JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Extracting and Checking Q-Resolution Proofs from a State-Of-The-Art QBF-Solver

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieurin

im Masterstudium

Informatik

Eingereicht von:
Aina Niemetz BSc

Angefertigt am:
Institut für Formale Modelle und Verifikation

Beurteilung:
Univ.-Prof. Dr. Armin Biere

Mitwirkung:
DI Florian Lonsing
Assist.-Prof. Dr. Martina Seidl

Linz, März, 2012

Abstract

The logic of Quantified Boolean Formulas (QBF) is an extension of propositional logic and provides compact encodings of real world problems in various fields of application, e.g., formal verification, reasoning and artificial intelligence. In recent years, the development of efficient decision procedures for QBF has progressed considerably. However, most current state-of-the-art QBF-solvers return mere *true*/*false* answers to a given problem, which is often not sufficient. Hence, it is a highly requested feature for QBF-solvers to provide the possibility to extract so-called certificates of (un)satisfiability in order to give evidence of the correctness of a solver's result. Further, certificates enable the identification of concrete solutions for a given problem, which is one of the key requirements for real world problems in many fields of application.

In this thesis, we provide the extraction and validation of resolution proofs of (un)satisfiability as part of the certification workflow for the state-of-the-art QBF-solver DepQBF. DepQBF is a dependency-aware search-based solver for QBF in prenex conjunctive normal form and placed first in the main track of the QBFEVAL competition in 2010.

We give a brief overview of QBF solving as implemented in DepQBF and introduce the QRP format, a novel text-based format for resolution-based traces and proofs. We then present the extension of DepQBF to record resolution-based traces in QRP format and introduce QRPcheck, a tool for extracting and validating the corresponding resolution proofs of (un)satisfiability, in detail.

We apply tracing, proof extraction and proof checking on the benchmark sets of the QBFEVAL competitions 2008 and 2010 and present an extensive evaluation of the results. It shows that within given time and memory constraints, over 95% of all solved instances were validated successfully by QRPcheck. Further, all instances validated by QRPcheck proved to have been solved correctly by DepQBF.

Zusammenfassung

Die Logik der quantifizierten Booleschen Formeln (QBF) ist eine Erweiterung der Aussagenlogik, die kompakte Kodierungen von praxisnahen Problemstellungen in unterschiedlichen Anwendungsbereichen, wie bspw. der Formalen Verifikation oder Künstlichen Intelligenz, ermöglicht. Im Laufe der letzten Jahre hat die Entwicklung von effizienten Entscheidungsverfahren für QBF erhebliche Fortschritte gemacht. Ein Großteil der aktuellen Programme für die Evaluierung von QBF (auch QBF-Solver genannt) gibt jedoch nur *wahr/falsch* Antworten zurück, die für eine Vielzahl von Anwendungen nicht ausreichend sind. Ein besonders wünschenswertes Feature für QBF-Solver ist daher die Extraktion sogenannter Zertifikate, die die Erfüllbarkeit bzw. Nicht-Erfüllbarkeit einer Formel bestätigen und die Korrektheit des Ergebnisses eines QBF-Solvers beweisen. Darüber hinaus ermöglichen Zertifikate die Identifikation von konkreten Lösungen für ein gegebenes Problem, was in vielen Anwendungsfällen eine zentrale Anforderung darstellt.

Thema dieser Arbeit ist die Extraktion und Validierung von Resolutionsbeweisen für QBF als Teil des Zertifizierungs-Workflow für den QBF-Solver DepQBF. DepQBF ist ein Solver für QBF in Konjunktiver Pränexnormalform, der erweiterte Abhängigkeitsschemata von Variablenmengen unterstützt und 2010 den ersten Platz im Hauptbewerb des QBFEVAL-Wettbewerbs belegt hat. Wir geben eine kurzen Überblick über die Arbeitsweise von DepQBF und führen das QRP Format, ein neues textbasiertes Format für resolutionsbasierte Beweise und Protokolle von Entscheidungsprozeduren, ein. Wir präsentieren unsere Erweiterung von DepQBF, mit deren Hilfe wir interne Abläufe im QRP Format mitprotokollieren, und stellen in weiterer Folge QRPcheck, ein Werkzeug zur Extraktion und Validierung von resolutionsbasierten Beweisen, in allen Einzelheiten vor. Wir wenden die Protokollierung von Entscheidungsprozeduren, die Extraktion von resolutionsbasierten Beweisen und deren Validierung auf die Benchmark-Tests der QBFEVAL-Berwerbe 2008 und 2010 an und präsentieren eine ausführliche Evaluierung der Ergebnisse. Unsere Ergebnisse zeigen, dass über 95% aller gelösten Instanzen innerhalb der gegebenen Zeit- und Speichereinschränkungen mit Hilfe von QRPcheck validiert werden konnten und alle validierten Instanzen korrekt von DepQBF gelöst wurden.

Contents

1 Introduction **1**

2 Preliminaries **5**
- 2.1 Quantified Boolean Formulas 5
- 2.2 Q-Resolution 9
- 2.3 Q-Resolution Proofs 10

3 QBF Solving in DepQBF **13**
- 3.1 QDLL with Learning 13
- 3.2 Learning: CDCL and SDCL 16

4 Proof Extraction **23**
- 4.1 The QRP File Format 23
- 4.2 Tracing in DepQBF 26

5 QRPcheck **29**
- 5.1 QRPcheck Overview 29
- 5.2 Checking Q-Resolution Proofs 31
- 5.3 Checking Initial Cubes 39

6 Experimental Results **45**

7 Conclusion **55**

Chapter 1

Introduction

The logic of quantified Boolean formulas (QBF) is a powerful generalization of propositional logic and supports compact encodings for real world problems, e.g., in the field of Formal Verification [5, 7, 14, 32], Reasoning [15] and Artificial Intelligence [18, 27, 28]. QBF are basically propositional formulas extended by universal and existential quantifiers—however, in constrast to the NP-complete satisfiability problem for propositional logic (SAT), the satisfiability problem for QBF (QSAT) is PSPACE-complete. In the same way SAT is considered as *the* prototypical NP-complete problem, QSAT is considered as *the* prototypical PSPACE-complete problem, and therefore assumed to be considerably more complex. Even so, QBF encodings become increasingly attractive as many problems (such as formulations of safety properties in bounded model checking [14]) have QBF encodings that are exponentially more compact than their corresponding SAT encodings. Hence, in order to exploit the potential of QBF efficient automated reasoning tools are required.

In recent years, the development of efficient QBF-solvers has progressed considerably and even though the achievements are still far from the progress made in the domain of SAT-solving, various efficient decision procedures and automated reasoning tools for QBF have been proposed [11, 12, 19, 17, 23, 24, 36, 37]. However, most current state-of-the-art QBF-solvers give mere *true/false* answers to a given problem, which often proves to be insufficient for various reasons. If the result of the satisfiability problem for a given QBF is unknown, verifying its validity has to be based on a majority vote of different disagreeing solvers, which is a guess rather than an actual verification. There's no way to guarantee that the outcome of such a vote is indeed correct. Further, in many real world applications of Formal Verification (such as bounded model checking) the result of a solved instance should provide a concrete solution, e.g., a basis for counterexamples to be able to identify error traces. Similarly, in the field of Artificial Intelligence, witnesses of a QBF-solver's result do not only prove the validity of the result

but further permit to identify strategies in game-like scenarios. Hence, it is a highly requested feature for QBF-solvers to provide the possibility to extract so-called *certificates of (un)satisfiability*.

Over the last 10 years, various QBF-solvers have been extended to support certification and validation of their results (see [25] for an overview of the status-quo 2009). Search-based solvers such as yQuaffle [35] and QuBE [17] have been instrumented to generate resolution proofs for satisfiable and unsatisfiable instances during the search. In both cases, independent proof checkers for validating the correctness of the corresponding proofs were employed. However, checking resolution proofs as in [35] suffers exponential worst-case behaviour due to non-optimal specifics of the proof format employed [34]. Approaches for other QBF-solvers such as EBDDRES [21], Squolem [21] and sKizzo [4], which are not based on search-based decision procedures, mainly suffer from the fact that they do not provide dual means to certify satisfiable and unsatisfiable instances. EBDDRES and Squolem produce Q-refutations for unsatisfiable instances, whereas sKizzo only supports the extraction of a so-called unsatisfiable core, i.e., an unsatisfiable subset of the input formula. For satisfiable instances, EBDDRES, Squolem and sKizzo represent certificates with so-called Skolem functions [16], i.e., functions that define each existential variable of the input formula over its resp. dominating universal variables. In any case, most of these QBF-solvers and their corresponding certification tools are currently unmaintained. In a more recent approach, the circuit-based QBF-solver CirQit has been extended to produce Q-refutations for both satisfiable and unsatisfiable instances [20]. However, CirQit cannot exploit its full strength on instances that were originally given in prenex conjunctive normal form (PCNF).

A promising dual approach for extracting QBF certificates of satisfiability and unsatisfiability has recently been presented in [3]. Given a resolution proof (un)satisfiability, it is possible to extract a Skolem function-based (resp. in its dual form, Herbrand function-based) QBF certificate of (un)satisfiability, while decoupling certificate extraction and the actual solving process. This is especially desirable for search-based QBF-solvers, where—in contrast to Skolemization-based QBF-solvers—certification based on Skolem (resp. Herbrand) functions is not directly applicable. Further, Skolem (resp. Herbrand) function-based QBF certificates provide the benefit of a uniform representation of the (counter)model of a given input formula.

In this thesis, we extended the state-of-the-art QBF-solver DepQBF [23, 24] in order to extract resolution proofs of (un)satisfiability, which serve as a base for certificate extraction based on [3] as described in [26]. Figure 1.1 illustrates the complete certification workflow for DepQBF as employed in [26], for which we provide both the extraction and validation of the resolution proofs in question as follows.

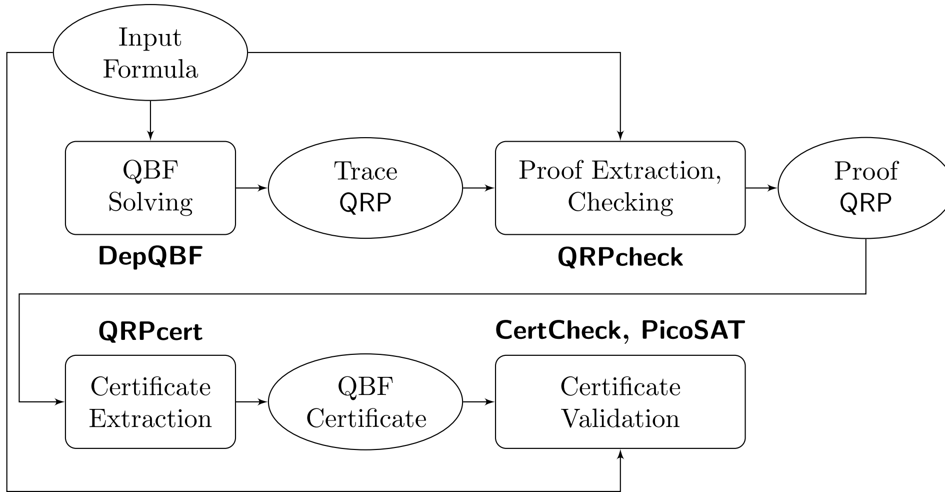


Figure 1.1: Certification workflow for the QBF-solver DepQBF.

DepQBF is a dependency-aware search-based solver for QBF in PCNF, which placed first in the main track of the QBFEVAL[1] competition 2010 and implements the Davis-Logeman-Loveland algorithm for QBF (QDLL) [11, 12] with clause and cube learning as in [17, 36, 37]. Based on [35], we instrumented DepQBF to record traces in our novel QRP format, a lightweight, text-based representation for resolution-based traces and proofs. From these QRP traces, we then extract and validate the corresponding resolution proofs with our proof checker QRPcheck. Given that such a QRP proof proved to be correct, a corresponding Skolem (resp. Herbrand) function-based QBF certificate is then extracted (QRPcert) and validated (CertCheck, PicoSAT) as described in [26].

This thesis is structured as follows. First, we introduce preliminaries necessary for the following chapters and give a brief overview of QBF solving as implemented in DepQBF. Our QRP format is introduced in Chapter 4, where we further describe how resolution-based traces are recorded by DepQBF. In Chapter 5 we introduce our proof checker QRPcheck in detail. An extensive evaluation of its results on various benchmark sets is given in Chapter 6. Chapter 7 finally discusses open problems and future work.

[1] http://www.qbflib.org/index_eval.php

Chapter 2

Preliminaries

2.1 Quantified Boolean Formulas

2.1.1 Syntax

A quantified Boolean formula (QBF) is a propositional formula with one or more variables bound by the quantifier *forall* ($\forall$) or *exists* ($\exists$). With no further limitations on the expressiveness of quantified Boolean formulas we restrict the set of logical connectives permitted to construct the propositional formula to *negation* ($\neg$), *conjunction* ($\wedge$), and *disjunction* ($\vee$). We define the set of quantified Boolean formulas as in [8] as follows.

Definition 2.1 (QBF). *The set of QBF is inductively defined as:*

1. *Propositional variables and the Boolean constants $\top$ and $\bot$ are quantified Boolean formulas.*
2. *If formulas ψ_1 and ψ_2 are quantified Boolean formulas, then $\neg\psi_1$, $\psi_1 \vee \psi_2$ and $\psi_1 \wedge \psi_2$ are quantified Boolean formulas.*
3. *If formula ψ is a quantified Boolean formula, then $\exists x.\psi$ and $\forall x.\psi$ are quantified Boolean formulas.*
4. *Only formulas as given in (1) to (3) are quantified Boolean formulas.*

Given a QBF ψ, the set of all variables in ψ is denoted as $var(\psi)$ and the set of all universally and existentially quantified variables in ψ is denoted as $var_\forall(\psi)$ and $var_\exists(\psi)$, respectively. Given a variable $x \in var(\psi)$, a *literal* l is either its *positive* occurrence x or its *negative* occurrence $\neg x$. We denote variable x by `lit2var(l)`. Given a QBF $\psi = \forall x.\phi$ (resp. $\psi = \exists x.\phi$), the occurrence of x in $\forall x.\phi$ (resp. $\exists x.\phi$) is called *quantified occurrence* and ϕ denotes the *scope* of the quantified variable x. If x occurs within the scope of $\forall x$ (resp. $\exists x$), we say that the occurrence of x is *bound.* All occurrences of variable x that are not bound are *free* occurrences. Variable x is called *free* (resp. *bound*) in ψ if there is a free (resp. bound) occurrence of x in ψ. If $var(\psi)$ does not contain any free variables, ψ is called a *closed* formula.

In the following, we require QBF to be closed and given in so-called *Prenex Normal Form (PNF)*. A QBF in PNF is in the form $Q.\phi$, where Q is a sequence of quantified variables denoted as the *prefix*, and ϕ is a propositional formula denoted as the *matrix* of the formula. Note that every QBF in non-prenex form can be transformed into PNF. A simple approach for such transformations is given in [8].

Definition 2.2 (PNF)**.** *Let ψ be a QBF in Prenex Normal Form (PNF):*

$$\psi := Q.\phi := \underbrace{q_1 X_1 \; q_2 X_2 \; ... \; q_n X_n}_{prefix} . \; \underbrace{\phi(x_1, x_2, ..., x_n)}_{matrix}$$

where q_i is a quantifier in $\{\forall, \exists\}$ and $q_i \neq q_{i+1}$ for $1 \leq i < n$, $X_i \subseteq var(\psi)$ is a set of variables bound by quantifier q_i at nesting level i and ϕ is a propositional formula over variables $x \in var(\psi)$.

Prefix Q is linearly ordered by nesting level, such that X_1 (resp. X_n) is at the *outermost* (resp. *innermost*) nesting level of formula ψ. We say that X_i is *outer* to X_{i+1}, i.e., $X_i < X_{i+1}$ and X_i *precedes* (resp. *dominates*) X_{i+1}. For the matrix ϕ we distinguish several different normal forms — in particular the *Conjunctive Normal Form (CNF)* and the *Disjunctive Normal Form (DNF)*, which are defined as follows.

Definition 2.3 (CNF)**.** *Let ϕ be a propositional formula. Let a clause $c_i = l_1 \vee ... \vee l_m$ be a disjunction of literals. If ϕ is a conjunction of clauses such that $\phi = c_1 \wedge ... \wedge c_n$, then ϕ is in Conjunctive Normal Form (CNF).*

Definition 2.4 (DNF)**.** *Let ϕ be a propositional formula. Let a cube $c_i = l_1 \wedge ... \wedge l_m$ be a conjunction of literals. If ϕ is a disjunction of cubes such that $\phi = c_1 \vee ... \vee c_n$, then ϕ is in Disjunctive Normal Form (DNF).*

Based on the normal form of the matrix we distinguish the following normal forms for quantified Boolean formulas in PNF:

Definition 2.5 (PCNF/PDNF)**.** *Let $\psi = Q.\phi$ be a QBF in PNF and let ϕ be the matrix of ψ. If ϕ is in CNF (resp. DNF), then ψ is in Prenex Conjunctive Normal Form (PCNF) (resp. Prenex Disjunctive Normal Form (PDNF)).*

Example 2.1. As an example for a quantified Boolean formula in PCNF consider Formula 2.1.

$$\psi = \forall x \, \exists y \, . \, (x \vee y) \wedge (\neg x \vee \neg y) \tag{2.1}$$

The set of variables in ψ is defined as $var(\psi) = \{x, y\}$, where x is universally and y existentially quantified. The nesting level of x is outer to the nesting level of y, i.e., x precedes (resp. dominates) y. The matrix of ψ is a propositional formula in CNF and consists of the clauses $(x \vee y)$ and $(\neg x \vee \neg y)$.

Furthermore, ψ is a closed formula. Note that Formula 2.1 represents the logical operation *exclusive or* (*xor*) and can be represented in PDNF as $\psi = \forall x \exists y.(x \wedge \neg y) \vee (\neg x \wedge y)$.

In the following, we often interpret the matrix of a QBF in PCNF (resp. PDNF) as a *set* of clauses (resp. cubes). Further, note that every propositional formula can be transformed into CNF resp. DNF by applying a set of simplification and transformation rules as shown in [8].

2.1.2 Semantics

The satisfiability problem for QBF (also denoted as QSAT) is the problem of deciding whether a quantified Boolean formula is satisfiable or unsatisfiable. As shown in [33], the satisfiability problem for closed QBF is PSPACE-complete. Note that the satisfiability problem for propositional logic (also denoted as SAT) is NP-complete, which is assumed to be considerably less complex. SAT can be regarded as a special case of QSAT with all variables considered to be existentially quantified.

Definition 2.6 (Satisifiability of Propositional Formulas)**.** *Let formulas ϕ, ϕ_1, and ϕ_2 be propositional formulas and let assignment α be a mapping $\alpha : var(\phi) \longrightarrow \{true, false\}$ from variables to truth values. The satisfiability of ϕ is recursively defined as follows.*

1. *If ϕ is a constant in $\{\top, \bot\}$, then ϕ is true (satisfied) iff $\phi = \top$.*
2. *If $\phi = x$ and x is a variable, then ϕ is true iff $\alpha(x) = true$.*
3. *If $\phi = \neg\phi_1$, then ϕ is true iff ϕ_1 is false.*
4. *If $\phi = \phi_1 \wedge \phi_2$, then ϕ true iff ϕ_1* and *ϕ_2 are true.*
5. *If $\phi = \phi_1 \vee \phi_2$, then ϕ is true iff ϕ_1* or *ϕ_2 is true.*
6. *If there exists an assignment α such that ϕ is true, then ϕ is satisfiable.*

In the following, we refer to α as *satisfying assignment* or *satisfiability model* of ϕ, if ϕ is satisfied under α.

Example 2.2. As an example, consider the satisfiable propositional formula $\phi = (x \vee y) \wedge (\neg x \vee \neg y)$ given as the matrix of Formula 2.1. Formula ϕ is either satisfied if $\phi[x/\top, y/\bot]$ or $\phi[x/\bot, y/\top]$, i.e., under the assignment $\{x = \top, y = \bot\}$ or $\{x = \bot, y = \top\}$.

Definition 2.7 (Satisfiability of QBF)**.** *Let $\psi = Q.\phi$ be a QBF in PNF. Based on Definition 2.6, the satisfiability of ψ is defined as follows.*

1. *$\exists x.\phi$ is satisfiable iff $\phi[x/\bot]$ is satisfiable* or *$\phi[x/\top]$ is satisfiable.*
2. *$\forall x.\phi$ is satisfiable iff $\phi[x/\bot]$ is satisfiable* and *$\phi[x/\top]$ is satisfiable.*

where $\phi[x/\bot]$ (resp. $\phi[x/\top]$) denotes the simultaneous substitution of all occurrences of x by $\bot$ (resp. $\top$) and of all occurrences of $\neg x$ by $\top$ (resp. $\bot$).

In contrast to satisfiability models in propositional logic, QBF satisfiability models have a tree-like structure due to the fact that an assignment $var_{\exists}(\psi) \longrightarrow \{\top, \bot\}$ depends on dominating universally quantified variables. Hence, the satisfiability model for quantified Boolean formulas can be represented as a set of satisfying assignments (resp. Boolean functions), which we often interpret as *satisfying assignment tree.* We introduce assignment trees as in [30] as follows.

Definition 2.8 (Assignment Tree)**.** *Let $\psi = Q.\phi$ be a QBF in PNF and let $var(\psi)$ be ordered with respect to the quantifier ordering of prefix Q. Let T be a tree of truth assignments with the empty truth assignment as root and let every node in T represent an assignment of a truth value to a variable $v \in var(\psi)$ with respect to the ordering of $var(\psi)$. An assignment tree T represents the set of all possible assignments $var(\psi) \longrightarrow \{\top, \bot\}$ such that each path from the root to a leaf corresponds to an assignment α and each leaf represents the truth value of ψ under α.*

Definition 2.9 (Satisfying Assignment Tree)**.** *Let ψ be a QBF in PNF and let T be the assignment tree of ψ. The satisfying assignment tree T' of ψ is a subtree of T with the empty truth assignment as root such that every universal (resp. existential) node in T' has exactly one (resp. none) sibling and every path from the root to a leaf represents a satisfying assignment.*

In the following, we refer to the satisfying assignment tree of a QBF ψ as the *QBF satisfiability model (QBF-model)* of ψ.

Example 2.3. As an example, consider the assignment tree for Formula 2.1 as shown in Figure 2.1. Given prefix $\forall x \exists y$, variable y is existentially quantified and dominated by universal variable x. Hence Formula 2.1 is satisfied iff we find an assignment $\alpha(y)$ for both assignments $\alpha(x) = \top$ and $\alpha(x) = \bot$ such that ψ is true. Therefore, Formula 2.1 is satisfied under the set of satisfying assignments $A = \{\{x = \top, y = \bot\}, \{x = \bot, y = \top\}\}$. The satisfying assignment tree representing A is extracted from the assignment tree given in Figure 2.1 by omitting all paths denoted in gray. Given satisfying assignment tree is a QBF satisfiability model of ψ. Note that the satisfiability model for the propositional formula given as the matrix of Formula 2.1 would be any assignment $\alpha \in A$, i.e., any path in the assignment tree that represents a satisfying assignment.

Note that even though the satisfiability problem for QBF is PSPACE-complete, the model-checking problem for QBF, i.e., the problem of deciding if a given set of Boolean functions is a QBF-model for QBF ψ, is coNP-complete [10, 8].

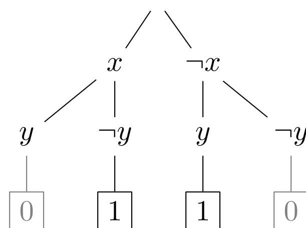


Figure 2.1: Assignment tree for Formula 2.1.

2.2 Q-Resolution

Q-Resolution is a sound and complete decision procedure for quantified Boolean formulas and has been first introduced for quantified Boolean formulas in PCNF as an extension of the resolution calculus for propositional logic in [9]. In propositional logic, we define *clause resolution* and its dual counterpart *cube* (or *term*) *resolution* as follows.

Definition 2.10 (Clause/Cube Resolution)**.** *Let ϕ be a propositional formula in CNF (resp. DNF) and let c_1, c_2 be clauses (resp. cubes) in ϕ. If there exists a variable $x \in var(\phi)$ such that $x \in c_1$ and $\neg x \in c_2$, x is denoted as pivot variable and resolving c_1 and c_2 over x yields the resolvent $c_r = (c_1 \cup c_2) \backslash \{x, \neg x\}$.*

Proposition 2.1 ([29])**.** *Resolvent c_r is a logical consequence of its antecedents c_1 and c_2, i.e., given an assignment α such that c_1 and c_2 are satisfied, c_r is also satisfied. Hence, c_r can be added to the formula without changing its satisfiability.*

For quantified Boolean formulas, the propositional resolution calculus has to be generalized with respect to universal and existential quantifiers, i.e., for QBF in PCNF (resp. PDNF) we require the pivot variable to be existentially (resp. universally) quantified.

Definition 2.11 (Clause/Cube Resolution for QBF)**.** *Let $\psi = Q.\phi$ be a QBF in PCNF (resp. PDNF) and let c_1, c_2 be clauses (resp. cubes) in matrix ϕ. If there exists a variable $x \in var_\exists(\psi)$ (resp. $var_\forall(\psi)$) such that $x \in c_1$ and $\neg x \in c_2$, x is denoted as pivot variable and resolving c_1 and c_2 over x yields $c_r = (c_1 \cup c_2) \backslash \{x, \neg x\}$. If c_r is non-tautological (resp. non-contradictory), then c_r is a resolvent. Otherwise, no resolvent exists.*

We further require the application of a general simplification rule for quantified Boolean formulas as introduced in [2, 8, 9] in each resolution step.

Definition 2.12 (Forall-/Existential-Reduction). *Let $\psi = Q.\phi$ be a QBF in PCNF (resp. PDNF) and let c be a non-tautological clause (resp. non-contradictory cube) in matrix ϕ. Let l be the nesting level of the innermost existentially (resp. universally) quantified literal in c. All universally (resp. existentially) quantified literals with a nesting level greater than l may be deleted from c without changing formula ψ's satisfiability.*

Based on Definition 2.11, we introduce *Q-Resolution* for quantified Boolean formulas in PCNF (resp. PDNF) as introduced in [9] (resp. [19]) as follows.

Definition 2.13 (Q-Resolution). *Let $\psi = Q.\phi$ be a QBF in PCNF (resp. PDNF) and let c_1, c_2 be non-tautological clauses (resp. non-contradictory cubes) in matrix ϕ. If there exists a pivot variable $x \in var_\exists(\psi)$ (resp. $var_\forall(\psi)$) such that $x \in c_1$ and $\neg x \in c_2$, we obtain a Q-resolvent by applying the following steps:*

1. *Reduce c_1, c_2 by forall- (resp. existential-) reduction and obtain c'_1, c'_2.*
2. *Resolve c'_1 and c'_2 over x and obtain the $c_r = (c'_1 \cup c'_2) \setminus \{x, \neg x\}$.*
   If c_r is non-tautological (resp. non-contradictory), c_r is a resolvent.
3. *Reduce resolvent c_r by forall- (resp. existential-) reduction and obtain the Q-resolvent c'_r.*

Note that in contrast to the resolution rule in propositional logic, the Q-resolution rule is not sound if the generation of tautological clauses (resp. contradictory cubes) is not explicitly excluded:

Example 2.4. As an example, consider the satisfiable quantified Boolean formula in PCNF given as Formula 2.1. Resolving the clauses $(x \vee y)$ and $(\neg x \vee \neg y)$ over y yields the tautological clause $c_r = (x \vee \neg x)$. We apply forall-reduction to c_r, obtain the empty clause, and derive by Theorem 2.1 (see Section 2.3) that ψ is unsatisfiable, which in fact it is *not*.

In the following, we denote a clause resp. cube as "constraint" and further use the terms "resolution" and "Q-resolution" interchangeably unless otherwise noted.

2.3 Q-Resolution Proofs

The satisfiability (resp. unsatisfiability) of a quantified Boolean formula can be shown by a sequence of Q-resolution steps—also referred to as *Q-resolution proof of satisfiability* (resp. *Q-resolution proof of unsatisfiability* or *Q-refutation*)—as follows [9, 19].

Theorem 2.1. *A QBF in PCNF is unsatisfiable iff there exists a clause resolution sequence leading to the empty clause, i.e., a clause without literals.*

Theorem 2.2. *A QBF in PDNF is satisfiable iff there exists a cube resolution sequence leading to the empty cube, i.e., a cube without literals.*

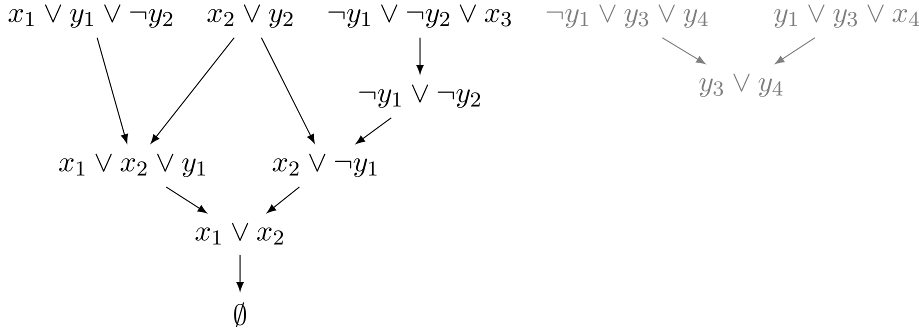


Figure 2.2: Q-resolution proof of unsatisfiability for Formula 2.2.

Given a QBF in PCNF, cube resolution is not directly applicable and thus not sufficient to prove the satisfiability of the formula. On the other hand, transformations from PCNF to PDNF are costly in terms of computation time and may further increase the size of the formula exponentially [22]. Hence, we introduce the *model generation rule* in order to derive a Q-resolution proof of satisfiability from a QBF in PCNF via cube resolution as follows [19].

Definition 2.14 (Model Generation Rule)**.** *Let $\psi = Q.\phi$ be a QBF in PCNF and let ϕ be the matrix of ψ. Let M be a set of cubes such that the disjunction of all cubes in M is propositionally logically equivalent to ϕ. M is a satisfiability model of ϕ iff each cube $t \in M$ is non-contradictory and for each clause $c \in \phi$, $t \cap c \neq \emptyset$.*

Theorem 2.3. *Cube resolution and model generation is a sound and complete proof system for deciding quantified Boolean formulas in PCNF. A QBF in PCNF is satisfiable iff the empty cube is derivable by cube resolution and model generation.*

We often interpret a Q-resolution proof as directed acyclic graph (DAG) with the empty clause (resp. cube) as root, the set of input clauses (resp. initial cubes) as leafs, and resulting clauses (resp. cubes) of intermediary proof steps as internal nodes. The set of *input clauses* is a subset of the set of clauses given as the matrix of the input formula ψ, whereas the set of *initial cubes* is a subset of the set of cubes given as the satisfiability model of the matrix of ψ. Even though we consider the application of the forall-/existential-reduction rule as part of the Q-resolution rule, we often treat it explicitly and distinguish three different types of proof steps: resolution steps with reduction (as in Definition 2.13), resolution steps without reduction (as in Definition 2.11), and reduction steps (as in Definition 2.12). Note that resolution steps always have two ancestor nodes, whereas reduction steps have only one. We also refer to a node's ancestors as its *antecedents*.

Example 2.5. As an example, consider the Q-resolution proof of unsatisfiability for Formula 2.2 as shown in Figure 2.2, where the empty clause is derived from the input clauses $(x_1 \vee y_1 \vee \neg y_2)$, $(x_2 \vee y_2)$, and $(\neg y_1 \vee \neg y_2 \vee x_3)$. Note that input clauses and steps denoted in gray are not required to derive the empty clause.

$$\forall x_1\, x_2\, \exists y_1\, y_2\, \forall x_3\, \exists y_3\, \forall x_4\, \exists y_4\, .\, (x_1 \vee y_1 \vee \neg y_2) \wedge (x_2 \vee y_2) \wedge (\neg y_1 \vee \neg y_2 \vee x_3) \wedge (\neg y_1 \vee y_3 \vee y_4) \wedge (y_1 \vee y_3 \vee x_4) \tag{2.2}$$

An example for a resolution step without reduction is the derivation of the clause $(x_1 \vee x_2 \vee y_1)$, whereas the clause $(\neg y_1 \vee \neg y_2)$ is obtained by a reduction step. The derivation of clause $(y_3 \vee y_4)$, which is not part of the Q-resolution proof, is an example for a resolution step with reduction.

Chapter 3

QBF Solving in DepQBF

The state-of-the-art QBF-solver DepQBF [23, 24] is a dependency-aware search-based solver for QBF in PCNF. It placed first in the main track of QBFEVAL'10[1] and implements an adaptation of the Davis-Logeman-Loveland (*DLL*) algorithm [13] with clause and cube learning as in [17, 36, 37]. In the following, we introduce the DLL algorithm for QBF (*QDLL*) [11] as implemented in DepQBF in more detail.

3.1 Algorithm Overview: QDLL with Learning

DepQBF implements an iterative version of the QDLL algorithm introduced in [11], extended by Conflict-Driven Clause Learning (*CDCL*) and Solution-Directed Cube Learning (*SDCL*) based on the approach introduced in [37]. *QDLL*, an extension of the *DLL* algorithm from SAT to QSAT, is a sound and complete search-based decision procedure for QBF in PCNF. Given a QBF ψ, QDLL traverses its assignment tree until either ψ proved to be unsatisfiable, or a satisfying assignment tree of ψ is derived. QDLL branches on variables by making *decisions* (on both a variable and its truth value), propagating implications of these decisions, and backtracking in case of a conflict resp. solution. Extending QDLL with CDCL and SDCL prunes the search space by generating implications from both conflicting and satisfying assignments encountered during the search. In the following, we introduce some basic notions before discussing the QDLL as implemented in DepQBF.

Definition 3.1 (Partial/Full Assignment)**.** *Given a QBF ψ and an assignment $\alpha(var(\psi))$, we say that α is a* full *assignment of ψ if all variables in $var(\psi)$ are assigned. Otherwise, assignment α is a* partial *assignment.*

Definition 3.2 (State of a Literal under an Assignment)**.** *Given a QBF ψ and a (partial) assignment $\alpha(var(\psi))$, we say that a literal is* satisfied *(resp.* unsatisfied*) if it is assigned and evaluates to true (resp. false).*

[1] `http://www.qbflib.org/index_eval.php`

In the following, we refer to the state of a formula with respect to the current (partial) assignment α and define the state of a clause under α as follows.

Definition 3.3 (State of a Clause under an Assignment)**.** *A clause is* satisfied *if one or more of its literals are satisfied;* falsified *or* conflicting *(empty) if all of its existential and all of its assigned universal literals are unsatisfied; and* undetermined *otherwise.*

If a clause is conflicting under an assignment α, then α is a conflicting assignment (*conflict*) and the current search branch is unsatisfiable. If all clauses of a formula are satisfied under α, then α is a satisfying assignment (*solution*) and the current search branch is satisfiable. Note that in case that the current branch represents a solution, in contrast to SAT we still have to check for all universal variables in the current assignment if both branches are satisfiable.

DepQBF keeps information derived from conflicts as *learnt clauses* and the knowledge of satisfying assignments as *learnt cubes* and maintains the input formula and all information learnt during the solving process in so called *Augmented CNF (ACNF)* [37]. That is, we interpret the prefix of the input formula and the set of input clauses, learnt clauses, and learnt cubes as a QBF in PNF with a matrix in ACNF, which is defined as follows.

Definition 3.4 (ACNF)**.** *Let ϕ be a propositional formula and let $\{c_1, ..., c_n\}$ be a set of clauses and let $\{C_1, ..., C_m\}$ be a set of cubes such that $C_i \Rightarrow c_1 \wedge ... \wedge c_n$ for $1 \leq i \leq m$. If $\phi = (c_1 \wedge ... \wedge c_n) \vee (C_1 \vee ... \vee C_m)$, then ϕ is in Augmented CNF.*

We define the state of a cube under the current (partial) assignment based on Definition 3.4 and further introduce the following implication rules for QBF in ACNF.

Definition 3.5 (State of a Cube under an Assignment)**.** *A cube is* falsified *if one or more of its literals are unsatisfied;* satisfied *or* satisfying *(empty) if all of its universal and all of its assigned existential literals are satisfied; and* undetermined *otherwise.*

Definition 3.6 (Unit Literal)**.** *Let $\psi = Q.\phi$ be a QBF in PNF with a matrix in CNF (resp. DNF). Let $c \in \phi$ be an undetermined clause (resp. cube) in matrix ϕ. If literal $l \in c$ is the unassigned occurrence of $x \in var_{\exists}(\psi)$ (resp. $x \in var_{\forall}(\psi)$) such that all other unassigned literals in c are universally (resp. existentially) quantified and dominated by x, then l is a unit literal and c is unit.*

Definition 3.7 (Unit Rule)**.** *Let $\psi = Q.\phi$ be a QBF in PNF with a matrix in ACNF and let $c \in \phi$ be unit under the current partial assignment $\alpha(var(\psi))$ with unit literal $l \in c$. If formula ψ is satisfiable under α, then variable $x =$ `lit2var(`l`)` is assigned such that l is satisfied.*

```
1  function qdll ()
2  {
3    loop
4    {
5      state ← simplify ()
6
7      if (state = UNDEFINED)
8      {
9        var ← select_decision_variable ()
10       assign_decision_variable (var)
11     }
12     else  /* conflict or solution */
13     {
14       btlevel ← analyze (state)
15
16       if (btlevel = INVALID)
17         return state
18       else
19         backtrack (btlevel)
20     }
21   }
22 }
```

Figure 3.1: Top-level view of the QDLL as implemented in DepQBF.

We say that a variable is *implied*, if the application of the unit rule forces the assignment of its truth value. In the following, we refer to a unit clause that implies a variable as its *antecedent* (which is not to be confused with a proof step's antecedent).

Definition 3.8 (Pure Literal). *Let $\psi = Q.\phi$ be a QBF in PNF and let $x \in var(\psi)$. If all occurrences of x in matrix ϕ are either only positive (x) or negative ($\neg x$), x is pure in ψ and all occurrences of x are referred to as pure literals.*

Definition 3.9 (Pure Literals Rule). *Let $\psi = Q.\phi$ be a QBF in PNF with a matrix in ACNF and let $x \in var(\psi)$ be pure. If $x \in var_{\exists}(\psi)$, we assign x such that all occurrences of x are satisfied. If $x \in var_{\forall}(\psi)$, we assign x such that all occurrences of x are unsatisfied.*

Figure 3.1 describes a top-level view of the QDLL algorithm implemented in DepQBF as introduced in [24]. The core of the algorithm is the function `simplify`, where implications from unit and pure literals are propagated until saturation. If neither a conflict nor solution was found, the state of

```
1  function analyze (STATE state)
2  {
3    reason = get_initial_reason ()
4
5    while (not stop_criterion_is_met (reason))
6    {
7      pivot ← get_pivot (reason)
8      antecedent ← get_antecedent (pivot)
9      reason ← resolve (reason, pivot, antecedent)
10   }
11   add_to_formula (reason)
12   return get_asserting_level (reason)
13 }
```

Figure 3.2: Analysis of conflict resp. solution by resolution.

the formula is `UNDEFINED` under the current (partial) assignment and we make a *decision*, i.e., we select and assign a variable `var` with respect to the quantifier ordering of the prefix, and continue the search. Note that we associate each decision with a *decision level* (starting from 1) and all implications following from a decision with the same decision level as the *decision variable* `var`. Further, note that assignments implied without decisions are associated with decision level 0 (*top level*).

In case of a conflict (resp. solution), we analyze the current state and redirect the search—if possible—by adding a learnt constraint and backtracking to decision level `btlevel` (referred to as *backtracking level*). If the learnt constraint is empty (`btlevel = INVALID`), `qdll` terminates and the formula is unsatisfiable (resp. satisfiable).

3.2 Learning: CDCL and SDCL

QDLL-based QBF-solvers are forced to backtrack in order to continue the search not only in case of a conflict, but also if a satisfying assignment was found. Hence, in both cases, the current assignment provides valuable knowledge for implications to be used for future reasoning. DepQBF implements conflict-driven (CDCL) and satisfiability-directed (SDCL) learning strategies based on an approach with a dual view on conflicting and satisfying assignments as introduced in [37]. CDCL for QBF [36] is based on CDCL strategies for SAT-solving (e.g., [31]) and is employed whenever a conflicting assignment is encountered during the search. Similarly, SDCL for QBF [37] is performed whenever a satisfying assignment was found. In the following, we briefly discuss CDCL and SDCL as implemented in DepQBF.

Conflict-driven and satisfiability-directed constraint learning as implemented in DepQBF is initiated whenever the state of the formula is either conflicting or satisfied under the current (partial) assignment (line 12, Figure 3.1). The core algorithm for analyzing the current state of the formula is described in Figure 3.2. Given the current assignment, function `analyze` first determines the reason for the current state (function `get_initial_reason`), derives a learnt constraint (lines 5-10, Figure 3.2) and adds it to the solver database (function `add_to_formula`), and finally determines the backtracking level (function `get_asserting_level`) for the current conflict (resp. solution). Note that in case of a conflict, the reason of the current state is a conflicting clause, whereas in case of a solution, the reason is a satisfying cube, which is either an existing cube in the matrix or generated in `get_initial_reason` by determining a so called *cover set* of the current satisfying assignment.

Definition 3.10 (Cover Set)**.** *Given a QBF ψ in PNF with a matrix in ACNF. A cover set C of ψ is a set of literals satisfied under the current assignment such that for each clause c in the matrix $c \cap C \neq \emptyset$.*

Example 3.1. As an example, consider the satisfiable QBF in ACNF given as Formula 3.1. Assignment $\alpha = \{(x_1, \top), (x_2, \bot), (y_1, \top), (y_2, \bot), (y_3, \bot)\}$ is a satisfying assignment. The set of literals $\{\neg x_2,\ y_1, \neg y_2\}$ is a cover set for Formula 3.1 under α. Hence, cube $(\neg x_2 \wedge y_1 \wedge \neg y_2)$ is a satisfying cube and may be added to the matrix as learnt constraint. The resulting formula is given as Formula 3.2.

$$\begin{aligned} \forall x_1\, x_2\, \exists y_1\, y_2\, y_3\, .\, &(\neg x_1 \vee y_1 \vee \neg y_2) \wedge (\neg x_1 \vee \neg x_2 \vee y_2 \vee y_3) \wedge \\ &(x_2 \vee \neg y_2 \vee \neg y_3) \wedge (\neg x_1 \vee \neg x_2 \vee y_1 \vee y_3) \end{aligned} \tag{3.1}$$

$$\begin{aligned} \forall x_1\, x_2\, \exists y_1\, y_2\, y_3\, .\, &(\neg x_1 \vee y_1 \vee \neg y_2) \wedge (\neg x_1 \vee \neg x_2 \vee y_2 \vee y_3) \wedge \\ &(x_2 \vee \neg y_2 \vee \neg y_3) \wedge (\neg x_1 \vee \neg x_2 \vee y_1 \vee y_3) \vee \\ &\qquad\qquad\qquad\qquad (\neg x_2 \wedge y_1 \wedge \neg y_2) \end{aligned} \tag{3.2}$$

Note that cube $(\neg x_2 \wedge y_1 \wedge \neg y_2)$ could be further reduced by applying existential-reduction. We refer to an existential-reduced cover set as *non-covering set*. Further, note that given a satisfying assignment α of a QBF ψ, the cover set of ψ under α is not unique.

Once the reason of a conflict (resp. solution) is determined, in function `analyze` we generate a learnt constraint by iteratively resolving the reason with the antecedent (function `get_antecedent`) of one of its variables (function `get_pivot`) until a predefined stop criterion is met. In each iteration, function `get_pivot` chooses an implied variable in reverse chronological order as pivot variable for the resolution (i.e., the variable implied last will be chosen first). The reason is then resolved with the antecedent of the pivot

variable and if the resulting constraint is *asserting*, we say that the *stop criterion* is fulfilled and add the asserting constraint as *learnt constraint* to the formula. If the asserting constraint is the empty clause (resp. cube), `qdll` terminates and the formula is unsatisfiable (resp. satisfiable).

Note that following the exact reverse chronological order is not always possible (for further details see [19]). Further, unlike [36, 37], DepQBF does not produce tautologies resp. contradictions as learnt constraints.

Definition 3.11 (Asserting Constraint)**.** *A clause (resp. cube) c is asserting if c is empty and there exists exactly one asserted literal l such that*

1. *l is assigned and existential (resp. universal), and at a decision level n such that all other assigned existential (resp. universal) literals are at a decision level smaller than n*
2. *decision variable v at decision level n is existential (resp. universal)*
3. *all universal (resp. existential) literals preceding l are assigned and unsatisfied (resp. satisfied), and at a decision level smaller than n*

Definition 3.12 (Asserting Level)**.** *Let clause resp. cube c be an asserting constraint and let $l \in c$ be the asserted literal at decision level n. Let L_e be the set of all assigned existential literals in c and let L_a be the set of all assigned universal literals in c that precede l. Let $L = L_e \cup L_a$ be the set of all assigned existential and universal literals at a decision level smaller than n (see Definition 3.11). We say that decision level N is asserting if it is the maximum decision level of all literals in L.*

Note that the asserting level is the backtracking level of the current state. Further, note that after backtracking to asserting level, an asserting constraint will become unit.

Example 3.2. As an example for determining the asserting level, consider a QBF $\psi = \exists y_1 \forall x_1 \exists y_2 . \phi$ and the asserting clause $c = (y_1 \vee x_1 \vee y_2)$. Given that y_1, x_1 and y_2 are assigned at decision level $dl_{y_1} = 5$, $dl_{x_1} = 3$ and $dl_{y_2} = 1$ respectively, the asserted literal is y_1. The maximum decision level of all literals in $L = \{y_2\}$ is dl_{y_2}, hence the asserting level is decision level 1. Note that c is unit at asserting level even though we do not consider universal literals preceded by y_1 (here: x_1) as they are eliminated by forall-reduction after backtracking. On the other hand, given the decision levels $dl_{y_1} = 1$, $dl_{x_1} = 3$ and $dl_{y_2} = 5$ with asserted literal y_2 at decision level 5, the maximum decision level of $L = \{y_1, x_1\}$ is dl_{x_1} and the asserting level is decision level 3.

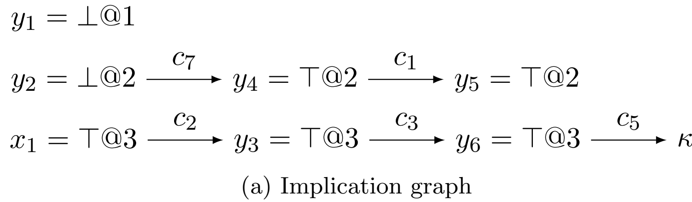
Example 3.3. As an example for the application of QDLL with Learning, consider the unsatisfiable QBF given as Formula 3.3. In the following, we describe the current assignment and implications following from the current assignment as an *implication graph*, which we will briefly introduce as in [22] as follows. An implication graph is a DAG with the set of literals of the

current assignment as its nodes, and the set of implications as its edges. Given a node n representing a literal l in an implication graph G, n is labelled with l and its decision level dl, written as $l@dl$. Given an edge $e = (n_i, n_j)$ with nodes $n_i, n_j \in G$ representing the literals l_i, l_j, we say that n_i is the antecedent of n_j and label edge e with the clause that implied l_j.

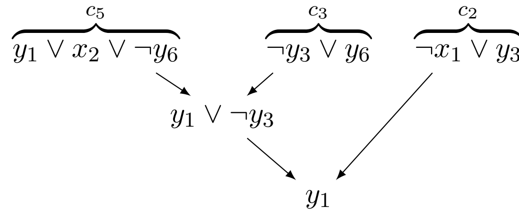
$$\begin{aligned}\exists\, y_1\, y_2\, \forall\, x_1\, \exists\, y_3\, y_4\, y_5\, \forall\, x_2\, \exists\, y_6\, y_7\,.\ &\underbrace{(\neg y_4 \vee y_5)}_{c_1} \wedge \underbrace{(\neg x_1 \vee y_3)}_{c_2} \wedge \\ &\underbrace{(\neg y_3 \vee y_6)}_{c_3} \wedge \underbrace{(\neg y_6 \vee y_7)}_{c_4} \wedge \underbrace{(y_1 \vee x_2 \vee \neg y_6)}_{c_5} \wedge \\ &\underbrace{(\neg y_1 \vee x_1 \vee \neg y_5)}_{c_6} \wedge \underbrace{(y_2 \vee y_4)}_{c_7} \wedge \underbrace{(\neg x_2 \vee \neg y_7)}_{c_8}\end{aligned} \tag{3.3}$$

As we mainly want to illustrate the process of deriving learnt constraints, we ignore (initial) implications at decision level 0 and further assume that initially no variables are assigned. For the sake of simplicity we also ignore implications from pure literals and do not apply the pure literal rule. We start with the first decision and choose decision variable y_1 with respect to the quantification order of the prefix. Variable y_1 is assigned to $\top$, which satisfies clause c_1 but does not imply any further assignments.

Therefore, as shown in Figure 3.3a, we continue with the second decision and assign variable y_2 to $\bot$, which implies y_4 and y_5 (as clause c_7 and—as a consequence—clause c_1 become unit). At decision level 2, there are no further implications to be derived. Hence, we make our third decision, assign variable x_1 to $\top$, propagate all implications arising from that decision and encounter a conflict (indicated by node κ in the implication graph).

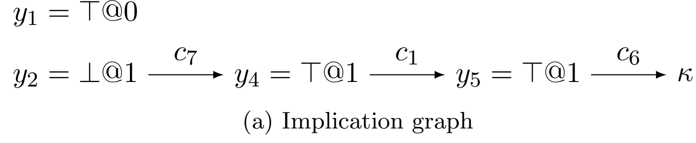


(a) Implication graph

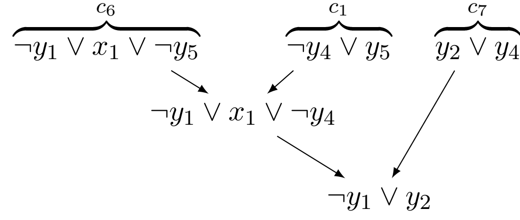


(b) Derivation of the first learnt constraint

Figure 3.3: First conflict in Example 3.3.

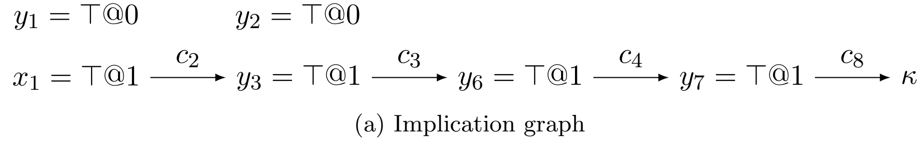


(a) Implication graph

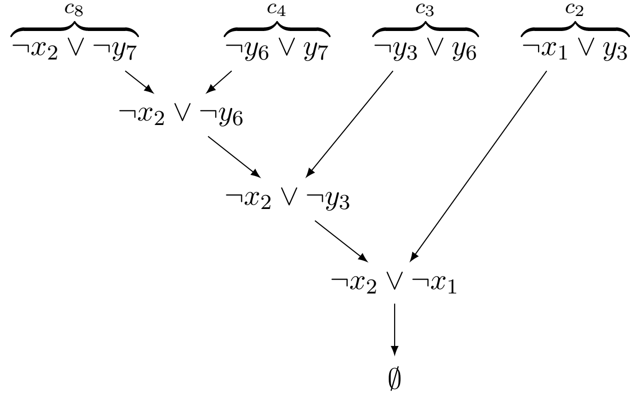


(b) Derivation of the second learnt constraint

Figure 3.4: Second conflict in Example 3.3.



(a) Implication graph



(b) Derivation of the empty clause

Figure 3.5: Third conflict in Example 3.3.

Clause c_5 is the reason for the conflict, therefore we resolve c_5 with the antecedents of its implied variables in reverse chronological order of their implication until the resolvent clause is asserting (as shown in Figure 3.3b). The resulting clause is (y_1), which is unit at asserting level 0. We add the clause to the formula and backtrack to decision level 0.

At backtracking level 0, the previously learnt constraint (y_1) is unit, hence implies y_1. With no further implied assignments at decision level 0, we propagate implications from decision $y_2 = \bot$ until we encounter a conflict (Figure 3.4a). This time, clause c_6 is conflicting and we derive clause $(\neg y_1 \vee y_2)$ by resolution (Figure 3.4b), which is—again—asserting at decision level 0. We add the asserting clause to the formula, backtrack to decision level 0, add implication y_2, choose x_1 as decision variable and assign it to $\top$ (Figure 3.5a). Again, propagating implications from this decision we encounter a conflict. We resolve the conflict clause c_8 with the antecedents of its implied variables and derive the empty clause (Figure 3.5b). Hence, we conclude that Formula 3.3 is unsatisfiable.

In the following, we introduce proof extraction on top of QDLL with CDCL and SDCL as implemented in DepQBF.

Chapter 4

Proof Extraction

We instrumented DepQBF to record a Q-resolution-based *trace* that represents all Q-resolution sequences generated by QDLL with CDCL and SDCL similar to the approach in [35]. From such a trace, it is possible to extract a Q-resolution proof of unsatisfiability (resp. satisfiability) by reconstructing the Q-resolution sequence leading to the empty clause (resp. cube). We represent Q-resolution-based traces and proofs in so-called QRP format, which we introduce as follows.

4.1 The QRP File Format

The QRP (Q-Resolution Proof) file format is a human-readable and easy-to-parse representation for Q-resolution proofs and Q-resolution-based traces based on the QDIMACS[1] input file format for QBF and the tracecheck[2] trace file format for SAT. In QRP format, a trace is described as a set of linked constraint nodes (cf. DAG representation of Q-resolution proofs in Chapter 2, Section 2.3), which represents a set of clause and cube resolution sequences with the sequence leading to the empty constraint as a subset. In the following, we refer to a constraint node as a *step* in a Q-resolution sequence. If this sequence represents a Q-resolution proof of (un)satisfiability, each step is referred to as *proof step*. Figure 4.1 describes the grammar of the QRP format in EBNF (Extended Backus-Naur Form), which is interpreted as follows.

`trace`. A `trace` in QRP format must provide a `preamble` (with a `header` statement and (optionally) a set of `comment` lines), the prefix of the input formula (represented by a set of quantified variables (`quant_set`)), a set of clause and cube resolution sequences (represented by a set of linked constraint nodes (`step`)), and a `result` statement (to indicate, if given proof is

[1] http://www.qbflib.org/qdimacs.html
[2] http://www.fmv.jku.at/tracecheck

```
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 4.1: The QRP format in Extended Backus-Naur Form (EBNF).

a proof of satisfiability (`sat`) or unsatisfiability (`unsat`)). Note that we do not explicitly distinguish between actual traces (where the set of steps may represent a *set* of clause *and* cube resolution sequences) and proofs (where the set of steps represents exactly *one* clause *or* cube resolution sequence, which leads to the empty constraint). Instead, we rather interpret the set of proof steps representing the Q-resolution proof of satisfiability resp. unsatisfiability depending on the result statement as a clause resp. cube resolution sequence.

`header`. Every file in QRP format must provide a `header` statement similar to the so-called `problem_line` in the QDIMACS format. The header statement denotes a file in QRP format with the QRP marker sequence `"p qrp"`, followed by two positive numbers. The first number indicates the number of variables in the input formula, whereas the second represents the number of steps of the trace. In the following, we will uniquely identify both variables and steps by indices and hence rather interpret both numbers as the respective maximum indices of variables and steps occurring in the trace.

`quant_set`. A quantified set (`quant_set`) represents a set of either existentially or universally quantified variables with respect to the quantifier ordering of the prefix of the input formula. As in the QDIMACS format, we denote the existential quantifier by `"e"` and the universal quantifier by `"a"`. We further require that each variable in the prefix is unique, consecutive quantified sets are not bound by the same quantifier, the innermost quantified set is existentially quantified, and free variables are considered to be existentially quantified in the outermost quantified set.

`step`. A step represents a linked constraint node and consists of an identifier (the step index `idx`), a set of `literals` (the actual constraint), and the step's `antecedents`. We distinguish between steps representing an input clause (resp. initial cube), steps representing the application of forall- (resp. existential-) reduction (explicit reduction steps), steps representing resolution with or without the application of forall- (resp. existential-) reduction (resolution steps), and steps representing the empty constraint.

`literals`. A step's constraint is represented as a set of `literals`. We do not further restrict this set of literals and allow occurrences of free variables.

`antecedents`. A step may have at most two `antecedents`, which are represented as a set of (unique) step indices. A step representing an input clause (resp. initial cube) has no antecedents, whereas a reduction (resp. resolution) step has exactly one (resp. two) antecedents.

`result.` The result statement indicates if the given Q-resolution proof is a proof of unsatisfiability or satisfiability. Note that in case of `unsat` (resp. `sat`), given proof is a clause (resp. cube) resolution sequence.

The QRP format is an explicit representation for resolution-based traces and proofs. Unlike [38] and [35], each resolution (resp. reduction) step has exactly two (resp. one) antecedent(s), which explicitly states the order of application of (forall- resp. existential-) reduction and resolution and avoids exponential worst-case behaviour when reconstructing resolvents from unordered lists of antecedents [34]. Alternatively, the QIR[3] format used in [20] allows multiple antecedents but predefines the order in which resolvents should be reconstructed. As far as resolution is concerned, QRP proofs can be extracted and checked in deterministic log space, a desirable property of proof formats suggested in [34]. Syntactically, QRP is a lightweight format as it does not distinguish between (resolution steps over) clauses and cubes explicitly.

4.2 Tracing in DepQBF

We instrumented DepQBF to record Q-resolution-based traces of all Q-resolution sequences generated by QDLL with CDCL and SDCL in QRP format in a similar manner as in [35] as follows.

1. We assign a unique *id* to every input clause and every constraint generated during the solving process (cf. step index in the QRP format).
2. The header statement of the QRP format, the prefix of the input formula, and all input clauses with their respective indices are recorded during the parsing process.
3. Every explicit application of forall- resp. existential-reduction during the parsing and solving process is recorded as explicit reduction step.
4. For each conflict (resp. solution) encountered, in function `analyze` the initial reason (i.e., the conflict clause resp. cover/non-covering set), the learnt constraint, and all resolvents generated in the process are recorded with their respective indices and antecedents.

Note that unlike [38] and [35], we explicitly encode the derivation of the empty clause (resp. cube) and the resolution order of any Q-resolution sequence in the trace. Hence, we do not encounter exponential worst case behaviour [34] when extracting a Q-resolution proof of (un)satisfiability from a trace in QRP format.

[3] `http://users.soe.ucsc.edu/~avg/ProofChecker/qir-proof-grammar.txt`

$$\underbrace{\exists y_1\, y_2\, \forall x_1\, \exists y_3\, y_4\, y_5\, \forall x_2\, \exists y_6\, y_7}_{\exists\, 1\, 2\, \forall\, 3\, \exists\, 4\, 5\, 6\, \forall\, 7\, \exists\, 8\, 9} \,.\, \underbrace{(\neg y_4 \vee y_5)}_{c_1\colon (\neg 5 \vee 6)} \wedge \underbrace{(\neg x_1 \vee y_3)}_{c_2\colon (\neg 3 \vee 4)} \wedge$$
$$\underbrace{(\neg y_3 \vee y_6)}_{c_3\colon (\neg 4 \vee 8)} \wedge \underbrace{(\neg y_6 \vee y_7)}_{c_4\colon (\neg 8 \vee 9)} \wedge \underbrace{(y_1 \vee x_2 \vee \neg y_6)}_{c_5\colon (1 \vee 7 \vee \neg 8)} \wedge$$
$$\underbrace{(\neg y_1 \vee x_1 \vee \neg y_5)}_{c_6\colon (\neg 1 \vee 3 \vee \neg 6)} \wedge \underbrace{(y_2 \vee y_4)}_{c_7\colon (2 \vee 5)} \wedge \underbrace{(\neg x_2 \vee \neg y_7)}_{c_8\colon (\neg 7 \vee \neg 9)}$$

Figure 4.2: Transformation of Formula 3.3 into QDIMACS format.

Example 4.1. As an example, consider the trace generated for the solving process described in Example 3.3 as shown in Figure 4.3b. The QDIMACS representation of the input formula ψ (Formula 3.3) is described in Figure 4.3a, where all variables in $var(\psi)$ are mapped to resp. indices as shown in Figure 4.2. Initially, we do not encounter any explicit reduction steps as it is not possible to apply forall-reduction to any clause in the matrix of ψ.

Hence, the first conflict encountered (Figure 3.3a) and the resulting derivation of the first learnt constraint (Figure 3.3b) is the first Q-resolution sequence to be traced. We record the intermediary resolvent $(y_1 \vee \neg y_3)$ (encoded in QDIMACS as $(1 - 4)$) with its antecedents (input clauses 3 and 5) and identify it by index 9. Next, we add the learnt constraint (y_1) (encoded as (1)) with index 10 and antecedents 2 and 9 to the trace, backtrack, continue, and encounter the second conflict (Figure 3.4a). We derive the second learnt constraint (Figure 3.4b) and record steps 11 and 12. Again, we backtrack, continue, encounter the third conflict (Figure 3.5a), record steps 13, 14, 15 and finally derive the empty clause (step 16). We conclude that formula ψ is unsat and add the respective result statement.

We extract the Q-resolution proof of unsatisfiability depicted in Figure 4.3c by reconstructing the Q-resolution sequence leading to the empty clause depicted in Figure 3.5b. That is, starting with the empty clause (step 16), we recursively follow the antecedents of each step encountered until an input clause is reached. All other steps (1, 5, 6, 7, 9, 10, 11, 12) are not part of the Q-resolution proof of unsatisfiability for Formula 3.3.

```
p cnf 9 8
e 1 2 0
a 3 0
e 4 5 6 0
a 7 0
e 8 9 0
-5 6 0
-3 4 0
-4 8 0
-8 9 0
1 7 -8 0
-1 3 -6 0
2 5 0
-7 -9 0
```

(a) Formula 3.3

```
p qrp 9 16
e 1 2 0
a 3 0
e 4 5 6 0
a 7 0
e 8 9 0
1 -5 6 0 0
2 -3 4 0 0
3 -4 8 0 0
4 -8 9 0 0
5 1 7 -8 0 0
6 -1 3 -6 0 0
7 2 5 0 0
8 -7 -9 0 0
9 1 4 0 3 5 0 0
10 1 0 2 9 0
11 -1 3 -5 0 1 6 0
12 -1 2 0 7 11 0
13 -7 -8 0 4 8 0
14 -7 -4 0 3 13 0
15 -3 -7 0 2 14 0
16 0 15 0
r unsat
```

(b) Trace of Example 3.3

```
p qrp 9 16
e 1 2 0
a 3 0
e 4 5 6 0
a 7 0
e 8 9 0
2 -3 4 0 0
3 -4 8 0 0
4 -8 9 0 0
8 -7 9 0 0
13 -7 8 0 4 8 0
14 -7 -4 0 3 13 0
15 -7 -3 0 2 14 0
16 0 15 0
r unsat
```

(c) Proof of Example 3.3

Figure 4.3: Formula 3.3 in QDIMACS format, the Q-resolution-based trace generated during the solving process in Example 3.3, and the resulting Q-resolution proof of unsatisfiability.

Chapter 5

Proof Checking with QRPcheck

QRPcheck is a tool for extracting and checking Q-resolution-based proofs of satisfiability and unsatisfiability in QRP format (as introduced in Chapter 4, Section 4.1). Further, given a proof of satisfiability, QRPcheck provides the possibility to check the validity of the set of initial cubes given. In the following, we introduce QRPcheck and proof checking as implemented in QRPcheck in more detail.

5.1 QRPcheck Overview

QRPcheck is a proof checker for Q-resolution-based traces and proofs of satisfiability and unsatisfiability in QRP format. Starting with the empty constraint, QRPcheck extracts the proof from a given trace, checks its correctness and provides the possibility to further check the validity of the set of initial cubes given in case of a proof of satisfiability. Note that currently, QRPcheck does not support the use of advanced dependency schemes as in [23]. In the following, we describe the general workflow and the main components of QRPcheck as illustrated in Figure 5.1.

QRPcheck expects a QRP trace (resp. proof) as input. For checking the validity of the set of initial cubes, it further requires the original formula in QDIMACS format. A trace (resp. proof) is represented as an array of linked constraint nodes with the empty constraint node as the root of the proof DAG given. Each constraint node maintains an internal id (independent of its given id) in order to prevent sparse arrays in case of step sequences with non-consecutive indices. A node is linked to its antecedents (if present) via their respective internal ids. Depending on whether and how often the solver had to backtrack during the search, traces (and in some cases proofs) may grow large in the number of steps given. In such cases, the overall number

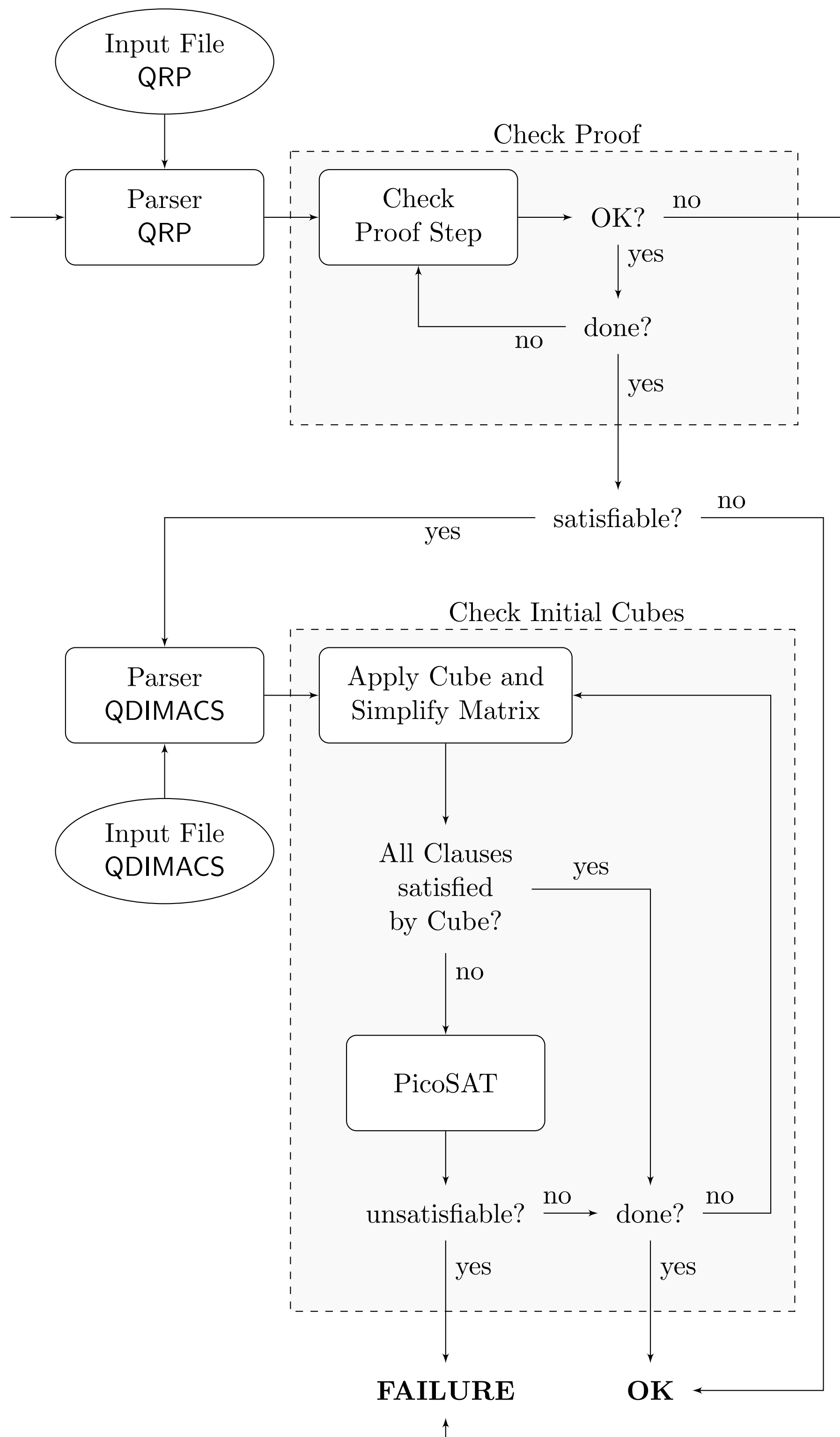


Figure 5.1: General workflow of QRPcheck.

of literals to be maintained may become enormous and we thus do not store any literals in-memory but read them on-demand via mapping the input QRP file to memory using virtual memory mechanisms.

As indicated in Figure 5.1, starting with the empty constraint the proof is extracted from a trace on-the-fly, while checking each proof step incrementally. If an incorrect proof step is encountered, we immediately abort the checking procedure and terminate with `FAILURE`. Else, if all steps proved to be correct and given proof is a proof of unsatisfiability, we are done (`OK`). If given proof is a proof of satisfiability, we further check if each of the initial cubes given indeed represents a valid satisfying assignment for the input formula. Hence, we apply each cube to the formula, simplify the matrix accordingly, and check if all clauses are satisfied under the assignment given. If this is not the case (which may happen if existential-reduction has been applied to the cube in question), we eliminate all universal variables from the simplified input formula and check the resulting propositional formula in CNF with the SAT-solver PicoSAT [6]. If the propositional formula is unsatisfiable, given initial cube is invalid and we terminate the checking procedure with `FAILURE`. Else, if all initial cubes proved to be valid, given proof is correct and we terminate with `OK`.

5.2 Checking Q-Resolution Proofs

QRPcheck expects a Q-resolution-based trace or proof in QRP format as input and checks each proof step incrementally, starting with the empty constraint. Apart from the empty constraint, a proof step may either represent an input clause (resp. initial cube), an explicit reduction step, or a resolution step (with or without reduction). We strictly require that neither resolvents nor their antecedents are tautological resp. contradictory (cf. Definition 2.13). Further, we do not allow that a proof step refers to itself as antecedent. Therefore, we immediately consider both cases as incorrect.

A top-level view of the proof checking algorithm implemented in QRPcheck is described in Figure 5.2. Starting with the empty constraint, function `check` traverses recursively in Depth-First-Search order over the proof DAG until either all steps proved to be correct or an incorrect proof step was found. In order to detect cycles resp. to prevent multiple checks on multi-referenced proof steps, we skip steps that have already been visited. For each (yet unvisited) step, we first check if a leaf of the proof DAG (function `is_initial`) has been reached. If the current step is not an input clause (resp. initial cube), we continue and check if the given constraint is a valid (explicit) reduction or resolution step (with or without reduction). We apply reduction and/or resolution to the step's antecedent(s) accordingly

(function `resolve_and_reduce`) and if the resulting constraint matches the given (function `check_constraint`), we continue to traverse the proof DAG. Otherwise, we abort the checking procedure and return with `ERROR`.

QRPcheck does not make any assumptions about how a solver handles forall- resp. existential-reduction and resolution. If executed by definition, a resolution step would strictly follow the order of application of (forall- resp. existential-) reduction and (clause resp. cube) resolution as given in Definition 2.13 and the resulting constraint of an explicit reduction step would not contain any literals to be further eliminated by (forall- resp. existential-) reduction. QRPcheck, however, does not require reduction and resolution steps to comply to the above but rather identifies all possible outcomes when reconstructing reduced constraints and resolvents.

In case of an explicit reduction step, determining which literals may be eliminated from its antecedent is solely based on the nesting level of its innermost existential (resp. universal) variable (cf. Definition 2.12) and thus, straightforward. In case of a resolution step, however, a literal might be eliminated when applying reduction to either the resolvent (cf. Definition 2.13, step 3), or one (or both) of its antecedents (cf. Definition 2.13, step 1), or both. Hence, QRPcheck checks for each literal l that may resp. must appear in the resulting constraint of a proof step if it might be eliminated by reducing its antecedent(s) or (in case of a resolution step) resolvent and marks variable `lit2var(`l`)` according to the marking scheme in Table 5.1.

Initially, for each proof step s, all variables are *unmarked*, i.e., marked as `NONE`. If a literal l *must* occur in the resulting constraint r of s, `lit2var(`l`)` is marked as positive (`POS`) or negative (`NEG`) occurrence, respectively. If a literal l may be eliminated and thus *may* or *may not* occur in r, `lit2var(`l`)` is treated as *don't care* and marked as `DCP` (positive occurrence), `DCN` (negative occurrence), or `DCPN` (either positive or negative occurrence), respectively. All variables that *must not* occur in r are marked as `NONE`. Note that in case of a resolution step, the pivot variable is treated as *must not occur* and therefore also marked as `NONE`.

Marking	**Description**
`NONE`	unmarked
`POS`	mandatory positive occurrence
`NEG`	mandatory negative occurrence
`DCP`	don't care, may be a positive occurrence
`DCN`	don't care, may be a negative occurrence
`DCPN`	don't care, may be either a positive or negative occurrence

Table 5.1: Marking scheme for variables to possibly occur in the resulting constraint of a resolution resp. reduction step as employed in QRPcheck.

```
1  function check (Step s)
2  {
3    if (is_visited (s) or is_initial (s))
4      return OK
5
6    m ← resolve_and_reduce (s)
7
8    if (m = INVALID)
9      return ERROR
10
11   if (check_constraint (s, m) = ERROR)
12     return ERROR
13
14   if (check (get_first_antecedent (s)) = ERROR)
15     return ERROR
16   else if (is_resolution_step (s))
17     return check (get_second_antecedent (s))
18
19   return OK
20 }
```

Figure 5.2: Top-level view of the proof checking algorithm in QRPcheck.

Example 5.1. As an example, consider a prefix $\exists\, x_1\, x_2\, x_3\ \forall\, y_1\, y_2\ \exists \ldots x_n$ and a clause resolution step with antecedents $(x_1 \vee x_3 \vee y_2)$ and $(\neg x_1 \vee \neg x_2 \vee y_1 \vee \neg y_2)$. Variable x_1 is the pivot variable and may not occur in the resulting constraint, hence it is marked as `NONE`. Literals x_3 (first antecedent) and $\neg x_2$ (second antecedent) may not be eliminated by forall-reduction and are marked as `POS` and `NEG`, respectively. However, the positive occurrence of y_1 (second antecedent) may be eliminated and is thus marked as `DCP`. Variable y_2 occurs in both antecedents as y_2 (first) and $\neg y_2$ (second), respectively. Hence, it must at least be eliminated in either one of them (else the resulting constraint would be tautological). Thus, y_2 may occur in the resulting constraint either positively, or negatively, or not at all, and is marked as `DCPN`. We conclude with a mapping $m = \{(x_1, \texttt{NONE}), (x_2, \texttt{NEG}), (x_3, \texttt{POS}), (y_1, \texttt{DCP}), (y_2, \texttt{DCPN})\}$ from variables to markings, which identifies constraints $(\neg x_2 \vee x_3)$, $(\neg x_2 \vee x_3 \vee y_1)$, $(\neg x_2 \vee x_3 \vee y_1 \vee y_2)$, $(\neg x_2 \vee x_3 \vee y_1 \vee \neg y_2)$, $(\neg x_2 \vee x_3 \vee y_2)$ and $(\neg x_2 \vee x_3 \vee \neg y_2)$ as valid possible resulting constraints.

Given a QRP proof for input formula ψ, QRPcheck determines all possible valid resulting constraints of a proof step `s` via a mapping $\texttt{m} : var(\psi) \longrightarrow \{\texttt{NONE}, \texttt{POS}, \texttt{NEG}, \texttt{DCP}, \texttt{DCN}, \texttt{DCPN}\}$ from variables to markings (cf. Example 5.1)

```
 1  function resolve_and_reduce (STEP s)
 2  {
 3    CONSTRAINT c ← union (get_antecedents (s))
 4    MARKING m ← init_marking (get_vars (c), NONE)
 5
 6    /* reduce antecedents and resolve */
 7    foreach (a in get_antecedents (s))
 8    {
 9      if (is_taut_contr (a))
10        return INVALID
11
12      foreach (l in get_literals (a))
13      {
14        if (is_complementary (l, m))
15        {
16          if is_dc (l, m))
17          {
18            if (is_reducible (l, a))
19              mark_dcpn (l, m)
20            else
21              mark (l, m)
22          }
23          else  /* marked as POS/NEG */
24          {
25            if (is_pivot (l, c))
26              mark_none (l, m)
27            else if (not is_reducible (l, a))
28              return INVALID
29          }
30        }
31        else if (!is_duplicate (l, m))
32        {
33          if (is_reducible (l, a)
34            mark_dc (l, m)
35          else
36            mark (l, m)
37        }
38      }
39    }
40
41    /* reduce resolvent */
42    if (is_resolution_step (s))
43      foreach (l in get_reducible_literals (c))
44        mark_dc (l, m)
45    return m
46  }
```

Figure 5.3: Apply resolution and/or reduction to a step's antecedents.

as described in Figure 5.3. Given a proof step `s`, function `resolve_and_reduce` initially generates a working constraint `c` by building the union of all antecedent constraints of `s`. Marking `m` is then initialized with the set of all variables in $var(\psi)$ that occur in constraint `c` (function `init_marking`), each of them initially marked as `NONE`. Note that we require each antecedent constraint of step `s` to be non-tautological resp. non-contradictory (function `is_taut_contr`), which we otherwise immediately treat as `INVALID`.

Both in case of a reduction and resolution step we first have to check, if (any of) its antecedent constraint(s) may be reduced by forall- (resp. existential-) reduction. Hence, for each literal `l` in antecedent constraint `a`, depending on whether any occurrence of variable `lit2var(l)` has already been encountered previously, we distinguish the following cases.

If `l` neither occurs complementary (function `is_complementary`) nor as non-reducible duplicate (function `is_duplicate`), we check if it may be eliminated by reduction (function `is_reducible`) and mark it as `DCP/DCN` (function `mark_dc`) or `POS/NEG` (function `mark`), respectively. Note that a non-reducible duplicate is a literal `l` that occurs non-complementary to its previous occurrence, where variable `lit2var(l)` is already identified as *must occur* (i.e., marked as `POS/NEG`). Further, note that in case of a reduction step, we should not encounter any other cases than the above (as we treat proofs with tautological clauses resp. contradictory cubes as incorrect).

If `l` is a complementary literal (function `is_complimentary`), given step is a resolution step and the current antecedent constraint is already the second. As variable `lit2var(l)` occurs complementary in both antecedent constraints, we distinguish two cases. If `l` may be eliminated from the first but not the second antecedent constraint, we conclude that it must occur in the resulting constraint and mark it accordingly (function `mark`). Otherwise, `lit2var(l)` may occur either positively, or negatively, or not at all, and we mark it as `DCPN`. However, if `l` may *not* be eliminated from either the first or the second antecedent constraint, it is either an occurrence of the pivot variable (function `is_pivot`) and marked as `NONE`, or given step is `INVALID` (as we obtain a tautogical resp. contradictory resulting constraint). Note that the latter case causes an immediate abort of the checking procedure (lines 8-9, Figure 5.2).

If both antecedents of a resolution step `s` have been reduced and resolved successfully, we further check if it is possible to apply reduction to the resulting resolvent. Finally, we return marking `m` as a representative of all possible valid resulting constraints of proof step `s`.

Given a proof step `s` and its marking `m`, we finally check if given resulting constraint `r` is valid w.r.t. to its antecedents as described in Figure 5.4. Function `check_constraint` first determines if the number of literals that *must* occur in `r` (`num_lits_min`) matches the number of literals in `r` that are *not* marked as *don't care* (`num_lits - num_lits_dc`). If not, we im-

```
1  function check_constraint (STEP s, MARKING m)
2  {
3    CONSTRAINT c ← union (get_antecedents (s))
4    CONSTRAINT r ← get_literals (s)
5
6    num_lits    ← size (r)
7    num_lits_dc ← size (get_dc (r, m))
8
9    num_lits_min ← size (c) - size (get_dc (c, m))
10
11   if (num_lits - num_lits_dc ≠ num_lits_min)
12     return ERROR
13
14   foreach (l in r)
15   {
16     m ← get_mark (l, m)
17
18     if (m = DCPN)
19       mark_dc (l, m)
20
21     if ((is_pos (l) and m ≠ POS and m ≠ DCP) or
22          (is_neg (l) and m ≠ NEG and m ≠ DCN)))
23       return ERROR
24   }
25
26   return OK
27 }
```

Figure 5.4: Check if resulting constraint `r` matches marking `m`.

mediately conclude that step `s` is invalid (`ERROR`). Otherwise, we continue to check if each literal in `r` matches its resp. marking in `m`. Note that in case that a variable is marked as `DCPN`, we explicitly check for tautologies (resp. contradictions) by resetting the marking to `DCP` resp. `DCN` (function `mark_dc`), as soon as the first occurrence of the variable is encountered.

Example 5.2. As an example for the application of proof checking as implemented in QRPcheck, consider the Q-resolution proof of unsatisfiability in QRP format as given in Figure 5.5. Starting with the empty constraint (proof step 12), we traverse the proof DAG (as given in QRP notation in Figure 5.6) in Depth-First order, mark each step and its antecedents as described in Figure 5.2, and check it as follows.

```
p qrp 6 12
e 1 2 3 0
a 4 5 0
e 6 0
1 -2 -3 -5 0 0
2 2 -5 6 0 0
3 -2 3 4 6 0 0
4 2 4 0 0
5 -1 5 0 0
6 1 -5 -6 0 0
7 -3 -5 6 0 1 2 0
8 3 4 6 0 3 4 0
9 -5 -6 0 5 6 0
10 4 -5 6 0 7 8 0
11 4 -5 0 9 10 0
12 0 11 0
r unsat
```

Figure 5.5: Q-resolution proof in QRP format as input for Example 5.2.

Proof step 12—the final step to derive the empty clause and thus the first step to be checked—is a reduction step with antecedent (`4 -5`). Hence, constraint `c`, which is the union of the antecedent constraints of step 12 (Figure 5.3, line 3), is defined as the set of literals $\{4, -5\}$. Both variables 4 and 5 are universal and may be eliminated by forall-reduction. Thus, 4 is marked as `DCP`, 5 as `DCN`, and function `resolve_and_reduce` yields the mapping $\mathtt{m} = \{(4, \mathtt{DCP}), (5, \mathtt{DCN})\}$. The constraint given by step 12 is the empty clause, which matches `m` (function `check_constraint`), and we continue with the antecedent of step 12.

Proof step 11, the antecedent of step 12, is a resolution step (with reduction) with antecedents 9 and 10. Hence, we first check if any literals of the constraints given by steps 9 and 10 may be eliminated by forall-reduction (which is not the case). Variable 6 is the pivot and thus marked as `NONE`. The resolvent (`4 -5`) can be further forall-reduced as both variables 4 and 5 are universal, which are therefore marked as `DCP` resp. `DCN`, accordingly. We obtain mapping $\{(4, \mathtt{DCP}), (5, \mathtt{DCN}), (6, \mathtt{NONE})\}$, conclude that it matches the constraint given by step 11 and continue with the antecedents of step 11.
Proof step 9, the first antecedent of step 11, is a resolution step (with reduction) with two input clauses as antecedents. This time, one of the antecedents (step 5) may be reduced by forall-reduction and the literal that may be eliminated is the occurrence of variable 5. Variable 5 occurs in both antecedents—positively in the first but negatively in the second—and may not be eliminated in step 6. Hence, we conclude that the occurrence of vari-

Step	**Antecedents** Step	Var.	Mark.	**c**	**Var.**	**Mark.**
12	11	4	DCP	$\{ 4, -5 \}$	4	DCP
		5	DCN		5	DCN
11	9	5	NEG	$\{ 4, -5, \underbrace{6, -6}_{pivot} \}$	4	DCP
		6	NEG		5	DCN
	10	4	POS		6	NONE
		5	NEG			
		6	POS			
9	5	1	NEG	$\{ \underbrace{1, -1}_{pivot}, 5, -5, -6 \}$	1	NONE
		5	DCP		5	NEG
	6	1	POS		6	NEG
		5	NEG			
		6	NEG			
5	-	-	-	$\{\}$	-	-
6	-	-	-	$\{\}$	-	-
10	7	3	NEG	$\{ \underbrace{3, -3}_{pivot}, 4, -5, 6 \}$	3	NONE
		5	NEG		4	POS
		6	POS		5	NEG
	8	3	POS		6	POS
		4	POS			
		6	POS			
7	1	2	NEG	$\{ \underbrace{2, -2}_{pivot}, -3, -5, 6 \}$	2	NONE
		3	NEG		3	NEG
		5	DCN		5	NEG
	2	2	POS		6	POS
		5	NEG			
		6	POS			
1	-	-	-	$\{\}$	-	-
2	-	-	-	$\{\}$	-	-
8	3	2	NEG	$\{ \underbrace{2, -2}_{pivot}, 3, 4, 6 \}$	2	NONE
		3	POS		3	POS
		4	POS		4	POS
		6	POS		6	POS
	4	2	POS			
		4	DCP			
3	-	-	-	$\{\}$	-	-
4	-	-	-	$\{\}$	-	-

Table 5.2: Steps (in traversal order), marking (antecedents), constraint `c` (union of a steps' antecedent constraints) and mapping `m`: $\{Var.\} \longrightarrow \{Mark.\}$ (as the result of function `resolve_and_reduce`) for Example 5.2.

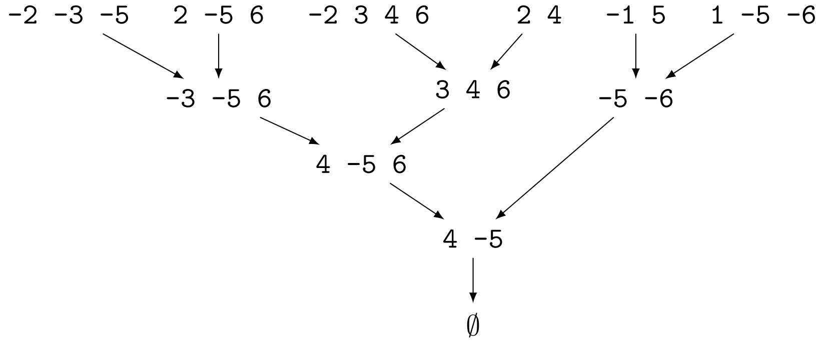


Figure 5.6: Proof DAG of the QRP proof in Figure 5.5 (Example 5.2).

able 5 *must* be eliminated from the first antecedent (else the resolvent would be tautological) and obtain the mapping $\{(1, \texttt{NONE}), (5, \texttt{NEG}), (6, \texttt{NEG})\}$. Both antecedents of step 9 are input clauses (and thus skipped) and we continue with the second antecedent of step 11, which is step 10.

Proof Step 10 again is a resolution step (without reduction) with antecedents 7 and 8, which in turn are both resolution steps with input clauses as antecedents. Both step 10 and its antecedent 7 and 8 prove to be correct and after finishing the traversal of the proof DAG with steps 3 and 4, which again are input clauses, we conclude that all proof steps given are valid and given Q-resolution proof of unsatisfiability is correct.

5.3 Checking Initial Cubes

Given a Q-resolution proof of (un)satisfiability, we interpret the set of input clauses (resp. initial cubes) as the set of leaf nodes of the corresponding proof DAG. In case of a proof of unsatisfiability, the set of input clauses is a subset of the matrix of the input formula and is hence regarded as given. In contrast, in case of a proof of satisfiability the set of initial cubes is a set of satisfying assignments and a subset of a QBF-model of the input formula. This set of satisfying assignments is determined by the QBF-solver during the solving process and hence not regarded as given and correct as-is. Given the input formula in QDIMACS format, QRPcheck provides the possibility to verify that the set of initial cubes given indeed represents a set of valid satisfying assignments by checking the validity of each initial cube as follows.

Definition 5.1. *Given a QBF ψ in PCNF and an initial cube $c = \{l_1, ..., l_n\}$, we interpret each literal $l_i \in c$ as an assignment $\alpha(v_i) = l_i$ with $v_i \in var(\psi)$ such that variable $v_i =$ `lit2var(`l_i`)` is assigned to true (resp. false) if literal l_i is a positive (resp. negative) occurrence of v_i. Initial cube c represents a valid satisfying assignment of ψ if ψ is true under assignment α.*

	`p cnf 10 12`	`p qrp 10 7`
	`a 7 8 0`	`a 7 8 0`
	`e 2 0`	`e 2 0`
	`a 1 3 4 6 9 0`	`a 1 3 4 6 9 0`
	`e 5 10 0`	`e 5 10 0`
1	`3 -8 -1 6 7 -9 -10 -2 0`	
2	`-4 3 1 -9 -8 -5 0`	
3	`9 -2 5 10 0`	
4	`-4 -8 6 -1 -3 -5 0`	
5	`8 2 0`	
6	`3 8 4 -1 -6 -9 -2 -10 0`	
7	`4 -6 -8 3 -9 -2 10 -5 0`	
8	`-10 -2 -5 0`	
9	`-2 5 0`	
10	`-6 -3 -10 0`	
11	`6 2 -10 0`	
12	`8 -1 -10 -2 5 0`	
13		`13 2 5 -8 -10 0 0`
14		`14 -8 2 0 13 0`
15		`15 -8 0 14 0`
16		
17		`17 -2 -5 8 0 0`
18		`18 8 -2 0 17 0`
19		`19 8 0 18 0`
20		`20 0 19 15 0`
		`r sat`

(a) Formula (b) Proof

Figure 5.7: Input formula for Example 5.3 in QDIMACS format (left) and corresponding proof of satisfiability in QRP format (right).

Given a QBF $\psi = Q.\phi$, in order to check the validity of an initial cube c we first apply c to ψ with respect to Definition 5.1 and check if all clauses in matrix ϕ are satisfied. Therefore, for each literal $l_i \in c$ we determine assignment $\alpha(\texttt{lit2var}(l_i))$ and simplify matrix ϕ by deleting all clauses (resp. literals) that are satisfied (resp. unsatisfied) under α. If the resulting matrix ϕ' is empty, formula ψ is *true* under α and cube c is a so-called *cover set* and a valid satisfying assignment of ψ. However, if this is not the case, cube c is not necessarily invalid but may have been reduced by existential-reduction prior to its recording (*non-covering set*). Hence, we have to determine, if ϕ' is satisfiable w.r.t. the remaining existential occurrences in ϕ'.

Lemma 5.1 (Trivial Truth of a QBF [11])**.** *Let $\psi = Q.\phi$ be a QBF in PCNF. Let $\phi = \{\phi_\exists \cup \phi_\forall \cup \phi_{\exists,\forall}\}$, such that $var(\phi_\exists) \subseteq var_\exists(\psi)$, $var(\phi_\forall) \subseteq var_\forall(\psi)$ and $var(\phi_{\exists,\forall}) \subseteq var(\psi)$, and let $\phi_\exists$, $\phi_\forall$, and $\phi_{\exists,\forall}$ be a set of clauses with only existential, only universal, and both existential and universal occurrences, respectively. Further, let $\phi'_\exists$ be the set of clauses obtained by deleting all universal occurrences from $\phi_{\exists,\forall}$. A QBF $\psi = Q.\{\phi_\exists \cup \phi_\forall \cup \phi_{\exists,\forall}\}$ is true if $\phi_\forall = \emptyset$ and $\phi' = \{\phi_\exists \cup \phi'_\exists\}$ is satisfiable.*

With respect to Lemma 5.1 we therefore eliminate all universal literals from ϕ' and check if the resulting propositional formula ϕ'' is satisfiable.

Example 5.3. As an example, consider the input formula given in QDIMACS format in Figure 5.7a. Given formula is satisfiable, the corresponding Q-resolution proof of satisfiability is given in QRP format in Figure 5.7b. We identify both $c_1 = \{\texttt{2}, \texttt{5}, \texttt{-8}, \texttt{-10}\}$ (step 13) and $c_2 = \{\texttt{-2}, \texttt{-5}, \texttt{8}\}$ (step 17) as initial cubes and start the checking procedure with initial cube c_1, which implies assignment $\alpha = \{(\texttt{2}, \top), (\texttt{5}, \top), (\texttt{8}, \bot), (\texttt{10}, \bot)\}$. We apply α to the matrix of the input formula, which results in the simplifications depicted in Table 5.3. All clauses are satisfied under α, hence we conclude that c_1 is valid (and a cover set) and continue with cube c_2.

Cube c_2 implies assignment $\alpha = \{(\texttt{2}, \bot), (\texttt{5}, \bot), (\texttt{8}, \top)\}$ and applying α to the matrix of the input formula yields the simplifications depicted in Table 5.4. This time, clauses 10 and 11 remain unsatisfied and $\phi' = \{\{\texttt{-6}, \texttt{-3}, \texttt{-10}\}, \{\texttt{6}, \texttt{-10}\}\}$. Cube c_2 therefore is a non-covering set and we eliminate all remaining occurrences of universal variables 3 and 6 and obtain the propositional formula $\phi'' = \{\{\texttt{-10}\}\}$, which is satisfiable. We conclude that c_2 is valid and are done.

Id	Clause	satisfied by
1	3 -8 -1 6 7 -9 -10 ~~-2~~	-8
2	-4 3 1 -9 -8 ~~-5~~	-8
3	9 ~~-2~~ 5 ~~10~~	5
4	-4 -8 6 -1 -3 ~~-5~~	-8
5	~~8~~ 2	2
6	3 ~~8~~ 4 -1 -6 -9 ~~-2~~ -10	-10
7	4 -6 -8 3 -9 ~~-2~~ ~~10~~ ~~-5~~	-8
8	-10 ~~-2~~ ~~-5~~	-10
9	~~-2~~ 5	5
10	-6 -3 -10	-10
11	6 2 -10	2
12	~~8~~ -1 -10 ~~-2~~ 5	5

Table 5.3: Matrix of the input formula after applying c_1 in Example 5.3.

Id	Clause	satisfied by
1	3 ~~-8~~ -1 6 7 -9 -10 -2	-2
2	-4 3 1 -9 ~~-8~~ -5	-5
3	9 -2 ~~5~~ 10	-2
4	-4 ~~-8~~ 6 -1 -3 -5	-5
5	8 ~~2~~	8
6	3 8 4 -1 -6 -9 -2 -10	-2
7	4 -6 ~~-8~~ 3 -9 -2 10 -5	-2
8	-10 -2 -5	-2
9	-2 ~~5~~	-2
10	-6 -3 -10	
11	6 ~~2~~ -10	
12	8 -1 -10 -2 ~~5~~	-2

Table 5.4: Matrix of the input formula after applying c_2 in Example 5.3.

```
1  function check_initial_cubes ()
2  {
3    foreach (c in get_initial_cubes ())
4    {
5      cnf ← simplify_matrix (c)
6
7      if (is_empty (cnf))
8        return OK
9
10     if (is_unsat (cnf))
11       return ERROR
12   }
13
14   return OK
15 }
```

Figure 5.8: Top-level view of initial cube checking in QRPcheck.

A top-level view of the initial cube checking algorithm implemented in QRPcheck is described in Figure 5.8. For each initial cube `c` given, we first apply `c` to the input formula (function `simplify_matrix`), simplify the matrix accordingly, and obtain a corresponding propositional formula `cnf` (cf. ϕ'' above). If `cnf` is empty, we immediately conclude that cube `c` indeed represents a valid satisfying assignment and continue.

Otherwise, we use the SAT-solver PicoSAT to check if formula `cnf` is satisfiable (function `is_unsat`). If this is not the case, we conclude that given cube is invalid and immediately abort the checking procedure (`ERROR`). Else, we conclude that initial cube `c` is valid and continue. If all initial cubes proved to be valid, we conclude that the given set of initial cubes indeed represents a set of satisfying assignments of the input formula and terminate with `OK`.

Given an initial cube `c`, we simplify the matrix of the input formula w.r.t. cube `c` as described in Figure 5.9. For each literal `l` in `c`, we determine variable `v = lit2var(l)` and apply assignment α(`v`) = `l` as follows. We retrieve a set of all clauses with occurrences of variable `v` (function `get_occurrence_list`) and delete all clauses satisfied and all literals unsatisfied by assignment α(`v`). If the resulting formula is empty (i.e., if all clauses are satisfied), given cube is a cover set and we return the empty matrix. Otherwise, cube `c` is a non-covering set and we delete all remaining universal occurrences (function `delete_universals`) and return the resulting propositional formula (function `get_matrix`).

```
1  function simplify_matrix (CUBE c)
2  {
3    foreach (l in get_literals (c))
4    {
5      v ← lit2var (l)
6
7      foreach (cl in get_occurrence_list (v))
8      {
9        if (is_satisfied (cl, l))
10         delete_clause (cl)
11       else
12         delete_occurrence (cl, v)
13     }
14   }
15
16   if (!all_clauses_satisfied ())
17     delete_universals ()
18
19   return get_matrix ()
20 }
```

Figure 5.9: Simplify the matrix of the input formula w.r.t. to initial cube `c`.

Chapter 6

Experimental Results

We applied tracing, proof extraction and proof checking as previously described in Chapters 3, 4, and 5 on the benchmark sets of the QBF competitions 2008 (QBFEVAL'08) and 2010 (QBFEVAL'10), which consist of 3326 and 568 formulas, respectively.[1] The proof checking workflow and test setup for our experimental evaluation is depicted in Figure 6.1. Given a QRP trace recorded by DepQBF, we checked the proof (and, in case of satisfiable instances, its set of initial cubes) with QRPcheck and further extracted its corresponding QRP representation to disk.

We considered all 1228 (QBFEVAL'08) resp. 362 (QBFEVAL'10) formulas solved by DepQBF within 900 seconds and did not enable the use of advanced dependency schemes. All experiments were performed on 2.83 GHz Intel Core 2 Quad machines with 8 GB of RAM, running Ubuntu 9.04. Time and memory limits for the whole proof checking workflow were set to 1800 seconds and 7 GB, respectively.

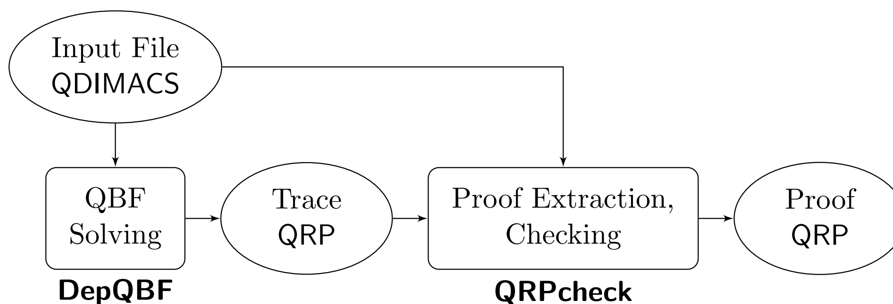


Figure 6.1: Proof checking workflow for experimental evaluation.

[1] Available at http://www.qbflib.org/index_eval.php

Family	Instances		Time DepQBF [s]			Time QRPcheck [s]		
	sv	ch	total	avg	med	total	avg	med
Abduction	284	283	577.3	2.0	0.1	237.5	0.8	0.0
Adder	5	5	1.3	0.3	0.0	0.8	0.2	0.0
blackbox-*-QBF	314	282	3918.3	13.9	0.1	1845.1	6.5	0.0
blackbox_design	1	1	0.4	0.4	0.4	0.5	0.5	0.5
Blocks	11	10	130.8	13.1	0.6	40.1	4.0	0.1
BMC	81	80	3280.9	41.0	0.4	169.7	2.1	0.0
Chain	10	8	481.0	60.1	16.3	429.3	53.7	14.6
circuits	5	4	3.0	0.7	0.6	0.4	0.1	0.1
conformant	11	9	207.7	23.1	4.1	28.8	3.2	0.4
Counter	10	10	131.0	13.1	0.0	8.9	0.9	0.0
Debug	1	1	379.4	379.4	379.4	6.2	6.2	6.2
DFlipFlop	10	10	1.9	0.2	0.1	0.3	0.0	0.0
evader-pursuer	16	14	330.1	23.6	0.2	26.6	1.9	0.1
FPGA_*_FAST	5	5	0.7	0.1	0.1	0.1	0.0	0.0
FPGA_*_SLOW	3	3	258.1	86.0	54.1	17.9	6.0	8.0
Impl	10	10	0.0	0.0	0.0	0.0	0.0	0.0
irqlkeapclte	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
MutexP	3	3	37.2	12.4	0.0	26.9	9.0	0.0
pan	162	157	10356.8	66.0	1.2	2583.1	16.5	0.3
Rintanen	2	2	39.0	19.5	19.5	16.7	8.4	8.4
Sakallah	1	1	0.1	0.1	0.1	0.0	0.0	0.0
Scholl-Becker	38	35	181.5	5.2	0.1	154.5	4.4	0.0
SortingNet	49	44	1690.2	38.4	2.5	369.1	8.4	0.7
SzymanskiP	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
tipdiam	78	76	1407.1	18.5	0.0	902.2	11.9	0.0
tipfixpoint	73	70	891.7	12.7	0.1	653.3	9.3	0.0
Toilet	8	7	67.9	9.7	0.7	32.3	4.6	0.1
Tree	12	11	150.3	13.7	0.3	181.6	16.5	0.2
uclid	0	0	0.0	0.0	0.0	0.0	0.0	0.0
VonNeumann	10	10	6.5	0.7	0.4	1.9	0.2	0.1
wmiforward	13	13	891.7	68.6	0.0	273.5	21.0	0.0
total	**1228**	**1166**	**25422.1**	**21.8**	**0.1**	**8007.3**	**6.9**	**0.0**

Table 6.1: QBFEVAL'08: family overview (runtime). Runtime considers checked instances only, all times are wall-clock times in seconds.

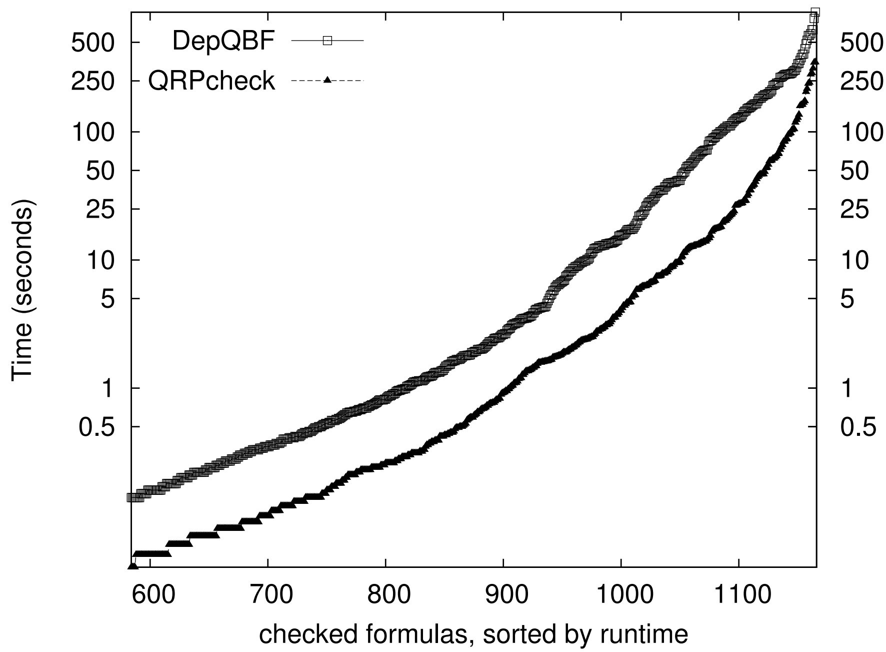
Family	**Instances**		**Time DepQBF [s]**			**Time QRPcheck [s]**		
	sv	ch	total	avg	med	total	avg	med
Abduction	48	48	47.8	1.0	0.0	6.2	0.1	0.0
Adder	0	0	0.0	0.0	0.0	0.0	0.0	0.0
blackbox-*-QBF	43	36	530.4	14.7	0.2	170.2	4.7	0.1
blackbox_design	0	0	0.0	0.0	0.0	0.0	0.0	0.0
Blocks	4	3	11.6	3.9	0.1	5.3	1.8	0.1
BMC	12	12	34.5	2.9	0.5	5.1	0.4	0.1
Chain	0	0	0.0	0.0	0.0	0.0	0.0	0.0
circuits	2	2	30.3	15.2	15.2	3.5	1.8	1.8
conformant	5	3	24.8	8.3	0.1	2.2	0.7	0.0
Connect4	8	8	40.8	5.1	0.1	4.0	0.5	0.0
Counter	2	2	227.7	113.8	113.8	13.5	6.7	6.7
Debug	0	0	0.0	0.0	0.0	0.0	0.0	0.0
evader-pursuer	10	9	719.9	80.0	0.7	76.3	8.5	0.1
FPGA_*_FAST	2	2	0.2	0.1	0.1	0.0	0.0	0.0
FPGA_*_SLOW	1	1	104.7	104.7	104.7	5.8	5.8	5.8
Impl	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
mqm	128	128	9727.2	76.0	24.2	3580.1	28.0	2.9
pan	24	24	3749.2	156.2	71.8	1295.7	54.0	13.2
Rintanen	1	1	12.7	12.7	12.7	1.5	1.5	1.5
Sakallah	0	0	0.0	0.0	0.0	0.0	0.0	0.0
Scholl-Becker	11	10	31.1	3.1	0.2	16.4	1.6	0.1
SortingNet	6	5	360.3	72.1	6.7	93.1	18.6	0.9
SzymanskiP	0	0	0.0	0.0	0.0	0.0	0.0	0.0
tipdiam	3	2	275.3	137.7	137.7	343.1	171.6	171.6
tipfixpoint	9	9	2.3	0.3	0.1	2.1	0.2	0.0
Toilet	40	40	108.4	2.7	0.0	57.4	1.4	0.0
VonNeumann	2	2	4.9	2.4	2.4	1.2	0.6	0.6
total	**362**	**348**	**16044.0**	**46.1**	**0.5**	**5682.8**	**16.3**	**0.2**

Table 6.2: QBFEVAL'10: family overview (runtime). Runtime considers checked instances only, all times are wall-clock times in seconds.

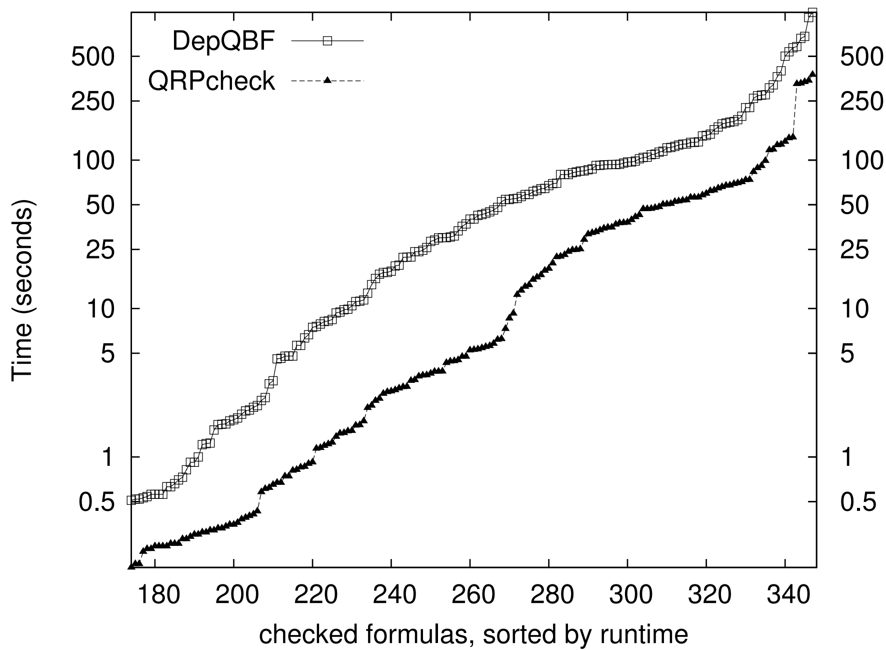
Tables 6.1 and 6.2 show the aggregated runtime results on the QBFEVAL'08 and QBFEVAL'10 benchmark sets, grouped by family. The first column ("Instances") indicates the number of instances solved by DepQBF per family ("sv") and the number of instances (out of the instances solved) that were checked by QRPcheck ("ch"), respectively. The second ("Time DepQBF") and third ("Time QRPcheck") columns state the total, average and median runtime (in seconds) for solving and checking all instances that were checked by QRPcheck within given time and memory constraints. Note that all times considered are wall-clock times. Further, both the solving and checking time are to be considered as the total runtime required by DepQBF (incl. tracing) and QRPcheck (incl. extracting the QRP proof to disk), respectively. Hence, in both cases, I/O operations consume a considerable amount of the resp. total runtime (cf. Tables 6.4, 6.5 and 6.6).

Within given time and memory constraints, out of 1228 (QBFEVAL'08) and 362 (QBFEVAL'10) solved by DepQBF we were able to check 1166 and 348, respectively. On average, on the benchmark set 2008 (2010) QRPcheck required a third of the runtime of DepQBF, with an average and median runtime of 6.9 (16.3) seconds and less than 0.1 (0.2) seconds, respectively. Further, on all families in both benchmark sets except for the "Tree" (2008) and the "tipdiam" (2010) families, which contain instances that consumed up to 65% of the total runtime when extracting the QRP proof representation to disk, QRPcheck required (considerably) less runtime than DepQBF. A more detailed runtime comparison of DepQBF and QRPcheck is given in Table 6.3 and Figure 6.2. Note that in Figure 6.2, checked instances that were solved in less than 0.2 seconds were not considered.

Table 6.4 gives an overview of the percentages of parsing (total for QRP and QDIMACS input) and checking (total for proof and initial cube checking) on both benchmark sets 2008 and 2010 with the total runtime of QRPcheck as a base. A more detailed runtime overview of QRPcheck is given in Table 6.5. On all checked instances of the benchmark set 2008 (2010), parsing the input QRP trace required 18% (15%) of the total runtime, with an average and median runtime of 1.2 (3.5) seconds and less than 0.1 seconds (on both sets), respectively. Proof checking (without initial cube checking) required 19% (26%) of the total runtime, with an average and median runtime of 1.3 (3.8) seconds and less than 0.1 seconds (on both sets), respectively. For satisfiable instances, on the benchmark set 2008 (2010) checking the set of initial cubes given required 20% (12%) of the total runtime, where 1704 (0) out of 18.3M (5.6M) initial cubes were non-covering sets. That is, for all instances of the benchmark set 2010, all initial cubes given were cover sets and PicoSAT was not required for initial cube checking on any instance. For all 44 instances of the benchmark set 2008 were the set of initial cubes contained non-covering sets, actually all initial cubes given were non-covering sets and initial cube checking required a total of less than 0.1 seconds.



(a) QBFEVAL'08



(b) QBFEVAL'10

Figure 6.2: Runtime comparison: DepQBF vs. QRPcheck. Considers checked instances that were solved within ≥ 0.2 seconds only, all times are wall-clock times in seconds.

		Instances		DepQBF			QRPcheck		
		sv	ch	total	avg	med	total	avg	med
2008	sat	494	476	13868.5	29.1	0.1	4293.8	9.0	0.0
	unsat	734	690	11553.6	16.7	0.1	3713.6	5.4	0.1
	total	**1228**	**1166**	**25422.1**	**21.8**	**0.1**	**8007.3**	**6.9**	**0.0**
2010	sat	157	153	11289.5	73.8	4.6	4737.8	31.0	1.6
	unsat	205	195	4754.5	24.4	0.3	945.1	4.8	0.1
	total	**362**	**348**	**16044.0**	**46.1**	**0.5**	**5682.8**	**16.3**	**0.2**

Table 6.3: Runtime: DepQBF vs. QRPcheck. Runtime consideres checked instances only, all times are wall-clock times in seconds.

		Time [s]	parse		check		other
2008	sat	4293.8	735.5	17.1 %	1483.5	34.5 %	48.3 %
	unsat	3713.6	843.1	22.7 %	889.8	24.0 %	53.3 %
	total	**8007.3**	**1578.6**	**19.7 %**	**2373.3**	**29.6 %**	**50.6 %**
2008	sat	4737.8	591.7	12.5 %	1684.0	35.5 %	52.0 %
	unsat	945.1	288.9	30.6 %	187.6	19.8 %	49.6 %
	total	**5682.8**	**880.6**	**15.5 %**	**1871.5**	**32.9 %**	**51.6 %**

Table 6.4: Runtime (percentage): QRPcheck total vs. parsing vs. checking. Column "other" roughly corresponds to I/O for proof extraction. Runtime considers checked instances only, all times are wall-clock times in seconds.

		Time [s]			parse QRP			parse QDIMACS		
		total	avg	med	total	avg	med	total	avg	med
2008	sat	4293.8	9.0	0.0	604.3	1.3	0.0	131.2	0.3	0.0
	unsat	3713.6	5.4	0.1	843.1	1.2	0.0	-	-	-
	total	**8007.3**	**6.9**	**0.0**	**1447.4**	**1.2**	**0.0**	**131.2**	**0.3**	**0.0**
2010	sat	4737.8	31.0	1.6	587.6	3.8	0.3	4.2	0.0	0.0
	unsat	945.1	4.8	0.1	288.9	1.5	0.0	-	-	-
	total	**5682.8**	**16.3**	**0.2**	**876.5**	**2.5**	**0.0**	**4.2**	**0.0**	**0.0**

		Time [s]			check proof			check initial		
		total	avg	med	total	avg	med	total	avg	med
2008	sat	4293.8	9.0	0.0	634.6	1.3	0.0	848.9	1.8	0.0
	unsat	3713.6	5.4	0.1	889.8	1.3	0.0	-	-	-
	total	**8007.3**	**6.9**	**0.0**	**1524.4**	**1.3**	**0.0**	**848.9**	**1.8**	**0.0**
2010	sat	4737.8	31.0	1.6	1119.9	7.3	0.0	564.1	3.7	0.0
	unsat	945.1	4.8	0.1	187.6	1.0	0.0	-	-	-
	total	**5682.8**	**16.3**	**0.2**	**1307.4**	**3.8**	**0.0**	**564.1**	**3.7**	**0.0**

Table 6.5: Runtime: QRPcheck total vs. parsing (QRP and QDIMACS) and QRPcheck total vs. checking (proof and initial cubes). Runtime considers checked instances only, all times are wall-clock times in seconds.

		sv	Time [s]				Tracing
			total	max	avg	med	
2008	sat	495	19686.9	868.0	39.8	0.1	no
	unsat	743	23245.7	878.7	31.3	0.1	
	total	**1238**	**42932.7**	**878.7**	**34.7**	**0.1**	
	sat	494	29979.1	1605.7	60.7	0.2	yes
	unsat	734	45769.0	1706.2	62.4	0.2	
	total	**1228**	**75748.1**	**1706.2**	**61.7**	**0.2**	
2010	sat	157	9364.7	824.5	59.6	2.8	no
	unsat	205	5970.7	531.4	29.1	0.2	
	total	**362**	**15335.4**	**824.5**	**42.4**	**0.4**	
	sat	157	14899.9	1095.9	94.9	4.8	yes
	unsat	205	11686.5	884.7	57.0	0.4	
	total	**362**	**26586.5**	**1095.9**	**73.4**	**0.6**	

Table 6.6: Runtime: DepQBF with tracing vs. without tracing. Runtime considers solved instances only, all times are wall-clock times in seconds.

Considering the overall percentages of parsing and checking w.r.t. the total runtime of QRPcheck as shown in Table 6.4, parsing required 19.7% (15.5%), whereas checking required 29.6% (32.9%) of the total runtime on the benchmark set 2008 (2010). The remaining 50.6% (51.6%) indicated in column “other” roughly corresponds to the runtime required for I/O operations when extracting the QRP proof representation to disk.

Table 6.6 gives a comparison of the runtime of DepQBF on both benchmark sets with and without tracing enabled. Considering all instances of the benchmark set 2008 (2010) that were solved by DepQBF within 900 seconds, with tracing enabled, DepQBF required 1.8 (1.7) times the runtime of DepQBF without tracing enabled. This, again, is due to I/O operations as the trace is recorded to disk on-the-fly. Hence, due to given time constraints DepQBF “looses” 10 (1) instances on the benchmark set 2008 (2010) if tracing is enabled.

An overview of the minimum, maximum, average and median file size of QRP traces and their corresponding QRP proofs is given in Tables 6.7 and 6.8. Table 6.7 considers file size in number of Bytes and compares trace sizes of all instances solved by DepQBF within 900 seconds with trace and corresponding proof sizes of all instances that were checked by QRPcheck within given time and memory constraints. Table 6.8 considers checked instances only and compares trace and proof sizes in the total number of steps resp. literals given. Considering all instances solved by DepQBF, on the benchmark set 2008 (2010) QRP trace file sizes in Byte range from 104B (477B) to 55.3 GB (26.7 GB), with an average and median size of 1.2 GB (on both sets) and 2.3 MB (11.2 MB), respectively. However, considering checked instances only, due to given memory constraints (7 GB) the maximum QRP trace size QRPcheck was able to check was 6.7 GB (6.4 GB) with a corresponding QRP

			File Size			
			min	max	avg	med
2008	Trace*	sat	104 B	38.2 GB	823.1 MB	1.6 MB
		unsat	866 B	55.3 GB	1.4 GB	3.0 MB
		total	**104 B**	**55.3 GB**	**1.2 GB**	**2.3 MB**
	Trace	sat	104 B	6.7 GB	295.2 MB	1.2 MB
		unsat	866 B	6.0 GB	285.9 MB	2.1 MB
		total	**104 B**	**6.7 GB**	**289.7 MB**	**1.8 MB**
	Proof	sat	52 B	5.2 GB	145.1 MB	154.4 kB
		unsat	598 B	5.5 GB	89.8 MB	351.5 kB
		total	**52 B**	**5.5 GB**	**112.4 MB**	**272.5 kB**
2010	Trace*	sat	477 B	23.8 GB	1.3 GB	78.7 MB
		unsat	670 B	26.7 GB	1.1 GB	9.2 MB
		total	**477 B**	**26.7 GB**	**1.2 GB**	**11.2 MB**
	Trace	sat	477 B	6.3 GB	904.5 MB	75.6 MB
		unsat	670 B	6.2 GB	327.4 MB	6.2 MB
		total	**477 B**	**6.3 GB**	**581.1 MB**	**9.1 MB**
	Proof	sat	173 B	5.0 GB	518.4 MB	2.8 MB
		unsat	274 B	5.8 GB	66.7 MB	729.9 kB
		total	**173 B**	**5.8 GB**	**265.3 MB**	**1.0 MB**

Table 6.7: File size (Byte): QRP trace vs. QRP proof. "Trace*" considers all instances solved by DepQBF, "Trace" and "Proof" considers checked instances only.

			Steps				Literals			
			min	max	avg	med	min	max	avg	med
2008	Trace	sat	7	13M	324k	8k	12	1396M	66M	242k
		unsat	46	18M	380k	14k	131	1295M	60M	379k
		total	**7**	**18M**	**357k**	**11k**	**12**	**1396M**	**62M**	**334k**
	Proof	sat	2	9M	127k	119	2	1365M	33M	32k
		unsat	3	18M	155k	1k	3	1212M	19M	58k
		total	**2**	**18M**	**144k**	**747**	**2**	**1365M**	**25M**	**43k**
2010	Trace	sat	29	7M	785k	96k	78	1501M	200M	17M
		unsat	43	11M	376k	25k	88	1358M	70M	1M
		total	**29**	**11M**	**556k**	**32k**	**78**	**1501M**	**127M**	**2M**
	Proof	sat	3	3M	308k	1k	3	1265M	117M	626k
		unsat	3	8M	135k	2k	3	1262M	14M	146k
		total	**3**	**8M**	**211k**	**2k**	**3**	**1265M**	**60M**	**175k**

Table 6.8: File size (Steps, Literals): QRP trace vs. QRP proof. File size considers checked instances only.

proof size of 483.8 MB (4.1 GB). On all checked instances of the benchmark set 2008 (2010), QRP proof file sizes in Byte range from 52B (173B) to 5.5 GB (5.8 GB), with an average and median size of 112.4 MB (265.3 MB) and 272.5 kB (1.0 MB), respectively. Note that maximum numbers of literals of 1396M (1501M) and 1365M (1265M) for traces resp. proofs were the main reason for not storing literals in memory but reading them on-demand via mapping the input QRP trace to memory. Further, note that both in Byte file size and in the number of steps (resp. literals) trace sizes are multiples of the size of their corresponding proofs, in general.

On all 62 (QBFEVAL'08) and 14 (QBFEVAL'10) instances that were not checked by QRPcheck within given time and memory constraints (1800 seconds, 7 GB), QRPcheck ran out of memory due to input traces with an average (maximum) file size of 17 GB (52 GB) and 16 GB (27 GB), respectively. For the 14 instances from the benchmark set 2010 that QRPcheck was not able to check due to the 7 GB memory limit, we lifted the memory limit and rerun the experiments on a 2.4 GHz Intel Xeon hexa-core machine with 96 GB RAM, running Ubuntu 11.10. As a consequence, we were able to check all 14 instances successfully. An overview of the resulting runtime (DepQBF and QRPcheck) and trace resp. proof sizes is given in Tables 6.9 and 6.10. Trace file sizes ranged from 7.4 GB to 26.7 GB with corresponding proof file sizes of 0.4 GB and 0.2 GB, respectively. The average (median) proof size was 3.3 GB (291.0 MB), with a maximum file size of 14.6 GB. The average (median) runtime of QRPcheck was 207.1 (94.5) seconds, out of which parsing the input (QRP and QDIMACS) and (proof and initial cube) checking required 64.3 (58.8) seconds and 50.0 (4.1) seconds, respectively. Further, for all 14 instances, QRPcheck required an average (median) of 16.1 GB (15.3 GB) of memory, with a maximum memory usage of 27.1 GB (which correlates with an input trace file size of 26.7 GB).

Note that all instances from both benchmark sets that were checked by QRPcheck within given time and memory constraints also proved to have been solved correctly by DepQBF.

Formula	**Trace**		**Proof**	
	File Size	Steps	File Size	Steps
BLOCKS4ii.7.2-shuffled	11.4 GB	38.0M	1.5 GB	5.2M
C432.blif_0.10_1.00_0_0_out_*-sh*	11.0 GB	37.5M	8.7 GB	29.7M
biu.*-b003-*-I*01-*.c*f04.*-shuffled	16.8 GB	9.7M	14.2 GB	8.0M
biu.*-b003-*-I*01-*.c*f06.*-shuffled	17.7 GB	2.0M	19.0 MB	37.4k
biu.*-b003-*-I*03-*.c*f02.*-shuffled	25.3 GB	14.4M	14.6 GB	8.4M
biu.*-b003-*-I*03-*.c*f05.*-005-sh*	12.3 GB	0.6M	33.7 MB	36.6k
biu.*-b003-*-I*03-*.c*f06.*-shuffled	26.7 GB	10.0M	0.2 GB	0.1M
biu.*-b003-*-M*04-*.c*f05.*-008-sh*	16.6 GB	5.1M	5.4 GB	1.8M
biu.*-b003-*-M*04-*.c*f06.*-002-sh*	12.2 GB	3.9M	12.7 kB	91
blocks_enc_2_b3_ser–opt-9_-shuffled	17.1 GB	50.9M	46.9 kB	38
emptyroom_e3_ser–opt-20_-shuffled	10.7 GB	51.9M	0.4 MB	0.2k
ev-pr-8x8-11-7-0-1-2-lg-shuffled	7.4 GB	1.0M	0.4 GB	0.4M
sortnetsort10.v.stepl.012-shuffled	23.8 GB	16.0M	45.4 MB	6.1k
vis.prodcell^01.E-d4-shuffled	13.8 GB	1.8M	1.0 GB	98.3k
max	26.7 GB	51.9M	14.6 GB	29.7M
avg	15.9 GB	17.3M	3.3 GB	3.8M
med	15.2 GB	9.8M	291.0 MB	102.9k
total	**222.9 GB**	**242.8M**	**46.1 GB**	**53.7M**

Table 6.9: Trace and proof size for the 14 instances from the QBFEVAL'10 benchmark set that were further investigated with extended time (3600s) and memory (96 GB) constraints.

Formula	**DepQBF**	**QRPcheck**		
		total	parse	check
BLOCKS4ii.7.2-shuffled	547.3	116.2	49.7	21.6
C432.blif_0.10_1.00_0_0_out_*-sh*	528.0	471.6	57.3	132.5
biu.*-b003-*-I*01-*.c*f04.*-shuffled	449.7	665.6	61.0	217.8
biu.*-b003-*-I*01-*.c*f06.*-shuffled	854.8	75.4	74.7	0.3
biu.*-b003-*-I*03-*.c*f02.*-shuffled	667.1	699.3	89.9	214.1
biu.*-b003-*-I*03-*.c*f05.*-005-sh*	639.4	56.6	55.3	0.5
biu.*-b003-*-I*03-*.c*f06.*-shuffled	700.1	97.2	90.2	2.6
biu.*-b003-*-M*04-*.c*f05.*-008-sh*	440.7	281.3	57.2	76.9
biu.*-b003-*-M*04-*.c*f06.*-002-sh*	611.9	53.4	53.3	0.0
blocks_enc_2_b3_ser–opt-9_-shuffled	981.6	85.4	85.2	0.0
emptyroom_e3_ser–opt-20_-shuffled	624.2	47.4	47.2	0.3
ev-pr-8x8-11-7-0-1-2-lg-shuffled	594.5	44.4	29.5	5.5
sortnetsort10.v.stepl.012-shuffled	935.3	91.8	89.6	1.0
vis.prodcell^01.E-d4-shuffled	833.2	113.9	60.3	26.7
max	981.6	699.3	90.2	217.8
avg	672.0	207.1	64.3	50.0
med	631.8	94.5	58.8	4.1
total	**9407.8**	**2899.5**	**900.4**	**699.8**

Table 6.10: Runtime (in seconds) of DepQBF and QRPcheck for the 14 instances from the QBFEVAL'10 benchmark set that were further investigated with extended time (3600s) and memory (96 GB) constraints.

Chapter 7

Conclusion

In this thesis, we presented the extraction and validation of Q-resolution proofs of (un)satisfiability as part of the certification workflow for the state-of-the-art QBF-solver DepQBF.

We introduced QRP format, a novel text-based and explicit representation for Q-resolution-based traces and proofs. Given a trace in QRP format, QRP proofs can be extracted and checked in deterministic log space, a desirable property of proof formats suggested in [34]. Checking QRP proofs further does not show exponential worst-case behaviour, as it is not necessary to reconstruct resolvents from unordered lists of antecedents as in [38, 35].

We extended DepQBF to record traces in QRP format and extraced and validated the corresponding Q-resolution proofs of (un)satisfiability with our proof checker QRPcheck. The QRP representation of these proofs serves as a base for extracting Skolem/Herbrand function-based certificates of (un)satisfiability as described in [26].

We applied tracing, proof extraction, and proof checking on the benchmark sets of the QBF competitions 2008 and 2010 and presented an extensive evaluation of the results. It shows that within given time and memory constraints of 1800 seconds and 7 GB, QRPcheck was able to validate over 95% of all solved instances. After lifting the memory limit of 7 GB, all instances solved by DepQBF within 900 seconds were validated successfully. Further, all solved instances of both benchmark sets that were successfully validated by QRPcheck proved to have been solved correctly by DepQBF.

Overall, even though extracting the QRP proof representation to disk required a considerable amount of the total runtime of QRPcheck, proof checking with QRPcheck proved to be time efficient. Considering that QRP proof extraction is optional and further not necessary if given QRP input is a proof rather than a trace, the runtime overhead for I/O operations may be disregarded. However, the runtime overhead for recording a trace in QRP format to disk is to be considered significant—especially since traces may grow up to several GB in file size very quickly. Further, even though huge

trace files do not produce any overhead in time for the actual proof checking in QRPcheck, the input trace file size does affect its overall memory usage. Hence, a compact binary representation of the QRP format would be desirable. However, to further enhance the overall performance of the proof checking workflow, in-memory proof maintenance rather than the current tracing approach would be a desirable future property to be integrated in DepQBF. Promising possible in-memory approaches have already been proposed for SAT solving in [1].

Note that currently, QRPcheck does not support one of the key features of DepQBF—the use of advanced dependency schemes. Even though proof checking might become more complex if advanced dependency schemes are supported, this is a desirable property of QRPcheck to be considered as future work.

QRPcheck and the extended version of DepQBF are available at `http://www.fmv.jku.at/cdepqbf/`.

Bibliography

[1] Achá, R.J.A., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Practical Algorithms for Unsatisfiability Proof and Core Generation in SAT Solvers. AI Communications (AICOM) 23(2-3), 145–157 (2010)

[2] Audemard, G., Sais, L.: A Symbolic Search Based Approach for Quantified Boolean Formulas. In: Proceedings 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)

[3] Balabanov, V., Jiang, J.H.R.: Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011). Lecture Notes in Computer Science, vol. 6806, pp. 149–164. Springer (2011)

[4] Benedetti, M.: sKizzo: A Suite to Evaluate and Certify QBFs. In: Proceedings of the 20th International Conference on Automated Deduction (CADE-20). Lecture Notes in Computer Science, vol. 3632, pp. 369–376. Springer (2005)

[5] Benedetti, M., Mangassarian, H.: QBF-Based Formal Verification: Experience and Perspectives. Journal on Satisfiability (JSAT) 5(1-4), 133–191 (2008)

[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: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99). Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer (1999)

[8] 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)

[9] Büning, H.K., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Information and Computation 117(1), 12–18 (1995)

[10] Büning, H.K., Zhao, X.: On Models for Quantified Boolean Formulas. In: Logic versus Approximation. Lecture Notes in Computer Science, vol. 3075, pp. 18–32. Springer (2004)

[11] Cadoli, M., Giovanardi, A., Schaerf, M.: An Algorithm to Evaluate Quantified Boolean Formulae. In: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98) and 10th Innovative Applications of Artificial Intelligence Conference (IAAI'98). pp. 262–267. AAAI Press / The MIT Press (1998)

[12] Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. Journal of Automated Reasoning (JAR) 28(2), 101–142 (2002)

[13] Davis, M., Logemann, G., Loveland, D.W.: A Machine Program for Theorem-Proving. Communications of the ACM 5(7), 394–397 (1962)

[14] Dershowitz, N., Hanna, Z., Katz, J.: Bounded Model Checking with QBF. In: Proceedings 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] Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In: Proceedings of the 7th National Conference on Artificial Intelligence (AAAI'00). pp. 417–422. AAAI Press / The MIT Press (2000)

[16] Fitting, M.: First-order Logic and Automated Theorem Proving (2nd ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA (1996)

[17] Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In: Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001). Lecture Notes in Computer Science, vol. 2083, pp. 364–369. Springer (2001)

[18] 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)

[19] 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)

[20] Goultiaeva, A., Gelder, A.V., Bacchus, F.: A Uniform Approach for Generating Proofs and Strategies for Both True and False QBF Formulas. In: Proceedings of the 22nd International Joing Conference on Artificial Intelligence (IJCAI 2011). pp. 546–553. IJCAI/AAAI (2011)

[21] Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A First Step Towards a Unified Proof Checker for QBF. In: Proceedings 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)

[22] Kröning, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Berlin, Heidelberg (2008)

[23] Lonsing, F., Biere, A.: DepQBF: A Dependency-Aware QBF Solver. Journal on Satisfiability (JSAT) 7(2-3), 71–76 (2010)

[24] Lonsing, F., Biere, A.: Integrating Dependency Schemes in Search-Based QBF Solvers. In: Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010). Lecture Notes in Computer Science, vol. 6175, pp. 158–171. Springer (2010)

[25] 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)

[26] Preiner, M.: Extracting and Validating Skolem/Herbrand Function-Based QBF Certificates. Master's thesis, Johannes Kepler University, Linz (2012)

[27] Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. Journal of Artificial Intelligence Research (JAIR) 10, 323–352 (1999)

[28] Rintanen, J.: Asymptotically Optimal Encodings of Conformant Planning in QBF. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI'07). pp. 1045–1050. AAAI Press (2007)

[29] Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1), 23–41 (1965)

[30] Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP 2006). Lecture Notes in Computer Science, vol. 4204, pp. 514–529. Springer (2006)

[31] Silva, J.P.M., Lynce, I., Malik, S.: Conflict-Driven Clause Learning SAT Solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009)

[32] Staber, S., Bloem, R.: Fault Localization and Correction with QBF. In: Proceedings 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)

[33] Stockmeyer, L.J., Meyer, A.R.: Word Problems Requiring Exponential Time: Preliminary Report. In: Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73). pp. 1–9. ACM (1973)

[34] Van Gelder, A.: Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In: Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007). Lecture Notes in Computer Science, vol. 4501, pp. 328–333. Springer (2007)

[35] Yu, Y., Malik, S.: Validating the Result of a Quantified Boolean Formula (QBF) Solver: Theory and Practice. In: Proceedings of the 2005 Conference on Asia South Pacific Design Automation (ASP-DAC 2005). pp. 1047–1051. ACM Press (2005)

[36] Zhang, L., Malik, S.: Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2002). pp. 442–449. ACM (2002)

[37] Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002). Lecture Notes in Computer Science, vol. 2470, pp. 200–215. Springer (2002)

[38] Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: Proceedings of the 2003 Conference and Exposition on Design, Automation and Test in Europe (DATE 2003). pp. 10880–10885. IEEE Computer Society (2003)