

Certifying Unsatisfiability in an Expansion-Based DQBF Solver

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Manuel Breitenbrunner, BSc

Matrikelnummer 01625734


an der Fakultät für Informatik


der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.rer.nat. Stefan Szeider

Mitwirkung: Dipl.-Ing. Dr. techn. Friedrich Slivovsky

Wien, 14. Jänner 2023


Manuel Breitenbrunner


Stefan Szeider

Certifying Unsatisfiability in an Expansion-Based DQBF Solver

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Manuel Breitenbrunner, BSc

Registration Number 01625734


to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.rer.nat. Stefan Szeider

Assistance: Dipl.-Ing. Dr. techn. Friedrich Slivovsky

Vienna, 14th January, 2023


Manuel Breitenbrunner

Stefan Szeider

Erklärung zur Verfassung der Arbeit

Manuel Breitenbrunner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. Jänner 2023



Manuel Breitenbrunner

Kurzfassung

Mithilfe von quantifizierten booleschen Formeln (QBFs) können schwierige Probleme aus Bereichen wie der formalen Verifikation und Planung in einer kurzen und prägnanten Weise dargestellt und beschrieben werden. Jedoch ist das Erfüllbarkeitsproblem von quantifizierten booleschen Formeln (QSAT) ein PSPACE-vollständiges und somit ein schwieriges Problem. Eine weitere Generalisierung von QBF ist Dependency QBF. DQBFs erlauben für jede existenzielle Variable eine explizite Spezifikation ihrer Abhängigkeiten von universellen Variablen. Dadurch können Probleme mit DQBFs in einer noch prägnanteren Form dargestellt werden. Diese Ausdrucksstärke zieht jedoch ein höheres Komplexitätslevel mit sich. Das Erfüllbarkeitsproblem für DQBFs (DQSAT) ist ein NEXPTIME-vollständiges Problem und damit noch schwieriger zu lösen als QSAT. Durch die Komplexität dieser Probleme ist die korrekte Implementierung bzw. die Korrektheit der Ergebnisse von Solvern nicht automatisch garantiert. Um Gewissheit in die Ergebnisse von Solvern zu bekommen, generieren diese daher zusätzlich Zertifikate. Ziel der vorliegenden Arbeit ist es, die Korrektheit der Ergebnisse eines expansionsbasierten Solvers für DQBF zu verifizieren. Wir implementieren ein Tracing-Modul in PEDANT, welches die Ausgabe von QRAT Beweisen für unerfüllbare QBFs ermöglicht. Dadurch kann die Unerfüllbarkeit von QBF Instanzen von QRAT-TRIM verifiziert werden. In unseren Experimenten verwenden wir unerfüllbare Instanzen aus den PCNF Tracks von QBFEVAL'19 und QBFEVAL'20. QRAT-TRIM kann unter Zuhilfenahme der von PEDANT erstellten Zertifikate die Unerfüllbarkeit dieser QBFs bestätigen. Das Tracing beeinflusst die Performance von PEDANT nicht nennenswert.

Abstract

Quantified Boolean Formulas (QBFs) can be used to represent problems in areas like formal verification and planning in a concise way. However, the satisfiability problem of quantified Boolean Formulas (QSAT) is PSPACE-complete and thus believed to be intractable in general. A further generalization of QBF is DQBF. In a dependency QBF each existential variable has an explicitly stated dependency set, which is a subset of the universal variables. The satisfiability problem of DQBFs is NEXPTIME-complete and thus believed to be even harder than QSAT. Because of the complexity of these problems the correct implementation of solvers, respectively the correctness of their output, is not easily guaranteed. In order to create certainty in solvers' results, they need to generate certificates which can be used to verify the correctness of their results. The goal of the present work is to verify the correctness of the results of an expansion-based solver for DQBF. We implement a tracing module in PEDANT which allows the generation of QRAT proofs for unsatisfiable QBFs. This allows the verification of unsatisfiable QBF instances using QRAT-TRIM. We perform experiments with unsatisfiable instances from the PCNF tracks of QBFEVAL'19 and QBFEVAL'20. Using the certificates created by PEDANT, QRAT-TRIM can indeed confirm the unsatisfiability of these QBFs. Furthermore, tracing does not deteriorate the performance of PEDANT.

Contents

Kurzfassung	vii
Abstract	ix
Contents	xi
1 Introduction	1
2 Preliminaries	5
2.1 Propositional Formulas	5
2.2 Quantified Boolean Formulas	7
2.3 Dependency QBF	8
2.4 QSAT	9
2.5 Q-Resolution	12
2.6 Expansion-based Proof System	14
3 Introduction of QRAT	19
3.1 Proof-Formats	20
3.2 DRAT	20
3.3 QRAT	23
4 Pedant - An Expansion-Based DQBF Solver	27
4.1 Decision Procedure	27
4.2 Example	30
5 QRAT Certificates from Expansion Proofs	33
5.1 Pre-Processing	34
5.2 Clause Definitions	35
5.3 Post-Processing	39
6 Implementation Details	43
6.1 File - Handling	44
6.2 Challenges	45
6.3 Future Improvements	47
	xi

7 Experimental Evaluation	49
8 Conclusion & Future Work	59
List of Figures	61
Bibliography	63

Introduction

Over the last few decades propositional satisfiability (SAT) solving has gained popularity resulting in steady progress [HJS19a, HJS19b, BEH⁺20]. Conflict Driven Clause Learning (CDCL) represents the state of the art algorithm in SAT solving [MSJPM09], which allows modern solvers to handle input formulas with thousands of variables and millions of clauses [MZ09]. Although the performance of modern solvers is remarkable, it does not automatically guarantee the correctness of each concrete solver implementation. Often SAT solvers implement non-trivial proof techniques which are hard to realize. Minor errors can lead to unintentional behaviour of the designed algorithm which subsequently might decide the satisfiability of an input formula incorrectly. For complex enough formulas it is not possible to verify the result of a SAT solver afterwards, given just the binary output (TRUE / FALSE). In order to prove the correctness of a result, additional information in form of certificates is needed, which provide enough evidence for proving the satisfiability of a formula. Given a satisfying assignment, one can easily certify a satisfiable propositional formula. In contrast, proving the unsatisfiability of a propositional formula is much harder. In these cases, the certificate needs to provide enough evidence that there can not be an assignment of variables, such that the propositional formula is satisfied. As an example, this can be done using a resolution proof. The SAT community established DRAT [HHW13b] as a unified proof system allowing the certification of both satisfiable and unsatisfiable formulas.

Although certificates allow a retrospective verification of the satisfiability of the corresponding formulas using an external checker, it is not automatically guaranteed that the checker is implemented correctly. However, typically it is easier to implement a checker than implementing a solver. The de-facto standard checker utilized in the SAT community is DRAT-TRIM [WHH14] allowing the verification of DRAT proofs. DRAT-TRIM is well tested and due its heavy usage in the community has gained a certain level of trust. Furthermore there are checkers which are formally verified, such that their correctness is guaranteed. For instance, GRAT [Lam17] or LRAT [CFHH⁺17] provide a formally

verified checkers. Both approaches build upon the DRAT proof format and extend it by adding additional information in order to simplify the validation algorithm. Although DRAT-TRIM is highly optimized, the verification of DRAT proofs is quite expensive. The simplification of the validation algorithm not only allows a faster proof checking but also facilitates the formal verification of it.

The interest in SAT solving is partly due to the NP-completeness of the problem [Coo71]. As a result each problem in the complexity class NP can be transformed or reduced to the SAT problem in polynomial time, such that an efficient algorithm for SAT also solves all problems in NP efficiently. The complexity class NP consists of decision problems which are decidable in polynomial time by a non-deterministic Turing machine. The research in NP-complete problems also connect to the very famous problem whether $P = NP$. The discovery of a polynomial time algorithm for an NP-complete problem would prove this major unsolved problem. Although the general opinion tends towards NP being a proper superset of P, this still remains open.

Regardless of the success and the effectiveness of SAT solvers, there are problems which do not have a short encoding in propositional logic. This especially applies to problems beyond NP. For these problems a more sophisticated encoding method like QBF and DQBF can dramatically decrease the size of the encoding. The generalization of propositional logic called Quantified Boolean Formulas (QBF) allows the explicit quantification over truth values. It introduces the universal (\forall) as well as the existential (\exists) quantification of propositional variables. The usage of Henkin quantifiers [BCJ14] on propositional variables further generalizes QBFs and results in Dependency Quantified Boolean Formulas (DQBF). In contrast to propositional formulas, QBFs and DQBFs allow a more succinct encoding of problems in fields like AI planning [Rin09], software verification [Kro09] or electronic design automation [VWM15].

The success achieved over the last decades in the field of SAT also inspired the research of the generalized satisfiability problems QSAT and DQSAT. Despite the steady improvement of decision procedures for QSAT and DQSAT, the anticipated performance gains compared to SAT solving are yet to be seen. However, the breakthroughs in SAT solving in the first place facilitated the progress in (D)QSAT solving. Nevertheless, it has been shown that the QBF approach can outperform SAT on particularly bounded synthesis problems [FERT17].

The generalized problems QSAT and DQSAT are complete problems in complexity classes beyond NP. QSAT decides the satisfiability of Quantified Boolean Formulas (QBFs) and is known to be PSPACE-complete [SM73], whereas the satisfiability problem of DQBF (DQSAT) is even NEXPTIME-complete [BCJ14]. The problems contained in PSPACE can be decided by a Turing machine with polynomial memory. Savitch's Theorem [Lip10] implies that the type of the Turing machine (deterministic / non-deterministic) does not matter with respect to the memory consumption. A non-deterministic Turing machine can be simulated by a deterministic Turing machine without significant more memory space. However, although allowing non-determinism does not affect the space

complexity, it might result in a higher time-complexity. Similar to $P \stackrel{?}{=} NP$, we still have no answer whether P is a proper subset of $PSPACE$ or they "collapse" into each other ($P \stackrel{?}{=} PSPACE$). In contrast, a fact that is known, is that NP is strictly contained in $NEXPTIME$. The $NEXPTIME$ complexity class is defined by all problems decidable by a non-deterministic Turing machine in time $2^{n^{O(1)}}$. This means problems cannot even be verified in polynomial time. Regardless of whether $P = NP$, the strict containment in of NP in $NEXPTIME$ implies the strict containment of P in $NEXPTIME$. Interestingly, by showing $EXPTIME \neq NEXPTIME$ it is possible to directly prove $P \neq NP$ by a padding argument [AB09].

As we can see the research on these complete problems is strongly connected to major unsolved problems and open questions in computer science. Although the research was not able to answer those questions, it generated various practical implementations of decision procedures for SAT/QSAT/DQSAT. Especially the breakthroughs and optimizations in SAT solving have made modern SAT solvers a practically applicable tool despite its exponential worst-case time complexity.

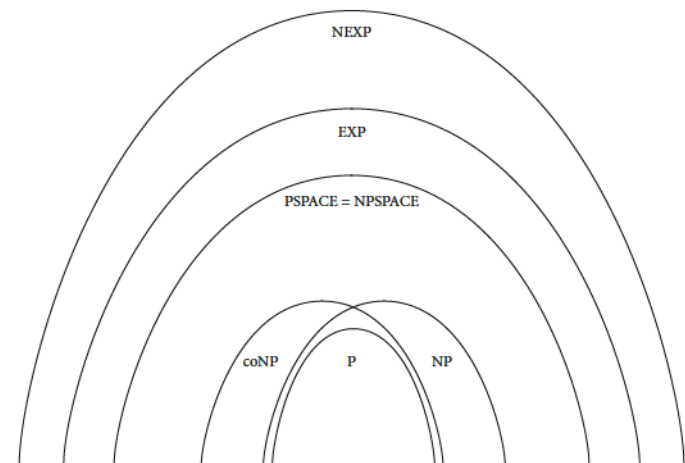


Figure 1.1: Hierarchy of complexity classes

The goal of this thesis is to certify unsatisfiability of QBFs in PEDANT [RSS21]. PEDANT is an expansion-based solver for DQBF which combines the extraction of propositional definitions with Counter-Example Guided Inductive Synthesis (CEGIS) to construct a candidate model. Currently, PEDANT is only able to provide certificates for satisfiable DQBFs. In this thesis, we propose a certificate workflow based on the QRAT proof format, which allows PEDANT to create QRAT certificates for unsatisfiable QBFs. This is a step towards certifying unsatisfiable DQBFs in PEDANT. Although there already exists a sound and complete refutational DQBF proof system called DQRAT [Bli20], the lack of a proof checker makes an implementation of DQRAT in PEDANT pointless at

the moment. The created QRAT certificates can be used to verify the unsatisfiability of the input QBFs using a trusted external QRAT checker. Currently there is only QRAT-TRIM [HSB14a] for checking QRAT proofs.

This thesis is organized as follows. In Chapter 2 we provide an overview of basic notation. Chapter 3 introduces the proof format QRAT followed by a basic explanation of PEDANT's decision procedure in Chapter 4. After that, Chapter 5 covers the proposed certificate structure and Chapter 7 presents the results of our experimental evaluation. Chapter 8 summarizes important points and provides an outlook on future improvements of the certification workflow for PEDANT.

CHAPTER 2

Preliminaries

In this chapter we introduce some standard definitions and notations that we will use in the subsequent chapters.

2.1 Propositional Formulas

Propositional logic, also known as sentential logic and statement logic, deals with propositions and the connection of propositions using logical operators. A proposition can be assigned a truth value. It either can be true or false. We further denote the truth value true by `TRUE` and false by `FALSE`. The main concepts in propositional logic are *variables*, *literals*, *terms*, *clauses* and *formulas*. A *variable* directly represents a proposition and its domain equals $\{\text{TRUE}, \text{FALSE}\}$ which maps to the truth values of a proposition. A *literal* l is either a variable x or the negation of a variable $\neg x$. A *term* is a conjunction of literals $(l_1 \wedge l_2 \wedge \dots \wedge l_n)$ and similar a *clause* is a disjunction of literals $(l_1 \vee l_2 \vee \dots \vee l_n)$. Terms, respectively clauses, can be identified as a set of literals $\{l_1, l_2, \dots, l_n\}$.

Definition 2.1.1 (Propositional Formulas). *The set of propositional formulas [KL99] is inductively defined as:*

1. *Propositional variables are propositional formulas.*
2. *The boolean constants \top and \perp are propositional formulas.*
3. *If φ is a propositional formula, then $\neg\varphi$ is also a propositional formula.*
4. *If φ_1, φ_2 are propositional formulas, then $(\varphi_1 \star \varphi_2)$ is also a propositional formula for \star being a binary operation symbol e.g $\vee, \wedge, \rightarrow, \dots$*

Definition 2.1.2 (CNF). A propositional formula φ is in *Conjunctive Normal Form (CNF)* if it is of the form $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_n$ where the C_i are clauses.

Definition 2.1.3 (DNF). A propositional formula φ is in *Disjunctive Normal Form (DNF)* if it is of the form $\varphi = T_1 \vee T_2 \vee \dots \vee T_n$ where the T_i are terms.

Using Tseitin's encoding [Tse70], each propositional formula can be transformed into a satisfiability equivalent formula in *Conjunctive Normal Form (CNF)*. The size of the transformed formula grows linearly to the size of the input formula. This is a major advantage compared to a naive transformation approach using De Morgan's law and the distributive property, which can result in an exponential blow up of the formula size. The worst-case of the naive transformation approach can be observed by transforming a propositional formula in DNF to its equivalent CNF representation.

$$\begin{aligned}
 \varphi_{DNF} &= (a_1 \wedge b_1 \wedge c_1) \vee (a_2 \wedge b_2 \wedge c_3) \vee (a_3 \wedge b_3 \wedge c_3) \\
 \varphi_{Naive} &= (a_1 \vee a_2 \vee a_3) \wedge (a_1 \vee a_2 \vee b_3) \wedge (a_1 \vee a_2 \vee c_3) \wedge (a_1 \vee b_2 \vee a_3) \wedge \\
 &\quad (a_1 \vee b_2 \vee b_3) \wedge (a_1 \vee b_2 \vee c_3) \wedge \dots \wedge (c_1 \vee c_2 \vee c_3) \\
 \varphi_{Tseitin} &= (\neg a_1 \vee \neg b_1 \vee \neg c_1 \vee x_1) \wedge (a_1 \vee \neg x_1) \wedge (b_1 \vee \neg x_1) \wedge (c_1 \vee \neg x_1) \wedge \\
 &\quad (\neg a_2 \vee \neg b_2 \vee \neg c_2 \vee x_2) \wedge (a_2 \vee \neg x_2) \wedge (b_2 \vee \neg x_2) \wedge (c_2 \vee \neg x_2) \wedge \\
 &\quad (\neg a_3 \vee \neg b_3 \vee \neg c_3 \vee x_3) \wedge (a_3 \vee \neg x_3) \wedge (b_3 \vee \neg x_3) \wedge (c_3 \vee \neg x_3) \wedge \\
 &\quad (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge \\
 &\quad (x_1 \vee x_4 \vee \neg x_5) \wedge (\neg x_1 \vee x_5) \wedge (\neg x_4 \vee x_5) \wedge \\
 &\quad (x_5)
 \end{aligned}$$

Figure 2.1: DNF to CNF Transformations

In Figure 2.1 we can see that the number of clauses in φ_{Naive} grows very fast with respect to the number of literals in the terms and the number of terms in φ_{DNF} . In the example φ_{Naive} contains $3^3 = 27$ clauses. If φ_{DNF} would contain 4 terms with 3 literals the number of clauses in φ_{Naive} would be even $3^4 = 81$. The Tseitin's encoding operates on the subformulas of the initial formula and introduces new variables for them (x_i). $\varphi_{Tseitin}$ illustrates the satisfiability equivalent formula after applying Tseitin's transformation on φ_{DNF} .

Definition 2.1.4 (Variable Assignment). A variable assignment σ is a function mapping variables to truth values (TRUE, FALSE).

$$\sigma : var(\varphi) \mapsto \{TRUE, FALSE\}$$

A variable assignment is called *complete* if all variables of the formula φ are mapped to a truth value. Respectively, if σ misses some variable mappings, it is called a *partial assignment*.

In the following sections we will use a set notation for the description of a variable assignment. For example we write $\sigma = \{x, \neg y\}$ for when the assignment σ maps the variable x to the truth value TRUE (denoted as $\sigma(x) = \text{TRUE}$), respectively maps the variable y to FALSE (denoted as $\sigma(y) = \text{FALSE}$).

Additionally we are going to denote $\text{var}(\varphi)$ as the set of variables occurring in a formula φ . Likewise this function is also used in combination with variable assignments. $\text{var}(\sigma)$ denotes the set of variables for which the variable assignment σ has a mapping. Furthermore we are using $\text{var}(l)$ to identify the variable present in a literal l .

2.2 Quantified Boolean Formulas

Quantified Boolean Formulas (QBF) are an extension of propositional logic. The extension introduces the possibility of explicit quantification over truth values. The addition of the universal quantifier \forall and the existential quantifier \exists not only extends the expressiveness of formulas, but also allows the representation of problems in a concise way.

Definition 2.2.1 (QBF). *The set of Quantified Boolean Formulas is inductively defined as:*

1. *Propositional variables are QBFs.*
2. *The boolean constants \top and \perp are QBFs.*
3. *If φ is a QBF, then $\neg\varphi$ is also a QBF.*
4. *If φ_1, φ_2 are QBF, then $(\varphi_1 \star \varphi_2)$ is also a QBF for $\star \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$.*
5. *If φ is a QBF and $x \in \text{var}(\varphi)$, then $\exists x.\varphi$ and $\forall x.\varphi$ are also QBFs.*

We denote the set of universally quantified variables of a QBF Φ as U_Φ and the set of existentially quantified variables as E_Φ .

Similar to propositional formulas, QBFs can also be in specific normal forms. An important normal form of QBFs is the *Prenex Normal Form* (PNF).

Definition 2.2.2 (Prenex Normal Form). *A QBF in prenex normal form consists of*

- (a) *a sequence of quantified variables $\mathcal{Q} = Q_1x_1 \dots Q_nx_n$ called quantifier prefix with $Q_i \in \{\forall, \exists\}$ and $x_i \in \text{var}(\varphi)$ at nesting level i*
- (b) *a propositional formula φ called the matrix*

Due to the linear ordering of the prefix by nesting level, we get that variables *precede* other variables with higher nesting level (i.e. x_i *precedes* x_{i+1}). We write $x_i \prec_\Phi x_j$ if

$$\Phi = \underbrace{\forall x \exists y \exists z}_{\text{prefix}} \cdot \underbrace{(x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg x \vee z)}_{\text{matrix}}$$

Figure 2.2: QBF in PCNF

$1 \leq i < j \leq n$ and $Q_i \neq Q_j$. If the context is clear, we omitted the subscript for the formula Φ .

Variable occurrences are called *bound*, if the occurrences of this variable are included in the scope of a quantified block of the corresponding variable. In a PNF the whole matrix is in scope of all quantifications in the prefix. If a variable occurrence is not *bound*, it is called *free* occurrence. We call a variable x of a formula Φ *free* if there is a *free* occurrence of x in Φ . If a QBF does not contain any *free* variable, we call this formula *closed*. With respect to a PNF, the prefix of a *closed* PNF needs to contain a quantification for each variable $x \in \text{var}(\varphi)$. In other words, for each variable $x \in \text{var}(\varphi)$ there must exist a quantification $Q_i x_i \in \mathcal{Q}$ such that $x = x_i$.

Each QBF can be transformed into an equivalent QBF in prenex normal form. Further, due to the separation of matrix and prefix in a PNF, the matrix can be independently represented in several different normal forms. If the matrix of a PNF is in Conjunctive Normal Form it is called *Prenex Conjunctive Normal Form* (PCNF). We further consider QBFs as closed and in prenex conjunctive normal form.

Definition 2.2.3 (Prenex Conjunctive Normal Form (PCNF)). *A QBF Φ is in prenex conjunctive normal form if $\Phi = \mathcal{Q}.\varphi$ is in prenex normal form (PNF) and its matrix φ is in conjunctive normal form (CNF).*

2.3 Dependency QBF

Dependency Quantified Boolean Formulas (DQBF) is a further generalization of QBF and allows the explicit specification of the *dependency sets* of existential variables. Like for QBFs we consider DQBFs in Prenex Conjunctive Normal Form (PCNF) which puts the variable quantifications in front of a propositional formula in CNF.

Definition 2.3.1 (DQBF). *In a DQBF in PCNF denoted as $\Phi = \mathcal{Q}.\varphi$ the matrix is a propositional formula in CNF. The quantifier prefix \mathcal{Q} explicitly encodes the dependency sets of existential variables and is a sequence*

$$\mathcal{Q} = \forall u_1 \dots \forall u_n \exists e_1(D_1) \dots \exists e_m(D_m)$$

where $U_\Phi = \{u_1 \dots u_n\}$ and $E_\Phi = \{e_1 \dots e_m\}$ are again the sets of universally and existentially quantified variables. Additionally for $1 \leq i \leq m$ the dependency sets D_i are subsets of U_Φ and represent the dependencies of e_i .

As we can see the prefix of a DQBF does not rely anymore on the order of the variable quantifications. In general the quantifications in the prefix of a DQBF can be sorted in any order but for the sake of readability and comprehensibility all universal quantifications precede all existential quantifications. In a regular QBF the dependency sets D_i are given implicitly by the ordering of the prefix. This means the set D_i contains all universal variables u preceding the corresponding existential variable e_i . To be precise the dependency set of a existential variable e_i is defined as $D_{e_i} = \{u \mid u \in U_\Phi \wedge u \prec_\Phi e_i\}$.

2.4 QSAT

The Satisfiability-Problem of quantified boolean formulas (QSAT) asks whether a given QBF is satisfiable.

2.4.1 Evaluation of Propositional Formulas

In order to define the evaluation of quantified boolean formulas, we first need to define a reduced evaluation function of propositional formulas. Our propositional evaluation function does not need to take variables into account. This means we are only able to evaluate the truth value of a propositional formula without any variables. Propositional formulas of this kind only consist of boolean constants, negation and connectives.

Definition 2.4.1 (Evaluation of Propositional Formulas without Variables). *The evaluation of a propositional formula φ without variables, is done by recursively applying the following rules:*

1. if $\varphi = \top$, then $value(\varphi) = TRUE$
2. if $\varphi = \perp$, then $value(\varphi) = FALSE$
3. if $\varphi = \neg\varphi'$, then $value(\varphi) = TRUE$ if $value(\varphi') = FALSE$
4. $\varphi = \varphi' \star \varphi''$ and \star is a binary connective in $\{\wedge, \vee, \Rightarrow\}$, then $value(\varphi) = value(\varphi') \star value(\varphi'')$

Furthermore, we need the concept of variable substitution to define the evaluation of QBFs. When applying a variable substitution on a propositional formula φ , all occurrences of a variable x are replaced consistently by another variable, symbol or even whole propositional formulas. In our case we only need the substitution of variables with boolean constants (\top, \perp). We denote these substitution operations as $\varphi[x/\top]$ and $\varphi[x/\perp]$. Figure 2.3 illustrates how the variable substitution works by giving simple examples.

Using variable substitution we can define the application of a variable assignment σ on a propositional formula φ . $\varphi[\sigma]$ denotes the propositional formula generated by iterative substitution of variables with corresponding boolean constants with respect to the variable assignment σ . Each variable x for which a truth assignment is available

$$\begin{aligned}
\varphi &= (x \vee y \vee z) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg z) \\
\varphi[x/\top] &= (\top \vee y \vee z) \wedge (\neg \top \vee y) \wedge (\neg \top \vee \neg z) \\
\varphi[y/\perp] &= (x \vee \perp \vee z) \wedge (\neg x \vee \perp) \wedge (\neg x \vee \neg z)
\end{aligned}$$

Figure 2.3: Variable Substitution Examples

in σ gets consistently replaced either with \top or \perp . If $\sigma(x) = \text{TRUE}$ the variable x gets replaced with \top , respectively if $\sigma(x) = \text{FALSE}$ the variable x gets replaced with \perp .

Definition 2.4.2 (Application of a Variable Assignment on a Propositional Formula). *Let φ be a propositional formula and let σ be a variable assignment. Further let $f : \{\text{TRUE}, \text{FALSE}\} \mapsto \{\top, \perp\}$ be a function mapping the truth value TRUE to \top , respectively FALSE to \perp . The application of σ on φ in terms of variable substitution is defined as:*

$$\varphi[\sigma] = \varphi[x_1/f(\sigma(x_1), x_2/f(\sigma(x_2), \dots, x_n/f(\sigma(x_n)) \text{ for } x_i \in \text{var}(\sigma)$$

For the sake of completeness we briefly define the evaluation of a propositional formula with variables under a variable assignment σ .

Definition 2.4.3 (Evaluation of Propositional Formulas under Variable Assignment). *The evaluation of a propositional formula φ under a variable assignment σ , denoted as $\text{value}(\varphi, \sigma)$, can be defined the following way:*

1. if $\varphi[\sigma]$ does not contain any variables, then $\text{value}(\varphi, \sigma) = \text{value}(\varphi[\sigma])$
2. otherwise $\text{value}(\varphi, \sigma) = \varphi[\sigma]$

A propositional formula φ is called *satisfiable* if there exists a truth assignment σ , such that $\text{value}(\varphi, \sigma) = \text{TRUE}$. Similarly, a propositional formula φ is called *inconsistent/unsatisfiable* if all truth assignments σ evaluate φ to FALSE .

In order to show the satisfiability of a propositional formula φ it is not always necessary to find a truth assignment σ that assigns all variables $x \in \text{var}(\varphi)$ a truth value. For showing satisfiability it is often sufficient to find a partial assignment which only covers a subset $X \subseteq \text{var}(\varphi)$. Evaluating a propositional formula upon a partial assignment $\text{value}(\varphi, \sigma)$ might not result in a truth value but a new propositional formula φ' . On such occasions the new propositional formula φ' substituted all variable occurrences in the original formula φ covered by the applied partial assignment with the corresponding boolean constants. However, with respect to propositional formulas in conjunctive normal form we can easily observe that it can be sufficient to find a partial assignment to evaluate it to a truth value. First, a clause $C = (l_1 \vee l_2 \vee \dots \vee l_n)$ under an assignment σ is satisfied if at least one literal l_i is mapped in the assignment σ to the truth value TRUE .

In order to satisfy a propositional formula in conjunctive normal form all clauses need to be satisfied. This means it is sufficient for an assignment to map at least one literal of all clauses to TRUE. By extending Definition 2.4.3 by this rule it is possible to evaluate a CNF to TRUE using only a partial assignment of variables.

2.4.2 Evaluation of QBFs

Using our reduced evaluation function on propositional formulas containing no variables and variable substitutions, we are now able to define the evaluation of QBFs. The truth value of a QBF can be determined by applying the following rules recursively.

Definition 2.4.4 (Evaluation of Quantified Boolean Formulas). *Let $\Phi = Q.\varphi$ be a closed QBF in prenex normal form.*

1. if $Q = \exists x Q'$, then $value(\Phi) = TRUE$ if $value(Q'.\varphi[x/\top]) = TRUE$ or $value(Q'.\varphi[x/\perp]) = TRUE$
2. if $Q = \forall x Q'$, then $value(\Phi) = TRUE$ if $value(Q'.\varphi[x/\top]) = TRUE$ and $value(Q'.\varphi[x/\perp]) = TRUE$
3. if $Q = \emptyset$, then the matrix φ does not contain any variables and $value(\Phi) = value(\varphi)$

In order to prove the satisfiability of a QBF Φ , it is not sufficient anymore to just provide an assignment of the variables. We need a set of functions that describe the truth value of existential variables with respect to the assignment of universal variables. For each existentially quantified variable $e_i \in E_\Phi$ let f_{e_i} be a function representing the truth value of e_i with respect to the assignment of its dependency set D_{e_i} . In the case of a QBF those sets D_{e_i} are the preceding universally quantified variables of the existentially quantified variable e_i .

Another way of picturing the evaluation of a QBF is using the idea of a two-player game, where two players assign truth values to their variables. Player P_\forall is the owner of all universally quantified variables and player P_\exists is the owner of all existentially quantified variables. The players pick the variables from left to right with respect to the quantifier ordering in the prefix. At each turn of the game the player owning the outermost unassigned variable assigns a truth value of their choice to the variable. At the time of each truth value assignment the players can only see the already assigned truth values of preceding variables. The goal of the universal player P_\forall is to falsify the formula, respectively the goal of the existential player P_\exists is to satisfy the formula. If a player is able to "win" the game by achieving its goal regardless of the opponent's choice of assignments, then this player has a so-called winning strategy. The winning strategy of player P_\exists is called winning \exists -strategy and is equivalent to a satisfiability model. According to the goal of the existential player, the formula needs to be satisfiable. Analogously, if the universal player finds a winning strategy, the formula is unsatisfiable.

2.4.3 Evaluation of DQBFs

The evaluation of a DQBF cannot be described in an intuitive way similar to the evaluation of QBF. There is no general solution of finding an order to assign truth values to variables due to explicitly stated dependency sets D_i . In order to show the satisfiability of a DQBF Φ we need a satisfiability model, which generally speaking is a set of boolean functions $\{f_{e_1}, f_{e_2}, \dots, f_{e_m}\}$ satisfying the matrix of Φ in combination with any assignment of the universal variables.

Definition 2.4.5 (Satisfiability Model). *Let $\Phi = \mathcal{Q}.\varphi$ be a closed QBF in prenex normal form with $E_\Phi = \{e_1, e_2, \dots, e_m\}$. Let D_{e_i} be the dependency set of the existential variable e_i . For $1 \leq i \leq m$ let $f_{e_i} = f(x_1, x_2, \dots, x_n)$ be a function which represent the truth value of e_i given a truth assignments of all variables in D_{e_i} (here denoted as x_1, \dots, x_n). Lets $F(\sigma) = \{f_{e_1}(\sigma_{D_{e_1}}), f_{e_2}(\sigma_{D_{e_2}}), \dots, f_{e_m}(\sigma_{D_{e_m}})\}$ be a satisfiability model of Φ for each assignment σ of universal variables $u \in U_\Phi$, such that $\sigma \cup F(\sigma)$ satisfies the matrix φ .*

A DQBF is TRUE if it has a model and FALSE otherwise.

2.5 Q-Resolution

Q-Resolution [BKF95, GNT06] is an inference rule and a complete and sound technique for showing the satisfiability of QBFs in PCNF. It is an extension of propositional resolution [Rob65], which is as well a complete and sound technique for showing the satisfiability of propositional formulas in CNF.

Propositional resolution operates on a set of clauses and generates a new clause implied by two clauses containing complementary literals.

Definition 2.5.1 (Propositional Resolution). *Let φ be a CNF and let $(C_1 \vee x)$ and $(C_2 \vee \neg x)$ be two clauses of the propositional formula φ with complementary literals of the variable x .*

$$\frac{C_1 \vee x \quad \neg x \vee C_2}{C_1 \vee C_2} \quad (\text{resolution})$$

Resolving those clauses on pivot variable x yields a new clause $C = C_1 \vee C_2$ without the variable x called the resolvent.

Because the resolvent C is a logical consequence of the inference's premises $(C_1 \vee x)$ and $(C_2 \vee \neg x)$, the resolvent clause can be added to φ and produce a logically equivalent formula φ' [Rob65]. A propositional formula is unsatisfiable if the empty clause can be derived using resolution.

We further are using \bowtie as a symbol denoting the resolution operation.

In order to lift resolution from propositional CNFs to QBFs in PCNF, the additional quantifier prefix needs to be taken into account. In general there are two additional properties necessary:

1. The pivot variable to perform resolution on must be existentially quantified.
2. There exists no Q-resolvent if the intended resolvent is a tautology.

Additionally the Q-Resolution proof system allows the removal of universally quantified literals using *Universal Reduction*.

Definition 2.5.2 (Universal Reduction [BKE95]). Let $\Phi = Q.\varphi$ be a QBF in PCNF and let $C = (u \vee l_1 \vee \dots \vee l_n)$ be a clause in φ with u being universally quantified and $l_i \prec u$ for all $1 \leq i \leq n$ and $l_i \in E_\Phi$.

$$\frac{C}{C \setminus \{u\}} \quad (\text{universal reduction})$$

The universal literal u can be safely removed from clause C without changing the truth value of Φ , iff all existential literals $e \in C$ precede u . The resulting clause $C \setminus \{u\}$ is called the forall reduct.

A Q-resolution derivation is a sequence of clauses such that each clause is either contained in the matrix or can be obtained by resolution or reduction on clauses appearing earlier in the sequence. Similar to propositional resolution, those clauses obtained by applying Q-resolution can be added to the matrix of Φ without changing the satisfiability of Φ . Therefore one can show, a QBF in PCNF is unsatisfiable iff there exists a derivation [BKE95] which contains the empty clause. A derivation containing the empty clause is also called a refutation.

2.5.1 Q-Resolution Proof Example

Here we are going to present an example of a Q-resolution refutation. For this example we are using the following unsatisfiable QBF:

$$\Phi = \forall w \exists x \exists y \forall z. (\neg w \vee y \vee z) \wedge (\neg w \vee x) \wedge (\neg x \vee \neg y \vee z)$$

In case Φ is indeed unsatisfiable we should be able to generate a q-resolution derivation which contains the empty clause. In Figure 2.4 we illustrate an example refutation of the stated QBF using the Q-resolution rules.

$$\begin{array}{c}
\frac{\neg w \vee y \vee z}{\neg w \vee y} \text{ (UR)} \qquad \frac{\neg w \vee x \qquad \frac{\neg x \vee \neg y \vee z}{\neg x \vee \neg y} \text{ (UR)}}{\neg w \vee \neg y} \text{ (RES)} \\
\hline
\frac{\neg w}{\emptyset} \text{ (UR)}
\end{array}$$

Figure 2.4: Q-Resolution Refutation

2.6 Expansion-based Proof System

Another sound and complete proof system is the expansion-based proof system $\forall\text{Exp}+\text{Res}$. This technique has been established for both QBF [JMS15] and DQBF [BBC⁺19]. Similar to Q-resolution the $\forall\text{Exp}+\text{Res}$ proof system is composed of two parts by utilizing two proof-rules. As the name of the proof system already indicates it uses expansion of universally quantified variables as well as propositional resolution. The core idea of $\forall\text{Exp}+\text{Res}$ is to first expand a QBF formula over universally quantified variables such that only existentially quantifications are left. After the expansion the QBF formula can be treated as a propositional formula and propositional resolution can be applied.

2.6.1 Expansion

The expansion of a QBF can be applied over universal quantified variables as well as existential quantified variables. The basic principle of an expansion is to translate a variable quantification to its equivalent encoding in propositional logic. Thereby the semantics of the individual quantifiers (\forall, \exists) come into play. The expansion effectively creates two copies of the formula Φ in which the truth value of the expansion variable x is fixed. One copy in which $x = \text{TRUE}$ and another copy for $x = \text{FALSE}$. By conjoining/disjoining together these two copies we can implement the behaviour of the initial quantification. At the expansion of a universally quantified variable both copies must be satisfiable which is realizable by a conjunction. For existential expansions it is enough to have at least one satisfiable copy which can be encoded using a disjunction. Formally the applied formula transformations of the expansions can be described with the following equivalences: (note that we are using the same on substitution based evaluation of QBFs as defined in Section 2.4)

1. $\forall x. \Phi = \Phi[x/\perp] \wedge \Phi[x/\top]$
2. $\exists x. \Phi = \Phi[x/\perp] \vee \Phi[x/\top]$

By applying expansions we can observe two problems:

1. Applying either universal or existential expansion on a QBF in PCNF does not preserve the prenex normal form. The application of universal expansion on $\Phi = \exists x \forall y \exists z. \varphi$ results in the formula $\Phi' = \exists x. (\exists z. \varphi[y/\perp]) \wedge (\exists z. \varphi[y/\top])$. Obviously Φ' is not in prenex normal form anymore.
2. Applying existential expansion does (a) not preserve the prenex normal form and does (b) not even result in a CNF.

Because $\forall\text{Exp}+\text{Res}$ only considers expansions of universal variables we do not need to go further into details of existential expansion. In order to resolve the issues with universal expansion we need to re-establish the prenex normal form. In our example to get from $\Phi' = \exists x. (\exists z. \varphi[y/\perp]) \wedge (\exists z. \varphi[y/\top])$ to a valid PCNF again, the introduction of new variables is required. In this example the creation of two copies of z are needed each for one part of the expansion. The copy z^y , respectively $z^{\bar{y}}$, represents the individual truth assignment of z in the corresponding sub-QBF with respect of the assignment of $y = \text{TRUE}$, respectively $y = \text{FALSE}$. After those modifications the final QBF in PCNF is $\Phi'' = \exists x \exists z^y \exists z^{\bar{y}}. \varphi[y/\perp][z/z^y] \wedge \varphi[y/\top][z/z^{\bar{y}}]$.

Although using universal expansions can eliminate universal quantifications, it comes at the cost of the exponential growth of the matrix. In the worst case each expansion effectively doubles the number of clauses in the QBF. However, it is sometimes possible to avoid the worst case scenario by partially expanding the formula instead of a complete expansion. For example in $\forall x \forall y \exists z. (x \vee z) \wedge (x \vee \neg z) \wedge (y \vee z)$ it is sufficient to solely expand x for the assignment FALSE. The expansion yields the copied clauses $(\perp \vee z^{\bar{x}})$ and $(\perp \vee \neg z^{\bar{x}})$. Semantically this results already in a conflict $z^{\bar{x}} \wedge \neg z^{\bar{x}}$ and no further expansions are necessary in order to prove the unsatisfiability of the QBF. Modern expansion-based solvers obviously try to utilize this technique for the sake of performance.

2.6.2 $\forall\text{Exp}+\text{Res}$

The $\forall\text{Exp}+\text{Res}$ proof system makes use of the partial expansion technique as well as propositional resolution. Like a Q-resolution derivation, a derivation using $\forall\text{Exp}+\text{Res}$ is again a sequence of clauses. The clauses in this sequence are either contained in the matrix or can be obtained using universal expansion or propositional resolution.

Because the clauses introduced by the axiom rule are part of the complete expansion of the formula they can be added to the matrix without changing its satisfiability. A similar argument regarding satisfiability of a QBF as presented for q-resolution derivations applies to $\forall\text{Exp}+\text{Res}$ derivations. A QBF in PCNF is unsatisfiable iff there exists a $\forall\text{Exp}+\text{Res}$ derivation which contains the empty clause [JMS15].

$$\frac{}{\{a^{\tau(a)} \mid a \in C \wedge a \in E_{\Phi}\}} \quad (\text{axiom rule})$$

Using the axiom rule we can derive a new clause with new annotated literals. In order to apply this rule the following requirements must hold:

1. C is a clause in the matrix of Φ
2. τ is a complete universal assignment that falsifies every universal literal in C
3. $\tau(a) := \{l \in \tau \mid \text{var}(l) \in D_i\}$, where $\text{var}(a) = e_i$

$$\frac{C_1 \vee x^{\tau} \quad \neg x^{\tau} \vee C_2}{C_1 \vee C_2} \quad (\text{resolution})$$

This proof system allows propositional resolution over literals created and annotated by the axiom rule.

Table 2.1: $\forall\text{Exp}+\text{Res}$ rules for DQBF

2.6.3 $\forall\text{Exp}+\text{Res}$ Proof Example

Here we are going to present an example of a $\forall\text{Exp}+\text{Res}$ refutation of the already presented unsatisfiable QBF in the q-resolution proof example. Because we already have proven the unsatisfiability of this formula using a q-resolution refutation, we know there also must exist a derivation containing the empty clause using the $\forall\text{Exp}+\text{Res}$ proof system. Below we illustrate a $\forall\text{Exp}+\text{Res}$ refutation.

$$\Phi = \forall w \exists x \exists y \forall z. (\neg w \vee y \vee z) \wedge (\neg w \vee x) \wedge (\neg x \vee \neg y \vee z)$$

$$\begin{array}{c}
\frac{}{y^w} \text{ (AX)} \qquad \frac{}{\neg y^w \vee \neg x^w} \text{ (AX)} \\
\hline
\frac{}{\neg x^w} \text{ (RES)} \qquad \frac{}{x^w} \text{ (AX)} \\
\hline
\frac{}{\emptyset} \text{ (RES)}
\end{array}$$

Figure 2.5: $\forall\text{Exp}+\text{Res}$ Refutation

Note that each of the original matrix clauses is used in one of the three axiom lemmas in this refutation. Conveniently, the same universal assignment can be applied in all axiom lemmas. The complete universal assignment used is $\{w, \neg z\}$. Although the matrix clause $(\neg x \vee \neg y \vee z)$ does not contain the universal variable w , the existentially quantified variables x and y is annotated with w anyway because the axiom rule operates on a complete universal assignment.

Introduction of QRAT

The Satisfiability Problem of propositional formulas (SAT) and QBFs (QSAT) are decision problems. Decision problems are defined in a way, such that the problem question can be answered with either yes or no. In this case SAT, respectively QSAT, ask whether a given propositional formula, respectively QBF, is satisfiable. The main purpose of a certificate is to provide evidence that the yes/no-answer of the problem is indeed correct. In other word, by means of a certificate we can verify the satisfiability or unsatisfiability of a given input formula. Because SAT is well known to be a NP-complete problem [Coo71], we know the certificate size is polynomial bounded in the size of the problem instance. Furthermore we know there must be a verification algorithm which time complexity is also bounded by a polynomial in the size of the problem instance [KT06]. Therefore the certificates of the SAT problem are rather easy to define. In order to certify a satisfiable SAT instance, it is sufficient to provide a satisfying truth assignment of the variables of the input formula. For example, it is easy to verify the satisfiability of the following propositional formula in combination with the given truth assignment:

$$\begin{aligned}\varphi &= (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg z) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg y \vee z) \\ \sigma &= \{x, \neg y, z\}\end{aligned}$$

Given an assignment σ and a propositional formula in conjunctive normal form, checking whether all clauses are satisfied can be done quite efficiently. In contrast to prove unsatisfiable instances, the certificate needs to provide evidence, that there cannot be a single assignment which satisfies the input formula. This evidence can be provided by a resolution proof which also can be checked efficiently. In both cases, there is an algorithm which can verify the result in polynomial time.

The verification of the output of the QSAT problem is significantly more complex than the equivalent of propositional formulas. For certifying satisfiable QBFs it is not sufficient anymore to provide just a satisfying assignment because of the introduction of quantifiers. In order to prove the satisfiability of a QBF, we need so-called Skolem-functions which

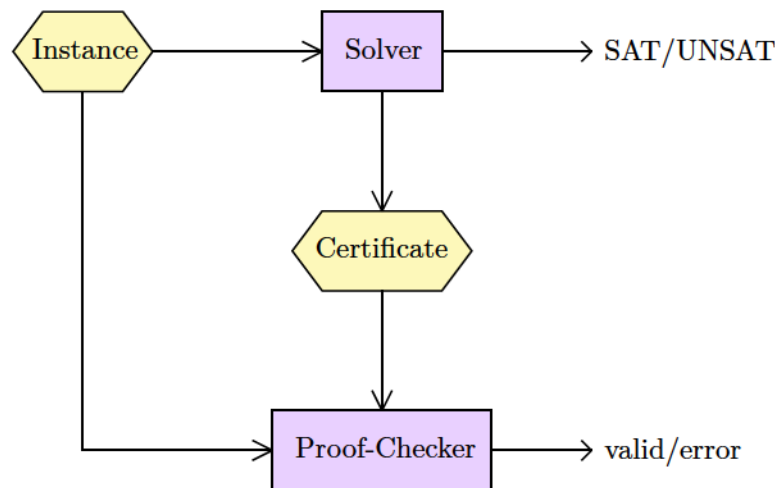


Figure 3.1: Certification Workflow

encode a strategy to set the existentially quantified variables (with respect to the assignment of preceding universal variables) to satisfy the QBF. For invalid QBFs, a certificate should answer the question, why there cannot be any satisfiability model, ideally in a concise way. Similar to certificates of unsatisfiable propositional formulas, the certification of invalid QBFs can also be done using resolution proofs (Q-Resolution).

3.1 Proof-Formats

Besides the previously mentioned methods for satisfiability certification, there are more sophisticated proof formats for certifying the satisfiability of propositional formulas and QBFs. The de-facto standard proof-format for propositional formula is DRAT [HHW13b] (Deletion Resolution Asymmetric Tautology). Such agreement on certain proof-systems, in order to validate the result of QBF-solvers, has not yet been established in the QSAT community. However, the QSAT community has been discussing possible proof formats for a long time [JBS⁺07]. One of the latest proof formats is QRAT [HSB14b] (Quantified Resolution Asymmetric Tautology), which is an extension of DRAT.

3.2 DRAT

The standard proof format for propositional formulas relies on the addition and deletion of redundant clauses. The basic idea behind this proof format is, given a CNF φ , a redundant clause C can be added to φ , respectively deleted from φ , and preserve the satisfiability, respectively unsatisfiability. The redundancy property that is applied in DRAT is also eponymous for the name of DRAT. Redundancy is checked in DRAT using the RAT property [HHW13a] (Resolution Asymmetric Tautology).

Definition 3.2.1 (Asymmetric Literal Addition (ALA) [HJB10]). *Let C be a clause and let φ be a CNF. The clause $ALA(\varphi, C)$ is the unique clause obtained by applying the following extension rule until fixpoint: if $l_1, \dots, l_k \in C$ and there is a clause $(l_1 \vee \dots \vee l_k \vee l) \in \varphi \setminus \{C\}$ for some literal l , let $C := C \cup \{l\}$.*

Definition 3.2.2 (Asymmetric Tautology (AT)). *Let C be a clause and let φ be a CNF. The clause C has property AT with respect to φ iff $ALA(\varphi, C)$ is a tautology.*

Unit-Propagation

Unit-propagation or Boolean Constraint Propagation (*BCP*) is the process of simplifying a CNF φ based on unit clauses. A clause is unit if it contains exactly one literal. Because in a CNF each clause needs to be satisfied, we know that this literal must be assigned to TRUE. As a result if there is a unit clause $(l) \in \varphi$ we say *BCP* assigns the literal l to TRUE, respectively \bar{l} to FALSE. As a result we can remove the unit-clause and remove all clauses which contain the literal l , because these clauses are already satisfied by the literal l . Furthermore we can remove \bar{l} from all clauses, because \bar{l} can not contribute anymore to satisfy any clause. This process is repeated until no unit clause can be found anymore. The resulting formula is denoted as $BCP(\varphi)$. We say *BCP* derives a conflict if it is possible to derive the empty clause $\emptyset \in BCP(\varphi)$.

Calculating whether a clause C has property AT with respect to a CNF φ can also be formulated in terms of unit-propagation (or boolean constraint propagation *BCP*). The property AT is also known as Reverse Unit Propagation (RUP).

Definition 3.2.3 (Reverse Unit Propagation (RUP)). *$ALA(\varphi, C)$ is a tautology if $BCP((\varphi \setminus \{C\}) \cup \bar{C})$ derives a conflict, where \bar{C} denotes a set of unit clauses that falsify all literals in C .*

Definition 3.2.4 (Resolution Asymmetric Tautology (RAT)). *A clause C has property RAT with respect to a CNF φ if there exists a literal $l \in C$ such that for all clauses $D \in \varphi$ with $\bar{l} \in D$, each resolvent $C \bowtie D$ has property AT.*

Notice that RAT property is a generalization of the AT property. A clause with property AT with respect to a formula φ , has also property RAT with respect to φ .

$$\begin{aligned} \varphi &= (a \vee b) \wedge (b \vee c) \wedge (\neg b \vee \neg c) \\ C_{AT} &= (a \vee \neg c) \\ C_{RAT} &= (\neg a \vee c) \end{aligned}$$

Figure 3.2: Example illustrating AT/RAT property of a clause

With Figure 3.2 we want to demonstrate the AT and the RAT property. In order to show C_{AT} having the AT property we can either calculate $ALA(\varphi, C_{AT})$ or apply Reverse Unit Propagation. When manually calculating $ALA(\varphi, C_{AT})$ we can apply the extension rule twice before reaching a fixpoint. First we can use $(a) \in C$ with $(a \vee b) \in \varphi$ to reach

p cnf 4 8		
1 2 -3 0	-1 0	-1 0
-1 -2 3 0	2 0	d -1 -2 3 0
2 3 -4 0	0	d -1 -3 -4 0
-2 -3 4 0		d -1 2 4 0
-1 -3 -4 0		2 0
1 3 4 0		d 1 2 -3 0
-1 2 4 0		d 2 3 -4 0
1 -2 -4 0		0
(a) DIMACS CNF	(b) DRAT proof	(c) DRAT proof with delete statements

Figure 3.3: Example DRAT Refutations

$C = (a \vee \neg c \vee \neg b)$. The second and last application of the extension rule uses $(\neg b) \in C$ with $(\neg b \vee \neg c) \in \varphi$ to reach the fixpoint $C = (a \vee \neg c \vee \neg b \vee c)$. The unique clause $ALA(\varphi, C_{AT}) = (a \vee \neg c \vee \neg b \vee c)$ is a tautology and therefore C_{AT} has the AT property. It is also easy to see that $BCP((a \vee b) \wedge (b \vee c) \wedge (\neg b \vee \neg c) \wedge (\neg a) \wedge (c))$ is able to derive a conflict. For each literal in C_{RAT} there is only one clause in φ to build the resolvent with. For literal $\neg a$ the resolvent is $(b \vee c)$ and for literal c the resolvent is $(\neg a \vee \neg b)$. Applying RUP on these resolvents both lead to conflicts and therefore C_{RAT} has the RAT property on both $\neg a$ as well as c .

A DRAT proof is a sequence of clause additions and clause deletions. The notation of a DRAT proof is based on the DIMACS notation of a CNF, where clauses are represented as a sequence of literals terminated by 0. In a DRAT proof the clause additions and the clause deletions are represented as separate lines. A clause addition is like in the DIMACS notation just a sequence of literals terminated by 0. Clause deletion statements are additionally prefixed with the character "d". In Figure 3.3 we can see examples of a CNF in DIMACS format and two DRAT proofs. Deletion informations are not necessary for proving unsatisfiability but can speed up the verification process by reducing the formula size [HHW13b].

Let φ be a CNF and let $P = \{L_0, L_1, \dots, L_{|P|}\}$ be a DRAT proof for φ . Let $L_i = \{p_i, C_i\}$ be a representation of the i^{th} line of the DRAT proof P with p_i being the prefix of the line and C_i being the corresponding clause of the line. Each line modifies a formula φ_P^i , by either adding or removing a clause, and thus generating a new formula φ_P^{i+1} .

$$\varphi_P^i = \begin{cases} \varphi & \text{if } i = 0 \\ \varphi_P^{i-1} \setminus \{C_i\} & \text{if } p_i = \text{"d"} \\ \varphi_P^{i-1} \cup \{C_i\} & \text{otherwise} \end{cases}$$

In order for a DRAT proof P to be a unsatisfiability certificate of a propositional formula

φ , all added clauses need to have the RAT property with respect to the current formula φ_P^i . Additionally P needs to end with the empty clause.

3.3 QRAT

As seen in definition 3.2.4, ALA and AT are sufficient for the definition of the RAT property for propositional logic. In order to lift up the RAT property to QBFs, we need to take quantifier dependencies into account which are captured by the notion of *outer clauses* and *outer resolvents* [HSB14b].

Definition 3.3.1 (Outer Clause). *Let C be a clause occurring in a QBF $\Phi = \mathcal{Q}.\varphi$. The outer clause of C on literal $l \in C$, denoted by $O(\Phi, C, l)$, is given by the clause $\{k \mid k \in C, k \prec_\Phi l, k \neq l\}$.*

Definition 3.3.2 (Outer Resolvent). *Let C be a clause with $l \in C$ and D a clause occurring in a QBF $\Phi = \mathcal{Q}.\varphi$ with $\bar{l} \in D$. The outer resolvent of C with D on literal l with respect to Φ , denoted by $R(\Phi, C, D, l)$, is given by the clause $O \cup (C \setminus \{l\})$ if $l \in U_\Phi$ and by $O \cup C$ if $l \in E_\Phi$ assuming $O = O(\Phi, D, \bar{l})$.*

Definition 3.3.3 (Quantified Resolution Asymmetric Tautology (QRAT)). *Given a QBF $\Phi = \mathcal{Q}.\varphi$ and a clause C . Then C has QRAT on literal $l \in C$ with respect to Φ iff it holds for all clauses $D \in \varphi$ with $\bar{l} \in D$ that $ALA(\varphi, R)$ is a tautology for the outer resolvent $R = R(\Phi, C, D, l)$.*

Given the QBF $\Phi = \forall x \exists y \exists z. C \wedge D \wedge E$ with $C = (x \vee y)$, $D = (\neg x \vee z)$ and $E = (y \vee \neg z)$ we will illustrate the terms Outer Clause and Outer Resolvent:

$$\begin{array}{ll}
 O(\Phi, C, y) & = (x) \\
 O(\Phi, C, x) & = \emptyset \\
 O(\Phi, C, z) & = (x \vee y) \\
 O(\Phi, E, \neg z) & = (y) \\
 R(\Phi, D, E, z) & = (\neg x \vee z \vee y) \\
 R(\Phi, E, D, z) & = (y \vee \neg z \vee \neg x) \\
 R(\Phi, (x \vee z), E, z) & = (x \vee z \vee y) \\
 R(\Phi, (x \vee z), D, x) & = (z)
 \end{array}$$

Constructing the outer resolvent is an asymmetric operation since $R(\Phi, D, E, z) \neq R(\Phi, E, D, z)$. The clause $F = (x \vee z)$ has QRAT on z with respect to Φ . The only clause to build a resolvent with is E (only clause with $\neg z$). The produced outer resolvent with respect to Φ is $(x \vee z \vee y)$. Because $ALA(C \wedge D \wedge E, (x \vee z \vee y)) = (x \vee \neg x \vee y \vee \neg y \vee z \vee \neg z)$ is a tautology all conditions are satisfied for F having the QRAT property on literal z with respect to Φ and therefore can be added to Φ without changing its satisfiability.

Similar to DRAT proofs, a QRAT-proof is also a sequence of clause additions, clause deletions and, additionally, clause modifications [HSB14b]. The addition of a clause having the QRAT or AT property with respect to a QBF Φ is denoted as **QRATA**. QRAT supports the introduction of new variables. In this process the prefix of the QBF has to be extended in order to maintain a closed QBF. Generally speaking the newly introduced variable can be placed at any position in the prefix. However QRAT places

newly introduced variables in the innermost active existential quantifier block. Analog the deletion of a clause having the QRAT or AT property with respect to a QBF Φ is denoted as **QRATE**. By clause modifications we are considering the elimination of a universally quantified literal out of a clause. This can be achieved by either using the QRAT property or using *extended universal reduction*. The former elimination method is denoted as **QRATU** and is the removal of a universal literal l from a clause C which has QRAT on l with respect to a QBF Φ . Extended universal reduction denoted as **EUR** is based on resolution paths [VG11] and allows the removal of independent universal variables.

Definition 3.3.4 (Inner Clause). *Let C be a clause occurring in a QBF $\Phi = \mathcal{Q}.\varphi$. The inner clause of C on literal $l \in C$, denoted by $\mathcal{I}(\Phi, C, l)$, is given by the clause $\{k \mid k \in C, k = \bar{l} \vee l \prec_{\Phi} k\}$.*

Definition 3.3.5 (Extended Universal Reduction (EUR)). *Let C be a clause and let $\Phi = \mathcal{Q}.\varphi$ be QBF. The clause $\mathcal{E}(\Phi, C, l)$ is the unique clause obtained by applying repeatedly the following extension rule until fixpoint:*

$$C := C \cup \mathcal{I}(\Phi, D, l) \text{ if exists } k \in C, D \in \varphi \text{ with } \bar{k} \in D, k \in E_{\Phi}, \text{ and } l \prec_{\Phi} k$$

Given a QBF $\Phi = \mathcal{Q}.\varphi \wedge \{E\}$ with a universal literal $l \in E$ such that $\bar{l} \notin \mathcal{E}(\Phi, E, l)$. Then, the removal of l from E is satisfiability preserving.

The notation of a QRAT-proof is an extension of the DRAT proof notation. The representation of clause additions and clause deletions are directly inherited from DRAT proofs. Additionally the QRAT-proof notation introduces a new prefix "u" for describing clause modifications.

Again, each line of a QRAT-proof $P = \{L_0, L_1, \dots, L_{|P|}\}$ for a QBF $\Phi = \mathcal{Q}.\varphi$ modifies the matrix φ_P^i of a QBF and generates a new matrix φ_P^{i+1} . As in DRAT, $L_i = \{p_i, C_i\}$ is the representation of the i^{th} line of the QRAT-proof P with p_i being the prefix of the line and C_i being the corresponding clause of the line. The first literal in C_i is denoted as l_i .

$$\varphi_P^i = \begin{cases} \varphi & \text{if } i = 0 \\ \varphi_P^{i-1} \setminus \{C_i\} & \text{if } p_i = \text{"d"} \\ \varphi_P^{i-1} \setminus \{C_i\} \cup \{C_i \setminus \{l_i\}\} & \text{if } p_i = \text{"u"} \\ \varphi_P^{i-1} \cup \{C_i\} & \text{otherwise} \end{cases}$$

For a QRAT-proof P to be a valid satisfaction proof for a QBF $\Phi = \mathcal{Q}.\varphi$ the following properties must hold:

1. Lines $L_i = \{p_i, C_i\} \in P$ with $p_i = \text{"d"}$ (clause deletion) C_i must have QRAT property on l_i with respect to φ_P^i . If l_i is universally quantified $ALA(\varphi_P^i, C_i)$ must be a tautology.

2. The QRAT-proof must empty the matrix ($\varphi_P^{|P|} = \emptyset$).

The following properties must hold for a QRAT-proof P to be a refutation proof for a QBF $\Phi = Q.\varphi$:

1. The clauses C_i of addition lines $L_i = \{p_i, C_i\} \in P$ must have QRAT property on l_i with respect to φ_P^{i-1} . If l_i is universally quantified $ALA(\varphi_P^{i-1}, C_i)$ must be a tautology.
2. Lines $L_i = \{p_i, C_i\} \in P$ with $p_i = "u"$ (clause modification) l_i must be universally quantified. Additionally either C_i must have QRAT property on l_i with respect to φ_P^{i-1} or l_i can be removed using EUR.
3. The last line of the proof P must be an addition of the empty clause.

p cnf 3 3	-1 -2 0	p cnf 3 3	-2 0
a 1 0	d 3 -1 0	a 1 0	d -2 -3 0
e 2 3 0	d -3 -2 0	e 2 3 0	1 0
1 2 0	d -2 -1 0	1 2 0	u 1 0
-1 3 0	d 2 1 0	1 3 0	0
-2 -3 0		-2 -3 0	
(a) TRUE QBF with QRAT proof		(b) FALSE QBF with QRAT refutation	

Figure 3.4: QRAT Examples for QBFs in QDIMACS notation

Pedant - An Expansion-Based DQBF Solver

The solver in which we implemented the generation of unsatisfiability certificates for QBFs is PEDANT [RSS21]. PEDANT is a DQBF-solver which is a solver of a further generalization of QBFs. Dependency Quantified Boolean Formulas (DQBF) is a further generalization of QBF and allows the explicit specification of the *dependency sets* of existential variables. This means in a DQBF it is possible to indicate, for each existential variable, on which universal variables it depends. In a QBF the dependency set of each existential variable is implicitly determined by the prefix order. An existential variable e depends on all preceding universal variables $u \prec e$.

Due to DQBF being a generalization of QBF, PEDANT can also be used as a decision procedure for plain QBFs. We put the focus on implementing the QRAT proof-format for unsatisfiable QBFs in PEDANT. This is motivated by the fact, that the certification of unsatisfiable instances is currently a missing feature in PEDANT. There is already a mechanism in PEDANT which allows an easy verification of satisfiable QBFs.

4.1 Decision Procedure

In order to understand the proposed certificate structure for unsatisfiable QBFs, we need some basic understanding of the decision procedure of PEDANT itself. Generally speaking, PEDANT utilizes a counter-example guided inductive synthesis (CEGIS) approach [JS17] by computing and incrementally refining candidate Skolem functions. The algorithm starts off by trying to find definitions for existential variables with respect to their dependency set. The found definitions can be used as candidate Skolem functions for the corresponding variables, due to either the definitions being proper Skolem functions or the input formula being unsatisfiable. Therefore, the algorithm does not change/refine

the initially found functions. The incremental refinement is only applied on the candidate functions of undefined variables. Initially the algorithm sets the candidate Skolem function of undefined variables to some default value.

In order to do so, the algorithm checks in each iteration, whether the current candidate model is indeed a model with respect to the current assignment of the *arbiter variables* (which will be discussed later). If the current candidate model is indeed a model the algorithm has determined that the input formula is **TRUE**. In case the current candidate model is not valid, the algorithm can obtain a counterexample σ . The algorithm then proceeds to reduce the counterexample using assumption-based SAT solving [ES04]. This means the algorithm checks the satisfiability of the matrix under σ , which necessarily leads the SAT solver to output **UNSAT**. Nevertheless, SAT solvers are very good at finding compact explanations of conflicts. Core extraction yields a set of failed assumptions, which will be used as a reduced counterexample, denoted by $\hat{\sigma}$.

Algorithm 4.1: PEDANT - An Expansion-Based DQBF Solver

```

1 Function SOLVE( $\Phi$ )
2    $model \leftarrow \text{COMPUTEDEFINITIONS}()$ 
3    $\text{INITIALIZEDDEFAULTS}(model)$ 
4    $arbiterFormula, arbiterAssignment \leftarrow \emptyset$ 
5   loop
6     if CHECKMODEL( $model, arbiterAssignment$ ) then
7       return TRUE
8      $\sigma \leftarrow \text{GETCOUNTEREXAMPLE}(model, arbiterAssignment)$ 
9      $\hat{\sigma} \leftarrow \text{GETCORE}(model, \sigma)$ 
10    if HASFORCINGCLAUSE( $\hat{\sigma}$ ) then
11      ADDFORCINGCLAUSE( $model, \hat{\sigma}$ )
12     $failedArbiters \leftarrow \hat{\sigma}|_A$ 
13    for  $l \in \hat{\sigma}|_E$  do
14       $e^{\sigma|_{D_e}} \leftarrow \text{GETARBITER}(model, l, \sigma|_{D(\text{var}(l))})$ 
15       $failedArbiters \leftarrow failedArbiters \cup e^{\sigma|_{D_e}}$ 
16     $blockingClause \leftarrow \text{CLAUSIFY}(failedArbiters)$ 
17     $arbiterFormula \leftarrow arbiterFormula \cup blockingClause$ 
18    if  $arbiterFormula$  is satisfiable then
19       $arbiterAssignment \leftarrow \text{GETMODEL}(arbiterFormula)$ 
20    else
21      return FALSE

```

In order to resolve the reduced counterexample, PEDANT deploys two repairing techniques. Those techniques depend on the number of failed existential assumptions in the reduced counterexample $\hat{\sigma}$.

If the reduced counterexample only contains a single existential variable e , the assignment of e needs to be flipped under the universal assignment of its dependency set $\hat{\sigma}|_{D_e}$. To

avoid the current counterexample in subsequent iterations, the algorithm is adding a *forcing clause* to the current candidate model. The *forcing clause* is responsible to rule out this exact assignment of variables leading to this conflict. A forcing clause represents an implication that asserts that under the projection of $\hat{\sigma}$ to D_e the variable e must be assigned to $\neg\hat{\sigma}(e)$.

If the counterexample contains multiple existential variables, the algorithm cannot know a priori which existential variables in the counterexample need to be assigned differently. However, at least one of them needs to be assigned differently. The second repair mechanism introduces multiple clauses in order to get control over the assignments of these existential variables under certain assignments of the arbiter variables. First, PEDANT creates for each existential variable e in $\hat{\sigma}$ a new auxiliary variable $e^{\sigma|D_e}$. This new arbiter variable $e^{\sigma|D_e}$ is then used in new *arbiter clauses*, which establish the connection between the arbiter variable $e^{\sigma|D_e}$ and the associated existential variable e . The introduced arbiter clauses linking arbiter variables and existential variables are $(e^{\sigma|D_e} \vee \neg\sigma|D_e \vee \neg e)$ and $(\neg e^{\sigma|D_e} \vee \neg\sigma|D_e \vee e)$. These clauses make sure that, under the specific assignment $\sigma|D_e$ the arbiter variable and the associated existential variable are assigned to the same truth value. This means, by fixing the assignment for the arbiter variable we can thus fix the assignment of the associated existential variable under the assignment $\sigma|D_e$.

To avoid running into the same conflict again in subsequent iterations, PEDANT additionally adds a clause to the *arbiter formula*. This *blocking clause* contains not only the negation of the assignment of all failed arbiter literals in $\hat{\sigma}$ but also includes for each existential literal in $\hat{\sigma}$ the corresponding arbiter variable in opposite polarity ($e^{\sigma|D_e}$ if $\neg e \in \hat{\sigma}$, respectively $\neg e^{\sigma|D_e}$ if $e \in \hat{\sigma}$).

In the end the unsatisfiability of the input (D)QBF is confirmed by the unsatisfiability of the arbiter formula. As long as the arbiter formula is satisfiable, the algorithm finds an assignment of arbiter variables that deals with all encountered counterexamples. In case the arbiter formula is unsatisfiable, all possible assignments of arbiter variables entail a counterexample thus resulting in an unsatisfiable DQBF.

4.1.1 Connection to $\forall\text{Exp}+\text{Res}$

The created arbiter variables which are linked to existential variables under a specific dependency assignment pretty much follow the concept of annotated variables in the $\forall\text{Exp}+\text{Res}$ proof system. Additionally, the arbiter formula consists of only clauses containing just arbiter variables. In case of an unsatisfiable (D)QBF the corresponding final arbiter formula has to be unsatisfiable. As a result it must be possible to find a refutation using propositional resolution confirming the unsatisfiability of the final unsatisfiable arbiter formula. A resolution refutation over clauses containing arbiter variables (annotated $\forall\text{Exp}+\text{Res}$ variables) is by definition a $\forall\text{Exp}+\text{Res}$ refutation. Although the clauses in the arbiter formula are not directly derived from matrix clauses, they are logically connected to them. Together with small modification this allows us to apply the

simulation of $\forall\text{Exp}+\text{Res}$ using QRAT [KS19]. This enables us to generate valid QRAT certificates for unsatisfiable QBFs which can be verified using QRAT-TRIM.

4.2 Example

In this section we are going to demonstrate the functionality of PEDANT by working out the crucial steps and decisions taken when solving a specific DQBF. We will have a close look at PEDANT solving the following unsatisfiable DQBF:

$$\Phi = \forall u_1 \forall u_2 \exists e_1(u_1) \exists e_2(u_2) . (\neg u_1 \vee e_1) \wedge (\neg u_2 \vee e_2) \wedge (\neg u_1 \vee \neg u_2 \vee \neg e_1 \vee \neg e_2)$$

PEDANT starts by initializing a default candidate model based on the initially found definitions for existential variables. Because PEDANT is not able to find definitions for the existential variables e_1 and e_2 , it uses default functions for their initial candidate model. In this case the candidate model would be:

$$F(\sigma) = \{ f_{e_1}(u_1) = \text{FALSE}, \quad f_{e_2}(u_2) = \text{FALSE} \}$$

In the first iteration of PEDANT's main loop the default candidate model gets rejected. As a result a SAT solver is tasked to obtain a compact explanation for the default candidate model not being a model for Φ . Lets assume the first obtained counterexample for this QBF would be $\hat{\sigma}_1 = \{u_1, \neg e_1\}$. This causes the introduction of the forcing clause $(\neg u_1 \vee e_1)$ because $\hat{\sigma}_1$ does only contain a single existentially quantified variable.

Additionally PEDANT adds arbiter and blocking clauses. First, a new arbiter variable $e_1^{u_1}$ gets created and linked to the existential variable e_1 with following clauses:

1. $(e_1^{u_1} \vee \neg u_1 \vee \neg e_1)$
2. $(\neg e_1^{u_1} \vee \neg u_1 \vee e_1)$

These linking clauses ensure the consistent assignment of e_1 and $e_1^{u_1}$ under the assignment $\sigma(u_1) = \text{TRUE}$.

Furthermore, PEDANT adds the clause $(e_1^{u_1})$ to the arbiter formula to avoid running again into the same conflict derived in this iteration. In this case the addition of this clause to the arbiter formula results in mapping $e_1^{u_1}$ to TRUE in the arbiter assignment. Because the arbiter formula is satisfiable a new candidate model is calculated like:

$$F(\sigma) = \{ f_{e_1}(u_1) = u_1, \quad f_{e_2}(u_2) = \text{FALSE} \}$$

Assume the next counterexample derived by the SAT solver is $\hat{\sigma}_2 = \{u_2, \neg e_2\}$. Again, because $\hat{\sigma}_2$ does only cover one existential variable, PEDANT first adds the forcing clause $(\neg u_2 \vee e_2)$ before applying the second conflict resolution strategy. A new arbiter variable $e_2^{u_2}$ gets created and linked to e_2 under the assignment $\sigma(u_2) = \text{TRUE}$ with the clauses:

1. $(e_2^{u_2} \vee \neg u_2 \vee \neg e_2)$
2. $(\neg e_2^{u_2} \vee \neg u_2 \vee e_2)$

To complete this step, PEDANT adds $(e_2^{u_2})$ to the arbiter formula and the arbiter assignment needs to be $\sigma = \{e_1^{u_1}, e_2^{u_2}\}$ to satisfy it. As such, the next candidate model is:

$$F(\sigma) = \{ f_{e_1}(u_1) = u_1, \quad f_{e_2}(u_2) = u_2 \}$$

Finally we reached the last iteration. Normally PEDANT would obtain the counterexample $\hat{\sigma}_3 = \{u_1, u_2\}$ which would lead to the addition of the empty clause to the arbiter formula (which makes it trivially unsatisfiable).

$$\frac{\frac{\neg e_1^{u_1} \vee \neg e_2^{u_2} \quad e_1^{u_1}}{\neg e_2^{u_2}} \text{ (RES)} \quad e_2^{u_2}}{\emptyset} \text{ (RES)}$$

Figure 4.1: Refutation of Arbiter Formula

For demonstration purposes we continue the solving process with the counterexample $\hat{\sigma}_3 = \{u_1, u_2, e_1, e_2\}$. In this case no forcing clause can be added and PEDANT directly continues with the addition of arbiter and blocking clauses. Because there are already existing arbiter variables which can be reused in this case PEDANT just adds the blocking clause $(\neg e_1^{u_1} \vee \neg e_2^{u_2})$ to the arbiter formula. As a consequence the arbiter formula gets unsatisfiable as shown in the resolution proof in Figure 4.1.

The unsatisfiability of the arbiter formula indicates that there is no assignment of arbiter variables and further no candidate model without entailing a counterexample. With this, PEDANT reports that the input formula is unsatisfiable and terminates.

QRAT Certificates from Expansion Proofs

The certificate structure which is proposed in this chapter closely follows the polynomial time simulation of $\forall\text{Exp}+\text{Res}$ by QRAT [KS19].

The structure of the proposed QRAT certificates for unsatisfiable QBFs is composed of multiple blocks. To be more precise, there are three major consecutive blocks, each block containing relevant information for the refutation of an unsatisfiable QBF. The first block includes information about applied pre-processing steps followed by a block depicting the definitions of forcing and arbiter clauses. In the last block we are applying some post-processing steps in order to create a valid refutation proof.

For illustration purposes of describing the certificate structure we are using the QBF in Figure 5.1 and the QRAT refutation in Figure 5.2 as an example. The latter is also providing a color-coded representation of the particular blocks.

```
p cnf 5 5
e  2  5  0
a  4  1  0
e  3  0
  1  2  5  0
  1 -3  0
  2  3  0
 -2 -3  0
  3  4  0
```

Figure 5.1: Running example in QDIMACS encoding

					...
	u	1	2	5	0
		2	5	0	
1:		1	2	0	
2:		2	1	0	
3:	u	1	2	0	
4:		2	0		
5:		-3	0		
6:		4	0		
7:		102	-2	0	
8:		-102	2	0	
9:		102	-4	1	0
10:		4	0		
11:		4	1	0	
					...
		d	3	2	0
		d	-3	1	0
		d	3	4	0
		d	-2	-3	0
		d	2	5	0
		d	-102	2	0
		d	-3	0	
		d	-2	102	0
		d	2	0	
		d	2	5	0
		d	4	0	
	u	1	102	-4	0
		102	-4	0	
	u	1	4	0	
		4	0		
	u	-4	102	0	
		102	0		
	u	4	0		
		0			

Figure 5.2: QRAT Refutation of QBF in Figure 5.1, QRAT-Blocks are highlighted with colors, red=pre-processing, orange=definitions, green=post-processing, yellow=deletions, blue=reductions

The following sections will give a more detailed explanation of each block of the proposed QRAT certificate.

5.1 Pre-Processing

This block is responsible for the representation of all applied pre-processing techniques upon a QBF before solving it. Generally speaking, this section could contain any pre-processing technique which is representable using QRAT. However, since PEDANT is just applying Universal Reduction beforehand, this block in our generated certificates is a list of universal literal elimination statements.

Let $C = (l_u \vee l_1 \vee \dots \vee l_n)$ be a clause of a QBF Φ with l_u being a universal literal such that each existential literal $e \in C$ precedes l_u . Using Universal Reduction the literal l_u

can be removed from C without changing the truth value of Φ . This pre-processing technique is represented in our QRAT-certificate using the following two lines:

$$\begin{array}{cccccc} u & l_u & l_1 & \dots & l_n & 0 \\ & & l_1 & \dots & l_n & 0 \end{array}$$

We start with a clause modification line for eliminating the literal u out of clause C followed by an explicit clause addition line for adding the reduced clause $C' = C \setminus \{u\}$. The clause modification line is safe because the universal quantified literal u can be removed using Extended Universal Reduction.

In our QRAT Refutation Example, we do have such a Universal Reduction as part of the applied pre-processing steps. More specifically, PEDANT is removing the universal literal 1 from the clause $1 \vee 2 \vee 5$. The corresponding QRAT lines representing this clause modification are:

$$\begin{array}{cccccc} u & 1 & 2 & 5 & 0 \\ & 2 & 5 & 0 & & \end{array}$$

One can see that the explicit clause addition of the reduced clause is redundant. According to the QRAT definition, the reduced clause is already added to the matrix by the clause modification line. We added this explicit clause addition line, to make life easier in the post-processing block. These redundant lines are helping in keeping track of *unnecessary* clauses, which we are going to delete in the post-processing block. For a more detailed explanation, please have a look into Section 5.3.

5.2 Clause Definitions

The second block covers all important clause definitions. This block almost represents the complete solving process of PEDANT by containing almost all auxiliary clauses created by PEDANT. This includes the introduction of forcing clauses, as well as the handling of arbiter clauses and blocking clauses. At this stage the direct addition of pure blocking clauses in QRAT might not be valid QRAT additions. Instead we can only add blocking clauses extended with a complete universal assignment (further details in Section 5.2.4). In addition to that, this block also contains some reasoning about the soundness of the addition of those auxiliary clauses in the form of DRAT proofs.

For each counterexample found by PEDANT, this block contains a corresponding SAT-proof and the addition of either a forcing clause or some arbiter clauses. This block is covered in the orange highlighted lines in our QRAT example in Figure 5.2.

5.2.1 SAT-Proofs

An important part of the decision procedure of PEDANT is the iterative search for counterexamples using a SAT-solver. Ideally, such a counterexample is a small assignment of variables, such that the matrix is falsified. PEDANT is using this knowledge and is introducing new forcing, arbiter and blocking clauses based on those counterexamples. In order to verify the addition of those auxiliary clauses, we are using the DRAT proofs outputted by the SAT-solver, which serve as a proof for the counterexample itself.

During the solving process, for each counterexample PEDANT collects a corresponding proof in DRAT format which is the output of a SAT-solver. We include those DRAT proofs in our QRAT-certificate. Our QRAT Refutation Example in Figure 5.2 contains exactly three DRAT proofs in the orange highlighted block:

1. The first DRAT proof is just the first line.
2. The second DRAT proof are the lines 5 and 6.
3. The third DRAT proof is just line number 10.

For example, the counterexample derived by the SAT-solver and represented by the first DRAT proof is $\hat{\sigma} = \{-1, -2\}$. By looking at the matrix we can easily see, that this assignment combination is not able to satisfy all clauses and one of them needs to be assigned differently. Because there is only one existential variable in this counterexample, PEDANT repairs this conflict using a forcing clause (more details in Section 5.2.2).

We are aware that the RAT property of propositional formulas differs from the RAT property of QBFs. However, we observed, that the generated DRAT proofs are merely using the AT property in this use-case. Reverse Unit propagation does also apply in QRAT, therefore we can safely include the outputs of the SAT-solver into our QRAT certificate as a verification of the found counterexamples.

5.2.2 Forcing Clauses

If the reduced counterexample $\hat{\sigma}$ is just containing a single existential variable e , PEDANT adds a clause that forces e to the opposite polarity $\neg\hat{\sigma}(e)$ under the (partial) universal assignment in $\hat{\sigma}$. Lets l_{u_1}, \dots, l_{u_n} be literals representing the assignment of the universal variables contained in the reduced counterexample $\hat{\sigma}$. Further, lets l_e be the literal representation of the existential variable e with respect to $\hat{\sigma}$. The clause which PEDANT introduces is $F = (\neg l_e \vee \neg l_{u_1} \vee \dots \vee \neg l_{u_n})$. Additionally, PEDANT restricts the set of universal variables contained in F to the dependency set of e , by removing those variables using universal reduction.

In the solver, only the restricted clause is added. In the QRAT certificate we also need to represent the reduction process. In general, it may requires multiple reduction steps in the QRAT certificate. For illustration purposes, lets assume the variable l_{u_1} is not in

the dependency set of e . We therefore represent the addition of the forcing clause using the following few lines:

$$\begin{array}{cccccc} & -l_e & -l_{u_1} & -l_{u_2} & \dots & -l_{u_n} & 0 \\ \text{u} & -l_{u_1} & -l_e & -l_{u_2} & \dots & -l_{u_n} & 0 \\ & & -l_e & -l_{u_2} & \dots & -l_{u_n} & 0 \end{array}$$

First we initially add the unrestricted forcing clause. Because the counterexample is derived by the SAT solver and the corresponding DRAT proof is included in the QRAT certificate beforehand, the addition of the unrestricted forcing clause is sound due to having the AT property. To cover the actual added (restricted) forcing clause, we need to apply universal reduction step-by-step because the clause modification line in QRAT only supports the removal of a single universal literal at a time. Similar to the pre-processing section, for each universal reduction we are adding a clause modification and a clause addition line. The former is removing a universal literal from the clause F , the latter is helping to keep track of *unnecessary* clauses. The same argument as in the pre-processing section regarding the soundness applies here.

We can find the tracing of a forcing clause addition in the orange block of our QRAT Refutation Example. This addition is covered by the lines 2-4:

$$\begin{array}{ccc} & 2 & 1 & 0 \\ \text{u} & 1 & 2 & 0 \\ & 2 & 0 & \end{array}$$

The corresponding counterexample derived beforehand by the SAT-solver is $\hat{\sigma} = \{-1, -2\}$. In the context of QBF there must be a satisfying assignment of the formula with both -1 and 1 due to the universal quantification. In this case the polarity swap of the existential variable is possible which is forced by the added forcing clause.

5.2.3 Arbiter Clauses

The more interesting part of the clause definition block is covering the introduction of arbiter clauses. As we can see in the algorithm description of PEDANT, the unsatisfiability of the input formula is merely determined by the unsatisfiability of the arbiter formula. This formula consists of clauses containing only arbiter variables, which again are defined in the introduced arbiter clauses. Those clauses are only created in case the counterexamples contain more than a single existential variable.

In order to represent this mechanism using QRAT, we need to cover (a) the creation of new arbiter variables, (b) the linking of arbiter variables to corresponding existential variables using arbiter clauses and (c) the addition of clauses to the arbiter formula (described in Section [5.2.4](#)).

For each existential variable e in the reduced counterexample, PEDANT creates a (new) auxiliary variable $e^{\sigma|_{D_e}}$ and two arbiter clauses $A_1 = (e^{\sigma|_{D_e}} \vee \neg\sigma|_{D_e} \vee \neg e)$ and $A_2 = (\neg e^{\sigma|_{D_e}} \vee \neg\sigma|_{D_e} \vee e)$. This auxiliary variables represent the concept of annotated variables in the $\forall\text{Exp}+\text{Res}$ proof system. Because QRAT allows the addition of new variables, we can represent the addition of the new arbiter variable and the introduction of the arbiter clauses together in one step. The arbiter clauses, which contain the new arbiter variable, can just be added in QRAT using the following two QRAT addition lines:

$$\begin{array}{rrrr} e^{\sigma|_{D_e}} & \neg\sigma|_{D_e} & \neg e & 0 \\ -e^{\sigma|_{D_e}} & \neg\sigma|_{D_e} & e & 0 \end{array}$$

Here, $\neg\sigma|_{D_e}$ denotes a list of literals representing the flipped universal assignment of the dependency set of e . For illustration purposes, assume $\sigma|_{D_e}$ is $\{u_1, \neg u_2, u_3\}$. The resulting QRAT lines would be:

$$\begin{array}{rrrrr} e^{u_1, \neg u_2, u_3} & -u_1 & u_2 & -u_3 & \neg e & 0 \\ -e^{u_1, \neg u_2, u_3} & -u_1 & u_2 & -u_3 & e & 0 \end{array}$$

The addition of the first arbiter clause A_1 is sound, because this clause has the QRAT property on the newly introduced arbiter variable $e^{\sigma|_{D_e}}$. In general, every clause C with a new variable x can be safely added, because there is no other clause D to build the outer resolvent with. Therefore the clause C has the QRAT property on literal x . Also the second arbiter clause A_2 has the QRAT property on the newly introduced arbiter variable $e^{\sigma|_{D_e}}$. At the time of the addition of A_2 , the only clause to form the outer resolvent with, is the before added clause A_1 . The outer resolvent of A_2 with A_1 on literal $e^{\sigma|_{D_e}}$ is $(e \vee \neg e)$ which obviously is a tautology. Therefore A_2 has the QRAT property on literal $\neg e^{\sigma|_{D_e}}$.

This process is also covered by our QRAT Refutation example at line 7 and 8. In the example the arbiter variable $e^{\sigma|_{D_e}}$ is 102 and the existential variable e is 2. The corresponding QRAT line representing the addition of A_1 is **102 -2 0**, respectively **-102 2 0** for the addition of A_2 .

5.2.4 Blocking Clauses

The unsatisfiability of the input formula is solely determined by the unsatisfiability of the arbiter formula. After adding the arbiter clauses in the QRAT certificate, the addition of the blocking clause added to the arbiter formula is still missing. Lets denote the missing clause as B . At this point we can not directly add B in the QRAT certificate. All variables contained in the blocking clause B are arbiter variables. By default we can not automatically guarantee, that B does have the AT or QRAT property (on any literal) and therefore might result in an invalid QRAT addition line. Instead we need to add B

together with the negation of the current complete universal assignment $\sigma|_U$. We denote the clause which we are tracing as B_U .

$$e_1^{\sigma|_{D_{e_1}}} \quad \dots \quad e_k^{\sigma|_{D_{e_k}}} \quad -\sigma|_U \quad 0$$

A similar argument as the one for forcing clauses applies regarding the soundness of this clause addition (B_U has the AT property). We can derive a conflict using reverse unit propagation on the derived counterexample by the SAT solver and the arbiter clauses. RUP starts off by creating $\overline{B_U}$ a set of unit clauses that falsify all literals in B_U . Because B_U includes a negated complete universal assignment, $\overline{B_U}$ includes unit clauses representing this complete universal assignment $\sigma|_U$. Further, $\overline{B_U}$ also contains unit clauses falsifying all arbiter literals a_i in B_U . For each of those unit clauses, there exists an arbiter clause containing the arbiter variable in the other polarity as in the unit clause. Those unit clauses can be used to remove not only the universal assignment part $\neg\sigma|_{D_e}$ from all arbiter clauses, but also some arbiter variables from some arbiter clauses. This results in additional unit clauses containing existential literals $\neg e_i$. At this point we have enough unit clauses to falsify the reduced counterexample $\hat{\sigma}$, which obviously must lead to a conflict in RUP. This conflict tells us B_U has indeed the AT property and can be safely added in the QRAT certificate at this point.

Unfortunately, we cannot remove the universal literals of B_U at the moment. However we need to correctly represent the internal arbiter formula of PEDANT in the QRAT certificate and this requires the derivation of the actual blocking clause B by eliminating universal variables out of B_U . This can only be done in a post-processing block at the end of the QRAT certificate once PEDANT finished solving and decided the formula to be unsatisfiable [KS19].

The lines 9 and 11 (102 -4 1 0 and 4 1 0) in our QRAT Refutation Example represent the addition of two unreduced blocking clauses.

5.3 Post-Processing

Once PEDANT has found the unsatisfiability of a formula, our so far generated QRAT certificate is not a valid refutation proof yet. In order to create a valid refutation proof we need to do some post-processing. As seen in Section 5.2.4, the current QRAT certificate does not contain the same clauses as the solver has added to its arbiter formula. Instead we needed to trace those clauses with additional universal literals in order to get a sound QRAT addition line. The main purpose of this post-processing block is to resolve this discrepancy by removing these universal literals using Extended Universal Reduction.

5.3.1 Delete Block

Let P_1 be the so far generated QRAT certificate with $|P_1|$ lines. Further let ψ_1 represent the matrix after applying all QRAT operations in P_1 . Before we can apply Extended

Universal Reduction to those clauses, we need to eliminate all other clauses. As we can see in the algorithm description of PEDANT, we can see that the unsatisfiability is merely determined by the clauses added to the arbiter formula. This means we can delete all remaining clauses, except the relevant B_U clauses, of ψ_1 because we don't need them anymore. This also contains clauses of the original matrix. Note that QRAT allows the elimination of arbitrary clauses in a refutation proof. For each unnecessary clause, we append a corresponding QRAT deletion line to P_1 . The yellow block in Figure 5.2 covers the deletions of the no more needed clauses.

As mentioned in previous sections, we added a redundant QRAT addition line after each QRAT modification line. In order to find those unnecessary clauses, we only need to iterate over P_1 once, and find all added clauses, which are not already deleted. By adding those redundant QRAT addition lines we make sure to catch all unnecessary clauses. Otherwise we would need to compute ψ_1 explicitly.

5.3.2 Reduction Block

After appending the QRAT deletion lines of unnecessary clauses we get a new QRAT certificate, which we will denote as P_2 . At this point the matrix ψ_2 consists only of B_U clauses. In order to convert ψ_2 to the internal arbiter formula we need to remove all universal literals from the clauses in ψ_2 by applying Extended Universal Reduction. The removal of universal variables is done in reverse prefix order. Starting with the innermost universal variable, we iteratively remove all occurrences of a variable from all clauses before continuing with the elimination of the next universal variable. For each removal we append a corresponding QRAT modification line to P_2 . This process is represented in our QRAT Refutation example by the blue highlighted lines.

It remains to show that all the removal steps are valid Extended Universal Reduction steps. In order to show this we need the concept of *resolution paths* [VG11].

$$\underbrace{u \vee \dots \vee e_1}_{C_1} \bowtie \underbrace{\neg e_1 \vee \dots \vee e_2}_{C_2} \bowtie \underbrace{\neg e_2 \vee \dots \vee e_3}_{C_3} \bowtie \underbrace{\neg e_3 \vee \dots \vee e_4}_{C_4}$$

Figure 5.3: A resolution path from u to e_4

Definition 5.3.1 (Resolution Path [PSS19]). *Let $\Phi = QxQ.\varphi$ be a QBF in PCNF. A resolution path from l_1 to l_{2k} in Φ is a sequence $\pi = l_1 \dots l_{2k}$ of literals satisfying the following properties:*

1. *for all $i \in \{1, \dots, k\}$, there is a $C_i \in \varphi$ such that $l_{2i-1}, l_{2i} \in C_i$*
2. *for all $i \in \{1, \dots, k\}$, $\text{var}(l_{2i-1}) \neq \text{var}(l_{2i})$*
3. *for all $i \in \{1, \dots, k-1\}$, $\neg l_{2i} = l_{2i+1}$*

Using resolutions paths, the validity of the Extended Universal Reduction steps is a consequence of the following lemma:

Lemma 5.3.1 (Independence of Arbiter Literals [KS19]). *If $\Phi = \mathcal{Q}.\psi_2$ contains a resolution path from a universal literal u to an existential literal e , then e must be an arbiter literal such that the associated assignment $\sigma|_{D_e}$ falsifies u .*

Proof. Suppose there exists a resolution path C_1, \dots, C_n from u to e . We show by induction on the resolution path length n that e is an arbiter literal such that the associated assignment $\sigma|_{D_e}$ falsifies u .

BASE CASE ($n = 1$):

C_1 contains both u and e . Because ψ_2 contains only arbiter variables and universal variables, e needs to be an arbiter literal a . Furthermore, looking at the construction of the remaining clauses, the universal literals occur in the opposite polarity as contained in the corresponding counterexample σ . The universal literal u must precede e , otherwise e would be trivially independent of u . Hence, the dependency set D_e must contain $\text{var}(u)$. As a result a must have been created with respect to $\sigma|_{D_e}$ which clearly falsifies u .

INDUCTION STEP ($n - 1 \rightarrow n$):

Let C_1, \dots, C_n be a resolution path from u to e . We know that $e \in C_n$ and that C_1, \dots, C_{n-1} is a resolution path from u to some existential literal e_{n-1} such that $e_{n-1} \in C_{n-1}$ and $\neg e_{n-1} \in C_n$. By the induction hypothesis, e_{n-1} is an arbiter literal such that the associated assignment $\sigma|_{D_{e_{n-1}}}$ falsifies u . Because $\neg e_{n-1} \in C_n$, we know that the counterexample σ leading to the creation of C_n must also falsify u . As a result, e is also an arbiter literal such that the associated assignment $\sigma|_{D_e}$ falsifies u . \square

We get from Lemma 5.3.1, that whenever we eliminate a universal literal u from a clause $C \in \psi_2$, that each existential literal $e \in C$ with $u \prec e$ is independent of u (existential literals preceding u are trivially independent of u). Each existential literal $e \in C$ is an arbiter literal a such that the associated assignment $\sigma|_{D_e}$ falsifies u . Hence, there cannot exist a resolution path from $\neg u$ to both e or $\neg e$. If this would be the case Lemma 5.3.1 would tell us that e is an arbiter literal such that the associated assignment $\sigma|_{D_e}$ falsifies $\neg u$. Given the independence from e of u , we know the removal of u using Extended Universal Reduction is sound.

Since we are not introducing any new resolution paths by removing a universal literal, Lemma 5.3.1 does apply not only for the first removal of an universal literal, but also for the remaining elimination operations.

5.3.3 Arbiter Solver proof

Lets denote the QRAT certificate after removing all universal variables as P_3 . At this point ψ_3 is exactly the internal arbiter formula of PEDANT. This means, all universal

5. QRAT CERTIFICATES FROM EXPANSION PROOFS

variables are eliminated and there are only existential variables left. Due to having the same instance as the arbiter solver determining the unsatisfiability of the QBF Φ , we can again append the DRAT proof of this SAT call.

In our QRAT Refutation example, PEDANT is able to find the empty clause as a blocking clause. Therefore we do not need an additional DRAT proof for completing the refutation.

Implementation Details

In this chapter we are going to give a brief overview of the implementation of the unsatisfiability tracing in PEDANT proposed in Chapter 5. Further, we would like to mention some of the challenges we faced on the journey to valid QRAT certificates. We believe that the readers might be interested in some details of the developed solution and some might benefit from the insights and the knowledge we gathered along the way.

The core of our proposed tracing module is a newly created class responsible for the creation of valid QRAT certificate files. Obviously, there is more to that and the proof tracer class covers a larger area of responsibilities. In order to live up to its purpose and allow PEDANT the creation of QRAT refutations for QBFs, the remit of the proof tracer class can be summarized by the following four tasks:

1. Compliance of Tracing Configuration
2. Management of File-Pointers
3. Provision of Interface for Tracing-Operations
4. Application of Post-Processing on Trace to complete $\forall\text{Exp}+\text{Res}$ Simulation

First of all we made the unsatisfiability tracing configurable. Via program arguments of PEDANT it is possible to either activate or deactivate the QRAT tracing, as well as the specification of the storage location of the final QRAT certificate. In case of a deactivated QRAT tracing the proof tracer class gets still instantiated and used in the decision procedure of PEDANT. However, the deactivation results in skipping all tracing operations in the tracer class.

The creation of valid QRAT certificates requires the management of a total of 3 output files. Two out of the three files are auxiliary files utilized during the solving process. The

first temporary file is used for storing decisions and actions of PEDANT and conflicts derived by a SAT solvers. In order to allow in the post-processing the identification of the last conflict derivation by the arbiter solver, the corresponding resolution proof is written to the second temporary file. The third and last file is designed to contain the final valid QRAT certificate in the end. This last file allows the verification of unsatisfiable QBF instances using QRAT-TRIM.

Using the interface of the proof tracer class PEDANT is able to record the main events of solving a formula. It exposes functions for tracing (a) forcing clauses, (b) the definition of arbiter variables using arbiter clauses, as well as (c) the blocking clauses together with the negation of the corresponding complete universal assignment. In general these functions simply transform the integer array representation used in PEDANT for clauses and assignments to valid QRAT lines. For the purpose of tracing the modified blocking clauses we augmented the QRAT format by introducing new individual lines. Those specific lines should mark clauses which should be not deleted in the post-processing.

If PEDANT detects the unsatisfiability of a QBF it triggers the post-processing in the tracing module. The post-processing takes the two auxiliary files as a input and generates the final QRAT certificate. This step requires minor modifications on the trace together with additional lines to complete the intended $\forall\text{Exp}+\text{Res}$ simulation.

6.1 File - Handling

In total the proof tracer class takes care of 3 file pointers altogether. Initially only the auxiliary files are created and used until PEDANT determines the unsatisfiability of the input formula. In this case the proof tracer class makes sure the writing on the auxiliary files is completed before starting the post-processing and prepares to read in again the contents of them. From this point we solely write to the final certificate file which is created at the beginning of the post-processing.

Because PEDANT uses SAT solvers as sub-procedures and their conflict derivations are needed in the final QRAT certificates we need to coordinate the output of the SAT solvers to end up in the right auxiliary files. The SAT solvers used by PEDANT can as expected produce DRAT proofs. Luckily, the main SAT solver (CaDiCaL [BFFH20]) can also be parameterized with an already open file pointer and the DRAT proofs produced by CaDiCaL automatically end up in the corresponding file without further assistance of PEDANT.

We are using a second temporary file in order to identify the complete DRAT proof of the last conflict derived by the arbiter solver. In the final QRAT certificate the last conflict resolution is appended after the completion of the post-processing. Roughly speaking the final QRAT certificate starts with the slightly modified contents of the first auxiliary file followed by the additional QRAT lines generated in the post-processing. The QRAT certificate is completed with the resolution proof of the second auxiliary file. In general it would be possible to just use a single provisional file. For example the introduction of

special separation lines would allow to write all contents into one file and still would be able to identify the needed parts in the post-processing. However, this would require a more sophisticated parser for reading back in the content. We wanted to stay as close to the official QRAT format in order to make use of a simple QRAT parser.

Figure 6.1: Structure of final QRAT certificate with respect to temporary files

Although write operations on the first auxiliary file are triggered both from PEDANT and the external SAT solvers, the implementation of PEDANT does not allow a race condition regarding write order. PEDANT is not doing anything in parallel while the external SAT solvers are running. This means we can be sure the SAT solver's DRAT proof of the conflict derivation is already present in the file when PEDANT takes actions on the derived conflict.

6.2 Challenges

During the development phase of the unsatisfiability tracing we encountered some problems which challenged us to find clever solutions in order to create a valid certification workflow for unsatisfiable QBFs.

A problem we faced in the development of the proposed QRAT certificate structure concerns the elimination order of universal variables in the reduction block. As you can see in Section 5.3.2 we are eliminating the universal variables from all remaining clauses in reverse prefix order. By proving Lemma 5.3.1 we showed that all remaining existential (arbiter) literals are independent of all universal variables. As a result it theoretically should be possible to eliminate universal variable in any desired order. However, this led to some certificates which could not be verified using QRAT-TRIM. We discovered QRAT-TRIM uses a different definition of resolution paths. According to Definition 3.3.5 the resolution paths are created using just existential literals as connectors. QRAT-TRIM also uses universal literals as path connectors resulting in a larger dependency set. With regard of the Extended Universal Reduction definition, this dependency set can contain

false-positives such that QRAT-TRIM might not be able to confirm the soundness of certain Extended Universal Reduction statements. As a result, the verification of certain certificates might fail. By eliminating universal variables in reversed prefix order we can force QRAT-TRIM to just use existential variables for building resolution paths. Lets assume u is the rightmost universal variable in the prefix which we want to eliminate. It is not possible to find another universal variable u_1 such that $u \prec u_1$. Otherwise u_1 would be the rightmost universal variable which we would be eliminating first. As a consequence there are only existential literals e left such that $u \prec e$ and QRAT-TRIM can utilize for building resolution paths. By forcing QRAT-TRIM to use only existential literals for building resolution paths, our Lemma 5.3.1 applies and the corresponding reduction lines can be verified.

At the point we were able to create valid QRAT certificates which are verifiable using QRAT-TRIM we were puzzled of the running-time performance of some instances. In the evaluation of our solution we determine the impact of our tracing module on the performance of PEDANT.

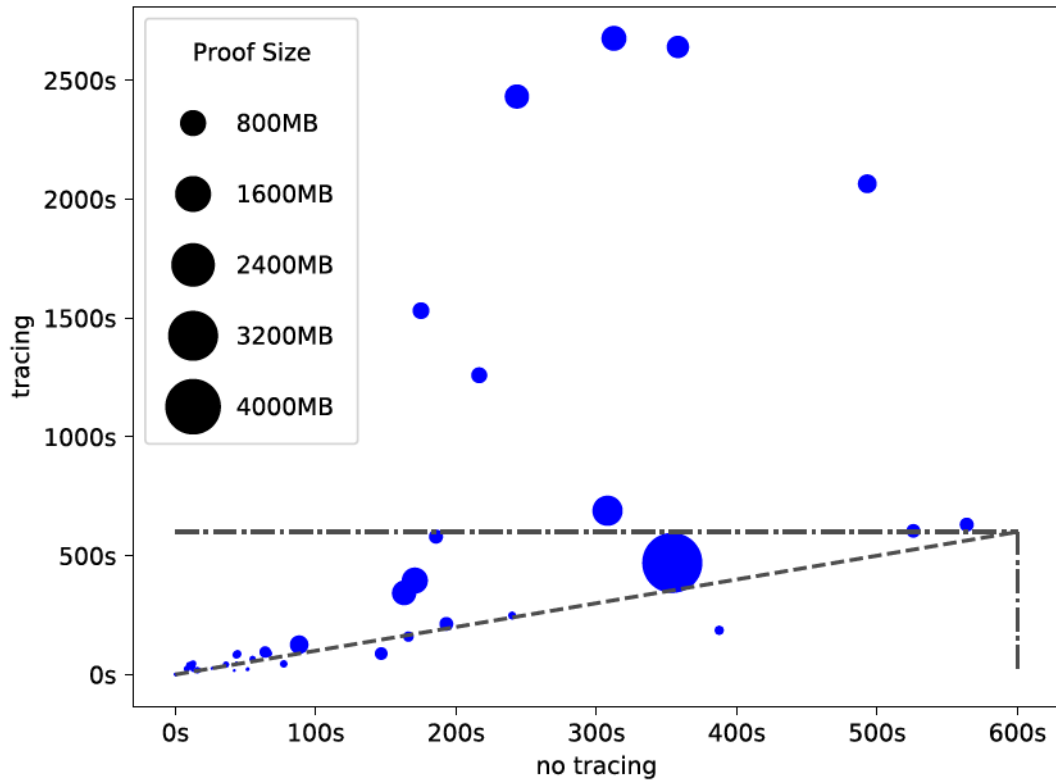


Figure 6.2: Former runtime impact of tracing module with respect to certificate size

We spotted some instances which were significantly slower with the unsatisfiability tracing enabled. Furthermore the size of the corresponding QRAT certificates generated by

the tracing module were quite big. We first figured this comes down to the masses of IO-operations we are generating in the post-processing. However, not long after we discovered an instance with a much bigger QRAT certificate which performed well with unsatisfiability tracing enabled. After profiling the tracing module we found the true cause of the performance loss. At one point in the post-processing we delete all clauses which are not needed anymore. In general at this point we would like to keep only blocking clauses. In other words we iterated over all so far traced clauses and checked whether this clause is a blocking clause. In order to be able to compare the integer array representations we sorted each clause producing most of the overhead. To get rid of this piece of code we already annotated blocking clauses in the first temporary file. Those annotations allow the effective deletion of unnecessary clauses. Instead of a quadratic worst-case runtime we were able to eliminate those clauses in linear time.

By looking at Figure 6.2 we find instances which are faster with the unsatisfiability tracing. This unintuitive behaviour should not be possible because pure logically we only can only add something to the runtime of PEDANT with the new tracing module. This discrepancy was a result of inconsistent solver runs on the cluster. It seems PEDANT is not totally deterministic on different machines. Through different implementations of hash-functions PEDANT can produce different runs on different machines. This is most likely the case in our example.

6.3 Future Improvements

We analyzed the generated QRAT certificates and observed PEDANT is occasionally able to derive the empty clause as a blocking clause. However, according to the $\forall\text{Exp}+\text{Res}$ simulation we are first adding the negation of the complete universal assignment as a clause before starting the post-processing. In this case the post-processing could be omitted and the empty clause can be derived in QRAT using Extended Universal Reduction on the corresponding clause. The premise for this to work is that QRAT-TRIM uses the resolution path definition as stated in Definition 3.3.5. By using only existential literals as path connectors it would be possible to directly resolve the empty blocking clause in QRAT without the complete post-processing. With the current implementation of resolution paths in QRAT-TRIM the complete post-processing is needed in order to create a by QRAT-TRIM verifiable certificate. By skipping the post-processing we would dramatically decrease the final QRAT certificate size, reduce the performance impact on PEDANT and facilitate a faster verification by QRAT-TRIM because of fewer QRAT lines.

Another point which can be addressed is the handling of universal assignments in the reduction of blocking clauses (see Section 5.3.2). Currently we store for each blocking clause the corresponding complete universal assignment in a dictionary in the proof tracer class. As it is implemented now this additional data is used to determine what literals should be removed from a clause. A similar approach to annotating blocking clauses in the temporary file could eliminate the additional storage dictionary. In order to work the

annotation not only need to identify the universal assignment, but also need to encode the prefix order. By doing so, the post-processing could theoretically be completely separated from PEDANT and realized as an individual program.

Experimental Evaluation

We extended PEDANT by adding an implementation of the certificate generation proposed in the previous section. To get an understanding of the overhead introduced and how the certificate generation affects the performance of PEDANT, we first evaluated PEDANT without tracing on a standard QBF benchmark set with a time limit of 600 seconds. The benchmark set we used for the evaluation were the instances from the PCNF tracks of QBFEVAL'19 and QBFEVAL'20. By doing this we got a set of 52 unsatisfiable instances which are solvable by PEDANT within the imposed time limit. Because our goal was to certify unsatisfiability in PEDANT we can obviously neglect solvable instance which are satisfiable and just focus on unsatisfiable instances. We used the performance results without tracing on those unsatisfiable instances as a baseline to compare to the performance of PEDANT with unsatisfiability tracing enabled. The experiments were performed on a cluster with AMD EPYC 7402 processors at 2.8 GHz running a 64-bit distribution of Linux.

The data presented in this section is a result of performing the experiment multiple times and taking the average. We let PEDANT solve each instance 5 times in each configuration (with/without tracing) and took the arithmetic mean.

In Figure 7.1 we can see a comparison of the measured runtimes of PEDANT on the previously defined set of 52 unsatisfiable instances. In general we can see that all instances lie in proximity of the diagonal indicating that in general the QRAT tracing does not have a huge performance impact. However we can see a trend, the longer an instance is taking to solve, the more the QRAT tracing impacts the performance. A deeper look into individual runs reveals that the majority of the produced overhead is added by the post-processing of the unsatisfiability tracing. As described in Section 5.3, the post-processing is responsible to trace the deletion of all unnecessary clauses and the reduction of B_U to the correct blocking clauses. If we take a look at the generated certificates, this block makes up the majority of the QRAT proof. This is because QRAT allows only the elimination of a single universal literal at a time. We see a blowup of certificate size

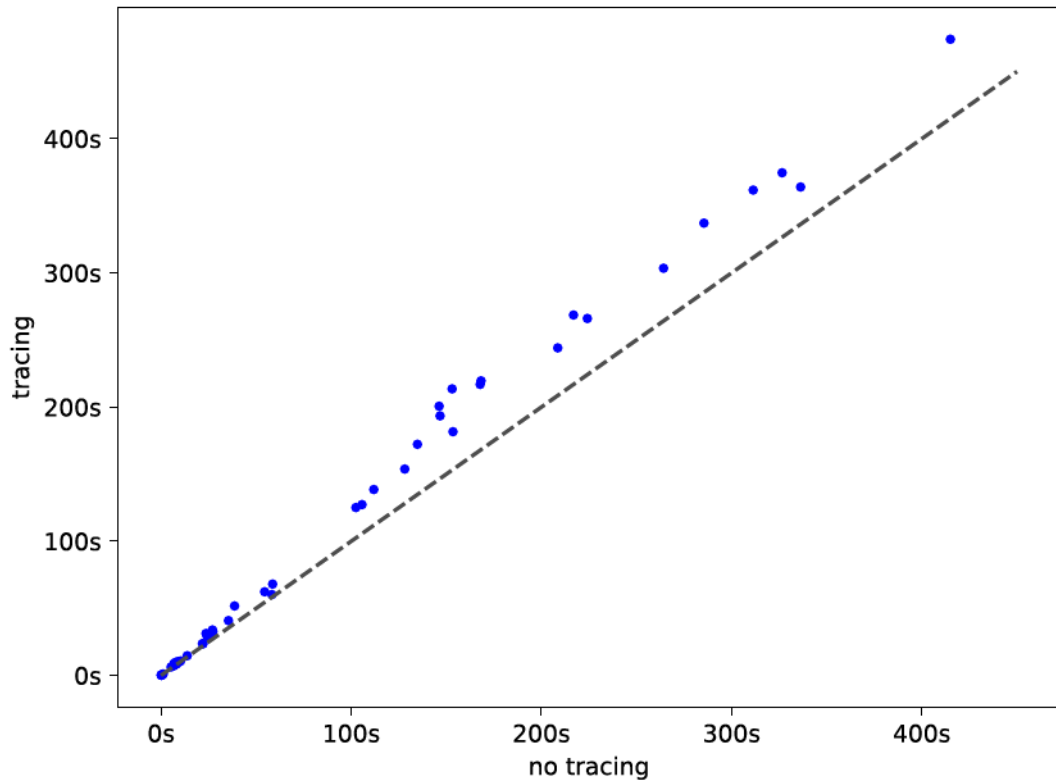


Table 7.1: Runtime Impact of Tracing per QBF Instance

Instance	Reference	Tracing	Delta
itc-b13-fixpoint-1	1.12 s	1.06 s	-0.06 s
small-synabs-fixpoint-3	0.65 s	0.66 s	0.01 s
trueque_query71_1344n	0.07 s	0.08 s	0.01 s
falsequ_query64_1344n	0.08 s	0.09 s	0.01 s
falsequ_query60_1344n	0.09 s	0.10 s	0.01 s
falsequ_query71_1344n	0.11 s	0.15 s	0.04 s
trueque_query60_1344n	0.13 s	0.19 s	0.06 s
trueque_query64_1344n	0.18 s	0.26 s	0.08 s
s00838_PR_6_90	8.22 s	8.64 s	0.42 s
s01423_PR_4_75	7.82 s	8.26 s	0.44 s
s01423_PR_4_90	6.61 s	7.08 s	0.47 s
itc-b13-fixpoint-2	10.22 s	10.70 s	0.48 s
incrementer-enc06-nonuniform-depth-5	9.46 s	10.07 s	0.61 s
cache-coherence-3-fixpoint-1	13.72 s	14.65 s	0.93 s
axquery_query42_1344n	5.22 s	6.24 s	1.02 s
small-synabs-fixpoint-10	7.94 s	9.41 s	1.47 s
ethernet-fixpoint-1	22.05 s	23.54 s	1.49 s
axquery_query71_1344n	8.87 s	10.37 s	1.50 s
exquery_query42_1344n	6.93 s	8.54 s	1.61 s
nxquery_query71_1344n	6.68 s	8.52 s	1.84 s
s05378_PR_4_90	21.72 s	23.69 s	1.97 s
cache-coherence-3-fixpoint-2	58.26 s	60.25 s	1.99 s
exquery_query64_1344n	7.50 s	9.61 s	2.11 s
cache-coherence-2-fixpoint-2	26.80 s	31.00 s	4.20 s
exquery_query71_1344n	24.27 s	28.90 s	4.63 s
nxquery_query42_1344n	25.66 s	30.33 s	4.67 s
s09234_PR_9_90	26.79 s	31.56 s	4.77 s
stmt19_83_412	35.43 s	40.84 s	5.41 s
stmt41_160_235	27.24 s	32.77 s	5.53 s
nxquery_query64_1344n	27.00 s	33.84 s	6.84 s
neclftp4001	23.61 s	31.25 s	7.64 s
stmt21_181_369	54.45 s	62.31 s	7.86 s
stmt29_226_376	58.64 s	68.04 s	9.40 s
small-pipeline-fixpoint-1	38.63 s	51.75 s	13.12 s
stmt21_310_360	105.65 s	127.27 s	21.62 s
arbiter-05-comp-error01-qbf-hardness-depth-8	102.43 s	125.11 s	22.68 s
ntrivil_query71_1344n	128.12 s	153.80 s	25.68 s
stmt19_313_412	111.85 s	138.55 s	26.70 s
klieber2017q-084-21-t1	336.28 s	364.03 s	27.75 s
trivial_query71_1344n	153.49 s	181.62 s	28.13 s

Continuation of Table 7.1			
Instance	Reference	Tracing	Delta
trivial_query60_1344n	208.57 s	244.10 s	35.53 s
k_branch_p-10	134.80 s	172.24 s	37.44 s
ntrivil_query42_1344n	264.20 s	303.48 s	39.28 s
small-pipeline-fixpoint-2	224.15 s	266.03 s	41.88 s
trivial_query64_1344n	146.69 s	193.44 s	46.75 s
k_branch_p-12	326.56 s	374.64 s	48.08 s
ntrivil_query64_1344n	167.78 s	216.90 s	49.12 s
gttt_2_1_001020_4x4_torus_w	311.34 s	361.74 s	50.40 s
stmt39_285_335	168.24 s	219.55 s	51.31 s
gttt_2_2_000111_4x4_torus_w	285.45 s	337.12 s	51.67 s
k_branch_p-11	216.85 s	268.60 s	51.75 s
stmt32_329_378	146.21 s	200.63 s	54.42 s
arbiter-06-comp-error01-qbf-hardness-depth-11	415.04 s	474.15 s	59.11 s
stmt52_295_394	153.03 s	213.55 s	60.52 s
End of Table 7.1			

Furthermore, we are interested in the time taken in the verification process of those instances we collected and solved previously. The verification tool we used is QRAT-TRIM [HSB14a]. For the verification processes we used a time limit of 1800 seconds.

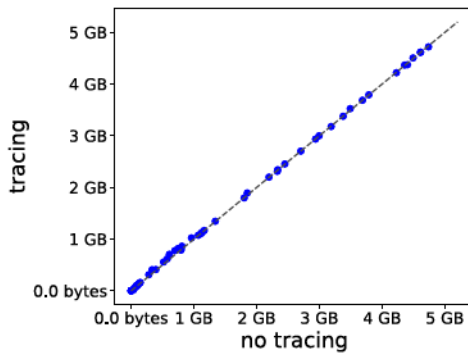


Figure 7.2: Memory Utilization

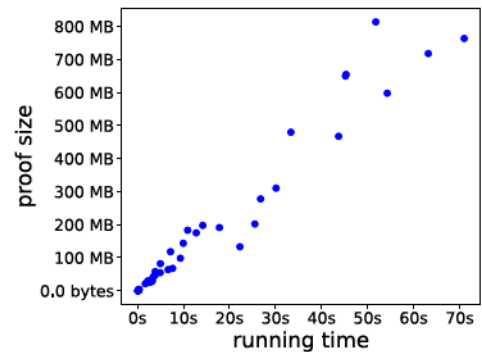


Figure 7.3: Certificate size and verification time

All except two of our generated certificates could be verified on the cluster within the imposed time limit. However, we managed to verify those two instances by manually increasing the time limit of QRAT-TRIM on a desktop machine. We increased the time limit to 9000 seconds and verified those instances in 4057 and 2028 seconds. With this, all instances could be verified and we can conclude that we are indeed generating valid QRAT certificates which certify the unsatisfiability of an input QBF.

In Figure 7.3 we can see the correlation of the certificate size and the amount of time

needed to verify the corresponding certificate. The larger a certificate is, the longer QRAT-TRIM is running in order to verify it.

Evaluation of QRAT Tracing on DQBF Instances

With the future goal of certifying unsatisfiable DQBF instances in mind, we analysed the impact of the in PEDANT implemented QBF unsatisfiability tracing on DQBF instances. In other words we are interested how the implemented unsatisfiability tracing performs on DQBF instances as well as when solving unsatisfiable QBF instances. We performed this experiment under the assumption that a future unsatisfiability tracing for DQBFs will not substantially differ from the implemented unsatisfiability tracing for QBFs. In fact, we believe this is just a matter of switching the underlying proof-format from QRAT to its corresponding generalization for DQBF [Bli20].

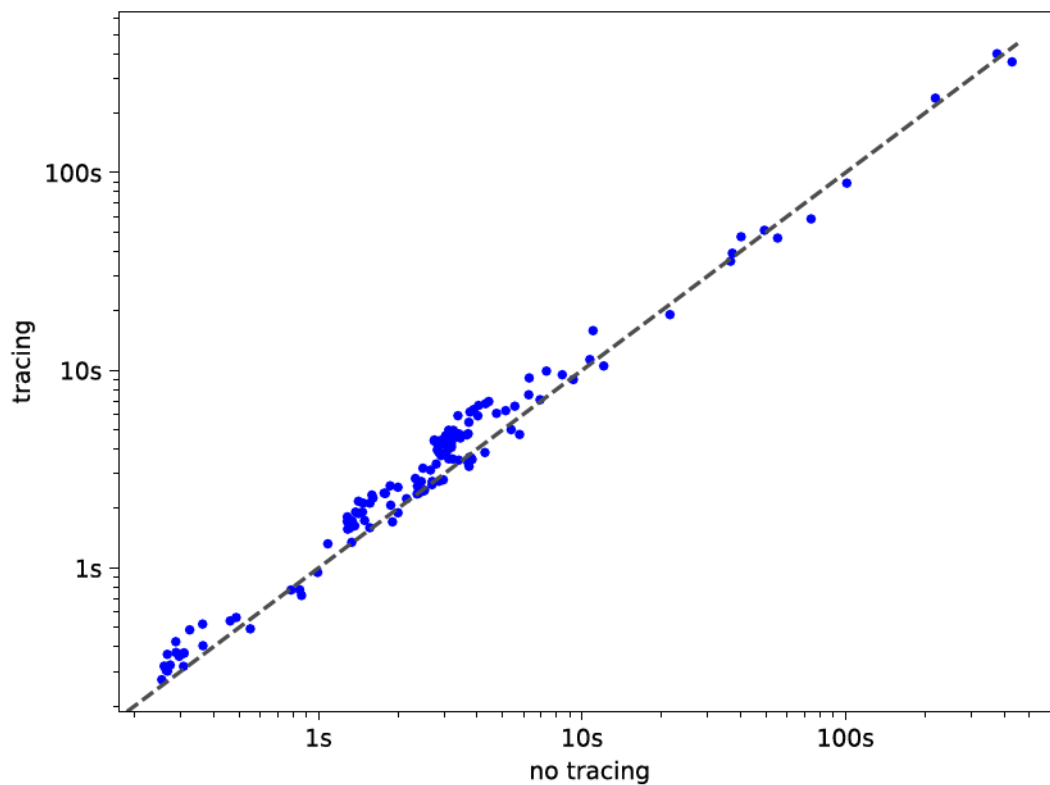


Figure 7.4: Runtime impact of proof tracing on DQBF Instances in Log Scale

In this experiment we used the same technique as before. We first evaluated PEDANT on DQBF instances from the DQBF track of QBFEVAL'20 with a time limit of 600 seconds to get a performance baseline to compare against. This yielded 145 unsatisfiable instances which were solvable by PEDANT within the imposed time limit. For these unsatisfiable instances we are interested in the runtime impact of the unsatisfiability tracing. Similar

to the QBF experiment, we performed the experiment on DQBF 5 times and took the arithmetic mean of the measurements.

In Figure 7.4 we can see the direct comparison of the measured runtimes of the runs with and without unsatisfiability tracing. Table 7.2 again shows the mean runtimes data for each of the 145 instances.

In principle we got similar results as in the QBF experiment. All instances lie in close proximity of the diagonal. This means the unsatisfiability tracing also does not significantly deteriorate the performance of PEDANT when solving DQBF instances. With this we feel confident that certifying unsatisfiable DQBFs in PEDANT can be done at a reasonably small overhead.

Table 7.2: Runtime Impact of Tracing per DQBF Instance

Instance	Reference	Tracing	Delta
crn_11_99_u.cnf	428.92 s	363.55 s	-65.37 s
tentrup_amba_decomposed_decode_environment_16	74.05 s	58.36 s	-15.69 s
tentrup_load_balancer_system_4	101.19 s	88.54 s	-12.65 s
tentrup_lilydemo19_environment_4	55.28 s	46.69 s	-8.59 s
marg3x3add8ch.shuffled-as.sat03-1448.cnf	21.58 s	19.14 s	-2.44 s
marg3x3add8.shuffled-as.sat03-1449.cnf	12.09 s	10.53 s	-1.56 s
scholl_term1.blif_0.20_1.00_3_2_henkin	5.80 s	4.74 s	-1.06 s
tentrup_lt12dba13_environment_8	36.65 s	35.65 s	-1.00 s
scholl_term1.blif_0.10_1.00_5_1_henkin	3.73 s	3.27 s	-0.46 s
battleship-6-9-unsat.cnf	4.28 s	3.84 s	-0.44 s
scholl_comp.blif_0.20_0.20_0_3_henkin	5.39 s	5.01 s	-0.38 s
scholl_term1.blif_0.60_1.00_5_1_henkin	3.70 s	3.41 s	-0.29 s
scholl_C499.blif_0.60_1.00_1_2_henkin	9.27 s	8.98 s	-0.29 s
scholl_term1.blif_0.10_1.00_5_3_henkin	3.83 s	3.54 s	-0.29 s
scholl_C499.blif_0.20_1.00_3_3_henkin	1.91 s	1.71 s	-0.20 s
scholl_term1.blif_0.50_1.00_9_2_henkin	2.98 s	2.79 s	-0.19 s
scholl_term1.blif_0.20_1.00_5_3_henkin	3.79 s	3.61 s	-0.18 s
scholl_comp.blif_0.10_0.10_0_2_henkin	0.86 s	0.73 s	-0.13 s
scholl_C432.blif_0.40_1.00_1_2_henkin	2.86 s	2.75 s	-0.11 s
scholl_term1.blif_0.50_1.00_5_3_henkin	3.74 s	3.64 s	-0.10 s
scholl_term1.blif_0.60_1.00_9_1_henkin	2.00 s	1.90 s	-0.10 s
scholl_C432.blif_0.50_1.00_3_3_henkin	0.85 s	0.78 s	-0.07 s
scholl_comp.blif_0.10_0.10_0_3_henkin	0.55 s	0.49 s	-0.06 s
scholl_C499.blif_0.10_1.00_4_2_henkin	2.70 s	2.64 s	-0.06 s
scholl_term1.blif_0.60_1.00_3_1_henkin	2.53 s	2.48 s	-0.05 s
scholl_term1.blif_0.10_1.00_5_2_henkin	2.49 s	2.45 s	-0.04 s
battleship-5-8-unsat.cnf	0.99 s	0.95 s	-0.04 s
scholl_C432.blif_0.10_1.00_7_3_henkin	0.79 s	0.77 s	-0.02 s
scholl_term1.blif_0.60_1.00_3_3_henkin	2.39 s	2.39 s	0.00 s

Continuation of Table 7.2			
Instance	Reference	Tracing	Delta
scholl_term1.blif_0.20_1.00_3_1_henkin	2.37 s	2.37 s	0.00 s
scholl_z4ml.blif_0.30_0.10_2_1_henkin	0.07 s	0.08 s	0.01 s
scholl_z4ml.blif_0.50_1.00_7_3_henkin	0.06 s	0.07 s	0.01 s
scholl_comp.blif_0.10_1.00_2_3_henkin	0.31 s	0.32 s	0.01 s
bloem_unrealizable	0.02 s	0.03 s	0.01 s
scholl_C499.blif_0.10_1.00_7_1_henkin	1.34 s	1.35 s	0.01 s
scholl_z4ml.blif_0.10_0.20_1_2_henkin	0.03 s	0.05 s	0.02 s
bloem_sh_u_o	0.03 s	0.05 s	0.02 s
scholl_z4ml.blif_0.10_1.00_1_3_henkin	0.03 s	0.05 s	0.02 s
scholl_z4ml.blif_0.10_0.50_2_3_henkin	0.05 s	0.06 s	0.01 s
scholl_z4ml.blif_0.10_0.10_2_3_henkin	0.05 s	0.07 s	0.02 s
scholl_z4ml.blif_0.30_1.00_1_3_henkin	0.03 s	0.05 s	0.02 s
scholl_comp.blif_0.10_1.00_5_2_henkin	0.25 s	0.27 s	0.02 s
scholl_z4ml.blif_0.20_1.00_0_3_henkin	0.04 s	0.05 s	0.01 s
scholl_z4ml.blif_0.20_0.20_2_2_henkin	0.05 s	0.08 s	0.03 s
scholl_z4ml.blif_0.30_0.10_1_2_henkin	0.07 s	0.10 s	0.03 s
scholl_comp.blif_0.10_0.50_0_3_henkin	1.57 s	1.60 s	0.03 s
scholl_comp.blif_0.10_1.00_4_1_henkin	0.27 s	0.30 s	0.03 s
scholl_term1.blif_0.20_1.00_5_2_henkin	2.70 s	2.74 s	0.04 s
scholl_comp.blif_0.50_1.00_2_2_henkin	0.26 s	0.30 s	0.04 s
scholl_comp.blif_0.20_0.20_2_2_henkin	0.36 s	0.40 s	0.04 s
scholl_comp.blif_0.10_1.00_9_3_henkin	0.27 s	0.32 s	0.05 s
scholl_comp.blif_0.10_1.00_0_1_henkin	0.26 s	0.32 s	0.06 s
scholl_z4ml.blif_0.50_0.10_2_2_henkin	0.31 s	0.37 s	0.06 s
scholl_comp.blif_0.10_0.20_1_3_henkin	0.30 s	0.36 s	0.06 s
scholl_comp.blif_0.20_0.50_2_3_henkin	0.31 s	0.37 s	0.06 s
scholl_comp.blif_0.10_0.20_2_1_henkin	0.49 s	0.56 s	0.07 s
scholl_comp.blif_0.30_0.50_2_1_henkin	0.46 s	0.54 s	0.08 s
scholl_C499.blif_0.10_1.00_7_2_henkin	2.16 s	2.24 s	0.08 s
scholl_comp.blif_0.60_1.00_9_2_henkin	0.29 s	0.37 s	0.08 s
scholl_comp.blif_0.10_0.50_1_3_henkin	0.27 s	0.37 s	0.10 s
scholl_comp.blif_0.20_1.00_5_1_henkin	3.41 s	3.51 s	0.10 s
scholl_comp.blif_0.10_1.00_4_2_henkin	0.29 s	0.42 s	0.13 s
scholl_comp.blif_0.10_1.00_2_1_henkin	0.36 s	0.52 s	0.16 s
scholl_z4ml.blif_0.50_0.10_2_3_henkin	0.32 s	0.49 s	0.17 s
tentrup_round_robin_arbiter_system_1	6.93 s	7.10 s	0.17 s
scholl_C499.blif_0.20_1.00_7_2_henkin	1.88 s	2.08 s	0.20 s
scholl_term1.blif_0.50_1.00_3_1_henkin	2.37 s	2.59 s	0.22 s
tentrup_pec_multiplexer_1_8	1.09 s	1.33 s	0.24 s
tentrup_pec_multiplexer_1_4	1.50 s	1.74 s	0.24 s
tentrup_pec_multiplexer_1_26	1.37 s	1.63 s	0.26 s

Continuation of Table 7.2			
Instance	Reference	Tracing	Delta
tentrup_pec_multiplexer_1_11	1.31 s	1.59 s	0.28 s
tentrup_pec_multiplexer_1_2	1.29 s	1.57 s	0.28 s
scholl_C499.blif_0.50_1.00_8_1_henkin	2.46 s	2.74 s	0.28 s
scholl_C499.blif_0.10_1.00_9_2_henkin	3.23 s	3.56 s	0.33 s
tentrup_pec_multiplexer_4_50	1.34 s	1.73 s	0.39 s
tentrup_pec_multiplexer_5_11	1.29 s	1.72 s	0.43 s
scholl_comp.blif_0.50_1.00_5_1_henkin	3.12 s	3.57 s	0.45 s
tentrup_pec_multiplier_1_10	1.47 s	1.92 s	0.45 s
tentrup_pec_multiplexer_5_2	2.66 s	3.13 s	0.47 s
tentrup_pec_multiplexer_6_84	1.42 s	1.89 s	0.47 s
tentrup_pec_multiplexer_3_6	2.33 s	2.84 s	0.51 s
tentrup_pec_multiplexer_2_2	1.29 s	1.82 s	0.53 s
tentrup_pec_multiplexer_3_30	1.38 s	1.92 s	0.54 s
php-010-008.shuffled-as.sat05-1171.cnf	2.00 s	2.56 s	0.56 s
scholl_comp.blif_0.50_1.00_9_1_henkin	2.80 s	3.36 s	0.56 s
tentrup_pec_multiplier_1_20	1.57 s	2.13 s	0.56 s
tentrup_pec_multiplier_3_3	1.79 s	2.38 s	0.59 s
tentrup_pec_multiplier_3_4	1.78 s	2.39 s	0.61 s
scholl_comp.blif_0.60_1.00_4_1_henkin	10.72 s	11.34 s	0.62 s
tentrup_pec_multiplier_1_11	1.61 s	2.26 s	0.65 s
tentrup_pec_multiplier_1_9	1.48 s	2.13 s	0.65 s
tentrup_pec_multiplexer_5_12	2.49 s	3.20 s	0.71 s
tentrup_pec_multiplier_3_2	1.87 s	2.60 s	0.73 s
tentrup_pec_multiplier_1_30	1.59 s	2.34 s	0.75 s
tentrup_pec_multiplier_4_14	1.42 s	2.18 s	0.76 s
tentrup_pec_look_ahead_arbiter_3_12	2.92 s	3.70 s	0.78 s
tentrup_pec_multiplexer_5_17	3.07 s	3.87 s	0.80 s
tentrup_pec_look_ahead_arbiter_2_24	3.19 s	4.07 s	0.88 s
tentrup_pec_look_ahead_arbiter_8_4	3.19 s	4.14 s	0.95 s
tentrup_pec_look_ahead_arbiter_3_3	2.87 s	3.85 s	0.98 s
tentrup_pec_multiplexer_6_51	5.56 s	6.59 s	1.03 s
tentrup_pec_multiplier_5_91	3.65 s	4.72 s	1.07 s
tentrup_pec_look_ahead_arbiter_1_14	3.19 s	4.26 s	1.07 s
tentrup_pec_multiplier_5_81	3.70 s	4.78 s	1.08 s
tentrup_pec_multiplexer_10_41	8.42 s	9.50 s	1.08 s
tentrup_pec_look_ahead_arbiter_5_10	3.45 s	4.54 s	1.09 s
tentrup_pec_look_ahead_arbiter_1_11	3.12 s	4.24 s	1.12 s
tentrup_pec_multiplier_7_13	5.13 s	6.25 s	1.12 s
tentrup_pec_look_ahead_arbiter_1_2	2.82 s	3.96 s	1.14 s
tentrup_pec_look_ahead_arbiter_5_3	3.21 s	4.42 s	1.21 s
tentrup_pec_look_ahead_arbiter_5_12	3.13 s	4.37 s	1.24 s

Continuation of Table 7.2			
Instance	Reference	Tracing	Delta
tentrup_pec_multiplexer_8_59	6.28 s	7.52 s	1.24 s
tentrup_pec_look_ahead_arbiter_8_11	3.03 s	4.27 s	1.24 s
tentrup_pec_look_ahead_arbiter_5_5	2.99 s	4.26 s	1.27 s
tentrup_pec_look_ahead_arbiter_6_70	2.95 s	4.23 s	1.28 s
tentrup_pec_multiplier_7_87	4.74 s	6.06 s	1.32 s
tentrup_pec_look_ahead_arbiter_7_9	3.39 s	4.74 s	1.35 s
tentrup_pec_look_ahead_arbiter_6_12	3.41 s	4.77 s	1.36 s
tentrup_pec_look_ahead_arbiter_7_22	2.90 s	4.27 s	1.37 s
tentrup_pec_look_ahead_arbiter_1_35	2.83 s	4.21 s	1.38 s
tentrup_pec_look_ahead_arbiter_10_2	3.20 s	4.58 s	1.38 s
tentrup_pec_look_ahead_arbiter_3_15	3.02 s	4.44 s	1.42 s
tentrup_pec_look_ahead_arbiter_2_4	2.94 s	4.44 s	1.50 s
tentrup_pec_look_ahead_arbiter_6_2	2.86 s	4.38 s	1.52 s
tentrup_pec_look_ahead_arbiter_6_93	3.04 s	4.67 s	1.63 s
tentrup_pec_look_ahead_arbiter_5_2	2.75 s	4.39 s	1.64 s
tentrup_pec_look_ahead_arbiter_6_40	2.75 s	4.44 s	1.69 s
tentrup_pec_look_ahead_arbiter_5_1	3.25 s	4.97 s	1.72 s
tentrup_pec_look_ahead_arbiter_9_8	3.72 s	5.46 s	1.74 s
tentrup_ltl2dba_alpha_environment_8	49.26 s	51.11 s	1.85 s
tentrup_pec_look_ahead_arbiter_9_36	3.12 s	4.98 s	1.86 s
tentrup_pec_adder_n_bit_3_1	4.02 s	5.89 s	1.87 s
tentrup_prioritized_arbiter_environment_4	37.22 s	39.15 s	1.93 s
tentrup_pec_adder_n_bit_4_4	3.76 s	6.16 s	2.40 s
tentrup_pec_adder_n_bit_1_35	3.88 s	6.33 s	2.45 s
tentrup_pec_adder_n_bit_4_11	4.31 s	6.78 s	2.47 s
tentrup_pec_adder_n_bit_2_10	3.38 s	5.90 s	2.52 s
tentrup_pec_adder_n_bit_5_90	4.43 s	6.96 s	2.53 s
tentrup_pec_adder_n_bit_10_32	7.33 s	9.92 s	2.59 s
tentrup_pec_adder_n_bit_3_38	4.05 s	6.64 s	2.59 s
tentrup_pec_adder_n_bit_7_26	6.31 s	9.15 s	2.84 s
urquhart3_25bis.shuffled.cnf	11.02 s	15.88 s	4.86 s
x1_40.shuffled.cnf	40.17 s	47.43 s	7.26 s
tentrup_genbuf2_system_4	219.49 s	238.26 s	18.77 s
battleship-7-12-unsat.cnf	376.20 s	399.61 s	23.41 s
End of Table 7.2			

Conclusion & Future Work

We presented a workflow to certify unsatisfiability of QBFs in the DQBF solver PEDANT using QRAT. With this, we made an important step towards a general certification mechanism in PEDANT for unsatisfiable DQBFs. We used the fact that QRAT simulates the expansion-based proof system $\forall\text{Exp}+\text{Res}$ [JMS15]. Our proposed QRAT certificate structure closely follows the polynomial time simulation of $\forall\text{Exp}+\text{Res}$ [KS19]. Our experiments on unsatisfiable QBF instances from the PCNF tracks of QBFEVAL'19 and QBFEVAL'20 showed that we can successfully verify unsatisfiable QBF instances with our generated certificates using the checker QRAT-TRIM [HSB14a]. We found that the majority of lines in our generated proofs are QRAT modification lemmas utilizing Extended Universal Reduction which are added in the post-processing. This bottleneck is a result of the definition of QRAT itself, which allows the elimination of just a single variable. In our case this leads to a quadratic blowup of lines with respect to the number of universal variables and the number of clauses to eliminate universal variables from.

One of the next steps in this project could be the full implementation of the QRAT format. This means, that all results of PEDANT could be certified using QRAT. In this step we concentrated on certifying just unsatisfiable QBF using QRAT. However, the QRAT format would also have the potential to certify satisfiable QBFs. By additionally certifying satisfiable QBFs using QRAT, we would create a consistent certification workflow.

Because PEDANT is a DQBF solver, certifying unsatisfiable QBFs using QRAT is a step into the right direction but ultimately not the final goal. In a DQBF solver it would be necessary to certify unsatisfiable DQBFs. There is already a sound and complete refutational DQBF proof system called DQRAT [Bl20] which lifts QRAT to DQBF. Similar to QRAT, DQRAT also simulates the expansion-based DQBF proof system $\forall\text{Exp}+\text{Res}$ [JMS15]. We believe that the generation of DQRAT certificates for unsatisfiable DQBFs would not substantially differ from the current generation of QRAT certificates. However, there is no DQRAT checker at the moment such that the generated DQRAT certificates could be verified.

8. CONCLUSION & FUTURE WORK

Further, we could also address the lack of a DQRAT checker. In order to provide a suitable checker we could either build a DQRAT checker from scratch or lift up QRAT-TRIM to DQRAT.

List of Figures

1.1 Hierarchy of complexity classes	3
2.1 DNF to CNF Transformations	6
2.2 QBF in PCNF	8
2.3 Variable Substitution Examples	10
2.4 Q-Resolution Refutation	14
2.5 \forall Exp+Res Refutation	16
3.1 Certification Workflow	20
3.2 Example illustrating AT/RAT property of a clause	21
3.3 Example DRAT Refutations	22
3.4 QRAT Examples for QBFs in QDIMACS notation	25
4.1 Refutation of Arbiter Formula	31
5.1 Running example in QDIMACS encoding	33
5.2 QRAT Refutation of QBF in Figure 5.1, QRAT-Blocks are highlighted with colors, red=pre-processing, orange=definitions, green=post-processing, yel- low=deletions, blue=reductions	34
5.3 A resolution path from u to e_4	40
6.1 Structure of final QRAT certificate with respect to temporary files	45
6.2 Former runtime impact of tracing module with respect to certificate size	46
7.1 Runtime impact of proof tracing	50
7.2 Memory Utilization	52
7.3 Certificate size and verification time	52
7.4 Runtime impact of proof tracing on DQBF Instances in Log Scale	53

Bibliography

- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [BBC⁺19] Olaf Beyersdorff, Joshua Blinkhorn, Leroy Chew, Renate Schmidt, and Martin Suda. Reinterpreting dependency schemes: Soundness meets incompleteness in dqbf. *Journal of Automated Reasoning*, 63(3):597–623, Oct 2019.
- [BCJ14] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. Henkin quantifiers and boolean formulae: A certification perspective of dqbf. *Theoretical Computer Science*, 523:86–100, 2014.
- [BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
- [BFH⁺20] Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2020.
- [BKF95] H.K. Buning, M. Karpinski, and A. Flogel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.
- [Bli20] Joshua Blinkhorn. Simulating dqbf preprocessing techniques with resolution asymmetric tautologies. *Electron. Colloquium Comput. Complex.*, 27:112, 2020.
- [CFHH⁺17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236. Springer International Publishing, 2017.

- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71)*, page 151–158. Association for Computing Machinery, 1971.
- [ES04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [FFRT17] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. Encodings of bounded synthesis. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 354–370. Springer Berlin Heidelberg, 2017.
- [GNT06] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006.
- [HHW13a] Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction, CADE'13*, page 345–359. Springer-Verlag, 2013.
- [HHW13b] Marijn J.H. Heule, Warren A. Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188, 2013.
- [HJB10] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for cnf formulas. pages 357–371, 10 2010.
- [HJS19a] Marijn Heule, Matti Järvisalo, and Martin Suda. Sat competition 2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11:133–154, 2019.
- [HJS19b] Marijn J.H. Heule, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Report Series B*. Department of Computer Science, University of Helsinki, Finland, 2019.
- [HSB14a] Marijn J. H. Heule, Martina Seidl, and Armin Biere. Efficient extraction of skolem functions from qrat proofs. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 107–114, 2014.
- [HSB14b] Marijn J. H. Heule, Martina Seidl, and Armin Biere. A unified proof system for qbf preprocessing. In *Automated Reasoning*, pages 91–106. Springer International Publishing, 2014.

- [JBS⁺07] Toni Jussila, Armin Biere, Carsten Sinz, Daniel Kröning, and Christoph M. Wintersteiger. A first step towards a unified proof checker for qbf. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 201–214, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [JMS15] Mikoláš Janota and Joao Marques-Silva. Expansion-based qbf solving versus q-resolution. *Theoretical Computer Science*, 577:25–42, 2015.
- [JS17] Susmit Jha and Sanjit Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54:693–726, 2017.
- [KL99] Hans Kleine Büning and Theodor Lettmann. *Propositional logic - deduction and algorithms*, volume 48 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1999.
- [Kro09] Daniel Kroening. Software verification. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 505–532. IOS Press, 2009.
- [KS19] Benjamin Kiesl and Martina Seidl. QRAT polynomially simulates \forall -Exp+Res. In *Theory and Applications of Satisfiability Testing - SAT 2019*, volume 11628 of *Lecture Notes in Computer Science*, pages 193–202. Springer, 2019.
- [KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.
- [Lam17] Peter Lammich. Efficient verified (un)sat certificate checking. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 237–254. Springer International Publishing, 2017.
- [Lip10] Richard J. Lipton. *Savitch’s Theorem*, pages 135–138. Springer US, Boston, MA, 2010.
- [MSJPM09] Marques-Silva, I. J. P., Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Biere, A., Heule, M., van Maaren, H., Walsh, T. (Eds.), Handbook of Satisfiability*, page 131–153. IOS Press, 2009.
- [MZ09] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52 (8), page 76–82, 2009.
- [PSS19] Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider. Combining resolution-path dependencies with dependency learning. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 306–318, Cham, 2019. Springer International Publishing.

- [Rin09] Jussi Rintanen. Planning and SAT. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 483–504. IOS Press, 2009.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [RSS21] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Certified DQBF solving by definition extraction. *CoRR*, abs/2106.02550, 2021.
- [SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time. In *Proc. Theory of Computing*, pages 1–9. Association for Computing Machinery, 1973.
- [Tse70] G.S. Tseytin. On the complexity of derivation in propositional calculus. In *Zapiski Nauchnykh Seminarov LOMI 8, 234-259 (1968)*, english translation of this volume: Consultants Bureau, N.Y., page 115–125. 1970.
- [VG11] Allen Van Gelder. Variable independence and resolution paths for quantified boolean formulas. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, pages 789–803, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [VWM15] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.
- [WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429. Springer International Publishing, 2014.