TU WIEN Informatics

Certified Circuit Reconstruction for QBF

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Mihai-Alexandru Weng
Matrikelnummer 12228521

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Stefan Szeider
Mitwirkung: Dr. Friedrich Slivovsky

Wien, 1. April 2024

Mihai-Alexandru Weng	Stefan Szeider

TU WIEN Informatics

Certified Circuit Reconstruction for QBF

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Mihai-Alexandru Weng
Registration Number 12228521

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Stefan Szeider
Assistance: Dr. Friedrich Slivovsky

Vienna, April 1, 2024

Mihai-Alexandru Weng	Stefan Szeider

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Mihai-Alexandru Weng

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, 1. April 2024

Mihai-Alexandru Weng

Acknowledgements

My deepest gratitude goes to my supervisors Prof. Dr. Stefan Szeider and Dr. Friedrich Slivovsky for the guidance throughout my thesis. I am truly appreciative of the numerous insightful discussions we had, delving into both theoretical and practical aspects of the work. I deeply appreciate their patience in explaining the necessary steps and provisioning of resources required for completion. Moreover, their prompt responses to my queries and constructive feedback on my writing have been invaluable. Last but not least, I want to thank you for finding this topic that I thoroughly enjoyed and pointing me toward a research direction I would like to pursue in the future.

Kurzfassung

An der Lösung von QBF sind mehrere Programme beteiligt, z. B. Vorverarbeitungstechniken, Proof Checker usw. Da es sich bei all diesen Schritten um Computerprogramme handelt, können sie jedoch schwer erkennbare Fehler enthalten. Zum Beispiel können wir einen QBF-Solver verwenden, um zu prüfen, ob eine QBF wahr oder falsch ist, aber wir haben keine Garantie, dass die Antwort des QBF-Solvers richtig ist. Daher können wir einen Proof-Trace anhängen, der in einem Proof-Checker als Zertifikat für das Ergebnis des QBF-Solvers verwendet werden kann.

In dieser Arbeit konzentrieren wir uns auf ein Programm, das eine QBF in der konjunktiven Normalform von Prenex in eine quantifizierte Schaltung im QCIR-Format transformiert. Dieses Programm muss eine äquivalente quantifizierte Schaltung für die Eingabe-QBF rekonstruieren. Wir wollen also eine Möglichkeit haben, die QCIR-Transformation zu zertifizieren.

Um diese Probleme zu lösen, schlagen wir eine Methode zur Zertifizierung der Rekonstruktion vor. Wir definieren die Bedingungen, unter denen eine Schaltung die Schaltungsrekonstruktion einer PCNF ist. Wir stellen ein Verfahren vor, das aus der Schaltung QBF und ihrer Widerlegung eine Widerlegung der ursprünglichen PCNF erzeugt. Der Beweis dient als Zertifikat für die rekonstruierte Schaltung.

In Experimenten haben wir die von verschiedenen Programmen erzeugten Schaltungsrekonstruktionen auf Standard-QBF-Benchmarks und zufälligen Instanzen zertifiziert.

Abstract

QBF solving involves several programs such as preprocessing techniques, proof checkers, etc. But all these steps, being computer programs, can contain elusive errors. For example, using a QBF solver we can check whether a QBF is true or false, but we don't have any guarantee that the answer of a QBF solver is correct. Therefore, we can attach a proof trace that can be used in a proof checker as a certificate of the QBF solver's result.

In this thesis we focus on a program that takes a QBF in prenex conjunctive normal form and transforms it into a quantified circuit in the QCIR format. This program must reconstruct an equisatisfiable quantified circuit for the input QBF. Thus, we want to have a way of certifying the QCIR transformation.

To address these issues, we propose a method for certifying the reconstruction. We define conditions under which a circuit is the circuit reconstruction of a PCNF. We present a procedure that generates a refutation of the original PCNF from the circuit QBF and its refutation. The proof serves as a certificate of the reconstructed circuit.

In the experiments, we certified the circuit reconstructions generated by different programs on standard QBF benchmarks and random instances.

Contents

CHAPTER 1

Introduction

Considering a Boolean formula we might be interested if there exists an assignment for the expression such that its evaluation is true, in other words, we are interested in whether a formula is satisfiable, the famous SAT problem.

An important characteristic of the SAT problem is its complexity class, NP-complete [Coo23]. A problem is said to be in NP if a given solution can be checked in polynomial time. A problem is NP-complete if any other NP problem can be reduced to it in polynomial time. This raises a useful application of SAT solvers, that is, their ability to solve other NP problems, such as the traveling salesman problem, graph coloring, and so on. Additionally, SAT solvers can be used in areas like model checking, combinational equivalence checking [MS08].

With the success of SAT solving, we can go one step further and investigate the case where the given formula is a quantified Boolean formula. Therefore, we get the equivalent of the SAT problem for the quantified Boolean formulas, the QSAT problem. QSAT is PSPACE-complete, where PSPACE is the class of problems that can be solved using a polynomial amount of space. Besides the usual application of the problem being complete, QBF solvers can be also used in verification and artificial intelligence applications [SBPS19].

Taking into consideration the importance of these problems, and their hard complexity classes, there is a lot of research in developing faster and correct programs involved in the process.

A SAT solver usually takes as an input a Boolean formula in conjunctive normal form and gives an assignment for the satisfying assignment or a proof that confirms the SAT instance does not have a satisfying assignment. The proof is based on a proof system, in the case of propositional logic, this can be the resolution proof system that only includes the resolution rule. Conflict-driven clause learning is one approach on which SAT solvers are based and is also capable of deriving proof of unsatisfiable in case of a false instance.

Most of the results in the SAT setting can be adapted to QSAT [BJLS21]. Similarly to conjunctive normal form inputs of an SAT solver, there is also a prenex conjunctive normal form for QSAT solvers. The widespread use of conjunctive normal form comes from the transformation of any formula in an equisatisfiable conjunctive normal form, thus making the solvers take a more specific formula input than a general formula. The output of a QSAT solver, in case of false instances, will be a proof of unsatisfiability, as in SAT solving, but with a different proof system. Although QBF solvers are similar in the case of false instances, there is a difference when dealing with true instances, where QBF solvers will also need to provide a proof. QSAT solvers can also be implemented using a quantified version of the algorithm used in SAT solvers.

Given a proof of a QBF one needs to check that this proof is correct, thus the need for proof checkers. The proof checker we will be using is based on the QRAT property [HSB14]. QRAT was developed to be a proof system for QBFs preprocessing, in order to get a modified formula that is equisatisfiable with the input. But, this approach can also be used as a refutation or satisfaction proof of the QBF.

So far we only spoke about inputs given as a prenex conjunctive normal form, but there is another interesting format for a QBF and that is its circuit form, presented in [qci]. Sometimes, this format can be beneficial for a QBF solver as seen in [JKS16]. In the same paper, is also presented a way in which one can get a circuit form of a QBF, which we will call circuit reconstruction. This technique is mainly based on pattern matching the input formula and trying to deduce which variables can be seen as gates. Additionally, an interpolation-based circuit reconstruction has been developed in [Sli20].

Even though these tools are built on a strong mathematical foundation, this cannot always guarantee that the implementations of the procedures do not contain errors. Thus, without slowing down the solver's efficiency, one can implement some part of the process that can be trusted. For example, in [CFHH+17] a proof format was developed that can be verified by a checker which was formally proved.

1.1 Aim of the Thesis

With the interest of non-conjunctive normal form solving [JKS16], one can apply the transformation from prenex conjunctive normal form to a circuit form and use the solver for the circuit input. But, *how can we be sure the transformation is sound? How can we be sure the new formula is equisatisfiable?*

In this thesis, we aim to answer these questions by providing a way of certifying the reconstruction step. Moreover, this procedure will also include a method for the proof checking of a QCIR proof, that was currently unavailable.

In order to certify the transformation, we start by defining what a circuit reconstruction needs to satisfy. With our definition in place, we want to show that a proof of the input formula can be derived from the reconstructed circuit. In the end, we show that a valid proof for the input formula can be derived by our approach if the program that

produced the circuit form of a PCNF respects the definition of a circuit reconstruction, thus certifying the program.

Once we have proven the soundness of our procedures, we can begin to implement and test them in the certification of a circuit converter. The testing will be based on random QBF instances and well-known benchmarks, and if we don't find any errors in the process of generating the input proof, we can say that the reconstruction was correct.

This approach mainly focuses on the false QBF instances due to the open question of transforming a Q-consensus proof to QRAT proof. Additionally, the refutational completeness of the Q-resolution proof system for prenex conjunctive normal forms [BJLS21] is another reason for the interest in false instances of QBF.

1.2 Structure of the Thesis

This thesis is structured as follows. In Chapter 2, we briefly walk through the preliminary concepts needed for the main result on circuit certification presented in Chapter 3 followed by its implementation in Chapter 4.
The experiments are in Chapter 5, where we test two circuit converters on various instances of QBFs.
In Chapter 6, we present our conclusions alongside with a view to future work.

CHAPTER 2

Preliminaries

In this preliminary chapter, we are going to set the building blocks needed for later chapters without the need for prior knowledge. In all sections of this chapter, every definition will be accompanied by its respective example, and a corresponding illustration for each procedure.

The structure of this chapter will be as follows: we start by introducing syntax and semantics for our formulas of interest, followed by a transformation of a generic formula into conjunctive normal form. After that, we present two proof systems for QBFs and show how an algorithm can solve a QBF. Lastly, we talk about the input formats a QBF should be given in to be understood by a solver.

2.1 Quantified Boolean Formulas

A quantified Boolean formula (QBF) is an extension of propositional Boolean formula with quantified variables. For example, $(x_1 \vee x_2 \wedge x_3) \rightarrow x_4$ is a propositional formula whilst $\forall x_1 x_3 \exists x_2 x_4 (x_1 \vee x_2 \wedge x_3) \rightarrow x_4$ is the prior formula quantified.

Every propositional formula can be represented as a QBF, by existentially quantifying the variables. For example, from $x_1 \vee x_2$ we get $\exists x_1 x_2 (x_1 \vee x_2)$.

In this section, the definition for QBF is given in the prenex form, where all quantifiers appear in front of a quantifier-free formula called matrix. It can be assumed that the matrix is in the conjunctive normal form. If the matrix is not in CNF, we can apply the Tseitin transformation presented in Section 2.3. This transformation will produce an equisatisfiable formula of the given input. Additionally, any generic QBF can be transformed into the prenex conjunctive normal form.

Definition 2.1 (Literal)**.** A **literal** is a Boolean variable x or its negation $\overline{x}$. We define $\mathbf{var}(x) = \mathbf{var}(\overline{x}) = x$.

Example 2.2. In formula $(x \vee \overline{y} \wedge z)$ the literals are $\{x, \overline{y}, z\}$ and $\text{var}(\overline{y}) = y$. If we want the negation of the literal $l = \overline{y}$, we will have $\overline{l} = y$.

Definition 2.3 (Clause)**.** A **clause** is a disjunction of literals.

Example 2.4. $(x_1 \vee \overline{x_2} \vee x_3)$ is a clause. But $(y_1 \vee \overline{y_2} \wedge y_3)$ is not because it contains an and operator, neither $(z_1 \vee (\overline{z_2} \wedge z_3))$ because the second term is not a literal.

A clause can also be represented as a set, $(x_1 \vee \overline{x_2} \vee x_3)$ to $\{x_1, \overline{x_2}, x_3\}$. This representation is useful for computer programs due to its processing as a list.

Definition 2.5 (Cube)**.** A **cube** is a conjunction of literals.

Example 2.6. $(x_1 \wedge \overline{x_2} \wedge \overline{x_3})$ is a cube. Whilst, $(y_1 \wedge (\overline{y_2} \vee \overline{y_3})), (x_1 \rightarrow \overline{x_2} \wedge \overline{z_3})$ are not.

Definition 2.7 (CNF)**.** A propositional formula is in **conjunctive normal form** if it is a conjunction of clauses.

Example 2.8. $x_1 \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$ is in conjunctive normal form.

Definition 2.9 (QBF in PCNF)**.** A **quantified Boolean formula** in **prenex conjunctive normal form** is of the form

$$\Pi\psi,$$

which consists of a CNF ψ called **matrix**, and a **prefix** $\Pi = Q_1 X_1 \dots Q_k X_k$, with $Q_i \in \{\exists, \forall\}$, $Q_i \neq Q_{i+1}$, and X_i pairwise disjoint sets of variables.

Example 2.10. $\forall x \exists y(x \vee y)$ is a QBF in PCNF. A QBF that is not a PCNF is $\forall x(x) \rightarrow \exists y(y)$. Also, it is not allowed to have two consecutive quantifiers of the same time $\forall x \forall y$, instead $\forall xy$ should be used.

Definition 2.11 (Quantifier Block)**.** A **quantifier block** is $Q_i X_i$ (from Definition 2.9). Moreover, $Q_1 X_1$ is the **outermost quantifier block** and $Q_k X_k$ is the **innermost quantifier block**.

A variable x is quantified at **level** i, if $x \in X_i$ and denoted by $\text{lv}(x) = i$. We can extend it for literals with $\text{lv}(l) = \text{lv}(\text{var}(l))$. Furthermore, we can define $\textbf{quant}(\Pi, x) = \textbf{quant}(\Pi, \overline{x}) = Q_i$.

Example 2.12. If we have the prefix $\Pi = \exists ab \forall uv \exists xyz$, then the outermost block is $\exists ab$, the innermost block is $\exists xyz$, $\text{lv}(u) = 2$, and $\text{quant}(\Pi, v) = \forall$.

Definition 2.13 (Substitution)**.** $\Pi\psi[t/x]$ denotes the **substitution** of x by t.

Example 2.14. $\forall xy \exists z(x \vee y) \wedge z[1/z]$ we get $\forall xy \exists z(x \vee y) \wedge 1$.

Definition 2.15 (QBF Semantics)**.** A QBF $\forall x \Pi\psi$ is true if $\Pi\psi[0/x]$ and $\Pi\psi[1/x]$ are true. A QBF $\exists x \Pi\psi$ is true if $\Pi\psi[0/x]$ or $\Pi\psi[1/x]$ is true.

Example 2.16. $\forall x(x \vee \overline{x})$ is a true QBF, we have $(0 \vee \overline{0})$ which evaluates to true and $(1 \vee \overline{1})$ which also evaluates to true. Another true QBF is $\exists x \forall y(x \vee y)$ because it will be true when we assign x to 1. A false QBF is $\forall x(x)$, because we can take $[0/x]$ producing false.

Definition 2.17 (Proof system [Was22])**.** A **proof system** is a quadruple $S = (\text{Lang}, \text{Exp}, \text{Ax}, \text{R})$, where Lang, Exp stands for the language and the well-formed formulas, expression, Ax is the set of axioms in the system, and the R are the rules of inference of S.

2.2 Proofs

In this section, we define what we mean by proof and introduce the notion of redundant clauses. This section is mainly based on [Zel19].

Definition 2.18 (Redundant)**.** Let ϕ be a propositional formula. A formula F is **redundant** w.r.t. ϕ if $\phi \vDash F$.

A redundant formula is a formula that does not add extra information because it is true from the context. With Definition 2.18 we can define what a proof is.

Definition 2.19 (Proof)**.** A sequence of clauses, ending with the empty clause, that are redundant w.r.t. ϕ.

In the following, we will list a couple of the properties that are sufficient for redundancy.

Definition 2.20 (Tautology)**.** Let p be a propositional formula, if the formula p is true given every assignment, then p is a **tautology**.

Example 2.21. $(\neg l \vee l)$ is a tautology.

Definition 2.22 (Asymmetric tautology)**.** Asymmetric literal addition, repeat until fix-point is $\text{ALA}(\phi, C)$: $\exists (C \vee l) \in \phi \backslash \{C\}$, then $C := C \vee \neg l$. A clause C is an **asymmetric tautology** w.r.t. ϕ if $\text{ALA}(\phi, C)$ is a tautology.

Definition 2.23 (Reverse unit propagation)**.** A clause C is a **RUP** w.r.t. propositional formula ϕ if the application of unit propagation of $\neg C$ to ϕ derives $\bot$. Where unit propagation is a procedure that takes a set of clauses, and for each literal in the set, eliminates the clauses that contain that literal, or removes it from the clauses that contain its negation. The procedure repeats these steps until fix-point. An empty clause is noted as $\bot$.

The definition of asymmetric tautology is equivalent to the reverse unit propagation definition [Zel19].

The SAT solver we will be using in this work will have as output a RUP proof, therefore, we will illustrate this proof using an example from [HHW13].

Example 2.24 (RUP proof)**.** Given the clauses of a CNF $(x_1 \vee x_2 \vee \neg x_3)$, $(\neg x_1 \vee \neg x_2 \vee x_3)$, $(x_2 \vee x_3 \vee \neg x_4)$, $(\neg x_2 \vee \neg x_3 \vee x_4)$, $(x_1 \vee x_3 \vee x_4)$, $(\neg x_1 \vee \neg x_3 \vee \neg x_4)$, $(\neg x_1 \vee x_2 \vee x_4)$ and $(x_1 \vee \neg x_2 \vee \neg x_4)$.

We have the RUP proof: $(x_1 \vee x_2)$, (x_1), (x_2), $\bot$. The negation of $(x_1 \vee x_2)$, will produce the literals $\neg x_1$ and $\neg x_2$, using unit propagation, from the first clause we will have $\neg x_3$. From $\neg x_2$ and $\neg x_3$, we can use them in clause $(x_2 \vee x_3 \vee \neg x_4)$ and get $\neg x_4$. But, $\neg x_1$ and $\neg x_3$ in $(x_1 \vee x_3 \vee x_4)$ produce x_4. Therefore, with x_4 and $\neg x_4$ we get $\bot$. Thus, our clause has RUP and can be added to our list of CNF, because RUP preserves logical equivalence.

For $\neg x_1$, we have the previous clause $(x_1 \vee x_2)$, and get x_2 by unit propagation. From $(x_1 \vee \neg x_2 \vee \neg x_4)$, we get $\neg x_4$. From $(\neg x_2 \vee \neg x_3 \vee x_4)$, we get $\neg x_3$. And the clause $(\neg x_1 \vee \neg x_3 \vee \neg x_4)$, produce $\bot$, thus making x_1 have RUP.

For $\neg x_2$, we get x_4 from $(\neg x_1 \vee x_2 \vee x_4)$. From $(x_2 \vee x_3 \vee \neg x_4)$, we get x_3. From $(\neg x_1 \vee \neg x_3 \vee \neg x_4)$, we get $\bot$, therefore, x_2 has RUP.

The last clause we check is $\bot$, and its negation is true. We get x_3, from $(\neg x_1 \vee \neg x_2 \vee x_3)$, x_1 and x_2. From $(\neg x_2 \vee \neg x_3 \vee x_4)$, we get x_4. From $(\neg x_1 \vee \neg x_3 \vee \neg x_4)$, we get $\bot$. Thus, $\bot$ has RUP. Making our CNF unsatisfiable.

Definition 2.25 (Resolution tautology (blocked clause))**.** A clause C is a **resolution tautology** w.r.t. propositional formula ϕ if:

- it is a tautology,
- or, there is a literal $l \in C$ with all $D \in \phi$ that contains $\neg l$, by applying resolution rule will produce a tautology.

Definition 2.26 (Resolution asymmetric tautology)**.** A clause C is a **resolution asymmetric tautology** w.r.t. propositional formula ϕ if:

- it is an asymmetric tautology,
- or, there is a literal $l \in C$ with all $D \in \phi$ that contains $\neg l$, by applying resolution rule will produce an asymmetric tautology.

These properties can be arranged in the hierarchy presented in Figure 2.1 based on their preserved equivalence and their overlaps with other properties.

2.3 Tseitin Transformation

Tseitin transformation is a procedure that takes a propositional formula and computes a new formula in conjunctive normal form that is equisatisfiable to the initial formula. Additionally, the transformation is linear in the size of the input.

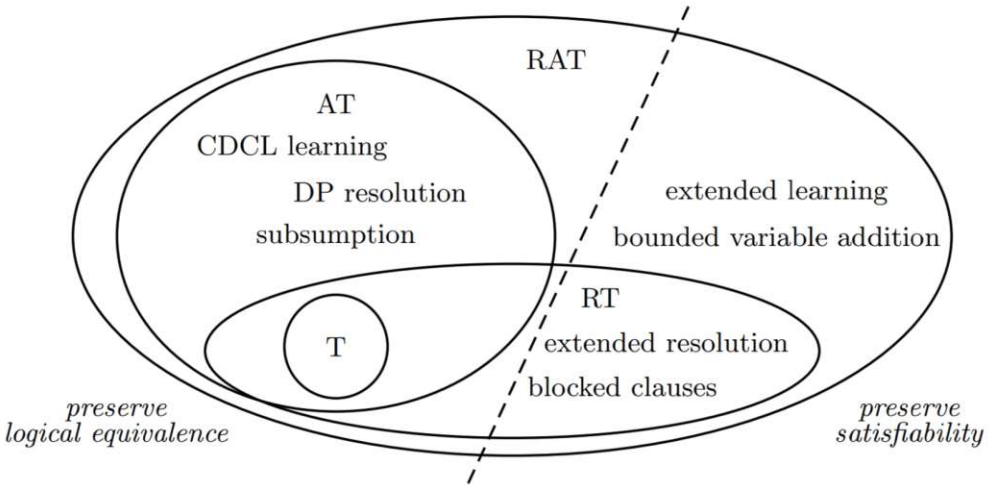

Figure 2.1: Hierarchy of redundant properties [Zel19].

As stated in [MVVdB16], for each logical connective, we introduce a new variable corresponding to its output. The new variables can replace the occurrence of a subformula, such that each logical connective is between two variables. These equivalences $T \leftrightarrow A \circ B$, where $\circ$ is a logical connective, can be transformed into a logically equivalent CNF. In Table 2.1, we display the clauses for the $\wedge$ and $\vee$.

Logical connective	Conjunctive normal form
$T \leftrightarrow A \wedge B$	$(T \vee \neg A \vee \neg B) \wedge (\neg T \vee A) \wedge (\neg T \vee B)$
$T \leftrightarrow A \vee B$	$(\neg T \vee A \vee B) \wedge (T \vee \neg A) \wedge (T \vee \neg B)$

Table 2.1: Tseitin transformation of a logical connective into its conjunctive normal form.

This transformation works as follows. Given a propositional formula ϕ, for each subformula we introduce an equivalent Tseitin variable. For each equivalence, we add its CNF translation to a formula ϕ_T. After introducing all the clauses in ϕ_T, ϕ is equisatisfiable with $\phi_T \wedge T_0$, where T_0 corresponds to the outermost logical connective of ϕ.

Example 2.27. If we have the formula $(A \wedge B) \vee C$, first we should point out that, this is not in CNF. We introduce a Tseitin variable for each subformula, as follows: $T_1 \leftrightarrow (A \wedge B)$ and $T_0 \leftrightarrow (T_1 \vee C)$. We write the conjunctive normal form for each equivalence

$$\varphi_T = (T_1 \vee \overline{A} \vee \overline{B}) \wedge (\overline{T_1} \vee A) \wedge (\overline{T_1} \vee B) \wedge (\overline{T_0} \vee T_1 \vee C) \wedge (T_0 \vee \overline{T_1}) \wedge (T_0 \vee \overline{C}).$$

Finally, $\varphi_T \wedge T_0$ is the transformation of the initial formula in CNF.

The reason it works is that given φ and $\varphi_T \wedge T_0$, if φ is satisfiable, we can use the assignment to set our T_i accordingly, and T_0 is true because φ is satisfiable. In case φ is unsatisfiable, suppose $\varphi_T \wedge T_0$ is satisfiable, but this can't be the case, because we can get an assignment that should be true for φ, so $\varphi_T \wedge T_0$ is also unsatisfiable.

With the transformation for propositional formulas to CNF in place, in a QBF PCNF setting we are raising the question of where we put the Tseitin variables in the prefix. If we do not assign them in the quantified block, they will be treated as free variables, thus in the outermost existential block (if the first block is universal, we just add another existential block outside). But this has the following counter-example: $\forall x(x \wedge \overline{x})$ that is true while $\exists t \forall x(t \vee \overline{x} \vee x) \wedge (\overline{t} \vee x) \wedge (\overline{t} \vee \overline{x}) \wedge t$ which is false. Thus, we can form the following claim:

Claim 2.28. *Consider a QBF of form $\Pi\varphi$, where φ is quantifier-free formula. $\Pi\varphi$ is equisatisfiable with $\Pi\exists T(\varphi_T \wedge T_0)$, where T is the set of Tseitin variables, φ_T is the Tseitin transformation of subformulas and T_0 is the first logical connective that is applied to the formula. (If the last quantified block of Π is $\exists$, then T is appended to it.)*

Proof. Similar to the propositional case, by applying the semantic Definition 2.15 of QBF, we end up with a matrix that is formed prefixed only by the existential block of Tseitin variables. Due to the prefix being only existential we can use the reasoning at the propositional level, and we can use the prior sketch proof. □

Logical connective	Conjunctive normal form
$T \leftrightarrow A_1 \wedge A_2 \wedge ... \wedge A_n$	$(T \vee \neg A_1 \vee ... \vee \neg A_n) \wedge (\neg T \vee A_1) \wedge ... \wedge (\neg T \vee A_n)$
$T \leftrightarrow A_1 \vee A_2 \vee ... \vee A_n$	$(\neg T \vee A_1 \vee ... \vee A_n) \wedge (T \vee \neg A_1) \wedge ... \wedge (T \vee \neg A_n)$

Table 2.2: Tseitin transformation extended form.

The Tseitin transformation can be used in an extended form, presented in Table 2.2. This form will be of use later when we will implement an and/or gate in a shorter form. To see the proof for this extended version, for and case, we can see that if all the variables are true, than the clause $(T \vee \neg A_1 \vee ... \vee \neg A_n)$ is true only if T is true, in case one variable is false, than respective $(\neg T \vee A_i)$ will make T false, thus we have the equivalence between T and $A_1 \wedge A_2 \wedge ... \wedge A_n$ in conjunctive normal form. For the or case we apply the same logic as in the previous case.

2.4 The Q-Resolution Proof System

In SAT solving, if a proposition is satisfiable, we can give a satisfying assignment. If it is unsatisfiable, we will need to ensure it has no satisfying assignment. This is where a proof system comes in our help. For an unsatisfiable formula, we can give the steps in a proof system to derive a contradiction. One proof system for proposition formulas is the resolution proof system.

In QBF solving, we have a quantified version for the prior resolution proof system. This is presented in Figure 2.2. Q-resolution is refutationally complete for QBFs in PCNF [BKF95]. This means if a formula is unsatisfiable, we can derive the empty clause, by applying rules from our proof system given the formula.

$$\frac{C \cup \{l\}}{C} \quad \text{for all } x \in \mathsf{vars}(\Pi)\colon \{x, \bar{x}\} \not\subseteq (C \cup \{l\}),\ \mathsf{quant}(\Pi, l) = \forall, \text{ and } l' \leq_\Pi l \text{ for all } l' \in C \text{ with } \mathsf{quant}(\Pi, l') = \exists \qquad (red)$$

$$\frac{C_1 \cup \{p\} \qquad C_2 \cup \{\bar{p}\}}{C_1 \cup C_2} \quad \text{for all } x \in \mathsf{vars}(\Pi)\colon \{x, \bar{x}\} \not\subseteq (C_1 \cup C_2),\ \bar{p} \notin C_1,\ p \notin C_2, \text{ and } \mathsf{quant}(\Pi, p) = \exists \qquad (res)$$

$$\frac{}{C} \quad \text{for all } x \in \mathsf{vars}(\Pi)\colon \{x, \bar{x}\} \not\subseteq C \text{ and } C \in \psi \qquad (cl\text{-}init)$$

Figure 2.2: Q-resolution proof system [BJLS21].

Example 2.29. $\exists y \forall x z \exists p (y \vee x \vee z) \wedge p$ by applying only (red) for $(y \vee x \vee z)$ we get $(y \vee x)$ because $y \leq_\Pi x$ and p is not in the clause, also we can get $(y \vee z)$ because z needs to be higher than existential variables and ignore the universal quantified variables, in plus, if we apply it repeatedly we can also get (y).

Example 2.30. $\forall x y \exists z (x \vee z) \wedge (y \vee \bar{z})$ with (res) on those 2 clauses, we get $(x \vee y)$.

Example 2.31. In this example we will apply the rules to get a refutation of

$$\forall x_1 x_2 \exists y (x_1 \vee x_2 \vee \overline{y}) \wedge (y \vee \overline{x_1}) \wedge (y \vee \overline{x_y}) \wedge y \wedge (x_1 \vee \overline{x_1}).$$

Firstly we apply (cl-init) where we get

$$(x_1 \vee x_2 \vee \overline{y}) \wedge (y \vee \overline{x_1}) \wedge (y \vee \overline{x_y}) \wedge y$$

without last clause because that is a tautology. On the first and last clause, we can apply (res) and get $(x_1 \vee x_2)$. On $(x_1 \vee x_2)$ we apply twice (red) and get the empty clause. Thus, our formula is false and has a proof in the Q-resolution proof system.

2.5 The QRAT Proof System

The main objective of this work is to provide a QRAT proof for an input PCNF, where the QRAT proof is derived from the quantified circuit of the PCNF. Thus, we dedicate this section to explaining QRAT in detail.

QRAT was defined in [HSB14] in order to have a proof system that can prove different types of preprocessing rules. Additionally, if we have a QRAT proof for a QBF, we can check whether it is a valid refutation or satisfaction proof in accordance with the proof system. In the following, we will present the definition of QRAT and the properties that use the definition of QRAT in applications.

Definition 2.32 (Outer Resolvent)**.** The **outer resolvent** of clauses $C \vee l$ and $D \vee \bar{l}$ on literal l w.r.t. quantifiers Π is:

$$\mathrm{OR}(\Pi, C \vee l, D \vee \bar{l}, l) = C \cup \{k \mid k \in D, \mathrm{lv}(k) \leq \mathrm{lv}(l), k \neq \bar{l}\}, \text{for } \mathrm{quant}(\Pi, l) = \exists,$$

and

$$\mathrm{OR}(\Pi, C \vee l, D \vee \bar{l}, l) = (C \backslash \{l\}) \cup \{k \mid k \in D, \mathrm{lv}(k) \leq \mathrm{lv}(l), k \neq \bar{l}\}, \text{for } \mathrm{quant}(\Pi, l) = \forall.$$

Example 2.33. Consider the prefix $\Pi = \forall x_1 x_2 \exists y_1 y_2 \forall x_3 x_4$.

For $C = (x_2 \vee x_4 \vee y_1), D = (x_1 \vee x_3 \vee \overline{y_1})$, with the existential quantifier y_1 as the pivot, we get $\mathrm{OR} = (x_2 \vee x_4 \vee y_1 \vee x_1)$.

For $C = (y_1 \vee y_2 \vee x_1), D = (x_2 \vee \overline{x_1})$, with the universal quantifier x_1 as the pivot, we get $\mathrm{OR} = (y_1 \vee y_2 \vee x_2)$.

Definition 2.34 (Implies via unit propagation)**.** A propositional formula ψ **implies via unit propagation** a clause C, denoted by $\psi \vdash_1 C$,

if applying unit propagation on $\psi \wedge \overline{C}$ we can derive the empty clause.

Example 2.35. Given $\psi = (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (\overline{b} \vee \overline{d})$ and the clause $C = (\overline{a} \vee \overline{c})$, the negation of C is $a \wedge c$, which implies b, d, but those two with the clause $(\overline{b} \vee \overline{d})$ will derive the empty clause, thus our formula implies the clause C via unit propagation.

Definition 2.36 (QRAT)**.** A clause C has **QRAT on literal** $l \in C$ w.r.t. QBF $\Pi\psi$, if for all $D \in \psi$ with $\bar{l} \in D$:

$$\psi \vdash_1 \mathrm{OR}(\Pi, C, D, l).$$

With the QRAT definition in place, in order to make use of it we use the following theorems, from [HSB14], that help us to transform a QBF into an equisatisfiable QBF:

Theorem 2.37 (QRAT for existential [HSB14])**.** *Given a QBF $\phi = \Pi.\psi$ and a clause $C \in \psi$ with QRAT on existential literal $l \in C$ w.r.t. QBF $\phi' = \Pi'.(\psi \backslash \{C\})$, where Π' is adapted for $(\psi \backslash \{C\})$. Then ϕ and ϕ' are equisatisfiable.*

Theorem 2.38 (QRAT for universal [HSB14])**.** *Given a QBF $\phi = \Pi.\psi$ and a clause $C \in \psi$ with QRAT on universal literal $l \in C$ w.r.t. QBF $\Pi.(\psi \backslash \{C\})$. Then ϕ and $\Pi.(\psi \backslash \{C\} \cup \{C \backslash \{l\}\})$ are equisatisfiable.*

Without going into detail, these theorems are used for checking the steps we will use in a QRAT proof. In a QRAT proof system, we have the following operations: addition of a clause, deletion of a clause, and universal elimination. For the addition or deletion of a clause, the QRAT checker will check that Theorem 2.37 is respected. For the universal elimination, Theorem 2.38 needs to be followed, or an extended universal reduction rule can be applied. The EUR rule has a similar output as the QRAT elimination of a universal literal, meaning that we remove the respective literal from the clause, but the EUR rule is based on resolution paths [CH22]. As we don't need to deep dive into the inner workings of a checker, we omit the definition of EUR.

In Figures 2.3 and 2.4, we present an example from [HSB14] of a refutation that uses all the available operations in a QRAT proof, addition, deletion, and universal elimination.

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

Figure 2.3: False QBF in QDIMACS.

```
  -2 0
d -2 -3 0
   1 0
u  1 0
   0
```

Figure 2.4: QCIR proof for example in Figure 2.3.

The first line is adding the clause `-2` to our set of clauses. The second line prefixed with `d` deletes the clause `-2 -3` from our set. Lastly, the `u` operation will delete the clause that is followed by the symbol from the set but will reintroduce the clause without the first literal (the literal that has QRAT), because this is how the step is defined in Theorem 2.38 for universal elimination, we cannot only delete directly the clause as in the existential case. Although these steps only manipulate a set of clauses, the correctness of each step needs to be verified according to the proof system, in order to be a valid proof, and not only a syntax manipulation. For the current example, the clause `-2` has QRAT because the outer resolvent of `-2` and `1 2` is `1 -2`, the negation of clause `1 -2` is `-1` and `2`, with unit propagation we will derive the literal `3`, and `-2 -3` will derive $\bot$. To delete `-2 -3`, we have two outer resolvents, but both of them will include `-2 -3`. The negation of the clause is `2` and `3`, but in our list, we already have `-2 -3`, thus we can use unit propagation to derive $\bot$ in both cases. Clause `1` has QRAT because we don't have a negation of it in the list, making it trivially true by definition. The universal reduction line `u 1 0` first is checking for the clause `1` to be in the list, because it will remove it and add its clause without the QRAT literal, as Theorem 2.38. For QRAT checking, the reason is exactly as before. In the end, we can add $\bot$, because it is already in the list of clauses from the universal reduction step, making it a valid QRAT refutation.

2.6 Quantified Conflict-Driven Clause Learning

In this section, we will present an algorithm that can solve a QBF instance, with the underlying proof given in the Q-Resolution. The quantified conflict-driven clause learning is the quantified version of CDCL used for SAT solving [ZM02]. For the propositional problem in the satisfiable case, it's enough to have an assignment, thus the CDCL algorithm tries to guess an assignment that evaluates the input to true, but if the decisions lead to a false formula, the procedure derives a clause that minimizes the search space. In a similar way, in QCDCL, in order to prune the search space for a conflict, we

learn a clause, while for a solution we learn a cube [BJLS21].

Definition 2.39 (Unit literal detection)**.** A clause $C \in \phi$ is **unit** if C contains one literal, and the literal is existentially quantified. That literal is also called a **unit literal**. **Unit literal detection** applied to a QBF collects all unit clauses in the QBF.

Example 2.40. Let our formula be $\forall x_1 x_2 \exists y_1 \forall x_3 \exists y_2 (x_1 \vee x_2) \wedge x_3 \wedge y_1 \wedge \overline{y_2}$. Unit literal detection gives us $\{y_1, \overline{y_2}\}$.

Definition 2.41 (QBCP)**.** Given a PCNF ϕ and the empty assignment $A = \{\}$. We apply the following:

1. Apply universal reduction (UR) on $\phi[A]$ to get $\phi[A]'$.
2. Apply unit literal detection (UL) on $\phi[A]$ and append the result to A.
3. Repeat from 1. Stop if A hasn't changed, or the formula is true or false.

Example 2.42. Given $\phi = \forall x_1 x_2 \exists y (x_1 \vee x_2 \vee \overline{y}) \wedge (y \vee \overline{x_1}) \wedge (y \vee \overline{x_y}) \wedge y$ we apply QBCP, we start with A empty:

- We cannot apply UR.
- From ϕ with UL, we get $A = \{y\}$.
- $\phi[1/y] = \forall x_1 x_2 (x_1 \vee x_2)$, we apply UR on x_2, and we get $\phi[1/y] = x_1$.
- We cannot apply UL because no existential literal is present.
- By applying UR, we get $\phi[1/y] = \bot$. Thus, we stop.

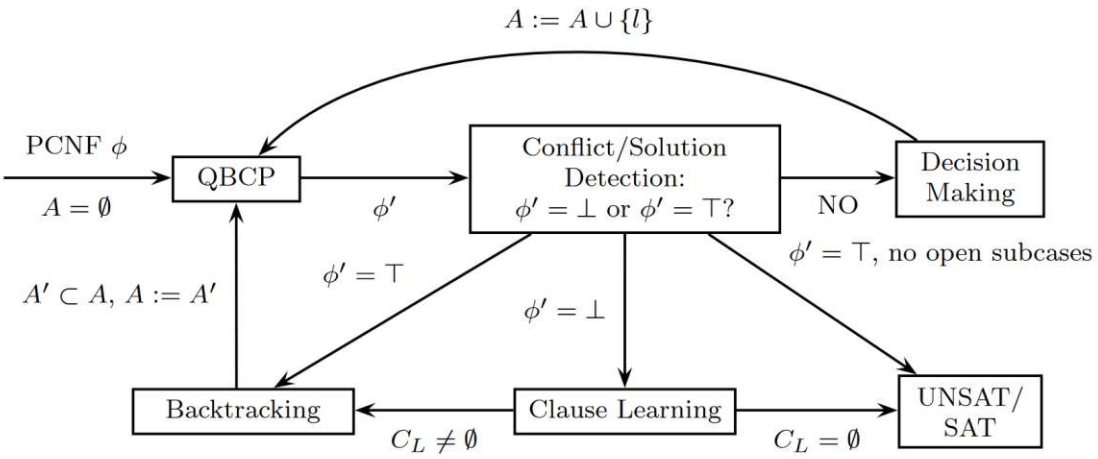

Figure 2.5: Flowchart of QCDCL [BJLS21].

In Figure 2.5, we have the flowchart of a QCDCL-based solver. The solver accepts a PCNF input and applies unit propagation if possible. If the QBCP step produces a conflict, then we can learn a clause, if it hasn't detected a conflict it will make a decision and restart unit propagation. In this work we are only interested in false PCNF, thus when we have a solution we will backtrack and continue the algorithm, only the conflicts will contribute to the refutation.

The scope of the work is not the solver but where the proof is derived, this proof is generated during clause learning, where we are applying the resolution.

2.7 The QDIMACS Format

QDIMACS is a format that is used to write a PCNF in a text file. The files we will use in our work will be composed of the following parts: the line that defines the number of variables and clauses, the quantifier blocks that need to appear in alternating ways, without having two existential or universal lines, and the clauses. Each line of the last two parts must end with a `0`. The variables are denoted by positive integers, and the negated literals are denoted by negative integers. The variables that are defined in available in the domain, but not presented in the clauses, will be considered as existential quantified in a block before the first block of the formula. In Figure 2.6, we present a valid example of a QDIMACS file. We can observe that `6` and `7` are free variables, as they do not appear in the prefix. Additionally, `6` is not even present in the clauses.

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

Figure 2.6: QDIMACS example.

In Figure 2.7 we present an invalid QDIMACS file. In the example we have three errors. One is for having fewer clauses than the number of clauses given at the beginning. Another error is for having two successive blocks of universal quantifiers. The last error is caused by the last line, because the clause does not end with a `0`.

2.8 The QCIR Format

The QCIR format is more general than the QDIMACS. The overall structure of QCIR is depicted in Figure 2.8. As before, we start with the quantifier blocks, but now we can have successive quantifier blocks with the same quantifier. Then, we have to define which

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

Figure 2.7: QDIMACS invalid example.

literal is used as the output of the circuit, and the gate definition of each variable. These gates can be and, or, xor, ite, or another quantified circuit. A prenex quantified Boolean formula example is available in Figure 2.9 in QCIR format.

```
#QCIR-G14
quant(var, ..., var)
...
quant(var, ..., var)
output(lit)
var = gate_exp
...
var = gate_exp
```

Figure 2.8: QCIR format [qci].

```
#QCIR-G14
forall(v1)
exists(v2, v3)
output(g3)
g1 = and(v1, v2)
g2 = and(-v1, -v2, v3)
g3 = or(g1, g2)
```

Figure 2.9: QCIR example from [qci].

$$\forall v_1 \exists v_2 v_3 \underbrace{\underbrace{(v_1 \wedge v_2)}_{g_1} \wedge \underbrace{(\neg v_1 \wedge \neg v_2 \wedge v_3)}_{g_2}}_{g_3}$$

Figure 2.10: Formula for Figure 2.9.

The definition of QCIR is very versatile, but for this work, we can assume that the formulas are in prenex form, and the gates we use are only of the type: and, or. Additionally,

suppose we have n gates in the order $g_1, g_2, ..., g_n$. In our circuits, if a gate g_i uses the output of another gate, this gate must be previously defined, i.e., if g_i uses the gate g_j, then $j < i$.

CHAPTER 3

Certifying Quantified Circuit Reconstruction

In this chapter, we tackle the main objective of the work: certiying quantified circuit reconstruction. Given a QBF ϕ in PCNF and a QCIR converter, our goal is to verify that the output of the converter is equisatisfiable with the given input. Restricting our approach only to the false instances of QBF, in order to check the satisfiability equivalence, we propose the following solution: after we apply the converter, and get the formula ϕ_{QCIR}, we can reconstruct a refutation of ϕ from a refutation of ϕ_{QCIR}. This way, with the input proof construction, we can ensure the soundness of the QCIR converter.

In the following, we assume that the input formula is false, and a QCIR converter gives a circuit that uses variables from the input without the addition of other new variables.

In the first section, we present the detailed steps for the main procedure, followed by sections that prove the soundness of our procedure.

3.1 Certified QCIR Reconstruction

Before presenting the proof construction we need to have a concise definition of what a QDIMACS to QCIR reconstruction program does.

Definition 3.1 (QCIR reconstruction)**.** A ϕ_{QCIR}, in QCIR format, is a **QCIR reconstruction** of ϕ, in PCNF format, if all the assignments σ of ψ_{QCIR} respects:

- if σ is a satisfying assignment of ψ_{QCIR}, then $\psi[\sigma]$ is satisfiable,
- if σ is a falsifying assignment of ψ_{QCIR}, then $\psi[\sigma]$ is unsatisfiable,

where ψ_{QCIR}, ψ are the matrices of ϕ_{QCIR}, ϕ, respectively. The prefix of ϕ_{QCIR} is the prefix of ϕ without the missing variables.

Example 3.2. For PCNF $\phi = \forall xy \exists t(x \vee y \vee \bar{t}) \wedge (\overline{x} \vee t) \wedge (\overline{y} \vee t) \wedge t$, we can have the following QCIR reconstruction $\phi_{\text{QCIR}} = \forall xy \underbrace{(x \vee y)}_{g}$, where $g = (x \vee y)$.

In Table 3.1, we can check that all the assignments for ϕ_{QCIR}'s matrix respects the Definition 3.1 making it a valid QCIR reconstruction for ϕ.

x	y	QCIR matrix $(x \vee y)$	ϕ matrix	Check SAT
0	0	0	$(0 \vee 0 \vee \neg t) \wedge t$	UNSAT
0	1	1	$(\neg 1 \vee t) \wedge t$	SAT $t = 1$
1	0	1	$(\neg 1 \vee t) \wedge t$	SAT $t = 1$
1	1	1	$(\neg 1 \vee t) \wedge (\neg 1 \vee t) \wedge t$	SAT $t = 1$

Table 3.1: Truth table for Example 3.2.

With Definition 3.1 in place, we begin by introducing a certifying method for the output of a QCIR reconstruction program.

Algorithm 1 Procedure for input proof generation from QCIR conversion.

Input: False PCNF ϕ, QCIR converter procedure QcirConv
Output: QRAT refutation P for ϕ

```
1: procedure GetInputProof(φ, QcirConv)
2:     φ_QCIR ← QcirConv(φ)
3:     φ_Tseitin, φ_DNF ← TseitinOfQcir(φ_QCIR)
4:     P_Q-Res ← QBFSolver(φ_Tseitin, φ_DNF)
5:     P_QRAT ← QresToQrat(P_Q-Res)
6:     P_Input-QRAT ← InputQratConstruction(φ, φ_Tseitin, P_QRAT)
7:     return P_Input-QRAT
8: end procedure
```

In Algorithm 1, we present the main procedure for getting a refutation of a given QBF using its circuit reconstruction form. The first step is the application of the QCIR reconstruction on the input QBF and get a circuit QBF ϕ_{QCIR}, respecting Definition 3.1. In the second step, we generate the PCNF of the QBF using Tseitin transformation, this way we introduce a variable for each of the gates, but more details will be presented in a dedicated Section 3.2. In addition to the Tseitin transformation, which produces a CNF of the circuit, we can also produce a DNF of the circuit. In the third step, we apply a QBF solver based on QCDCL on the new formula, and this will produce a Q-Resolution proof for the ϕ_{QCIR} in PCNF. Additionally, we can speed up the QCDCL solver by using the DNF of the circuit for term learning. In the following step, with the Q-Resolution proof we can transform it into a QRAT proof, detailed in Section 3.3. In the last step,

with the input QBF, the QCIR in PCNF, and its QRAT proof, we can derive a QRAT proof for the input formula.

With the proof for the input formula deducted from the QCIR, we can use it as a certification for QCIR conversion, but more will be explained in Section 3.4.

3.2 Tseitin Transformation of QCIR

Algorithm 2 PCNF from QCIR using Tseitin transformation.

Input: QCIR formula ϕ_{QCIR}
Output: Satisfiable equivalent PCNF of input ϕ_{Tseitin}, and PDNF of input ϕ_{DNF}

1: **procedure** TseitinOfQcir(ϕ_{QCIR})
2: $\phi_{\text{Tseitin}} \leftarrow$ empty formula
3: $\phi_{\text{DNF}} \leftarrow$ empty formula
4: **for** gate in ϕ_{QCIR} **do**
5: $\phi_{\text{Tseitin}} \leftarrow \phi_{\text{Tseitin}} \cup$ Tseitin(gate)
6: $\phi_{\text{DNF}} \leftarrow \phi_{\text{DNF}} \cup$ DNF(gate)
7: **end for**
8: $\phi_{\text{Tseitin}} \leftarrow \phi_{\text{Tseitin}} \cup t_{\text{output}}$ ▷ where t_{output} is the output gate of Tseitin encoding
9: $\phi_{\text{DNF}} \leftarrow \phi_{\text{DNF}} \cup t'_{\text{output}}$ ▷ where t'_{output} is the output gate of DNF encoding
10: **return** $\phi_{\text{Tseitin}}, \phi_{\text{DNF}}$
11: **end procedure**

In Algorithm 2, we present the procedure that takes as an input a formula in QCIR and outputs a PCNF that is equisatisfiable with the input. The procedure starts by initializing an empty formula where we will append clauses. Then, for each gate, we introduce a variable t, and write the CNF of the ($t \leftrightarrow$ gate), this is the role of Tseitin function. For simplicity, if we have a gate that takes multiple inputs, we break down the formula for each 2 variables with auxiliary Tseitin variables, for example having the gate $a \wedge b \wedge c \wedge d$ we will use t_1 for $a \wedge b$, then use $t_1 \wedge c$ for t_2, etc. After we translate all the gates, to keep a CNF that is equisatisfiable with a given formula we also need to add the last Tseitin variable as a clause. As for the prefix, the prefix of ϕ_{Tseitin} will be the same as ϕ_{QCIR}, and each Tseitin variable will be added in the innermost existential block. Similar to the construction of the circuit's CNF, we construct ϕ_{DNF} to be the circuit's DNF. To construct this DNF, we will introduce a variable t' for each gate. The equivalence ($t' \leftrightarrow$ gate) is true iff for a set of terms is false. For an or-gate ($t' \leftrightarrow a \vee b$), we have

$$(\neg a\ \neg b\ t')$$
$$(a\ \neg t')$$
$$(b\ \neg t').$$

This set of terms will evaluate to false if the assignment of $\{a, b, t\}$ satisfies ($t' \leftrightarrow a \vee b$). According to [AGS05], we can use $\phi_{\text{DNF}} = \Pi\forall(t'_1, ..., t'_n)\psi_{\text{DNF}} \vee t'_n$, where Π is the prefix

of the circuit, $\forall(t'_1, ..., t'_n)\psi_{\text{DNF}}$ are the variables and the DNF introduced by the DNF transformation, and t'_n will also be the output of the ϕ_{DNF}, for the term learning.

For later use in proving the main result of the transformation, we formulate the Claim 3.3 which states that Procedure 2 is sound, under the assumption that the circuit is false.

Claim 3.3. *Given a QCIR ϕ_{QCIR}, Algorithm 2 produces an equisatisfiable PCNF $\phi_{Tseitin}$.*

Proof. From Definition 2.15 we know that $\forall x(F) = F[0/x] \wedge F[1/x]$, and similarly for existential. We can think of each quantifier as a node in a tree, with 2 children representing the substitution with true and false. The leaves of our tree will be free of quantifiers, and will have an assignment for each quantified variable. If we construct the same tree for the ϕ_{Tseitin}, at the level of the ϕ_{QCIR} tree, we will have the same assignment as before, plus the existential Tseitin variables. But this node will have the same evaluation as the corresponding leaf in ϕ_{QCIR}, by Tseitin transformation. Therefore, ϕ_{QCIR} and ϕ_{Tseitin} are equisatisfiable. □

3.3 QRAT Proof from Q-Resolution Proof

Algorithm 3 Q-resolution to QRAT proof format.

Input: Q-Resolution proof of the ϕ_{Tseitin}: $P_{\text{Q-Res}}$
Output: QRAT proof format from Q-resolution: P_{QRAT}

```
1:  procedure QResToQRat(P_Q-Res)
2:      P_QRAT ← empty list
3:      for line in P_Q-Res do          ▷ line is of the form (resolvent, premise1, premise2)
4:          P_QRAT.append(resolvent)
5:          while resolvent highest level of a variable is universal do
6:              P_QRAT.append(universal reduction)
7:          end while
8:      end for
9:      return P_QRAT
10: end procedure
```

In Algorithm 3 we produce a QRAT proof from a Q-resolution proof. We start by initializing an empty list where we will store each step of the QRAT proof. Then we iterate through all the lines in the Q-resolution proof. Each of these lines contains the resolvent and the premises it came from, but we are only interested in the resolvent, because the premises should already be present at this step of the proof (either a resolvent that was added before or as a clause in the input formula). In a QRAT proof, the QRAT literal is in the first position, thus we need to see what literal we put in the resolvent, by Claim 3.5 it doesn't matter. After that, we need to check if we can apply universal reduction, thus we check the variable that has the highest level and see if it is universal.

In addition, it will be important for our implementation to know what variable is the pivot in the Q-resolution proof. But this can be easily found using the Claim 3.4, which states only one pivot is available between two premises.

Claim 3.4. *Given a proof line from Q-resolution proof that is formed of two premises and their resolvent, only one pivot is available between the premises. Where pivot is an existential variable that appears positively in one clause and negatively in the other.*

Proof. Suppose two literals or more are available as pivots for the resolution rule, having the following premises $C \vee a \vee b$ and $D \vee \overline{a} \vee \overline{b}$ this cannot be the case, because if we apply the resolution rule on one literal the other one will produce a tautology and is not allowed by the rule, a contradiction. Thus, our supposition was false, and between 2 premises there is only one pivot. □

Claim 3.5. *The resolvent is QRAT with respect to any formula that includes the premises.*

Proof. Our premises are of the form $C \vee p$ and $D \vee \overline{p}$. Our resolvent is $C \vee D$. We want to check that $C \vee D$ has QRAT on an arbitrary literal l. Thus, we want to check that we can use implicit unit propagation to derive the outer resolvent of $C \vee D$ and another clause that has $\overline{l}$. But, the outer resolvent includes $C \vee D$ and if we use the negated literals from this clause, we can apply unit propagation on the premises and derive p and $\overline{p}$, from where we can produce $\bot$. Thus, we can apply QRAT on any literal of the resolvent. □

3.4 Input QRAT Proof Construction

Algorithm 4 QRAT proof of a QBF from its QCIR.

Input: A PCNF ϕ, a PCNF ϕ_{Tseitin} that is a QCIR reconstruction of ϕ, QRAT proof P_{QRAT} of ϕ_{Tseitin}
Output: Input QRAT proof $P_{\text{Input-QRAT}}$ for ϕ

1: **procedure** InputQratConstruction($\phi, \phi_{\text{Tseitin}}, P_{\text{QRAT}}$)
2: t_{output} ▷ the Tseitin variable that corresponds to the value of the formula
3: $P_{\text{Input-QRAT}} \leftarrow$ empty list
4: $P_{\text{Input-QRAT}}$.append($\phi_{\text{Tseitin}} \backslash \{t_{\text{output}}\}$) ▷ where t_{output} is the output gate of Tseitin encoding
5: $P_{\text{Input-QRAT}}$.append(SATSolver($\phi, \phi_{\text{Tseitin}} \backslash \{t_{\text{output}}\}$, assume= $\{\neg t_{\text{output}}\}$))
6: $P_{\text{Input-QRAT}}$.append(t_{output})
7: $P_{\text{Input-QRAT}}$.append(P_{QRAT})
8: **return** $P_{\text{Input-QRAT}}$
9: **end procedure**

In Algorithm 4 we present the last function used in the main procedure, which is responsible for the proof generation of the input formula. Firstly, we need to find what

Tseitin variable is set for the QCIR result because it cannot be added trivially in the proof of the input QBF. In the remainder, we will call this variable the output gate.

The idea of this construction comes from the fact that for an unsatisfiable propositional formula in CNF, the refutation of it is provided by a subset of its clauses. Thus, we need to find a way to add the ϕ_{Tseitin} in the proof while keeping the satisfiable equivalence, then we can apply the proof from the ϕ_{Tseitin}.

The first part of the input proof is the appending of the Tseitin variables, without the last output gate. We can do this in a sound manner using Claim 3.6 for and-gates, and similarly prove for or-gates. For now, we can assume the Tseitin variables are introduced only for and/or gates, if not, one can transform a QBF circuit in a form that contains only those 2 gates.

Claim 3.6. *Let ϕ be a QBF. For a new variable t the clauses $t \vee \overline{a} \vee \overline{b}$, $\overline{t} \vee a$ and $\overline{t} \vee b$ can be added subsequently as a QRAT step to ϕ with the QRAT literal t.*

Proof. The first clause, $t \vee \overline{a} \vee \overline{b}$, can be added because t is not found in the formula, thus respecting the QRAT property.

Due to t having the highest level, this will make the literals a and $\overline{a}$ to appear in the outer resolvent of $\overline{t} \vee a$ and $t \vee \overline{a} \vee \overline{b}$. This outer resolvent is contradictory, thus the clause $\overline{t} \vee a$ is QRAT on t.

Similarly, $\overline{t} \vee b$ can be added to the proof. □

With the added Tseitin clauses to the proof, it remains to add the output gate. This cannot be added trivially as in previous clauses, and it will be needed the use of a SAT solver to produce the QRAT steps. Using Claim 3.7, we can always have a proof for the output gate if ϕ_{QCIR} is the QCIR reconstruction of the input ϕ. Calling a SAT solver, we can generate a proof in the RUP proof system. This proof can be appended to our proof because a clause that has RUP also has QRAT on any literal due to outer resolvent containing the clause that has RUP, thus producing the $\bot$ using unit propagation.

Claim 3.7. *Let ϕ be a PCNF, and PCNF $\phi_{Tseitin}$ that is the CNF encoding of the QCIR reconstruction ϕ_{QCIR} of ϕ. We have $\psi, (\psi_{Tseitin} \backslash \{t_{output}\}) \vDash t_{output}$, where t_{output} is the output gate of $\phi_{Tseitin}$, and $\psi, \psi_{Tseitin}$ are the matrices for $\phi, \phi_{Tseitin}$, respectively.*

Proof. Suppose there is a satisfying assignment σ for ψ, that makes t_{output} false. From the QCIR reconstruction Definition 3.1 we have σ is also a satisfying assignment for ϕ_{QCIR}, where ϕ_{QCIR} is the circuit of ϕ_{Tseitin}. But, $t_{\text{output}} \leftrightarrow \phi_{\text{QCIR}}$ that translates to $0 \leftrightarrow 1$, contradiction. Thus, $\psi, (\psi_{\text{Tseitin}} \backslash \{t_{\text{output}}\}) \vDash t_{\text{output}}$. □

The last part that needs to be added is the proof of the added ϕ_{Tseitin}. Having already the clauses from ϕ_{Tseitin}, we can append the proof and solve this subset to get the empty clause to derive the refutation. The QRAT proof checker when checking for the refutation

will skip the redundancy checking for deletion line, thus after the ϕ_{Tseitin} appending to the output we can delete the lines from the ϕ and making the proof more clearly on this subset of clauses, ϕ_{Tseitin}.

Finally, with Procedure 1 we can derive an input proof, this proof must be a valid proof for the input QBF if it has been derived from the QCIR reconstruction. Thus, we have a way of testing circuit conversion on different QBFs and certifying its reconstruction.

CHAPTER 4

Implementation

This chapter is dedicated to describing the implementation steps for the procedures in Chapter 3 with the respective auxiliary information, the usage of the already existent tools like QBF solver, checker, and the helper scripts used for the experiment.

4.1 Existing Tools

The first tool needed in our process is certainly a QBF solver. The solver we have chosen to use is MiniQU [Sli22]. This solver is based on QCDCL, however, instead of producing the proof in the Q-resolution proof system, it is using a variant of it named QU-resolution. The difference between QU-resolution and its former is its ability to apply the resolution rule even on a universal quantified variable [BCJ19]. Fortunately, the use of this new proof system does not prevent us from using the same reasoning. We can keep the same approach because in QU-resolution we still have the condition of not producing tautologies in the resolvent, which our Claims 3.4 and 3.5 are based on. The input formats accepted by the solver are the QDIMACS and QCIR, but for our usage, we will only be interested in the QDIMACS format. As for the output, each line of the proof trace has the form shown in Figure 4.1. The `ID` it's an identifier used by the solver to know which clause it is referring to, followed by the `TYPE` of the formula. For the type, we are only interested in types that are equal to 0, because 0 stands for a clause, the other type for formula is a cube, but cubes are used for the satisfiable case, and for our scope we are going to use only the clauses for refutation. In `LIT`$\star$ we are expressing our clause by enumerating its literals, the $\star$ is used to denote that we can have zero or more literals. In the end, we write the premises needed to derive the resolvent. Here, `PREMISE_ID`$\star$ is given in a compact form, meaning that, we apply resolution on the first two, then apply it on the intermediate resolvent with the next premise, and so on. An example of this repeated application of resolution can be seen in Figure 4.2, where n is the number of premises, and R_{ID} is the clause found in `LIT`$\star$.

```
ID TYPE LIT* 0 PREMISE_ID* 0
```

Figure 4.1: MiniQU output format.

$$\dfrac{\dfrac{\dfrac{P_1 \quad P_2}{R_1}\,\text{res} \qquad P_3}{\vdots}\,\text{res} \qquad P_n}{R_{\text{ID}}}\,\text{res}$$

Figure 4.2: Resolution from MiniQU output.

With the proof from a solver, we want to transform it into a QRAT proof. For this task, we are going to use a Python program. This program, ToQRAT, takes as an input of a similar form as the MiniQU output, given in Figure 4.1, and produces a translation to a QRAT proof. The procedure that is implemented in the program is exactly the one explained in Algorithm 3. Besides the procedure, the program also included the code needed for parsing the formula and throwing errors in case the transformation was not feasible.

The circuit reconstruction program we want to test is `qcir-conv` [JKS16]. As illustrated in [JKS16], this convertor is looking for patterns in QDIMACS. If a pattern is found, then it can use a gate variable in the new QCIR instead of a bounded variable in QDIMACS. An important flag we need to add to the invocation of the program is `-keep-varnames`. This flag adds in the comments of the QCIR the mapping of the new circuit variables to the old variable in the QDIMACS.

As for the proof checker, we will be using `qrat-trim` presented alongside the QRAT proof system in [HSB14]. This program accepts as inputs a QDIMACS and a proof in QRAT, and by default, it will check for refutation, there is also a flag that will check the proof for satisfiability. The checker does not verify each line and check if the operation is sound for QRAT, instead, it starts from an empty clause and starts checking the lines in reverse order and takes only the lines it needs for the proof. If the proof is a valid refutation of the input it will print the word `VERIFIED`. An important note to be added, in case we make an automated system that checks the exit code of the program to be 0 this code can also be found while the program has a mismatch of the clauses (a miss match can occur when we delete a clause that is not present).

4.2 Procedures Implementation

With the existing tools present, we know need to implement what is missing from the process. The procedures that are missing are the Tseitin transformation of QCIR, Algorithm 2, and the input QRAT proof construction, Algorithm 4. We chose to implement both of them in Python.

For the circuit form to conjunctive normal form, we take a QCIR as input and output

an equisatisfiable QDIMACS. This QCIR besides the usual definition also contains the comments with the input variable notation which we need to take into account. Looping through the gates we introduce a Tseitin variable for each gate. Our input is a form where the only used gates are and-gates and or-gates. For example, given a gate of form `and(2, 4, 6)`, we will introduce a variable `t_1 = and(2, 4)` and `t_2 = and(t_1, 6)`, where `t_2` is the variable for the gate. For an or-gate a similar approach can be employed. For the naming variables in the new QDIMACS we will rename each variable in the QCIR to $\{1, 2, ..., \text{number of QCIR variables}\}$, and the Tseitin variables to $\{\text{number of QCIR variables} + 1, ..., \text{last Tseitin gate}\}$. For the construction of the QDIMACS file, we will append at the beginning of the file the comments for the translation of our variables to the input variables, and also mark the beginning of the Tseitin variables. We also need to append the Tseitin variable of the circuit output gate to the formula to keep the satisfiablility equivalence between input and output.

We improved the last procedure by using the extended version of the Tseitin transformation. Using the extended form, for a gate in the QCIR format we are introducing only one Tseitin variable for it, therefore, reducing the number of clauses used in the CNF encoding of the circuit. We will call this extended version the long encoding, while the previous one will be called the short encoding.

The final step in our process is the input proof construction. For this program we take as inputs: the input QDIMACS, the Tseitin converted circuit QDIMACS produced from the QCIR version of the input QBF, and a QRAT proof for the circuit QDIMACS. For the output, we want a QRAT proof for the input QDIMACS. To construct the required QRAT proof, we start by appending the lines from the Tseitin converted circuit QDIMACS to the proof, without the variable for the output, according to the mapping to the input QDIMACS variable naming. The Tseitin variables will start from the `(input number of variables + 1)`. For testing purposes we also make sure that the output gate is mapped to the first element after the variables, i.e., `(input number of variables + 1)`. In order to successfully append the output Tseitin variable to the proof, we need to derive the proof from the input QDIMACS clauses and the Tseitin clauses from circuit QDIMACS. Thus, we will make a call to a SAT solver with these clauses, and negation of the output variable. The output of this SAT solver call will produce a proof that we can append to the input proof we want to construct, and not forget to add the output gate variable to this proof too. In case the previous call is satisfiable, our program will stop its execution because it means the reconstruction was not valid. The last step of this procedure is the addition of the QRAT proof from the input with the mapping to the input variables.

4.3 Workflow Scripts

Having all of these tools we want to automate the process of certifying a circuit reconstruction program. The program we will want to check is `qcir-conv` [JKS16]. Thus, we wrote a Bash script that takes as an input a QDIMACS, applies this conversion, generates

the input proof according to Procedure 1, and lastly, uses the `qrat-trim` [HSB14] to check if the produced proof is a valid proof for the input QDIMACS, thus certifying the conversion. To pass information from one step to the next step, we will save the output of each step in a temporary file. This script is described in the flowchart form in Figure 4.3, where we have the steps used for the input proof construction, and in Figure 4.4, where we check if the proof can be used as a valid certification of the circuit conversion.

In Figure 4.3, we can also have the possibility to derive a DNF form from QCIR for the QBF solver. This DNF is not used in the proof generation, but it can speed up the QBF solver that uses term learning. Term learning helps by pruning the satisfiable search space [GNT06].

With all procedures defined, we will be testing their functionality and their application for certification in the following Chapter 5.

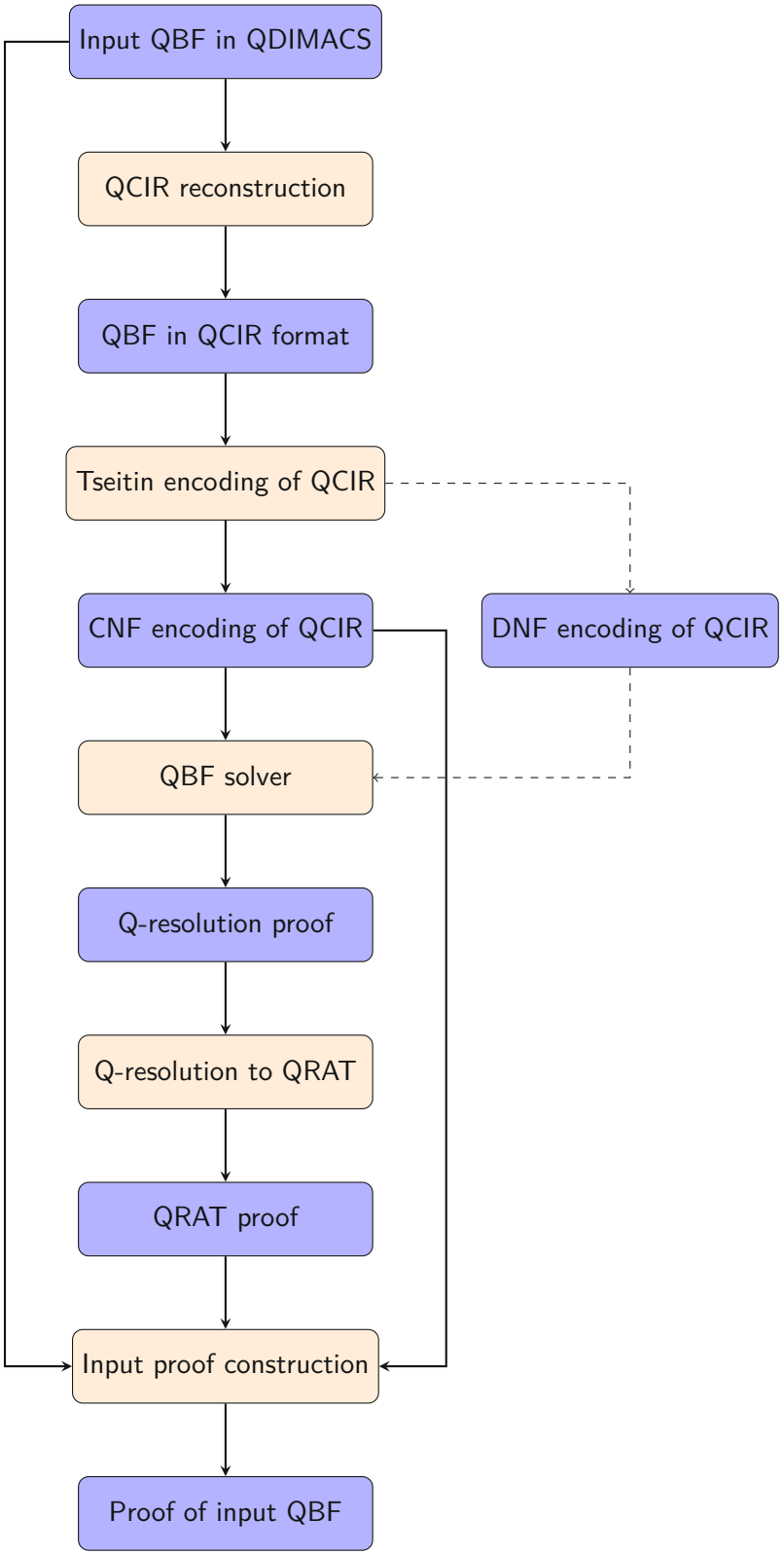

Figure 4.3: Proof generation flowchart.

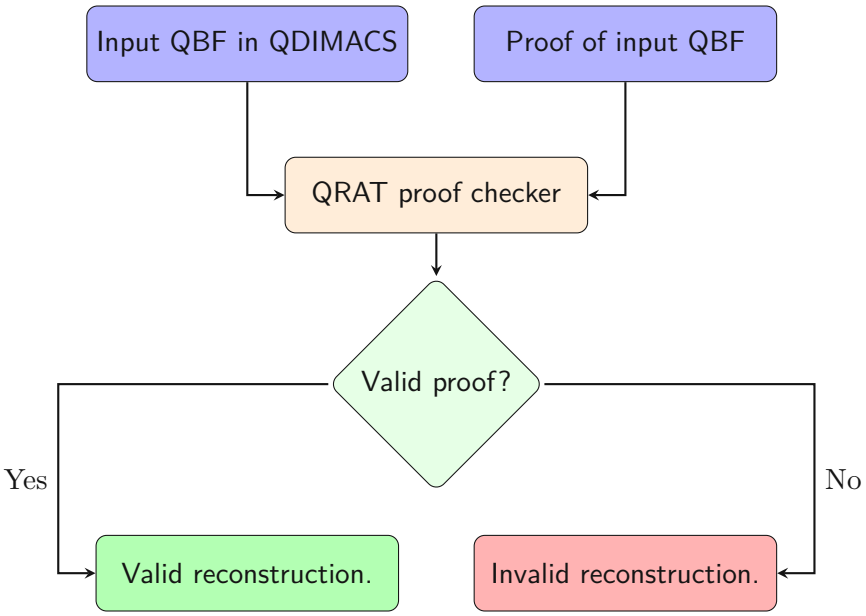

Figure 4.4: QCIR reconstruction certification flowchart.

CHAPTER 5

Experiments

In this chapter, we will present the methods we used to check a circuit reconstruction and the correctness of our implementation. The first section is dedicated to manually testing our procedures, and verifying the output of each step. The second and last sections are assigned for certifying the transformation from QDIMACS to QCIR, using randomly generated tests and common benchmarks.

5.1 Initial Testing

The basic test we will use to check our implementation is the QBF

$$\forall x_1 x_2 \exists y_3 (x_1 \vee x_2 \vee \overline{y_3}) \wedge (\overline{x_1} \vee y_3) \wedge (\overline{x_2} \vee y_3) \wedge y_3,$$

with QDIMACS format in Figure 5.1.

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

Figure 5.1: Single or-gate example in QDIMACS.

The first step we need to do is the application of the circuit reconstruction program to the QBF. The output is presented in Figure 5.2. We can see that the variable y_3 corresponds to an or gate definition, thus excluding it from the input variables. The last clause y_3 in QBF, in the QCIR maps to the circuit output. Furthermore, at the beginning of the file,

```
#QCIR-G14

#VarName   2 : v1
#VarName   4 : v2

forall(2, 4)
output(102)

102 = or(2, 4)
```

Figure 5.2: QCIR reconstruction of Figure 5.1.

we can notice the `VarName` introduction, where the first index corresponds to the QCIR variable and the second number denotes the input variable in the QDIMACS.

The second step is the transformation of QCIR to QDIMACS by introducing the Tseitin variables for each gate. Presented in Figure 5.3, the comment lines, starting with `c`, are the extra information we need in the reconstruction. The pair of new variables with the input variable, and the starting position of the Tseitin variables. The next line specifies the number of variables and clauses. Followed by the quantifier blocks, and the clauses. The last clause will always be the variable that corresponds to the output Tseitin variable.

```
c VarOld 1 : 1
c VarOld 2 : 2
c TseitinStart 3
p cnf 3 4
a 1 2 0
e 3 0
1 2 -3 0
-1 3 0
-2 3 0
3 0
```

Figure 5.3: PCNF encoding of the QCIR.

In the third step, we use the QBF solver to get a solution for the previous step's output. In the left part of Figure 5.4, we have the trace generated by the solver in the Q-resolution proof system, the line denotes the application of the resolution rule on the first and fourth clause and gets the empty clause, with the use of universal reduction rule. In the following step, we translate the Q-resolution proof to the QRAT proof on the right of Figure 5.4. Checking each line of the QRAT translation, we can see that it starts with the first clause in the premises and the next line is the resolvent. The last two lines are the application of the universal reduction for the QRAT. The addition of the empty clause in the QRAT proof is optional, because the checker when checking for a refutation already starts from the empty clause, and reads the QRAT proof backward.

Q-resolution	QRAT
5 0 0 1 4 0	1 2 -3 0 1 2 0 u 2 1 0 u 1 0

Figure 5.4: Q-resolution proof of the QBF in Figure 5.3, left. QRAT proof translation, right.

The last step is the QRAT proof generation for the input QDIMACS. With input QDIMACS, its PCNF form of the QCIR, and the corresponding proof of the CNF in QRAT. Figure 5.5 presents the proof for the current example. This proof is composed of 3 parts: the introduction of the Tseitin variables, in color blue, which we can introduce using Claim 3.6, then we use an SAT solver call to produce a proof for the addition of the output variable gate, in color yellow with last line being the output, finally, we append the QRAT proof, in color red, this is possible because we introduced in an equisatisfiable way the clauses needed to use this proof from the input, and we get a proof for the input QDIMACS.

```
-4 2 1 0
4 -1 0
4 -2 0
4 0
4 0
1 2 -4 0
1 2 0
u 2 1 0
u 1 0
```

Figure 5.5: Input QRAT proof.

With the produce input proof, we can notice that the introduction of the output gate was trivial, and if we use a checker we can see that we can use only the QRAT from the input to check the input formula. Thus, we can ask: *do we really need the SAT call to introduce the output gate?* And, *Is the input proof enough for the input QDIMACS?*

In order to answer these questions we propose the following methods. Firstly, we will manually check if the input QRAT is enough. Secondly, we check if we don't need the proof generated by the SAT. As the test is not relevant, we check these methods, and we will see that it will fail, indicating that all the steps are needed for the final output.

One interesting error that was made in the first implementation of the program, was the wrong ordering of the Tseitin translation, not having the QRAT variable on the first position. As we didn't know from Figure 5.5 which part had the error, the translation, or the introduction of the input gate, we manually included the output gate as an assumption in the input QBF, this is the reason we made the output gate to be the first available

variable to be easier to be included in the QBF, and use the checker, and saw that it is still failing.

For the following sections, with the implementation we have, assuming we don't have any error, we can use our procedure to test `qcir-conv` [JKS16] using different false QBFs as input and check that it is successfully producing QRAT refutations.

5.2 Random Testing

For random testing, we will use the script we made that calls the random QBF generator QBFuzz [BLB10]. This random generator accepts as parameters the number of variables and the number of clauses for the QBF.

Number of variables	Number of clauses	Number of tests	Number of failed tests
5	10	58	8
50	200	141	4
80	350	70	1
150	500	80	1
300	1000	84	2

Table 5.1: Benchmark of qcir-conv [JKS16] using randomly generated QBFs and MiniQU [Sli22] solver.

In Table 5.1, we have the columns for the random generator, and then we generate a number of unsatisfiable instances. With the unsatisfiable QBF, we run them through the script to get the input proof. As we can see in the results Table 5.1, we have failed tests. Upon examining these tests, all the errors come from the step where we transform a Q-resolution proof to QRAT proof because the Q-resolution proof uses a clause `ID` that is already in the input, making it a problem that came from the QBF solver. Using another solver this problem is fixed. An explanation for why this can happen is that the MiniQU [Sli22] is simplifying some duplicate lines making the formula shorter and using the `ID` of already existing clauses in the input file. Therefore, we will repeat the same experiment using a simple QBF solver that is also based on QCDCL, and the output has the same format, but the proof system is Q-resolution. We will call this solver the Basic QCDCL solver. In the case of the Basic QCDCL solver presented in Table 5.2, the reason for failed tests is the timeout of the solver. Additionally, for the Basic QCDCL solver, we can extend it to accept a DNF to speed up the search with term learning.

Besides `qcir-conv` [JKS16], we can also use a different for testing circuit reconstruction. `Unique` [Sli20] is an interpolation-based circuit reconstruction. Because MiniQU [Sli22] produces wrong proofs using our pipeline, we will use only the Basic QCDCL solver when we test `unique`. In the experiment presented in Figure 5.3 we have again only failed instances due to timeout.

Number of variables	Number of clauses	Number of tests	Number of failed tests
5	10	56	0
50	200	81	0
80	350	172	1
150	500	170	6
300	1000	168	2

Table 5.2: Benchmark of qcir-conv [JKS16] using randomly generated QBFs and Basic QCDCL solver.

Number of variables	Number of clauses	Number of tests	Number of failed tests
5	10	109	0
50	200	165	0
80	350	173	0
150	500	169	5
300	1000	170	0

Table 5.3: Benchmark of unique [Sli20] using randomly generated QBFs and Basic QCDCL solver.

5.3 QBF Benchmarks

As random generating may miss some edge cases, in this section we will use various tests from the QBF solving competitions. In this experiment, we will use inputs used in the 2017, 2019, and 2022 competitions. As these tests are quite large and hard to solve, we will select a part of them. For this selection, we will use the MiniQU [Sli22] solver to check if the QBF is unsatisfiable, and also we will set a timeout of 2 seconds for the solver. We will use this timeout for sorting the tests because the CNF encoding will also grow the formula. The number of tests used from each year is displayed in Figure 5.4.

QBF benchmark	Number of tests	Verified instances with 10 seconds timeout
2022	14	14
2019	52	2
2017	8	0

Table 5.4: Benchmark of qcir-conv [JKS16] using inputs from QBF competition - long encoding with timeout 10 seconds.

For the first attempt displayed in Table 5.4, where we set a timeout of 10 seconds on our verification script. For the successfully finished tests, all the input proofs are valid for their inputs. As we implemented a long version of CNF encoding for each gate variable, slow verification is expected. Most of the time for the tests that timeout is spent in the

SAT solver call. One reason is that the input of the SAT call is quite large, including the Tseitin clauses and the input clauses.

QBF benchmark	Number of tests	Verified instances with 20 seconds timeout
2022	14	14
2019	52	6
2017	8	0

Table 5.5: Benchmark of qcir-conv [JKS16] using inputs from QBF competition - long encoding with timeout 20 seconds.

In the second attempt, Table 5.5, we doubled the time used for a timeout in order to give the SAT call more time to the instance. From the results, we can observe that we had an increase in the number of solved instances. It is important to notice that we haven't got a test that failed by having a satisfiable instance when we derive the proof for the output gate or an invalid QRAT proof for the input instance, therefore, we can assume that with sufficient computational power and enough time, the SAT call will find a proof for the remaining tests.

QBF benchmark	Number of tests	Verified instances with 20 seconds timeout
2022	14	14
2019	52	21
2017	8	0

Table 5.6: Benchmark using inputs from QBF competition - short encoding with timeout 20 seconds.

Lastly, in Table 5.6 we experiment using the short encoding of the Tseitin transformation, instead of adding a variable for each operation, we add the variable only for the gate. In this attempt, we achieved a massive increase in the solved instances from 2019.

QBF benchmark	Number of tests	Verified instances qcir-conv	Verified instances qcir-conv with term learning	Verified instances unique
2022	14	14	14	11
2019	52	19	38	10
2017	8	1	2	1

Table 5.7: Benchmark of qcir-conv [JKS16] and unique [Sli20] for inputs from QBF competition with 5 minutes timeout.

In Table 5.7, we present a run where we check our certified procedure with longer timeout on QBF competition benchmarks, with two circuits convertors, `qcir-conv` [JKS16] and

`unique` [Sli20]. Due to the use of `unique`, we will need to use the Basic QCDCL solver. In this evaluation, we find that `unique` cannot convert two instances. In the rest of the failed cases, the checking was not finished due to the QBF solver or the SAT solver being stopped after the timeout. Moreover, if we use the Basic QCDCL solver with term learning, we can see that it has a positive effect on the speed of the verification.

With all the experiments done, besides the failed instances that we already discussed, we can assume our implementation can be used as a valid certification for a QCIR reconstruction.

CHAPTER 6

Conclusion

In this chapter, we present the conclusion of this work. The aim of the thesis was to provide a way of certifying quantified circuit reconstruction for QBF. In order to achieve this goal we broke it into two parts. Firstly, we presented a procedure that given a QBF in conjunctive normal form, and a circuit reconstruction program, produces a proof for the QBF input from its reconstruction. Secondly, we show that the ability to produce the proof for the input formula following the procedure is possible if the program of circuit reconstruction respects our definition of circuit, thus serving us as a certification of the transformation. Finally, using Claims 3.3 and 3.6, we provide a method for generating a proof for a QCIR that can be used by a proof checker to verify the answer of the QCIR solver.

On the practical side, we implemented it and tested it on circuit reconstruction programs. We wrote exactly the steps used in the theory part in order to produce an input proof. For the experiment, given this input proof and the original PCNF, we run them through a proof checker to see if the proof really is a refutation of the QBF. If we find a test that produces an input proof that cannot verify the refutation of the input or a failed step in the proof generation procedure, based on the defined claims in the theory, then we can claim the circuit reconstruction failed. Otherwise, we can conclude that the program for circuit reconstruction works as intended.

For future work, we are planning to formally verify our implementation of the pipeline. This way we can be sure that we don't have any errors in our implementation. A possibility of achieving this goal is by the usage of theorem provers. Similar work has been done in [CFHH+17], where a proof checker was developed in Coq, and also in [BNAH23], where Lean4 was used in an analogous way for certification of model counting. Furthermore, we want to extend our procedures to proofs of true QBF.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AGS05] Carlos Ansótegui, Carla P Gomes, and Bart Selman. The achilles' heel of qbf. In *AAAI*, volume 2, pages 2–1, 2005.

[BCJ19] Olaf Beyersdorff, Leroy Chew, and Mikoláš Janota. New resolution-based qbf calculi and their proof complexity. *ACM Transactions on Computation Theory (TOCT)*, 11(4):1–42, 2019.

[BJLS21] Olaf Beyersdorff, Mikoláš Janota, Florian Lonsing, and Martina Seidl. Quantified boolean formulas. In *Handbook of Satisfiability*, pages 1177–1221. IOS Press, 2021.

[BKF95] Hans Kleine Buning, Marek Karpinski, and Andreas Flogel. Resolution for quantified boolean formulas. *Information and computation*, 117(1):12–18, 1995.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010: 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 13*, pages 44–57. Springer, 2010.

[BNAH23] Randal E Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn JH Heule. Certified knowledge compilation with application to verified model counting. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[CFHH+17] Luís Cruz-Filipe, Marijn JH Heule, Warren A Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 220–236. Springer, 2017.

[CH22] Leroy Chew and Marijn JH Heule. Relating existing powerful proof systems for qbf. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[Coo23] Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. 2023.

[GNT06] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006.

[HHW13] Marijn JH Heule, Warren A Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *2013 Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[HSB14] Marijn JH Heule, Martina Seidl, and Armin Biere. A unified proof system for qbf preprocessing. In *International Joint Conference on Automated Reasoning*, pages 91–106. Springer, 2014.

[JKS16] Charles Jordan, Will Klieber, and Martina Seidl. Non-cnf qbf solving with qcir. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[MS08] Joao Marques-Silva. Practical applications of boolean satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008.

[MVVdB16] Wannes Meert, Jonas Vlasselaer, and Guy Van den Broeck. A relaxed tseitin transformation for weighted model counting. In *Proceedings of the Sixth International Workshop on Statistical Relational AI (StarAI)*, pages 1–7, 2016.

[qci] QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas. *QBF Gallery 2014*.

[SBPS19] Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified boolean formulas. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 78–84. IEEE, 2019.

[Sli20] Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to qbf preprocessing. In *International Conference on Computer Aided Verification*, pages 508–528. Springer, 2020.

[Sli22] Friedrich Slivovsky. Quantified cdcl with universal resolution. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

[Was22] Anita Wasilewska. Lecture notes in cse 541: Logic for computer science. Stony Brook University, 2022.

[Zel19] Aleksandar Zeljić. Lecture notes in cs 357: Advanced topics in formal methods. Stanford University, 2019.

[ZM02] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 442–449, 2002.