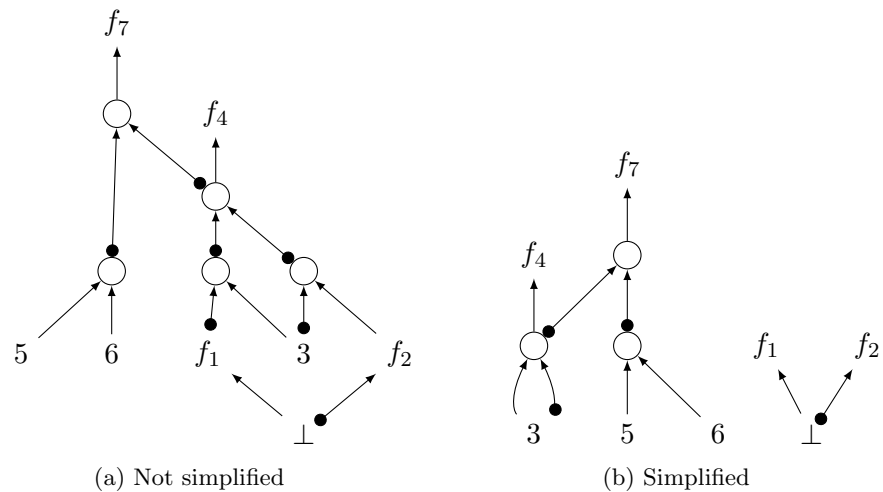


Figure 4.13: AIG representation of the certificate extracted by QRPCert from the example clause resolution trace in Figure 4.2b, without (a) and with (b) simplification applied.

*Example 4.6.* Given the Herbrand functions (not simplified) of variables 1, 2, 4 and 7 from Example 4.3, the resulting certificate AIG is depicted in Figure 4.13a. The certificate AIG with the simplified Herbrand functions is illustrated in 4.13b.

The representations of the AIGs in Figure 4.13a and 4.13b in ASCII AIGER format are depicted in Figure 4.14a and 4.14b, respectively. The first line of the file starts with format identifier string `aag`, which indicates that the file is in ASCII AIGER format. The subsequent five integers denote the maximum variable index, the number of inputs, the number of latches, the number of outputs and the number of AND-gates used. The header line is followed by the definitions of inputs, latches, outputs and AND-gates in the same order as defined in the header with one definition per line. Each AND-gate is defined by a triple of literals with the first literal denoting the output and the second and third denoting the two inputs of the AND-gate. In the AIGER format, a variable is transformed into a literal by multiplying it by two. In case a literal is negated its least significant bit is flipped to one. The Boolean constants  $\top$  and  $\perp$  are represented as 1 and 0, respectively.

<pre> aag 12 5 0 4 7 6 10 12 16 18 2 4 8 14 2 1 0 4 1 1 20 3 6 22 7 4 8 23 21 24 10 12 14 25 9 (a) Not simplified </pre>	<pre> aag 10 5 0 4 5 6 10 12 16 18 2 4 8 14 2 0 0 4 1 1 8 7 6 20 10 12 14 21 9 (b) Simplified </pre>
--------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------

Figure 4.14: Certificate of unsatisfiability generated by QRPcert for the example input formula in Figure 4.2a in ASCII AIGER format, without (a) and with (b) simplification applied.

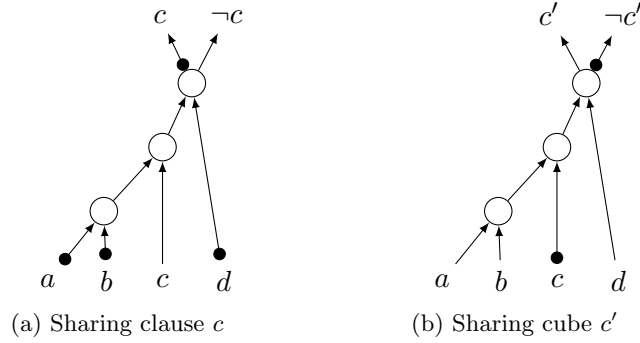


Figure 4.15: Clause (resp. cube) sharing in AIGs.

*Example 4.7.* The AIG in Figure 4.14a has a maximum variable index of 12 and further has five inputs, no latches, four outputs and a total of seven AND-gates. The first five lines after the header (lines 2-6) define the input literals denoting the existentially quantified variables 3, 5, 6, 8 and 9, respectively. The primary outputs of the AIG are defined by the output literals 2, 4, 8 and 14 (lines 7-10), which represent the Herbrand functions of variables 1, 2, 4 and 7, respectively. The structure of the Herbrand functions is defined by the two-input AND-gates defined in lines 11-17.

Note that one advantage of employing AIGs with structural hashing to represent RFAO formulas is the fact that a vertex has to be constructed only once (if the certificate AIG is not constructed incrementally), even if the vertex occurs negated. This is due to the fact, that the AIG representations of a vertex and its negation share the same AIG structure except that the output of the latter is negated.

*Example 4.8.* Given a clause  $c$  and a cube  $c'$ , which are defined as follows.

$$\begin{aligned} c &= a \vee b \vee \neg c \vee d \\ c' &= a \wedge b \wedge \neg c \wedge d \end{aligned}$$

The AIG representation of  $c$  and  $\neg c$  (resp.  $c'$  and  $\neg c'$ ) share the same AIG structure, which is illustrated in Figure 4.15a (resp. 4.15b).

In this chapter, we presented QRPCert, a tool for extracting Skolem/Herbrand function-based QBF certificates from Q-resolution proofs in QRP format. We represent the extracted set of Skolem (resp. Herbrand) functions as AIGs, which are simplified by employing structural hashing. The generated certificates are represented in ASCII AIGER format. In the following, we describe the process of certificate validation in more detail.

## Chapter 5

# CertCheck: Certificate Validation

CertCheck is a tool that transforms a given input formula in QDIMACS format into an AIG and merges the result with the corresponding certificate in ASCII AIGER format extracted by QRPCert. The resulting AIG is translated into a propositional formula in CNF via Tseitin transformation [48], which is then used for validating the correctness of the certificate. In the following, we describe the tool CertCheck and the certificate validation process in more detail.

### 5.1 Overview

CertCheck requires two different kinds of input: an input formula in QDIMACS format and its certificate of (un)satisfiability in ASCII AIGER format as extracted by QRPCert. The input formula is transformed into an AIG and then merged with given certificate by substituting each existentially (resp. universally) quantified input variable with its Skolem (resp. Herbrand) function. The resulting AIG is constructed from universally (resp. existentially) quantified input variables only. The correctness of a certificate of satisfiability (resp. unsatisfiability) is validated by checking if the resulting AIG is tautological (resp. unsatisfiable). Hence, we transform the AIG into a propositional formula in CNF using Tseitin transformation, write it to an output file in DIMACS<sup>1</sup> format, and check with a SAT solver if the formula is unsatisfiable. Note that in case of a certificate of satisfiability, we have to check if the merged AIG generated by CertCheck is tautological. In order to show that a formula  $\phi$  is tautological, we check if its negation  $\neg\phi$  is unsatisfiable. Hence, we negate the output of the merged AIG and translate it into a propositional formula in CNF.

---

<sup>1</sup>[www.satlib.org/Benchmarks/SAT/satformat.ps](http://www.satlib.org/Benchmarks/SAT/satformat.ps)

```

1 function cnf_to_aig (MATRIX m)
2 {
3   aig ← init_aig ()
4
5   foreach c in get_clauses (m)
6   {
7     if is_first_clause (c)
8       aig ← clause_to_aig (c)
9     else
10      aig ← aig_and (clause_to_aig (c), aig)
11   }
12
13   if is_sat_certificate ()
14     return aig_not (aig)
15
16   return aig
17 }

```

Figure 5.1: Transform matrix  $m$  in CNF into an AIG.

The top-level algorithm for transforming the input formula into an AIG is depicted in Figure 5.1. Given the matrix  $m$  of a QBF in PCNF, algorithm `cnf_to_aig` transforms each clause  $c$  into an AIG (function `clause_to_aig`) by applying De Morgan’s law (as in algorithm `vertex_to_aig` in Figure 4.11). In case clause  $c$  is the first clause to be processed, we initialize `aig` with the AIG of  $c$ . Else, we create a new AND-gate with both the AIG of  $c$  and the AIG built so far (`aig`) as inputs. After all clauses are processed and if a certificate of satisfiability is given, we negate the output of the resulting AIG `aig` and return it. In case of a certificate of unsatisfiability, we return `aig` as-is.

Figure 5.2 illustrates the AIG that is obtained by transforming the example input formula in Figure 4.2a into an AIG and merging it with the simplified certificate AIG in Figure 4.13b. The dashed edges in the AIG indicate that an universally quantified input variable is substituted by its Herbrand function in the certificate. Note that we did not reuse inputs 3, 5 and 6 of the input formula AIG in the certificate AIG, which is for illustration purpose only. The representation of the AIG in Figure 5.2 in ASCII AIGER format is depicted in Figure 5.3a.

We translate the merged AIG into a propositional formula in CNF via the standard Tseitin encoding and yet do not support optimizations like Plaisted-Greenbaum [37]. Hence, each AND-gate of the merged AIG with



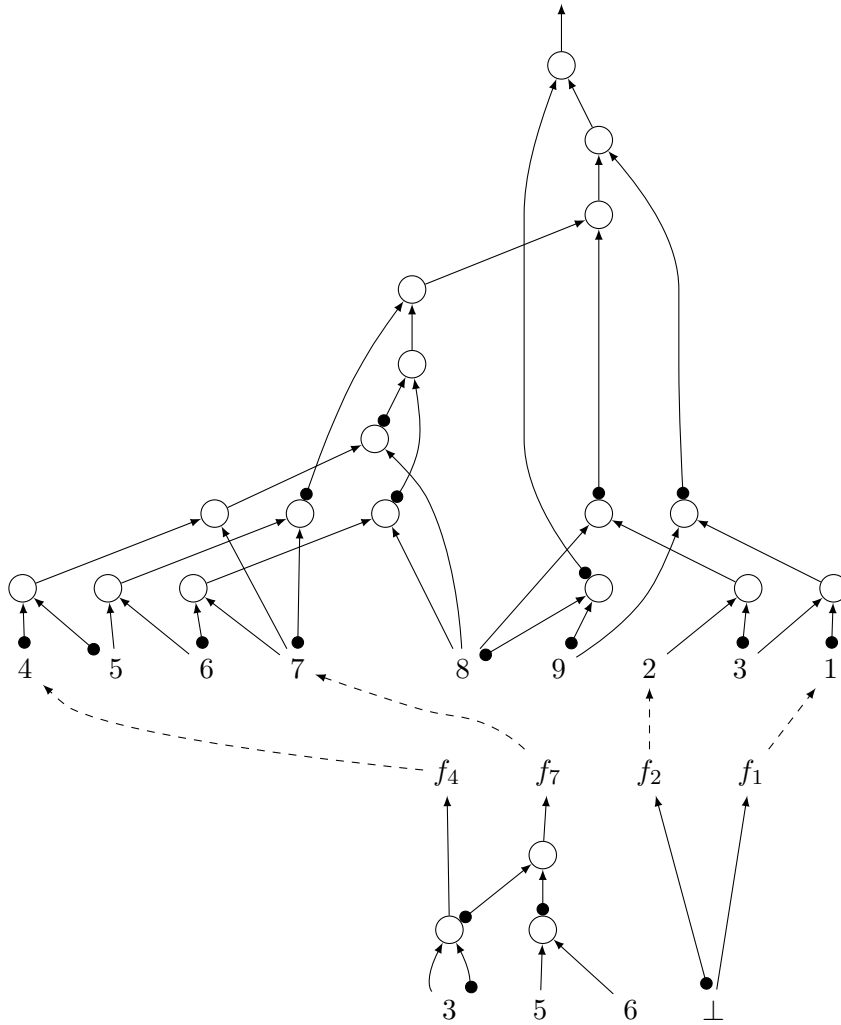


Figure 5.2: Merging AIG of input formula (Figure 4.2a) with certificate AIG (Figure 4.13b).

an output  $out$  and two inputs  $in_1$  and  $in_2$  is encoded as follows.

$$\begin{aligned}
& out = in_1 \wedge in_2 \\
& \equiv out \leftrightarrow (in_1 \wedge in_2) \\
& \equiv (out \rightarrow (in_1 \wedge in_2)) \wedge ((in_1 \wedge in_2) \rightarrow out) \\
& \equiv (\neg out \vee (in_1 \wedge in_2)) \wedge (\neg in_1 \vee \neg in_2 \vee out) \\
& \equiv (\neg out \vee in_1) \wedge (\neg out \vee in_2) \wedge (\neg in_1 \vee \neg in_2 \vee out)
\end{aligned}$$

The output of the merged AIG is encoded as a single clause with a single literal, which is negated in case that the output of the AIG is negated. Further, the Boolean constant  $\top$  is also encoded as a single clause with a single literal.

The Tseitin encoding of the AIG in Figure 5.2 in DIMACS format is illustrated in Figure 5.3b, where each AND-gate in Figure 5.3a is encoded with three clauses. As an example, consider line "14 39 9" in Figure 5.3a, which represents the AND-gate  $7 = (\neg 19 \wedge \neg 4)$ . By applying the transformation described above, we obtain the three clauses "-7 -19 0", "-7 -4 0" and "7 19 4 0" in Figure 5.3b, respectively. Further, clause "37 0" and "36 0" represent the Boolean constant  $\top$  and the output of the AIG, respectively.

## 5.2 Certificate Validation

We validate the correctness of a certificate of satisfiability (resp. unsatisfiability) by checking if the Skolemization (resp. Herbrandization) of the input formula is tautological (resp. unsatisfiable). The AIG that we obtain by merging an input formula  $\Phi$  with its certificate of satisfiability (resp. unsatisfiability) represents the Skolemization (resp. Herbrandization) of  $\Phi$  and is denoted as  $\Phi_S$  (resp.  $\Phi_H$ ). Note that  $\Phi_S$  (resp.  $\Phi_H$ ) contains universal (resp. existential) variables only.

In case of a certificate of satisfiability, we have to check if the extracted Skolem functions represent valid assignments to the existentially quantified variables. Therefore, we check if  $\Phi_S$  is satisfiable under all possible assignments, which is only the case if  $\Phi_S$  is tautological as it contains universally quantified variables only. Hence, we negate the output of the AIG representing  $\Phi_S$  and translate it into a propositional formula in CNF. If the resulting CNF is unsatisfiable, we conclude that  $\Phi_S$  is tautological and given certificate of satisfiability is valid. Otherwise, it is invalid.

In case of a certificate of unsatisfiability, we have to check if the extracted Herbrand functions yield an assignment to the universal variables such that  $\Phi$  evaluates to false for any assignment to the existential variables. Formula  $\Phi$  is unsatisfiable if and only if  $\Phi_H$  is unsatisfiable. Hence, we translate the AIG representing  $\Phi_H$  into a propositional formula in CNF. We conclude that given certificate of unsatisfiability is valid if the resulting CNF is unsatisfiable, and invalid otherwise.

aag 36 5 0 1 22	p cnf 37 68	
6	-1 -37 0	-26 25 0
10	-1 -37 0	26 7 -25 0
12	1 37 37 0	-27 -3 0
16	-2 37 0	-27 2 0
18	-2 37 0	27 3 -2 0
72	2 -37 -37 0	-28 -8 0
2 0 0	-4 -3 0	-28 27 0
4 1 1	-4 3 0	28 8 -27 0
8 7 6	4 3 -3 0	-29 -1 0
38 10 12	-19 5 0	-29 3 0
14 39 9	-19 6 0	29 1 -3 0
40 11 9	19 -5 -6 0	-30 9 0
42 14 40	-7 -19 0	-30 29 0
44 16 42	-7 -4 0	30 -9 -29 0
46 13 14	7 19 4 0	-31 -9 0
48 16 46	-20 -5 0	-31 -8 0
50 10 12	-20 -4 0	31 9 8 0
52 15 50	20 5 4 0	-32 -24 0
54 7 4	-21 7 0	-32 -22 0
56 17 54	-21 20 0	32 24 22 0
58 3 6	21 -7 -20 0	-33 -26 0
60 18 58	-22 8 0	-33 32 0
62 19 17	-22 21 0	33 26 -32 0
64 49 45	22 -8 -21 0	-34 -28 0
66 53 64	-23 -6 0	-34 33 0
68 57 66	-23 7 0	34 28 -33 0
70 61 68	23 6 -7 0	-35 -30 0
72 63 70	-24 8 0	-35 34 0
(a)	-24 23 0	35 30 -34 0
	24 -8 -23 0	-36 -31 0
	-25 5 0	-36 35 0
	-25 6 0	36 31 -35 0
	25 -5 -6 0	37 0
	-26 -7 0	36 0
	(b)	

Figure 5.3: Merged AIG represented in ASCII AIGER format (a) and translated into CNF in DIMACS format (b).

Note that during the whole certification process, i.e., from certificate extraction to the generation of the propositional formula, all variables of the input formula are preserved and thus, retraceable. This can be helpful for tracking down the cause of incorrect certificates, as a SAT solver may provide a satisfying assignment, which can be used to analyze the extracted certificate.

In this chapter, we discussed the process of certificate validation, where we used `CertCheck` to merge the input formula with the corresponding certificate extracted by `QRPcert`. The correctness of the certificate is validated by checking with a SAT solver if the resulting propositional formula is unsatisfiable.

## Chapter 6

# Experimental Results

We implemented a framework to certify and validate the results of DepQBF, a dependency-aware search-based QBF solver for QBF in PCNF. It consists of a chain of loosely coupled stand-alone tools on top of DepQBF, which support proof extraction and checking (QRPcheck [36]) as well as certificate extraction (QRPcert) and validation (CertCheck and PicoSAT [6]). The workflow of the certification framework is illustrated in Figure 6.1.

Given a QBF in QDIMACS format, DepQBF records a trace of all Q-resolution sequences that are derived during the solving process, which is described in [36] in more detail. The resulting trace in QRP format is used by QRPcheck to extract and check the corresponding Q-resolution proof of satisfiability (resp. unsatisfiability). The extracted Q-resolution proof is used by QRPcert to generate a QBF certificate in AIGER format, as described in Chapter 4. We then use CertCheck to merge the certificate with the input formula in order to generate a propositional formula in CNF for validating the correctness of the certificate (Chapter 5). Finally, we employ the SAT solver PicoSAT to check if the resulting propositional formula is unsatisfiable.

In the following experiments, we focus on certification and validation only and therefore do not consider the results of QRPcheck. An in-depth evaluation of the results of QRPcheck is done in [36].

### 6.1 Overview

We conducted our experiments on the benchmark sets of the QBF competitions 2008 (QBFEVAL'08) and 2010 (QBFEVAL'10), which consist of 3326 and 568 formulas, respectively<sup>1</sup>. We applied the certification framework on those 1228 and 362 formulas of the benchmark sets that were solved by DepQBF (with tracing) within 900 seconds with advanced dependency schemes disabled. All experiments were performed on 2.83 GHz Intel Core 2 Quad machines each equipped with 8 GB of main memory and running

---

<sup>1</sup>Available at [http://www.qbflib.org/index\\_eval.php](http://www.qbflib.org/index_eval.php)

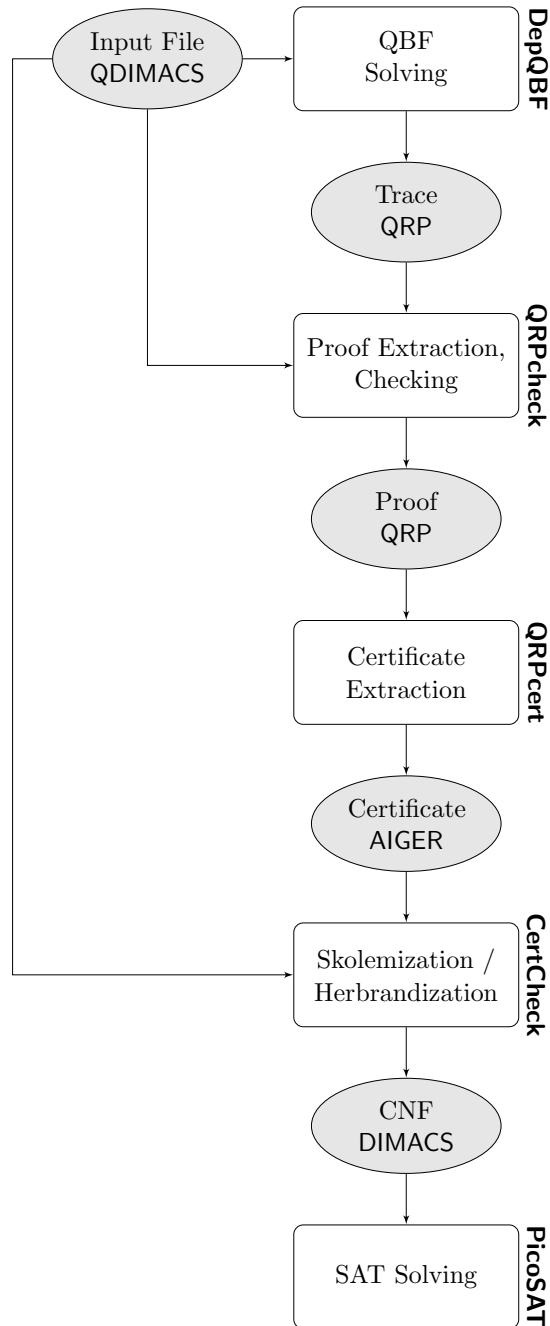


Figure 6.1: Certification workflow for experiments.

Family	Instances			Time Solv. [s]			Time Cert. [s]		
	sv	ex	va	total	avg.	med.	total	avg.	med.
Abduction	48	48	48	48.3	1.0	0.0	14.4	0.3	0.0
Adder	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
blackb*-01X-*	43	36	36	531.3	14.8	0.2	56.9	1.6	0.2
blackb*_design	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
Blocks	4	3	3	11.4	3.8	0.1	305.7	101.9	0.0
BMC	12	12	12	34.4	2.9	0.5	55.6	4.6	2.6
Chain	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
circuits	2	2	2	30.2	15.1	15.1	0.2	0.1	0.1
conformant	5	3	3	24.7	8.2	0.1	118.9	39.6	0.2
Connect4	8	8	8	40.8	5.1	0.1	7.4	0.9	0.8
Counter	2	2	2	229.6	114.8	114.8	4.9	2.5	2.5
Debug	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
evader-pursuer	10	9	9	719.5	79.9	0.7	89.0	9.9	1.0
FPGA*FAST	2	2	2	0.2	0.1	0.1	0.5	0.2	0.2
FPGA*SLOW	1	1	0	0.0	0.0	0.0	0.0	0.0	0.0
Impl	1	1	1	0.0	0.0	0.0	0.0	0.0	0.0
jmc_quant	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
mqm	128	121	70	1227.0	17.5	0.6	8.0	0.1	0.1
pan	24	21	14	1785.3	127.5	9.0	3014.4	215.3	0.7
Rintanen	1	1	1	12.8	12.8	12.8	1.7	1.7	1.7
Sakallah	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
Scholl-Becker	11	10	10	30.4	3.0	0.2	15.1	1.5	0.5
Sorting_net	6	5	4	187.4	46.9	3.7	89.7	22.4	18.2
SzymanskiP	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
tipdiam	3	1	1	0.0	0.0	0.0	0.0	0.0	0.0
tipfixpoint	9	9	9	2.2	0.2	0.1	3.5	0.4	0.3
Toilet	40	40	38	23.2	0.6	0.0	682.1	18.0	0.0
VonNeumann	2	2	2	4.9	2.5	2.5	21.9	11.0	11.0
<b>total</b>	<b>362</b>	<b>337</b>	<b>275</b>	<b>4944</b>	<b>18.0</b>	<b>0.2</b>	<b>4490</b>	<b>16.3</b>	<b>0.1</b>

Table 6.1: QBFEVAL'10 family overview of certification workflow.

Ubuntu 9.04. The time and memory limits for the whole certification workflow were set to 1800 seconds and 7 GB, respectively. Note that for the following experiments, we did not enable simplification in QRPCert. However, the results with simplification enabled are summarized in Section 6.4.

Table 6.2 and 6.1 show the aggregated results of the QBFEVAL'08 and QBFEVAL'10 benchmark set grouped by family (column 1). Columns 2-4 contain the number of instances solved by DepQBF ("sv"), the number of certificates extracted by QRPCert ("ex"), and the number of certificates validated by PicoSAT ("va"). We omit the number of CNFs generated by CertCheck as it equals the number of certificates extracted. Columns 5-7 resp. columns 8-10 indicate the runtime required for solving and certifying all instances that were successfully validated by PicoSAT within given time and memory constraints. Note that the time required for solving includes tracing (DepQBF), whereas the runtime of certification includes certificate extraction (QRPCert) and certificate validation (CertCheck and PicoSAT).

Family	Instances			Time Solv. [s]			Time Cert. [s]		
	sv	ex	va	total	avg.	med.	total	avg.	med.
Abduction	284	283	283	562.4	2.0	0.1	916.0	3.2	0.1
Adder	5	5	5	1.4	0.3	0.0	5.8	1.2	0.3
blackb*-01X-*	314	279	279	3377.7	12.1	0.1	410.4	1.5	0.1
blackb*_design	1	1	1	0.4	0.4	0.4	0.3	0.3	0.3
Blocks	11	10	10	129.0	12.9	0.7	1511.1	151.1	0.1
BMC	81	80	80	3277.6	41.0	0.4	256.1	3.2	0.5
Chain	10	6	3	6.9	2.3	1.9	1637.6	545.9	259.0
circuits	5	4	4	3.0	0.8	0.6	0.0	0.0	0.0
conformant	11	9	8	197.0	24.6	2.2	97.9	12.2	0.4
Counter	10	10	10	130.9	13.1	0.0	4.6	0.5	0.0
Debug	1	1	0	0.0	0.0	0.0	0.0	0.0	0.0
DFlipFlop	10	10	10	1.9	0.2	0.1	5.4	0.5	0.2
evader-pursuer	16	14	14	332.8	23.8	0.2	23.9	1.7	1.1
FPGA*FAST	5	5	5	0.7	0.1	0.1	2.7	0.5	0.1
FPGA*SLOW	3	3	1	9.1	9.1	9.1	0.1	0.1	0.1
Impl	10	10	10	0.0	0.0	0.0	0.0	0.0	0.0
irqkeapclte	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jmc_quant	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
MutexP	3	3	2	0.1	0.0	0.0	9.1	4.6	4.6
pan	162	152	114	3289.7	28.9	0.4	7893.2	69.2	0.0
Rintanen	2	2	2	39.3	19.6	19.6	7.7	3.9	3.9
Sakallah	1	1	1	0.1	0.1	0.1	1.5	1.5	1.5
Scholl-Becker	38	35	35	183.4	5.2	0.1	714.7	20.4	0.0
Sorting.net	49	44	33	1069.8	32.4	1.8	1238.7	37.5	0.4
SzymanskiP	2	2	2	0.0	0.0	0.0	0.0	0.0	0.0
terminator	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
tipdiam	78	73	59	10.3	0.2	0.0	1848.0	31.3	0.0
tipfixpoint	73	69	68	707.6	10.4	0.1	290.0	4.3	0.3
Toilet	8	7	6	28.6	4.8	0.4	206.6	34.4	0.0
Tree	12	10	6	0.4	0.1	0.0	420.6	70.1	0.3
uclid	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
VonNeumann	10	10	10	6.5	0.6	0.4	34.8	3.5	2.2
wmiforward	13	11	9	0.0	0.0	0.0	0.0	0.0	0.0
<b>total</b>	<b>1228</b>	<b>1149</b>	<b>1070</b>	<b>13367</b>	<b>12.5</b>	<b>0.1</b>	<b>17537</b>	<b>16.4</b>	<b>0.1</b>

Table 6.2: QBFEVAL'08 family overview of certification workflow.

		QRPcheck		QRPcert		CertCheck		PicoSAT	
		mem.	time	mem.	time	mem.	time	mem.	time
<b>2008</b>	sat	18	0	12	0	0	0	12	55
	unsat	44	0	5	0	0	0	0	12
	<b>total</b>	<b>62</b>	<b>0</b>	<b>17</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>12</b>	<b>67</b>
<b>2010</b>	sat	4	0	10	0	0	0	45	12
	unsat	10	0	1	0	0	0	0	5
	<b>total</b>	<b>14</b>	<b>0</b>	<b>11</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>45</b>	<b>17</b>

Table 6.3: Number of instances lost due to given memory resp. time limit.



Out of 362 (1228) solved instances of the QBFEVAL'10 (QBFEVAL'08) benchmark set, we were able to extract 337 (1149) certificates, of which 275 (1070) were validated successfully. DepQBF required almost 5000 (13400) seconds for solving and tracing the 275 (1070) instances that were validated by PicoSAT, whereas certification of those instances was done in about 4500 (17500) seconds.

Table 6.3 provides an overview of the number of instances that were not successfully processed by QRPcheck, QRPcert, CertCheck and PicoSAT due to given memory and time limits. On 14 (62) instances, QRPcheck ran out of memory as the traces produced by DepQBF were 16 GB (18 GB) on average, with a maximum of 27 GB (52 GB). QRPcert ran out of memory on 11 (17) proofs with an average file size of 3.5 GB (3.6 GB) and a maximum of 5.9 GB (5.5 GB). By far the most instances were lost during the validation process with PicoSAT, where 17 (67) instances timed out and 45 (12) ran out of memory. The CNFs generated by CertCheck for the 17 (67) instances that timed out had an average of 8 (6) million variables and 25 (18) million clauses, whereas the 45 (12) instances that ran out of memory had 60 (49) million variables and 179 (147) million clauses on average. From the 62 instances of the QBFEVAL'10 benchmark set that were not validated by PicoSAT, 51 instances are members of the 'mqm' family, which consists of a total of 128 formulas with 70 unsatisfiable instances and 58 satisfiable instances. PicoSAT was able to validate all 70 unsatisfiable instances, but did not succeed in validating any of the satisfiable instances. Similarly, almost half of the instances (34) of the QBFEVAL'08 benchmark set that were not validated by PicoSAT are part of the 'pan' family, where 32 (resp. 2) instances are satisfiable (resp. unsatisfiable). In total, 57 (67) instances out of the 62 (79) instances of the QBFEVAL'10 (QBFEVAL'08) benchmark set that were not validated by PicoSAT are satisfiable. This is due to the fact that certificates of satisfiability tend to grow much larger than certificates of unsatisfiability, mostly because of the size of the initial cubes in the proofs. For example, the proofs of the 51 instances of the 'mqm' family that were not validated by PicoSAT have 40000 initial cubes on average, where each cube has an average size of 970 literals. The corresponding certificates have 52 million AND-gates on average, whereas the resulting CNFs generated by CertCheck have 52 million variables and 156 million clauses on average.

Table 6.4 shows a comparison of the average and median size of the generated proofs, AIGs and CNFs in terms of number of vertices and literals, number of AND-gates, and number of variables and clauses, respectively. The comparison shows that the generated files for satisfiable instances are a multiple times larger on average compared to files generated for unsatisfiable instances. Proofs of satisfiability for the instances of the QBFEVAL'10 (QBFEVAL'08) benchmark set have 8 (1.5) times the number of literals compared to proofs of unsatisfiability. The largest proof of satisfiability has a maximum of 3 (5) million vertices with 1.2 (1.4) billion literals, whereas

		Proof				AIG		CNF			
		vertices		literals		AND-gates		variables		clauses	
		avg.	med.	avg.	med.	avg.	med.	avg.	med.	avg.	med.
<b>2008</b>	sat	127k	119	33M	32k	2M	4k	2M	19k	6M	51k
	unsat	155k	1k	19M	58k	68k	58	164k	19k	409k	40k
	<b>total</b>	<b>144k</b>	<b>747</b>	<b>25M</b>	<b>43k</b>	<b>872k</b>	<b>197</b>	<b>953k</b>	<b>19k</b>	<b>3M</b>	<b>45k</b>
<b>2010</b>	sat	308k	1k	117M	626k	20M	24k	20M	62k	59M	183k
	unsat	135k	2k	14M	146k	170k	193	336k	23k	846k	55k
	<b>total</b>	<b>211k</b>	<b>2k</b>	<b>60M</b>	<b>175k</b>	<b>8M</b>	<b>369</b>	<b>8M</b>	<b>28k</b>	<b>25M</b>	<b>71k</b>

Table 6.4: Comparison of generated proofs, AIGs and CNFs.

		Proof		AIG		CNF	
		avg.	med.	avg.	med.	avg.	med.
<b>2008</b>	sat	145.1 MB	154.4 kB	49.2 MB	68.5 kB	133.7 MB	779.0 kB
	unsat	89.8 MB	351.5 kB	1.5 MB	14.9 kB	7.4 MB	626.1 kB
	<b>total</b>	<b>112.4 MB</b>	<b>272.5 kB</b>	<b>20.7 MB</b>	<b>26.5 kB</b>	<b>58.4 MB</b>	<b>675.0 kB</b>
<b>2010</b>	sat	518.4 MB	2.8 MB	449.4 MB	378.9 kB	1.2 GB	2.8 MB
	unsat	66.7 MB	729.9 kB	3.6 MB	13.8 kB	15.5 MB	874.7 kB
	<b>total</b>	<b>265.3 MB</b>	<b>1.0 MB</b>	<b>192.7 MB</b>	<b>23.6 kB</b>	<b>524.2 MB</b>	<b>1.1 MB</b>

Table 6.5: File size comparison of generated proofs, AIGs and CNFs.

the largest proof of unsatisfiability consists of 8 (7.5) million vertices with 1.3 (1.2) billion literals. Further, certificates of satisfiability are 117 (30) times the size of certificates of unsatisfiability in terms of number of AND-gates on average. The size of CNFs generated for validating certificates of satisfiability compared to certificates of unsatisfiability in terms of clauses is up to 70 (29) times larger on average. The maximum number of clauses for CNFs generated for satisfiable (resp. unsatisfiable) instances is 441 (350) million (resp. 30 (43) million) clauses. A comparison of the actual file size of the generated files is given in Table 6.5, where the file sizes for satisfiable (resp. unsatisfiable) instances correlate with the ratios discussed in Table 6.4.

## 6.2 Runtime Comparison

We evaluated the runtime of each tool in the framework with respect to the 275 (1070) instances that were validated by PicoSAT. First, we compared the time required by DepQBF for solving and tracing to the aggregated time needed by QRPcert, CertCheck and PicoSAT for certification. The comparison of solving (incl. tracing) and certification on the QBFEVAL'10 (QBF-EVAL'08) instances is given in Figure 6.2a (6.3a), where all instances that were solved by DepQBF in median solving time (0.2 (0.1) seconds) and above are considered. In total, a few instances require most of the certification runtime. In fact, 5 (8) instances out of 275 (1070) require over 85% (51%) of total certification runtime, where DepQBF requires only a fraction for solv-

ing those instances. The five most time-consuming instances in terms of certification are given in Table 6.6.

Certification ("Cert.") requires by far more time than DepQBF needs for solving and tracing ("Solv."). Interestingly, PicoSAT requires over 99% of certification runtime for validating (Valid.) those instances. Conversely, the five most time-consuming instances in terms of solving and tracing are given in Table 6.7. An interesting fact is that the instances that are hard to certify require less time for being solved, whereas the instances that required most time for being solved are certified in a fraction of solving time.

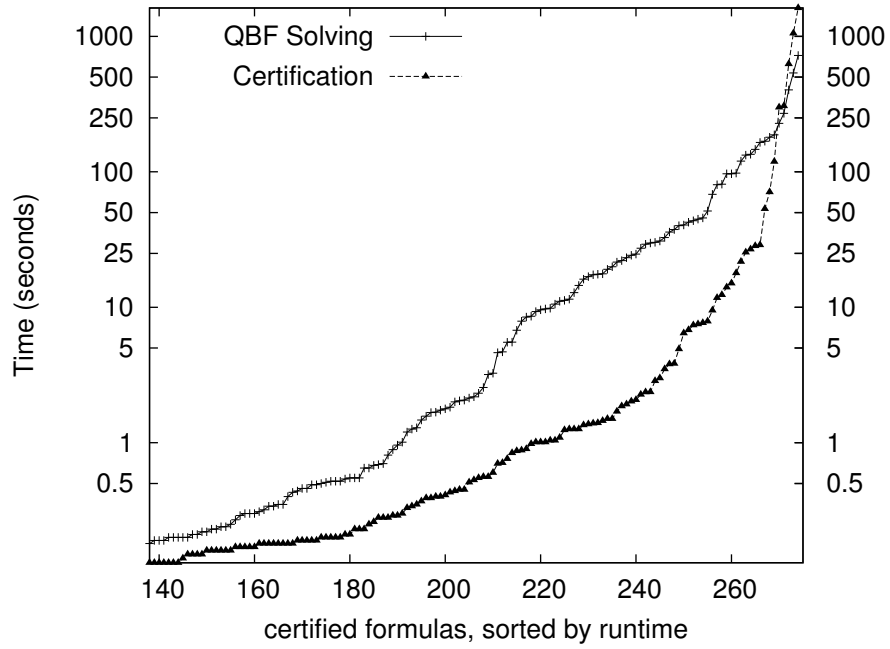
Figure 6.2b shows the comparison of the runtime of DepQBF, QRPCert, CertCheck, and PicoSAT on the 275 validated instances of the QBFEVAL'10 benchmark set. It clearly shows that validating certificates with PicoSAT is the most time-consuming task of the certification process. In fact, over 96% of total certification runtime are required for validation, whereas certificate extraction and CNF conversion takes approximately 3% and 1%, respectively. Similar results were obtained on the QBFEVAL'08 benchmark set (Figure 6.3b), where 94% of total certification runtime is required for

	Instance		Time [s]		
	Name	Result	Cert.	Solv.	Valid.
<b>2008</b>	BLOCKS4ii.6.3	unsat	1475.6	83.3	1464.3
	k.ph.n-15	sat	1471.1	41.5	1464.7
	CHAIN14v.15	sat	1300.7	4.3	1293.4
	eijk.S953.S-d3	sat	1136.1	1.7	1133.5
	k.branch.n-3	sat	1106.7	1.6	1105.1
<b>2010</b>	k.ph.n-15-shuffled	sat	1619.3	96.7	1613.3
	k.ph.n-14-shuffled	sat	1053.6	42.5	1049.6
	toilet.c.08.01.13-shuffled	unsat	624.5	1.7	623.3
	BLOCKS3i.5.3-shuffled	unsat	305.7	11.3	303.2
	k.ph.n-13-shuffled	sat	299.4	16.1	296.7

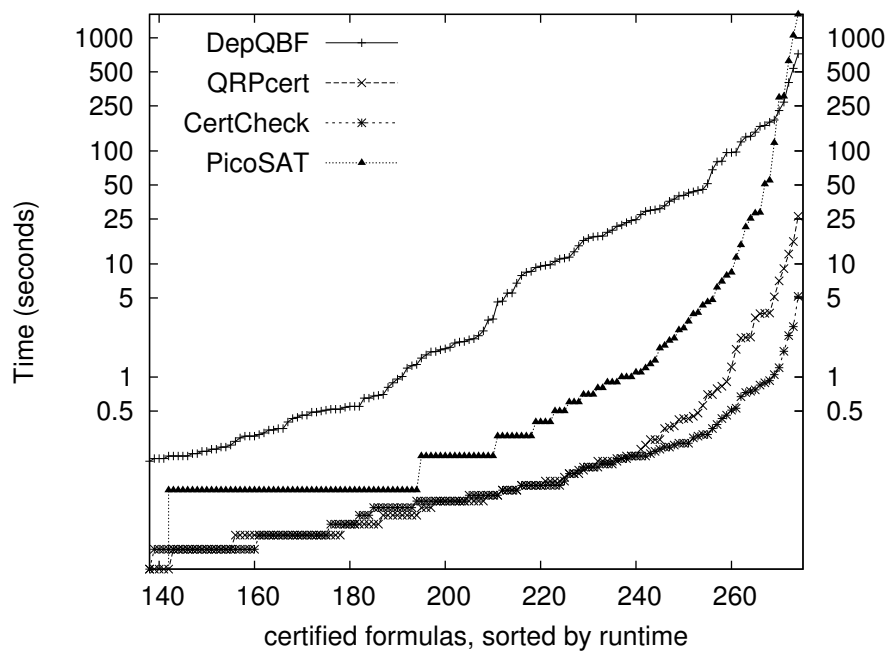
Table 6.6: Top 5 time-consuming instances for certification.

	Instance		Time [s]		
	Name	Result	Solv.	Cert.	Valid.
<b>2008</b>	c5_BMC_p1_k8	sat	768.2	1.5	0.8
	c5_BMC_p2_k8	unsat	631.4	5.5	4.8
	k.path.p-7	unsat	571.0	0.2	0.0
	c3_BMC_p1_k16	sat	556.6	6.3	5.4
	k.grz.p-13	unsat	407.5	0.1	0.0
<b>2010</b>	k.t4p.p-4-shuffled	unsat	721.9	0.3	0.0
	k.d4.p-6-shuffled	unsat	536.6	1.1	0.0
	ev-pr-6x6-13-5-0-1-2-lg-shuffled	unsat	402.6	70.9	54.8
	k.grz.p-10-shuffled	unsat	269.8	0.2	0.0
	counter.r.8-shuffled	sat	227.7	3.0	2.6

Table 6.7: Top 5 time-consuming instances for solving.



(a) Solving vs. certification



(b) Tool comparison

Figure 6.2: QBFEVAL'10 runtime comparison, all instances with solving time  $\geq 0.2$ s considered.

		Instances			Total Time [s]			
		sv	ex	va	DepQBF	QRPcert	CertCheck	PicoSAT
<b>2008</b>	sat	494	464	397	3502.9	95.3	38.4	13874.1
	unsat	734	685	673	9863.7	831.8	57.4	2639.8
	<b>total</b>	<b>1228</b>	<b>1149</b>	<b>1070</b>	<b>13366.6</b>	<b>927.1</b>	<b>95.8</b>	<b>16513.9</b>
<b>2010</b>	sat	157	143	86	701.8	30.9	6.4	3247.0
	unsat	205	194	189	4241.9	86.8	28.9	1090.0
	<b>total</b>	<b>362</b>	<b>337</b>	<b>275</b>	<b>4943.7</b>	<b>117.6</b>	<b>35.4</b>	<b>4337.0</b>

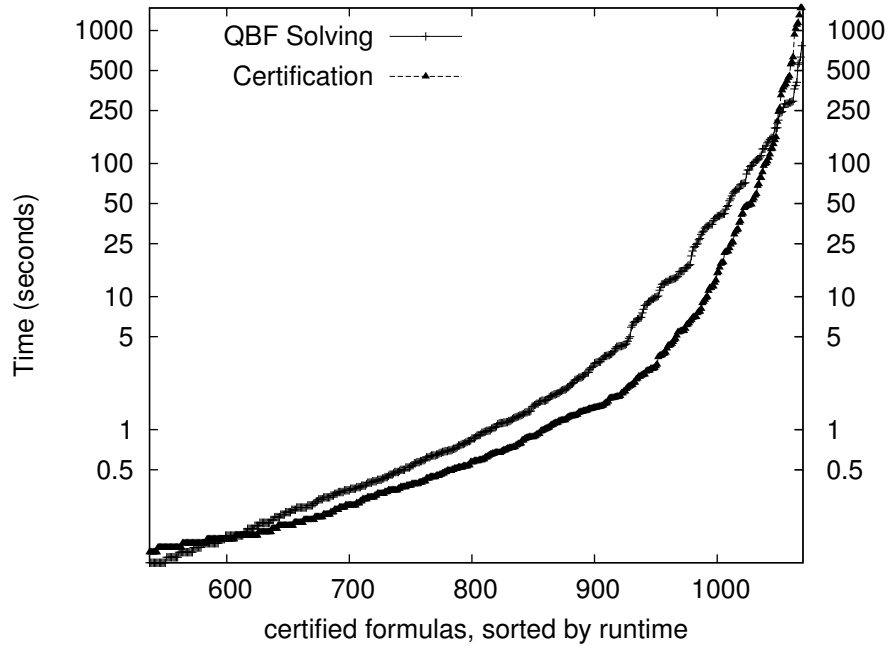
Table 6.8: QBFEVAL'10 and QBFEVAL'08 certification summary.

validation. Certificate extraction and CNF conversion took up 5% and 1% of total certification runtime, respectively.

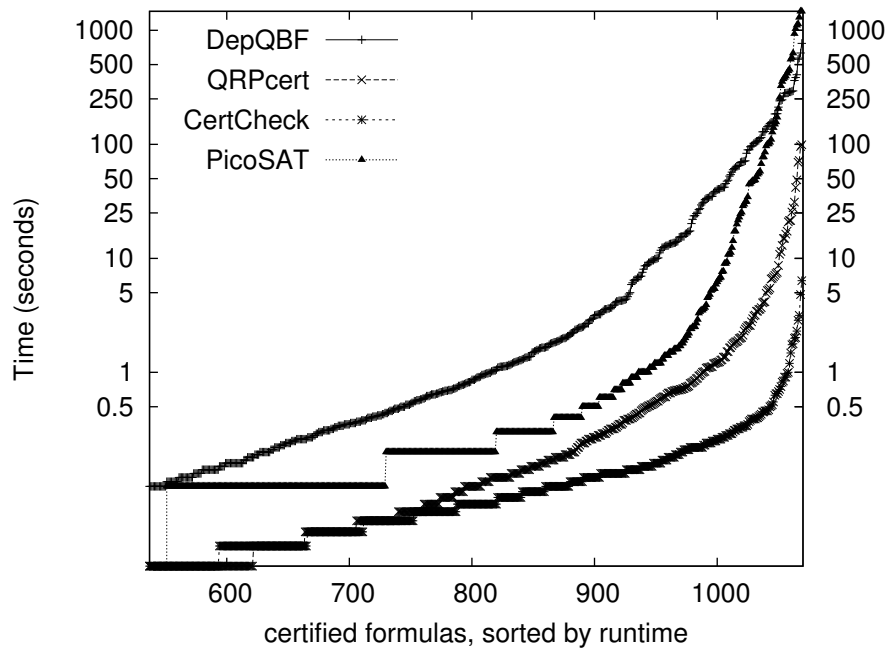
Table 6.8 summarizes the results of our framework applied to the QBFEVAL'10 and QBFEVAL'08 benchmark sets. It shows that certification heavily depends on whether an instance is satisfiable or unsatisfiable, especially for certificate validation. From the 1070 validated instances of the QBFEVAL'08 benchmark set, 37% (397) were satisfiable instances, which were validated in 79% of total certification time. Conversely, validating the 673 (63%) unsatisfiable instances required 15% of total certification time. Similarly, on the QBFEVAL'10 benchmark set, validation of the 86 (31%) satisfiable instances required 72% of total certification runtime, whereas validating 189 (69%) unsatisfiable instances took only 24% of total certification runtime. Hence, we conclude that validation of certificates of satisfiability is most time-consuming.

### 6.3 Certificate Statistics

We further evaluated the certificates extracted by QRPcert with respect to the structure of the constructed AIGs. The AIG library we employed supports structural hashing of two-input AND-gates, which has the advantage that each AND-gate in the certificate is unique with respect to its two inputs. The results based on the 337 (1149) extracted certificates of the QBFEVAL'10 (QBFEVAL'08) are given in Table 6.9. Certificates of satisfiability have 117 (30) times the number of AND-gates on average compared to certificates of unsatisfiability. The maximum number of AND-gates used for representing a certificate of satisfiability and unsatisfiability are 147 (116) and 10 (14) million AND-gates, respectively. In case of certificates of satisfiability, we are able to share an average of 65.2% (70.7%) of the overall number of AND-gates constructed. In contrast, only 23% (7.8%) of the AND-gates in certificates of unsatisfiability can be shared. The high degree of AND-gate sharing in certificates of satisfiability is due to the fact that initial cubes themselves share a large number of literals. The results show that representing certificates as AIGs is beneficial in case structural hashing



(a) Solving vs. certification



(b) Tool comparison

Figure 6.3: QBFEVAL'08 runtime comparison, all instances with solving time  $\geq 0.1s$  considered.

		In	Out	AND-gates			AND-gates shared			
		avg.	avg.	max.	avg.	med.	avg.%	med.%	avg.	med.
<b>2008</b>	sat	75	6k	116M	2M	4k	70.7	80.0	8M	16k
	unsat	13k	72	14M	68k	58	7.8	0.0	64k	0
	<b>total</b>	<b>8k</b>	<b>2k</b>	<b>116M</b>	<b>872k</b>	<b>197</b>	<b>33.2</b>	<b>12.1</b>	<b>3M</b>	<b>11</b>
<b>2010</b>	sat	125	3k	147M	20M	24k	65.2	66.8	23M	190k
	unsat	20k	95	10M	170k	193	23.0	23.7	64k	22
	<b>total</b>	<b>12k</b>	<b>1k</b>	<b>147M</b>	<b>8M</b>	<b>369</b>	<b>40.9</b>	<b>46.6</b>	<b>10M</b>	<b>327</b>

Table 6.9: Overview of total (shared) number of AND-gates of certificates.

		Functions (avg.)				Proof (avg.)		Vertex util.	
		func.	d.c.	vert.	lit.	red. vert.	red. lit.	avg.	med.
<b>2008</b>	sat	6k	49	6M	3M	19k	6M	4418	467
	unsat	72	24	16k	119k	3k	16k	83	43
	<b>total</b>	<b>2k</b>	<b>34</b>	<b>3M</b>	<b>1M</b>	<b>10k</b>	<b>3M</b>	<b>2142</b>	<b>100</b>
<b>2010</b>	sat	3k	67	18M	24M	72k	18M	1761	255
	unsat	95	40	25k	213k	4k	25k	42	33
	<b>total</b>	<b>1k</b>	<b>51</b>	<b>8M</b>	<b>10M</b>	<b>33k</b>	<b>8M</b>	<b>806</b>	<b>58</b>

Table 6.10: Certificate overview of Skolem/Herbrand functions.

is employed, especially in the case of certificates of satisfiability.

We also evaluated the certificates with respect to the extracted Skolem (resp. Herbrand) functions, which is given in Table 6.10. Column "func." indicates the average number of Skolem (resp. Herbrand) functions extracted, whereas column "d.c." denotes the average number of don't care variables in a certificate. A variable is denoted as don't care if its RFAO stack is empty, i.e., the variable is not required for showing that a formula is satisfiable (resp. unsatisfiable). Column "vert." refers to the average number of pushed vertices on the RFAO stacks of the extracted Skolem (resp. Herbrand) functions, whereas column "lit." denotes the average number of literals of the pushed vertices (with duplicate vertices not considered). Columns "red. vert." and "red. lit." denote the average number of vertices that are reduced via existential- (resp. universal-) reduction and the average number of literals that are reduced in a proof, respectively. "Vertex util." indicates how often a vertex is reused in the Skolem (resp. Herbrand) functions on average. The results show that a certificate of satisfiability consists of an average of 3k (6k) Skolem functions, which are composed of 18 (6) million vertices with 24 (3) million literals on average. In contrast, certificates of unsatisfiability consist of 95 (72) Herbrand functions with 25k (16k) vertices and 213k (119k) literals on average. Further, we extracted Skolem functions for 98% (97%) of the existentially quantified variables on average, where the remaining 2% (3%) were don't care variables and thus not required. In case of certificates of unsatisfiability, we extracted Herbrand functions for 75% (70%) of the universally quantified variables on average. An interesting figure is the vertex utilization ("Vertex util."), which states how often a

reduced vertex is pushed onto the RFAO stacks and thus, how many literals are reduced on average via existential- (resp. universal-) reduction from a vertex. For example, given a proof of satisfiability, an average of 1761 (4418) literals is eliminated via existential-reduction from a cube, whereas an average of 42 (83) literals are eliminated via universal-reduction from a clause in a proof of unsatisfiability.

## 6.4 Simplification Results

We enabled simplification in QRPcert (as described in Section 4.6) and rerun the experiments on the QBFEVAL'08 and QBFEVAL'10 benchmark sets. A summary of the results is given in Table 6.11, where we compare the run with simplification enabled ("simpl.") to the previous run without simplification enabled ("no simpl."). Note that the runtime comparison considers only those instances of the QBFEVAL'10 (QBFEVAL'08) benchmark set that were validated in both test runs ("simpl." and "no simpl."), which are 274 (1068) instances in total. With simplification enabled, QRPcert timed out on two instances from the QBFEVAL'08 benchmark set, which were previously extracted in 7 and 15 seconds, respectively. Further, QRPcert required almost four times the runtime of the previous run ("no simpl.") for simplifying and extracting the 1068 validated certificates, whereas validation of those instances was 1100 seconds faster. On the QBFEVAL'10 benchmark set, QRPcert extracted all certificates that were extracted in the previous run, but also required almost four times the runtime. The time required for validation did not change significantly. Considering validated instances only, simplification eliminated an average of 2700 (3300) AND-gates from certificates of satisfiability, which have a total of 22000 (28000) AND-gates on average. Certificates of unsatisfiability were reduced by a mere 70 (60) AND-gates on average resulting in certificates with an average of 4300 (2300) AND-gates in total. Simplification of the 337 (1147) extracted certificates reduced the total number of AND-gates by less than 0.7%, while the runtime of QRPcert increased by a factor of 4. Hence, we

		Instances				Total Time [s]			
		no simpl.		simpl.		no simpl.		simpl.	
		ex	va	ex	va	QRPcert	PicoSAT	QRPcert	PicoSAT
<b>2008</b>	sat	464	397	462	395	67.5	12378.7	2586.0	11085.3
	unsat	685	673	685	673	831.8	2639.8	857.0	2772.7
	<b>total</b>	<b>1149</b>	<b>1070</b>	<b>1147</b>	<b>1068</b>	<b>899.3</b>	<b>15018.5</b>	<b>3443.0</b>	<b>13858.0</b>
<b>2010</b>	sat	143	86	143	85	25.8	1633.7	337.6	1747.9
	unsat	194	189	194	189	86.8	1090.0	87.1	1004.3
	<b>total</b>	<b>337</b>	<b>275</b>	<b>337</b>	<b>274</b>	<b>112.5</b>	<b>2723.7</b>	<b>424.7</b>	<b>2752.2</b>

Table 6.11: Comparison of certification with ("simpl.") and without ("no simpl.") simplification enabled.



conclude that simplification in its current implementation is time-consuming and not considered to be beneficial for the certification workflow.

## 6.5 Increasing the Memory Limit

We analyzed the 14 and 11 instances of the QBFEVAL'10 benchmark set, where QRPcheck and QRPcert ran out of memory due to given memory limit of 7 GB. For that purpose, we lifted the previous memory limit to 80 GB and set the time limit to 3600 seconds. We performed the experiments on a 2.4 GHz Intel Xeon hexa-core machine with 96 GB of main memory, running Ubuntu 11.10.

First, we rerun the experiments on the 14 instances (4 sat., 10 unsat.), where QRPcheck previously ran out of memory. As a result, we were able to obtain all 14 proofs of which QRPcert was able to extract 14 certificates. Further, PicoSAT was able to validate 12 out of 14 certificates, but timed out on 2 instances (1 sat. and 1 unsat.) while validating CNFs with 30 and 3 million clauses, respectively. Not considering the runtime required by QRPcheck, the runtime required for certifying the 12 instances was 175 seconds on average, whereas DepQBF required an average of 645 seconds for solving them. The average (median) memory usage for the whole certification workflow was 19 GB (18 GB) with a maximum of 28 GB.

Finally, we rerun the experiments on the 11 instances (10 sat., 1 unsat.) on which QRPcert previously ran out of memory. We were able to extract certificates for all 11 instances, but we did not succeed in validating any given instance with PicoSAT. Two instances ran out of memory, whereas the remaining nine timed out. The extracted certificates had an average (median) of 176 (154) million AND-gates, which approximately results in CNFs with an average of 528 (162) million clauses. As an extreme case, the certificates of the satisfiable instances `Core1108_tbm_02.tex.moduleQ3.2S.000077` and `Core1108_tbm_02.tex.moduleQ3.2S.000007` had over 310 and 320 million AND-gates, respectively. The average (median) memory usage of the whole certification workflow for the timed out instances was 40 GB (38 GB) with a maximum of 60 GB.

By lifting the memory limit we were able to extract certificates for 100% of the solved instances of the QBFEVAL'10 benchmark set. However, we were only able to validate 12 out of 25 certificates with PicoSAT, where 11 instances timed out and the remaining two ran out of memory. In the next chapter, we discuss some ideas that might improve the validation process.



# Chapter 7

## Conclusion

In this thesis, we presented `QRPcert`, a tool for extracting Skolem/Herbrand function-based certificates of (un)satisfiability based on the algorithm presented in [2]. Certificates are extracted from Q-resolution proofs and traces in QRP format, a novel text-based and explicit format for representing Q-resolution proofs and traces introduced in [36]. The extracted sets of Skolem (resp. Herbrand) functions are represented as AIGs, which we simplify by common basic simplification techniques such as structural hashing and constant propagation. The certificates generated by `QRPcert` are represented in the ASCII version of the AIGER format.

We further presented the tool `CertCheck`, which we used for transforming the input formula into an AIG and merging the result with the corresponding certificate extracted by `QRPcert`. We then transformed the resulting AIG into a propositional formula and validated the correctness of the result of `DepQBF` by means of the SAT solver `PicoSAT`.

We performed an extensive evaluation on the benchmark sets of the QBF competitions 2008 and 2010. The results showed that `QRPcert` was able to extract certificates for over 90% of the instances solved by `DepQBF`. We also showed that by lifting the memory limit of 7 GB, `QRPcert` was able to extract certificates for 100% of the solved instances. Further, we were able to validate over 80% of the extracted certificates, which all were proved to be correct by `PicoSAT`. Most of the instances that were not validated were satisfiable instances, which turned out to be much harder to validate than unsatisfiable instances. Our runtime comparison showed that certificate extraction with `QRPcert` requires only a fraction of the runtime needed for solving instances with `DepQBF`. In contrast, the runtime required for certificate validation with `PicoSAT` heavily depended on whether given instance was satisfiable or unsatisfiable. Further, employing AIGs with structural hashing for representing the certificates extracted by `QRPcert` proved to be beneficial. Our results show that even basic simplification techniques like structural hashing of two-input AND-gates reduced over 65% of the AND-

gates required for representing certificates of satisfiability on average. We also showed that simplification as currently implemented in `QRPcert` is not beneficial as the number of AND-gates reduced does not justify the resulting increase in runtime.

However, the extraction of Skolem/Herbrand function-based QBF certificates from Q-resolution proofs is a promising approach, which, we believe, will enable many applications of QBF solving in practice.

## 7.1 Future Work

Our experiments have shown that the validation of the certificates is a bottleneck in the current certification framework. A possible enhancement of `CertCheck` would be the integration of an incremental SAT solver in order to validate a certificate incrementally, as suggested in [4]. Another enhancement of `CertCheck` would be to improve AIG to CNF translation by allowing multi-input AND-gates and employing optimizations like Plaisted-Greenbaum.

Further, extending our AIG library to support more advanced simplification techniques may further reduce the size of the certificates extracted by `QRPcert`. Another idea would be to employ tools like ABC [46] in order to reduce the overall size of the certificates.

A desirable property of `QRPcert` would be to support advanced dependency schemes as employed in `DepQBF`. Hence, it would be interesting to add this extension to `QRPcert` in order to observe the impact of advanced dependency schemes on Skolem/Herbrand function-based certificates. Another useful extension would be to add support for more input and output formats in `QRPcert`.

One of the most interesting topics would be the extension of `DepQBF` to support the direct extraction of Skolem/Herbrand function-based certificates, which would require to maintain Q-resolution proofs within `DepQBF`.

`QRPcert` and `CertCheck` are available at <http://fmv.jku.at/cdepqbf/>.

# Bibliography

- [1] Audemard, G., Sais, L.: A Symbolic Search Based Approach for Quantified Boolean Formulas. In: Proc. of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005). Lecture Notes in Computer Science, vol. 3569, pp. 16–30. Springer (2005)
- [2] Balabanov, V., Jiang, J.H.R.: Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In: Proc. of the 23rd International Conference on Computer Aided Verification (CAV 2011). Lecture Notes in Computer Science, vol. 6806, pp. 149–164. Springer (2011)
- [3] Benedetti, M.: Evaluating QBFs via Symbolic Skolemization. In: Proc. of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004). Lecture Notes in Computer Science, vol. 3452, pp. 285–300. Springer (2004)
- [4] Benedetti, M.: Extracting Certificates from Quantified Boolean Formulas. In: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005). pp. 47–53. Professional Book Center (2005)
- [5] Biere, A.: Resolve and Expand. In: SAT (Selected Papers). Lecture Notes in Computer Science, vol. 3542, pp. 59–70. Springer (2004)
- [6] Biere, A.: PicoSAT Essentials. Journal on Satisfiability (JSAT) 4(2-4), 75–97 (2008)
- [7] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Proc. of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999). Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer (1999)
- [8] Bjesse, P., Borälv, A.: DAG-Aware Circuit Compression for Formal Verification. In: Proc. of the International Conference on Computer-Aided Design (ICCAD 2004). pp. 42–49. IEEE Computer Society / ACM (2004)

- [9] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8), 677–691 (1986)
- [10] Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Convergence Testing in Term-Level Bounded Model Checking. In: *Proc. of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, (CHARME 2003)*. *Lecture Notes in Computer Science*, vol. 2860, pp. 348–362. Springer (2003)
- [11] Büning, H.K., Bubeck, U.: Theory of Quantified Boolean Formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 735–760. IOS Press (2009)
- [12] Büning, H.K., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. *Information and Computation* 117(1), 12–18 (1995)
- [13] Davis, M., Logemann, G., Loveland, D.W.: A Machine Program for Theorem-Proving. *Communications of the ACM (CACM)* 5(7), 394–397 (1962)
- [14] Dershowitz, N., Hanna, Z., Katz, J.: Bounded Model Checking with QBF. In: *Proc. of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*. *Lecture Notes in Computer Science*, vol. 3569, pp. 408–414. Springer (2005)
- [15] Fitting, M.: *First-Order Logic and Automated Reasoning* (2. ed.). *Graduate texts in computer science*, Springer (1996)
- [16] Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In: *Proc. of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001)*. *Lecture Notes in Computer Science*, vol. 2083, pp. 364–369. Springer (2001)
- [17] Giunchiglia, E., Narizzano, M., Tacchella, A.: QBF Reasoning on Real-World Instances. In: *Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*. *Lecture Notes in Computer Science*, vol. 3542, pp. 105–121. Springer (2004)
- [18] Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *Journal of Artificial Intelligence Research (JAIR)* 26, 371–416 (2006)
- [19] Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantifier Structure in Search Based Procedures for QBFs. In: *Proc. of the Conference on*

- Design, Automation and Test in Europe (DATE 2006). pp. 812–817. European Design and Automation Association, Leuven, Belgium (2006)
- [20] Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or Not QB: An Efficient Execution Verification Tool for Memory Orderings. In: Proc. of the 16th International Conference Computer Aided Verification (CAV 2004). Lecture Notes in Computer Science, vol. 3114, pp. 401–413. Springer (2004)
- [21] Goultiaeva, A., Gelder, A.V., Bacchus, F.: A Uniform Approach for Generating Proofs and Strategies for Both True and False QBF Formulas. In: Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011). pp. 546–553. IJCAI/AAAI (2011)
- [22] Goultiaeva, A., Iverson, V., Bacchus, F.: Beyond CNF: A Circuit-Based QBF Solver. In: Proc. of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009). Lecture Notes in Computer Science, vol. 5584, pp. 412–426. Springer (2009)
- [23] Heijenoort, J.V.: From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931. Harvard University Press (1967)
- [24] Jussila, T., Biere, A.: Compressing BMC Encodings with QBF. Electronic Notes in Theoretical Computer Science (ENTCS) 174(3), 45–56 (2007)
- [25] Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A First Step Towards a Unified Proof Checker for QBF. In: Proc. of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007). Lecture Notes in Computer Science, vol. 4501, pp. 201–214. Springer (2007)
- [26] Jussila, T., Sinz, C., Biere, A.: Extended Resolution Proofs for Symbolic SAT Solving with Quantification. In: Proc. of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006). Lecture Notes in Computer Science, vol. 4121, pp. 54–60. Springer (2006)
- [27] Kröning, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Berlin, Heidelberg (2008)
- [28] Kuehlmann, A., Ganai, M.K., Paruthi, V.: Circuit-based Boolean Reasoning. In: Proc. of the 38th Design Automation Conference (DAC 2001). pp. 232–237. ACM (2001)
- [29] Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean Reasoning for Equivalence Checking and Functional Property Verifica-

- tion. *IEEE Transactions on CAD of Integrated Circuits and Systems* 21(12), 1377–1394 (2002)
- [30] Lonsing, F., Biere, A.: Integrating Dependency Schemes in Search-Based QBF Solvers. In: *Proc. of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*. pp. 158–171. LNCS, Springer (2010)
- [31] Mangassarian, H., Veneris, A.G., Safarpour, S., Benedetti, M., Smith, D.E.: A Performance-Driven QBF-based iterative Logic Array Representation with Applications to Verification, Debug and Test. In: *Proc. of the 2007 International Conference on Computer-Aided Design (ICCAD 2007)*. pp. 240–245. IEEE (2007)
- [32] McMillan, K.L.: Applications of Craig Interpolants in Model Checking. In: *Proc. of the 11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*. Lecture Notes in Computer Science, vol. 3440, pp. 1–12. Springer (2005)
- [33] Mishchenko, A., Chatterjee, S., Brayton, R.K.: DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In: *Proc. of the 43rd Design Automation Conference (DAC 2006)*. pp. 532–535. ACM (2006)
- [34] Mneimneh, M.N., Sakallah, K.A.: Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution. In: *Selected Revised Papers of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*. Lecture Notes in Computer Science, vol. 2919, pp. 411–425. Springer (2003)
- [35] Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and Certifying QBFs: A Comparison of State-of-the-Art Tools. *AI Communications (AICOM)* 22(4), 191–210 (2009)
- [36] Niemetz, A.: Extracting and Checking Q-Resolution Proofs from a State-of-the-Art QBF Solver. Master’s thesis, Johannes Kepler University, Linz (2012)
- [37] Plaisted, D.A., Greenbaum, S.: A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation* 2(3), 293–304 (1986)
- [38] Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. *Journal of Artificial Intelligence Research (JAIR)* 10, 323–352 (1999)
- [39] Rintanen, J.: Partial Implicit Unfolding in the Davis-Putnam Procedure for Quantified Boolean Formulae. In: *Proc. of the 8th International*



- Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001). Lecture Notes in Computer Science, vol. 2250, pp. 362–376. Springer (2001)
- [40] Rintanen, J.: Asymptotically Optimal Encodings of Conformant Planning in QBF. In: Proc. of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007). pp. 1045–1050. AAAI Press (2007)
- [41] Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM (JACM)* 12(1), 23–41 (1965)
- [42] Sinz, C., Biere, A.: Extended Resolution Proofs for Conjoining BDDs. In: Proc. of the 1st International Computer Science Symposium in Russia on Computer Science - Theory and Applications (CSR 2006). Lecture Notes in Computer Science, vol. 3967, pp. 600–611. Springer (2006)
- [43] Skolem, T.: Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit und Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen. *Skrifter utgit av Videnskabselskapet i Kristiania, I; Matematisk-naturvidenskabelig klasse 4*, 1–36 (1920), english translation ‘Logico-combinatorial investigations in the satisfiability or provability of mathematical propositions: A simplified proof of a theorem by L. Löwenheim and generalizations of the theorem’ in [23], pp. 252–263
- [44] Staber, S., Bloem, R.: Fault Localization and Correction with QBF. In: Proc. of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007). Lecture Notes in Computer Science, vol. 4501, pp. 355–368. Springer (2007)
- [45] Stockmeyer, L.J., Meyer, A.R.: Word Problems Requiring Exponential Time: Preliminary Report. In: Proc. of the 5th Annual ACM Symposium on Theory of Computing (STOC 1973). pp. 1–9. ACM (1973)
- [46] Synthesis, B.L., Group, V.: ABC: A system for sequential synthesis and verification (Nov 2011), <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [47] Synthesis, B.L., Group, V.: MVSIS: Logic synthesis and verification (Nov 2011), <http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/>
- [48] Tseitin, G.S.: On the Complexity of Derivation in the Propositional Calculus. *Zapiski nauchnykh seminarov (LOMI)* 8, 234–259 (1968), english translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125

- [49] Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming and SAT. In: Proc. of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, (CHARME 2003). Lecture Notes in Computer Science, vol. 2860, pp. 81–95. Springer (2003)
- [50] Yu, Y., Malik, S.: Validating the Result of a Quantified Boolean Formula (QBF) Solver: Theory and Practice. In: Proc. of the 2005 Conference on Asia South Pacific Design Automation (ASP-DAC 2005). pp. 1047–1051. ACM Press (2005)
- [51] Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002). LNCS, vol. 2470, pp. 200–215. Springer (2002)

# Appendix A

## Appendix

### A.1 QRP format

```
trace      = preamble { quant_set } { step } result EOF.

preamble   = { comment } header.
comment    = "#" text EOL.
header     = "p qrp" pnum pnum EOL.

quant_set  = quantifier { var } "0".
quantifier = "a" | "e".
var        = pnum.

step       = idx literals antecedents.
idx        = pnum.
literals   = { lit } "0".
lit        = ["-"] var.
antecedents = [idx [idx]] "0".

result     = "r " sat EOL.
sat        = "sat" | "unsat".

text       = ? a sequence of non-special ASCII chars ?.
pnum       = ? a 32-bit signed integer > 0 ?.
EOL        = ? end-of-line marker ?.
EOF        = ? end-of-file marker ?.
```

Figure A.1: The QRP format in Extended Backus-Naur Form (EBNF) [36].