Informatics

Improving Synthesis of Skolem Functions and Boolean Circuits

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Franz-Xaver Reichl
Matrikelnummer 01356186

an der Fakultät für Informatik

der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

Christoph Scholl	Jie-Hong Roland Jiang

Wien, 28. November 2024

Franz-Xaver Reichl

Informatics

Improving Synthesis of Skolem Functions and Boolean Circuits

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Franz-Xaver Reichl
Registration Number 01356186

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Friedrich Slivovsky
Second advisor: Prof. Dr. Stefan Szeider

The dissertation has been reviewed by:

Christoph Scholl	Jie-Hong Roland Jiang

Vienna, November 28, 2024

Franz-Xaver Reichl

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Franz-Xaver Reichl

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, 28. November 2024

Franz-Xaver Reichl

Acknowledgements

First and foremost, I would like to thank my supervisor Friedrich Slivovsky for his guidance and his advices, and for introducing me to the world of (D)QBF. Even though, due to the pandemic the conditions for starting a PhD were not optimal, Friedrich's support helped me to make first steps in the academic world. I really enjoyed working with you, thanks! I am also very glad that I had Stefan Szeider as my second advisor, thanks a lot for your support. Moreover, I would like to thank my reviewers Christoph Scholl and Jie-Hong Roland Jiang for their detailed and valuable feedback.

Special thanks also go to my colleagues at the Algorithms and Complexity group. In particular, I would like to thank Doris Brazda for helping to shoulder all the bureaucratic burdens. Moreover, I would like to thank Alexis, Hai, Johannes and Vaidyanathan for having lots of great lunch and coffee breaks. Further, I would like to thank Anna Prianichnikova for her support in the LogiCS doctoral college.

I would also like to thank my parents for their continuous support. In particular, I would like to thank my mother for always encouraging my interest in math. Moreover, I also want to thank my grandparents for their part in a childhood I happily look back at.

The work presented in this thesis was funded by the Vienna Science and Technology Fund (WWTF) (10.47379/ICT19060) and the Austrian Science Fund (FWF) (10.55776/W1255).

Kurzfassung

Vom Standpunkt der Komplexitätstheorie geht man davon aus, dass es für das Erfüllbarkeitsproblem der Aussagenlogik (SAT) keinen effizienten Algorithmus gibt. Nichtsdestotrotz können moderne Entscheidungsprozeduren für SAT Formeln mit Millionen von Variablen und Klauseln lösen. Der Erfolg dieser Prozeduren motiviert die Arbeit an Entscheidungsprozeduren für stärkere Logiken, wie abhängigkeitsquantifizierte boolsche Formeln (DQBF). Aufgrund der effizienten Entscheidungsprozeduren für das SAT-Problem wurden viele verschiedene Probleme mittels einer Kodierung in Aussagenlogik gelöst. Dies umfasst im Speziellen das Problem der Berechnung von kleinen Schaltkreisen.

Eine Entscheidungsprozedur für DQBF DQBF erweitern die Aussagenlogik mit universellen und existenziellen Quantoren über Wahrheitswerte. Im Gegensatz zu quantifizierten boolschen Formeln (QBF) werden in DQBF die Abhängigkeiten von existentiellen Variablen explizit angegeben und nicht implizit durch die Ordnung der Quantoren bestimmt. Wir präsentieren einen neuen Algorithmus für das Erfüllbarkeitsproblem von DQBF, der direkt mit Skolemfunktionen arbeitet. Der Algorithmus nutzt Definitionen im Sinne der Aussagenlogik, um eindeutig bestimmte Skolemfunktionen zu berechnen. Des Weiteren nutzt der Algorithmus durch Gegenbeispiele geleitete induktive Synthese (CEGIS), um die Skolemfunktionen der verbleibenden Variablen zu adaptieren. Dadurch kann der Algorithmus für erfüllbare DQBF Skolemfunktionen ohne Mehraufwand bestimmen. Wir werden zeigen, dass der Algorithmus tatsächlich das Entscheidungsproblem für DQBF löst. Dieser Algorithmus wurde in einem Programm names PEDANT implementiert. In einer experimentellen Evaluierung werden wir zeigen, dass PEDANT für Standardbenchmarks mehr Formeln als andere aktuelle Entscheidungsprozeduren lösen kann.

Minimierung von Schaltkreisen Es gibt keine effizienten Algorithmen, um kleinstmögliche Schaltkreise für eine bestimmte boolsche Funktion zu bestimmen. Aus diesem Grund müssen heuristische Methoden angewandt werden, um kleine Schaltkreise zu berechnen. Wir schlagen eine neue Methode zum Minimieren von Schaltkreisen vor. Diese Methode ist eine SAT-basierte lokale Verbesserungsmethode (SLIM). Um einen Schaltkreis zu verkleinern, werden iterativ Teilschaltkreise eines gegebenen Schaltkreises durch kleinstmögliche Schaltkreise ersetzt. Die für die Ersetzungen verwendeten Schaltkreise werden dabei entweder mittels einer SAT- oder einer QBF-Kodierung ermittelt. Hierfür ist es möglich, sowohl Teilschaltkreise mit einem einzelnen Ausgang als auch Schaltkreise mit mehreren Ausgängen zu betrachten. Um eine möglichsts große Freiheit für das Auffinden einer Ersetzung zu erhalten, werden *don't cares* des Teilschaltkreises berücksichtigt. Diese Minimierungsmethode wurde in einem Programm namens eSLIM implementiert. Basierend auf einer experimentellen Evaluierung zeigen wir, dass eSLIM zusammen mit dem Standardprogramm für die Schaltkreissynthese ABC bessere Ergebnisse liefert als ABC alleine. Schlussendlich zeigen wir auch, dass mit eSLIM viele der aktuell kleinsten Schaltkreise der EPFL Benchmarkmenge weiter verkleinert werden können.

Abstract

While the propositional satisfiability problem SAT is intractable, from a theoretical perspective, modern SAT solvers can still handle formulas with millions of variables and clauses. This success of SAT solving motivates research on decision procedures for even stronger logics like *Dependency Quantified Boolean Formulas* (DQBF). To make use of the capabilities of modern SAT solvers, various different problems have been encoded in propositional logic. In particular, this includes the problem of finding small circuits.

Deciding DQBF DQBF extend propositional logic by universal and existential quantification over truth values. Unlike to Quantified Boolean Formulas (QBF), in DQBF, dependencies of existentially quantified variables are explicitly stated and not implicitly determined by the ordering of quantifiers. We propose a new decision procedure for DQBF that directly reasons at the level of Skolem functions. The procedure makes use of propositional definitions to extract uniquely defined Skolem functions and counterexample guided inductive synthesis (CEGIS) to refine Skolem functions for the remaining variables. This allows to derive Skolem functions for satisfiable DQBF with no overhead. We will proof that the proposed algorithm is indeed a decision procedure for DQBF. Moreover, in an experimental evaluation, we will show that our solver Pedant that implements the proposed algorithm, surpassed the performance of state-of-the-art DQBF solvers on standard benchmarks.

Circuit Minimization Determining provably minimum size circuits computing a given Boolean function is computationally intractable. For this reason heuristic, methods need to be considered for obtaining small circuits. We propose a new circuit minimization approach that belongs to the *SAT-based Local Improvement Method* (SLIM) framework. In this approach, we incrementally replace subcircuits of a given circuit by minimum size replacement circuits. This is achieved by either making use of a QBF or of a SAT encoding. The presented method allows to consider both single- and multi-output subcircuits. Additionally, it makes use of the full implementational freedom for determining a replacement circuit by taking don't cares into account. We implemented this minimization method in the tool eSLIM. In an experimental evaluation, we show that using eSLIM together with the state-of-the-art synthesis tool ABC results in significantly smaller circuits compared to applying ABC alone. Moreover, we used eSLIM to further improve the current best solutions for the EPFL benchmark set.

Contents

Preface

Deciding satisfiability of propositional logic (SAT) is the prototypical NP-complete problem [Coo71]. As such, from a theoretical point of view, the SAT problem is considered to be intractable. Nevertheless, SAT-solvers show ever improving performance [HJS19; Fro+21], and modern SAT solvers can handle formulas with millions of variables [MLM21]. This motivates research on decision procedures for the satisfiability problem of logics with an even higher complexity, like Dependency Quantified Boolean Formulas (DQBF) [PR79]. While deciding satisfiability of DQBF is harder than deciding satisfiability of propositional logic, the higher expressiveness of DQBF might outweigh the slower decision procedures for practical applications. We will present a new decision procedure for DQBF in Part I.

Several applications of DQBF were proposed in recent years. One of the most prominent applications is partial equivalence checking (PEC) [Git+13b; BCJ14a]. In PEC we are given a circuit containing unspecified parts and a specification for the circuit. The goal is to check whether there are implementations for the missing parts such that the circuit satisfies the specification. If such implementations exist, it is also of interest to find some implementations. Deciding PEC can be done by encoding the problem into DQBF and by checking the satisfiability of the encoding. Moreover, implementations for the unspecified parts can be obtained from a model of the encoding.

Partial equivalence checking can be applied in the integrated circuit engineering change order (ECO) problem [JKL20]. In ECO an already optimized circuit is given, whose specification is changed. Now the task is to modify the given circuit such that the new specification is satisfied. The ECO problem is of practical relevance, as it can for example be used to fix functional errors in an already optimized circuit design without the need of computing a new design from scratch. By cutting out subcircuits from the given implementation, we can make use of PEC in order to check if there are implementations for the now missing parts of the circuit such that the new specification is satisfied. Of course, we are not only interested if such implementations exist, but we also want to obtain some implementations. As discussed above, the missing parts in PEC can be obtained from a model of a DQBF encoding.

As the given circuit is in general already optimized, we do not want to use any circuit for realizing the modifications, but we want to use already optimized ones. One metric for optimizing circuits that can be of interest is the circuit size. This is a motivation for the second part of this thesis, which is concerned with reducing the size of circuits.

For computing circuits with a small size the efficiency of modern SAT solvers can be harnessed [Haa+20; KPS22]. We propose a new SAT/QBF-based circuit minimization method that improves circuits by rewriting subcircuits in Part II. This method allows handling single-output as well as multi-output subcircuits. Additionally, it exploits the full implementational freedom of subcircuits by taking *don't cares* into account.

Part I Deciding DQBF by Model Extraction Dependency Quantified Boolean Formulas (DQBF) extend Quantified Boolean Formulas (QBF) by allowing to explicitly specify the dependencies of existentially quantified variables. Consequently, in general, DQBF have existential variables with incomparable dependency sets. Deciding satisfiability of DQBF is NExpTime-complete [PAR01]. As such, it has a higher complexity than SAT or even the satisfiability problem of Quantified Boolean Formulas (QBF), which is PSPACE-complete [SM73]. While deciding DQBF is hard, the possibility of explicitly specifying dependencies allows to concisely encode several problems, which is not possible with SAT or even QBF. As mentioned earlier, a prominent example for an application of DQBF is partial equivalence checking [Git+13b].

In practice encoding problems in DQBF requires efficient procedures for deciding DQBF. Recent years showed a significant increase in the performance of available solvers. Still one shortcoming of several solvers is that they only provide yes/no answers but no certificates. Certificates are useful not just for validating the results of solvers, but models for satisfiable formulas can be of relevance in practice. As we saw earlier, models can be used to find implementations of the missing parts in partial equivalence checking.

In this part, we present a new decision procedure for DQBF that is able to generate certificates for satisfiable DQBF with no overhead. We will first discuss the fundamental idea of this procedure based on a simplified decision procedure for DQBF in Chapter 3. Next, in Chapter 4 we will introduce an advanced decision procedure for DQBF that is based on incrementally refining a candidate model. Finally, in Chapter 5, we will compare our solver Pedant, that is based on the proposed decision procedure, with state-of-the-art DQBF solvers on standard benchmark sets.

Part II Local Improvement of Circuits The ever-increasing size and complexity of modern integrated circuits makes both the design and the optimization of circuits a challenging task, both are unthinkable without some form of automation. Among others, one task for optimizing circuits is the reduction of a circuit's size. The increasing prices of silicon wafers in recent years [Gai21] indicate that small circuits are of importance in practice.

Computing a minimum size circuit for a given Boolean function is an NP-complete problem [ILO20]. This is reflected by the observation that, in practice, finding minimum size circuits is limited to very small circuits with not many more than 10 fanin-2 gates [KKY09; CMM23]. For this reason, heuristic methods need to be considered. One approach is to improve a circuit by replacing subcircuits with smaller circuits. Typically, this approach is limited to single-output [Rie+19] subcircuits, or it does not fully capture the implementational freedom for finding a replacement circuit [KPS22; Lee+18].

In this part, we will introduce a new method for minimizing circuit sizes that is based on replacing subcircuits by optimal replacement circuits. This method makes use of the full implementational freedom of multi-output subcircuits. First, we will discuss SAT and QBF encodings for computing minimum size circuits in Chapter 9. Next, we will adapt these encodings for computing minimum size replacement circuits of subcircuits in Chapter 10. These encodings can then be used for incrementally replacing subcircuits. This approach for minimizing circuits is implemented in the eSLIM system. In Chapter 11, we will experimentally evaluate eSLIM on standard benchmarks.

The two parts were written such that they can be read independently. This was not possible without introducing some redundancies in the preliminaries of the two parts.

Publications This thesis is based on the following publications:

1. Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. "Certified DQBF Solving by Definition Extraction". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 499–517
2. Franz-Xaver Reichl and Friedrich Slivovsky. "Pedant: A Certifying DQBF Solver". In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:10
3. Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. "Circuit Minimization with QBF-Based Exact Synthesis". In: *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Washington, DC, USA, February 7-14, 2023*. AAAI Press, 2023, pp. 4087–4094
4. Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. "eSLIM: Circuit Minimization with SAT Based Local Improvement". In: *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*. Vol. 305. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 23:1–23:14 – **Runner-up for the best student paper award.**

Software The following software was implemented as part of this thesis.

Pedant A model-generating DQBF solver.
Available at: `https://github.com/fslivovsky/pedant-solver`

eSLIM A tool for minimizing Circuits applying the SLIM framework.
Available at: `https://github.com/fxreichl/eSLIM`

Part I

Deciding DQBF by Model Extraction

CHAPTER 1

Introduction – Part I

The last decades showed tremendous progress in propositional satisfiability (SAT) solving [HJS19; Fro+21]. This resulted in a growing number of applications of SAT in various domains, ranging from AI planning [Rin21] over software verification [Kro21] to the exact synthesis of circuits [KKY09; Haa+20]. Moreover, efficient SAT solvers were essential for recent progress in constrained sampling and counting [Mee+16], two problems with many applications in artificial intelligence. In these cases, SAT solvers are used to deal with problems from complexity classes beyond NP. Consequently, propositional encodings grow super-polynomially in the size of the original instances. For this reason, these problems are not directly encoded in propositional logic but have to be reduced to a sequence of SAT instances.

The success of SAT solving on the one hand, and the inability of propositional logic to succinctly encode problems of interest on the other hand, have prompted the development of decision procedures for more succinct generalizations of propositional logic such as Quantified Boolean Formulas (QBF). Deciding QBF is PSPACE-complete [SM73] and thus believed to be much harder than SAT, but in practice, the benefits of a smaller encoding may outweigh the disadvantage of slower decision procedures [Fay+17]. A QBF is true if it has a *model*, which is a family of Boolean functions (often called *Skolem functions*) that satisfy the matrix of the input formula for each assignment of universal variables. The arguments of each Skolem function are implicitly determined by the nesting of existential and universal quantifiers.

Going even further, we can generalize QBF to Dependency Quantified Boolean Formulas (DQBF). DQBF extend QBF by allowing to explicitly state a *dependency set* for each existential variable, which is a subset of universal variables allowed as arguments of the corresponding Skolem function [PAR01; BCJ14a]. Consequently, unlike in QBF, the sets of arguments of the Skolem functions may no longer be ordered with respect to set inclusion. As Skolem functions in DQBF may have incomparable dependency sets, it is not necessarily possible to coordinate the Skolem functions for two existential variables. Consider the DQBF

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

In this formula, we want to find an assignment of e_1 that only depends on u_1 and an assignment of e_2 that only depends on u_2, such that at least one existentially quantified variable is assigned to true if and only if both universally quantified variables are assigned to true. As the dependency sets are not comparable, the Skolem function for e_1 may not "react" to the Skolem function for e_2 and vice versa. Since the clause $e_1 \vee e_2$ must be satisfied in case both u_1 and u_2 are assigned to true, we can assume w.l.o.g. that e_1 is assigned to true in this case. But e_1 must not react to u_2. This causes a counterexample, as e_1 is also assigned to true, whenever u_1 is assigned to true and u_2 to false. Therefore, we cannot find Skolem functions for the formula.

The additional expressiveness provided by the explicit dependency sets comes with a price: deciding DQBF is NExpTime-complete [PR79; PAR01]. Thus, it is believed to be even much harder than QBF. Before, we mentioned that, in practice, it can be beneficial to use QBF instead of SAT, due to the succinctness of QBF encodings, despite the slower decision procedures for QBF. Similarly, the expressiveness of DQBF can outweigh its slower decision procedures.

DQBF have been used for encoding several different problems. The most prominent one is probably partial equivalence checking (PEC). In PEC, we are given a circuit that contains unspecified parts. The goal is to check whether there is some implementation of these parts, such that the entire circuit satisfies a given specification [Git+13b]. DQBF have also been used for succinctly encoding the existence of Boolean functions subject to a set of constraints [Rab17] and for bounded synthesis [Fay+17]. Moreover, the problem of succinct graph 3-colorability or the problem of checking the existence of Hamiltonian cycles were reduced to DQBF [Che+22]. Furthermore, DQBF solvers could be used as backend solvers for other logics in NExpTime. This includes *Effectively Propositional Logic* (EPR) [Lew80], or certain *bit-vector* logics for *satisfiability modulo theories* (SMT) [KFB16].

Several decision procedures for DQBF have been developed in recent years. Conceptually, these solvers either reduce to SAT or QBF by instantiating [Frö+14] or eliminating universal variables [Git+15; Wim+17; GSW19; Síč20], or lift Conflict-Driven Clause Learning (CDCL) to non-linear quantifier prefixes by imposing additional constraints [FKB12; TR19].[1] We believe these methods should be complemented with algorithms that directly reason at the level of Skolem functions [RS16]. A strong argument in favor of such an approach is the fact that DQBF instances often have a large fraction of unique Skolem functions that can be obtained by definition extraction, but the current solving paradigms have no direct way of exploiting this [Sli20].

[1] An approach that does not fit this simplified classification is the First-Order solver iProver [Kor08].

As in the case of SAT solvers, it is not always sufficient to “only” obtain a yes/no answer from a DQBF solver for deciding the satisfiability of a formula. For example, solving a PEC instance, a yes/no answer only allows to conclude whether an implementation of the missing parts exists. But in case there is such an implementation, we do not know how such an implementation looks like. For this reason, it can be beneficial if DQBF solvers also return Skolem functions for each existentially quantified variable for satisfiable DQBF—similar to SAT solvers returning satisfying assignments for satisfiable formulas. In the case of partial equivalence checking, the Skolem functions describe the input-output relations of each unknown part. While DQBF solvers that can generate Skolem functions [Wim+16a] exist, most solvers do not compute Skolem functions. Using decision procedures for DQBF that reason directly at the level of Skolem functions could thus help to generate Skolem functions with no overhead.

In this part, we describe a new decision procedure for DQBF in the Counter-Example Guided Inductive Synthesis (CEGIS) paradigm [Sol+06; SJB08; JS17]. The procedure follows a dual strategy. To show satisfiability of formulas, it keeps a *candidate model* for the considered formula. This candidate is initialized by computing propositional definitions for existentially quantified variables in terms of their dependency sets. The algorithm then tries to refine this candidate to obtain an actual model. For this purpose, it incrementally determines counterexamples for the current candidate. These counterexamples are then used to refine the candidate, such that the updated candidate does not yield the same counterexample again in subsequent iterations. If at some point no further counterexamples can be found, we can conclude that the current candidate is an actual model, which means that the formula is satisfiable. In this case, the algorithm can return a Skolem function for each existentially quantified variable with no overhead.

To show unsatisfiability of formulas, the algorithm computes a sequence of clauses. The literals in these clauses correspond to existentially quantified variables, annotated by universal assignments of their dependency set. We use a SAT solver to check whether these clauses are satisfiable or not. In case they are unsatisfiable, it is not possible to refine the candidate to a model, i.e., the given DQBF is unsatisfiable.

We will show that the proposed algorithm is indeed a decision procedure for DQBF. For this purpose, we will show that the clauses, used to show unsatisfiability, correspond to clauses that can be derived by the $\forall$Exp+Res proof system for DQBF [JM13; Bey+19]. As this proof system is sound, the unsatisfiability of these clauses shows the unsatisfiability of the given DQBF.

We implemented the decision procedure in a system named Pedant. In an experimental evaluation on standard benchmark sets, Pedant could solve more instances than other state-of-the-art DQBF solvers.

Organization of Part I First, we will cover notations and definitions used throughout this part in Chapter 2. In this chapter, we will also give a brief overview of related work. Next, we will introduce a two-phase algorithm for deciding DQBF in Chapter 3. This algorithm is not indented for a practical application. Instead, it introduces some key ideas. In Chapter 4, we describe our main decision procedure for DQBF. In Chapter 5, we experimentally evaluate our DQBF solver Pedant, which implements this procedure. Finally, we conclude this part of the thesis in Chapter 6.

This part is mainly based on our previously published papers on DQBF solving. In particular, the two-phase algorithm, presented in Chapter 3, was covered in [RSS21]. A basic form of the CEGIS algorithm, presented in Chapter 4, was first discussed in [RSS21]. The version of the algorithm, presented in this thesis, is based on the algorithm given in [RS22]. We will discuss some parts of the algorithm that have not been covered in detail before, this in particular includes the conflict minimization. While in an extended version of [RSS21] we already showed that the algorithms are decision procedures, we both simplified the arguments and adapted them to the newest version of the CEGIS algorithm.

CHAPTER 2

Background – Part I

In this chapter, we will introduce terminology and notations used throughout this part of the thesis. Moreover, we will give a brief overview of related work.

2.1 Basic Concepts

We will denote the set of positive natural numbers by $\mathbb{N}^*$ and for $n \in \mathbb{N}^*$ we will denote the set $\{1, \ldots, n\}$ by $[n]$.

2.1.1 Graphs

In this section, we will briefly introduce concepts and notations from graph theory, used in this part of the thesis. For a comprehensive introduction to graph theory, we refer the reader to Diestel's book [Die00].

A *directed graph* G is a pair (V, E) s.t. $E \subseteq V \times V$. We denote elements of V as *vertices* and elements of E as *edges* of the graph G. Given a directed graph G, we denote its vertices by $V(G)$ and its edges by $E(G)$. A directed graph is *finite* if it has a finite number of vertices and edges. Throughout this thesis we will only consider finite directed graphs. Thus, if we talk about graphs we always mean finite directed graphs.

Let G be a graph with vertices v_1 and v_2 s.t. G contains the edge (v_1, v_2). Then we denote v_2 as a *successor* of v_1 and v_1 as a *predecessor* of v_2. Let $X \subseteq V(G)$ be a set of vertices then we define the graph $G - X$ as the graph with vertices $V(G) \setminus X$ and edges $\{(v_1, v_2) \in E(G) \mid v_1 \notin X, v_2 \notin X\}$.

A *path* P in a graph G is a sequence of vertices $v_0, v_1, \ldots, v_n$ with $(v_i, v_{i+1}) \in E(G)$ for each $0 \leq i < n$. Additionally, for each $i \in [n]$, we say that P contains the edge (v_{i-1}, v_i). We say that P *connects* the vertices v_0 and v_n. If there is a path connecting two vertices x and y, we say that x is *connected* to y. The *length* of a path P is the number of edges in P. Let P be a path of length ℓ with $\ell \geq 1$. The path P is *cyclic* if $v_0 = v_\ell$. A graph is *cyclic* if it contains a cyclic path and *acyclic* if it does not. A graph that is both directed and acyclic is often referred to as *directed acyclic graph* (DAG). All graphs considered in this thesis are DAGs.

Let G be a graph and $A, B, S \subseteq V(G)$ sets of vertices. If, for every pair of vertices a, b with $a \in A$ and $b \in B$, and every path P connecting a and b, there is a vertex $s \in S$ s.t. P contains s, then S is a *separator* for A and B. A separator S for A and B is *minimal* if, for every subset $S' \subsetneq S$, the set S' is not a separator for A and B. A *minimum separator* for A and B is a separator S of minimum size.

2.1.2 Flows

For a comprehensive introduction of flows we again refer to Diestel's book [Die00] respectively to the book by Cormen et al. [Cor+09], which also covers algorithms for max-flow computation.

A *flow network* F is a tuple (G, s, t, c) where G is a graph (V, E), $s, t \in V$ and c is a function $c : V^2 \to \mathbb{R}_{\geq 0}$. We call the vertex s the *source* of F, t the *sink* of F and the function c the *capacity* of F. For the sake of simplicity we always assume that $c(u, v) = 0$ if $(u, v) \notin E$. Additionally, we assume that there are no vertices v with $(v, s) \in E$ or $(t, v) \in E$. A *flow* f of F is a function $f : V^2 \to \mathbb{R}$. The flow f has to satisfy the following two conditions:

1. For all $(u, v) \in V^2$ we have $0 \leq f(u, v) \leq c(u, v)$.
2. For each vertex $u \in V \setminus \{s, t\}$ we have $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

We define $|f|$ as the value $\sum_{v \in V} f(s, v)$. A *maximal* flow of F is a flow f with maximal $|f|$. A *cut* C of the flow network F is a pair (S, T), s.t., S and T form a partition of V with $s \in S$ and $t \in T$. The *capacity* $c(S, T)$ of a cut (S, T) is given by the value $\sum_{u \in S} \sum_{v \in T} c(u, v)$. A *minimum* cut is a cut with the smallest possible capacity. The *net flow* $f(S, T)$ of a cut (S, T) is given by the value $\sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$. The *residual capacity* c_f of a flow f is a function $c_f : V^2 \to \mathbb{R}_{\geq 0}$ that is defined as follows. For edges $e = (v_1, v_2)$ we have $c_f(v_1, v_2) = f(e) - c(e)$ and $c_f(v_2, v_1) = f(e)$. For all other pairs of vertices c_f is defined as 0. The *residual graph* G_f for a flow f is a graph with vertices $V(G)$ and edges defined as follows: Let $v_1, v_2 \in V(G)$ then G_f contains the edge (v_1, v_2) if $c_f(v_1, v_2) > 0$.

2.1.3 Decision Trees

Throughout this thesis we will only consider decision trees for classifying binary features with binary classes. Thus, we will only introduce a simplified version of decision trees.

The set of *features* F is defined as $F = \mathbb{B}^n$ for some $n \in \mathbb{N}$. A *decision tree* for F is a rooted binary tree T with root T_r. Every leaf vertex ℓ in T is labeled by a boolean value *class*(ℓ). We refer to *class*(ℓ) as the *class* of ℓ. Every non-leaf vertex v in T is labeled by an integer *feature*$(v) \in [n]$, we refer to *feature*(v) as the *feature* of v.

For a non-leaf vertex v we denote its left child by *left*(v) and its right child by *right*(v).

A decision tree T for $\mathbb{B}^n$ represents a Boolean function $f_T : \mathbb{B}^n \to \mathbb{B}$. For defining f_T, we first introduce a function $f' : V(T) \times \mathbb{B}^n \to \mathbb{B}$. In the following let $b \in \mathbb{B}^n$. For a non-leaf v we define

$$f'(v, b) = \begin{cases} f'(\mathit{left}(v), b), & \text{if } b_{\mathit{feature}(v)} = 0 \\ f'(\mathit{right}(v), b), & \text{otherwise} \end{cases}.$$

For a leaf node ℓ we define $f'(\ell, b)$ as *class*(ℓ). Finally, $f(b)$ is defined as $f'(T_r, b)$.

Decision Tree Learning Classical decision tree learning procedures like ID3 [Qui86] or C4.5 [Qui93] induce a decision tree given a set of samples (i.e., realizations of the features) which are labeled with a class. For the application of decision trees which we will consider later, samples are not given a priori, but incrementally generated. Because of that, we consider *Hoeffding trees* [DH00]. *Hoeffding trees* are incrementally induced decision trees. Starting from a single root node, the leaves of the tree are periodically checked for splits. For this purpose, new samples are first sorted into one of the leaves of the tree. To represent the encountered samples, each leaf is equipped with counters for each realization of each feature and each possible class. The leaves are then labeled with the class of the majority of the samples. As soon as sufficiently many samples—a value n_{min} that is set as a parameter of the tree—were sorted into a leaf, the leaf is checked for a split. In this case the best two features for a split are selected based on a heuristic like information gain. If the heuristic evaluation function does not sufficiently differ between these features no split is introduced and the node is checked again as soon as n_{min} new samples arrived. Otherwise, the node is split with respect to the optimal feature computed before. In the new leaves the counters are initialized by 0, which means that old samples do not need to be remembered. It was shown that *Hoeffding trees* can be asymptotically arbitrarily close to a decision tree learned by traditional batch learning [DH00].

2.1.4 Propositional Logic

In this section, we will summarize notations and concepts for propositional logic, and the propositional satisfiability problem (SAT). For a comprehensive overview of SAT, including solving techniques, complexity results and applications, we refer to the Handbook of Satisfiability [Bie+21].

A *literal* is either a propositional variable or its negation. Given a literal ℓ, we denote the variable in ℓ by *var*(ℓ). A *clause* is a disjunction of literals and a *term* a conjunction of literals. A formula in *conjunctive normal form* (CNF) is a disjunction of clauses. We identify formulas in CNF by sets of clauses and both clauses and terms with sets of literals. Let C be a clause and φ a CNF. We define the set $var(C)$ of variables in C as $\{var(\ell) \mid \ell \in C\}$ and the set $var(\varphi)$ of variables in φ as $\bigcup_{C\in\varphi} var(C)$.

We denote the set $\{0,1\}$ of boolean values by $\mathbb{B}$. An *assignment* σ of a set of variables V is a function $V \to \mathbb{B}$. We denote the domain of an assignment σ by $\mathbf{dom}(\sigma)$. The set of all assignments of V is denoted by $\mathbb{B}^V$. Let σ be an assignment of V then for a set of variables W with $W \subseteq V$, we define the restriction of σ to W as the function $\sigma|_W$ that is defined by $\sigma|_W(x) = \sigma(x)$ for each x in W. Moreover, for two disjoint sets of variables V and W and assignments σ_1 of V and σ_2 of W, we define the union of σ_1 and σ_2 as $\sigma_1 \cup \sigma_2 : V \cup W \to \mathbb{B}$ such that $\sigma_1 \cup \sigma_2(x) = \sigma_1(x)$ for $x \in V$, and $\sigma_1 \cup \sigma_2(x) = \sigma_2(x)$ for $x \in W$. An assignment σ can be extended to literals by setting $\sigma(\neg v) = 1 - \sigma(v)$. We define the *instantiation* $C[\sigma]$ of a clause C by an assignment σ as follows. If there is a literal $\ell \in C$ such that $var(\ell) \in \mathbf{dom}(\sigma)$ and $\sigma(\ell) = 1$ then $C[\sigma] = \top$. Otherwise, $C[\sigma] = \{\ell \in C \mid var(\ell) \notin \mathbf{dom}(\sigma)\}$. We say that σ *satisfies* C if $C[\sigma] = \top$, and we say that σ *falsifies* C if $C[\sigma] = \emptyset$. Similarly, we can define instantiations of terms. We define the instantiation $\varphi[\sigma]$ of a CNF φ by an assignment σ as follows. If every clause $C \in \varphi$ is satisfied by σ then $\varphi[\sigma] = \top$. Otherwise, $\varphi[\sigma] = \{C[\sigma] \mid C \in \varphi, C[\sigma] \neq \top\}$. We say that φ is *satisfied* by σ if $\varphi[\sigma] = \top$, and we say that φ is *falsified* by σ if σ falsifies a clause in φ. A CNF φ is *satisfiable* if there is an assignment that satisfies φ, and it is *unsatisfiable* if there is no satisfying assignment. Whenever convenient, we identify an assignment σ of a set X with the term $\{x \mid x \in X, \sigma(x) = 1\} \cup \{\neg x \mid x \in X, \sigma(x) = 0\}$. In particular, for an assignment σ we identify $\neg\sigma$ with the clause that is the result of negating the associated term.

While satisfiability of a CNF is witnessed by a satisfying assignment, we can use *propositional resolution* to show unsatisfiability. Resolution is a refutational proof system that allows to derive a sequence of clauses $C_1, \dots, C_n$ from a CNF φ. For each $i \in [n]$ the clause C_i is either a clause from φ or it is derived by the resolution rule, which is shown below.

$$\frac{C_1 \cup \{v\} \qquad C_2 \cup \{\neg v\}}{C_1 \cup C_2}\ \text{(resolution)}$$

If we can derive a sequence of clauses $C_1, \dots, C_n$ from φ, we say that C_n is *derivable* from φ. If the empty clause $\bot$ can be derived from φ, we call the derivation a *refutation*. The resolution proof system is sound and refutationally complete. This means there is a refutation of a CNF φ if and only if φ is unsatisfiable.

2.2 Propositional Definitions

In this section, we recall basic definitions/notations for propositional definitions. We stick to the notation used by Slivovsky [Sli20].

Definition 2.1. Let φ be a propositional formula, $X \subseteq var(\varphi)$ a set of variables and $x \in var(\varphi)$. Then x is *implicitly defined* in φ by X if $\sigma(x) = \tau(x)$ for any satisfying assignments σ and τ of φ with $\sigma|_X = \tau|_X$. A *definition* for x in φ by X is a formula ψ with $var(\psi) \subseteq X$ such that $\psi[\sigma] = \sigma(x)$ holds for every satisfying assignment σ of φ. If there is a definition ψ for x in φ by X then x is *explicitly defined* in φ by X.

Remark 2.1. *We can see that for any satisfying assignment σ of φ and any definitions ψ_1 and ψ_2 for a variable v in φ by X we have $\psi_1[\sigma] = \psi_2[\sigma]$. Conversely, for a falsifying assignment σ, we can have $\psi_1[\sigma] \neq \psi_2[\sigma]$. This in particular means that definitions are not uniquely determined.*

For propositional logic it can be shown that implicit and explicit definability coincide.

Theorem 2.1 (Propositional version of Beth's theorem)**.** *Let φ be a propositional formula, $X \subseteq var(\varphi)$ a set of variables and $x \in var(\varphi)$. Then x is explicitly defined in φ by X iff it is implicitly defined in φ by X.*

For a proof of this theorem we refer to [LM08]. Thus, we just say that a variable x is *defined* in φ by X if it is implicitly defined. Given a formula φ and a variable $v \in var(\varphi)$, let v' denote some new variable. We now define the formula φ'_X as the formula which is obtained by replacing each $v \in var(\varphi) \setminus X$ by v'. To check whether a variable is defined we can use the following theorem.

Theorem 2.2 (Padoa's method)**.** *Let φ be a formula, $x \in var(\varphi)$ and $X \subseteq var(\varphi)$ s.t. $x \notin X$. Then x is defined in φ by X iff for each satisfying assignment σ of $\varphi \wedge \varphi'_X$ the assignment σ also satisfies $\neg x \vee x'$.*

For a proof of this theorem we again refer to [LM08]. As an immediate corollary we get the following.

Corollary 2.1. *Let φ be a formula, $x \in var(\varphi)$ and $X \subseteq var(\varphi)$ s.t. $x \notin X$. Then x is defined in φ by X iff $\varphi \wedge x \wedge \varphi'_X \wedge \neg x'$ is unsatisfiable.*

In order to compute definitions, we first need to recall the definition of an interpolant. We follow the notation by Slivovsky [Sli20].

Definition 2.2. Let φ and ψ be two propositional formulas s.t. $\varphi \wedge \psi$ is unsatisfiable. Then an *interpolant* I for φ and ψ is formula with the following properties:

- φ implies I, i.e., $\varphi \wedge \neg I$ is unsatisfiable.
- $I \wedge \psi$ is unsatisfiable
- $var(I) \subseteq var(\varphi) \cap var(\psi)$

Slivovsky proved the subsequent theorem that allows to obtain definitions from interpolants [Sli20].

Theorem 2.3. *Let φ be a propositional formula and $X \subseteq var(\varphi)$ a set of variables. A variable $x \in var(\varphi) \setminus X$ is defined by X in φ iff $\varphi \wedge x \wedge \varphi'_X \wedge \neg x'$ is unsatisfiable, and every interpolant of $\varphi \wedge x$ and $\varphi'_X \wedge \neg x'$ is a definition for x in φ by X.*

An approach for computing interpolants from resolution refutations is given in [McM03]. To compute interpolants in practice, an interpolating version of MiniSat [ES03] bundled with the ExtAvy model checker [GV14; VGM15] can be used.

2.3 DQBF

Dependency Quantified Boolean Formulas (DQBF) combine QBF with the idea of *Henkin* quantifiers, which were introduced for first-order logic [Hen61]. Thus, DQBF extend QBF by allowing to explicitly state dependencies for existential (universal) variables. This is in contrast to QBF, where dependencies of a variable are implicitly determined by the syntactic order of quantifiers. DQBF were first described by Peterson and Reif [PR79].

Syntax We only consider closed DQBF in *prenex conjunctive normal form* (PCNF). That is, each DQBF Φ has the shape $\mathcal{Q}.\varphi$, where φ denotes the *matrix* of Φ and $\mathcal{Q}$ the *prefix* of Φ. The matrix φ is a propositional formula in CNF that only consists of variables contained in the prefix. The prefix has the shape $\mathcal{Q} = \forall u_1, \ldots, u_n \exists e_1(D_1), \ldots, e_m(D_m)$. We refer to the variables $U = \{u_1, \ldots, u_n\}$ as the *universal variables* of Φ ($var_\forall(\Phi)$) and to the variables $E = \{e_1, \ldots, e_m\}$ as the *existential variables* of Φ ($var_\exists(\Phi)$). For each $i \in [m]$ we require that $D_i \subseteq U$. We call the set D_i the *dependencies* of e_i ($D_\Phi(e)$). If the DQBF is clear from the context we will just use $D(e)$. A *Quantified Boolean Formula* (QBF) can be considered as a special form of DQBF, where existential variables can be linearly ordered according to their dependencies. This means for a QBF Φ there is a linear ordering $<_\exists$ on $var_\exists(\Phi)$ such that for every existential variables x and y with $x <_\exists y$, we have $D(x) \subseteq D(y)$.

DQBF of the above shape are also said to be in *S-Form* [BCJ14a]. An alternative form are DQBF in *H-Form* [BCJ14a]. These formulas extend Henkin quantifiers by allowing to explicitly state dependencies of universal variables in terms of existential variables. Thus, the prefix of a DQBF in H-Form has the shape $\mathcal{Q} = \exists e_1 \ldots, e_n \forall u_1(D_1), \ldots, u_m(D_m)$. As in the case of formulas in S-Form, the prefix contains a set of existential variables E and a set of universal variables U, but for each $i \in [m]$ we require $D_i \subseteq E$. Throughout this thesis we will only consider DQBF in S-Form.

Semantics Let $\Phi = \mathcal{Q}.\varphi$ be a DQBF and let $e \in var_\exists(\Phi)$. We call a function $f_e : \mathbb{B}^{D(e)} \to \mathbb{B}$ a *Skolem function* for e. Let $f = (f_e)_{e \in var_\exists(\Phi)}$ be a family of Skolem functions. For a universal assignment σ, we define the existential assignment $f(\sigma)$ by $f(\sigma)(e) = f_e(\sigma|_{D(e)})$. We will refer to the assignment $f(\sigma)$ as the *response* of f to σ. The family f is a *model* if for each universal assignment σ the assignment $\sigma \cup f(\sigma)$ satisfies the matrix φ. A DQBF is said to be *satisfiable* or *true* if it has a model. And it is said to be *unsatisfiable* or *false* if it does not.

Deciding DQBF was shown to be NExpTime-complete [PR79; PAR01]. Recently, it was shown that even deciding 3-DQBF[1] is NExpTime-complete [FT23]. Since, deciding QBF is PSPACE-complete [SM73], deciding DQBF is harder than deciding QBF—assuming that NExpTime is not contained in PSPACE. Recently some tractable subclasses of DQBF have been identified [Sch+19; Gan+20].

Decision Procedures Let P be a procedure that takes a DQBF as input and returns *satisfiable* or *unsatisfiable*. If the procedure P always terminates and yields satisfiable if and only if the given DQBF is satisfiable then P is *decision procedure* for DQBF.

Dependency Schemes Dependency schemes allow to remove spurious dependencies from a DQBF. We follow the definition of dependency schemes used in [BBP20]. For this purpose, we first introduce a relation $\leq_{\mathcal{Q}}$ as follows. Let Φ and Φ' be two DQBF with the same matrix. Then we have $\Phi \leq_{\mathcal{Q}} \Phi'$ if $var_{\exists}(\Phi) = var_{\exists}(\Phi')$ and for each $e \in var_{\exists}(\Phi)$ we have $D_{\Phi}(e) \subseteq D_{\Phi'}(e)$. We can now define dependency schemes as follows:

Definition 2.3. A *dependency scheme* is a polynomial-time computable function $\mathcal{D}$ that maps DQBF to DQBF s.t. $\mathcal{D}(\Phi) \leq_{\mathcal{Q}} \Phi$ for each formula Φ.

A dependency scheme $\mathcal{D}$ is said to be *fully exhibited* if Φ is satisfiable if and only if $\mathcal{D}(\Phi)$ is satisfiable.

In this thesis we will only consider the *reflexive resolution path dependency scheme* $\mathcal{D}^{RRS}$. $\mathcal{D}^{RRS}$ was first introduced for QBF [SS16]. We follow the definition of $\mathcal{D}^{RRS}$ in [BBP20].

Definition 2.4. Let $\Phi = \mathcal{Q}.\varphi$ be a DQBF. Then the *reflexive resolution path dependency scheme* $\mathcal{D}^{RRS}(\Phi)$ is defined as the DQBF $\Phi' = \mathcal{Q}'.\varphi$, whose prefix is defined as follows. The prefix $\mathcal{Q}'$ contains the same existential and universal variables as $\mathcal{Q}$. Moreover, for each existential variable e the set $D_{\Phi'}(e)$ is defined as the subset of $D_{\Phi}(e)$ s.t. for each $u \in D_{\Phi'}(e)$ there must be a sequence of clause $C_1, \ldots, C_k$ in the matrix φ and a sequence of existential literals $\ell_1, \ldots, \ell_{k-1}$ that satisfy the following conditions:

- $u \in C_1$ and $\neg u \in C_k$
- for some $j \in [k-1]$ we have $var(\ell_j) = e$
- for each $j \in [k-1]$ we have $\ell_j \in C_j, \neg\ell_j \in C_{j+1}$ and $u \in D(var(\ell_j))$
- for each $j \in [k-2]$ we have $var(\ell_j) \neq var(\ell_{j+1})$

[1]A k-DQBF is a DQBF with k existentially quantified variables.

It can be shown that $\mathcal{D}^{RRS}$ is fully exhibited [Wim+16b]. This means to check if a DQBF Φ is satisfiable, we can always check if $\mathcal{D}^{RRS}(\Phi)$ is satisfiable instead. Moreover, a model f of $\mathcal{D}^{RRS}(\Phi)$ yields a model for Φ—the Skolem functions in the model for Φ only need to have additional unused arguments for each variable removed from the dependencies.

2.4 Proof Systems for DQBF

While a model shows that a DQBF is satisfiable, unsatisfiability can be shown by applying *proof systems*. Different proof systems for DQBF have been proposed. These in particular include QBF proof systems that have been lifted to DQBF, like *DQBF-Q-Res* [KKF95; BCJ14a], *DQBF-QU-Res* [Gel12; Bey+19], *DQBF-∀Exp+Res* [JM13; Bey+19] and *DQBF-IR-calc* [BCJ14b; Bey+19]. Throughout this thesis, we will only consider the DQBF variants of the proof systems, thus we will omit the *DQBF* prefix when referring to the proof systems. All of these proof systems are refutational clausal proof systems. This means that the proof systems consist of sets of rules that allow to derive a sequence of clauses $C_1, \ldots, C_n$. We call such a sequence of clauses a *P-derivation* of C_n, where P denotes the respective proof system. In these derivations each clause can be derived from previously derived clauses, or from the DQBF itself. A derivation of the empty clause $\bot$ is called a *refutation*. All proof systems mentioned above are sound, but only ∀Exp+Res and IR-calc are refutationally complete [BCJ14a; Bey+19]. In this thesis, we will only consider the expansion based proof systems ∀Exp+Res and IR-calc. Next we will describe ∀Exp+Res and then IR-calc.

The intuition behind expansion-based proof systems is that we can transform a (D)QBF to an equisatisfiable propositional formula by instantiating each universal variable both by 0 and 1 and by conjoining the resulting formulas. This procedure is called *expansion*. Unsatisfiability of the original QBF/DQBF can then be shown by resolution on the expanded formula. For the expansion it is necessary that existential variables are replaced by new variables for each assignment to its dependencies. Consider the following DQBF:

$$\forall u_1, u_2 \exists e_1(u_1), e_2(u_2).(e_1 \lor u_1) \land (e_2 \lor u_2) \land (u_1 \lor \neg e_1 \lor \neg e_2).$$

Expansion yields the following propositional formula:

$$e_1^{\neg u_1} \land e_2^{\neg u_2} \land (\neg e_1^{\neg u_1} \lor \neg e_2^{u_2}) \land (\neg e_1^{\neg u_1} \lor \neg e_2^{\neg u_2})$$

We annotate existential variables with assignments of their dependencies to distinguish variables for different assignments. By applying propositional resolution, one can now easily verify that this propositional formula is unsatisfiable. Thus, also the initial (D)QBF is unsatisfiable.

The above example also shows that not all clauses from the expansion are necessary for the propositional resolution step (e.g., $(\neg e_1^{\neg u_1} \lor \neg e_2^{u_2})$ is not part of the refutation). Instead of expanding all clauses of the matrix and using these clauses for propositional

$$\frac{}{\{\ell^{\sigma|_{D(var(\ell))}} \mid \ell \in C, var(\ell) \in var_\exists(\Phi)\}}\ \text{(axiom)}$$

The *axiom* rule allows to derive a new clause from Φ. Here σ denotes a total assignment of the universal variables that falsifies each universal literal in a clause C from the matrix. For a variable x^τ we call τ the *annotation* of x. Note that variables with different annotations denote different variables.

$$\frac{C_1 \cup \{e^\tau\} \qquad C_2 \cup \{\neg e^\tau\}}{C_1 \cup C_2}\ \text{(resolution)}$$

The *resolution* rule applies propositional resolution to derive a new clause from the clauses $C_1 \cup \{e^\tau\}$ and $C_2 \cup \{\neg e^\tau\}$. The clauses only consist of annotated existential literals. This in particular means that e is an existential variable and τ an assignment for $D(e)$.

Figure 2.1: The rules of ∀Exp+Res, where Φ is the given DQBF.

resolution, we can consider ∀Exp+Res proof system. The rules for ∀Exp+Res are given in Figure 2.1. The proof system allows to expand any clause C from the matrix with a total assignment of the universal variables that falsifies each universal literal in C. The resulting clauses can then be used for propositional resolution. Here it is important, that variables with different annotations are considered as different variables. Thus, resolution can only be applied if the annotations match. As mentioned above, ∀Exp+Res is sound and refutationally complete, i.e., the empty clause $\bot$ can be derived if and only if the given DQBF is unsatisfiable [Bey+19].

In the ∀Exp+Res system, we always have to instantiate with total assignments. Even for clauses with few universal literals, total assignments for the universal variables must be used. By weakening this constraint to partial assignments, we obtain the IR-calc proof system. The rules for IR-calc are given in Figure 2.2. In IR-calc, we can instantiate clauses with the smallest possible universal assignment that falsifies all universal literals. We can then extend the annotations of the existential literals in case this is necessary for a resolution step. Like ∀Exp+Res, IR-calc is sound and refutationally complete [Bey+19].

We say that a proof system P_1 *p-simulates* a proof system P_2 if every P_1 proof of some formula φ can be converted to a P_2 proof of φ by a polynomial time procedure. Additionally, a family $\mathcal{F}$ of (DQBF) PCNFs *separates* P_1 from P_2 if $\mathcal{F}$ has polynomial size refutations in P_1 but not in P_2 [BB20; CR79]. It is easy to see that IR-calc p-simulates ∀Exp+Res. It was shown that there is a family of QBF PCNFs $\mathcal{F}$ that separates IR-calc from ∀Exp+Res [JM15; BB20]. As DQBF-IR-calc, respectively DQBF-∀Exp+Res extend their QBF variants, the separation for QBF is carried over to DQBF. This means that IR-calc can be considered as stronger than ∀Exp+Res.

$$\frac{}{\{\ell^{\sigma|_{D(var(\ell))}} \mid \ell \in C, var(\ell) \in var_{\exists}(\Phi)\}}\ (\text{axiom})$$

The *axiom* is similar to the same named rule in ∀Exp+Res. The major difference is that here σ denotes a partial assignment of the universal variables that falsifies each universal literal in a clause C from the matrix.

$$\frac{C}{\{l^{\tau\circ\sigma|_{D(var(l))}} \mid l^{\tau} \in C\}}\ (\text{instantiation})$$

The *instantiation* rule allows to specialize the annotations in a clause C by some universal assignment σ

$$\frac{C_1 \cup \{e^{\tau}\} \qquad C_2 \cup \{\neg e^{\tau}\}}{C_1 \cup C_2}\ (\text{resolution})$$

Similarly as ∀Exp+Res also IR-calc uses the propositional resolution rule. The difference is that now annotations are not necessarily total.

Figure 2.2: The rules of IR-calc.

2.5 Solving DQBF

Several decision procedures for DQBF have been developed in recent years. In contrast to SAT, where CDCL-based solvers play a dominant role [MLM21], there is a larger variety of different solving approaches for DQBF. We give a brief overview of some existing solving techniques for DQBF. For a more detailed discussion of DQBF solving methods, we refer to the surveys by Scholl and Kovásznai [Kov16; SW18].

- One approach to solving DQBF is realized in the *DQDPLL* procedure. DQDPLL lifts the DPLL procedure for SAT and for QBF to DQBF. As DPLL assigns variables one after another, it entails a linear ordering of variables, whereas in a DQBF there is generally no linear ordering of variables with respect to their dependencies. As a consequence, DPLL needs to be equipped with additional constraints: it can be necessary to enforce the same assignment for an existential variable for different paths in the assignment tree [FKB12]. A solving technique related to DQDPLL is implemented in the solver dCAQE. dCAQE lifts the idea of clausal abstraction from QBF to DQBF. The core idea is that each existential (universal) variable is associated with a propositional formula that indicates which clauses can be satisfied (falsified) by assigning this variable. The solver uses these formulas to construct an assignment step by step, where the order is determined by the dependencies. In this process the formulas associated with each variable are used to obtain assignments for the variables. If there is no suitable assignment—every assignment to an existential variable falsifies the matrix, respectively every assignment to a universal variable satisfies the matrix—the associated formula gives a reason. This reason can then be used to jump back to a previously considered variable and to modify its associated formula [RT15; TR19].

- Another approach is to reduce DQBF either to SAT or QBF. The solver iDQ instantiates each clause C with a partial universal assignment that falsifies every universal literal in C. Existential literals for different instantiations are considered as different. Thus, if the resulting propositional formula is unsatisfiable, then also the initial DQBF is unsatisfiable. On the other hand, if the propositional formula is satisfiable, then the initial DQBF is not necessarily satisfiable, as some existential variables might not be assigned consistently. In that case additional clauses that further constrain the formula might need to be added [Frö+14].

- While iDQ reduces DQBF to SAT, the solver HQS reduces DQBF to QBF. The basic idea of HQS is to eliminate universal variables by expanding them to obtain a QBF. In order to reduce the number of introduced additional existential variables, this approach is refined by only removing universal variables from individual dependency sets. These variables are selected such that a smallest possible number of eliminations suffices to obtain a QBF [Git+15; Wim+17; GSW19]. HQS is a competitive DQBF solver that won the DQBF track of the QBF Evaluation in the years 2018 and 2019 [PSS24]. The solver DQBDD builds on the ideas used in HQS. DQBDD constructs a *Binary Decision Diagram* (BDD) for a given DQBF. To do this, quantifiers are pushed into the matrix. Then, a BDD is constructed by expanding both universal and existential quantifiers [Síč20; SS21b]. DQBDD is a competitive solver that won the DQBF track of the QBF Evaluation in 2020 [PSS24].

- Besides the solvers presented above, DQBF can also be solved by a translation to *Effectively Propositional Logic* (EPR). The satisfiability problem for EPR is NExpTime-complete [Lew80]. Thus, there is a polynomial reduction from DQBF to EPR. After translating a DQBF to EPR, the solver iProver [Kor08] can be applied, for instance.

- Last but not least, we also want to mention Manthan a tool for synthesizing Skolem Functions. Manthan first computes satisfying assignments of the matrix. Then it applies decision tree learning to obtain initial candidate Skolem functions for each existential variable, from the sampled satisfying assignments. These candidates are then iteratively refined. While Manthan is incomplete, it could still solve some formulas that could not be handled by any other state of the art DQBF solver [GRM20; Gol+21; GRM23].

2.6 Applications of DQBF

While DQBF is not as widely used to solve (practical) problems as SAT or QBF, several interesting applications were developed in recent years. One of these is *Partial Equivalence Checking* (PEC) [Git+13b; Git+13a]. In PEC, circuits that contain unspecified parts (black boxes) are considered. For these parts we only know the inputs and the outputs, but not the implementation. The goal of PEC is to decide whether there is some implementation of the black boxes such that the resulting circuit satisfies a given specification. Gitina et al. [Git+13b] showed that PEC can be encoded as DQBF. The core idea of the encoding is to represent the outputs of the black boxes by existentially quantified variables that only depend on variables representing the inputs of the unspecified part.

Several other applications of DQBF have been proposed. DQBF can be considered to encode the existential quantification of Boolean functions [Rab17]. Here, the task is that a formula containing applications of an unspecified function is given. We want to check whether there is some Boolean function that satisfies the given formula. The core idea of the DQBF encoding is to represent applications of the function by existentially quantified variables with disjoint dependency sets. DQBF have also been used for *Reactive Synthesis* [Fay+17]. Here, a system specification is given in temporal logic. The goal is to find a transition system that satisfies the given specification. Additionally, Chen et al. [Che+22] discussed the reduction of further NExpTime-complete problems to DQBF. These reductions include the problem of succinct graph 3-colorability or the problem of checking the existence of Hamiltonian cycles.

Moreover, DQBF solvers could be used as backend solvers for other logics in NExpTime. This includes the already mentioned logic EPR [Lew80], but also certain *bit-vector* logics for *satisfiability modulo theories* (SMT) [KFB16].

CHAPTER 3

A Two-Phase Algorithm

In this section, we will introduce a fairly simple algorithm for deciding DQBF. The primary purpose of this algorithm is to introduce basic concepts for the main algorithm which we will introduce in the next chapter. The description given in this chapter is based on our paper [RSS21].

The algorithm is presented in Algorithm 1. First we will discuss the algorithm step by step. Then we will show that Algorithm 1 is a decision procedure for DQBF.

Description of the algorithm The main idea of Algorithm 1 is to compute a Skolem function for each existential variable. To obtain the Skolem functions, propositional definitions are computed. The algorithm then checks whether the Skolem functions describe a model.

To realize this idea the algorithm proceeds in two phases. In the first phase (GENERATE-DEFINITIONS), it computes a definition γ_e for each existential variable e. In general, not every existential variable e has a definition by $D(e)$. To still obtain definitions the algorithm introduces a set A of auxiliary *arbiter variables* whose semantics are encoded in a set ψ_A of *arbiter clauses*, both of which are empty initially.

For each existential variable e, we now first use Theorem 2.3 to check definability of e (line 7)—we assume that a procedure SAT, which checks satisfiability of a propositional formula, is given. If a variable e is not defined then there must be an assignment σ for $D(e)$ such that the assignment of e is not fixed under σ. This means that there are satisfying assignments ρ_1 and ρ_2 of the matrix φ with $\rho_1|_{D(e)} = \rho_2|_{D(e)} = \sigma$ and $\rho_1(e) \neq \rho_2(e)$. We can obtain such an assignment σ by considering the condition from Theorem 2.3 (line 8)—we assume that a procedure GETMODEL, which computes a satisfying assignment for a satisfiable propositional formula, is given. Next, we fix the assignment for e under σ. For this purpose, we introduce an *arbiter variable* e^σ. The aim of this new variable is to determine the value of the Skolem function for e under

Algorithm 1 Solving DQBF by definition extraction.

```
1:  procedure twoPhase($\Phi$)
2:      $\psi_D, \psi_A \leftarrow$ generateDefinitions($\Phi$)
3:      return findArbiterAssignment($\Phi, \psi_D, \psi_A$)

4:  procedure generateDefinitions($\Phi$)
5:      $\psi_D \leftarrow \emptyset,\ \psi_A \leftarrow \emptyset,\ A \leftarrow \emptyset$
6:      for each $e \in var_\exists(\Phi)$ do
7:          while not isDefined($e, \varphi \wedge \psi_A, D(e) \cup A$) do
8:              $\sigma =$ getUndefinedReason$(e, \varphi \wedge \psi_A, D(e) \cup A)|_{D(e)}$
9:              $A \leftarrow A \cup \{e^\sigma\}$
10:             $\psi_A \leftarrow \psi_A \wedge (e^\sigma \vee \neg\sigma \vee \neg e) \wedge (\neg e^\sigma \vee \neg\sigma \vee e)$
11:         $\gamma_e \leftarrow$ getDefinition($e, \varphi \wedge \psi_A, D(e) \cup A$)
12:         $\psi_D \leftarrow \psi_D \wedge (e \leftrightarrow \gamma_e)$
13:     return $\psi_D, \psi_A$

14: procedure findArbiterAssignment($\Phi, \psi_D, \psi_A$)
15:     $\tau \leftarrow \bigwedge_{a \in A} a,\ blockingClauses \leftarrow \emptyset$
16:     loop
17:         if isSat($\neg\varphi \wedge \psi_D \wedge \tau$) then
18:             $\sigma \leftarrow$ getModel$(\neg\varphi \wedge \psi_D \wedge \tau)|_{var_\forall(\Phi)}$
19:             $\tau' \leftarrow$ getCore$(\varphi \wedge \psi_A \wedge \sigma, \tau)|_A$
20:             $blockingClauses \leftarrow blockingClauses \wedge \neg\tau'$
21:             if isSat($blockingClauses$) then
22:                 $\tau \leftarrow$ getModel($blockingClauses$)
23:             else
24:                 return unsatisfiable
25:         else
26:             return satisfiable

27: procedure isDefined($v, \delta, X$)
28:     return ¬isSat($\delta \wedge v \wedge \delta'_X \wedge \neg v'$)

29: procedure getUndefinedReason($v, \delta, X$)
30:     return getModel($\delta \wedge v \wedge \delta'_X \wedge \neg v'$)

31: procedure getDefinition($v, \delta, X$)
32:     return getInterpolant($\delta \wedge v, \delta'_X \wedge \neg v'$)
```

σ. This means that e shall be assigned to true under σ iff e^σ is assigned to true. In subsequent iterations, we include these arbiter variables in the set of variables that can be used in a definition of e. To enforce the correspondence of the assignments of e and e^σ we also add the clauses $e^\sigma \vee \neg\sigma \vee \neg e$ and $\neg e^\sigma \vee \neg\sigma \vee e$ to subsequent definability checks. If there are multiple assignments σ of $D(e)$ for which the assignment of e is not fixed, we introduce arbiter variables for all of them. When the inner loop terminates, we know that e is defined by $D(e) \cup A$ in $\varphi \wedge \psi_A$. We can thus obtain a definition by computing an interpolant (line 11)—for this purpose we assume that a procedure GETINTERPOLANT is given. The definition is then added to the formula ψ_D that contains the definitions for every existential variable.

In the second phase (FINDARBITERASSIGNMENT) we then want to find an assignment of the arbiter variables under which the definitions obtained in the first phase are a model. Starting with an initial assignment τ[1], we use a SAT solver to check whether the formula $\neg\varphi \wedge \psi_D$ consisting of the negated matrix of the input DQBF and the definitions from the first phase is satisfiable under τ (line 17). If the formula is unsatisfiable then there is no universal assignment σ that can falsify φ under τ and ψ_D. Thus, τ and ψ_D describe a model for Φ and so the algorithm returns SATISFIABLE. If, on the other hand, the formula is satisfiable, then the matrix can be falsified by some universal assignment σ under τ and ψ_D. This in particular means that $\varphi \wedge \psi_A$ must be unsatisfiable under τ and σ—if the formula would be satisfiable the assignments of the existential variables would need to adhere to the definitions in ψ_D, but we know that φ is falsified under σ, τ and ψ_D. Thus, we can compute a sub-assignment τ' for τ such that $\varphi \wedge \psi_A \wedge \sigma$ remains unsatisfiable under τ'. But this means that for every arbiter assignment that contains τ', we can falsify the matrix. For this reason arbiter assignments used in subsequent iterations must differ from τ'. This is realized by adding the clause $\neg\tau'$ to a set of *blocking clauses* that is empty initially. A new assignment τ for the arbiter variables is obtained by a satisfying assignment for the blocking clauses. If the set of blocking clauses is unsatisfiable, we cannot find a new assignment for the arbiters that differs from all previous assignments τ'. Thus, for every arbiter assignment τ the matrix φ is falsified under $\psi_D \wedge \tau$ for some assignment to the universals σ. So there cannot be a model and the DQBF is unsatisfiable.

Proofs Next we will show that Algorithm 1 is a decision procedure for DQBF. For this purpose, we will first show that the algorithm always terminates. Then, we show that if the algorithm reports satisfiability of a DQBF Φ then Φ is satisfiable. Finally, we show that if a DQBF Φ is satisfiable then the algorithm reports the satisfiability of Φ.

Lemma 3.1 (termination). *Algorithm 1 terminates.*

Proof. The procedure GENERATEDEFINITIONS terminates as there are only finitely many assignments σ and an assignment σ cannot be repeated within the inner loop. The proce-

[1]In Algorithm 1 we initialize τ as the assignment that maps each arbiter variable to true, but we could use any assignment.

dure findArbiterAssignment terminates as there are only finitely many assignments for the arbiter variables A and these are not repeated. □

Theorem 3.1. *If Algorithm 1 reports satisfiability of a DQBF Φ, then Φ is satisfiable.*

Proof. Let Φ be a DQBF s.t. the algorithm reports satisfiability of Φ. We have to show that Φ is indeed satisfiable. As the algorithm reports satisfiability of Φ there must be a formula ψ_D and an arbiter assignment τ s.t. $\neg\varphi \wedge \psi_D \wedge \tau$ is unsatisfiable. The construction of ψ_D ensures that every assignment σ of the universal variables can be extended to a unique assignment $\hat{\sigma}$ of all variables that satisfies $\psi_D \wedge \tau$. We define for each existential variable e a Skolem function f_e that is given by $f_e(\sigma|_{D(e)}) = \hat{\sigma}(e)$. We denote the family of all Skolem functions by f. Now let σ be an arbitrary but fixed assignment of the universal variables. As $\neg\varphi \wedge \psi_D \wedge \tau$ is unsatisfiable, we can conclude that $\sigma \cup f(\sigma)$ falsifies $\neg\varphi$. But this means that $\sigma \cup f(\sigma)$ satisfies φ. As σ was arbitrary this means that f is a model for Φ and this in turn means that Φ is satisfiable. □

Theorem 3.2. *If a DQBF Φ is satisfiable, then Algorithm 1 reports satisfiability of Φ.*

Proof. Let Φ be a satisfiable DQBF. If Φ does not contain any existential variables, then the satisfiability of Φ implies that $\neg\varphi$ is unsatisfiable. Since this implies that Algorithm 1 reports satisfiability of Φ, in the following we can assume that Φ contains at least one existential variable. As the formula is satisfiable, we know that there is a model f. We have to show that there is an arbiter assignment τ s.t. $\neg\varphi\wedge\psi_D\wedge\tau$ is unsatisfiable. For this purpose, we define the arbiter assignment τ by $\tau(e^\sigma) = f_e(\sigma)$. Next let $\hat{\sigma}$ be an arbitrary but fixed assignment of the universal, existential and arbiter variables. Moreover, let e be an arbitrary but fixed existential variable and $\sigma = \hat{\sigma}|_{var_\forall(\Phi)}$. As Φ is satisfiable there must be total assignment ρ satisfying φ with $\rho|_{var_\forall(\Phi)} = \sigma$. We distinguish between two cases. Either there is a satisfying assignment μ for φ with $\mu|_{var_\forall(\Phi)} = \rho|_{var_\forall(\Phi)}$ and $\mu(e) \neq \rho(e)$ or there is no such assignment. In the first case we introduced the arbiter variable e^σ. We also know that for a total assignment δ with $\delta|_{var_\forall(\Phi)} = \sigma$ the definition γ_e evaluates to $\delta(e^\sigma)$. In the second case we must have $\rho(e) = f_e(\sigma)$—otherwise f could clearly not be a model. Moreover, the definition γ_e must evaluate to $f_e(\sigma)$ under any assignment coinciding with σ. This allows to conclude that $\hat{\sigma}(e) = f_e(\sigma)$. As e was arbitrary this means that $\hat{\sigma}$ assigns the existential variables according to the response of f to σ. But this means that $\hat{\sigma}$ satisfies φ, thus it falsifies $\neg\varphi \wedge \psi_D \wedge \tau$. As $\hat{\sigma}$ was arbitrary this means that $\neg\varphi \wedge \psi_D \wedge \tau$ is unsatisfiable. □

As an immediate consequence of the above theorems, we can conclude the following corollary.

Corollary 3.1. *Algorithm 1 is a decision procedure for DQBF.*

CHAPTER 4

Counterexample-Guided Algorithm

In this chapter, we will introduce another algorithm for solving DQBF. The algorithm builds upon the idea of using arbiter variables for fixing the response of Skolem functions for specific universal assignments (cf. Chapter 3). We will first briefly discuss the motivation behind the algorithm. Then we will discuss the individual components of the algorithm step by step. Last but not least, we will show that the algorithm defines a decision procedure for DQBF. The presentation closely follows our work on DQBF solving [RSS21; RS22].

Discounting SAT calls, the running time of Algorithm 1 is essentially determined by the number of assignments of a dependency set for which the corresponding existential variable is not defined: it introduces an arbiter variable for each such assignment in the first phase, and the number of iterations in the second phase is bounded by the number of arbiter assignments. As a result, even a single existential variable being unconstrained and having a large dependency set can cause the algorithm to get stuck enumerating universal assignments. Thus, in practice, Algorithm 1 is too inefficient.

A key insight underlying the success of counter-example guided solvers for QBF [Jan+16; Jan18; Ten19] is that it is typically overkill to perform complete expansion of universal variables. Instead, they incrementally refine Skolem functions by taking into account universal assignments that pose a problem for the current solution candidate.[1]

Following this idea, we now present an improved algorithm (Algorithm 2) in the style of Counter-Example Guided Inductive Synthesis (CEGIS) [JS17].

[1]In these QBF solvers, Skolem functions are typically only indirectly represented by trees of formulas (*abstractions*) that encode viable assignments.

Algorithm 2 CEGIS-based DQBF Solving.

```
1: procedure PEDANT($\Phi' = \mathcal{Q}'.\varphi'$)
2:     $\Psi \leftarrow$ INITIALIZE($\Phi'$)
3:     $\Delta \leftarrow \emptyset$, $\tau \leftarrow \emptyset$
4:     loop
5:         if CHECKCANDIDATE($\Psi$) then
6:             return SATISFIABLE
7:         $\mathcal{C}, \sigma \leftarrow$ GETCONFLICT($\Psi, \tau$)
8:         REFINECANDIDATE($\Psi, \Delta, \mathcal{C}, \sigma$)
9:         if ISSAT($\Delta$) then
10:            $\tau \leftarrow$ GETMODEL($\Delta$)
11:        else
12:            return UNSATISFIABLE
```

4.1 Overview

The underlying idea of Algorithm 2 is to simultaneously refine a family of Skolem functions Ψ—which we call *candidate model* (or just candidate for short)—and to compute a sequence of clauses Δ for a given DQBF. Here, Ψ and Δ have opposing purposes: To show satisfiability, we want to construct a model using Ψ. On the other hand, we show unsatisfiability by ensuring the existence of a $\forall$Exp+Res refutation using Δ. Eventually, the algorithm is able to refine Ψ to an actual model or the clauses in Δ become unsatisfiable. All clauses in Δ correspond to clauses that can be derived by $\forall$Exp+Res, so this shows unsatisfiability of the given DQBF.

As in Algorithm 1 we want to make use of definitions. For this purpose, we initialize the candidate Skolem functions for defined variables by their definitions. The remaining Skolem functions can be initialized to some arbitrary function. Another similarity with Algorithm 1 is that we use arbiter variables. These variables have a similar purpose as before: if it is not immediately clear how to assign an existential under a universal assignment, we introduce an arbiter variable. The corresponding arbiter clauses are added to the candidate. This means that the existential response of the candidate depends on the arbiter assignment τ.

To refine the candidate, we check if the current candidate already serves as a model. If this is not the case, we compute a reason for the inadequacy of the candidate. This reason can basically be understood as a universal assignment and a subset of the corresponding response of the current candidate. We can conclude that no candidate with this response can serve as a model, so we modify the response by using the obtained reason. To do this, we consider two different approaches. First, if we can conclude that assigning a single existential variable differently resolves the problem, we introduce a *forcing clause*. A forcing clause can be understood as an implication $p \Rightarrow e$, where e might depend on each variable in the premise p. If, on the other hand, we just know that some set of existentials must be assigned differently, but we don't yet know how, we introduce arbiter variables.

Algorithm 3 Initialization of the candidate.

```
1: procedure INITIALIZE($\Phi$)
2:     $D \leftarrow$ REDUCEDEPENDENCIESRRS($\Phi$)
3:     $ED \leftarrow$ COMPUTEEXTENDEDDEPENDENCIES($\Phi$)
4:     $\varphi \leftarrow$ REDUCEMATRIX($\Phi$)
5:     $\Psi \leftarrow$ INITCANDIDATE($\Phi$)
6:     UPDATEEXTENDEDDEPENDENCIES($\Psi$)
7:     return $\Psi$

8: procedure INITCANDIDATE($\Phi$)
9:     for each $e \in var_\exists(\Phi)$ do
10:        if ISDEFINED($e, \varphi, ED(e)$) then
11:            $\psi_D^e \leftarrow$ GETDEFINITION($e, \varphi, ED(e)$)
12:        else
13:            $\psi_D^e \leftarrow \emptyset$
14:    return CANDIDATE($(\psi_D^e)_{e \in var_\exists(\Phi)}$)

15: procedure UPDATEEXTENDEDDEPENDENCIES($\Phi$)
16:    for each $e \in var_\exists(\Phi)$ do
17:        if no definition for $e$ in $\Psi$ then
18:            for each definition $\psi_D^x$ in $\Psi$ do
19:                $S \leftarrow var(\psi_D^x)$
20:                if $S \subseteq ED(e)$ then
21:                    $ED(e) \leftarrow ED(e) \cup \{x\}$
```

Based on the reason for the conflict, we then add a clause consisting of arbiter literals to Δ. If Δ is still satisfiable, we can obtain a new assignment for the arbiter variables as a satisfying assignment for Δ. This ensures that the new arbiter assignment differs from previous ones that caused a problem.

This refinement procedure is then repeated until a model is found, or no new arbiter assignment can be found. If no new arbiter assignment can be found, we know that under no arbiter assignment the candidate yields a model. We will show that this is a sufficient condition for the existence of a $\forall$Exp+Res refutation.

In the subsequent sections we will give a more detailed description of the individual parts of the algorithm.

4.2 Initialization

In this section, we will describe the individual steps of the initialization phase (Algorithm 3).

As mentioned above, the algorithm follows a dual strategy of trying to derive a refutation on the one hand, and trying to compute a model on the other hand. For these two subtasks we use slightly different dependency sets. For the first task, we prefer small dependency sets, as smaller dependency sets yield shorter annotations in the clauses derivable by ∀Exp+Res. Thus, we apply the reflexive resolution path dependency scheme to remove spurious dependencies (line 2). We denote the prefix obtained by applying the dependency scheme by $\mathcal{Q}$ and the resulting dependencies by $D(e)$, whereas we denote the original prefix by $\mathcal{Q}'$ and the original dependencies by $D'(e)$. For computing compact Skolem functions we want to allow that under certain conditions existential variables can depend on other existential variables. By adding existential variables to the dependencies of a variable e, we can use their Skolem functions in the definition of the Skolem function for e without the need of "copying" their definition. For this purpose, we compute for each existential variable e, the *extended dependencies* of e denoted by $\mathit{ED}(e)$ (line 3). We first need some linear order $<_\exists$ on the existential variables. The particular order does not have any influence on the description of the algorithm nor on the correctness/completeness of the algorithm, thus we will not assume a specific order in this chapter.[2] To compute $\mathit{ED}(e)$, we initially set $\mathit{ED}(e)$ to the dependencies in the original formula Φ'.[3] Next, we add every existential variable x whose dependencies are contained in $\mathit{ED}(e)$ to $\mathit{ED}(e)$. If two variables e_1 and e_2 with $e_1 <_\exists e_2$ have the same dependencies we add e_1 to $\mathit{ED}(e_2)$. Overall, the extended dependencies of an existential variable e are defined as follows:

$$\mathit{ED}(e) = D_{\Phi'}(e) \cup \{x \in \mathit{var}_\exists(\Phi) \mid D_{\Phi'}(x) \subset D_{\Phi'}(e) \vee (D_{\Phi'}(x) = D_{\Phi'}(e) \wedge x <_\exists e)\}.$$

One can easily verify that if we can find Skolem functions with respect to the extended dependencies, we can construct Skolem functions with respect to the original dependencies by replacing existential variables with their Skolem function.

Next, we apply *universal reduction* [BCJ14a] to strengthen the clauses in the matrix (line 4). To define the result of applying universal reduction to φ, we first describe universal reduction of a single clause $C \in \varphi$. Let $C_\exists = \{\ell \in C \mid \mathit{var}(\ell) \in \mathit{var}_\exists(\Phi')\}$ and $C_\forall = \{\ell \in C \mid \mathit{var}(\ell) \in \mathit{var}_\forall(\Phi')\}$. Then universal reduction yields the clause $C_\exists \cup \{\ell \in C_\forall \mid \exists x \in C_\exists . \mathit{var}(\ell) \in D(\mathit{var}(x))\}$. Applying universal reduction to φ yields the formula that is obtained by applying universal reduction to each clause. We denote the resulting formula by φ and the DQBF $\mathcal{Q}\varphi$ by Φ. It is easy to see that universal reduction preserves satisfiability of a DQBF, as universal reduction does not change the clauses that can be derived with the axiom rule by ∀Exp+Res.

To compute a model, the algorithm refines a candidate model Ψ. We represent the candidate Ψ by a family of tuples $(\psi^e_D, \psi^e_F, \psi^e_A, \psi^e_{\mathit{Def}})_{(e \in \mathit{var}_\exists(\Phi))}$ together with an arbiter assignment τ, which is empty initially. Here, ψ^e_D either contains the definition, if e is

[2]In practice, we order the variables according to their occurrence in the prefix.

[3]We could also start from the reduced dependencies instead. Our initial intention for using the original dependencies was to keep the set of extended dependencies large to find more forcing clauses. While using the original dependencies does not necessarily lead to the largest possible extended dependencies, we saw that using the reduced dependencies instead (slightly) decreases the performance of our solver.

defined, or ψ_D^e is empty otherwise. If there is a definition, then $\Psi(e)$ is solely defined by the definitions, and the other entries ψ_F^e, ψ_A^e and ψ_{Def}^e are empty. Next, ψ_F^e is the set of *forcing clauses*, which we will introduce later. ψ_A^e is the set of *arbiter clauses* for e, which are defined similarly as for the two-phase algorithm. Each clause in ψ_F^e, respectively ψ_A^e, has to contain either e or $\neg e$. Thus, such a clause C can be considered as an implication $p \Rightarrow \ell$, where $var(\ell) = e$ and $p = \{\neg x \in C \mid x \neq \ell\}$. Finally, ψ_{Def}^e is the *default function* for e, i.e., a Boolean function with the domain $D(e)$. The default functions fix the existential assignment in case neither an arbiter nor a forcing clause entails an assignment for e. Thus, the default functions ensure that the candidate always corresponds to at most one Boolean function for each existential variable—later we will see that it is possible that a candidate does not allow any Boolean function for an existential variable. The default functions are determined by decision tree learning. This means that whenever we obtain a counterexample to the current candidate, and we know that the assignment of a single existential variable e needs to be flipped to repair the counterexample, we first derive the assignment of the dependencies of e in that counterexample. This assignment is then used as a sample for learning the tree. The label of the sample is the assignment e should have. In order to iteratively insert samples into the decision tree, we use Hoeffding decision trees. Whenever a new split is inserted into this decision tree, we update the default function accordingly. The idea of using machine learning for guessing Skolem functions is motivated by the success of using machine learning for Boolean function synthesis in the MANTHAN tool [GRM20; Gol+21; GRM23].

We initialize the candidate in INITCANDIDATE. Here we first compute definitions for existential variables by their extended dependencies in φ. Note that this differs from Algorithm 1, where we only allow definitions by the dependencies and arbiters. Using extended dependencies has two advantages. First, we can find more definitions, we illustrate this in Example 4.1. Second, the computed definitions are often more compact (cf. [Sli20] where a similar extension of definitions is proposed for finding more definitions).

Example 4.1. *Let $\Phi = \forall u \exists e_1(\{u\}), e_2(\{u\})(e_1 \vee \neg e_2) \wedge (\neg e_1 \vee e_2) \wedge (u \vee e_1)$ be a (D)QBF. We can see that neither e_1 nor e_2 is defined by its dependencies. Let $e_1 <_\exists e_2$ so that $e_1 \in ED(e_2)$. As e_2 is defined by e_1, there is a definition for e_2 by its extended dependencies.*

We can see that in every model of a DQBF, Skolem functions for variables defined by their extended dependencies must comply with the definitions.

Lemma 4.1. *Let Φ be a DQBF and e an existential variable that has a definition ψ by $ED(e)$ in φ. If Φ is satisfiable then in every model f the Skolem function f_e is defined by $f_e(\sigma) = \psi[\sigma \cup \rho]$, where ρ is an assignment of $ED(e) \cap var_\exists(\Phi)$ that is defined by $\rho(x) = f_x(\sigma|_{D(x)})$.*

Proof. Let Φ be a satisfiable DQBF and let e and ψ be as above. Now let f be an arbitrary but fixed model for Φ. We know that for any assignment σ of the universal variables the joint assignment $\sigma \cup f(\sigma)$ satisfies the matrix φ. From the properties of a definition, we know that $\psi[\sigma \cup f(\sigma)] = f_e(\sigma|_{D(e)})$. As ψ is a definition by $ED(e)$, we can conclude that $f_e(\sigma|_{D(e)}) = \psi[\sigma \cup \rho]$, where ρ is defined as above. $\square$

Also note that in the above description, definitions are only computed once at the beginning of the algorithm. This differs from Algorithm 1 and from the CEGIS algorithm described in [RSS21].[4] The computed definitions are then used for initializing the candidate (line 14). In addition, for each existential variable e the sets of forcing and arbiter clauses are initialized as empty sets. Furthermore, the decision trees representing the default functions are initialized as trees that only consist of a single root node, which is labeled by 1.[5] Thus, each initial decision tree represents the function that maps every input to true.

In the last step of the initialization (line 6), we further extend the extended dependencies by using the definitions. Let e be a defined existential variable, X' be the set of defined existential variables in $var(\psi^e_D)$ and X'' the set of undefined variables. We can now recursively define the *support* of ψ^e_D as the set $supp(\psi^e_D) = \bigcup_{x \in X'} supp(\psi^x_D) \cup \bigcup_{x \in X''} D(x)$. Now let $Y = D(e) \setminus supp(\psi^e_D)$. By Lemma 4.1 we know that Skolem functions in models need to comply with definitions by the extended dependencies. From this we can conclude that for any assignment σ of $supp(\psi^e_D)$ and any two distinct assignments ρ_1 and ρ_2 of Y we have $f_e(\sigma \cup \rho_1) = f_e(\sigma \cup \rho_2)$. Thus, the dependencies in Y can be considered as spurious. So for any existential variable x with $supp(\psi^e_D) \subset D(x)$, we can add e to the extended dependencies of x. Similarly, for an existential variable x with $supp(\psi^e_D) = D(x)$ such that $var(\psi^e_D) \subseteq ED(x)$, we can add e to the extended dependencies of x. We do this even if $x <_\exists e$, as all existential variables in $var(\psi^e_D)$ are contained in $ED(x)$, which guarantees that no cyclic dependencies are introduced. Because of Lemma 4.2, it is not necessary to consider these additional dependencies for the computation of definitions.

Lemma 4.2. *Let φ be a propositional formula, X and Y sets of variables from φ, and x and y variables in φ such that x has a definition ψ by X in φ. If $var(\psi) \subseteq Y$ then y is defined by Y in φ if and only if y is defined by $Y \cup \{x\}$ in φ.*

Proof. First, we can easily see that if y is defined by Y in φ, then y is also defined by $Y \cup \{x\}$ in φ. To prove the other side of the equivalence, we assume that y is defined by $Y \cup \{x\}$. Let σ_1 and σ_2 be two satisfying assignment for φ with $\sigma_1|_Y = \sigma_2|_Y$—if we cannot find such assignments, then y is defined by Y in φ. As x is defined by a subset of Y in φ this means that $\sigma_1(x) = \sigma_2(x)$. Since y is defined by $Y \cup \{x\}$ this implies that $\sigma_1(y) = \sigma_2(y)$. $\square$

[4]In this thesis, we will only consider the version of the CEGIS algorithm that computes definitions at the beginning. On the one hand, this makes the algorithm conceptually easier. And on the other hand, experiments showed that most definitions are found at the beginning and that subsequent definability checks do not pay off.

[5]Instead of initializing default functions as the constant true function, also other initialization schemes can be used. This in particular means, if a decision tree for guessing the assignment of an existential variable is already available, we can use this tree to initialize the default function of this variable.

Algorithm 4 Candidate model check.

```
1: procedure CHECKCANDIDATE($\Psi$)
2:     if ISSAT($\neg\varphi \wedge \Psi \wedge \tau$) then
3:         return FALSE                                ▷ The candidate yields a counterexample.
4:     else
5:         if ISCONSISTENT($\Psi, \tau$) then
6:             return TRUE                                       ▷ The candidate is a model.
7:         else
8:             return FALSE                          ▷ The candidate is inconsistent under $\tau$.
```

In an earlier version of the algorithm, we also identified *unate* existential literals [Aks+18], in addition to the techniques discussed above. Unate literals generalize *pure* literals—i.e., literals that either only occur positively or only negatively. A variable v is *positive unate* if $\varphi[v = 0] \wedge \neg\varphi[v = 1]$ is unsatisfiable, it is *negative unate* if $\neg\varphi[v = 0] \wedge \varphi[v = 1]$ is unsatisfiable, and it is unate if it is either positive unate or negative unate. If an existential variable e is positive unate, we can initialize the Skolem function in Ψ for e as the function that maps each input to true, similarly if e is negative unate we can initialize the Skolem function as the function that maps each input to false. Due to the definition of unates one can easily verify that no other Skolem functions need to be considered for unate variables. To compute the set of all unate variables we first negate the matrix φ then we replace each existential variable e by a new variable e' in $\neg\varphi$—we denote the resulting formula by $\neg\varphi'$. Additionally, we introduce for every existential variable e a new variable x_e and the constraint $x_e \Rightarrow (e \Leftrightarrow e')$—we denote these equivalence constraints by E. We can then make use of incremental SAT solving to find all unate existentials. For this purpose, we check for each existential variable e if the formula $\varphi \wedge \neg\varphi' \wedge E$ is unsatisfiable under to different sets of assumptions. First, we use the assumption $X_1 = \bigwedge_{g \neq e} x_g \wedge \neg e \wedge e'$ and second the assumption $X_2 = \bigwedge_{g \neq e} x_g \wedge e \wedge \neg e'$. If the encoding is unsatisfiable under the assumption X_1 then e is positive unate and if it is unsatisfiable under X_2 then e is negative unate. As in practice computing unates does not improve the performance of the algorithm, we will not consider them in the remaining part of this thesis.

4.3 Conflict Extraction

In this section, we will first describe how we can check whether the candidate gives rise to a conflict. Then we will introduce conflict graphs and show how to use them to represent conflicts.

4.3.1 Conflict Detection

After initializing a candidate the algorithm iteratively refines the candidate until the candidate is either a model or it cannot be further refined. For this purpose, we need to be able to check whether the candidate is a model. This is achieved by the checkCandidate procedure. We will refer to this check as *validity-check*. In this method, we first check whether there is some universal assignment σ such that the matrix φ is falsified under σ and the existential response of Ψ for σ and τ. If the matrix can be falsified in this way, then the current candidate can not be a model, and we say that there is a *counterexample* to Ψ. To perform this check we first represent $\neg\varphi \wedge \Psi$ by a set of clauses and then determine whether the formula is satisfiable under τ by using a SAT solver. In the following, we describe the individual components of the encoding.

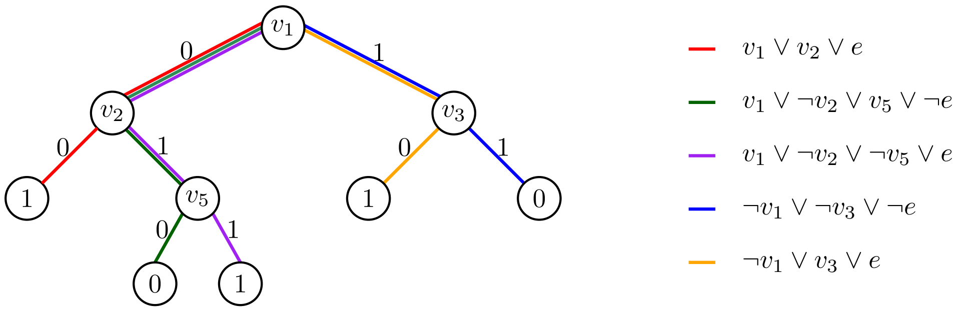
Negated matrix We use the Plaisted and Greenbaum encoding [PG86] to represent $\neg\varphi$. While this is a standard technique, we will briefly summarize the idea, as the representation of $\neg\varphi$ will be of relevance later. Let $C_1, \dots, C_n$ denote the clauses in φ. Then we first introduce a new variable f_i for each clause C_i. Next, we represent $\neg\varphi$ by $(\bigwedge_{i\in[n]} \bigwedge_{\ell\in C_i} (\neg\ell \vee f_i)) \wedge \bigvee_{i\in[n]} \neg f_i$. We will identify $\neg\varphi$ with its clausal encoding.

Definitions For each definition ψ^e_D in Ψ, we use the Tseitin transformation [Tse83] of $e \Leftrightarrow \psi^e_D$.

Forcing clauses As the assignment of an existential variable is determined by a default function in case the assignment is not entailed by any other rule in the candidate, we need to be able to check whether an existential variable is entailed by a forcing clause. For this purpose, we introduce for each forcing clause $C = C' \vee e$ ($C = C' \vee \neg e$), an *activity variable* α^e_C. This variable shall be assigned to true if and only if every literal in C' is falsified. Thus, if the assignment of an existential variable is entailed by a forcing clause, there must be an activity variable that is set to true. In order to ensure this behavior for each $\ell \in C'$, we add the clause $\neg\ell \vee \neg\alpha^e_C$ and the clause $C' \vee \alpha^e_C$. If the variable α^e_C is assigned to true, then e must be assigned to true if $C = C' \vee e$, and it must be assigned to false if $C = C' \vee \neg e$. Thus, we add the clause $\neg\alpha^e_C \vee e$ if C contains e positively and $\neg\alpha^e_C \vee \neg e$ otherwise.

Arbiter clauses Arbiter clauses are represented similarly as forcing clauses. In particular, for each arbiter clause $C = C' \vee e$ ($C = C' \vee e$), we introduce an activity variable α^e_C that shall be assigned to true if C' is falsified.

Default Activity In order to determine whether an existential variable shall be assigned with respect to its default function, we introduce a new variable d_e for each undefined existential variable e. The variable d_e shall be assigned to true if and only if the default function shall be applied. As mentioned earlier, a default function shall only determine the assignment for an existential variable e if its assignment is not yet fixed by a forcing clause or an arbiter clause. This means that we assign d_e to true if there is no activity variable for e that is assigned to true. Let $\mathcal{C} = \psi^e_F \cup \psi^e_A$. We require $d_e \Leftrightarrow \bigwedge_{C\in\mathcal{C}} \neg\alpha^e_C$. For the one side of the equivalence, we can just add the clause $\neg\alpha^e_C \vee \neg d_e$ for each clause $C \in \mathcal{C}$. Next, we have to find a suitable representation for the clause $\bigvee_{C\in\mathcal{C}} \alpha^e_C \vee d_e$. We cannot directly use this clause as



v_1
0
1
v_2
v_3
0
1
0
1
1
v_5
1
0
0
1
0
1
$v_1 \vee v_2 \vee e$
$v_1 \vee \neg v_2 \vee v_5 \vee \neg e$
$v_1 \vee \neg v_2 \vee \neg v_5 \vee e$
$\neg v_1 \vee \neg v_3 \vee \neg e$
$\neg v_1 \vee v_3 \vee e$

Figure 4.1: Clausal representation of default functions.

we would need to add additional literals to this clause for every new forcing/arbiter clause. But an already introduced clause cannot directly be modified in a SAT solver. Thus, we first introduce a new variable t_e^0 and another variable t_e^i for each clause $C_i \in \mathcal{C}$—the clauses can be sorted chronologically. Then we add the clause $d_e \vee t_0^e$ and for each clause $C_i \in \mathcal{C}$ we add $\neg t_{i-1}^e \vee \alpha_{C_i}^e \vee t_i^e$. Now let $m = |\mathcal{C}|$. We can see that if t_m^e is set to be false then for every $i \in [m]$ the variable t_i^e must be assigned to false if for each $j \in [m]$ with $i < j$ the variable $\alpha_{C_i}^e$ is assigned to false. Thus, if for every $i \in [m]$ the variable $\alpha_{C_i}^e$ is assigned to false then d_e must be assigned to true in order to satisfy the clauses. If, on the other hand, some variable $\alpha_{C_i}^e$ is assigned to true, then t_0^e can be assigned to true, which in turn does not pose a constraint on d_e.

Default functions As mentioned above, default functions are given by Hoeffding decision trees. To represent these trees let us denote the set of paths from the root of the tree to positively labeled leaves by P^+ and the set of paths to negatively labeled leaves by P^-. The default function is then represented by introducing, for each $p \in P^+$ a clause $\bigvee_{x \in p} \neg x \vee \neg d_e \vee e$ and for each $p \in P^-$ a clause $\bigvee_{x \in p} \neg x \vee \neg d_e \vee \neg e$. By adding $\neg d_e$ to these clauses, they only constrain the assignment of e in case every activity variable is set to false. We illustrate this in Figure 4.1.

We will denote the above encoding by $\mathit{validity}(\Psi, \varphi)$. Moreover, we will refer to a total assignment of $\mathit{var}(\mathit{validity}(\Psi, \varphi))$ that satisfies $\mathit{validity}(\Psi, \varphi)$ as a *conflicting assignment* for Ψ, or *conflict* for short. We can see that if there is a satisfying assignment δ for the above encoding, then there is a universal assignment $\sigma = \delta|_{var_\forall(\Phi)}$ such that the response of Ψ under τ for σ falsifies the matrix. As discussed above, this means that Ψ is not a model.

Assume that the above encoding is unsatisfiable. This does not immediately mean that the candidate Ψ is a model. It is possible, that for some assignment σ to the universal variables there is no assignment δ for the existential variables such that $\sigma \cup \delta \cup \tau$ satisfy the encoding of Ψ. This means that Ψ does not allow any response under σ and τ. In this case, we say that Ψ is *inconsistent*. One can easily see that this is only possible if there are two forcing or arbiter clauses for an existential variable e that both are active, but one clause implies e and the other clause implies $\neg e$. If Ψ is inconsistent, it can obviously not represent a model. To check consistency, we apply the method isConsistent. Similarly to the validity-check, we also make use of a SAT solver for the consistency check. To construct the SAT-encoding, we use the encoding for Ψ as a starting point. We now want to check if there is some existential variable e that needs to be assigned both to true and false according to Ψ. For this purpose, we use a *dual rail* [Bry+87] modification of the encoding of Ψ. The core idea of the encoding is to introduce for every existential variable e two new variables e^+ and e^-. The intention behind these variables is that e^+ shall be assigned to true if and only if e is forced to be true in Ψ and e^- shall be assigned to true if e is forced to be false. In the following we will discuss the components of the encoding.

Definitions For each definition ψ_D^e in Ψ we first introduce an auxiliary variable β_e. Next we introduce the Tseitin transformation of $\beta_e \Leftrightarrow \psi_D^e$. In this encoding, we replace each existential literal ℓ with $var(\ell) = x$ by x^+ if $\ell = x$ and by x^- otherwise.[6] Moreover, we add clausal representations of $e^+ \Leftrightarrow \beta_e$ and $e^- \Leftrightarrow \neg\beta_e$.[7]

Forcing clauses Let $C = C' \vee e$ ($C = C' \vee \neg e$) be a forcing clause. Assume that $C' = \{\ell_1, \ldots, \ell_m\}$ and let $v_i = var(\ell_i)$ for $i \in [m]$. Next we introduce for each $i \in [m]$ a literal ℓ'_i that is defined as follows. If v_i is a universal variable we set $\ell'_i = \ell_i$. Otherwise, we define $\ell'_i = v_i^-$ if $\ell = v_i$ and we define $\ell'_i = v_i^+$ if $\ell = \neg v_i$. Similarly, as for the validity-check we introduce an activity variable α_C^e for C. The activity variables are constrained by the clauses $\ell'_i \vee \neg\alpha_C^e$ for $i \in [m]$ and $\neg\ell'_1 \vee \ldots \vee \neg\ell'_m \vee \alpha_C^e$. Note that C can only be active if all literals in C' are falsified. In case $\ell_i = v_i$, the literal is falsified if v_i is assigned to false, i.e., v_i^- needs to be assigned to true. Thus, we define $\ell'_i = v^-$ instead of $\ell'_i = v^+$ in this case. Finally, we add the clause $\alpha_C^e \vee e^+$ if C contains e positively and $\alpha_C^e \vee e^-$ otherwise.

Arbiter clauses Arbiter clauses can be represented analogous to forcing clauses.

Default functions Similarly, as in the encoding used for the validity-check, the activity variables can be used to introduce a variable d_e that is true if and only if the default function shall be used. Moreover, let P^+ and P^- be defined as before. We now introduce new variables δ_e^+ and δ_e^-. The variable δ_e^+ shall be assigned to true if e is set to true by the default function and δ_e^- shall be assigned to true if e is set to false by the default function. For this purpose, we introduce for each $p \in P^+$ a

[6]Suppose an existential variable e is defined by the formula $x \wedge u$, where x is an existential and u a universal variable. This definition would result in clauses $\neg\beta_e \vee x^+$, $\neg\beta_e \vee u$ and $\beta_e \vee \neg u \vee x^-$. Therefore, if the candidate does not yield an inconsistency for x and x gets assigned to true, then x^+ is assigned to true and x^- is assigned to false. We can see that in this case the clauses ensure that β_e is assigned to true if and only if u is assigned to true. Similarly, if x is set to false then x^+ is assigned to false and x^- is assigned to true, which means that β_e is assigned to false. If there is an inconsistency for x then both x^- and x^+ are assigned to true, which means that β_e can be assigned either to true or to false.

[7]Actually, it is not necessary to introduce e^+ and e^- for defined variables, as a defined variable is always uniquely determined and can thus not cause an inconsistency. Nevertheless, we decided to use this representation as it requires fewer distinctions between defined and undefined variables, which should make the encoding a bit simpler.

clause $\bigvee_{x \in p} \neg x \vee \neg d_e \vee \delta_e^+$ and for each $p \in P^-$ a clause $\bigvee_{x \in p} \neg x \vee \neg d_e \vee \delta_e^-$. Later we will need the variables δ_e^+ and δ_e^- in order to determine if e was set by a default function. Finally, we add the clauses $\neg\delta_e^+ \vee e^+$ and $\neg\delta_e^- \vee e^-$.

Enforcing an inconsistency To constrain the variables e^+ and e^- for an undefined existential e, we proceed as follows: Let $\mathcal{C}^+$ denote the set of forcing and arbiter clauses that contain e positively and $\mathcal{C}^-$ the corresponding set of clauses that contain e negatively. We add the clauses $\neg e^+ \vee \delta_e^+ \vee \bigvee_{C \in \mathcal{C}^+} \alpha_C$ and $\neg e^- \vee \delta_e^- \vee \bigvee_{C \in \mathcal{C}^-} \alpha_C$.[8] These clauses ensure that e^+ (e^-) is set to false in case neither a forcing clause nor an arbiter clause nor the default functions set e to true (false). Finally, we introduce a new variable c_e for each existential variable and add a clausal representation of $c_e \Leftrightarrow (e^+ \wedge e^-)$ and the clause $\bigvee_{e \in var_\exists(\Phi)} c_e$. The last constraint ensures that the encoding can only be satisfied if there are two rules in the current candidate that simultaneously require that some existential variable needs to be assigned both to true and false.

We will denote the above encoding by *consistency*(Ψ). Moreover, we will denote a total assignment of *var*$($*consistency*$(\Psi))$ that satisfies *consistency*(Ψ) as a *conflicting assignment* for Ψ, or *conflict* for short. If the above encoding is satisfiable we can conclude that we can find a universal assignment σ such that Ψ is inconsistent under σ and the arbiter assignment τ. If, on the other hand, the encoding is unsatisfiable, we can conclude that Ψ is consistent, i.e., Ψ describes a Boolean function for each existential variable. Thus, Ψ describes a model of Φ.

4.3.2 Conflict Graphs

If the verification of the candidate fails, i.e., either the validity check or the consistency check fails, we want to find a reason for the conflict. To do this, we introduce *conflict graphs*. These graphs allow to explain why a particular existential variable got a specific assignment. The presented conflict graphs are closely related to *implication graphs* for SAT [SS99; Zha+01].

We will first discuss the construction of *conflict graphs* for counterexamples. Then we will consider the very similar construction for inconsistencies.

In a conflict graph, each vertex corresponds either to a universal or an existential literal, for this reason we will identify literals and vertices.

In the following, let us assume that we found a counterexample, i.e., there is a conflict σ satisfying *validity*(Ψ, φ). This means for some $i \in [n]$, the assignment σ falsifies the variable f_i, and thus it falsifies the clause $C_i \in \varphi$. We now define the *conflict graph* $\mathcal{C}$ for σ as follows. For each $\ell \in C_i$ the graph $\mathcal{C}$ shall contain the vertex $\neg\ell$. We denote these vertices as the *sink vertices* of $\mathcal{C}$ (*sinks*$(\mathcal{C})$). Next, if $\mathcal{C}$ contains an existential literal ℓ with

[8]For the clausal representation, we use a similar idea as for the default activity in the validity-check, in order to easily extend a clause in a SAT solver.

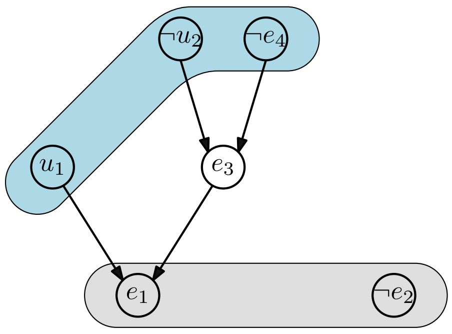


Figure 4.2: The figure illustrates the construction of a conflict graph for a DQBF Φ that contains universal variables u_1 and u_2, existential variables e_1, e_2, e_3 and e_4, and a clause $\neg e_1 \vee e_2$. For this example we do not care about the exact definition of Φ. But let us assume that e_3 is defined by $\neg u_2 \wedge \neg e_4$, that $\{u_1, e_3\} \subseteq ED(e_1)$ and that $D(e_4) = \{u_1, u_2\}$. Additionally, there shall be forcing clauses $\neg u_1 \vee \neg e_3 \vee e_1$ and $\neg e_2$—a forcing clause with an empty premise—and an arbiter clause $e_4^{u_1,\neg u_2} \vee \neg u_1 \vee u_2 \vee \neg e_4$. The arbiter assignment shall assign $e_4^{u_1,\neg u_2}$ to false. Then an assignment σ with, $\sigma(u_1) = 1$, $\sigma(u_2) = 0$, $\sigma(e_1) = 1$, $\sigma(e_2) = 0$, $\sigma(e_3) = 1$ and $\sigma(e_4) = 0$ is a counterexample. The figure illustrates the corresponding conflict graph. In this illustration the sink vertices are highlighted in gray and the source vertices are highlighted in blue.

$var(\ell) = e$ with a definition ψ_D^e in Ψ, then $\mathcal{C}$ shall contain for each variable $v \in var(\psi_D^e)$ a vertex w where $w = v$ if $\sigma(v) = 1$ and $w = \neg v$ otherwise. Additionally, we require that for each such vertex w there is an edge (w, ℓ) in $\mathcal{C}$. If $\mathcal{C}$ contains an existential literal ℓ with $var(\ell) = e$ such that for some forcing clause C for e the activity variable α_C^e is assigned to true by σ, then $\mathcal{C}$ shall contain a vertex $\neg k$ for each literal $k \in C \setminus \{\ell\}$. Additionally, we require that for each $k \in C \setminus \{\ell\}$ there is an edge $(\neg k, \ell)$ in $\mathcal{C}$. The graph $\mathcal{C}$ shall not contain any other edges or vertices. We denote the set consisting of the universal variables and the undefined existential variables without an active forcing clause as the *source vertices* of $\mathcal{C}$ ($sources(\mathcal{C})$). We illustrate the construction of a conflict graph for a counterexample in Figure 4.2.

For the inconsistency check we proceed similarly. We assume that we found an inconsistency, i.e., there is a conflict σ satisfying $consistency(\Psi)$. We know that there must be some existential variable e such that σ satisfies both e^+ and e^-. The conflict graph $\mathcal{C}$ shall contain the literals e and $\neg e$—we will refer to these literals as the *sink vertices*. The remaining construction of $\mathcal{C}$ is similar to the construction of conflict graphs for counterexamples. The only difference is that for an active forcing clause we add a vertex v if the clause contains v^+, and we add the vertex $\neg v$ otherwise.

Remark 4.1. *Let ℓ be a vertex in a conflict graph, X be the set of all vertices with an outgoing edge to ℓ and ρ an assignment that satisfies all literals associated to the vertices in X. Then the construction of the conflict graph ensures that the candidate enforces the literal ℓ to be assigned to true under ρ. If we would also add edges for arbiter clauses and we would still like to preserve this property then we would need to also introduce vertices for arbiter literals. Later we will see that the main purpose of a conflict graph is to find*

Algorithm 5 Conflict graph analysis and candidate refinement.

```
1: procedure REFINECANDIDATE($\Psi, \Delta, \mathcal{C}, \sigma$)
2:     $X \leftarrow$ GETSOURCEVERTICES($\mathcal{C}$)
3:     if CONTAINSFORCEDLITERAL($X$) then
4:         $\ell \leftarrow$ GETFORCEDLITERAL($X$)
5:         $S \leftarrow$ GETSEPARATOR($\mathcal{C}, \ell$)
6:         $S' \leftarrow$ REDUCE($S \cup \{\ell\}$)
7:         if CONTAINSFORCEDLITERAL($S'$) then
8:             ADDFORCINGCLAUSE($\Psi, S'$)
9:             UPDATEDEFAULT($\Psi, S'$)
10:        else
11:            ADDARBITERCLAUSE($\Psi, \Delta, S', \sigma$)
12:    else
13:        ADDARBITERCLAUSE($\Psi, \Delta, X, \sigma$)
```

compact forcing clauses. In our experience it is not helpful to allow arbiter literals in forcing clauses. For this reason we do not want to have arbiter literals in the conflict graph, and so we do not add any incoming edges for a literal that is entailed by an arbiter clause.

Remark 4.2. *If there is some existential variable e with a forcing clause consisting only of the literal e or $\neg e$, then the corresponding vertex in the conflict graph does not have an incoming edge, but still it is not a source vertex.*

Finally, let σ be satisfying assignment of $validity(\Psi, \varphi)$ in case we found a counterexample, respectively a satisfying assignment of $consistency(\Psi)$ in case we found an inconsistency. In the GETCONFLICT procedure we then return a conflict graph $\mathcal{C}$ constructed from σ and the projection of σ to the universal variables, i.e., $\sigma|_{var_\forall(\Phi)}$.

4.4 Conflict Refinement

If CHECKCANDIDATE returns FALSE we know that the candidate Ψ is not a model. In this case, we use the conflict graph $\mathcal{C}$ to refine the candidate Ψ (REFINECANDIDATE). For the refinement we use two different approaches: we either use *forcing clauses* or *arbiter clauses*. Subsequently, we will first describe the general idea of the refinement. Then we will discuss the forcing clauses and arbiter clauses. After the discussion of forcing clauses and arbiter clauses we will give an example that shows why arbiter clauses alone, are not enough in practice.

To refine the candidate, we first retrieve the source vertices from $\mathcal{C}$ (line 2). We denote the existential literals in $sources(\mathcal{C})$ by $X_\exists$ and the universal literals by $X_\forall$. Now let σ be an arbitrary total assignment of the universal variables that satisfies each literal in $X_\forall$. By the construction of $\mathcal{C}$ one can conclude that whenever the existential response by Ψ

satisfies $X_\exists$ under σ we either obtain a counterexample or an inconsistency. Thus, to refine Ψ we have to modify the Skolem functions for the variables in $X_\exists$. For this purpose, we use the already mentioned forcing and arbiter clauses. We introduce forcing clauses if we can conclude that a particular literal in $X_\exists$ needs to be assigned differently and arbiter clauses otherwise.

4.4.1 Forcing Clauses

First, suppose we can find a literal $\ell \in \mathit{source}(\mathcal{C})$ such that, for each other existential literal $j \in \mathit{source}(\mathcal{C})$, we have $\mathit{var}(j) \in \mathit{ED}(\mathit{var}(\ell))$. In this case, we call ℓ a *forced literal*, and we want to introduce a forcing clause for $\mathit{var}(\ell)$. Throughout this subsection, we will assume that there is such a source literal ℓ. In the algorithm this is checked by the method containsForcedLiteral. Now the idea is that we want to modify Ψ such that for every assignment satisfying each literal in $\mathit{source}(\mathcal{C}) \setminus \{\ell\}$, the literal ℓ gets falsified. This could be realized by adding the clause $C = \{\neg x \mid x \in \mathit{sources}(\mathcal{C})\}$ to the set of forcing clauses of $\mathit{var}(\ell)$. But we have to remember that the set for forcing clauses is used to represent the Skolem function for $\mathit{var}(\ell)$, thus, the clause must not contain variables not included in $\mathit{ED}(\mathit{var}(\ell)) \cup \{\mathit{var}(\ell)\}$. By the initial condition this is true for every existential literal, but not necessarily for every universal literal. To fix this we apply universal reduction on C to obtain a clause C'. We can then add C' to the set of forcing clauses. The intention behind this approach is that Ψ can only ensure the satisfiaction of C if we ensure that C' is satisfied since the Skolem function cannot depend on any additional variables from C.

To improve this procedure, we argue that we do not necessarily need to take the source literals as the forcing clause. Instead, we could also consider a separator S of the source and sink vertices. This separator shall contain an existential source k, such that for every existential literal $x \in S$ with $x \neq k$, we have $\mathit{var}(x) \in \mathit{ED}(\mathit{var}(k))$. The separator then gives a forcing clause for $\mathit{var}(k)$. We require that k is a source as we do not want to introduce forcing clauses for non-sources as for these variables we already know how they must be assigned. Thus forcing clauses for non-sources would introduce inconsistencies. One can verify that for any edge (ℓ_1, ℓ_2) in the conflict graph, where both ℓ_1 and ℓ_2 are existential literals, we must have $\mathit{ED}(\mathit{var}(\ell_1)) \subset \mathit{ED}(\mathit{var}(\ell_2))$. Therefore, for every existential vertex x in the separator S, and every existential vertex y in the conflict graph that has a path to x, we know that $\mathit{ED}(\mathit{var}(y)) \subset \mathit{ED}(\mathit{var}(x))$. This means that k and ℓ—the existential source that contains all the other existential sources in its extended dependencies—must coincide.

The process of finding a forcing clause to refine a conflict is closely related to learning clauses from implication graphs [MLM21] to repair conflicts in SAT. Here, the task is also to find a clause that prevents the current conflict from arising in the future. For this purpose, a separator of the decision variable and the conflicting literals needs to be found. Different techniques for finding this separator have been considered [Zha+01]. Unfortunately, strategies making use of unique implication points (UIPs) cannot be directly applied to the conflict graphs considered in this thesis as we do not have any notion of decision levels—we do not assign variables one by one. Instead, we make use of minimum size separators. While for SAT the strategy of computing minimum separators seems to be inefficient [Zha+01], it is a viable approach in our setting. On the one hand, these separators allow a compact representation of repairs of conflicts. On the other hand, the relative time needed to compute these separators is much lower for us compared to SAT—we already have up to two SAT calls for obtaining a conflict. Thus, the relative overhead of computing minimum size separators might be smaller compared to SAT.

To sum up the above considerations, we want to find a set of vertices $S \subseteq V(\mathcal{C})$ with the following properties.

- S is a separator for $\mathit{sources}(\mathcal{C})$ and $\mathit{sinks}(\mathcal{C})$ in $\mathcal{C}$
- S shall contain the vertex ℓ
- For every existential literal $x \in S$ with $x \neq \ell$ we require $\mathit{var}(x) \in \mathit{ED}(\mathit{var}(\ell))$
- There shall not be any other set of vertices S' with the above properties and $|S'| < |S|$

In order to ensure that the separator contains the forced literal ℓ, we want to remove it from $\mathcal{C}$ first and then add it to a separator.

Lemma 4.3. *Let G be a graph and $A, B \subseteq V(G)$ be sets of vertices and $v \in A$ be a vertex. If S is a minimum separator for $A \setminus \{v\}$ and $B \setminus \{v\}$ in $G - \{v\}$ then $S' = S \cup \{v\}$ is a separator for A and B in G and there is no smaller separator containing v.*

Proof. Let S be as above. We can easily verify that $S' = S \cup \{v\}$ is a separator for A and B in G. Now assume there is a separator S'' for A and B in G that contains v and $|S''| < |S'|$. We can see that removing v from S'' yields a separator for $A \setminus \{v\}$ and $B \setminus \{v\}$ in $G - \{v\}$ that is smaller than S. As S was assumed to be a minimum separator, this yields a contradiction. □

We can first compute a minimum separator S for $\mathit{sources}(\mathcal{C}) \setminus \{\ell\}$ and $\mathit{sinks}(\mathcal{C}) \setminus \{\ell\}$ in $\mathcal{C} - \{\ell\}$. Then by the above lemma there is no smaller separator than $S \cup \{\ell\}$ for $\mathit{sources}(\mathcal{C})$ and $\mathit{sinks}(\mathcal{C})$ that contains ℓ. In order to compute a forcing clause, we also need to ensure that every existential literal in the separator, different from the forced literal ℓ, is contained in $\mathit{ED}(\mathit{var}(\ell))$. This can be achieved by proceeding as follows. Let G be a graph and $A, B, F \subseteq V(G)$ be sets of vertices. To compute a minimum size separator for A and B that is disjoint from F, we can first remove the vertices F from G and then compute a separator in the obtained graph. We illustrate this in Figure 4.3. We can show that a minimum size separator in the resulting graph is also a minimum size separator for A and B, disjoint from F.

Lemma 4.4. *Let G be a graph and $A, B, F \subseteq V(G)$ be sets of vertices s.t. $A \cap F = \emptyset$. Moreover, for every vertex $f \in F$ and every path P from f to some vertex $b \in B$, the*

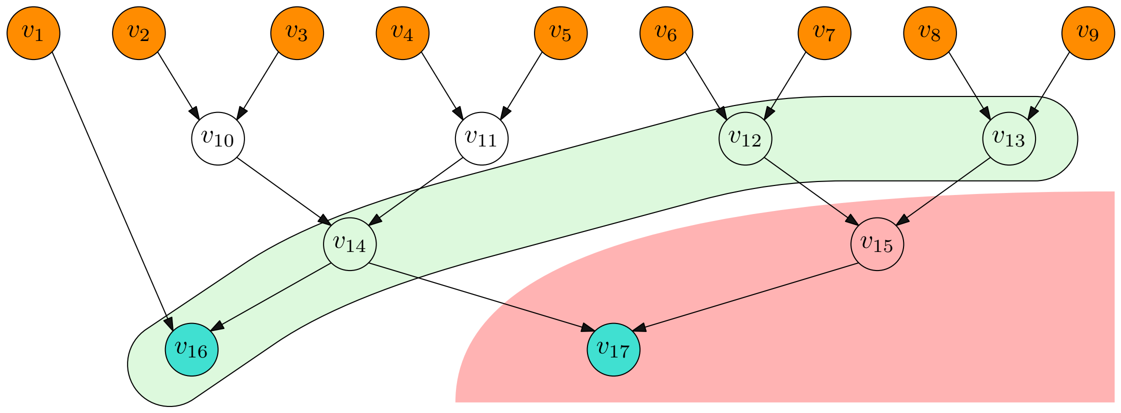


Figure 4.3: The figure shows a graph G for which we want to compute a separator of the orange and the turquoise marked vertices. The separator shall not contain any vertex from the red marked area. To compute such a minimum separator, we can compute a minimum separator for the orange and the green marked vertices in $G - \{v_{15}, v_{17}\}$.

path P shall be contained in F. We define X as the set of all vertices $v \in V(G) \setminus F$, s.t. there is a vertex $f \in F$ with $(v, f) \in E(G)$ and f is connected to some $b \in B$. Moreover, we define $Y = (B \setminus F) \cup X$. If S is a minimum separator for A and Y in $G - F$ then S is also a separator for A and B in G and there is no other separator S' with $S' \cap F = \emptyset$ and $|S'| < |S|$.

Proof. First we show that S is a separator for A and B. Let $a \in A$ and $b \in B$ be arbitrary but fixed. We have to show that for any path P from a to b, the path P contains a vertex from S. For this purpose, we need to distinguish between two cases. 1) If $b \in Y$, i.e., $b \in B \setminus F$ then we can conclude from the properties of F that there is no path from a to b that contains a vertex of F—otherwise b would be contained in F. Thus, as S is a separator for A and Y in $G - F$ we can conclude that any path from a to b contains a vertex from S. 2) Otherwise, if $b \notin Y$, i.e., $b \in B \cap F$ then P must contain an edge (v_1, v_2) with $v_1 \notin F$ and $v_2 \in F$. We can see that v_1 must be contained in Y. As there cannot be a path from a to v_1 that contains a vertex from F, we can conclude that S contains a vertex from each path from a to v_1.

Now assume that there is a smaller separator S' with $S' \cap F = \emptyset$. We can verify that such a separator S' would also be a separator for A and Y in $G - F$. But this is not possible as S was assumed to be a minimum separator for A and Y in $G - F$ and $|S'| < |S|$. □

Now let F be the set of existentials x in $\mathcal{C}$ different from ℓ such that $var(x) \notin ED(var(\ell))$. We can see that if there is a path from an existential literal x_1 to another existential literal x_2 then we must have $ED(var(x_1)) \subseteq ED(var(x_2))$. Thus, if an existential x is contained in F then also all its successors in $\mathcal{C}$ are contained in F. We illustrate this in Figure 4.4. In the following we denote the graph $\mathcal{C} - (F \cup \{\ell\})$ by G the set of vertices $sources(\mathcal{C}) \setminus \{\ell\}$ by A. Moreover, we denote the set $Y \setminus \{\ell\}$, where Y is defined as in Lemma 4.4, by B. As the forced literal ℓ is chosen such that for each existential source

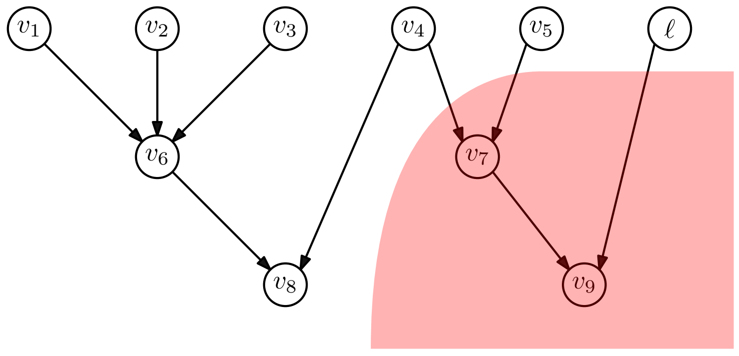


Figure 4.4: Conflict graph with forced literal ℓ. The marked vertices must not be contained in a separator as $var(v_7) \notin ED(var(\ell))$, and $var(v_9) \notin ED(var(\ell))$.

x with $x \neq \ell$ we have $var(x) \in ED(var(\ell))$ we also know that $A \cap F = \emptyset$. This means that F fulfills the condition from Lemma 4.4. So to compute a separator that does not contain any vertex from F by Lemma 4.4, we can compute a minimum separator for A and B in G instead. We assume, that for every vertex v in G, there is some vertex $a \in A$ such that there is a path from a to v. Obviously, removing vertices without such a path has no influence on the separators of the graph.

To compute the separator for A and B in G we first represent G by a flow network N and then compute the separator from a maximum flow in N. For this purpose, we introduce a graph G' which is defined as follows. For every vertex v in G the graph G' shall contain two vertices v' and v'' and there shall be an edge (v', v'') in G'. We will denote such an edge as *inner edge* for v. If G contains an edge (v_1, v_2) then G' shall contain the edge (v_1'', v_2'). Additionally, G' shall contain two new vertices s and t. For each $a \in A$, we require that G' contains the edge (s, a') and for each $b \in B$ the edge (b'', t). The capacity c of N shall assign each inner edge to 1 and every other edge to 2. We illustrate the generation of a flow network in Figure 4.5.

A related approach was considered by Even and Tarjan [ET75]. They used maximum flows to determine the connectivity (the minimum size of a separator) of undirected graphs.

Now let f be a maximum flow for N. The maximum flow can for example be computed by the Boykov Kolmogorov max flow algorithm [BK04].[9] To construct a separator from the flow f and to show that it has minimum size, we will first introduce a cut of the flow network N. For this purpose, we define X as the set of all vertices in the residual graph G'_f that are reachable from s. Additionally, we define $\overline{X} = V(G') \setminus X$. By [Cor+09, Max-flow min-cut theorem] we can conclude that t is not contained in X. Thus, $\chi = (X, \overline{X})$ is a cut for N. We now want to obtain a minimum separator from this cut. For this purpose, we first need the following property.

[9]We use the Boykov Kolmogorov algorithm as in our implementation we use the *Boost* C++ library for computing max flows. Among the max flow algorithms implemented in *Boost* the Boykov Kolmogorov algorithm yielded the best results.

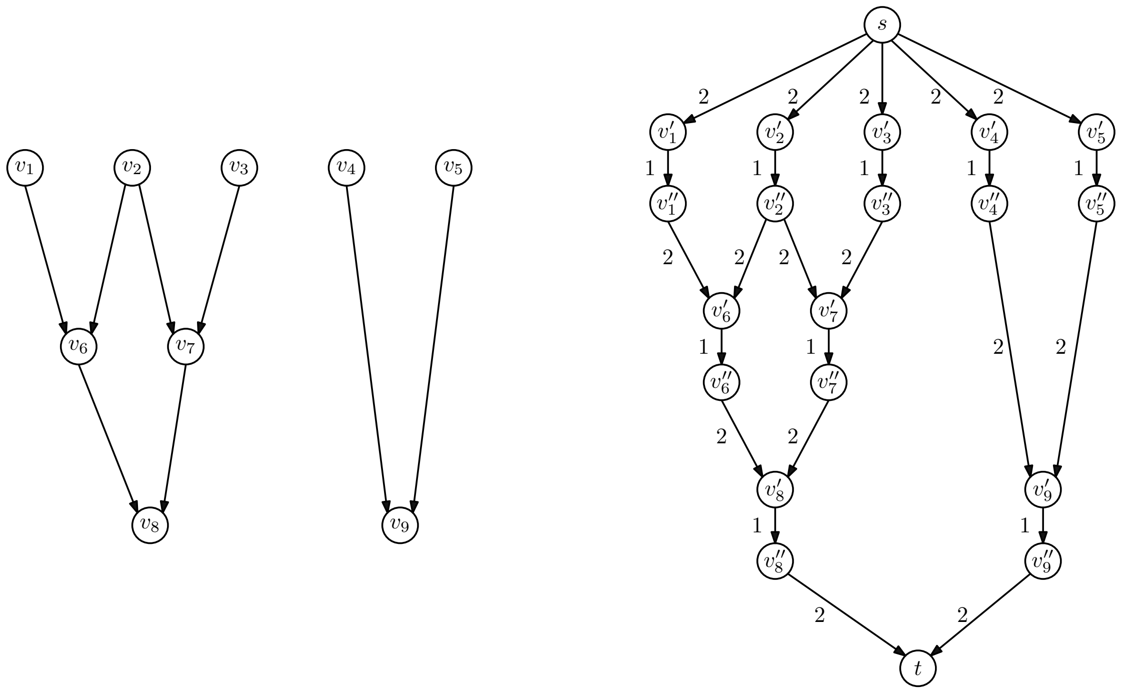


Figure 4.5: A reduced conflict graph and its associated flow network. The labels for the edges in the right graph correspond to the capacities of the edges in the flow network.

Lemma 4.5. *Let X and $\overline{X}$ be as above. Moreover, let $e = (x_1, x_2)$ be an edge in G' such that $x_1 \in X$ and $x_2 \in \overline{X}$. Then e is an inner edge.*

Proof. In the following let $e = (x_1, x_2)$ be an arbitrary but fixed edge from X to $\overline{X}$. First, we can see that $f(e) = c(e)$—otherwise x_2 would be contained in X. Now assume that e is not an inner edge. There are three possibilities to consider. First, assume $x_1 = v_1''$ and $x_2 = v_2'$ for some vertices v_1'' and v_2'. We know that the only incoming edge for v_1'' is the edge $e' = (v_1', v_1'')$. As $c(e') = 1$ we know $f(e') \leq 1$. By the flow preservation property, we can conclude that $f(e) \leq 1 < c(e) = 2$. This yields a contradiction as we must have $f(e) = c(e)$. For the cases $x_1 = s$ and $x_2 = v'$, where v' is some vertex, respectively $x_1 = v''$ and $x_2 = t$, where v'' is some vertex, we can show a contradiction analogously. □

By Lemma 4.5 we know that each edge from X to $\overline{X}$ can be associated with some vertex v from G. We can now define S as the set of vertices corresponding to the inner edges between X and $\overline{X}$. It can be easily verified that S is a separator. The construction is illustrated in Figure 4.6.

Lemma 4.6. *The set S is a separator for A and B.*

Proof. We assume that S is not a separator. Then there is a path P from some $a \in A$ to some $b \in B$ s.t. P contains no vertex from S. From P we can construct a path P'

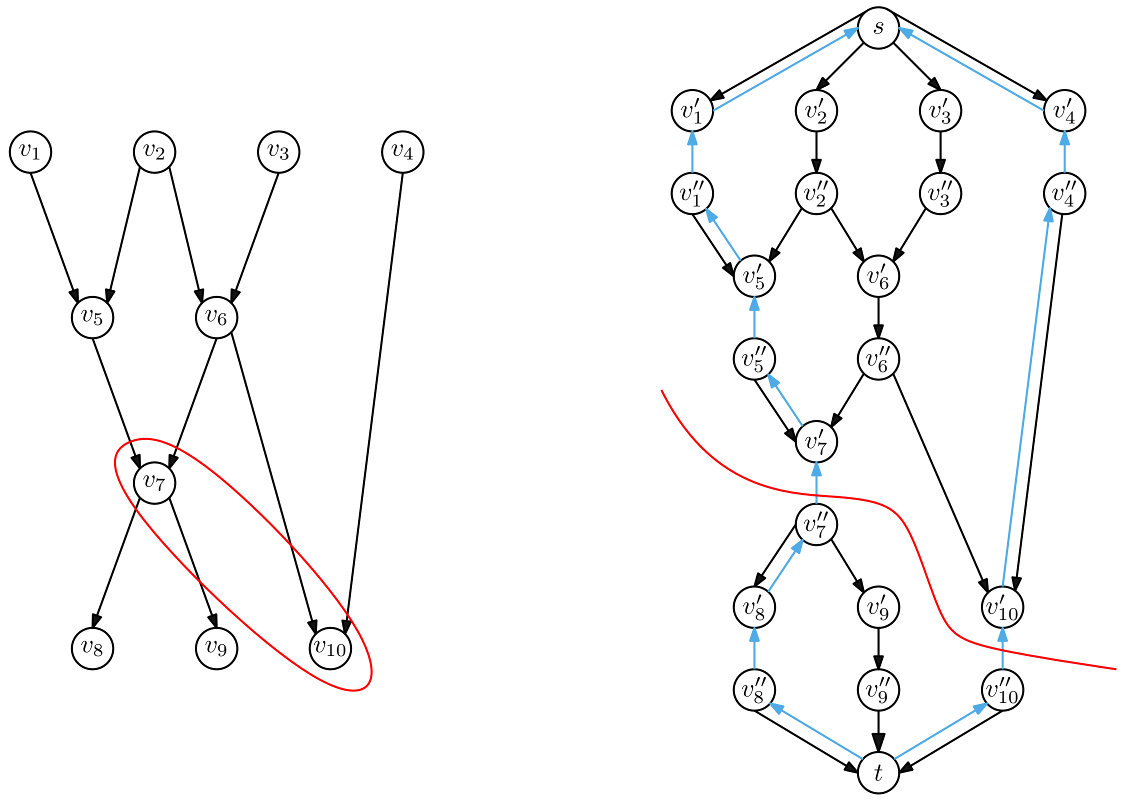


Figure 4.6: A reduced conflict graph and an associated residual graph for a maximum flow f. The flow f assigns each edge of the path $s, v'_1, v''_1, v'_5, v''_5, v'_7, v''_7, v'_8, v''_8, t$ and each edge on the path $s, v'_4, v''_4, v'_{10}, v''_{10}, t$ to 1 and all other edges to 0. This means that for every edge (a, b) with an upwards direction in the illustration of the residual graph we have $f(b, a) = 1$. We have highlighted these edges in blue for easier identification. On the left side a minimum size separator for $\{v_1, v_2, v_3, v_4\}$ and $\{v_8, v_9, v_{10}\}$ is marked, whereas on the right side the cut obtained from the residual graph is marked.

from s to t in G'. As P contains no vertex from S, the path P' does not contain any edge e with $c(e) = f(e)$. But this means that f is not a maximum flow. □

To show that S has minimum size we need the following property first.

Lemma 4.7. *Let X and $\overline{X}$ be as above. Moreover, let $e = (x_2, x_1)$ be an edge in G' such that $x_1 \in X$ and $x_2 \in \overline{X}$. Then $f(e) = 0$.*

Proof. As $x_1 \in X$ there is a path from s to x_1 in G'_f. Now suppose that $f(e) \neq 0$. Then the edge (x_1, x_2) is contained in G'_f. But this means that there is a path from s to x_2 in G'_f. Thus, we can conclude $x_2 \in X$, which yields a contradiction. □

An immediate consequence of the above lemma is the following corollary.

Corollary 4.1. *The net-flow of χ is equal to the capacity of χ.*

Proof. As we saw in the proof for Lemma 4.5 we know that for every edge e from X to $\overline{X}$ we have $f(e) = c(e)$. By applying Lemma 4.7 we can see that $f(\chi) = c(\chi)$ □

To show that there cannot be any separator S' with $|S'| < |S|$, we want to show that if there would be such an S' then there would be a cut whose capacity is smaller than $c(\chi)$, which is not possible.

Definition 4.1. Let T be a separator for A and B in G. Then we define the cut $\mathit{cut}(T) = (Y, \overline{Y})$ as follows. For every $x \in T$ the set $\overline{Y}$ shall contain the vertex x''. Additionally, for every path P in G from $x \in T$ to a vertex $b \in B$ with $P \cap T = \{x\}$, we require that for any vertex v different from x in P the vertices v' and v'' are contained in $\overline{Y}$. Finally, $\overline{Y}$ shall contain the vertex t. $\overline{Y}$ shall not contain any other vertices. We define Y as $V(G') \setminus \overline{Y}$. We can see that $s \notin Y$, thus $(Y, \overline{Y})$ is a cut of N.

Lemma 4.8. *Let T be separator for A and B in G and $(Y, \overline{Y}) = \mathit{cut}(T)$. Then every edge e from Y to $\overline{Y}$ is an inner edge.*

Proof. We assume the opposite. First, we can see that there cannot be an edge (s, v') from Y to $\overline{Y}$—the definition of $\mathit{cut}(T)$ ensures that for every vertex $v \in A$ only v'' and not v' can be contained in $\overline{Y}$. So assume there is an edge $e = (v_1'', v_2')$ from Y to $\overline{Y}$. Then there must be some $x \in T$ s.t. there is a path P from x to $b \in B$ containing v_2 but no vertex from T other than x. From this we can conclude that there is no $y \in T$ s.t. there is a path from y to v_1—otherwise v_1 would be contained in $\overline{Y}$. We assume that for every vertex v in G there is some $a \in A$ s.t. a and v are connected. This means there is some $a \in A$ and $b \in B$ such that there is a path from a to b not containing any vertex from T. This means that T is not a separator. Last but not least assume that there is an edge $e = (v'', t)$ from Y to $\overline{Y}$. Similarly, as before we can conclude that there is no $x \in T$ such that x and v are connected. Thus, we can again find a path from some $a \in A$ to some $b \in B$ not containing any vertex from T. Which yields a contradiction again. □

We can now prove that S has minimum size.

Theorem 4.1. *Let S be defined as above. Then S is a separator of minimum size for A and B.*

Proof. By Lemma 4.6 S is a separator for A and B. It remains to be shown that S has minimum size. Assume the opposite. This means there is a separator S' for A and B with $|S'| < |S|$. By applying Lemma 4.5 we can conclude that $c(\chi) = |S|$ and by Lemma 4.8 $c(\mathit{cut}(S')) = |S'|$. From [Cor+09, Corollary 26.4] we know that for any flow f and any cut C we have $|f| = f(C)$. Additionally, by [Cor+09, Corollary 26.5] we know that for any flow f and any cut C we have $|f| \leq c(C)$. In Corollary 4.1 we showed that the capacity of χ is equal to its net-flow. All in all, this allows to conclude

$$|f| \leq c(\mathit{cut}(S')) = |S'| < |S| = c(\chi) = f(\chi) = |f|.$$

But this yields a contradiction. □

Now assume we found a separator S for the sources and the sinks in the conflict graph with the properties stated above. Then we can introduce a forcing clause $\{\neg x \mid x \in S\}$ for $\mathit{var}(\ell)$. But we can potentially still do better as S might contain literals that are not needed to derive a conflict. Remember that the conflict graph is constructed for a single falsified clause, respectively, a single inconsistent literal. Thus, there might be other falsified clauses / inconsistencies we can make use of.

If we find a separator S, then the formula $\varphi \wedge \Psi \wedge S$ is unsatisfiable.[10] The construction of $\mathcal{C}$ and S ensure that by assuming S, we replicate the current conflict, thus the formula is unsatisfiable. If the separator S is based on a counterexample then any assignment that satisfies S and Ψ will falsify at least one clause in the matrix φ. If S is based on an inconsistency then we know that S is sufficient to enforce the inconsistency, i.e., there is at least one existential variable e that has both an active rule for e and $\neg e$ under S. This means we can look for a subset $S' \subseteq S$ s.t. $\varphi \wedge \Psi \wedge S'$ is unsatisfiable. To compute S' we make use of assumption-based SAT solving [ES03]. More specifically, we check satisfiability of $\varphi \wedge \Psi$ under the assumption S and compute S' as the set of failed assumptions. The used reduction of S is related to the reduction of assignments presented in [RS04].

We have to distinguish between two cases: First, if S' contains ℓ, we can introduce a forcing clause as discussed before. Otherwise, we know that there is no reason to introduce a forcing clause for $\mathit{var}(\ell)$, as S' suffices to obtain a conflict. If S' contains another forced literal ℓ', we add a forcing clause for $\mathit{var}(\ell')$. If, on the other hand, S' contains no such literal, we repair the conflict by using arbiter clauses. We will discuss arbiter clauses in the next subsection.

4.4.2 Arbiter Clauses

Let us now assume that there is no forced source.[11] This is possible if there is no existential source or there are at least two existential sources with incomparable dependencies—this means that the assignment for one variable may not depend on the assignment for the other one. In the first case, we can conclude that the given DQBF must be unsatisfiable—no matter how we refine Ψ, we cannot repair the conflict. Thus, we add the empty clause to Δ.

For the remaining part of this section, we assume the other case. In this case we do not a priori know how to assign the existential sources. We only know that the variables need to be assigned differently. To handle this case, we can reuse the idea of arbiter variables, respectively clauses, presented in Chapter 3. First, we obtain a total assignment σ of the universal variables from the satisfying assignment for $\mathit{validity}(\Psi, \varphi)$, respectively

[10] Here Ψ shall denote the encoding for Ψ used in $\mathit{validity}(\Psi, \varphi)$. In practice the representation of Ψ used here does not need to contain the definitions, as the definitions follow propositionally from the matrix. Additionally, the representation does not need to contain arbiter clauses, as we do not require any assignment for the arbiter variables.

[11] Reduced separators without a forced literal can be handled analogously.

Algorithm 6 Add arbiter clause.

```
1: procedure addArbiterClause($\Psi, \Delta, X, \sigma$)
2:     $\delta \leftarrow \emptyset$
3:     for $\ell \in X$ do
4:         $e \leftarrow var(\ell)$
5:         if Arbiter clauses for $e^{\sigma|_{D(e)}}$ not yet in $\Psi$ then
6:             insert($\Psi, e^{\sigma|_{D(e)}} \vee \neg\sigma|_{D(e)} \vee \neg e$)
7:             insert($\Psi, \neg e^{\sigma|_{D(e)}} \vee \neg\sigma|_{D(e)} \vee e$)
8:         $\delta \leftarrow \delta \vee \neg\ell^{\sigma|_{D(e)}}$
9:     addClause($\Delta, \delta$)
```

consistency(Ψ, φ). Then, we introduce for each existential literal $\ell \in sources(\mathcal{C})$ with $var(\ell) = e$ the arbiter variable $e^{\sigma|_{D(e)}}$ if we have not yet introduced $e^{\sigma|_{D(e)}}$ before. In case we introduced a new arbiter variable $e^{\sigma|_{D(e)}}$, we also add the clauses $e^{\sigma|_{D(e)}} \vee \neg\sigma|_{D(e)} \vee \neg e$ and $\neg e^{\sigma|_{D(e)}} \vee \neg\sigma|_{D(e)} \vee e$ to Ψ.

To ensure that the same assignment of the existential sources is not used again, we introduce a clause δ. We will refer to δ as *blocking clause*. We define $\delta = \{\neg\ell^{\sigma|_{D(var(\ell))}} \mid \ell \in sources(\mathcal{C}), var(\ell) \in var_\exists(\Phi)\}$. To prevent the same conflict in subsequent iterations, we can then add δ to Δ. Furthermore, later we will show that at any point of the execution of the algorithm the arbiter assignment τ has to satisfy the clauses in Δ. Thus, we can be sure that arbiter assignments are not reused. The procedure for introducing arbiters, respectively blocking clauses is given in Algorithm 6.

To find a new arbiter assignment, τ we check if Δ is satisfiable (Algorithm 2 line 9). If Δ is satisfiable, then τ is assigned to some total satisfying assignment of Δ. As τ satisfies all blocking clauses, the candidate Ψ will not repeat conflicts under τ. If, on the other hand Δ is unsatisfiable, then any arbiter assignment falsifies at least one blocking clause. Thus, no matter how we refine the candidate, we will not obtain a model. So the algorithm reports unsatisfiability of the given DQBF.

4.4.3 The Necessity of Using Forcing Clauses

Even if we find a forced literal in the sources of the conflict graph, we could introduce arbiter variables for the existential sources instead. The resulting algorithm would still be a decision procedure for DQBF. While the usage of forcing clauses is not necessary to decide if a DQBF is satisfiable, forcing clauses still have a major impact on the performance of the algorithm in practice. We illustrate this with the following example.

Example 4.2. *Let $n \in \mathbb{N}^*$ and $\Phi = \forall u_1, \ldots, u_n \exists e(u_1, \ldots, u_n)\, \mathcal{Q}'.\, (u_1 \vee e) \wedge (u_1 \vee \neg e) \wedge \varphi'$. We are not interested in the remaining part of the prefix $\mathcal{Q}'$ nor are we interested in the remaining part of the matrix φ'. We just want to assume that e is not defined by its extended dependencies and no dependency of e can be removed by the $\mathcal{D}^{RRS}$ dependency* *scheme. First, we discuss a (possible) trace of the algorithm with forcing clauses then one without. Initially, the candidate for e is the function that maps each input to true. This means, the first counterexample can be given by the term $\neg u_1 \wedge \ldots \wedge \neg u_n \wedge e$. Under this counterexample the clause $u_1 \vee \neg e$ is falsified. We can see that the conflict graph just consists of two vertices for $\neg u_1$ and e. These vertices are both source and sink vertices, thus the separator is given by $\{\neg u_1, e\}$. We can now derive the forcing clause $\neg u_1 \vee \neg e$—we assume that the separator cannot be further reduced by using failed assumptions. In the second iteration, we can then get the counterexample given by the term $\neg u_1 \wedge \ldots \wedge \neg u_n \wedge \neg e$. Here, the forcing clause introduced before ensures that no matter how the variables $u_2, \ldots, u_n$ are assigned, the variable e needs to be assigned to false if u_1 is assigned to false. In this counterexample the clause $(u_1 \vee e)$ is falsified. Due to the forcing clause derived in the last iteration the conflict graph contains an edge from $\neg u_1$ to $\neg e$. This means that there is no existential source literal in the conflict graph. As discussed before, the algorithm reports unsatisfiability in this case. If we would not use forcing clauses, we would introduce an arbiter variable $e^{\neg u_1, \ldots, \neg u_n}$ and arbiter clauses $e^{\neg u_1, \ldots, \neg u_n} \vee u_1 \vee \ldots \vee u_n \vee \neg e$ and $\neg e^{\neg u_1, \ldots, \neg u_n} \vee u_1 \vee \ldots \vee u_n \vee e$ for the counterexample $\neg u_1 \wedge \ldots \wedge \neg u_n \wedge e$. To guarantee a different counterexample in the next iteration, we would add the blocking clause $\neg e^{\neg u_1, \ldots, \neg u_n}$. Unlike to the forcing clauses this arbiter variable does not generalize the universal part of the counterexample. This means for any assignment for $u_2, \ldots, u_n$ we can get a counterexample that assigns both e and u_1 to false. Consequently, we have to consider 2^{n-1} assignments just to fix the candidate's response for the clause $u_1 \vee e$.*

4.4.4 Default Function Refinement

In case the algorithm finds a forcing clause for an existential variable e the default function for e is updated by the UPDATEDEFAULTS method. For this purpose, the conflicting assignment is first restricted to the dependencies of e, we denote the resulting assignment by σ. We then add σ as a sample to the decision tree. If the forcing clause contains e positively we label the sample with true and otherwise with false. In case the Hoeffding tree updates the classification of a leaf, respectively it splits a leaf, we need to update the representation of the default function in subsequent iterations.

We only introduce new samples for the decision tree in case there is a forcing, because only in this case we know how to label a sample. Remember that if we introduce arbiter clauses, we do not necessarily how to assign each individual existential variable under an assignment to the dependencies.

Moreover, we want to point out that the labeled samples we add to the decision tree cannot be considered as independent (in the statistical sense)—inserting a sample into the tree might modify the default function, and thus the sample might have an influence on the conflicts we find in the future. This means that the Hoeffding tree does not converge to a decision tree learned by batch learn. For us this does not matter, as the purpose of the decision trees is anyway just to get good guesses for the existential assignment in case they are not yet fixed.

We illustrate the influence of using default functions with the following example.

Example 4.3. *Let n be some positive integer then consider the DQBF*

$$\forall u_1 \ldots \forall u_n \, \exists e_1(u_1, \ldots, u_n) \exists e_2(u_2, \ldots, u_n). \\ (e_1 \Leftrightarrow \mathrm{XOR}(u_1, \ldots, u_n)) \wedge (u_1 \vee u_2 \vee e_1 \vee \neg e_2) \wedge (\neg u_1 \vee \neg u_2 \vee e_1 \vee e_2).$$

While e_1 has a definition e_2 does not. In this example, we will get a large number of counterexamples: A lower bound for the number of counterexamples is given by the number of assignments that assign an even number of universal variables to true, u_1 and u_2 to true and e_2 to false. Each counterexample gives a forcing clause. Moreover, we can see that in each counterexample where we have $\neg e_2$ we also have u_2, respectively that in counterexamples with e_2 we have $\neg u_2$. Thus, after a sufficiently large number of counterexamples the decision tree learning will introduce a split for u_2. We can thus obtain the default function $f(u_2, \ldots, u_n) = u_2$. By using this function we can then immediately show that the formula evaluates to true. Thus, learned default functions can reduce the number of conflicts.

4.4.5 Proofs

In this section, we will prove that Algorithm 2 is a decision procedure for DQBF. For this purpose, we will first show that each blocking clause corresponds to a clause that is derivable from the given DQBF by ∀Exp+Res. We will see that this implies the existence of a ∀Exp+Res proof in case Algorithm 2 reports unsatisfiability of a DQBF. As ∀Exp+Res is sound this allows to conclude that whenever the algorithm reports unsatisfiability of a DQBF the DQBF is indeed unsatisfiable. Next, we will show termination of the algorithm by making use of the fact that there can only be finitely many arbiter assignments and counterexamples. Finally, we will show that if the algorithm reports satisfiability then the candidate describes a model, which means that the DQBF is satisfiable.

Lemma 4.9. *Let φ be a propositional formula and $t = \ell_1 \wedge \ldots \wedge \ell_n$ a term. If $\varphi \wedge t$ is unsatisfiable, then there is a subterm $t' \subseteq t$ s.t. $\neg t'$ can be derived from φ by resolution.*

Proof. If φ is unsatisfiable, then the empty clause can be derived from φ. So assume that φ is satisfiable. As $\varphi \wedge t$ is unsatisfiable there must be a minimal subset $t' \subseteq t$ s.t., for each $s \subsetneq t'$ the formula $\varphi \wedge s$ is satisfiable. Now let φ' be the formula that is obtained from φ by removing all clauses subsumed by literals in t'. We can see that $\varphi' \wedge t'$ is unsatisfiable. As propositional resolution is refutationally complete there must be a refutation $P = C_1, \ldots, C_n, \bot$ of $\varphi' \wedge t'$. W.l.o.g. we can assume the resolutions on literals in t' are considered last. As t' was assumed to be minimal, P must contain the clause that is obtained by negating the literals in t'. □

Lemma 4.10. *Let φ be a propositional formula and C a clause that can be derived from φ by resolution. Moreover, let σ be an assignment that does not satisfy C. Then we can derive a subclause of $C[\sigma]$ from $\varphi[\sigma]$.*

Proof. As C is derivable from φ, we can see that $\varphi \wedge \neg C$ is unsatisfiable—resolving C with the literals in $\neg C$ yields the empty clause. Consequently, for any assignment σ that does not satisfy C the CNF $(\varphi \wedge \neg C)[\sigma] = \varphi[\sigma] \wedge \neg C[\sigma]$ is unsatisfiable. By Lemma 4.9 this means that a subclause of $C[\sigma]$ is derivable from $\varphi[\sigma]$. □

Lemma 4.11. *Let φ and ψ be propositional formulas such that for each clause C in ψ a subclause C' can be derived from φ by resolution. Then φ is unsatisfiable if and only if $\varphi \wedge \psi$ is unsatisfiable.*

Proof. If φ is unsatisfiable then obviously also $\varphi \wedge \psi$ is unsatisfiable. So assume that $\varphi \wedge \psi$ is unsatisfiable. Let ψ' be the formula consisting of the derivable subclauses of ψ. We can see that $\varphi \wedge \psi'$ must be unsatisfiable. Thus, the empty clause can be derived from this formula be resolution. Since the clauses in ψ' can be derived from φ, this refutation is also a refutation for φ. Therefore, φ is unsatisfiable. □

Lemma 4.12. *Let φ be a propositional formula, $v \in var(\varphi)$ a variable, ψ_v a definition for v in φ by a set of variables $X \subseteq var(\varphi)$ and σ an assignment of X. Moreover, let $v_\psi(\sigma)$ denote the literal v if $\psi_v[\sigma]$ is true and $\neg v$ otherwise. Then for any assignment σ of X, we can derive a subclause of $\neg\sigma \vee v_\psi(\sigma)$ by resolution.*

Proof. Let σ be an arbitrary but fixed assignment for X. If $\varphi \wedge \sigma$ is unsatisfiable, then by Lemma 4.9 the statement of the lemma holds. So assume that $\varphi \wedge \sigma$ is satisfiable. As ψ_v is a definition for v, we know that every satisfying assignment has to satisfy the literal $v_\psi(\sigma)$. Thus, $\varphi \wedge \sigma \wedge \neg v_\psi(\sigma)$ is unsatisfiable. By Lemma 4.9 this means that we can derive a subclause of $\neg\sigma \vee v_\psi(\sigma)$. □

As discussed earlier, each vertex in a conflict graph corresponds to a literal. Thus, for sake of simplicity, we identify sets of vertices with terms consisting of the corresponding literals.

Lemma 4.13. *Suppose Algorithm 2 is applied to a DQBF $\Phi = \mathcal{Q}.\varphi$. We denote the conflict graph in the i^{th} iteration by $\mathcal{C}_i$, and the set of forcing clauses by ψ_F^i. Moreover, let S denote a separator of $\mathcal{C}_i$. Then the formula $\varphi \wedge \psi_F^i \wedge S$ is unsatisfiable.*

Proof. We proceed by Noetherian induction (cf. [Win96]). In preparation of this we first define a relation $\rightarrow$ on separators of $\mathcal{C}_i$ as follows. Let S_1 and S_2 be two separators of $\mathcal{C}_i$ then we have $S_1 \rightarrow S_2$ if there is a vertex $x \in S_1$ s.t. $S_2 = (S_1 \setminus \{x\}) \cup \{y \mid (y, x) \in E(\mathcal{C}_i)\}$. That is S_2 can be obtained from S_1 by replacing x by its predecessors in $\mathcal{C}_i$. As the sets of variables occurring in definitions, respectively forcing clauses, of a variable e may only contain variables from the extended dependencies of e, we can conclude that there is no infinite chain for the relation $\rightarrow$. Consequently, we can apply Noetherian induction with respect to $\rightarrow$. Now let S be an arbitrary but fixed separator of $\mathcal{C}_i$. If there is no separator T with $T \rightarrow S$ then S consists of the sink vertices of $\mathcal{C}_i$. We now have to consider two cases: Either a counterexample or an inconsistency was found in the i^{th} iteration. In the first case, we know that there is a clause C in the matrix φ

s.t. $S = \{\neg x \mid x \in C\}$. In the second case there is a variable e such that the set of sink vertices is given by $\{e, \neg e\}$. Unsatisfiability of the formula follows in both cases. Now assume there is a separator T with $T \to S$. This means there is a literal ℓ in T that is replaced by its predecessors denoted by the set P for obtaining S. Now there are two cases. First assume there is a forcing clause with $\{\neg p \mid p \in P\} \cup \{\ell\}$. From the induction hypothesis, we know that $\varphi \wedge \psi_F^i \wedge T$ is unsatisfiable. We can see that we can derive the literal ℓ from S and the forcing clause $\{\neg p \mid p \in P\} \cup \{\ell\}$ by resolution. So if $\varphi \wedge \psi_F^i \wedge S$ would be satisfiable, then also $\varphi \wedge \psi_F^i \wedge T$ would be satisfiable. This means $\varphi \wedge \psi_F^i \wedge S$ is unsatisfiable. Second, assume there is a definition ψ_e that yields the edges. We can conclude that $e_\psi(P) = \ell$. By Lemma 4.12, we can derive a subclause of $\neg P \vee \ell$. If the subclause contains ℓ, we can conclude as above that $\varphi \wedge \psi_F^i \wedge S$ is unsatisfiable. Otherwise, unsatisfiability directly follows from applying resolution on the term S and the derived subclause. □

Definition 4.2. Let $\Phi = \mathcal{Q}.\varphi$ be a DQBF and C a clause with $var(C) \subseteq var(\Phi)$. Now let σ be a total assignment to the universal variables in Φ that falsifies each universal literal in C. Then we define the clause C^σ as $\{\ell^{\sigma|_{D(var(\ell))}} \mid \ell \in C, var(\ell) \in var_\exists(\Phi)\}$. Similarly, for a CNF λ with $var(\lambda) \subseteq var(\Phi)$, we define λ^σ as the CNF $\{C^\sigma \mid C \in \lambda, C[\sigma] \neq 1\}$.

Theorem 4.2. *Suppose Algorithm 2 is applied to a DQBF $\Phi = \mathcal{Q}.\varphi$. We denote the conflict graph in the i^{th} iteration by $\mathcal{C}_i$. Moreover, let S denote a minimal separator of $\mathcal{C}_i$ and $S' \subseteq S$ s.t. $\varphi \wedge \psi_F^i \wedge S'$ is unsatisfiable. Then for every universal assignment σ that satisfies each universal literal in S', we can derive a subclause of $(\neg S')^\sigma$ by ∀Exp+Res from Φ.*

Proof. First, due to Lemma 4.13 the set S' is well-defined for each iteration i. We now show by induction that for each universal assignment σ that satisfies each universal literal in S', we can derive a subclause of $(\neg S')[\sigma]$ by resolution from $\varphi[\sigma]$. Initially the set of forcing clauses ψ_F^i is empty. Thus, $\varphi \wedge S'$ is unsatisfiable. By Lemma 4.9, we can derive a subclause of $\neg S'$ from φ by resolution, and so by Lemma 4.10 we can derive a subclause of $\neg S'[\sigma]$. Now suppose $i = n$ for some $n \in \mathbb{N}^*$ and the proposition holds for each $j < n$. First, we see that by the induction hypothesis we can conclude that for each forcing clause $C \in \psi_F^i$ that is not satisfied by σ, a subclause of $C[\sigma]$ can be derived by resolution from $\varphi[\sigma]$, since forcing clauses correspond to separators from previous iterations. We denote the set consisting of these forcing clauses by $\hat{\psi}_F$. As $\varphi \wedge \hat{\psi}_F \wedge S'$ is unsatisfiable also $(\varphi \wedge \hat{\psi}_F \wedge S')[\sigma]$ is unsatisfiable. By Lemma 4.11, we can conclude that $(\varphi \wedge S')[\sigma]$ is unsatisfiable. But by Lemma 4.9 this means that we can derive a subclause of $\neg S'[\sigma]$ from $\varphi[\sigma]$. This shows that in every iteration we can derive a subclause of $(\neg S')[\sigma]$ from $\varphi[\sigma]$. As a consequence of this we can derive $(\neg S')^\sigma$ from Φ by ∀Exp+Res. □

Corollary 4.2. *Let $a_1^{\sigma_1} \vee \ldots \vee a_n^{\sigma_n}$ be a clause that is added to the set of blocking clauses in the i^{th} iteration of the algorithm for the DQBF Φ, where for each $j \in [n]$ the literal $a_j^{\sigma_j}$ corresponds to the existential variable e_j. Let ℓ_j be e_j if $a_j^{\sigma_j}$ occurs positively in the clause and $\neg e_j$ otherwise. Then we can derive a subclause of $\ell_1^{\sigma_1} \vee \ldots \vee \ell_n^{\sigma_n}$ by ∀Exp+Res from Φ.*

Proof. If we add an arbiter clause for existential literals $\ell_1, \ldots, \ell_n$ at the i^{th} iteration, then there is a separator of $\mathcal{C}_i$ that contains $\{\neg\ell_1, \ldots, \neg\ell_n\}$ as a subset. Let σ be a total universal assignment s.t. for each $j \in [n]$ we have $\sigma_j = \sigma|_{D(var(\ell_j))}$. We can see that the construction of Algorithm 2 guarantees that such an assignment exists. There must be a universal assignment σ' contained in σ s.t. the separator contains σ'. Next, by Theorem 4.2 we can conclude that a subclause of $\ell_1^{\sigma_1} \vee \ldots \vee \ell_n^{\sigma_n}$ is derivable by ∀Exp+Res from Φ. □

Corollary 4.3. *If Algorithm 2 reports unsatisfiability of a DQBF Φ, then Φ is unsatisfiable.*

Proof. If Algorithm 2 reports that the given DQBF is unsatisfiable, then the set of blocking clauses must be unsatisfiable at some iteration i. By Corollary 4.2, we know that for each blocking clause a corresponding clause can be derived by ∀Exp+Res. As a consequence there is a ∀Exp+Res refutation for Φ. As ∀Exp+Res is sound, we can conclude that Φ is unsatisfiable. □

Theorem 4.3 (termination)**.** *Algorithm 2 terminates.*

Proof. We can see that all subprocedures terminate, thus it remains to show that the main loop terminates. Moreover, we can see that there can only be finitely many different arbiter variables and thus there is only a finite number of arbiter assignment. If we add a blocking clause to Δ, we can be sure that arbiter assignments in subsequent iterations differ from the current one. Thus, it is not possible that infinitely many blocking clauses are added to Δ. The only case, which remains to consider is that for some arbiter assignment τ infinitely many forcing clauses are introduced. First, we can see that if we introduce a forcing clause for a counterexample, then we cannot get the same counterexample again—the sources of the conflict graph must be assigned differently. Similarly, we cannot get the same inconsistency again. As there are only finitely many counterexamples, respectively inconsistencies, only finitely many forcing clauses can be introduced for one arbiter assignment. All in all, this means that the main loop terminates and thus the algorithm itself terminates. □

Theorem 4.4. *If Algorithm 2 reports satisfiability of a DQBF Φ, then Φ is satisfiable.*

Proof. If Algorithm 2 concludes that Φ is satisfiable, then at some iteration i the formula $\neg\varphi \wedge \psi_D \wedge \psi_F^i \wedge \psi_A^i \wedge \psi_{Def}^i$ is unsatisfiable. We also know that $\psi_D \wedge \psi_F^i \wedge \psi_A^i \wedge \psi_{Def}^i$ is consistent, i.e., for each universal assignment σ, there is an existential assignment γ_σ s.t. $\sigma \cup \gamma_\sigma$ satisfies the $\psi_D \wedge \psi_F^i \wedge \psi_A^i \wedge \psi_{Def}^i$. We can see that by the construction of Algorithm 2 there is a unique assignment γ_σ for each σ. Next, let e be an existential variable and σ an arbitrary total universal assignment. We can define a Skolem function $f_e : D(e) \to \mathbb{B}$ as $f_e(\sigma) = \gamma_\sigma(e)$. Now suppose the family f of Skolem functions $(f_e)_{e \in var_\exists(\Phi)}$ is not a model of Φ. Consequently, there is a universal assignment σ s.t. $\sigma \cup f(\sigma)$ falsifies the matrix. But this means there is an assignment of $var(\Phi)$ that satisfies $\neg\varphi \wedge \psi_D \wedge \psi_F^i \wedge \psi_A^i \wedge \psi_{Def}^i$, a contradiction. Thus, f is a model and therefore Φ is satisfiable. □

Corollary 4.4. *Algorithm 2 is a decision procedure for DQBF.*

Proof. Follows by Corollary 4.3, Theorem 4.3 and Theorem 4.4. □

This shows that Algorithm 2 can indeed be used for determining if a DQBF is satisfiable or not.

CHAPTER 5

Experiments – Part I

In this chapter, we will experimentally compare an implementation of the CEGIS algorithm, presented in Chapter 4 (Algorithm 2), with state-of-the-art DQBF solvers.

5.1 Implementation

We implemented Algorithm 2 as described in Chapter 4 in a program called PEDANT. PEDANT is implemented in C++ and it is publicly available.[1] PEDANT internally uses the following tools and libraries.

- For the computation of definitions we use a subroutine from the QBF preprocessor UNIQUE [Sli20] that in turn relies on an interpolating version of MINISAT [ES03] bundled with the EXTAVY model checker [GV14; VGM15].
- All remaining SAT checks from Chapter 4 are realized by applying the SAT solver CADICAL [Bie+20].[2]
- For computing separators of conflict graphs, we use the BOOST library.[3]
- To learn the Hoeffding decision trees, which represent the default functions, we use MLPACK [Cur+23].[4]
- For generating AIG certificates for satisfiable DQBF, we use the AIGER library [Bie07; BHW11].[5]

[1]Available at: `https://github.com/fslivovsky/pedant-solver`

[2]We also tested CRYPTOMINISAT [SNC09] and GLUCOSE [AS09]. But overall CADICAL performed (slightly) better than the other solvers. As we make use of incremental SAT solving, we did not consider the solver KISSAT [BF22].

[3]Available at: `http://www.boost.org/`

[4]Available at: `https://www.mlpack.org/`

[5]Available at: `https://github.com/arminbiere/aiger`

5.2 Experimental Evaluation

In this section, we will compare Pedant with state-of-the-art DQBF solvers on standard benchmark sets. More specifically, we chose the solvers DQBDD [Síč20], HQS [Git+15], iProver [Kor08], iDQ [Frö+14] and dCAQE [TR19]. We did not consider the Boolean function synthesizer Manthan [GRM23], as it is not a decision procedure for DQBF. In addition to these solvers, we also considered the DQBF preprocessor HQSPre [WSB19]. We used the following configurations of the aforementioned solvers.

Pedant Pedant with enabled model logging (`--aig`).

PedantHQ Pedant together with HQSPre. For preprocessing we first applied HQSPre[6] to a given formula. If HQSPre was able to solve the formula, we returned its result. Otherwise, we applied Pedant to the preprocessed formula. Here we used Pedant with its default configuration.

DQBDD DQBDD with its default configuration. As DQBDD applies HQSPre internally, we did not explicitly apply HQSPre.

HQS HQS with its default configuration. HQS applies HQSPre internally. Thus, we did not explicitly apply HQSPre.

iProver iProver with enabled QBF mode (`--qbf_mode true`). When used in QBF mode iProver also accepts DQBF. iProver was used together with HQSPre with default options for preprocessing.

iDQ iDQ with its default configuration. iDQ was used together with HQSPre with default options for preprocessing.

dCAQE dCAQE with its default configuration. dCAQE was used together with HQSPre with default options for preprocessing.

We also applied the solvers iProver, iDQ and dCAQE without preprocessing, but since all the three mentioned solvers benefit from using HQSPre, in the following we only consider the configurations with preprocessing.

To evaluate the solvers we considered two benchmark sets. First, we used the benchmarks from the QBF Gallery 2023 [PSH23]—we denote these benchmark as *QBF23* benchmarks. Second, we used a benchmark set which has been considered in previous work on HQS [GSW19].[7] These benchmarks consist of encodings of partial equivalence checking [SB01; Frö+14; Git+13b; FT14] and controller synthesis [BKS14], as well as succinct DQBF representations of propositional satisfiability [BCJ14a]. We denote this set of instances as the *Compound* benchmarks.

[6]With the options `--resolution 1 --univ_exp 0 --substitute 0 --preserve_gate 1`

[7]Available at: `http://abs.informatik.uni-freiburg.de/src/projectfiles/21/DQBFBenchmarks.zip`

All experiments were conducted on a cluster with AMD EPYC 7402 processors at 2.8 GHz running 64-bit Linux. Moreover, the presented results are based on single runs with a time and memory limit of 1800 seconds and 8 GB, respectively, which were enforced using RunSolver [Rou11]. For each configuration that applies HQSPre we used a timeout of 180 seconds for HQSPre. The time used for preprocessing is included in the total running time.

We first compare the number of solved instances and the PAR2 scores[8] of the aforementioned configurations for the QBF23 benchmarks. The results are summarized in Table 5.1. We compare our solver Pedant with the configurations PedantHQ, DQBDD and HQS in Table 5.1a and with the remaining configurations in Table 5.1b.[9]

The results show that overall Pedant clearly outperformed the other considered configurations, both in terms of solved instances and in PAR2 scores. Only for the "Balabanov" family iDQ could solve one more formula, respectively for the "Tentrup" family Pedant used together with HQSPre could solve two more formulas. The results in particular also show that unlike to iProver, iDQ and dCAQE, our solver Pedant did not benefit from preprocessing with HQSPre.

Next, we compare the performance of the solvers for the Compound instances. The results are given in Table 5.2.[10] While, Pedant could still solve more instances than any other configuration, the difference to PedantHQ and DQBDD is now much smaller. In particular, it is no longer clear if enabling preprocessing for Pedant is disadvantageous—by applying HQSPre the instances from both the "Biere" and the "Finkbeiner" family could be solved faster on average, while in total only five instances less could be solved.

In addition to the tables presented above, we also visualized the performance of the configurations for both benchmark sets with cactus plots in Figure 5.1. We can see that for the QBF23 instances Pedant could clearly outperform all other considered solvers in terms of solved instances. For the Compound instances, both Pedant and DQBDD show a comparable performance, whereas Pedant could clearly solve more instances than any of the remaining configurations.

To compare runtimes of the solvers, we give scatter plots that compare Pedant and DQBDD (Figure 5.2), Pedant and HQS (Figure 5.3), and Pedant and PedantHQ (Figure 5.4).

For the Pedant configuration, we computed certificates for each satisfiable instance. Certificates are given by And-Inverter Graphs (AIGs) representing the final candidates

[8]The Penalized Average Runtime (PAR) is the average runtime, with the time for each unsolved instance calculated as a constant multiple of the timeout.

[9]For two instances dCAQE reported their satisfiability, whereas both Pedant configurations and HQS reported their unsatisfiability. While neither Pedant nor HQS provide certificates for unsatisfiable formulas, we think that dCAQE reported incorrect results—as we have independent solutions that claim unsatisfiability.

[10]As for the QBF23 instances, for five instances dCAQE reported their satisfiability, while at least one other configuration reported their unsatisfiability. For a similar reason as before, we think that dCAQE reported incorrect results and not the other solvers.

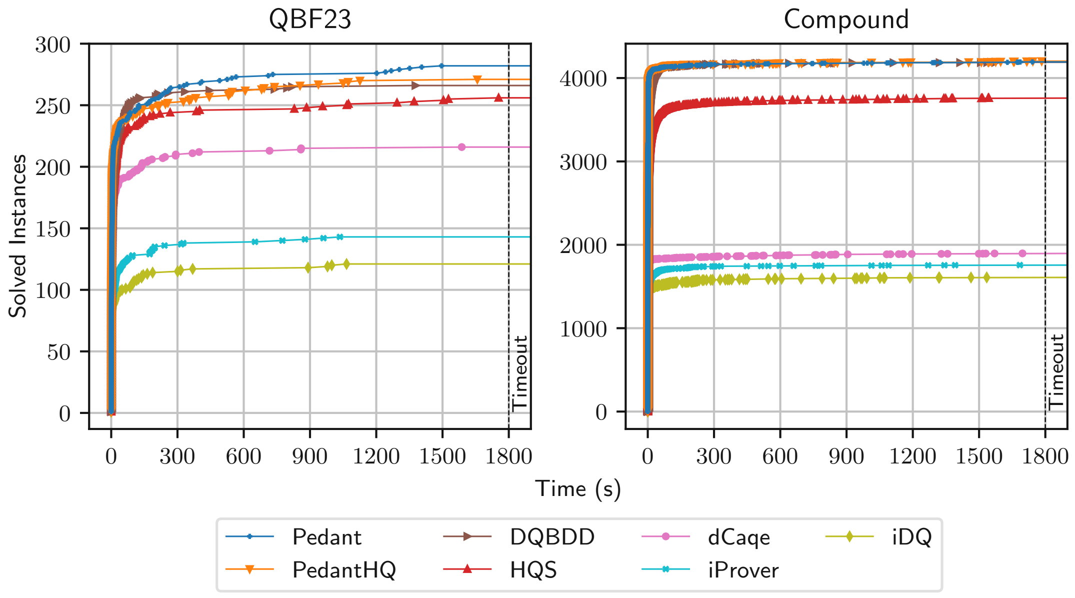


Figure 5.1: Cactus plots for the QBF23 and the Compound instances.

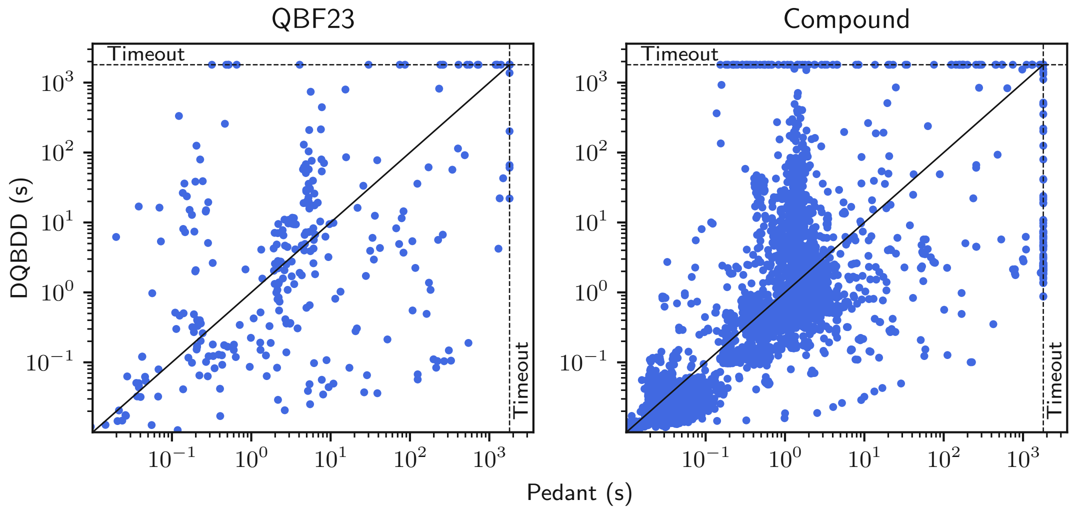


Figure 5.2: Scatter plots for the QBF23 and the Compound instances, comparing Pedant and DQBDD.

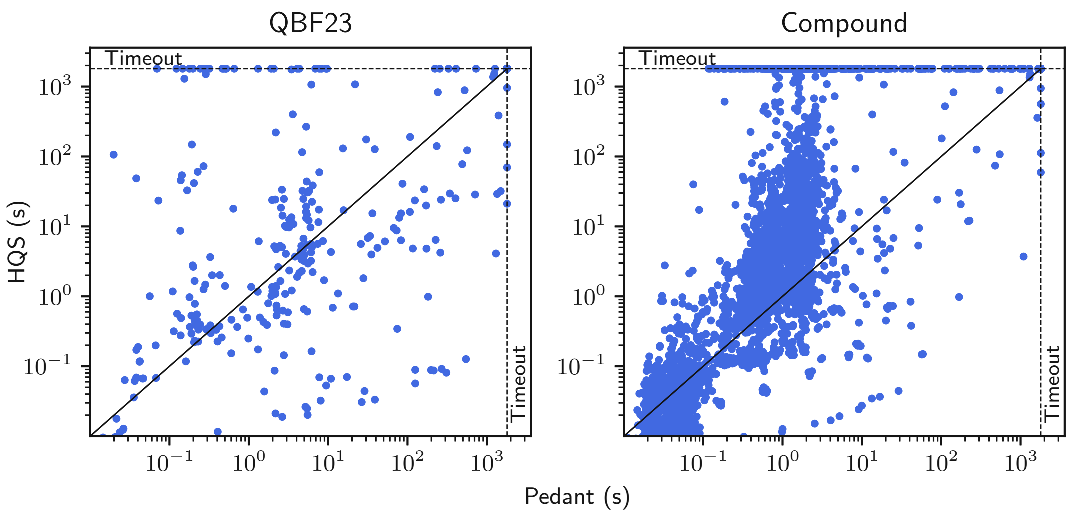


Figure 5.3: Scatter plots for the QBF23 and the Compound instances, comparing PEDANT and HQS.

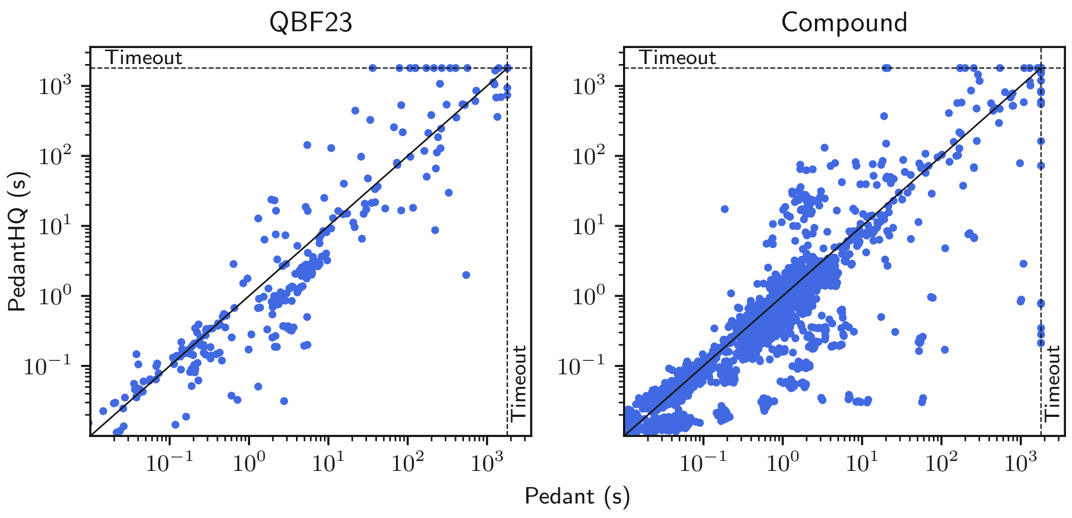


Figure 5.4: Scatter plots for the QBF23 and the Compound instances, comparing PEDANT and PEDANTHQ.

	Pedant		PedantHQ		DQBDD		HQS	
Family (Total)	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2
Balabanov (34)	20	$1.7 \cdot 10^3$	19	$1.8 \cdot 10^3$	14	$2.2 \cdot 10^3$	19	$1.8 \cdot 10^3$
Bloem (90)	**41**	$\mathbf{2.0 \cdot 10^3}$	**41**	$2.0 \cdot 10^3$	34	$2.3 \cdot 10^3$	33	$2.3 \cdot 10^3$
Kullmann (50)	**50**	$7.5 \cdot 10^1$	39	$8.4 \cdot 10^2$	**50**	$2.2 \cdot 10^1$	41	$7.2 \cdot 10^2$
Scholl (90)	**85**	$\mathbf{2.1 \cdot 10^2}$	84	$2.4 \cdot 10^2$	83	$2.8 \cdot 10^2$	80	$4.1 \cdot 10^2$
Tentrup (90)	86	$2.2 \cdot 10^2$	**88**	$\mathbf{1.3 \cdot 10^2}$	85	$2.4 \cdot 10^2$	83	$3.3 \cdot 10^2$
Total (354)	**282**	$\mathbf{7.9 \cdot 10^2}$	271	$8.9 \cdot 10^2$	266	$9.2 \cdot 10^2$	256	$1.0 \cdot 10^3$

(a) Comparison of Pedant and PedantHQ with the other participants of the QBF Gallery 2023.

	Pedant		iProver		iDQ		dCaqe	
Family (Total)	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2
Balabanov (34)	20	$1.7 \cdot 10^3$	20	$1.6 \cdot 10^3$	**21**	$\mathbf{1.5 \cdot 10^3}$	20	$1.6 \cdot 10^3$
Bloem (90)	**41**	$\mathbf{2.0 \cdot 10^3}$	22	$2.7 \cdot 10^3$	14	$3.1 \cdot 10^3$	31	$2.4 \cdot 10^3$
Kullmann (50)	**50**	$7.5 \cdot 10^1$	**50**	$\mathbf{4.6 \cdot 10^0}$	**50**	$6.8 \cdot 10^0$	35	$1.1 \cdot 10^3$
Scholl (90)	**85**	$\mathbf{2.1 \cdot 10^2}$	30	$2.4 \cdot 10^3$	19	$2.9 \cdot 10^3$	52	$1.6 \cdot 10^3$
Tentrup (90)	86	$2.2 \cdot 10^2$	21	$2.8 \cdot 10^3$	17	$2.9 \cdot 10^3$	78	$5.0 \cdot 10^2$
Total (354)	**282**	$\mathbf{7.9 \cdot 10^2}$	143	$2.2 \cdot 10^3$	121	$2.4 \cdot 10^3$	216	$1.4 \cdot 10^3$

(b) Comparison of the default Pedant configuration with additional solvers.

Table 5.1: Number of solved instances and PAR2 scores per solver configuration and benchmark family for the QBF23 instances. The configurations that solved most instances from a family, or which have the lowest PAR2 score are highlighted in bold.

under the final arbiter assignments. These AIGs were then logged in the binary AIGER format [Bie07; BHW11]. The time needed to generate these AIGs is included in the total runtime.

The other evaluated solvers do currently not support the generation of Skolem functions. While a previous version of HQS was able to compute models [Wim+16a], this is no longer possible in the current version.

To validate models, we implemented a simple workflow in Python using the PySAT library [IMM18].[11] We first perform a syntactic check to make sure that the encoded Skolem functions only depend on variables in the corresponding dependency set. Then it checks if the matrix can be falsified under the encoded model by using a SAT solver. We applied this tool to check each certificate. The time needed for these checks is not included in the total runtime of Pedant.

In addition to the experiments presented above, Pedant also showed good performance in recent evaluations of DQBF solvers. Pedant received a gold medal at the FLoC Olympic Games 2022 for achieving the first place in the DQBF track of the QBF Evaluation

[11]The tool is available as part of Pedant at `https://github.com/fslivovsky/pedant-solver`

	Pedant		PedantHQ		DQBDD		HQS	
Family (Total)	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2
Balabanov (34)	19	$1.8 \cdot 10^3$	18	$1.9 \cdot 10^3$	14	$2.2 \cdot 10^3$	19	$1.8 \cdot 10^3$
Biere (1200)	**1200**	$7.8 \cdot 10^{-2}$	**1200**	$5.6 \cdot 10^{-2}$	**1200**	$\mathbf{3.6 \cdot 10^{-2}}$	**1200**	$4.6 \cdot 10^{-2}$
Bloem (461)	**121**	$\mathbf{2.7 \cdot 10^3}$	**121**	$2.7 \cdot 10^3$	88	$2.9 \cdot 10^3$	81	$3.0 \cdot 10^3$
Finkbeiner (2000)	**2000**	$1.5 \cdot 10^0$	**2000**	$\mathbf{1.2 \cdot 10^0}$	**2000**	$9.8 \cdot 10^0$	1801	$3.9 \cdot 10^2$
Scholl (1116)	867	$8.4 \cdot 10^2$	863	$8.5 \cdot 10^2$	**887**	$\mathbf{7.5 \cdot 10^2}$	659	$1.5 \cdot 10^3$
Total (4811)	**4207**	$\mathbf{4.6 \cdot 10^2}$	4202	$4.7 \cdot 10^2$	4189	$4.7 \cdot 10^2$	3760	$8.0 \cdot 10^2$

(a) Comparison of Pedant and PedantHQ with the other participants of the QBF Gallery 2023.

	Pedant		iProver		iDQ		dCaqe	
Family (Total)	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2	#Sol	PAR2
Balabanov (34)	19	$1.8 \cdot 10^3$	20	$1.6 \cdot 10^3$	**21**	$\mathbf{1.4 \cdot 10^3}$	20	$1.6 \cdot 10^3$
Biere (1200)	**1200**	$7.8 \cdot 10^{-2}$	1198	$7.7 \cdot 10^0$	1185	$6.0 \cdot 10^1$	**1200**	$1.3 \cdot 10^{-1}$
Bloem (461)	**121**	$\mathbf{2.7 \cdot 10^3}$	68	$3.1 \cdot 10^3$	50	$3.2 \cdot 10^3$	84	$3.0 \cdot 10^3$
Finkbeiner (2000)	**2000**	$1.5 \cdot 10^0$	52	$3.5 \cdot 10^3$	6	$3.6 \cdot 10^3$	34	$3.5 \cdot 10^3$
Scholl (1116)	867	$8.4 \cdot 10^2$	419	$2.3 \cdot 10^3$	347	$2.5 \cdot 10^3$	559	$1.8 \cdot 10^3$
Total (4811)	**4207**	$\mathbf{4.6 \cdot 10^2}$	1757	$2.3 \cdot 10^3$	1609	$2.4 \cdot 10^3$	1897	$2.2 \cdot 10^3$

(b) Comparison of the default Pedant configuration with additional solvers.

Table 5.2: Number of solved instances and PAR2 scores per solver configuration and benchmark family for the Compound instances. The configurations that solved most instances from a family, or which have the lowest PAR2 score are highlighted in bold.

2022 [PSS24]. Additionally, Pedant achieved the first place in the DQBF track of the QBF Gallery 2023 [PSH23].

CHAPTER 6

Conclusions – Part I

In this part, we introduced two decision procedures for DQBF. The first one is conceptually rather simple. Its main purpose is to demonstrate how Skolem functions can be determined by computing propositional definitions. Additionally, it illustrates the usage of arbiter variables for fixing the response of a Skolem function for a particular assignment of the corresponding dependency set. Next, we discussed the main contribution of this part, the CEGIS-based decision procedure for DQBF. This algorithm follows a dual strategy for deciding DQBF. On the one hand, to show unsatisfiability, it generates a set of clauses that corresponds to a $\forall$Exp+Res proof. On the other hand, to show satisfiability, it tries to construct a model for the given formula. For this purpose, the algorithm looks for counterexamples of the current candidate model and uses them to refine the model. We proved for both algorithms that they are indeed decision procedures for DQBF.

In the experimental evaluation of our solver Pedant, which implements the CEGIS-based algorithm, we showed its effectivity. Pedant was able to solve considerably more instances from the QBF23 benchmark set, and it could solve slightly more instances from the "Compound" benchmark set, than other state-of-the-art DQBF solvers.

A core feature of Pedant, respectively the underlying decision procedure, is that it is certifying by design for satisfiable DQBF. While our solver is not the first one that can compute models for satisfiable DQBF, computing models is still not a standard, as some solvers can only report yes/no answers. Still, an open problem for our decision procedure is to not only provide certificates for satisfiable DQBF but also for unsatisfiable DQBF.

Future Work

There are four directions into which we would like to extend our work.

Decision List Semantics for Candidates In Algorithm 2, Skolem functions are either determined by propositional definitions or by forcing clauses, arbiter clauses and default functions. We can think of these clauses as rules that, if their premise holds, fix the assignment of an existentially quantified variable. As these rules are unordered, they can be considered as a decision set (cf. [Ign+21]) for assigning an existential variable. This also means that multiple rules can be active simultaneously. Hence, opposing assignments can be enforced at the same time. For this reason, we had to introduce the consistency check. To avoid this check, a decision list (cf. [Riv87]) based representation of the rules could be used. In this representation, forcing clauses would be ordered chronologically—an earlier introduced forcing clause would have precedence over a later introduced one. Additionally, forcing clauses would have precedence over arbiter clauses. The arbiter clauses do not need to be ordered, as arbiter clauses alone cannot yield an inconsistency. Furthermore, default functions can be used in case neither an arbiter nor a forcing clause applies. In combination, this would ensure that in each iteration of the algorithm the candidate describes exactly one response to each universal assignment.

Circuit-based Inputs When encoding a problem in DQBF, usually several variables are known to be defined, e.g., Tseitin variables. Currently, we have to apply definition extraction in order to obtain definitions for these variables. It would be interesting to extend Algorithm 2 to the non-clausal DQCIR format [SS21b]. This would allow making use of definitions known a priori, without the need of computing them.

Partial Annotations for Arbiters Remember that in case no forcing clause can be introduced, we add a blocking clause. Such a clause is obtained by annotating the existential source literals from the conflict tree by total assignments of their dependencies. The sets of source literals in general do not contain all universal variables from the dependencies of the existential sources. Thus, the use of total assignments for the annotations does not result in the most general refinement of the candidate. This was illustrated in Example 4.2, where we obtained multiple similar counterexamples that only differed on universal variables which were not part of the source vertices of the conflict graph. If we could use partial annotations for the arbiters instead, we could add only those universal variables that are part of the source vertices. Consequently, the obtained blocking clauses would correspond to IR-calc. While this would solve the problem discussed in Example 4.2, this would require some hierarchy on the arbiter variables that ensures that arbiters with compatible annotations get the same assignments.

Learning Default Functions from Samples Recently, Manthan showed promising results for synthesizing Skolem functions for DQBF [GRM23]. Manthan samples satisfying assignments of the matrix, which are then used to learn decision trees for each existential variable. These decision trees give initial candidates for the Skolem functions, which are then refined. In our solver, we initialize the default functions as constant values. We think that this could be enhanced by applying the idea proposed by Manthan to learn initial default functions from samples of the matrix.

Part II

Local Improvement of Circuits

CHAPTER 7

Introduction – Part II

The steady increase in complexity of modern integrated circuits makes both their design and optimization challenging tasks. Therefore, circuit design would be unthinkable without some automated methods. This includes the automated improvement of circuits (logic optimization), and the automated creation of circuits from specifications (logic synthesis), which jointly yield substantial reductions in the number of gates and circuit depth [DeM94; BHS90].

Small circuits are not only of interest from a theoretical point of view [Fin+16; KKY09], but they can also be of relevance in practice. Recent years showed increasing prices of silicon wafers [Gai21], amplifying the importance of being able to find small circuit designs.

Applying exact methods for computing provably minimum size circuits is computationally intractable. Recently, it was shown that for a (multi-output) Boolean function given as a truth table, the task of finding a minimum size circuit consisting of *and*, *or* and *not* gates is NP-complete [ILO20]. This is reflected by the observation that in practice, we can generally not compute minimum size circuits with many more than 10 fanin-2 gates [KKY09; CMM23].

Still, exact methods can be applied to reduce the size, even of large circuits. To do this, one can partition a large circuit into subcircuits that are sufficiently small for applying exact methods. Then, exact methods can be used to compute replacement circuits. While generally this does not yield optimum size circuits, it still allows to reduce the size of circuits. These replacement circuits can be either obtained from a pre-computed database of optimal circuits [MCB06], or on the fly by exact synthesis [Rie+19].

For computing replacement circuits, one does not need to restrict the search, to circuits computing exactly the same function. It is possible that certain assignments of the inputs can never arise within the encompassing circuit. Similarly, the outputs of the subcircuit might be masked under some assignment to the inputs. In these cases, the particular behavior of the subcircuit does not matter, and the subcircuit can yield arbitrary values as outputs. Making use of these so called *don't cares* for computing new implementations for subcircuits can help to significantly reduce their size [SB90; Sav92; MB05].

Typically, improving circuits by rewriting subcircuits is limited to single-output subcircuits, or the computation of the replacement circuits does not take advantage of the full implementational flexibility. In [KPS22] multi-output subcircuits are replaced by locally optimal circuits. While, this approach handles multi-output subcircuits, it cannot make use of don't cares, as it is only considering the subcircuit, and it does not take the encompassing circuit into account. For single-output subcircuits, propositional satisfiability (SAT) solvers have been used to computed don't cares [MB05]. In subsequent resynthesis, equivalence of the new implementation and the original subcircuit is not required for don't care assignments. Don't cares for multi-output subcircuits have been considered in [Lee+18]. Here, unattainable assignments for the inputs of the subcircuits (controllability don't cares) are detected by simulating the transitive fanin cones of each input. But, this method cannot make use of masked outputs, thus it still does not make use of the full implementational flexibility.

In this part, we present a new method for heuristically minimizing the sizes of circuits. The method belongs to the *SAT-based Local Improvement Method* (SLIM) framework [LOS16]. To improve a circuit, one incrementally selects subcircuits of a given circuit. Then new size optimal replacements for these subcircuits are computed by means of either SAT or Quantified Boolean formula (QBF) solvers. The computed circuits can then be used to replace the original implementation. Two core features of our approach are that both single- and multi-output subcircuits can be considered, and that the full implementational flexibility for computing the replacement is exploited by making use of don't cares.

The QBF-based approach directly encodes the problem of finding a replacement circuit of a given size as a QBF. This encoding implicitly describes the don't cares of the subcircuit, which means that don't cares do not need to be computed explicitly. For obtaining a minimum size circuit, we first consider the original size of the subcircuit and then decrease it until the encoding gets unsatisfiable. As soon as the QBF solver reports unsatisfiability for a given size, we know that the previously considered size is the smallest possible one. The replacement circuit can then be obtained from a model for the previous satisfiable QBF call.

The SAT-based approach first computes a representation of the don't cares, and then a replacement circuit that makes use of these don't cares. In the first step, incremental SAT solving is applied to find a Boolean relation [BHS90] that describes the input-output behavior of the subcircuit, taking don't cares into account. To obtain a replacement circuit, a slightly modified SAT encoding for exact synthesis from the literature [Haa+20] is used.

In both encodings, one must pay attention to not introduce any cycles into the encompassing circuit. While the computed replacement circuits are guaranteed to be acyclic on their own, they can cause cycles upon replacing the original implementation. By adding suitable conditions to the encodings, one can guarantee that the encompassing circuit remains acyclic.

Both approaches get harder if larger encompassing circuits are considered. For this reason, the proposed minimization approach can be applied to windows [MB05] instead of entire circuits. This does not only allow considering larger encompassing circuits, but it also enables minimizing several parts of a given circuit simultaneously.

We implemented this circuit minimization procedure in a system called eSLIM. In an experimental evaluation on standard benchmarks, eSLIM showed a promising performance.

Organization of Part II First, we will cover notations and definitions used throughout this part in Chapter 8. In this chapter, we will also give a brief overview of related work. In Chapter 9 we will first recap a SAT encoding for exact synthesis from the literature. Then we will introduce a new QBF-based encoding for exact synthesis. Next, in Chapter 10, we will adapt the encodings for exact synthesis for computing minimum size replacement circuits for subcircuits of a given circuit. In Chapter 11, we experimentally evaluate our tool eSLIM, which implements these minimization procedures. We conclude this part of the thesis in Chapter 12.

This part is based on the following papers on circuit minimization. We first presented the QBF-based encoding in [RSS23]. The SAT-based circuit minimization using Boolean relations and the usage of windowing is presented in [RSS24].

CHAPTER 8

Background – Part II

In this chapter, we will introduce terminology and notations used throughout this part of the thesis. Moreover, we will give a brief overview of related work.

8.1 Basic Concepts

We will denote the set of positive natural numbers by $\mathbb{N}^*$, and for $n \in \mathbb{N}^*$ we will denote the set $\{1, \ldots, n\}$ by $[n]$.

8.1.1 Formulas

A *literal* is either a propositional variable or its negation. Given a literal ℓ, we denote the variable in ℓ by $var(\ell)$. A *clause* is a disjunction of literals and a *term* a conjunction of literals. A formula in *conjunctive normal form* (CNF) is a disjunction of clauses. We identify formulas in CNF by sets of clauses and both clauses and terms with sets of literals. Let C be a clause and φ a CNF. We define the set $var(C)$ of variables in C as $\{var(\ell) \mid \ell \in C\}$ and the set $var(\varphi)$ of variables in φ as $\bigcup_{C \in \varphi} var(C)$.

We denote the set $\{0, 1\}$ of boolean values by $\mathbb{B}$. An *assignment* σ of a set of variables V is a function $V \to \mathbb{B}$. We denote the domain of an assignment σ by $\mathbf{dom}(\sigma)$. The set of all assignments of V is denoted by $\mathbb{B}^V$. Let σ be an assignment of V, then for a set of variables W with $W \subseteq V$, we define the restriction of σ to W as the function $\sigma|_W$ that is defined by $\sigma|_W(x) = \sigma(x)$ for each x in W. An assignment σ can be extended to literals by setting $\sigma(\neg v) = 1 - \sigma(v)$. Moreover, for two disjoint sets of variables V and W and assignments σ_1 of V and σ_2 of W, we define the union of σ_1 and σ_2 as $\sigma_1 \cup \sigma_2 : V \cup W \to \mathbb{B}$ such that $\sigma_1 \cup \sigma_2(x) = \sigma_1(x)$ for $x \in V$, and $\sigma_1 \cup \sigma_2(x) = \sigma_2(x)$ for $x \in W$. We define the *instantiation* $C[\sigma]$ of a clause C by an assignment σ as follows. If there is a variable $v \in \mathbf{dom}(\sigma)$ and a literal $\ell \in C$ with $var(\ell) = v$ such that $\sigma(\ell) = 1$, then $C[\sigma] = \top$. Otherwise, $C[\sigma] = \{\ell \in C \mid var(\ell) \notin \mathbf{dom}(\sigma)\}$. If $C[\sigma] = \top$ we say that σ *satisfies* C.

Similarly, we can define instantiations of terms. We define the instantiation $\varphi[\sigma]$ of a CNF φ by an assignment σ as follows. If every clause $C \in \varphi$ is satisfied by σ then $\varphi[\sigma] = \top$. Otherwise, $\varphi[\sigma] = \{C[\sigma] \mid C \in \varphi, C[\sigma] \neq \top\}$. We say that φ is *satisfied* by σ if $\varphi[\sigma] = \top$. A CNF φ is *satisfiable* if there is an assignment that satisfies φ, and it is *unsatisfiable* if there is no satisfying assignment. Whenever convenient, we identify an assignment σ of a set X with the term $\{x \mid x \in X, \sigma(x) = 1\} \cup \{\neg x \mid x \in X, \sigma(x) = 0\}$. In particular, for an assignment σ we identify $\neg\sigma$ with the clause that is the result of negating the associated term.

Quantified Boolean formulas (QBF) generalize propositional logic by introducing universal and existential quantifiers. We only consider closed formulas in conjunctive normal form. Such a QBF Φ has the shape $\mathcal{Q}.\varphi$, where φ denotes the *matrix* of Φ and $\mathcal{Q}$ the *prefix* of Φ. The prefix is given by a sequence $Q_1 x_1 \ldots Q_n x_n$, where for each $i \in [n]$, $Q_i \in \{\forall, \exists\}$ and x_i is a variable. We denote the existentially quantified variables by $var_\exists(\Phi)$ and the universally quantified variables by $var_\forall(\Phi)$. Moreover, we assume for each $1 \leq i < j \leq n$ that $x_i \neq x_j$, i.e., variables are not repeated in the prefix. The prefix defines an ordering $<_\mathcal{Q}$ on the variables. A variable x_i is smaller than a variable x_j if x_i is left of x_j in the prefix, i.e., $i < j$. The matrix is a propositional formula φ where each variable occurs in the prefix. A propositional formula can be considered as a QBF where each variable is existentially quantified.

For each existential variable e we can define its *dependencies* $D(e)$ by $D(e) = \{x \in var_\forall(\Phi) \mid x <_\mathcal{Q} e\}$. A *Skolem function* f_e for an existential variable e is a function in $\mathbb{B}^{D(e)}$. Let $f = (f_e)_{e \in var_\exists(\Phi)}$ be a family of Skolem functions. For a universal assignment σ, we define the existential assignment $f(\sigma)$ by $f(\sigma)(e) = f_e(\sigma|_{D(e)})$. The family f is a *model* of the QBF Φ if for every assignment σ of the universal variables the assignment $\sigma \cup f(\sigma)$ satisfies φ. A QBF is said to be *satisfiable* if it has a model, and it is *unsatisfiable* if it does not have one. Deciding satisfiability of a QBF is a PSPACE-complete problem [SM73].

For a thorough overview of QBF including proof systems and solving techniques we refer to the Handbook of Satisfiability [KB21; GMN21].

8.1.2 Graphs

In this section, we will briefly introduce concepts and notations from graph theory, used in this thesis. For a comprehensive introduction to graph theory, we refer the reader to Diestel's book [Die00].

A *directed graph* G is a pair (V, E) such that $E \subseteq V \times V$. We denote elements of V as *vertices* and elements of E as *edges* of the graph G. Given a directed graph G, we denote its vertices by $V(G)$ and its edges by $E(G)$. A directed graph is *finite* if it has a finite number of vertices and edges. Throughout this thesis we will only consider finite directed graphs. Thus, if we talk about graphs we always mean finite directed graphs.

Let G be a graph with vertices v_1 and v_2 such that G contains the edge (v_1, v_2). Then we denote v_2 as a *successor* of v_1 and v_1 as a *predecessor* of v_2. Let $X \subseteq V(G)$ be a set of vertices then we define the graph $G - X$ as the graph with vertices $V(G) \setminus X$ and edges $\{(v_1, v_2) \in E(G) \mid v_1 \notin X, v_2 \notin X\}$.

A *path* P in a graph G is a sequence of vertices $v_0, v_1, \dots, v_n$ with $(v_{i-1}, v_i) \in E(G)$ for each $i \in [n]$. For every $i \in [n]$ we say that P contains the edge (v_{i-1}, v_i). Moreover, we say that P *connects* the vertices v_0 and v_n. If there is a path connecting two vertices x and y, we say that x is *connected* to y. A *subpath* of a path P is a contiguous subsequence of P. The *length* of a path P is the number of edges in P. Let P be a path of length ℓ with $\ell \geq 1$, then P is *cyclic* if $v_0 = v_\ell$. A graph is *cyclic* if it contains a cyclic path and *acyclic* if it does not. A graph that is both directed and acyclic is often referred to as *directed acyclic graph* (DAG). All graphs considered in this thesis are DAGs.

8.1.3 Boolean Circuits

A *Boolean function* f is a function $f : \mathbb{B}^n \to \mathbb{B}^m$, where $n, m \in \mathbb{N}^*$. We denote the set of all Boolean functions from $\mathbb{B}^n$ to $\mathbb{B}^m$ by $(\mathbb{B}^m)^{\mathbb{B}^n}$. A Boolean function f is *normal* if $f(0, \dots, 0) = (0, \dots, 0)$. We define the *normalization* of f, as the function $\overline{f}(x_1, \dots, x_n) = f(x_1, \dots, x_n) \oplus f(0, \dots, 0)$.

We define a *Boolean circuit* $\mathcal{C}$ as a non-empty labeled DAG (cf. *logic networks* [DeM94]).[1] A Boolean circuit is *k-regular* if there is some $k \in \mathbb{N}^*$ with $k > 1$ such that every vertex in $\mathcal{C}$ has either 0 or k incoming edges. Unless stated otherwise, we assume that circuits are *k-regular* for some $k \in \mathbb{N}^*$ with $k > 1$. We refer to the vertices without an incoming edge as *primary inputs* ($\mathrm{in}(\mathcal{C})$) and to the remaining vertices as *gates* ($\mathrm{gates}(C)$). Next, we require that G is a vertex labeling function that maps each gate to a function f in $\mathbb{B}^{\mathbb{B}^k}$. For the sake of simplicity, we will refer to the function associated to a vertex by the vertex itself. The size of a circuit $\mathcal{C}$, denoted by $|\mathcal{C}|$, is the number of its gates. Moreover, we assume that some ordering for the primary inputs is given—whenever convenient, we index primary inputs according to this ordering. Finally, O shall be a non-empty set of vertices of $\mathcal{C}$. We call the elements of O *primary outputs*, and for a circuit $\mathcal{C}$ we denote its primary outputs by $\mathrm{out}(\mathcal{C})$. Similarly as for the primary inputs we assume that some ordering for O is given, and we index primary outputs according to this ordering.

A special type of circuit, which we will consider, are *And-Inverter Graphs* (AIGs). AIGs are circuits whose gates have either one or two incoming edges, where each fanin-2 gate is labeled with the *and*-function and each fanin-1 gate is labeled with the *not*-function. Usually, we are not interested in the number of *and*-gates. Thus, we define the size of an AIG as the number of occurring *and*-gates [Mis+05].

Let g be a gate in a circuit $\mathcal{C}$, we define the *fanin* of g, denoted by $\mathrm{fanin}(g)$, as the set of vertices that have an edge to g. For each gate g we require that there is some ordering of its fanin. Similarly, we define the *fanout* of a vertex v, denoted by $\mathrm{fanout}(v)$, as the set of all gates that have an edge from v. The *transitive fanin cone* $\mathrm{TFI}(v)$ is defined as the set of all vertices w from which there is a path to v. The *transitive fanout cone*

[1]There are also alternative representations, e.g., *Boolean chains* [Knu11].

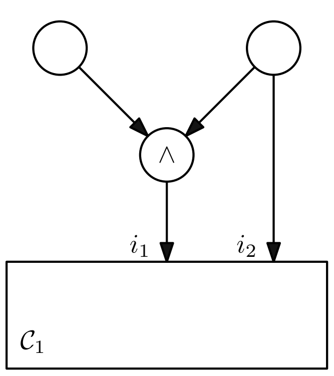
TFO(v) is defined as the set of all vertices w for which there is a path from v to w. These definitions can be extended to sets of vertices. For a set of vertices V we define the transitive fanin (fanout) cone as $\bigcup_{v \in V} \mathrm{TFI}(v)$, respectively as $\bigcup_{v \in V} \mathrm{TFO}(v)$. Moreover, for a vertex n in a circuit $\mathcal{C}$ we define its *level* $\mathrm{lv}(n)$ as 0 if n is a primary input and else as $1 + \max(\{\mathrm{lv}(x) \mid x \in \mathrm{fanin}(n)\})$.

A boolean circuit $\mathcal{C}$ *computes* a Boolean function $\mathcal{C}_f : \mathbb{B}^{|\operatorname{in}(\mathcal{C})|} \to \mathbb{B}^{|\operatorname{out}(\mathcal{C})|}$. To define the function $\mathcal{C}_f$, we first recursively define for each vertex v the Boolean function $v^{\mathcal{C}} : \mathbb{B}^{|\operatorname{in}(\mathcal{C})|} \to \mathbb{B}$. Now let $b \in \mathbb{B}^{|\operatorname{in}(\mathcal{C})|}$ be arbitrary but fixed. For every primary input v_j we define $v_j^{\mathcal{C}}(b) = b_j$. For every gate v we define $v^{\mathcal{C}}(b) = v(v_1^{\mathcal{C}}(b), \dots, v_k^{\mathcal{C}}(b))$, where $\mathrm{fanin}(v) = \{v_1, \dots, v_k\}$ and for each $i \in [k]$ the vertex v_i is the i^{th} fanin of v. We refer to $v^{\mathcal{C}}(b)$ as the value of v under b. Then $\mathcal{C}_f$ is defined as $(v_1^{\mathcal{C}}(b), \dots, v_{|\operatorname{out}(\mathcal{C})|}^{\mathcal{C}}(b))$, where for each $i \in [|\operatorname{out}(\mathcal{C})|]$ the vertex v_i is the i^{th} primary output of $\mathcal{C}$. Whenever convenient, we identify a circuit and the Boolean function it computes. Given two circuits $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same number of primary inputs and outputs, we say that $\mathcal{C}_1$ and $\mathcal{C}_2$ are *equivalent*, denoted as $\mathcal{C}_1 \equiv \mathcal{C}_2$, if they compute the same Boolean function. Additionally, whenever convenient, we identify assignments of the inputs and outputs of a circuit with tuples of Boolean values and vice versa. An assignment γ of the primary inputs of $\mathcal{C}$ induces an assignment σ of $V(\mathcal{C})$ that is defined as $\sigma(v) = v^{\mathcal{C}}(\gamma)$. A circuit is *normal* if every gate is a normal function. It can easily be shown that a normal circuit computes a normal function. For this purpose, we can first verify that the vertices of a circuit $\mathcal{C}$ can be topologically sorted, i.e., there is a linear order $<_{\mathcal{C}}$ for the vertices in $\mathcal{C}$ such that for each gate g and each $x \in \mathrm{fanin}(g)$ we have $x <_{\mathcal{C}} g$. Now we can prove by induction—with respect to this ordering—that all gates yield false if all primary inputs are false. This shows that the computed function is normal.

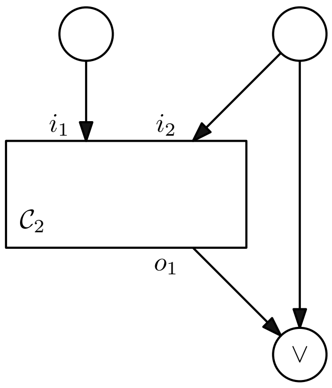
Let $\mathcal{C}$ be a circuit and $V' \subseteq V(\mathcal{C})$ a set of vertices such that for each $v \in V'$ either every fanin of v or no fanin of v is contained in V'. Then V' induces a subcircuit $\mathcal{S}$ that is given by the graph $\mathcal{C} - (V(\mathcal{C}) \setminus V')$. As before the primary inputs are the vertices in $\mathcal{S}$ without fanins and the remaining vertices are the gates. Each gate in $\mathcal{S}$ gets labeled by the same function as in $\mathcal{C}$ and also the fanins are ordered the same way. The primary outputs of $\mathcal{S}$ are the vertices that are primary outputs of $\mathcal{C}$ and the vertices that have fanouts in $V(\mathcal{C}) \setminus V(\mathcal{S})$. Typically, our main interest in a subcircuit $\mathcal{S}$ is its functionality within the encompassing circuit $\mathcal{C}$ and not a particular Boolean function it computes. Therefore, we can fix some arbitrary ordering for the primary inputs and outputs of subcircuits.

8.1.4 Don't cares

Let $\mathcal{C}$ be a circuit and $\mathcal{S}$ a subcircuit of $\mathcal{C}$. *Don't cares* of the circuit $\mathcal{S}$ are certain patterns of its inputs, or outputs that do not have an effect on the function computed by the encompassing circuit $\mathcal{C}$. We distinguish between two types of don't cares. *Controllability don't cares* are patterns of the inputs of $\mathcal{S}$ which cannot be attained within $\mathcal{C}$, and *observability don't cares* are patterns of the inputs of $\mathcal{S}$ for which its outputs do not have an effect on $\mathcal{C}$ [DeM94]. A method for computing don't cares is given in [MB05]. If some



(a) Controllability don't cares



(b) Observability don't cares

Figure 8.1: The left image shows a controllability don't care of a subcircuit $\mathcal{C}_1$. One can see that whenever the first input i_1 of $\mathcal{C}_1$ is true then also its second input i_2 needs to be true. This means that the input pattern where i_1 is true and i_2 false cannot be attained within the entire circuit. The right image shows an observability don't care of a subcircuit $\mathcal{C}_2$. Here, one can see that whenever the input i_2 is true, then the value of the output o_1 does not matter, as the subsequent *or*-gate yields true anyway.

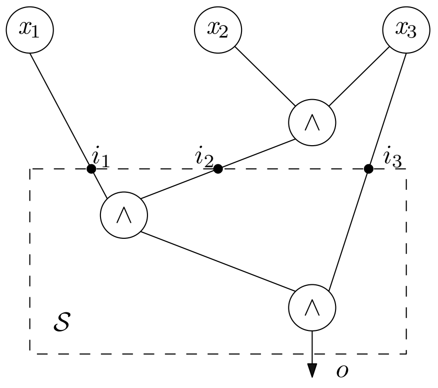
assignment of the inputs of the subcircuit is never attained, we can change the values of each output yielded by the subcircuit for this input assignment, without altering the function computed by the encompassing circuit. Similarly, if for some input assignment of the subcircuit there is some output that has no effect on the entire circuit, then this output can be modified without altering the function computed by the encompassing circuit. Consequently, don't cares can be used to improve a circuit [Rie+22; DeM94; Rie+19; Tes+20].

8.1.5 Boolean Relations

For $n, m \in \mathbb{N}^*$ a *Boolean relation* is a function[2] $R : \mathbb{B}^n \to (\mathcal{P}(\mathbb{B}^m) \setminus \{\emptyset\})$ [BHS90]. A Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ is *compatible* with a Boolean relation R for $\mathbb{B}^n$ and $\mathbb{B}^m$ if, for every $\sigma \in \mathbb{B}^n$ we have $f(\sigma) \in R(\sigma)$. A Boolean circuit $\mathcal{C}$ *implements* a Boolean relation R if the function computed by $\mathcal{C}$ is compatible with R.

Boolean relations can be used to represent don't cares of a subcircuit $\mathcal{S}$ in a circuit $\mathcal{C}$. This can be achieved by assigning each input assignment to a set of permissible output assignments, i.e., every circuit computing the relation could be used for replacing the subcircuit without altering the function computed by the encompassing circuit. We illustrate this idea in Figure 8.2. Boolean relations have been used for representing don't cares of multi-output subcircuits for reducing the size of circuits [Lee+18].

[2]Alternatively a Boolean relation can be defined as a relation $R \subseteq \mathbb{B}^n \times \mathbb{B}^m$. We follow the definition of Brayton et al. [BHS90] that for each $x \in \mathbb{B}^n$ there must be at least one $y \in \mathbb{B}^m$ that is related to x—relations with this property are also called *well-defined* [Lee+18]. Such relations can be naturally represented by functions on $\mathbb{B}^n$.



i_1	i_2	i_3	o
0	0	0	$\{0\}$
0	0	1	$\{0\}$
0	1	0	$\{0,1\}$
0	1	1	$\{0\}$
1	0	0	$\{0\}$
1	0	1	$\{0\}$
1	1	0	$\{0,1\}$
1	1	1	$\{1\}$

Figure 8.2: The table on the right gives a Boolean relation for the subcircuit $\mathcal{S}$. One can see that the input pattern $i_2 = 1, i_3 = 0$ is not possible within the entire circuit. The relation given in the table makes use of this controllability don't care by assigning $\{0, 1\}$ to the corresponding assignments of the inputs.

8.2 Exact Synthesis

Exact synthesis is the problem of finding an optimal circuit with respect to some metric. This in particular includes the problem of finding minimum size (also called minimum area) circuits and minimum depth (also called minimum delay) circuits. Typically, one is interested in circuits of a particular structure. This includes constraints on the fanin size of the gates, on the functions used for the individual gates or the depth of the circuit. For example, this might mean that we are only interested in And-Inverter Graphs (AIGs) or Majority-Inverter Graphs (MIGs).

To compute optimal size circuits various techniques have been developed. While in this thesis we are only considering SAT-based methods, we want to briefly mention some non-SAT-based ones. The *Quine–McCluskey algorithm* [Qui52; McC56] computes minimum size sum-of-product (SOP) circuits. Davidson's method [Dav69] allows computing minimum size circuits for multi-output functions consisting of *NAND* gates. Lawler's method [Law64] computes size optimal circuits for single output functions where each gate either represents the *and* or *or* function and only primary inputs may be negated.

Several SAT-based methods for exact synthesis have been developed. The first application of SAT to the problem of exact synthesis, we are aware of, was proposed by Eén [Eén07]. In that work, a SAT encoding was used to compute size optimal AIGs. A more general SAT encoding that allows computing minimum size circuits with arbitrary fanin-2 gates was proposed by Kojevnikov et al. [KKY09]. A different SAT encoding for the same task was proposed by Knuth [Knu15, Solution to exercise 477]. Later, Haaswijk et al. [Haa+20] compared different SAT encodings. Here the *single selection variable encoding* corresponds to Knuth's encoding. Similar SAT encodings were also used to compute minimum size circuits consisting of different kinds of fanin-3 gates [Mar+21]. To compute minimum depth circuits with arbitrary 2-fanin gates, Soeken refined Knuth's encoding [SMM17]. Moreover, Soeken introduced *satisfiability modulo theory (SMT)* encodings [Soe+17] to find minimum size, respectively minimum depth Majority-Inverter Graphs (MIGs). Similarly, size optimal sum-of-product (SOP) circuits have been computed via SAT by Ignatiev et al. [IPM15]. The purpose of the above list is to give an overview and does not claim to be complete.

In this thesis, we are interested in computing size optimal circuits using k-fanin gates. Thus, in the following we will refer to this problem as exact synthesis.

The TWOEXACT method provided by ABC [BM10] allows computing minimum size circuits with fanin-2 gates. To compute minimum size circuits consisting of fanin-k gates the PERCY library [Soe+22] can be used.

It was shown that computing minimum size SOPs is NP-complete [UVS06]. Similarly, it was also shown that computing minimum size circuits computing multi-output functions consisting of binary *and*, *or*, and *not* gates is NP-complete [ILO20]. This corresponds to the observation that in practice exact synthesis is limited to very small circuits with not many more than 10 fanin-2 gates [KKY09; CMM23].

8.3 Circuit Minimization

As computing provably minimal circuits is limited to very small circuits, in practice, heuristic methods need to be considered to find small circuits. Several different methods for improving circuits exist. Among these methods, those that fully capture the properties of Boolean functions implemented by circuits (rather than viewing them as polynomials, for instance) are considered the most effective in logic optimization [Tes+20]. In this thesis we will solely consider these methods. *Circuit rewriting* [MCB06] improves a circuit by replacing subcircuits by smaller circuits that do not change the Boolean function computed by the entire circuit. Circuit rewriting can be combined with exact synthesis by replacing subcircuits with the minimum size circuit computing the respective functions [RMS20; KPS22]. We denote such minimum size circuits as *locally optimal* as they do not make use of the structure of the entire circuit. By taking don't cares into account, subcircuits can also be replaced by circuits that compute different functions. We denote the minimum size circuit that makes use of don't cares as *globally optimal*. Riener et al. combined exact synthesis and don't cares to replace single output subcircuits by globally optimal circuits [Rie+19].

Another circuit minimization technique is *circuit resubstitution*. In resubstitution, one tries to express a gate g in terms of other gates in the circuit. This means that we first want to find some gates already present in the circuit. Then we introduce new gates that can depend on the previously found gates. These gates must be chosen such that using them to replace g does not alter the function computed by the circuit. Replacing g also allows to remove every gate in the maximum fanout free cone (MFFC) of g—after removing g these gates would no longer be connected to a primary output. If replacing g results in a smaller circuit, we perform the replacement [Rie+22; Mis+11; Ama+18]. The above list is not complete by any means and is intended to give a small overview.

Several other methods exist; for example detection and merging of functional equivalent gates [Mis+05] or the *transduction method* that adds additional fanins to gates and then tries to remove redundant fanins [Miy24].

Many circuit minimization techniques are implemented in the industrial-strength tool ABC [BM10].

8.4 The SAT-based Local Improvement Method (SLIM)

Even if a SAT encoding for solving some problem is known, it may be infeasible to encode certain problem instances and check them by a SAT solver due to the complexity of the problem. To still make use of SAT solving, the SAT-based Local Improvement Method (SLIM) metaheursitic can be considered. For applying SLIM we first need to compute some initial solution for the problem. This is done by applying some heuristic method that is still capable of handling the problem instance. In general this initial solution is not optimal, thus we want to improve it. For this purpose, we repeatedly select parts of the initial solution. These parts need to be chosen such that they still can be encoded and checked by a SAT solver. This allows to compute a replacement for the initial solution by means of a SAT solver. Here it is crucial that the replacement is chosen such that, upon inserting the replacement into the current global solution does not destroy the solution—i.e., the solution obtained by the replacement must still be a valid solution for the original problem. We can now repeat the selection of parts and their replacement as long as necessary. This means SLIM is an *anytime* heuristic—i.e., realizations of the SLIM paradigm do not have a fixed termination condition, and they can be run indefinitely.

The *SAT-based Local Improvement Method (SLIM)* was first used for computing branch decomposition of graphs with small branch width [LOS16]. Since then SLIM has been applied to various problems from different domains. These include further graph decomposition problems [LOS17; FLS17; LOS19; RS20]. Moreover, SLIM has been applied for Bayesian network structure learning [RS21a; RS21b], graph coloring [Sch22; SS23] and finding low complexity decision trees [SS21a]. SLIM is not limited to SAT. Instead of using SAT solvers also other SAT-related solving techniques like MaxSAT can be considered [RS21b].

8.5 Applications of QBF for Circuit Synthesis

In the context of logic synthesis, QBF have been used for bi-decomposition [CJM12], synthesis of reversible quantum circuits [Wil+08], and synthesis of lookup tables (LUTs) [Fuj+13; Fuj15; Fuj+20]. The latter two problems are more constrained than the setting considered here in that the topology of the circuits is fixed, whereas the synthesis tasks we encode as QBF also involve deriving a suitable topology. Moreover, the application of DQBF to partial equivalence checking is related to synthesis as one can obtain an implementation for the missing parts from a model [Git+13b; Git+13a].

CHAPTER 9

Encodings for Exact Synthesis

In this section, we will first discuss that it is sufficient to consider normal circuits for exact synthesis. Then we will recap a SAT encoding for exact synthesis from the literature. Finally, we will introduce a QBF encoding based on this SAT encoding.

9.1 Normalizing Circuits

As briefly noted by Knuth [Knu11], it suffices to consider normal circuits for the exact synthesis of Boolean functions. Making use of normal circuits allows reducing the search space for exact synthesis, since in a normal circuit there are only half as many functions to consider for each gate. Below, we will argue why this restriction is possible.

Let $\mathcal{C}$ be a circuit computing a normal function f. We will show that there is a normal circuit with the same size as $\mathcal{C}$ that computes f. For this purpose, we define *normalization* $\overline{\mathcal{C}}$ of $\mathcal{C}$. The normalization $\overline{\mathcal{C}}$ shall only consist of normal gates. For every primary input v of $\mathcal{C}$ the normalization shall contain a primary input $\overline{v}$. The ordering of the inputs in $\overline{\mathcal{C}}$ is given by the ordering of the corresponding primary inputs in $\mathcal{C}$. Additionally, for every gate g in $\mathcal{C}$ we add a gate $\overline{g}$ to $\overline{\mathcal{C}}$. Since we only allow normal gates in $\overline{\mathcal{C}}$, we represent non-normal gates from $\mathcal{C}$ by negating them and modifying their fanouts accordingly. We say that a gate $\overline{g}$ in $\overline{\mathcal{C}}$ is marked as negated if it represents $\neg g$. This means that for a gate g in $\mathcal{C}$, we define $\overline{g}$ as follows. Let us assume that g has fanins $g_1, \ldots, g_k$. Then the fanins of $\overline{g}$ are given by $\overline{g_1}, \ldots, \overline{g}_k$. To define the function for $\overline{g}$, we first define the function $g'(x_1, \ldots, x_k) = g(y_1, \ldots, y_k)$, where for each $i \in [k]$ we have $y_i = \neg x_i$ if $\overline{g_i}$ is marked as negated and $y_i = x_i$ otherwise. Next we define $\overline{g}(x_1, \ldots, x_k) = g'(x_1, \ldots, x_k)$ if g' is normal, otherwise we define $\overline{g}(x_1, \ldots, x_k) = \neg g'(x_1, \ldots, x_k)$. In the latter case, we say that $\overline{g}$ is marked as negated. Finally, if a vertex v is a primary output of $\mathcal{C}$ then $\overline{v}$ is a primary output of $\overline{\mathcal{C}}$. Similarly, as the ordering of the primary inputs also the ordering of the primary outputs of $\overline{\mathcal{C}}$ is given by the ordering of the corresponding primary outputs in $\mathcal{C}$. We can immediately see that $\overline{\mathcal{C}}$ is indeed a normal circuit and that $|\mathcal{C}| = |\overline{\mathcal{C}}|$. One can also easily verify that $\mathcal{C}$ and $\overline{\mathcal{C}}$ compute the same Boolean function.

Lemma 9.1. *Let $\mathcal{C}$ be a circuit computing a normal Boolean function. Then $\mathcal{C}$ and $\overline{\mathcal{C}}$ compute the same function.*

Proof. Let $b \in \mathbb{B}^{|\operatorname{in}(\mathcal{C})|}$ be arbitrary but fixed. We will first show that for each vertex v in $\mathcal{C}$ we have $v^{\mathcal{C}}(b) = \neg\overline{v}^{\overline{\mathcal{C}}}(b)$ if v is marked as negated and $v^{\mathcal{C}}(b) = \overline{v}^{\overline{\mathcal{C}}}(b)$ otherwise. This can be proved by induction. Obviously, the property holds for the primary inputs. So let g be a gate with fanins $g_1, \dots, g_k$ and assume the property holds for all fanins. If g is not marked as negated then we know that $\overline{g}(x_1, \dots, x_k) = g'(x_1, \dots, x_k)$—where g' is defined as before. By the induction hypothesis it immediately follows that $g^{\mathcal{C}}(b) = \overline{g}^{\overline{\mathcal{C}}}(b)$. Otherwise, if g is marked as negated we can proceed analogously. It only remains to show that there is no primary output that is marked as negated. Assume there is an output v that is marked as negated. We know that $v^{\mathcal{C}}(b) = \neg\overline{v}^{\overline{\mathcal{C}}}(b)$ for any $b \in \mathbb{B}^{|\operatorname{in}(\mathcal{C})|}$. Since $\overline{\mathcal{C}}$ is a normal circuit, we can conclude that $\overline{v}^{\overline{\mathcal{C}}}$ is a normal Boolean function. But this means that $\mathcal{C}$ does not compute a normal Boolean function. This is a contradiction to our initial assumption. From this we can now conclude that $\mathcal{C}$ and $\overline{\mathcal{C}}$ compute the same function. □

The following corollary is an immediate consequence of the above result.

Corollary 9.1. *Let $\mathcal{C}$ be a minimum size circuit computing a normal function f, then there is a normal circuit of the same size computing f.*

As a consequence of the above result, it suffices to consider normal circuits for synthesizing normal functions. For a non-normal function f, we can first compute a normal minimum size circuit $\mathcal{C}$ computing $\overline{f}$. The circuit $\mathcal{C}$ can then be used to construct a circuit $\mathcal{D}$ computing f. For this purpose, let $n = |\operatorname{in}(\mathcal{C})|$, $m = |\operatorname{out}(\mathcal{C})|$ and $I \subseteq [m]$ such that for each $i \in I$ we have $f_i(x_1, \dots, x_n) \neq \overline{f}_i(x_1, \dots, x_n)$. The circuit $\mathcal{D}$ can be obtained by negating all vertices in $\mathcal{C}$ that represent a primary output with index in I. Additionally, we have to modify the functions for all fanouts of such gates. These modifications can be done similarly to the modifications of the fanouts of a negated gate in the normalization of a circuit. Finally, we have to add for every $i \in [m]$ such that there is a $j \in [m]$ and $f_i(x_1, \dots, x_n) = \neg x_j$ an additional gate representing $\neg x_j$. This gate is then used as the corresponding output in $\mathcal{D}$. We denote the set of all such indices $j \in [m]$ by J. Similarly, as in the proof of Lemma 9.1 we can show that $\mathcal{D}$ computes f. It remains to show that there is no smaller circuit computing f.

Lemma 9.2. *Let $\mathcal{C}$, $\mathcal{D}$ and f be defined as before. Then $\mathcal{D}$ is a minimum size circuit computing f.*

Proof. For proving this property, we will assume that there is a smaller circuit $\mathcal{D}'$ that computes f and show a contradiction. This means that $|\mathcal{D}'| < |\mathcal{D}| = |\mathcal{C}| + |J|$. We will show that from this we can conclude that there is a smaller circuit than $\mathcal{C}$ which computes $\overline{f}$. Let us assume that there is some $j \in J$ such that the vertex v representing the j^{th} output has a non-empty fanout. Moreover, let $l \in [n]$ such that $v^{\mathcal{D}'}(x_1, \ldots, x_n) = x_l$. We can easily verify that each gate $g \in \text{fanout}(v)$ can be replaced by a gate g' that uses the l^{th} primary input instead of v. Next, we can obtain a circuit $\mathcal{C}'$ from $\mathcal{D}$ by first negating the vertices representing primary outputs with an index in I and normalizing this circuit. In $\mathcal{C}'$ we can remove the vertices representing a primary output with index in J and directly use the corresponding primary inputs as primary outputs. This results in a circuit computing $\overline{f}$ with $|\mathcal{C}'| < |\mathcal{C}|$. As $\mathcal{C}$ was assumed to be a minimum size circuit computing $\overline{f}$ this is a contradiction. □

As a consequence of the above results, we can assume that Boolean functions are normal. If a function f is not normal, we can obtain a minimum size circuit from a normal circuit computing $\overline{f}$.

9.2 SAT Encoding

As pointed out in Section 8.2 several SAT encodings for exact synthesis exist. In this section, we recap the *multiple selection variable encoding* by Haaswijk et al. [Haa+20]. While, we mainly try to stick to the description from their paper, we introduce three minor modifications which we will use later. First, we allow computing minimum size circuits for constant functions. Second, we also consider functions that yield a projection to one of their inputs. Third, we modify the encoding for arbitrary k-regular circuits instead of 2-regular circuits.

The core idea of the encoding is that the whole circuit is represented by a truth table. This means that for every possible combination of the inputs of the circuit the truth table contains a line. Each line then contains a variable for each gate. These variables, can then be used to represent the value of each gate under each input pattern. We will first describe the variables used in the encoding, then the constraints on these variables, and finally we will discuss some symmetry breaking constraints.

Suppose we want to compute a minimum size k-regular circuit computing a given Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$. We assume that $k \leq n$. As discussed in Section 9.1, w.l.o.g. we can assume that f is normal. So it suffices to only consider normal circuits. The encoding does not directly allow to find a minimum size circuit, instead, it allows checking whether there is a circuit of size ℓ that computes the given function f. To find a minimum size circuit, we can then first check if there is a circuit with 0 gates representing the function. If this is the case, we found a minimum size circuit. Otherwise, we increment ℓ until we can find a circuit of size ℓ. Such a circuit is then necessarily of minimum size.

Now suppose, we want to find a circuit of size ℓ that computes the function f. For any circuit, we can find a topological ordering of its gates. Consequently, we can index the ℓ gates by 1 to ℓ. Furthermore, we can index the $n + \ell$ vertices by 1 to $n + \ell$. We, can then require that all fanins of the i^{th} gate are either primary inputs or gates with a smaller index. In the following let $i \in [\ell]$ denote the index of a gate. Moreover, let $j \in [m]$ be the index of an output. Then the encoding consists of the following variables.

Gate variables $G_i = \{g_{it} \mid 0 \leq t < 2^n\}$. These variables determine the value of the i^{th} gate under all possible assignments of the primary inputs. That is, if g_{it} is true, then the i^{th} gate yields true under the t^{th} assignment of the primary inputs, according to the truth table. As we only need to consider normal functions, we know that g_{i0}—we assume that the first line of the truth table corresponds to the case where all inputs are false—can always be assigned to false, i.e., we can omit this variable.

Selection variables $S_i = \{s_{il} \mid 1 \leq l < i + n\}$. These variables determine the fanins of gate i. The vertex l shall be a fanin of gate i if and only if the variable s_{il} is assigned to true. The order of the fanins is determined by the index l. For $k = 2$ this means that if two selection variables s_{ix} and s_{iy} with $x < y$ are assigned to true, then vertex x corresponds to the first fanin and vertex y to the second. We do not loose any generality by using this ordering, as by choosing the function at gate i properly we can account for different orderings.

Function variables $F_i = \{f^i_{b_1 \ldots b_k} \mid (b_1, \ldots, b_k) \in \mathbb{B}^k\}$. These variables describe the Boolean function f^i at the i^{th} gate. This means that $f^i(b_1 \ldots b_k)$ shall be true if and only if the variable $f^i_{b_1 \ldots b_k}$ is assigned to true. As we only need to consider normal circuits, we can assume that the i^{th} gate is a normal function. Thus, the variable $f^i_{0 \ldots 0}$ can be always assigned to false, i.e., we can omit the variable.

Output variables $O_j = \{o_{lj} \mid 0 \leq l \leq n + \ell\}$. These variables determine the j^{th} output of the circuit. If the variable o_{0j} is assigned to true, then the j^{th} output is the constant value false. As the considered function is assumed to be normal, we do not need to consider the case of constant true outputs. If, for some $l \in [n]$, the variable o_{lj} is assigned to true, then the j^{th} output corresponds to the i^{th} input of the circuit—i.e., the considered function is the projection to the i^{th} input. Finally, if the variable o_{lj} for some $n < l \leq n + \ell$ is assigned to true, then the j^{th} output is given by the $(l - n)^{\text{th}}$ gate.

Additionally, the encoding contains the following constraints.

- Each gate must have exactly k fanins. This can be ensured by requiring that for each $i \in [\ell]$, exactly k variables from S_i are assigned to true. Such constraints are called *cardinality constraints*. For the case $k = 2$ it would still suffice to represent the cardinality constraints by clauses

$$\bigwedge_{1 \leq u < v < w < i+n} (\neg s_{iu} \vee \neg s_{iv} \vee \neg s_{iw}) \wedge \bigwedge_{l=1}^{n+i-1} s_{i1} \vee \ldots \vee s_{i(l-1)} \vee s_{i(l+1)} \vee \ldots \vee s_{i(n+i-1)}$$

as described in [Haa+20]. Since we also want to handle the case $k > 2$, we want to use more efficient representations of this constraint. This can be achieved by using a sequential counter [Sin05]. We illustrate a sequential counter in Figure 9.1. The illustration shows a sequential counter that ensures that exactly two out of four variables are assigned to true. The counter can be easily modified for the case of more variables, respectively different cardinalities. For a clausal representation of sequential counters we refer to [Sin05].

- Each output of the circuit must be uniquely determined. The uniqueness can be enforced by requiring that for each $j \in [m]$, exactly one variable from O_j must be assigned to true.

- We need to ensure that for each gate i, the assignment of the gate variables G_i is compatible with the assignment of the function variables F_i and the gate variables for its fanins. In order to ensure compatibility, we use for $1 \leq x_1 < \ldots < x_k < n+i$, every $(b_1, \ldots b_k) \in \mathbb{B}^k$, and any $0 < t < 2^n$, the constraints

$$(\bigwedge_{l=1}^{k} (s_{ix_l} \wedge \hat{g}_{x_l t} = b_l)) \Rightarrow g_{it} = f^i_{b_1 \ldots b_k}.$$

  Here, $\hat{g}_{x_l t}$ denotes the gate variable $g_{(x_l - n)t}$ in case $x_l > n$, and it denotes the value of the ${x_l}^{\text{th}}$ input in the t^{th} line of the truth table otherwise. For $(b_1, \ldots, b_k) = (0, \ldots, 0)$ the constraint can be omitted since we are considering normal circuits.

- Finally, we need to ensure that the circuit computes the given function f. For this purpose, we introduce the following constraint, for each $j \in [m]$, and each line t in the truth table. If f_j yields true for the input assignment corresponding to the t^{th} line in the truth table, then we add for each $i \in [\ell]$ the constraint $\neg o_{(i+n)j} \vee g_{it}$. Otherwise, we use the constraint $\neg o_{(i+n)j} \vee \neg g_{it}$. These constraints ensure that if the i^{th} gate represents f_j, then in each line of the truth table the gate variable of gate i must be assigned according to f_j. Moreover, if f_j yields true for some argument, we add the constraint $\neg o_{0j}$—in this case the constant false cannot represent f_j. Finally, for every $l \in [n]$, if $f_j(b_1, \ldots, b_{l-1}, x_l, b_{l+1}, \ldots, b_n) \neq x_l$ for some Boolean values $b_1, \ldots, b_n$, we add the constraint $\neg o_{lj}$—in this case, f_j is not a projection to its l^{th} input.

The above encoding is sufficient to check whether there exists a circuit of size ℓ that computes the given function f. Moreover, if the encoding is satisfiable, the assignment for the variables immediately describes a circuit.

Haaswijk et al. [Haa+20] showed that in practice, the above encoding can be improved by adding additional *symmetry breaking* constraints. The purpose of these constraints is to reduce the search space by excluding irrelevant circuits. In the following, we describe a

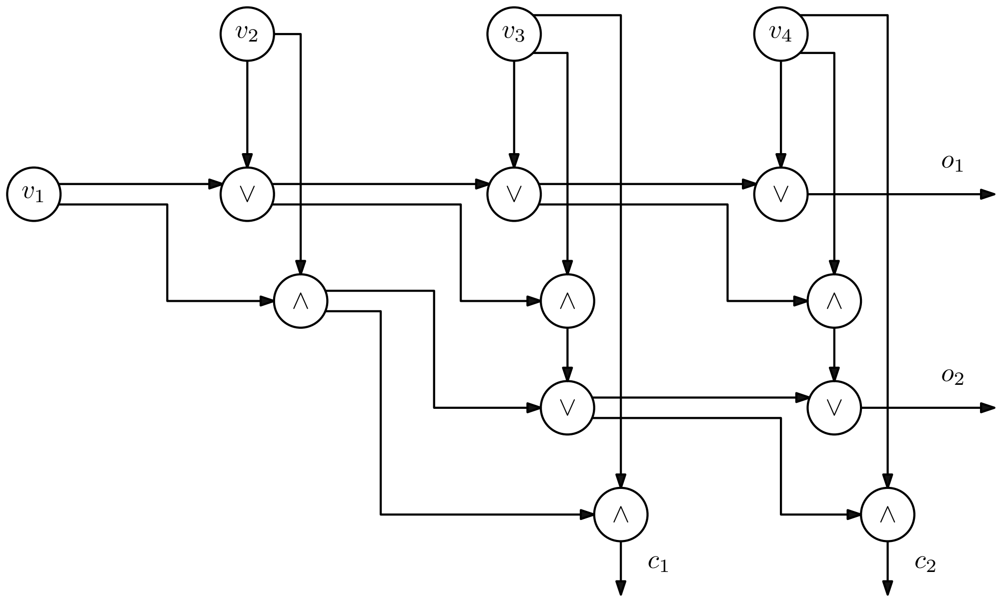


Figure 9.1: Illustration of a sequential counter that can be used to enforce that exactly two variables from the set $V = \{v_1, v_2, v_3, v_4\}$ are assigned to true. The figure shows that two variables from V are assigned to true if and only if o_2 gets assigned to true and both c_1 and c_2 get assigned to false. By requiring the mentioned assignments for o_2, c_1 and c_2 only assignments that satisfy exactly two variables from V can satisfy the encoding of the counter. We can see that o_1 gets assigned to true if and only if at least one variable from V is assigned to true. Thus, the *or*-gate representing o_1 is actually not necessary. The figure still includes o_1 as we think it makes the construction of the sequential counter clearer.

subset of the symmetry breaking constraints from their paper.[1] For the sake of simplicity, we describe these constraints for the case $k = 2$.

- The Boolean functions at each gate shall be non-trivial. This means that constant functions and projections are ruled out. This is realized by constraining the function variables. To ensure that the function is different from the constant false, we can use $f^i_{01} \vee f^i_{10} \vee f^i_{11}$ for each $i \in [n]$. The second constraint can be expressed by $f^i_{01} \vee \neg f^i_{10} \vee \neg f^i_{11}$ and $\neg f^i_{01} \vee f^i_{10} \vee \neg f^i_{11}$ for each $i \in [n]$. The function at the i^{th} gate is projection to its first fanin iff the variables f^i_{10} and f^i_{11} are assigned to true and f^i_{01} is assigned to false. Similarly, it is a projection to is second fanin iff the variables f^i_{01} and f^i_{11} are assigned to true and f^i_{10} is assigned to false. Both cases are ruled out by the constraint.

- Every gate is either a fanin of another gate or it is an output of the circuit. This

[1]We do not give the *Ordered Symmetric Variables* constraint, as this constraint is difficult to realize for the subcircuit replacement, which is our main interest and which will be described in the next chapter. Moreover, we do not consider the *(Co-)Lexicographically Ordered Operands* rule as checking the encoding does not benefit from this constraint in practice [Haa+20].

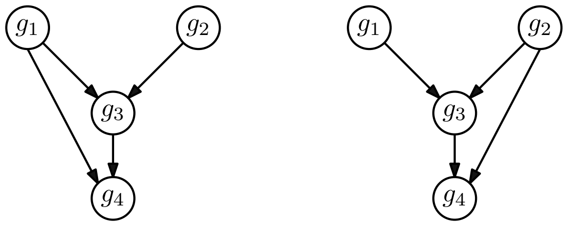


Figure 9.2: In both cases we could replace g_4 by a gate with fanins g_1 and g_2. This is possible as g_3 only depends on g_1 and g_2.

can be expressed by adding the constraint $(\bigvee_{i<j\leq\ell} s_{j(n+i)}) \vee (\bigvee_{j\in[m]} o_{(i+n)j})$ for each $i \in [\ell]$.

- If possible, we want to use a fanin x of the i^{th} gate g instead of g itself. We illustrate the idea in Figure 9.2. This idea can be realized by the clauses $\neg s_{ij} \vee \neg s_{ir} \vee \neg s_{li} \vee \neg s_{lj}$ and $\neg s_{ij} \vee \neg s_{ir} \vee \neg s_{li} \vee \neg s_{lr}$ for each $i \in [\ell]$, each $1 \leq j < r < i+n$ and each $i < l \leq i+n$.
- Gates can be ordered colexicographically according to their fanins. This means if gate i depends on vertex j, then every gate $i' > i$ must have some fanin with index greater j. We use the constraint $s_{ij} \Rightarrow \bigvee_{l=j+1}^{n+i} s_{(i+1)l}$ for each $i \in [\ell-1]$ and each $j \in [n+i-1]$. This constraint ensures that if the j^{th} vertex is a fanin of the i^{th} gate, then the $(i+1)^{\text{th}}$ gate must have a fanin whose index is at least $j+1$. Consequently, for each $i \in [\ell-1]$ the $(i+1)^{\text{th}}$ gate has at least one input with a larger index as all inputs of the i^{th} gate. From this it immediately follows that for each $i \in [\ell-1]$ and each $i < i' \leq \ell$ the i'^{th} gate has at least one fanin, whose index is greater as the indices of all fanins of the i^{th} gate.

The encoding can also be modified to compute circuits that only allow specific functions for its gates. To restrict the set of permitted functions, we can add additional constraints for the function variables of each gate. When restricting the permitted functions, care must be taken to ensure that still minimum size circuits can be found. For example, if we allow gates to be set to a function $f(x,y)$, but not to the function $g(x,y) = f(y,x)$, then it is not necessarily possible to directly find minimum size circuits. To illustrate this, suppose we are considering fanin-2 gates, and we forbid the function $\neg x \wedge y$. Obviously, such gates allow to represent the function $\neg x \wedge y$ by a single gate. In our encoding, we assume that primary inputs with a lower index are used first as a fanin. Therefore, we cannot introduce a gate representing $y \wedge \neg x$, which means that we cannot find a circuit of size one computing the function. In order to still compute a minimum size circuit with the above restrictions, we can first compute a circuit that also allows the function g for its gates. The required circuit can then be obtained by modifying each gate that is set to g by reordering its fanins and changing its function to f. Also, for AIGs some care must be taken. First, we can see that a 2-regular circuit can be obtained from an AIG by pulling all *not*-gates into the subsequent *and*-gates. For example if both fanins of an *and*-gate are *not*-gates, we can represent the *and*-gate by the function $\neg x_1 \wedge \neg x_2$. Thus, gates can attain non-normal functions. But this is not a problem, as for each gate we can change the function from $\neg x_1 \wedge \neg x_2$ to $x_1 \vee x_2$ if we negate the corresponding fanins of all fanouts of the gate. So to find a minimum size AIG we first have to impose the constraint $\neg f^i_{01} \vee \neg f^i_{10} \vee f^i_{11}$ for each $i \in [\ell]$—this ensures that no gate is set to the *XOR*-function. In a second step the computed circuit can then be easily converted to an AIG.

9.3 QBF Encoding

In this section, we will give a QBF encoding that allows to check whether there is a circuit consisting of ℓ gates that computes some Boolean function f. The encoding builds upon the multiple selection variable encoding discussed in Section 9.2, thus we will not repeat all the details.

The core idea of the encoding is to represent each input of the function by a universally quantified variable. We can then represent the individual gates by single existential variables in the scope of the universal variables. As these existential variables can depend on the universal variables, they can attain different assignments for each possible assignment to the universal variables. This is in contrast to the SAT encoding where this dependency had to be represented by means of truth tables.

Similarly as for the SAT encoding, we will now first discuss the variables occurring in the encoding and then the used constraints.

Input variables $I = \{i_1, \ldots, i_n\}$. These variables represent the inputs of the function. By universally quantifying these variables, we can represent every argument of the function f.

Gate variables $G = \{g_1, \ldots, g_\ell\}$. These variables represent the values yielded by the individual gates under an assignment for the primary inputs. In order to allow a dependence on the input variables, these variables will be existentially quantified in the scope of the input variables.

Selection variables $S_i = \{s_{il} \mid 1 \leq l < i + n\}$. Analogous to the SAT encoding, these variables determine the fanins of each gate. So the variable s_{il} is assigned to true iff the j^{th} vertex is a fanin of the i^{th} gate. As the fanins of a variable must not depend on the assignments of the primary inputs, these variables must not depend on the input variables. Thus, the selection variables are existentially quantified on the outermost level.

Function variables $F_i = \{f^i_{b_1 \ldots b_k} \mid (b_1, \ldots, b_k) \in \mathbb{B}^k\}$. Similarly, as in the SAT encoding these variables determine the function at each gate, i.e., $f^i_{b_1,\ldots,b_k}$ is true iff the function at the i^{th} gates yields true for the arguments $b_1, \ldots, b_k$. The functions used at each gate must not depend on the primary inputs, consequently the variables need to be existentially quantified on the outermost level.

Output variables $O_j = \{o_{lj} \mid 0 \leq l \leq n + \ell\}$. As in the SAT encoding the output variables determine the outputs of circuit. That means that o_{0j} is true iff the j^{th} output is the constant value false and for $l > 0$ the variable o_{lj} is true if the l^{th} vertex is the j^{th} output. The variables need to be quantified on the outermost level, for the same reason as the selection and function variables.

Now let S denote the set of all selection variables, i.e., $S = \bigcup_{i \in [\ell]} S_i$. Similarly, let F be the set of all function variables and O the set of all output variables. Then the quantifier prefix of the encoding is given by $\exists S, F, O \, \forall I \, \exists G$.

Also, the used constraints are very similar to those in the SAT encoding.

- Each gate i must have exactly k inputs. We can again represent this constraint by using a sequential counter for the selection variables. We denote this constraint by $Count(S_i, k)$.

- Each output j must correspond to exactly one vertex. We denote this constraint by $Count(O_j, 1)$.

- The assignments for the gate variables need to be compatible with the assignment for the selection variables, the function variables and the variables representing the fanins. As we represent gates by single variables, instead of truth tables, this constraint differs from the corresponding constraint in the SAT encoding. To ensure the compatibility, we have to consider for each gate all possible combinations of fanins and every assignment of these fanins. Consequently, we add the constraint

$$\bigwedge_{l=1}^{k} (s_{ix_l} \wedge \hat{g}_{x_l} = b_l) \Rightarrow g_i = f^i_{b_1 \ldots b_k} \tag{9.1}$$

  for every $i \in [\ell]$, for any $1 \leq x_1 < \ldots < x_k \leq n + i - 1$ and $(b_1, \ldots, b_k) \in \mathbb{B}^k$. Here $\hat{g}_{x_l}$ denotes the gate variable $g_{x_l - n}$ in case $n < x_l$ and it denotes the $x_l{}^{\text{th}}$ input variable otherwise. Thus, this constraint ensures that if the fanins of the i^{th} gate are assigned to $(b_1, \ldots, b_k)$ then the gate variable g_i needs to be assigned according to $f^i_{b_1 \ldots b_k}$. Since it suffices to consider normal gates, we can omit the case $(b_1, \ldots, b_k) = (0, \ldots, 0)$. We refer to these constraints as *compatibility* constraints, denoted by $Comp_i$.

- Finally, we have to ensure that the represented circuit computes the function f. This can be achieved analogously to the SAT encoding. We refer to these constraints as *correctness* constraints, denoted by $Corr$.

The QBF encoding is then given by the formula

$$\exists S, F, O\, \forall I\, \exists G.\, \mathit{Corr} \wedge \bigwedge_{j=1}^{m} \mathit{Count}(O_j, 1) \wedge \bigwedge_{i=1}^{\ell} (\mathit{Count}(S_i, k) \wedge \mathit{Comp}_i).^{2}$$

This encoding suffices to check if there is a size ℓ circuit computing f. The coding can be extended by using the same symmetry-breaking constraints as for the SAT encoding.

Finally, we can see that if the encoding is satisfiable, then we can obtain assignments of S, F and O from a model, which we can then be used to construct a circuit—similarly as with the SAT encoding.

9.3.1 Modified Compatibility Constraints

An issue with the QBF encoding (and also the SAT encoding) is that the number of cases which need to be considered for the compatibility constraints increases with the number of fanins k. For the i^{th} gate, we have to choose k fanins from the n primary inputs and the $i-1$ preceding gates. Consequently, there are $\binom{n+i-1}{k}$ possible combinations of fanins. Additionally, there are 2^k different assignments for these fanins. As, we are not interested in the case where all fanins are assigned to zero, we need to consider $2^k - 1$ different assignments. Thus, for the i^{th} gate, we need to consider $(2^k - 1) \cdot \binom{n+i-1}{k}$ cases for the compatibility constraints (eq. (9.1)). To reduce this number, we now use variables that indicate the value of each fanin for each gate (cf. with the *Distinct Input Truth Tables Encoding (DITT)* [Haa+20]). For each $i \in [\ell]$ and each $j \in [k]$, we add a *fanin variable* fi_{ij}, which shall be assigned to true if the j^{th} input of the i^{th} gate is true.

As before, we can require that fanins are ordered with respect to the ordering of vertices in the circuit. To constrain the fanin variables, we can now make use of the following observation. Let $i \in [\ell]$, $j \in [n+i-1]$ and $l \in [k]$. Then the j^{th} vertex is the l^{th} fanin of the i^{th} gate g, iff the j^{th} vertex is a fanin of g and exactly $l-1$ vertices from the set of vertices $\{v_1, \ldots, v_{j-1}\}$ are fanins of g. From this condition we can conclude that the j^{th} vertex is the l^{th} fanin of the i^{th} gate iff j is the smallest index such that exactly l variables from the set $\{s_{i1}, \ldots, s_{ij}\}$ are assigned to true. To check this condition, we can ask whether less than l variables from the set $\{s_{i1}, \ldots, s_{i(j-1)}\}$ are assigned to true and at least l variables from $\{s_{i1}, \ldots, s_{ij}\}$ are assigned to true—assuming that $j > l$. For this purpose, we can now consider the sequential counter for the selection variables for the i^{th} gate. The sequential counter contains gates x^l_j that yield true iff at least l variables from the set $\{s_{i1}, \ldots, s_{ij}\}$ are assigned to true. So, if x^l_j yields true and x^l_{j-1} yields false then the j^{th} vertex is the l^{th} fanin of the i^{th} gate g—again we assume $j > l$. We illustrate this idea in Figure 9.3.

[2]Note that the propositional part of this QBF is not a CNF. It can be converted to a CNF by Tseitin transformation, where the Tseitin variables get existentially quantified on the innermost level. In practice, we prefer a different approach. Instead of a clausal representation, we use the circuit based *QCIR* [JKS16] format that allows to directly represent the constraints. By using a circuit based representation, we also can omit the inner existential quantifier, as each gate variable is uniquely determined by the variables from the outer levels.

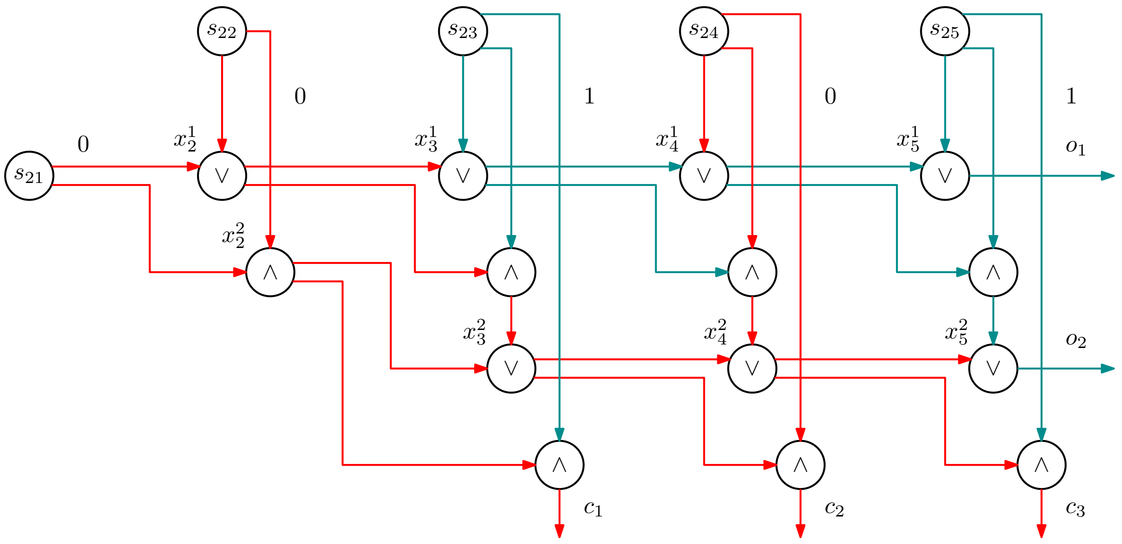


Figure 9.3: The figure illustrates the sequential counter for the selection variables of the second gate g for a circuit with four inputs. For each vertex in the figure a red outgoing edge indicates that the vertex yields false, and a blue outgoing edge indicates that the vertex yields true. Moreover, the gates x_j^i are marked. We know that the gate x_j^i gets assigned to true iff at least i selection variables from the set $\{s_{21}, \ldots, s_{2j}\}$ are assigned to true. Thus, for instance the gates x_3^1 and x_5^2 yield true, whereas the gates x_2^1 and x_4^2 yield false. As, x_2^1 yields false and x_3^1 yields true, we can conclude that 3 is the smallest index j such that one variable from $\{s_{21}, \ldots, s_{2j}\}$ is assigned to true. Consequently, the third vertex is the first fanin of g. Similarly, as x_4^2 yields false and x_5^2 yields true we can conclude that the fifth vertex is the second fanin of gate g.

In the following, we will generalize the above considerations. For this purpose, let $i \in [\ell]$ and $l \in [k]$. We can see that the first $l-1$ vertices can never be the l^{th} fanin of a gate—we remember that the ordering of the fanins is determined by the indices of the corresponding vertices. Similarly, vertices with index $j \geq n+i-k+l$ can never be the l^{th} fanin of the i^{th} gate. Additionally, for $j = l$ the condition whether the j^{th} vertex is the l^{th} fanin slightly differs from the general case discussed above. In this case it is not possible that l variables out of $\{s_{i1}, \ldots, s_{i(j-1)}\}$ get assigned to true. Consequently, in this case the j^{th} vertex is the l^{th} fanin of the i^{th} gate if and only if l variables from the set $\{s_{i1}, \ldots, s_{ij}\}$ are assigned to true, i.e., the gate x_j^l yields true.

Based on the above considerations, we now define for $i \in [\ell]$, $l \in [k]$ and $l \leq j < n+i-k+l$ a term t_{jl}^i. These terms shall be assigned to true iff the j^{th} vertex is the l^{th} fanin of the i^{th} gate. As discussed before, for $j < l$ or $j \geq n+i-k+l$, the j^{th} vertex cannot be the l^{th} fanin of the i^{th} gate. Thus, we do not need to introduce terms for these indices. First, we define $t_{11}^i = s_{i1}$—if the first selection variable is assigned to true then the first vertex is necessarily the first fanin. Next, for $1 < j \leq k$ we define $t_{jj}^i = x_j^j$. For the remaining combinations of indices, we can define $t_{jl}^i = \neg x_{j-1}^l \wedge x_j^l$.

Next, we can use these terms to constrain the fanin variables by

$$fi_{il} \Leftrightarrow \bigvee_{j=l}^{n+i-k+l-1} t^i_{jl} \wedge \hat{g}_j$$

where $\hat{g}_j$ is defined as before—i.e., it is either an input variable or a gate variable depending on the index. The intuition for this constraint is that the l^{th} fanin of the i^{th} gate g is true iff some predecessor vertex of g yields true and it is the l^{th} fanin of g.

The original compatibility constraint for the i^{th} gate can now be replaced by

$$(\bigwedge_{l=1}^{k} fi_{il} = b_l) \Rightarrow g_i = f^i_{b_1,\ldots,b_k}.$$

In this new constraint, we no longer have to distinguish between the possible combinations of fanins. Thus, we only have to consider $2^k - 1$ cases for the new compatibility constraints.

We remember that we use the circuit-based QCIR format, instead of a clausal representation of the encoding. Consequently, in the following we can count the number of gates used for the new encoding instead of the number of clauses. We can see that for each $i \in [\ell]$ in total we need $k(n+i-k-1)$ gates for representing the terms t^i_{jl}. $k(n+i-k-1)$ gates instead of $k(n+i-k)$ suffice as for each i the first term is either given by a selection variable or a gate in the sequential counter. For introducing the fanin variables we need for each $i \in [\ell]$, $k(n+i-k+1)$ gates—a fanin variable for the i^{th} gate can be represented by a disjunction of $n+i-k$ conjunctions. This means, in total we need $\sum_{i=1}^{\ell}(k(n+i-k-1) + k(n+i-k+1)) = k\ell(2n-2k+\ell+1)$ gates for introducing the fanin variables for a circuit of size ℓ with n inputs and whose gates have k fanins. By inserting the definition of each t^i_{jl} directly into the definition of fi_{il}, we could further reduce this number, but for the sake of simplicity we refrain of doing so.

To express the compatibility constraints we thus need $k\ell(2n-2k+\ell+1) + \ell(2^k-1)$ gates. Originally, we needed $(2^k-1)\sum_{i=1}^{\ell}\binom{n+i-1}{k} = (2^k-1)(\binom{n+\ell}{k+1} - \binom{n}{k+1})$ gates for the compatibility constraints.[3] For circuits with few fanins this does not make a big difference in practice. For example for $n=6, k=2, \ell=4$ we need 116 gates in the modified encoding, whereas in the original encoding we needed 300 gates. In case larger circuits are considered e.g., $n=10, k=6, \ell=4$—such circuit parameters are of relevance for circuit minimization—we can observe a larger difference. In the new encoding we need 564 gates whereas in the original encoding we need 208656 gates.

While roughly 200000 gates is still not prohibitively large, the original motivation for this modified encoding was to consider subcircuits with even more inputs for circuit minimization. In our experiments, we limited the number of inputs of subcircuits to 10 for gates with $k=6$ fanins. Thus, also the original encoding would be sufficient. Nevertheless, preliminary experiments did not show any disadvantages of using the above modifications, so we still use them.

[3]We know that $\sum_{i=0}^{n}\binom{i}{k} = \binom{n+1}{k+1}$. Shifting indices yields $\sum_{i=1}^{\ell}\binom{n+i-1}{k} = \sum_{i=n}^{n+\ell-1}\binom{i}{k}$. We can now rewrite this sum to $\sum_{i=0}^{n+\ell-1}\binom{i}{k} - \sum_{i=0}^{n-1}\binom{i}{k}$. By applying the first equality we then obtain $\binom{n+\ell}{k+1} - \binom{n}{k+1}$.

CHAPTER 10

Circuit Minimization

As noted in Section 8.2, exact synthesis can only be applied to find relatively small circuits. In this chapter, we will introduce a rewriting algorithm that is based on the SLIM paradigm. For this purpose we extend the encodings for exact synthesis discussed in Chapter 9 to compute globally optimal replacements for multi-output subcircuits.

We realize this idea in Algorithm 7, which we call eSLIM (exact Synthesis with SLIM). This algorithm takes a k-regular circuit $\mathcal{D}$[1] and a budget B which states the available time for the procedure. The algorithm then computes another k-regular circuit that computes the same function as $\mathcal{D}$ and which has at most as many gates as $\mathcal{D}$.

Similarly as for exact synthesis, it is sufficient to only consider normal circuits. For this purpose, we first normalize the given circuit. As the given circuit does not necessarily compute a normal function, the normalized circuit $\mathcal{C}$ might compute a different function as $\mathcal{D}$. To restore the original function, we keep track of all negated outputs of $\mathcal{D}$. In the end we then denormalize the minimized circuit $\mathcal{C}$ by negating the previously negated gates again. As the normalized circuit $\mathcal{C}$ only contains normal gates, every subcircuit computes a normal function. Thus, for the replacement, it is sufficient to only consider normal circuits. Consequently, after replacing a subcircuit the encompassing circuit $\mathcal{C}$ remains a normal circuit.

In the following, we will discuss the individual components of Algorithm 7. Further, we will describe how the algorithm can make use of *windows*, which enables the algorithm to handle larger circuits.

[1] In practice, this requirement can be weakened to circuits whose gates have at most k fanins. But to simplify the presentation, we will assume that each gate has exactly k fanins.

Algorithm 7 Exact synthesis with SLIM.

```
1: procedure eSLIM($\mathcal{D}$, $B$)
2:     $\mathcal{C} \leftarrow$ normalize($\mathcal{D}$)
3:     while budget remaining do
4:         $\mathcal{S} \leftarrow$ selectSubcircuit($\mathcal{C}$)
5:         $\mathcal{T} \leftarrow$ findReplacement($\mathcal{C}, \mathcal{S}$)
6:         $\mathcal{C} \leftarrow$ replace($\mathcal{C}, \mathcal{S}, \mathcal{T}$)
7:     return denormalize($\mathcal{C}$)
```

Algorithm 8 Subcircuit selection.

```
1: procedure selectSubcircuit($\mathcal{C}$, $n$)
2:     $r \leftarrow$ getRandomGate($\mathcal{C}$)
3:     $\mathcal{S} \leftarrow \{r\}$
4:     $\mathit{candidates} \leftarrow$ Queue(fanout($r$))
5:     while $|\mathcal{S}| < n \wedge |\mathit{candidates}| > 0$ do
6:         $g \leftarrow$ deQueue($\mathit{candidates}$)
7:         if random then
8:             $\mathcal{S} \leftarrow \mathcal{S} \cup \{g\}$
9:             $\mathit{candidates} \leftarrow$ enQueue($\mathit{candidates}$, fanout($g$))
10:    return $\mathcal{S}$
```

10.1 Subcircuit Selection

In this section, we will describe how subcircuits are selected. The subcircuit selection procedure is given in Algorithm 8.

To compute a subcircuit, we first pick a randomly selected root gate r. Then we expand r into a set of gates $\mathcal{S}$, which induces a subcircuit. For the expansion, we make use of randomized breadth-first search. For this purpose, we initially set $\mathcal{S}$ to $\{r\}$. Moreover, we store all fanouts of r in a queue. We will refer to this queue as *candidates*. To extend the set $\mathcal{S}$, we first dequeue the gate g from *candidates*. With a certain probability, we then add g to $\mathcal{S}$. In case we add the gate, we also enqueue all fanouts of g in *candidates*. This expansion of $\mathcal{S}$ is repeated until $\mathcal{S}$ either has sufficiently many gates or the queue *candidates* is empty. By adding the fanins of each gate in $\mathcal{S}$, we then obtain a set of vertices which induces the subcircuit that shall be replaced.

In a preliminary version of our circuit minimization algorithm, we used different strategies for expanding root gates. For a root gate r, we tried to compute an expansion that either has few inputs or few outputs. The underlying motivation for this strategy was that both the number of inputs and the number of outputs affect the size of the search space for computing a replacement circuit. Therefore, finding a replacement circuit for a subcircuit with few inputs, respectively few outputs, should be easier to handle by the SAT/QBF-based replacement procedure. Being able to handle individual subcircuits

Algorithm 9 Computing replacement circuits.

```
1: procedure findReplacement($\mathcal{C}$, $\mathcal{S}$)
2:     $n \leftarrow |\mathcal{S}|$
3:     while hasReplacement($\mathcal{C}, \mathcal{S}, n$) do
4:         $\mathcal{T} \leftarrow$ getReplacement($\mathcal{C}, \mathcal{S}, n$)
5:         $n \leftarrow n - 1$
6:     return $\mathcal{T}$
```

efficiently has an impact on the overall performance of the minimization procedure as more subcircuits can be analyzed within a given timeout. If more subcircuits can be analyzed then there is a higher potential of finding improvements. However, we achieved better results with the randomized approach, so we will not cover the other strategies. A reason why the randomized version performed better, might be that, especially in long runs of the algorithm, the greater variety of considered subcircuits in the randomized strategy is advantageous.

For the upper bounds of the subcircuit sizes we use handpicked values. These bounds are only changed if rewriting subcircuits of the given size takes too long. In this case the bound is decremented until computing new implementations of the subcircuit can be done sufficiently fast.

In a preliminary version we used a different strategy for controlling the subcircuit size. We started with a handpicked bound for the subcircuits. As in the current strategy, we decremented the bounds in case replacing subcircuits was too hard. Additionally, we also increased the bounds in case computing circuits of this size was easy. In practice, this often resulted in bounds that were too large. Since we achieved better results when not increasing bounds, we will not cover this strategy.

10.2 Synthesizing Subcircuits

In this section, we will first discuss the general framework of replacing subcircuits. Then we will discuss how to modify the QBF and the SAT encoding from Chapter 9 in order to compute replacement circuits for subcircuits. For both encodings, we do not only harness the implementational freedom within the subcircuit, but we make use of the freedom within the entire circuit by making use of don't cares.

The general framework for replacing subcircuits is given in Algorithm 9. We first compute the size of the original subcircuit $\mathcal{S}$. Then we repeatedly check if $\mathcal{S}$ can be replaced by a circuit of size n in $\mathcal{C}$. If the circuit is replaceable, we compute a replacement circuit and decrement the size n. For checking if there is a replacement circuit, we either use the SAT or the modified QBF encoding, which are discussed below. In case the check succeeds, we obtain a satisfying assignment of the encoding or an assignment for the outermost existential variables, respectively. This assignment can then be used to obtain the replacement circuit.

It may seem redundant that in the first check we use the original size of the subcircuit. Checking the original size does not result in a smaller subcircuit and serves a different purpose. As we start with the original size, in general we always replace the subcircuit by a different circuit. Even if the newly computed implementation is not smaller, its substitution modifies the circuit and can help escape local minima. In practice, we saw that this improves the overall performance of the method.

Next, we will discuss how to adapt the QBF encoding for computing a replacement circuit. Then we will discuss how to make use of Boolean relations in order to reuse the SAT encoding.

10.2.1 Extending the QBF encoding

In this section, we will describe how to modify the QBF encoding from Section 9.3 for finding replacements for subcircuits. The main difference between the QBF encoding presented in this section and the previous one is that now we do not necessarily have to find a circuit computing one particular Boolean function. Due to don't cares, circuits computing different functions might be used for replacing the initial subcircuit, without altering the function computed by the entire circuit. As we no longer want to compute a circuit for a specific function, in particular we have to modify the correctness constraints of the encoding.

In the following, let $\mathcal{C}$ denote the encompassing circuit and $\mathcal{S}$ denote the subcircuit that shall be replaced. We refer to the new implementation of $\mathcal{S}$ by $\mathcal{T}$. Let us denote the number of inputs of $\mathcal{S}$ by n and the number of its outputs by m. Additionally, we denote the primary inputs from $\mathcal{S}$ in $\mathcal{C}$ by $\{x_1, \ldots, x_n\}$ and the primary outputs by $\{y_1, \ldots, y_m\}$—we can assume arbitrary orderings. Moreover, we assume that for any output o of $\mathcal{S}$, the transitive fanout cone $\mathrm{TFO}(o)$ with respect to $\mathcal{C}$ does not contain any other gates from $\mathcal{S}$—we will consider this case later in Section 10.2.1. As in the case of exact synthesis, we want to use the encoding to check if there is a replacement circuit of size ℓ.

The encoding uses the same classes of variables as the encoding for exact synthesis. For this reason we only briefly summarize the meaning of the variables and refer to Section 9.3 for more details.

Input variables $I = \{i_1, \ldots, i_{|\operatorname{in}(\mathcal{C})|}\}$. These variables represent the inputs of the encompassing circuit $\mathcal{C}$. Note that we use an input variable for each primary input of $\mathcal{C}$ instead of one for each input of the subcircuit $\mathcal{S}$.

Gate variables $G = \{g_1, \ldots, g_\ell\}$. For each gate in the replacement circuit $\mathcal{T}$, we use a gate variable to represent the value of the gate under an assignment of the inputs of $\mathcal{C}$.

Selection variables $S_i = \{s_{il} \mid 1 \leq l < i + n\}$ for each $i \in [\ell]$. To determine the fanins of a gate in $\mathcal{T}$ we use the selection variables. The variable s_{il} is assigned to true iff the i^{th} gate in $\mathcal{T}$ uses the l^{th} vertex as a fanin.

Function variables $F_i = \{f^i_{b_1 \dots b_k} \mid (b_1, \dots, b_k) \in \mathbb{B}^k\}$ for each $i \in [\ell]$. For representing the function at a gate in $\mathcal{T}$, we use for each assignment of the fanins a function variable. The variable $f^i_{b_1 \dots b_k}$ is assigned to true iff the i^{th} gate yields true for $(b_1, \dots, b_k)$.

Output variables $O_j = \{o_{lj} \mid 0 \leq l \leq n + \ell\}$ for each $j \in [m]$. To indicate which vertex is used to represent an output of $\mathcal{T}$, we use the output variables. The variable o_{0j} is assigned to true iff the j^{th} output is the constant value false and for $l > 0$ the variable o_{lj} is assigned to true iff the l^{th} vertex yields the j^{th} output.

As in Section 9.3, we require that each gate has exactly k fanins and each output of the subcircuit must correspond to exactly one vertex.

In order to make use of don't cares, we have to proceed differently for the remaining constraints. As already mentioned before, the synthesized circuit $\mathcal{T}$ does not necessarily need to compute the same function as $\mathcal{S}$. Instead, to guarantee that replacing $\mathcal{S}$ by $\mathcal{T}$ in $\mathcal{C}$ does not alter the function computed by $\mathcal{C}$, we have to ensure that $\mathcal{C}$ and $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ compute the same function—where $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ denotes the circuit obtained by replacing $\mathcal{S}$ in $\mathcal{C}$ with $\mathcal{T}$. For this purpose, we first encode $\mathcal{C}$ in a formula $\varphi_{\mathcal{C}}$. Then we add a representation $\psi_{\mathcal{C}}$ of a copy of $\mathcal{C}$, where all gates from $\mathcal{S}$ are removed[2]. For each vertex v in $\mathcal{C}$, we denote the variable representing v in $\varphi_{\mathcal{C}}$ by $\hat{v}$. Moreover, for every vertex v in $\mathcal{C}$ that is not a gate in $\mathcal{S}$, we denote the variable representing v in $\psi_{\mathcal{C}}$ by $\tilde{v}$. In both representations, we use the input variables to represent the primary inputs. For every other vertex of $\mathcal{C}$, we require $\hat{v} \neq \tilde{v}$. In the following we will see that by combining $\psi_{\mathcal{C}}$ with the implementation for the subcircuit, given by the encoding, we obtain a representation of $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$. Thus, we can use $\varphi_{\mathcal{C}}$ and $\psi_{\mathcal{C}}$ to guarantee that the function computed by the encompassing circuit remains unchanged[3].

To realize this idea we first have to adapt the compatibility constraints. We remember that in the original QBF-based encoding, we used the compatibility constraints in order to ensure that gate variables are assigned according to the variables representing the fanins and the function variables. In case a gate used a primary input as a fanin, we used the corresponding input variable in the compatibility constraint. This is no longer possible, as now the input variables represent the primary inputs of $\mathcal{C}$. To express the dependency on the encompassing circuit, we use the variables $\{\tilde{x}_1, \dots, \tilde{x}_n\}$—i.e., the representations

[2]This can be achieved by introducing Tseitin variables for each gate. As noted in Section 9.3 we don't necessarily need a clausal encoding. Thus, in practice we represent gates from the circuit by gates in the circuit-based QCIR format.

[3]In practice, we do not have to consider the entire circuit for $\psi_{\mathcal{C}}$. Obviously, every gate that is not contained in TFO($\mathcal{S}$) is not affected by replacing $\mathcal{S}$. Thus, for every gate g not in TFO($\mathcal{S}$), we can use $\hat{v}$ instead of $\tilde{v}$, i.e., we only have to consider TFO($\mathcal{S}$) for the construction of $\psi_{\mathcal{C}}$. But for illustrating the idea, we assume that the entire circuit gets copied.

of in($\mathcal{S}$) in $\psi_{\mathcal{C}}$—instead of the input variables in the compatibility constraints. Thus, if in the new implementation $\mathcal{T}$ of the subcircuit a gate depends on the i^{th} primary input, for $i \in [n]$, then we use $\tilde{x}_i$ in the constraint. Otherwise, the compatibility constraints remain unchanged. This change of the constraint ensures that in the encoding the subcircuit is driven by the encompassing circuit.

Additionally, we have to modify the correctness constraints. First, we introduce for each $j \in [m]$, a variable $\tilde{y}_j$ that is true iff the j^{th} output of the encoded subcircuit is true. We remember that an output of the subcircuit is either the constant value false, a primary input or some gate. In the first case, the output can trivially not be true. Thus, the j^{th} output is true iff one of the following two conditions holds:

- There is some $i \in [n]$ such that $o_{ij} \wedge \tilde{x}_i$, i.e., the i^{th} input of the subcircuit describes the j^{th} output and the i^{th} input is true.
- There is some $i \in [\ell]$ such that $o_{(n+i)j} \wedge g_i$, i.e., the i^{th} gate g of the subcircuit describes the j^{th} output and g yields true.

The above considerations result in the following constraint

$$\tilde{y}_j \Leftrightarrow \bigvee_{l=1}^{n} (o_{lj} \wedge \tilde{x}_j) \vee \bigvee_{l=1}^{\ell} (o_{(n+l)j} \wedge g_l).$$

For every gate g in $\mathcal{C}$ that uses some output y_j of $\mathcal{S}$ as a fanin, we now require that in $\psi_{\mathcal{C}}$ we use $\tilde{y}_j$ in the definition of $\tilde{g}$. This ensures that $\psi_{\mathcal{C}}$ depends on the synthesized circuit. Finally, the new correctness constraint is given by $\bigwedge_{v \in \text{out}(\mathcal{C}) \cap \text{TFO}(\mathcal{S})} \hat{v} = \tilde{v}$—as mentioned before outputs not part of TFO($\mathcal{S}$) necessarily get the same assignment. We illustrate the whole idea in Figure 10.1. Note that the construction used for the correctness constraint is related to *miters*, which are used for checking equivalence of circuits [PPC96]. The difference is that for a miter, we check if there is some input assignment of the given circuits, such that the circuits differ in at least one output, whereas we check if $\mathcal{C}$ and $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ yield the same value for each output under each assignment of the inputs.

Similarly as for exact synthesis these constraints can be extended by symmetry breaking constraints. We use the same constraints as before. We refer to Section 9.2 for a description of these constraints.

A core feature of this encoding is that don't cares do not need to be explicitly represented. Instead, the encoding implicitly takes care of the don't cares. For example, if $\mathcal{S}$ has some controllability don't care, then variables in the encoding can describe an arbitrary behavior of the synthesized subcircuit for this input pattern. This is possible as there cannot be an assignment to the universally quantified input variables such that the variables $\tilde{x}_1, \ldots, \tilde{x}_n$ show the unattainable input pattern. Thus, the specific behavior for this pattern does not have an influence on the correctness constraint.

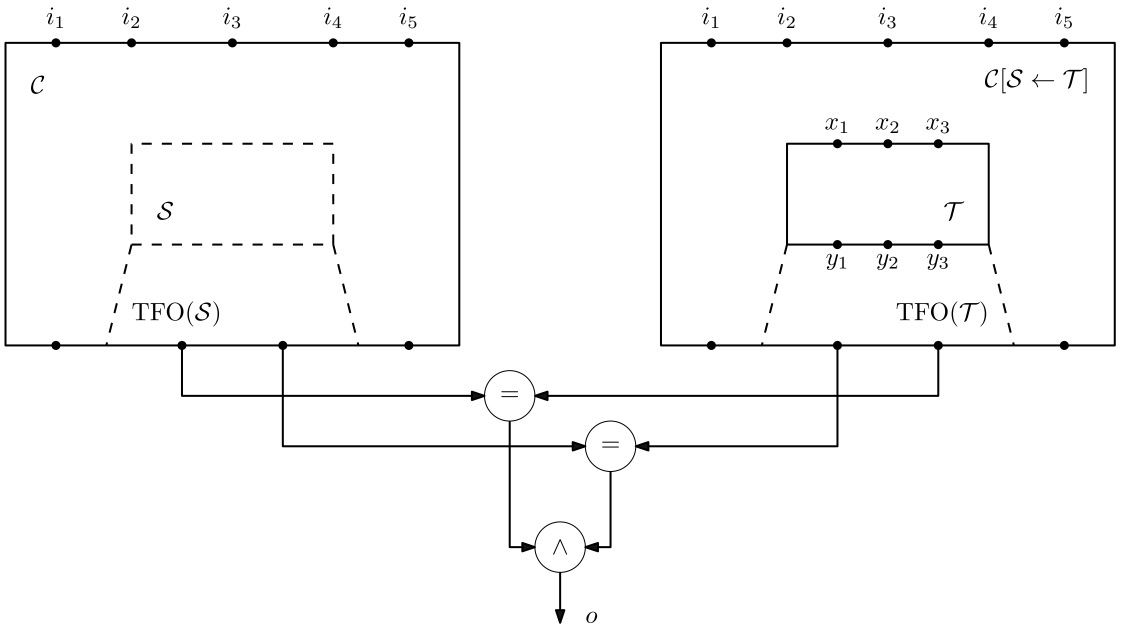


Figure 10.1: On the left side of the image we see the original circuit $\mathcal{C}$, whereas on the right side of the image we see a copy of $\mathcal{C}$ where the subcircuit $\mathcal{S}$ is replaced by $\mathcal{T}$. In both versions of the circuit the transitive fanout cone of the subcircuit is marked by dashed lines. Every output that is not contained by this cone obviously yields the same value in both versions of the circuit for every assignment of the inputs. The replacement circuit $\mathcal{T}$ must be chosen such that all remaining outputs yield the same value in both versions of the circuit for each assignment of the inputs. This can be checked, by ensuring that the gate o is true for every assignment of the inputs.

Ensuring Acyclicity

Let $\mathcal{C}$ and $\mathcal{S}$ be as before. It is possible that for some primary output β of $\mathcal{S}$ the set TFO(β) $\cap\, \mathcal{S}$ contains other vertices than β—here TFO(β) means the transitive fanout cone with respect to $\mathcal{C}$. This means $\mathcal{S}$ has a primary input α such that there is a path from β to α in $\mathcal{C}$. We illustrate this in Figure 10.2. In this case, we have to ensure that in the synthesized circuit the gate representing the output β does not depend on α.

For this purpose, we first compute all pairs (α, β) of inputs α and outputs β of $\mathcal{S}$ such that there is a path from β to α in $\mathcal{C}$—we denote these pairs as *forbidden* pairs. In practice, we compute these pairs by effectively computing the transitive fanout cones for each primary output and intersecting the cones with the subcircuit. In the following let us assume that we found at least one pair (α, β) with the above property—otherwise we can directly apply the encoding discussed in the previous section.

To ensure acyclicity, we first introduce, for each input α that occurs in a forbidden pair, a set $\{c_1^\alpha \ldots, c_{(n+i)}^\alpha\}$ of variables. We call these variables the *connection variables* for α, denoted by C_α. The purpose of the variable c_i^α is that it shall be assigned to true iff there is a path from α to the i^{th} vertex—i.e., the i^{th} vertex structurally depends on α. Now suppose we have a forbidden pair (α, β), where α is the i^{th} input of $\mathcal{S}$ and β the j^{th} output.

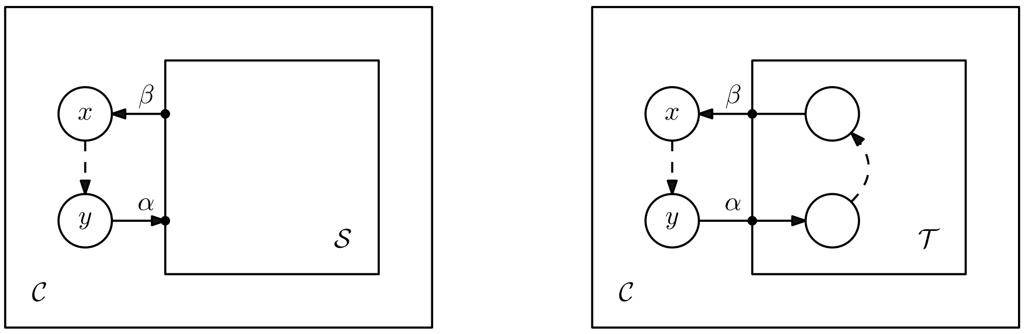


Figure 10.2: On the left we see a circuit $\mathcal{C}$ containing a subcircuit $\mathcal{S}$. Moreover, $\mathcal{C}$ contains a gate x that uses the output β of $\mathcal{S}$ as a fanin and a gate y that is connected to x, both x and y are not contained in $\mathcal{S}$. Additionally, the gate y is a fanin of the primary input α of the circuit $\mathcal{S}$. As $\mathcal{C}$ needs to be acyclic, the primary output β of $\mathcal{S}$ does not depend upon the primary input α. Synthesizing a replacement circuit changes the structure of $\mathcal{S}$. Thus, in the replacement circuit $\mathcal{T}$ the primary output β might depend on the input α. In that case, replacing $\mathcal{S}$ by $\mathcal{T}$ in $\mathcal{C}$ can cause a cycle.

Thus, there must not be a path from α to the vertex representing β—otherwise we would obtain a cycle upon replacing the subcircuit. By adding the constraint $\neg c_i^\alpha \vee \neg o_{ij}$ for every $i \in [n+\ell]$, we ensure that the vertex representing β does not depend on α.

It remains to discuss how to constrain the connection variables. First, assume that there is only a single forbidden pair. Then we generalize to the case of multiple forbidden pairs.

Let us assume that there is a single forbidden pair (α, β), where α is the i^{th} input and β the j^{th} output of $\mathcal{S}$. We first require that c_i^α is set to true—the i^{th} vertex is the input α. Next, for each $a \in [\ell]$ and each $b \in [n+a-1]$ we require that $c_b^\alpha \wedge s_{ab} \Rightarrow c_{n+a}^\alpha$. This means that if the a^{th} gate uses the b^{th} vertex as a fanin and the b^{th} vertex is connected to α then also the a^{th} gate is connected to α.

If there is more than one forbidden pair, we first add the above constraint for each pair. But this is not enough. We illustrate this in Figure 10.3. In this example, it is possible that the output y depends on the input α. As there is a forbidden pair (x, y), this means that x depends on α in this case. This in turn means that every gate g that depends on x also depends on α, thus it is not allowed that g represents the output β. In this case, the constraints used for the single pair case do not suffice to assign the connection variables properly.

To fix the constraints for this case, we now let P denote the set of all forbidden pairs, A denote the set of all inputs occurring in a forbidden pair, and B the set of all outputs occurring in a forbidden pair. Moreover, for each $x \in B$ we define the set $\Phi_x = \{a \in A \mid (a, x) \in P\}$ and the set $\Psi_x = \{a \in A \mid a \notin \Phi_x\}$. Thus, Φ_x is the set of all inputs that occur together with x in a forbidden pair, and Ψ_x is the set of all inputs occurring in some forbidden pair but not together with x. Next, let $b \in B$ be the output with index j, $x \in \Psi_b$, $y \in \Phi_b$ and $i \in [n+\ell]$ the index of a vertex. Additionally, let s be the index of the input y. We now add the constraint $(o_{ij} \wedge c_i^x) \Rightarrow c_s^x$. This means that if the

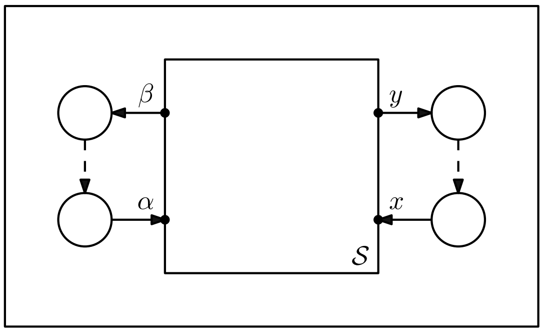


Figure 10.3: A subcircuit with two forbidden pairs (α, β) and (x, y).

vertex representing b depends on x then the vertex representing the input y also depends on x. As, $y \in \Phi_b$ we know that the input y is connected to the output b outside of the subcircuit. Since, b is connected to x in this case, also y needs to be connected to x. By adding these constraints, we can ensure that also in the case of multiple forbidden pairs the connection variables get assigned as intended. Thus, the output variables are forced to be assigned such that no cycle is introduced. We denote these acyclicity constraints by *Acyc*.

As before let S, F, O denote the sets of all selection variables, respectively function or output variables. Moreover, let C denote the set of all connection variables and V the set of all auxiliary variables that were used in $\varphi_{\mathcal{C}}$ and $\psi_{\mathcal{C}}$. Then the complete QBF encoding is given by

$$\exists S, F, O, C\, \forall I\, \exists G, V.\ \varphi_{\mathcal{C}} \wedge \psi_{\mathcal{C}} \wedge \mathit{Corr} \wedge \mathit{Acyc} \wedge \bigwedge_{j=1}^{m} \mathit{Count}(O_j, 1) \wedge \bigwedge_{i=1}^{\ell} (\mathit{Count}(S_i, k) \wedge \mathit{Comp}_i).$$

As in Section 8.2, in practice we use a circuit based representation of the above encoding. This in particular means that in case there are no forbidden pairs, we can remove the innermost existentially quantified variables, as all of them are uniquely determined by the remaining variables. Consequently, there is only a single quantifier alternation and thus the encoding is a circuit 2QBF. If there are forbidden pairs, we cannot remove all of these variables. Therefore, the encoding is not a 2QBF in this case. It is not possible to remove these variables as the QCIR format requires that gates are ordered topologically, i.e., fanins of gates are introduced before they are used. Remember that we used a variable $\tilde{y}$ to represent the value of each output y of the subcircuit. As a QCIR encoding must be ordered topologically, the definition of the gate representing $\tilde{y}$ must occur after the definitions of the gate variables—$\tilde{y}$ is defined by means of all gate variables. Suppose, we have a forbidden pair (α, β). The gate representing α must appear later in the encoding as $\tilde{\beta}$—as there is a path from β to α in the encompassing circuit. But α is also an input of the subcircuit, thus the gate representing α must be introduced before the gate variables, i.e., before $\tilde{\beta}$. As this is not possible, we need to represent $\tilde{y}$ by an existentially quantified variable in this case.

10.2.2 Extending the SAT encoding

The SAT encoding discussed in Chapter 9 could directly be used to find a locally optimal circuit, i.e., a smallest possible circuit computing the same function as the initial subcircuit. The obtained new implementations could then be used for rewriting the initial subcircuit. But this approach would not make use of don't cares, as only the current subcircuit is considered, and not the encompassing circuit.

Instead, first we will compute a Boolean relation that captures the don't cares of the subcircuit. Then we compute a circuit that implements this relation using a slightly modified version of the SAT encoding.

Computing Boolean Relations

In this section, we will discuss how to construct a Boolean relation R for a subcircuit $\mathcal{S}$ of $\mathcal{C}$ such that, for any circuit $\mathcal{T}$ that implements R, the circuit $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ computes the same Boolean function as $\mathcal{C}$. To compute such a relation, we will present an approach based on incremental SAT-solving.

In the following, let $\mathcal{S}$ be a subcircuit of $\mathcal{C}$ with n inputs and m outputs. The core idea of the presented approach is to compute a set $\Phi \subset \mathbb{B}^n \times \mathbb{B}^m$ of conflicting input-output behaviors. This set shall contain every pair (σ, ρ), such that for any circuit $\mathcal{T}$ with $\mathcal{T}(\sigma) = \rho$ the circuits $\mathcal{C}$ and $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ compute different functions. From this set we can then obtain the relation R that is defined by $R(\sigma) = \{\rho \in \mathbb{B}^m \mid (\sigma, \rho) \notin \Phi\}$. It can be easily verified that any circuit implementing R is eligible for replacing $\mathcal{S}$.

Lemma 10.1. *For every circuit $\mathcal{T}$ implementing R we have $\mathcal{C} \equiv \mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$.*

Proof. We assume the opposite. This means that there is some circuit $\mathcal{T}$ implementing R s.t. for some $\gamma \in \mathbb{B}^{|\operatorname{in}(\mathcal{C})|}$ we have $\mathcal{C}(\gamma) \neq \mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}](\gamma)$. Let σ denote the assignment to the inputs of $\mathcal{T}$ and ρ the assignment of its outputs induced by γ. We can conclude that there must be some circuit $\mathcal{T}'$ with $\mathcal{T}'(\sigma) = \rho$ that is eligible for replacing $\mathcal{S}$—otherwise the pair (σ, ρ) would be contained in Φ. This yields a contradiction, as we must have $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}](\gamma) = \mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}'](\gamma)$. □

Moreover, every circuit that is eligible for replacing $\mathcal{S}$ also implements R.

Lemma 10.2. *Let $\mathcal{T}$ be a circuit s.t. $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}] \equiv \mathcal{C}$. Then $\mathcal{T}$ implements R.*

Proof. Assume there is such a circuit $\mathcal{T}$ that does not implement R. Then there is a $\sigma \in \mathbb{B}^n$ s.t. $\mathcal{T}(\sigma) \notin R(\sigma)$. We can conclude that $(\sigma, \mathcal{T}(\sigma)) \in \Phi$. But this means that replacing $\mathcal{S}$ by $\mathcal{T}$ alters the function computed by the encompassing circuit $\mathcal{C}$. This is a contradiction to the initial assumption. □

The above considerations mean that the relation R describes all possible replacement circuits for $\mathcal{S}$. Thus, in order to find the minimum size replacement circuit for $\mathcal{S}$, we can instead compute a minimum size implementation of R.

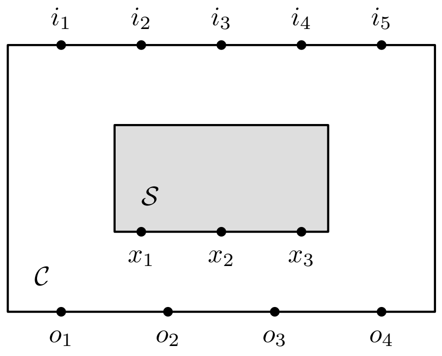


Figure 10.4: By removing the subcircuit $\mathcal{S}$ from $\mathcal{C}$ we obtain the circuit $\mathcal{C}'$. The circuit $\mathcal{C}'$ has additional inputs x_1, x_2, x_3.

To compute the set Φ, we first define the circuit $\mathcal{C}'$ as the circuit that is obtained by removing all non-output gates in $\mathcal{S}$ from $\mathcal{C}$.[4] We illustrate the construction of the circuit $\mathcal{C}'$ in Figure 10.4. In addition to the primary inputs of $\mathcal{C}$, this new circuit has an additional primary input for each output of $\mathcal{S}$. For an assignment σ for $\text{in}(\mathcal{C}')$ with $\sigma|_{\text{in}(\mathcal{C})} = \sigma_1$ and $\sigma|_{\text{in}(\mathcal{C}')\setminus\text{in}(\mathcal{C})} = \sigma_2$, we can now use $\mathcal{C}'$ to simulate the effect of replacing $\mathcal{S}$ by a circuit that yields σ_2 under the assignment σ_1 for the primary inputs of $\mathcal{C}$. Let ρ be the induced assignment for the inputs of $\mathcal{S}$. We can see that if $\mathcal{C}(\sigma_1) \neq \mathcal{C}'(\sigma)$, then any circuit $\mathcal{T}$ with $\mathcal{T}(\rho) = \sigma_2$ changes the function computed by $\mathcal{C}$ upon replacing $\mathcal{S}$. This construction can be used to find all pairs of assignments for the inputs and outputs of $\mathcal{S}$ that alter the function computed by $\mathcal{C}$. These pairs then give the set Φ.

To realize this idea, we encode both $\mathcal{C}$ and $\mathcal{C}'$ in propositional formulas φ_1 and φ_2 by Tseitin transformation. In these encodings, we use distinct variables for every vertex, except for the primary inputs of $\mathcal{C}$. For every vertex v in $\mathcal{C}$, we denote the variable representing v in φ_1 by $\hat{v}$ and for every vertex v in $\mathcal{C}'$, we denote the variable representing v in φ_2 by v'. Moreover, we denote the set of variables representing $\text{in}(\mathcal{C})$ by X, the set of variables $\{v' \mid v \in \text{in}(\mathcal{S})\}$ by Y and the set of variables representing $\text{in}(\mathcal{C}) \setminus \text{in}(\mathcal{C}')$ by Z. To ensure that $\mathcal{C}$ and $\mathcal{C}'$ differ in at least one output, we add the constraint $\bigvee_{v \in \text{out}(\mathcal{C}')} \hat{v} \neq v'$. We denote this constraint by *diff*. First, assume that the formula $\varphi_1 \land \varphi_2 \land \mathit{diff}$ is satisfied by an assignment σ. Due to the constraint *diff*, we know that for $\sigma_1 = \sigma|_X$ and $\sigma_2 = \sigma|_Z$, the circuits $\mathcal{C}$ and $\mathcal{C}'$ differ in some output, i.e., we have $\mathcal{C}(\sigma_1) \neq \mathcal{C}'(\sigma_1 \cup \sigma_2)$—here we identify assignments for vertices and variables. By restricting σ to Y we then obtain an assignment $\rho = \sigma|_Y$ such that the pair (ρ, σ_2) must be contained in Φ. If the formula is unsatisfiable no matter how we modify the outputs of $\mathcal{S}$, we cannot modify the function computed by $\mathcal{C}$. In that case, Φ is the empty set.

The above procedure computes one pair in Φ. By excluding already found pairs, we can make use of incremental SAT solving to compute Φ starting from the empty set. This

[4]In practice, we do not need to consider any gate that is not contained in $\text{TFO}(\mathcal{S})$ for the construction of $\mathcal{C}'$, as such gates are not affected by the subcircuit. This is similar to the correctness constraints for the QBF encoding. As $\mathcal{C}'$ is simpler for illustrating the idea, we consider this circuit instead.

Algorithm 10 Computing a Boolean relation for don't cares.

```
1:  procedure computeRelation($\mathcal{C}, \mathcal{S}$)
2:      $\Phi \leftarrow \emptyset$
3:      $\varphi_1, \varphi_2, \mathit{diff}, \mathit{equiv} \leftarrow \textsc{setup}(\mathcal{C}, \mathcal{S})$
4:      while $\textsc{isSAT}(\varphi_1 \wedge \varphi_2 \wedge \mathit{diff} \wedge \bigwedge_{(y,z)\in\Phi} \neg y \vee \neg z)$ do
5:          $\sigma \leftarrow \textsc{getModel}(\varphi_1 \wedge \varphi_2 \wedge \mathit{diff} \wedge \bigwedge_{(y,z)\in\Phi} \neg y \vee \neg z)$
6:          $\hat{\sigma} \leftarrow \textsc{getCore}(\varphi_1 \wedge \varphi_2 \wedge \mathit{equiv}, \sigma|_{X\cup Z})$
7:          $y \leftarrow \sigma|_Y$
8:          $z \leftarrow \hat{\sigma}|_Z$
9:          $\Phi \leftarrow \Phi \cup \{(y,z)\}$
10:     return $\Phi$
```

means, we iteratively check the formula $\varphi_1 \wedge \varphi_2 \wedge \mathit{diff} \wedge \bigwedge_{(y,z)\in\Phi} \neg y \wedge \neg z$—where φ_1, φ_2 and *diff* can be defined as before. As long as the formula is satisfiable, we can determine a new pair from a satisfying assignment. This pair can then be added to Φ. As the formula contains the subformula $\bigwedge_{(y,z)\in\Phi} \neg y \wedge \neg z$, we ensure that no pair is considered twice. Thus, eventually the formula will become unsatisfiable and Φ contains all conflicting input-output behaviors.

The described procedure for computing Φ requires to enumerate all conflicting input-output behaviors. For improving the procedure, we want to generalize the assignments. This is realized in Algorithm 10. Algorithm 10 computes a set Φ consisting of pairs of total assignments for the inputs of the subcircuit and partial assignment for its outputs. The set Φ is computed such that for each conflicting input-output behavior (σ, ρ) there is a pair $(\sigma, \rho_1) \in \Phi$ with $\rho_1 \subseteq \rho$.

To generalize the assignments of the outputs, we proceed similarly as [RS04]. We first negate the constraint *diff* and obtain the new constraint $\bigwedge_{v\in\mathrm{out}(\mathcal{C})} \hat{v} = v'$, which we will denote as *equiv*. Now we compute a subset $\hat{\sigma}$ of $\sigma|_{X\cup Z}$ such that $\varphi_1 \wedge \varphi_2 \wedge \mathit{equiv} \wedge \hat{\sigma}$ is unsatisfiable. Such a subset must exist, as we know that under $\sigma|_{X\cup Z}$, the circuits $\mathcal{C}$ and $\mathcal{C}'$ yield different values. To compute $\hat{\sigma}$, we make use of incremental SAT solving [ES03]. We solve $\varphi_1 \wedge \varphi_2 \wedge \mathit{equiv}$ under the assumption literals $\sigma|_{X\cup Z}$. We then obtain $\hat{\sigma}$ as the set of failed assumptions—the failed assumptions are those used to prove unsatisfiability, cf. with the IPASIR interface [Bal+16]. The corresponding pair, which we can add to Φ, is then given by $y = \sigma|_Y$ and $z = \hat{\sigma}|_Z$ While the above procedure reduces the size of the output assignment it leaves the input assignment unchanged. As the input assignment is entailed by the assignments of X and Z, it would be redundant to add assumptions for the inputs. Thus, it is not possible to reduce the input assignment in the same way.

All the SAT checks in Algorithm 10 can be performed by a single instance of an incremental SAT solver. To do this, we equip both the *diff* and the *equiv* constraints with selector variables that allow to disable the respective constraint. Similarly, we equip the clauses obtained from the assignment pairs with selectors in order to disable them in the reduction step.

Finally, Algorithm 10 yields a set Φ of pairs of assignments that describes all conflicting input-output behaviors. This means that for each conflicting input-output behavior (σ, ρ) there is a (partial) assignment of the outputs ρ' with $\rho' \subseteq \rho$ such that $(\sigma, \rho') \in \Phi$. If there would not be such a pair in Φ then we can see that Algorithm 10 would not have terminated yet.

Synthesizing Boolean Relations

To compute a circuit that implements a relation computed by Algorithm 10, we can apply a slightly modified version of the SAT encoding used for exact synthesis. For describing these modifications, let Φ be a set consisting of pairs of assignments for the inputs and outputs of $\mathcal{S}$ as computed by Algorithm 10.

As before, we use the following variables in the encoding.

Gate variables $G_i = \{g_{it} \mid 0 \leq t < 2^n\}$. These variables determine the value of the i^{th} gate for each line of the truth table.

Selection variables $S_i = \{s_{il} \mid 1 \leq l < i + n\}$. These variables determine the fanins of the i^{th} gate.

Function variables $F_i = \{f^i_{b_1 \dots b_k} \mid (b_1, \cdots, b_k) \in \mathbb{B}^k\}$. These variables describe the function at the i^{th} gate.

Output variables $O_j = \{o_{lj} \mid 0 \leq l \leq n + \ell\}$. These variables determine the j^{th} output of the circuit.

These variables have the same meaning as before, so we will refer to Section 9.2 for more details. We also use all constraints used before, except the correctness constraint. In order to ensure that the computed circuit implements the relation R, we first introduce, for each output $x \in \text{out}(\mathcal{S})$, and each $0 \leq t < 2^n$, a new variable y^x_t. These variables represent the value of x in the t line of the truth table. We can then enforce the intended meaning of these variables, similarly as we did for the variables $\tilde{y}_j$ in Section 10.2.1.

Remember that for each (y, z) in Φ, y is a total assignment to the inputs of $\mathcal{S}$. Thus, we can determine the line t in the truth table that corresponds to y. Then we introduce a clause C that contains for every positive literal ℓ in z the literal $\neg y^\ell_t$ and for each negative literal $\neg\ell$ the literal y^ℓ_t. By adding the clause C for each pair in Φ, we can ensure that the computed circuit implements the relation that is described by Φ.

As in Section 10.2.1, we have to ensure that the synthesized circuit does not introduce any cycles into the circuit upon replacing the original subcircuit $\mathcal{S}$. To prevent cycles, we can proceed similarly as in the QBF encoding. That is, we first compute the set of all forbidden pairs. Then we add for every vertex, connection variables that can be constrained analogous as before. Finally, we use these connection variables in order to prevent that for some forbidden pair (x, y) the vertex representing y depends on x.

An advantage of the SAT-based approach, compared to the QBF-based approach, is that we can use incremental SAT solving for checking different circuit sizes with one encoding.[5] The core idea for adapting the SAT encoding to incremental solving is to introduce an activation variable z_i for each $i \in [|\mathcal{S}|]$. Moreover, we add for each $i \in [|\mathcal{S}|]$ the constraint $\bigwedge_{j\in[m]} z_i \vee \neg o_{ij}$. Thus, if for some $0 \leq \ell \leq |\mathcal{S}|$, we assume $\neg z_i$ for every $\ell < i \leq |\mathcal{S}|$ then the ℓ^{th} gate is the last gate that gives an output. This means that the remaining gates do not have an effect and can thus be ignored. To find a minimum size replacement circuit, we first use no assumptions. If the encoding for size ℓ is satisfiable, we add the variable $\neg z_\ell$ to the assumptions.

10.3 Windowing

For both the QBF and the SAT based approach, the difficulty of finding a new implementation for a subcircuit mainly depends on the size of the subcircuit. But the difficulty also depends on the size of the encompassing circuit. This dependency on the encompassing circuit cannot be avoided, as in order to make use of don't cares we need to consider the entire circuit. In the QBF approach, we add a representation of the entire circuit to the encoding, and for the SAT based approach, we use a representation of the entire circuit in the SAT encoding that is used for determining the relation. Consequently, for large circuits, the task of finding new implementations of subcircuits gets harder.

In order to control the effect of large encompassing circuits, we can apply Algorithm 7 to a *window* (a subcircuit) [MB05] instead of the entire circuit. Minimizing a window preserves the function computed by the window, so we can use the optimized implementation to replace the original window. As a consequence of optimizing a window instead of the entire circuit, we can only make use of don't cares within this window. Nevertheless, the use of windows gives us more control over the hardness of synthesizing new implementations for subcircuits. By choosing windows that are still manageable for our QBF/SAT-based approach, we can minimize circuits that could otherwise not be handled by our method.

Another advantage of using windows is that we can minimize multiple windows simultaneously. In Algorithm 7, we cannot rewrite multiple subcircuits in parallel. This is not possible as a modification of one subcircuit might modify the don't cares of another subcircuit. Thus, while replacing a single subcircuit alone would not change the function computed by the encompassing circuit, replacing multiple subcircuits simultaneously could change it. Unlike the replacement of subcircuits in Algorithm 7, where a circuit can be replaced by circuits computing different functions, windows are always replaced by circuits computing the same function. Consequently, if we select multiple disjoint windows, we can minimize multiple parts of the circuit simultaneously.

A major challenge in combining windowing with Algorithm 7 is that we have to ensure that replacing windows by optimized implementations does not introduce cycles to the

[5]While there are QBF solvers that support incremental solving [LE14] the QBF solvers, we use, do not support incremental solving.

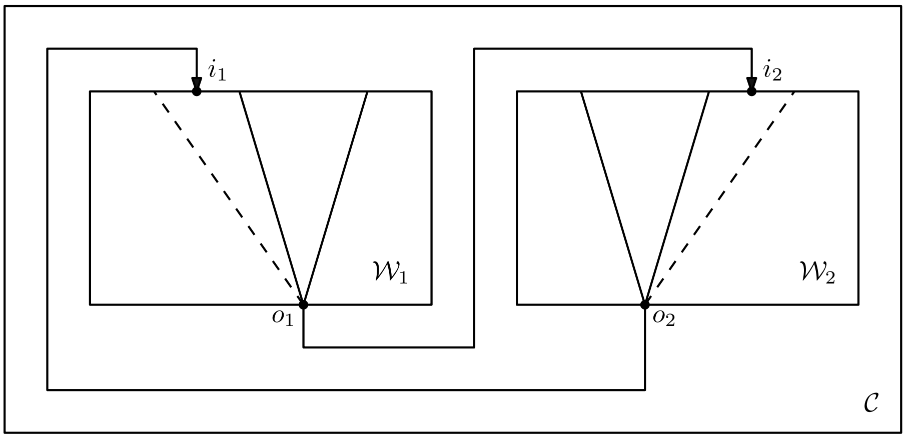


Figure 10.5: We consider two windows $\mathcal{W}_1$ and $\mathcal{W}_2$ in the circuit $\mathcal{C}$. The cone marked with solid lines shows the TFIs of the outputs o_1, respectively o_2. Initially, the output $o_1(o_2)$ does not depend on the input $i_1(i_2)$. Minimizing the windows might change the TFIs of o_1 and o_2. The changed cones are indicated by the dashed lines. The changed cones cause a cycle as i_1 is reachable from itself via o_1, i_2 and o_2.

encompassing circuit. This is possible since in the optimized windows, some primary outputs might contain additional primary inputs in their TFIs.[6] We illustrate this in Figure 10.5.

In order to ensure acyclicity, we introduce what we call the *level constraint* for a window W. The level constraint asserts that inputs of W have smaller levels than any output, formally:

$$\max(\{\mathrm{lv}(x) \mid x \in \mathrm{in}(W)\}) < \min(\{\mathrm{lv}(x) \mid x \in \mathrm{out}(W)\}).$$

Intuitively a cycle can only arise if replacing windows causes a cycle of windows, i.e., in the encompassing circuit there is a path leaving a window and entering the same window later again. The level constraints ensure that for each window all of its outputs occur deeper within the encompassing circuit than all of its inputs. Therefore, even after replacing the window every gate that depends on an output of the window needs to be deeper within the encompassing circuit than all the inputs of the window. Thus, it is not possible to introduce a cycle.

Lemma 10.3. *Let $\mathcal{C}$ be a circuit and $W_1, \ldots, W_n$ be windows of $\mathcal{C}$ that satisfy the level constraint. We require, that whenever two windows contain a common vertex, then this vertex must be a primary input of both windows. That is, for any $i, j \in [n]$ with $i \neq j$ we have $V(W_i) \cap V(W_j) = \mathrm{in}(W_i) \cap \mathrm{in}(W_j)$. Then replacing the windows by any other windows $W'_1, \ldots, W'_n$ does not introduce cycles into $\mathcal{C}$.*

[6]Since the optimized window computes the same function as the original window, assignments of the additional dependencies do not have an effect on the corresponding outputs. Still, such additional structural dependencies can cause cycles in the encompassing circuit.

Algorithm 11 Computing disjoint windows.

```
1:  procedure EXPAND($root, prohibited, size$)
2:      $toConsider \leftarrow$ GETSUCCESSORS($root$) $\setminus prohibited$
3:      $window \leftarrow \emptyset$, $level \leftarrow \infty$
4:      while $toConsider \neq \emptyset$ do
5:          $g \leftarrow$ pop $g$ from $toConsider$ with minimum level
6:          $repair \leftarrow \{x \in TFI(g) \mid \mathrm{lv}(x) \geq level\}$
7:          if $prohibited \cap repair \neq \emptyset \vee |window \cup repair \cup \{g\}| > size$ then
8:              continue
9:          $window \leftarrow window \cup repair \cup \{g\}$
10:         $level \leftarrow \min(\{\mathrm{lv}(x) \mid x \in \mathrm{out}(window)\})$
11:         UPDATE($toConsider, prohibited, repair \cup \{g\}$)
12:     return $window$
```

Proof. For every pair of windows, all common vertices are primary inputs of both windows. Thus, the replacement is well-defined—as there are no overlaps. Suppose towards a contradiction, that replacing the windows causes a cycle. Therefore, after replacing the windows there is some gate g in the circuit that is reachable from itself. Let $P = p_1, \dots, p_m$ be a path from g to itself, i.e, $p_1 = p_m = g$. We know that P cannot be disjoint from all windows $W'_1, \dots, W'_n$, otherwise $\mathcal{C}$ would already contain a cycle. Thus, we can assume that $g \in \mathrm{in}(W_i)$ for some $i \in [n]$. Next, we construct a sequence $Q = q_1, \dots, q_k$ of vertices in the original circuit $\mathcal{C}$ from the path P. To construct Q, we traverse the vertices in P. For this purpose, we distinguish between two cases. If p is an output of a window, we add the corresponding output of the original window to Q. Otherwise, if p is an input of a window or not contained in any window then we add p to Q. This means that any gate in a window that is not an output of the window is not considered for the construction of Q. The construction of Q ensures that for each $i \in [k-1]$ we have $\mathrm{lv}(q_1) < \mathrm{lv}(q_{i+1})$. If q_{i+1} was obtained from an output of a window, then q_i must be an input of this window. As the window satisfies the level constraint, we can conclude that $\mathrm{lv}(q_i) < \mathrm{lv}(q_{i+1})$. Otherwise, if q_{i+1} was not obtained from an output of a window then there is some $l \in [m-1]$ such that q_i and q_{i+1} were obtained from p_l and p_{l+1}, where p_l is a fanin of p_{l+1}. Again this implies that $\mathrm{lv}(q_i) < \mathrm{lv}(q_{i+1})$. As p_1 is an input of a window, we can conclude that $q_1 = q_k$. This yields a contradiction, since this means that $\mathrm{lv}(q_1) < \mathrm{lv}(q_1)$. □

We use Algorithm 11 to compute a window W satisfying the level constraint. The window is chosen such that it has a primary input *root*, $|W| \leq size$ and $(W \setminus \mathrm{in}(W)) \cap prohibited = \emptyset$. In the algorithm the window is represented by a set *window*, which only contains gates and no primary inputs—a proper subcircuit can easily be obtained by adding all inputs of the considered gates. To compute the window the algorithm keeps a set of candidates (toConsider) for the insertion to the window. This set is given by the successors of the current window and *root*. We want to obtain a compactly arranged window, thus we determine a candidate g from *toConsider* with minimum level. Adding the candidate to the window can introduce additional inputs of the window. As these inputs do not necessarily have smaller levels than all current outputs of the window, this can violate the level constraint. To fix this, we compute the set of all vertices (*repair*) in the transitive fanin cone of g, whose level is greater or equal than the current minimal level of an output of the window. Adding g to the current window only preserves the level constraint if *repair* is also added to the window. If *repair* either contains a forbidden gate or the insertion of *repair* would result in a too large window, we cannot insert *repair*. Consequently, it is also not possible to add g to the window. This procedure is repeated until no candidates are remaining.

To apply Algorithm 11, we pick a random primary input x of the circuit $\mathcal{C}$. Then, the first window is obtained by $\textsc{Expand}(x, \emptyset, \mathit{size})$. If the window is too small, a different primary input can be considered as root. To compute an additional window, we either use a primary input previously not considered or an output of a previously computed window as root x. Let *prohibited* denote the set of all gates in previously computed windows. The new window is then computed by $\textsc{Expand}(x, \mathit{prohibited}, \mathit{size})$.

CHAPTER 11

Experiments – Part II

In this chapter, we will give an experimental evaluation of an implementation of Algorithm 7, presented in the previous chapter. We will first compare the exact synthesis of Boolean functions with QBF and SAT. Then we will discuss the minimization of AIGs. Finally, we will consider the minimization of LUT-6 circuits.

We want to point out that this approach is not limited to AIGs, respectively to LUT-6 circuits. For example, we used Algorithm 7 to minimize the size of XAIGs (AIGs that allow XOR-gates) in the IWLS'23 competition. Additionally, our tool can also handle other fanin sizes and it can be adapted for other other types of gates by introducing suitable constraints for the function variables (as discussed in Section 9.2). Nevertheless, here we will only discuss the minimization of AIGs and LUT-6 circuits.

11.1 Implementation

We implemented Algorithm 7 as described in Chapter 10 in a program called eSLIM (exact Synthesis with SLIM). Most parts of eSLIM are implemented in Python and some are implemented in C++.[1] We considered the following QBF backend solvers[2].

QFUN the original version of QFUN [Jan18].[3]

QFUN2 an improved newer version of QFUN.[4]

SMSG a 2QBF solver that is part of the *SAT Modulo Symmetries* framework [KS21].[5]

[1]Available at: `https://github.com/fxreichl/SAT24-eSLIM`

[2]In preliminary experiments we also used the solvers miniQU [Sli22] and QuAbS [Ten16]. As we achieved better results with QFUN, we will not consider miniQU and QuAbS in this thesis.

[3]Available at: `https://sat.inesc-id.pt/~mikolas/sw/qfun/qfun_2018_04_28.tgz`

[4]Available at: `https://github.com/MikolasJanota/qfun`

[5]Available at: `https://github.com/markirch/sat-modulo-symmetries`

As a backend solver for SAT we applied CaDiCaL [Bie+20]. Additionally, eSLIM internally makes use of the following libraries.

- Reading and writing of AIGs is done by the aiger library [Bie07; BHW11].[6]
- For using C++ subroutines in Python, we used pybind11.[7]

11.2 Experimental Setup

Unless stated otherwise, experiments were conducted on a cluster with AMD EPYC 7402 processors at 2.8 GHz running 64-bit Linux. Moreover, we limited the memory usage to 4 GB. For the parallelized minimization we used a memory limit of 4 GB per thread.

To read and write AIGs we used the binary AIGER format [Bie07; BHW11]. For reading and writing non-AIGs we used the BLIF format [Ber92].

In the following, we distinguish between eSLIM using the SAT approach, and eSLIM using the QBF approach. We refer to eSLIM in the aforementioned configuration simply by *sat*. For the latter configuration we further distinguish between the backend solver. We refer to these configurations by *qfun*, *qfun2* and *smsg*.

11.3 Exact Synthesis

In this section, we will compare our implementation of QBF-based exact synthesis with SAT-based exact synthesis to determine the overhead of using QBF solvers. For the comparison, we used the four and six-input functions from a previous experimental evaluation [Haa+20]. Since the five-input functions of the aforementioned evaluation were not available, we used randomly generated five-input functions instead. All, considered functions are given by truth tables. As in [Haa+20], we computed circuits whose gates have two fanins for the four-input functions, circuits whose gates have three fanins for the five-input functions, and circuits whose gates have four fanins for the six-input functions. To get results for the SAT-based approach, we applied the tool Percy [Soe+22]. For each of the three benchmark families, we applied Percy with the optimal configuration according to [Haa+20]. As eSLIM is mainly intended for circuit minimization, some minor modifications of eSLIM were necessary. eSLIM requires that inputs are given as circuits. For this reason, we represented each Boolean function f by a circuit in the BLIF format that consist of single gate representing the function f. We then applied eSLIM to replace this single gate. Moreover, we incremented subcircuit sizes—unlike to circuit minimization, where we start with the original subcircuit size and then decrement the size.

Table 11.1 summarizes the results of the experiments. For each considered configuration, average and median runtimes are shown. Both for the SAT- and the QBF-based approach,

[6]Available at: `https://github.com/arminbiere/aiger`
[7]Available at: `https://github.com/pybind/pybind11`

	sat		qfun		qfun2		smsg	
Instance	mean	median	mean	median	mean	median	mean	median
4-input	0.29	0.03	3.07	0.21	1.66	0.30	1.81	0.28
5-input	3.18	2.54	21.38	10.38	8.70	5.94	12.39	6.25
6-input	0.04	0.01	0.09	0.06	0.21	0.12	0.12	0.07

Table 11.1: Average and median times for SAT and QBF-based exact synthesis.

we only measured the time for synthesizing circuits. The results for the four-input functions are given first, followed by the results for the five- and six-input functions. The table shows that on average it takes less time to synthesize circuits for the six-input functions compared to the four- and five-input functions. This can be explained by the observation that on average fewer gates are needed to represent the six-input functions compared to the other two sets of functions. On average approximately 5 fanin-2 gates are needed to represent the four-input functions, 4.4 fanin-3 gates for the five-input functions and 2.3 fanin-4 gates for the six-input functions. Therefore, on average fewer SAT, respectively QBF, calls are needed for the six-input functions, which explains the shorter runtimes.

The results show that, in general, the SAT-based approach is faster than all QBF configurations. However, eSLIM was not optimized for exact synthesis, and optimizing the encoding to single output functions could reduce the size of the gap. Moreover, the table shows that there are significant differences between different used QBF solvers. In particular, the comparison between the results by QFUN and the newer QFUN2 for the four- and five-input functions suggest that further progress in the development of QBF solvers, or tuning to this application, might close the gap between the SAT- and the QBF-based approach.

11.4 AIG Minimization

In this section, we will analyze the performance of eSLIM for minimizing AIGs.

We evaluated our tool on the instances from the IWLS'23[8] and the IWLS'24[9] programming contests. Both benchmark sets consist of 100 Boolean functions, given as truth tables. The goal is to construct an *And-Inverter Graph* (AIG) that computes the specified Boolean function with as few gates as possible.

Since the instances are given as truth tables, and our tool requires that specifications are given as circuits, we preprocessed the instances using ABC [BM10]. As a naive transformation of truth tables to circuits by using ABC in general results in relatively large circuits, we used the ABC command DEEPSYN to reduce the size of the initial

[8]https://github.com/alanminko/iwls2023-ls-contest
[9]https://github.com/alanminko/iwls2024-ls-contest

circuit. We used DEEPSYN with a timeout of one hour. DEEPSYN is widely regarded as on one of the most effective optimization strategies for computing compact AIGs [CMM23]. Additionally, unlike many other optimization strategies in ABC, DEEPSYN allows anytime optimization. This made it possible to use a fixed time for each instance. For preprocessing we used a workstation with an Intel i7-4790 processor at 3.6 GHz running 64-bit Linux. We processed up to four instances in parallel. While we did not use memory limits for individual instances, in total 16 GB of memory were available.

In our evaluation setup, we considered eSLIM both in the QBF-based and the SAT-based configuration. As in the last section, we used the QBF solvers QFUN, QFUN2 and SMSG for the QBF-based approach. The solver SMSG only supports 2QBF—i.e., QBF which only have a single quantifier alternation. As discussed in Chapter 10, we need another block of existentially quantified variables in case there are forbidden pairs for a subcircuit. Consequently, we could not use SMSG for subcircuits that have forbidden pairs. For this reason we used QFUN2 in the SMSG configuration for subcircuits with forbidden pairs. Since both benchmark sets contain only few instances that are sufficiently large for a reasonable application of windowing, we did not apply it here.

We compared eSLIM with the DEEPSYN procedure from ABC. We did not compare our tool with other entries from the IWLS 2023 competition. Neither the tool used by the Google DeepMind team (ranked first) nor the tool used by the EPFL team (ranked third) are publicly available, so we could not run their tools ourselves. Moreover, we do not know the computational resources that were used by the two teams for the generation of their submission to the competition. Thus, we think that a comparison with the instances from their submissions would not be fair.[10] We also did not consider the transduction method [Miy24], which is implemented as part of ABC. In preliminary experiments the transduction method did not report results for some instances within a week. Consequently, to give a fair comparison between eSLIM and the transduction method very long run times would have been necessary. The required experiments would have been beyond the capabilities of the computational infrastructure available to us.

In our experiments, we alternated between 27-minute runs of eSLIM and 3-minute runs of DEEPSYN for inprocessing. This was repeated eight times. For comparison, DEEPSYN was run for four hours.[11] We set the initial bound for the subcircuit size to 6 for all considered configurations of eSLIM. As both DEEPSYN and eSLIM internally use some kind of randomization, we performed five independent runs for each configuration.

For both benchmark sets and all configurations, the application of DEEPSYN for inprocessing is responsible for approximately 20% of the total reduction. We visualized the reduction of gates over time for eSLIM in Figure 11.1. We do not compare with the four-hour run of DEEPSYN as we do not have intermediate circuit sizes for these experiments. The figure shows that both our method and the inprocessing step show

[10]At the presentation of the IWLS 23 competition results it was mentioned that the DeepMind team used much longer runtimes—weeks instead of hours.

[11]In general, all tools benefited from longer runtimes. Nevertheless, we limited the runs to roughly four hours due to constraints on the available computational infrastructure.

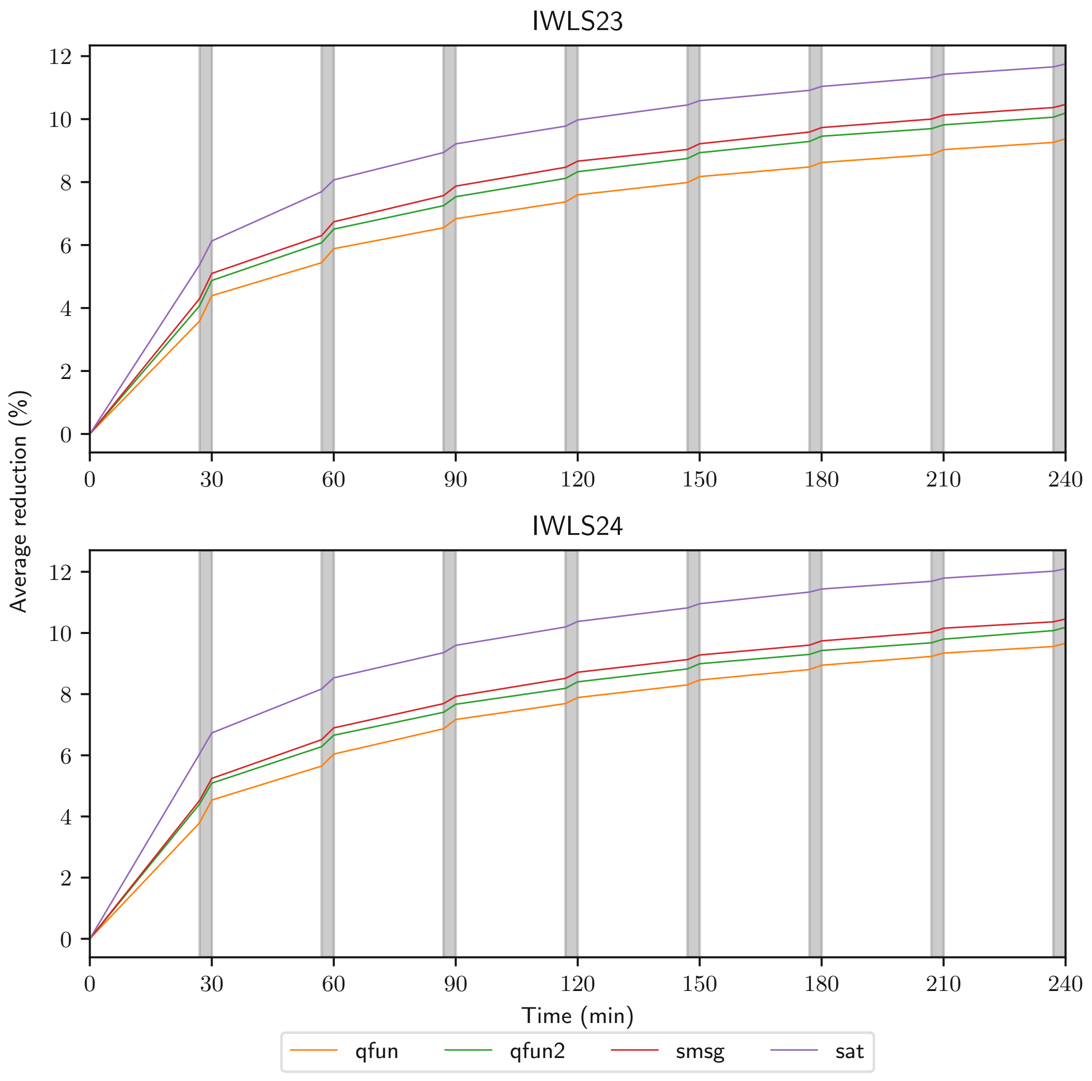


Figure 11.1: The plot shows the average reduction of gates (%) over time. The gray marked areas visualize the applications of ABC for inprocessing.

diminishing returns over time. Nevertheless, we can observe that until the end of the runs, further reductions could be achieved.

Next, for both benchmark sets, we grouped the instances into four subsets of 25 based on the initial number of gates. We then determined, for each run and each group of instances the average size reductions compared to the preprocessed circuits. Furthermore, we computed averages and standard deviations among the individual runs of the configurations. Results for the IWLS'23 instances are given in Table 11.2, and results for the IWLS'24 instances are given in Table 11.3. Visualizations of the results for both benchmark sets are given in Figure 11.2.

	deepsyn		qfun		qfun2		smsg		sat	
#Gates	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
10 – 39	0.59	0.4	2.97	0.42	3.16	0.22	3.07	0.21	2.93	0.28
40 – 100	0.68	0.18	8.11	0.27	8.24	0.2	8.32	0.35	8.7	0.26
131 – 492	9.32	0.44	15.69	0.8	17.56	0.55	18.12	0.85	19.76	0.59
505 – 7839	12.64	0.47	10.73	0.25	11.82	0.58	12.36	0.27	15.61	0.52
Overall	5.82	0.22	9.37	0.21	10.2	0.18	10.47	0.25	11.75	0.17

Table 11.2: Average reduction (%) of gates compared to the preprocessed IWLS'23 instances, by configuration and initial size, and standard deviations of the average reductions per configuration.

	deepsyn		qfun		qfun2		smsg		sat	
#Gates	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
10 – 39	0.09	0.07	1.97	0.17	1.82	0.13	1.9	0.07	1.92	0.08
40 – 119	0.78	0.3	11.87	0.33	12.3	0.69	12.44	0.43	12.65	0.37
126 – 403	4.89	0.67	13.71	0.79	14.28	0.25	14.69	0.18	17.27	0.57
417 – 7569	11.51	0.39	11.11	0.32	12.34	0.67	12.79	0.38	16.57	0.35
Overall	4.32	0.07	9.66	0.32	10.18	0.22	10.46	0.19	12.1	0.19

Table 11.3: Average reduction (%) of gates compared to the preprocessed IWLS'24 instances, by configuration and initial size, and standard deviations of the average reductions per configuration.

For both the IWLS'23 and the IWLS'24 instances, we obtained similar results. First, eSLIM using the SAT approach as well as eSLIM using the QBF approach could outperform deepsyn. This indicates that the combination of deepsyn for preprocessing and inprocessing with eSLIM is better than just applying deepsyn alone. Moreover, the SAT-based strategy outperformed the QBF-based strategy. Analyzing the experiments showed that in general the SAT-based approach was able to handle individual subcircuits faster. Thus, the SAT-based approach could process more subcircuits compared with the QBF-based approach.

Further, we can also see that for both benchmark sets there is a difference between the different QBF solvers. Overall, smsg was able to achieve more reductions than both QFUN and QFUN2, and QFUN2 could outperform QFUN. As for exact synthesis, these results suggest that further progress in the development of QBF solvers might close the gap between the QBF and the SAT-based approach.

Above, we discussed average reductions among the individual runs of the considered configurations. Additionally, we determined for each configuration the best implementation found by one of the runs. We computed the average reductions per instance class for these implementations. Results for the IWLS'23 instances are given in Table 11.4, and results for the IWLS'24 instances are given in Table 11.5. These results indicate that for all considered configurations, it may be advantageous to consider multiple runs.

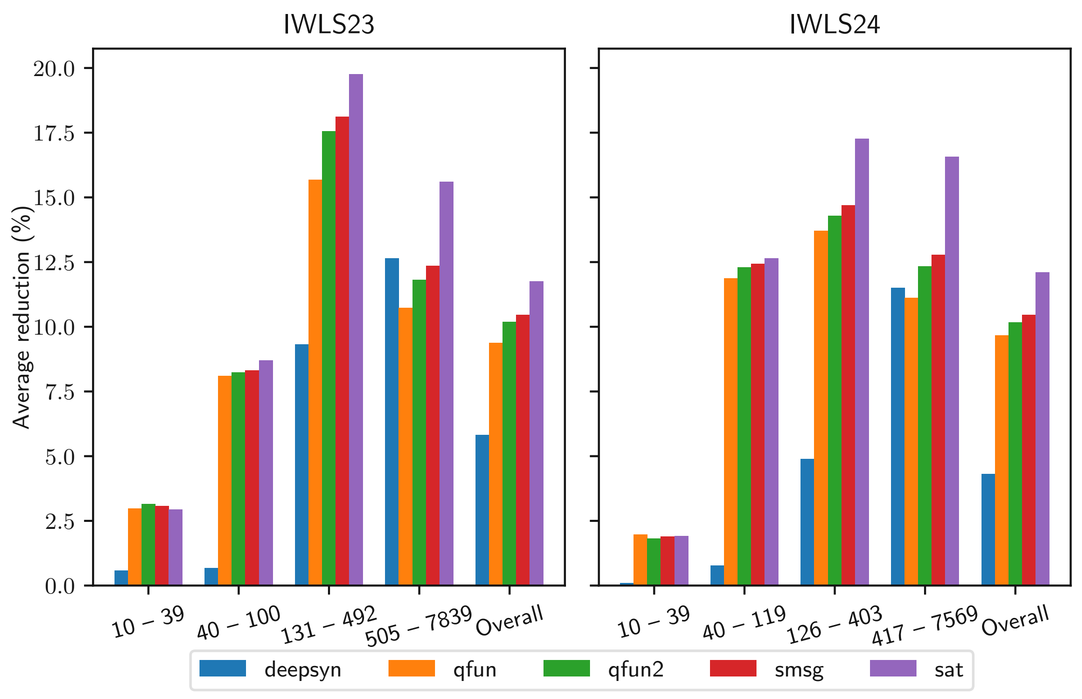


Figure 11.2: Bar plots showing the average reductions of gates per instance class and minimization approach.

#Gates	deepsyn	qfun	qfun2	smsg	sat
10 – 39	1.65	3.6	3.86	4.04	3.38
40 – 100	1.63	9.68	9.56	9.56	10.46
131 – 492	12.73	18.59	20.3	20.92	22.46
505 – 7839	14.99	12.3	13.67	14.11	18.33
Overall	7.75	11.04	11.85	12.16	13.66

Table 11.4: For each instance and each configuration the best implementation among the individual runs is selected. The table reports the average reduction (%) of these best implementations per instance class for the IWLS'23 instances.

As our method makes use of a randomized subcircuit selection, different runs result in different sequences of replaced subcircuits. Consequently, one run might get stuck in a local minimum (which is difficult to escape from).

Parameters In the above experiments, we used parameters for eSLIM that seemed to be good choices according to preliminary experiments. Here we want to highlight subcircuit sizes. As mentioned above, we always used an upper bound of 6 for the subcircuit sizes. In practice, this bound yielded the best results for the IWLS instances. For smaller bounds the task of finding replacement circuits gets easier, fewer reductions

#Gates	deepsyn	qfun	qfun2	smsg	sat
10 – 39	0.28	2.14	2.11	2.14	1.98
40 – 119	1.55	14.11	14.45	13.7	14.3
126 – 403	7.79	16.63	16.99	17.1	20.25
417 – 7569	14.03	12.67	14.22	14.84	18.12
Overall	5.91	11.39	11.94	11.95	13.66

Table 11.5: For each instance and each configuration the best implementation among the individual runs is selected. The table reports the average reduction (%) of these best implementations per instance class for the IWLS'24 instances.

are possible. Conversely, for larger bounds, there are more potential reductions but the task of computing replacement circuits gets harder, so that fewer subcircuits can be considered. A bound of 6 seems to be a good tradeoff between the hardness of the replacement and the potential for finding improvements.

In addition to the experiments presented above, eSLIM also showed a good performance in recent IWLS programming contests. A preliminary version eSLIM achieved the third place in the IWLS'22[12] programming contest, and it achieved the second place in all three tracks of the IWSL'23[13] programming contest.

11.5 LUT-6 Minimization

In order to evaluate eSLIM for circuits whose gates have more than two fanins, we considered the *EPFL Combinational Benchmark Suite* [AGM15]. This benchmark set consists of twenty circuits.[14] The goal is to find small 6-input lookup table (LUT-6) implementations of the specifications. Gates have six fanins, and arbitrary functions are allowed at each gate. In addition to a specification given as circuit with binary gates, the benchmark suite also provides the best known LUT-6 realizations so far, which are continuously updated. We took the best known realizations as of 2022 (commit *42c1f31*) as initial specifications in our experiments.[15]

We ran our reduction tool for 12 hours both with the SAT and the QBF configuration. Unlike in the previous experiments we only considered the original version of the QBF solver QFUN. We did not consider the other QBF solvers as preliminary experiments showed that there are no significant differences. As the EPFL benchmark set contains sufficiently large circuits, we also applied windowing. Every hour, we applied the ABC

[12]`https://github.com/alanminko/iwls2022-ls-contest`

[13]`https://github.com/alanminko/iwls2023-ls-contest`

[14]We did not consider the MtM instances as the EPFL repository does only contain the original specifications but not the current best implementations for these circuits.

[15]We did not consider the best results of 2023 because half of them were provided by us. As it is difficult for our tool to further reduce these circuits, we think the circuits from 2022 are better suited for the evaluation.

Instance	Initial	QBF	SAT
Lookahead XY router	**19**	19	19
int to float converter	20	**18**	19
Alu control unit	**25**	25	25
Coding-cavlc	54	**49**	53
Priority encoder	94	93	**92**
Adder	**129**	129	129
I2c controller	182	179	**177**
Decoder	**264**	264	264
Round-robin arbiter	273	272	**267**
Max	**511**	511	511
Barrel shifter	**512**	512	512

Table 11.6: Results for EPFL instances with fewer than 1000 gates. The table gives for each instance the number of LUT-6 gates of the initial circuit and of the improved circuits per configuration. For instances that could not be improved the initial sizes are marked in boldface. For the instances that could be improved the best results are marked in boldface.

command &MFS as an inprocessing step. &MFS allows to directly optimize a LUT-6 circuit. We recombined the windows for the inprocessing step and computed new windows afterwards.

In addition to a bound for the size for the subcircuits, we also used a limit of 10 on the number of inputs of the subcircuits considered for resynthesis. Preliminary tests showed that such a limit is required to reliably generate the Boolean relation for the SAT-based approach within time and memory limits. Additionally, we always set the initial bound for the subcircuit size to 4—we had to use lower bounds compared to the AIG experiments, as gates have six fanins. For the experiments with windowing, we considered bounds of 500 and 1000 gates for the construction of the windows. First we minimized single windows and second up to 8 windows concurrently. We only applied windowing for instances with at least 1000 gates.

Results for instances with at most 1000 gates are given in Table 11.6 and results for instances with at least 1000 gates are given in Table 11.7.

Since the initial circuits are already highly optimized by state-of-the-art methods, the relative improvements for the EPFL instances were small compared to the IWLS instances, and it is difficult to draw any definitive conclusions about the superiority of any configuration from these results. Nevertheless, the results suggest that parallel optimization was able to beat single-threaded optimization. Moreover, our tool could improve on the best implementation for the majority of instances.

The aim of the experiments discussed above, was to give a systematic comparison between different configurations of eSLIM. In addition, we used a preliminary version of eSLIM for generating our 2023 submission to the EPFL benchmark suite. Moreover,

		No Windowing		Single Window		Up to 8 Windows	
Instance	Initial	QBF	SAT	QBF	SAT	QBF	SAT
Sine	1114	1095	1085	1076	1069	1057	**1036**
Voter	1217	1217	1179	1184	**1166**	1177	1172
Memory controller	1735	**1722**	1727	1731	1730	1724	1731
Square-root	2994	2994	2994	2991	2985	2992	**2980**
Square	3018	3014	2997	2992	2994	**2942**	2943
Divisor	3096	3096	3096	3096	**3095**	3096	**3095**
Multiplier	4360	4358	4346	4346	4346	4326	**4317**
Log2	6133	6132	6109	6127	6129	6078	**6063**
Hypotenuse	39452	39452	39452	39230	39251	**38459**	38781

Table 11.7: Results for EPFL instances with more than 1000 gates. The table gives for each instance the number of LUT-6 gates of the initial circuit and of the improved circuits per configuration. For the configurations with windowing, we report for each instance the size of the smallest circuit among the two considered window sizes. The best results are marked in boldface.

for preliminary experiments we performed additional runs of eSLIM. In Table 11.8 we give the best known results for the last years. This also include our 2023 submission. Additionally, we give for every instance the overall best solution, which we could compute with eSLIM. We did not use the best known results from 2023 as inputs. Therefore, for some instances our best solution is larger as the current best one.

Instance	2015	2016	2017	2018	2019	2020	2021	2022	2023	Ours
router	26	26	25	25	23	23	23	23	19	18
ctrl	28	28	26	26	26	26	26	25	25	25
int2float	34	34	28	26	26	24	24	24	**19**	18
cavlc	107	107	101	72	68	68	68	67	**50**	49
priority	118	118	110	108	102	102	100	100	**93**	92
adder	201	192	192	192	192	192	191	134	129	129
i2c	215	215	212	210	199	199	199	197	**177**	176
dec	272	272	270	264	264	264	264	264	264	264
arbiter	429	429	409	403	328	313	306	299	268	267
bar	512	512	512	512	512	512	512	512	512	512
max	532	532	523	523	522	522	522	512	511	511
sin	1347	1347	1229	1228	1227	1221	1205	1198	**1053**	1036
voter	1515	1515	1301	1297	1297	1293	1281	1254	**1180**	1166
mem_ctrl	2125	2125	2080	2077	2001	2001	1979	1946	1708	1719
sqrt	3286	3286	3077	3076	3074	3031	3027	2996	**2983**	2980
square	3798	3307	3244	3242	3239	3239	3231	3231	**2959**	2942
div	3813	3813	3268	3268	3267	3267	3248	3104	3090	3092
multiplier	5681	5192	4923	4923	4919	4915	4898	4485	**4330**	4317
log2	7344	7344	6574	6570	6567	6557	6513	6447	**6076**	6063
hyp	44635	44635	40357	40338	40322	40322	39826	39556	36836	38459

Table 11.8: Best known implementations of the EPFL instances by year and best implementations computed by eSLIM.[16] For each year the corresponding circuits are available as part of the EPFL benchmark suite.[17] The reported numbers of LUT-6 gates were computed with the ABC command `&get -m; &ps`.[18] The implementations submitted by us are marked in boldface.

[16]For most years, new results were published multiple times. We used the release version for each year.

[17]`https://github.com/lsils/benchmarks`

[18]Only for the 2015 and 2016 entries for the Hypotenuse (hyp) instance, numbers reported in the EPFL benchmark suite had to be used, as the circuit is not available for these years. We did not use the numbers reported in the EPFL benchmark suite for the remaining entries of the table, since these numbers were generated by slightly differing counting methods (for example the reported size for the 2022 version of the *memory controller* instance seems to be computed by the ABC command `sweep -s; ps` while the reported number for the 2023 version seems to be based on `&get -m; &ps`). As a consequence of the varying counting methods, some entries in the table slightly differ from the corresponding numbers reported in the EPFL benchmark suite.

CHAPTER 12

Conclusions – Part II

In this part of the thesis, we first introduced a QBF encoding for the problem of finding minimum size circuits, computing a given Boolean function. We then extended this QBF encoding for computing minimum size replacements for subcircuits of a given circuit. Our encoding exploits using the full implementational flexibility for replacing multi-output subcircuits by making use of don't cares. Don't cares do not need to be computed explicitly, as the encoding implicitly takes don't cares into account. While the computed subcircuits are acyclic, we saw that we need to pay attention in order to not introduce any cycles in the encompassing circuit upon replacement.

As an alternative, we also considered a SAT-based approach. In the SAT-based approach the computation of a replacement circuit is divided into two subtasks. First, a Boolean relation representing the original subcircuit is computed by means of incremental SAT solving. Second, a circuit that is compatible with this relation is determined. This circuit can then be used for replacing the original subcircuit.

Both approaches do not only depend on the local subcircuit but also on the encompassing circuit. This is necessary in order to capture don't cares. Consequently, the task of replacing subcircuits does not only get harder for larger subcircuits but also for larger encompassing circuits. To still handle large circuits, we can minimize a window instead of the entire circuit. Windowing does not only allow handling larger circuits, but it also allows optimizing multiple parts of a circuit simultaneously.

In the experimental evaluation, we saw that our tool eSLIM that implements both the QBF and the SAT-based approach, yielded promising results for the instances from the IWLS23 and IWLS24 programming contests. A combination of the state-of-the-art circuit optimization method DEEPSYN provided by ABC for preprocessing and inprocessing with eSLIM allowed computing smaller circuits as DEEPSYN alone. Moreover, eSLIM was able to further reduce the size of most circuits from the EPFL benchmark suite.

Future Work

There are four directions in which we would like to continue our work.

Implementation as Part of ABC We will integrate the proposed circuit minimization method into ABC. This would make it much easier to use our minimization method in practice. In particular, this would also simplify the pre- and inprocessing steps.

CEGAR-based Approach It also might be worthwhile to combine ideas from QBF solving and the SAT based approach. Instead of strictly separating the task of computing a Boolean relation and finding a circuit that implements this relation, we could try to combine these tasks. One could try to synthesize a replacement circuit. If the substitution of the synthesized circuit alters the function, one could extract a reason for the failure of the substitution. This reason could then be used as an additional constraint for subsequent attempts of finding replacement circuits.

Computing Relations by Simulation In the current implementation of the SAT-based approach, Boolean relations are computed by means of incremental SAT calls. In general, the time needed for synthesizing the relation outweighs the time needed to compute it. Still, if the performance of computing relations could be improved, individual subcircuits could be analyzed faster. This could lead to more improvements within a given time limit. An alternative approach for obtaining a Boolean relation would be to use circuit simulation [Lee+22] in order to capture the necessary behavior of the subcircuit.[1] While this approach would not be applicable to circuits, respectively windows, with many inputs it could improve the performance of the relation generation in case there are not too many inputs.

Allowing Additional Inputs for Synthesizing Subcircuits The proposed circuit minimization approach, tries to reduce a circuit by replacing subcircuits by other circuits with the same inputs. In contrast, *circuit resubstitution* [Mis+11] replaces subcircuits by circuits using different inputs. This can be understood as reusing some functionality that is already available in the circuit. The proposed minimization approach cannot make use of such shared functionalities in the circuit, and it has to use ABC for this purpose. To directly make use of such shared functionality, we could allow to make use of additional gates as inputs for synthesizing new implementations of subcircuits. In a prototype, we randomly selected for each considered subcircuit a small number of additional gates. We then added these gates as additional inputs for the synthesis of a new implementation of the subcircuit. In preliminary experiments with the prototype we saw that this improved the performance of the QBF-based approach for the IWLS'23 instances. With the described modifications the QBF-based approach on average could reduce (slightly) more gates than the standard SAT-based approach. This motivates further work, to for instance find a better selection strategy for the additional inputs.

[1] This was suggested to us by Alan Mishchenko.

Postface

In this thesis, we covered two main topics. First, we presented a new decision procedure for DQBF. Second, we discussed a procedure for minimizing Boolean Circuits. Both parts extend upon our previously published work. The first part, which is concerned with deciding DQBF, extends our SAT 2021 [RSS21] and our SAT 2022 [RS22] papers. The second part is based on our AAAI 2023 paper [RSS23] and our SAT 2024 paper [RSS24], which was recognized as a runner-up for the best student paper.

Both parts resulted in tools that performed well in different competitions. Our decision procedure for DQBF was implemented in the DQBF solver Pedant. Pedant won a gold medal at the FLoC Olympic Games 2022 for achieving the first place in the DQBF track of the QBF evaluation. Additionally, Pedant ranked first in the DQBF track of the QBF-Gallery 2023. Pedant was not only able to solve more instances at these evaluations than the winner of the DQBF track of the previous QBF-Evaluation (DQBDD), but it also allows computing models for satisfiable DQBF, which can be of relevance in practice. Our circuit minimization procedure was first implemented in a prototype Ciops. This tool was able to achieve the third place in the IWLS 2022 competition. Improving and extending this prototype resulted in the tool eSLIM. eSLIM ranked second in the IWLS 2023 competition behind the Google Deepmind team. Moreover, eSLIM could further improve some of the best known implementations from LUT-6 circuits in the EPFL benchmark suite.

For both the decision procedure for DQBF as well as for the circuit minimization approach, we pointed out several possible directions for extending our work. In addition to the considered extensions, it would also be interesting to bring the two main topics of this thesis—DQBF solving and circuit minimization—together. As discussed earlier our solver Pedant generates AIG certificates for satisfiable DQBF. This means that models computed by Pedant are circuits. In this thesis, we were not in particular interested in how these circuits actually look like, as long as they certify the satisfiability of the underlying DQBF. For applications of DQBF, like partial equivalence checking (PEC) in integrated circuit engineering change order (ECO) an arbitrary model would not necessarily be sufficient. Instead, one might be interested in small models. For this purpose, one could start with the circuits generated by Pedant and then apply ABC and eSLIM for reducing the size of this circuit. But it would also be interesting to already try to compute small models within Pedant. One possible approach in this direction would be to reduce the size of definitions. As briefly indicated earlier, definitions can attain an arbitrary value for falsifying assignments of the matrix. This means that falsifying assignments of the matrix can be considered as don't cares for the definitions. Using these don't cares for reducing the size of definitions, could on the one hand help PEDANT as candidates would become smaller and on the other hand it would make the constructed certificates smaller, which could be helpful in applications.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AGM15] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "The EPFL combinational benchmark suite". In: *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*. 2015. URL: https://github.com/lsils/benchmarks.

[Aks+18] S. Akshay, Supratik Chakraborty, Shubham Goel, Sumith Kulal, and Shetal Shah. "What's Hard About Boolean Functional Synthesis?" In: *CAV (1)*. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 251–269. DOI: 10.1007/978-3-319-96145-3_14.

[Ama+18] Luca Gaetano Amarù, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Janet Olson, Robert K. Brayton, and Giovanni De Micheli. "Improvements to boolean resynthesis". In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. Ed. by Jan Madsen and Ayse K. Coskun. IEEE, 2018, pp. 755–760. DOI: 10.23919/DATE.2018.8342108.

[AS09] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*. Ed. by Craig Boutilier. 2009, pp. 399–404.

[Bal+16] Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. "SAT Race 2015". In: *Artif. Intell.* 241 (2016), pp. 45–65. DOI: 10.1016/J.ARTINT.2016.08.007.

[BB20] Olaf Beyersdorff and Joshua Blinkhorn. "Lower Bound Techniques for QBF Expansion". In: *Theoretical Computer Science* 64.3 (2020), pp. 400–421. DOI: 10.1007/S00224-019-09940-0.

[BBP20] Olaf Beyersdorff, Joshua Blinkhorn, and Tomás Peitl. "Strong (D)QBF Dependency Schemes via Tautology-Free Resolution Paths". In: *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*. Ed. by Luca Pulina and Martina Seidl. Vol. 12178. Lecture Notes in Computer Science. Springer, 2020, pp. 394–411. DOI: 10.1007/978-3-030-51825-7_28.

[BCJ14a] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong Roland Jiang. "Henkin quantifiers and Boolean formulae: A certification perspective of DQBF". In: *Theoretical Computer Science* 523 (2014), pp. 86–100. ISSN: 0304-3975. DOI: 10.1016/J.TCS.2013.12.020.

[BCJ14b] Olaf Beyersdorff, Leroy Chew, and Mikolas Janota. "On Unification of QBF Resolution-Based Calculi". In: *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II.* Ed. by Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik. Vol. 8635. Lecture Notes in Computer Science. Springer, 2014, pp. 81–93. DOI: 10.1007/978-3-662-44465-8_8.

[Ber92] University of California Berkeley. *Berkeley Logic Interchange Format (BLIF).* 1992. URL: https://course.ece.cmu.edu/~ee760/760docs/blif.pdf (visited on 2024-06-05).

[Bey+19] Olaf Beyersdorff, Joshua Blinkhorn, Leroy Chew, Renate A. Schmidt, and Martin Suda. "Reinterpreting Dependency Schemes: Soundness Meets Incompleteness in DQBF". In: *J. Autom. Reason.* 63.3 (2019), pp. 597–623. DOI: 10.1007/S10817-018-9482-4.

[BF22] Armin Biere and Mathias Fleury. "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022". In: *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions.* Ed. by Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2022-1. Department of Computer Science Series of Publications B. University of Helsinki, 2022, pp. 10–11.

[BHS90] Robert K. Brayton, Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli. "Multilevel logic synthesis". In: *Proc. IEEE* 78.2 (1990), pp. 264–300. DOI: 10.1109/5.52213.

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond.* Tech. rep. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, July 2011. DOI: 10.35011/fmvtr.2011-2.

[Bie+20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions.* Ed. by Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[Bie+21] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability - Second Edition*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. ISBN: 978-1-64368-160-3. DOI: 10.3233/FAIA336.

[Bie07] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. 07/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, Oct. 2007. DOI: 10.35011/fmvtr.2007-1.

[BK04] Yuri Boykov and Vladimir Kolmogorov. "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 26.9 (2004), pp. 1124–1137. DOI: 10.1109/TPAMI.2004.60.

[BKS14] Roderick Bloem, Robert Könighofer, and Martina Seidl. "SAT-Based Synthesis Methods for Safety Specs". In: *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014*. Ed. by Kenneth L. McMillan and Xavier Rival. Vol. 8318. Lecture Notes in Computer Science. Springer, 2014, pp. 1–20. DOI: 10.1007/978-3-642-54013-4_1.

[BM10] Robert K. Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *CAV*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[Bry+87] Randal E. Bryant, Derek L. Beatty, Karl S. Brace, Kyeongsoon Cho, and Thomas J. Sheffler. "COSMOS: A Compiled Simulator for MOS Circuits". In: *Proceedings of the 24th ACM/IEEE Design Automation Conference. Miami Beach, FL, USA, June 28 - July 1, 1987*. Ed. by A. O'Neill and D. Thomas. IEEE, 1987, pp. 9–16. DOI: 10.1145/37888.37890.

[Che+22] Fa-Hsun Chen, Shen-Chang Huang, Yu-Cheng Lu, and Tony Tan. "Reducing NEXP-complete problems to DQBF". In: *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*. Ed. by Alberto Griggio and Neha Rungta. IEEE, 2022, pp. 199–204. DOI: 10.34727/2022/ISBN.978-3-85448-053-2_26.

[CJM12] Huan Chen, Mikolás Janota, and João Marques-Silva. "QBF-based boolean function bi-decomposition". In: *DATE*. Ed. by Wolfgang Rosenstiel and Lothar Thiele. IEEE, 2012, pp. 816–819. DOI: 10.1109/DATE.2012.6176606.

[CMM23] Andrea Costamagna, Alan Mishchenko, and Giovanni De Micheli. "The Combinational-Complexity Game For Symmetric Functions". In: *Proceedings of the 32nd International Workshop on Logic & Synthesis (IWLS)*. 2023. URL: https://si2.epfl.ch/~demichel/publications/archive/2023/AC.pdf (visited on 2024-06-05).

[Coo71] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[CR79] Stephen A. Cook and Robert A. Reckhow. “The Relative Efficiency of Propositional Proof Systems”. In: *Journal of Symbolic Logic* 44.1 (1979), pp. 36–50. DOI: 10.2307/2273702.

[Cur+23] Ryan R. Curtin, Marcus Edel, Omar Shrit, Shubham Agrawal, Suryoday Basak, James J. Balamuta, Ryan Birmingham, Kartik Dutt, Dirk Eddelbuettel, Rishabh Garg, Shikhar Jaiswal, Aakash Kaushik, Sangyeon Kim, Anjishnu Mukherjee, Nanubala Gnana Sai, Nippun Sharma, Yashwant Singh Parihar, Roshan Swain, and Conrad Sanderson. “mlpack 4: a fast, header-only C++ machine learning library”. In: *Journal of Open Source Software* 8.82 (2023), p. 5026. DOI: 10.21105/joss.05026.

[Dav69] Edward S. Davidson. “An Algorithm for NAND Decomposition Under Network Constraints”. In: *IEEE Trans. Computers* 18.12 (1969), pp. 1098–1109. DOI: 10.1109/T-C.1969.222593.

[DeM94] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw Hill, 1994.

[DH00] Pedro M. Domingos and Geoff Hulten. “Mining high-speed data streams”. In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. Ed. by Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa. ACM, 2000, pp. 71–80. DOI: 10.1145/347090.347107.

[Die00] Reinhard Diestel. *Graph Theory, 2nd Edition*. Vol. 173. Graduate texts in mathematics. Springer, 2000.

[Eén07] Niklas Eén. “Practical SAT – A tutorial on applied satisfiability solving”. In: *Invited talk at FMCAD 2007*. Presented by Alan Mishchenko. 2007. URL: http://minisat.se/Papers.html (visited on 2024-06-05).

[ES03] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *SAT 2003*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[ET75] Shimon Even and Robert Endre Tarjan. “Network Flow and Testing Graph Connectivity”. In: *SIAM J. Comput.* 4.4 (1975), pp. 507–518. DOI: 10.1137/0204043.

[Fay+17] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. “Encodings of Bounded Synthesis”. In: *TACAS (1)*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 354–370. DOI: 10.1007/978-3-662-54577-5_20.

[Fin+16] Magnus Gausdal Find, Alexander Golovnev, Edward A. Hirsch, and Alexander S. Kulikov. “A Better-Than-3n Lower Bound for the Circuit Complexity of an Explicit Function”. In: *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. Ed. by Irit Dinur. IEEE Computer Society, 2016, pp. 89–98. DOI: 10.1109/FOCS.2016.19.

[FKB12] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. “A DPLL Algorithm for Solving DQBF”. presented at *Workshop on Pragmatics of SAT (POS)*. 2012. URL: http://fmv.jku.at/papers/FroehlichKovasznaiBiere-POS12.pdf (visited on 2024-06-05).

[FLS17] Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. “SAT-Based Local Improvement for Finding Tree Decompositions of Small Width”. In: *SAT*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 401–411. DOI: 10.1007/978-3-319-66263-3_25.

[Frö+14] Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. “iDQ: Instantiation-Based DQBF Solving”. In: *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*. Ed. by Daniel Le Berre. Vol. 27. EPiC Series in Computing. EasyChair, 2014, pp. 103–116. DOI: 10.29007/1S5K.

[Fro+21] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. “SAT Competition 2020”. In: *Artif. Intell.* 301 (2021), p. 103572. DOI: 10.1016/J.ARTINT.2021.103572.

[FT14] Bernd Finkbeiner and Leander Tentrup. “Fast DQBF Refutation”. In: *Theory and Applications of Satisfiability Testing - SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, 2014, pp. 243–251. DOI: 10.1007/978-3-319-09284-3_19.

[FT23] Long-Hin Fung and Tony Tan. “On the Complexity of k-DQBF”. In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 10:1–10:15. DOI: 10.4230/LIPICS.SAT.2023.10.

[Fuj+13] Masahiro Fujita, Satoshi Jo, Shohei Ono, and Takeshi Matsumoto. "Partial synthesis through sampling with and without specification". In: *ICCAD*. Ed. by Jörg Henkel. IEEE, 2013, pp. 787–794. DOI: 10.1109/ICCAD.2013.6691203.

[Fuj+20] Masahiro Fujita, Yusuke Kimura, Xingming Le, Yukio Miyasaka, and Amir Masoud Gharehbaghi. "Synthesis and Optimization of Multiple Portions of Circuits for ECO based on Set-Covering and QBF Formulations". In: *DATE*. IEEE, 2020, pp. 744–749. DOI: 10.23919/DATE48585.2020.9116459.

[Fuj15] Masahiro Fujita. "Toward Unification of Synthesis and Verification in Topologically Constrained Logic Design". In: *Proc. IEEE* 103.11 (2015), pp. 2052–2060. DOI: 10.1109/JPROC.2015.2476472.

[Gai21] Dale Gai. *TSMC Price Hike Indicates Capacity Tightness to Persist in 2022, May Hit Smartphone Shipments*. 2021. URL: https://www.counterpointresearch.com/insights/tsmc-price-hike/ (visited on 2024-06-05).

[Gan+20] Robert Ganian, Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. "Fixed-Parameter Tractability of Dependency QBF with Structural Parameters". In: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020*. Ed. by Diego Calvanese, Esra Erdem, and Michael Thielscher. 2020, pp. 392–402. DOI: 10.24963/KR.2020/40.

[Gel12] Allen Van Gelder. "Contributions to the Theory of Practical Quantified Boolean Formula Solving". In: *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*. Ed. by Michela Milano. Vol. 7514. Lecture Notes in Computer Science. Springer, 2012, pp. 647–663. DOI: 10.1007/978-3-642-33558-7_47.

[Git+13a] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. "Equivalence Checking for Partial Implementations Revisited". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Warnemünde, Germany, March 12-14, 2013*. Ed. by Christian Haubelt and Dirk Timmermann. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2013, pp. 61–70.

[Git+13b] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. "Equivalence checking of partial designs using dependency quantified Boolean formulae". In: *IEEE 31st International Conference on Computer Design, ICCD 2013,* IEEE Computer Society, 2013, pp. 396–403. DOI: 10.1109/ICCD.2013.6657071.

[Git+15] Karina Gitina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker. “Solving DQBF through quantifier elimination”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015*. Ed. by Wolfgang Nebel and David Atienza. ACM, 2015, pp. 1617–1622. DOI: 10.7873/date.2015.0098.

[GMN21] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. “Reasoning with Quantified Boolean Formulas”. In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 1157–1176. DOI: 10.3233/FAIA201014.

[Gol+21] Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel. “Engineering an Efficient Boolean Functional Synthesis Engine”. In: *Proceedings of International Conference On Computer Aided Design (ICCAD)*. July 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643583.

[GRM20] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. “Manthan: A Data-Driven Approach for Boolean Function Synthesis”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 611–633. DOI: 10.1007/978-3-030-53291-8_31.

[GRM23] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. “Synthesis with Explicit Dependencies”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*. IEEE, 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137282.

[GSW19] Aile Ge-Ernst, Christoph Scholl, and Ralf Wimmer. “Localizing Quantifiers for DQBF”. In: *Formal Methods in Computer Aided Design, FMCAD 2019*. Ed. by Clark W. Barrett and Jin Yang. IEEE, 2019, pp. 184–192. DOI: 10.23919/FMCAD.2019.8894269.

[GV14] Arie Gurfinkel and Yakir Vizel. “DRUPing for interpolates”. In: *FMCAD 2014*. IEEE, 2014, pp. 99–106. DOI: 10.1109/FMCAD.2014.6987601.

[Haa+20] Winston Haaswijk, Mathias Soeken, Alan Mishchenko, and Giovanni De Micheli. “SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.4 (2020), pp. 871–884. DOI: 10.1109/TCAD.2019.2897703.

[Hen61] Leon Henkin. “Some Remarks on Infinitely Long Formulas”. In: *Infinitistic Methods*. Pergamon Press, 1961, pp. 167–183.

[HJS19] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. “SAT Competition 2018”. In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 133–154. DOI: 10.3233/SAT190120.

[Ign+21] Alexey Ignatiev, Edward Lam, Peter J. Stuckey, and João Marques-Silva. “A Scalable Two Stage Approach to Computing Optimal Decision Sets”. In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 3806–3814. DOI: 10.1609/AAAI.V35I5.16498.

[ILO20] Rahul Ilango, Bruno Loff, and Igor C. Oliveira. “NP-Hardness of Circuit Minimization for Multi-Output Functions”. In: *35th Computational Complexity Conference, CCC 2020, July 28-31, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by Shubhangi Saraf. Vol. 169. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 22:1–22:36. DOI: 10.4230/LIPICS.CCC.2020.22.

[IMM18] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. “PySAT: A Python Toolkit for Prototyping with SAT Oracles”. In: *SAT*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26.

[IPM15] Alexey Ignatiev, Alessandro Previti, and João Marques-Silva. “SAT-Based Formula Simplification”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*. Ed. by Marijn Heule and Sean A. Weaver. Vol. 9340. Lecture Notes in Computer Science. Springer, 2015, pp. 287–298. DOI: 10.1007/978-3-319-24318-4_21.

[Jan+16] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. “Solving QBF with counterexample guided refinement”. In: *Artif. Intell.* 234 (2016), pp. 1–25. DOI: 10.1016/J.ARTINT.2016.01.004.

[Jan18] Mikolás Janota. “Towards Generalization in QBF Solving via Machine Learning”. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 6607–6614. DOI: 10.1609/AAAI.V32I1.12208.

[JKL20] Jie-Hong Roland Jiang, Victor N. Kravets, and Nian-Ze Lee. “Engineering Change Order for Combinational and Sequential Design Rectification”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*. IEEE, 2020, pp. 726–731. DOI: 10.23919/DATE48585.2020.9116504.

[JKS16] Charles Jordan, Will Klieber, and Martina Seidl. “Non-CNF QBF Solving with QCIR”. In: *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*. Ed. by Adnan Darwiche. Vol. WS-16-05. AAAI Technical Report. AAAI Press, 2016. URL: https://aaai.org/papers/aaaiw-ws0186-16-12601/.

[JM13] Mikolás Janota and João Marques-Silva. “On Propositional QBF Expansions and Q-Resolution”. In: *Theory and Applications of Satisfiability Testing - SAT 2013*. Ed. by Matti Järvisalo and Allen Van Gelder. Vol. 7962. Lecture Notes in Computer Science. Springer, 2013, pp. 67–82. DOI: 10.1007/978-3-642-39071-5_7.

[JM15] Mikolás Janota and João Marques-Silva. “Expansion-based QBF solving versus Q-resolution”. In: *Theor. Comput. Sci.* 577 (2015), pp. 25–42. DOI: 10.1016/J.TCS.2015.01.048.

[JS17] Susmit Jha and Sanjit A. Seshia. “A theory of formal synthesis via inductive learning”. In: *Acta Informatica* 54.7 (2017), pp. 693–726. DOI: 10.1007/S00236-017-0294-5.

[KB21] Hans Kleine Büning and Uwe Bubeck. “Theory of Quantified Boolean Formulas”. In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 1131–1156. DOI: 10.3233/FAIA201013.

[KFB16] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. “Complexity of Fixed-Size Bit-Vector Logics”. In: *Theory Comput. Syst.* 59.2 (2016), pp. 323–376. DOI: 10.1007/S00224-015-9653-1.

[KKF95] Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. “Resolution for Quantified Boolean Formulas”. In: *Inf. Comput.* 117.1 (1995), pp. 12–18. DOI: 10.1006/INCO.1995.1025.

[KKY09] Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. “Finding Efficient Circuits Using SAT-Solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 32–44. DOI: 10.1007/978-3-642-02777-2_5.

[Knu11] Donald Ervin Knuth. *The Art of Computer Programming. Volume 4A, Combinatorial algorithms, Part 1*. 1st edition. Addison Wesley, 2011.

[Knu15] Donald Ervin Knuth. *The Art of Computer Programming : Satisfiability, Volume 4, Fascicle 6*. 1st edition. Addison Wesley, 2015.

[Kor08] Konstantin Korovin. “iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description)”. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 292–298. DOI: 10.1007/978-3-540-71070-7_24.

[Kov16] Gergely Kovásznai. “What is the State-of-the-Art in DQBF solving?” In: *MaCS-16. Joint Conference on Mathematics and Computer Science. 2016.* Ed. by Emil Vatei. Vol. 2046. CEUR-WS, 2016, pp. 185–196. URL: https://ceur-ws.org/Vol-2046/kovasznai.pdf.

[KPS22] Alexander S. Kulikov, Danila Pechenev, and Nikita Slezkin. “SAT-Based Circuit Local Improvement”. In: *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22-26, 2022, Vienna, Austria.* Ed. by Stefan Szeider, Robert Ganian, and Alexandra Silva. Vol. 241. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 67:1–67:15. DOI: 10.4230/LIPICS.MFCS.2022.67.

[Kro21] Daniel Kroening. “Software Verification”. In: *Handbook of Satisfiability - Second Edition.* Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 791–818. DOI: 10.3233/FAIA201004.

[KS21] Markus Kirchweger and Stefan Szeider. “SAT Modulo Symmetries for Graph Generation”. In: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021.* Vol. 210. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 34:1–34:16. DOI: 10.4230/LIPIcs.CP.2021.34.

[Law64] Eugene L. Lawler. “An Approach to Multilevel Boolean Minimization”. In: *J. ACM* 11.3 (1964), pp. 283–295. DOI: 10.1145/321229.321232.

[LE14] Florian Lonsing and Uwe Egly. “Incremental QBF Solving”. In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings.* Ed. by Barry O’Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, 2014, pp. 514–530. DOI: 10.1007/978-3-319-10428-7_38.

[Lee+18] Tung-Yuan Lee, Chia-Cheng Wu, Chia-Chun Lin, Yung-Chih Chen, and Chun-Yao Wang. “Logic optimization with considering boolean relations”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018.* Ed. by Jan Madsen and Ayse K. Coskun. IEEE, 2018, pp. 761–766. DOI: 10.23919/DATE.2018.8342109.

[Lee+22] Siang-Yun Lee, Heinz Riener, Alan Mishchenko, Robert K. Brayton, and Giovanni De Micheli. “A Simulation-Guided Paradigm for Logic Synthesis and Verification”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.8 (2022), pp. 2573–2586. DOI: 10.1109/TCAD.2021.3108704.

[Lew80] Harry R. Lewis. “Complexity Results for Classes of Quantificational Formulas”. In: *J. Comput. Syst. Sci.* 21.3 (1980), pp. 317–353. DOI: 10.1016/0022-0000(80)90027-6.

[LM08] Jérôme Lang and Pierre Marquis. "On propositional definability". In: *Artif. Intell.* 172.8-9 (2008), pp. 991–1017. DOI: 10.1016/J.ARTINT.2007.12.003.

[LOS16] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. "A SAT Approach to Branchwidth". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 179–195. DOI: 10.1007/978-3-319-40970-2_12.

[LOS17] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. "SAT-Encodings for Special Treewidth and Pathwidth". In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 429–445. DOI: 10.1007/978-3-319-66263-3_27.

[LOS19] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. "A SAT Approach to Branchwidth". In: *ACM Trans. Comput. Log.* 20.3 (2019), 15:1–15:24. DOI: 10.1145/3326159.

[Mar+21] Dewmini Sudara Marakkalage, Eleonora Testa, Heinz Riener, Alan Mishchenko, Mathias Soeken, and Giovanni De Micheli. "Three-Input Gates for Logic Synthesis". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 40.10 (2021), pp. 2184–2188. DOI: 10.1109/TCAD.2020.3032625.

[MB05] Alan Mishchenko and Robert K. Brayton. "SAT-Based Complete Don't-Care Computation for Network Optimization". In: *DATE*. IEEE Computer Society, 2005, pp. 412–417. DOI: 10.1109/DATE.2005.264.

[MCB06] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. "DAG-aware AIG rewriting a fresh look at combinational logic synthesis". In: *DAC*. Ed. by Ellen Sentovich. ACM, 2006, pp. 532–535. DOI: 10.1145/1146909.1147048.

[McC56] E. J. McCluskey. "Minimization of Boolean functions". In: *The Bell System Technical Journal* 35.6 (1956), pp. 1417–1444. DOI: 10.1002/j.1538-7305.1956.tb03835.x.

[McM03] Kenneth L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1.

[Mee+16] Kuldeep S. Meel, Moshe Y. Vardi, Supratik Chakraborty, Daniel J. Fremont, Sanjit A. Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. "Constrained Sampling and Counting: Universal Hashing Meets SAT Solving". In: *Beyond NP, Papers from the 2016 AAAI Workshop*. Ed. by Adnan Darwiche. Vol. WS-16-05. AAAI Workshops. AAAI Press, 2016.

[Mis+05] Alan Mishchenko, Satrajit Chatterjee, Jie-Hong Roland Jiang, and Robert Brayton. *FRAIGs: A unifying representation for logic synthesis and verification*. Tech. rep. ERL, 2005.

[Mis+11] Alan Mishchenko, Robert K. Brayton, Jie-Hong Roland Jiang, and Stephen Jang. "Scalable don't-care-based logic optimization and resynthesis". In: *ACM Trans. Reconfigurable Technol. Syst.* 4.4 (2011), 34:1–34:23. DOI: `10.1145/2068716.2068720`.

[Miy24] Yukio Miyasaka. "Transduction Method for AIG Minimization". In: *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2024, pp. 398–403. DOI: `10.1109/ASP-DAC58780.2024.10473816`.

[MLM21] João Marques-Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 133–182. DOI: `10.3233/FAIA200987`.

[PAR01] Gary Peterson, Salman Azhar, and John Reif. "Lower bounds for multiplayer noncooperative games of incomplete information". In: *Journal of Computers and Mathematics with Applications* 41.7 (2001), pp. 957–992. ISSN: 0898-1221. DOI: `10.1016/S0898-1221(00)00333-3`.

[PG86] David A. Plaisted and Steven Greenbaum. "A Structure-Preserving Clause Form Translation". In: *J. Symb. Comput.* 2.3 (1986), pp. 293–304. DOI: `10.1016/S0747-7171(86)80028-1`.

[PPC96] Dhiraj K. Pradhan, Debjyoti Paul, and Mitrajit Chatterjee. "VERILAT: verification using logic augmentation and transformations". In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*. Ed. by Rob A. Rutenbar and Ralph H. J. M. Otten. IEEE, 1996, pp. 88–95. DOI: `10.1109/ICCAD.1996.569111`.

[PR79] Gary L. Peterson and John H. Reif. "Multiple-Person Alternation". In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)* (1979), pp. 348–363. DOI: `10.1109/SFCS.1979.25`.

[PSH23] Luca Pulina, Martina Seidl, and Simone Heisinger. *QBFGallery 2023*. 2023. URL: `https://qbf23.pages.sai.jku.at/gallery/` (visited on 2024-06-05).

[PSS24] Luca Pulina, Martina Seidl, and Ankit Shukla. *QBFEVAL*. 2024. URL: `http://www.qbflib.org/index_eval.php` (visited on 2024-04-08).

[Qui52] W. V. Quine. “The Problem of Simplifying Truth Functions”. In: *The American Mathematical Monthly* 59.8 (1952), pp. 521–531. ISSN: 00029890, 19300972. URL: `http://www.jstor.org/stable/2308219`.

[Qui86] J. Ross Quinlan. “Induction of Decision Trees”. In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: `10.1023/A:1022643204877`.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN: 1-55860-238-0.

[Rab17] Markus N. Rabe. “A Resolution-Style Proof System for DQBF”. In: *Theory and Applications of Satisfiability Testing - SAT 2017*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 314–325. DOI: `10.1007/978-3-319-66263-3_20`.

[Rie+19] Heinz Riener, Winston Haaswijk, Alan Mishchenko, Giovanni De Micheli, and Mathias Soeken. “On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis”. In: *DATE*. Ed. by Jürgen Teich and Franco Fummi. IEEE, 2019, pp. 1649–1654. DOI: `10.23919/DATE.2019.8715185`.

[Rie+22] Heinz Riener, Siang-Yun Lee, Alan Mishchenko, and Giovanni De Micheli. “Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis”. In: *ASP-DAC*. IEEE, 2022, pp. 395–402. DOI: `10.1109/ASP-DAC52403.2022.9712526`.

[Rin21] Jussi Rintanen. “Planning and SAT”. In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 765–789. DOI: `10.3233/FAIA201003`.

[Riv87] Ronald L. Rivest. “Learning Decision Lists”. In: *Mach. Learn.* 2.3 (1987), pp. 229–246. DOI: `10.1007/BF00058680`.

[RMS20] Heinz Riener, Alan Mishchenko, and Mathias Soeken. “Exact DAG-Aware Rewriting”. In: *DATE*. IEEE, 2020, pp. 732–737. DOI: `10.23919/DATE48585.2020.9116379`.

[Rou11] Olivier Roussel. “Controlling a Solver Execution with the runsolver Tool”. In: *J. Satisf. Boolean Model. Comput.* 7.4 (2011), pp. 139–144. DOI: `10.3233/SAT190083`.

[RS04] Kavita Ravi and Fabio Somenzi. “Minimal Assignments for Bounded Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 31–45. DOI: `10.1007/978-3-540-24730-2_3`.

[RS16] Markus N. Rabe and Sanjit A. Seshia. “Incremental Determinization”. In: *Theory and Applications of Satisfiability Testing - SAT 2016*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 375–392. DOI: 10.1007/978-3-319-40970-2_23.

[RS20] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “MaxSAT-Based Postprocessing for Treedepth”. In: *CP*. Ed. by Helmut Simonis. Vol. 12333. Lecture Notes in Computer Science. Springer, 2020, pp. 478–495. DOI: 10.1007/978-3-030-58475-7_28.

[RS21a] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “Learning Fast-Inference Bayesian Networks”. In: *NeurIPS*. Ed. by Marc’Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan. 2021, pp. 17852–17863.

[RS21b] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “Turbocharging Treewidth-Bounded Bayesian Network Structure Learning”. In: *AAAI*. AAAI Press, 2021, pp. 3895–3903. DOI: 10.1609/AAAI.V35I5.16508.

[RS22] Franz-Xaver Reichl and Friedrich Slivovsky. “Pedant: A Certifying DQBF Solver”. In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*. Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:10. DOI: 10.4230/LIPICS.SAT.2022.20.

[RSS21] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. “Certified DQBF Solving by Definition Extraction”. In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 499–517. DOI: 10.1007/978-3-030-80223-3_34.

[RSS23] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. “Circuit Minimization with QBF-Based Exact Synthesis”. In: *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Washington, DC, USA, February 7-14, 2023*. Ed. by Brian Williams, Yiling Chen, and Jennifer Neville. AAAI Press, 2023, pp. 4087–4094. DOI: 10.1609/AAAI.V37I4.25524.

[RSS24] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. “eSLIM: Circuit Minimization with SAT Based Local Improvement”. In: *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*. Ed. by Supratik Chakraborty and Jie-Hong Roland Jiang. Vol. 305. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 23:1–23:14. DOI: 10.4230/LIPICS.SAT.2024.23.

[RT15] Markus N. Rabe and Leander Tentrup. “CAQE: A Certifying QBF Solver”. In: *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*. Ed. by Roope Kaivola and Thomas Wahl. IEEE, 2015, pp. 136–143. DOI: 10.1109/FMCAD.2015.7542263.

[Sav92] Hamid Savoj. “Don’t cares in multi-level network optimization”. PhD thesis. University of California, Berkeley, 1992.

[SB01] Christoph Scholl and Bernd Becker. “Checking Equivalence for Partial Implementations”. In: *Proceedings of the 38th Design Automation Conference, DAC 2001*. ACM, 2001, pp. 238–243. DOI: 10.1145/378239.378471.

[SB90] Hamid Savoj and Robert K. Brayton. “The Use of Observability and External Don’t Cares for the Simplification of Multi-Level Networks”. In: *DAC*. IEEE Computer Society Press, 1990, pp. 297–301. DOI: 10.1145/123186.123280.

[Sch+19] Christoph Scholl, Jie-Hong Roland Jiang, Ralf Wimmer, and Aile Ge-Ernst. “A PSPACE Subclass of Dependency Quantified Boolean Formulas and Its Effective Solving”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*. AAAI Press, 2019, pp. 1584–1591. DOI: 10.1609/AAAI.V33I01.33011584.

[Sch22] André Schidler. “SAT-Based Local Search for Plane Subgraph Partitions (CG Challenge)”. In: *SoCG*. Ed. by Xavier Goaoc and Michael Kerber. Vol. 224. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 74:1–74:8. DOI: 10.4230/LIPICS.SOCG.2022.74.

[Síč20] Juraj Síč. “Satisfiability of DQBF using binary decision diagrams”. MA thesis. Brno, Czech Republic: Masaryk University, 2020.

[Sin05] Carsten Sinz. “Towards an Optimal CNF Encoding of Boolean Cardinality Constraints”. In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, 2005, pp. 827–831. DOI: 10.1007/11564751_73.

[SJB08] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. “Sketching concurrent data structures”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 136–148. DOI: 10.1145/1375581.1375599.

[Sli20] Friedrich Slivovsky. “Interpolation-Based Semantic Gate Extraction and Its Applications to QBF Preprocessing”. In: *CAV (1)*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 508–528. DOI: 10.1007/978-3-030-53288-8_24.

[Sli22] Friedrich Slivovsky. “Quantified CDCL with Universal Resolution”. In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel.* Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 24:1–24:16. DOI: 10.4230/LIPICS.SAT.2022.24.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. “Word problems requiring exponential time(Preliminary Report)”. In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing.* Ed. by Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong. STOC ’73. Austin, Texas, USA: Association for Computing Machinery, 1973, pp. 1–9. ISBN: 9781450374309. DOI: 10.1145/800125.804029.

[SMM17] Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. “Busy man’s synthesis: Combinational delay optimization with SAT”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017.* Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 830–835. DOI: 10.23919/DATE.2017.7927103.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. “Extending SAT Solvers to Cryptographic Problems”. In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009.* Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[Soe+17] Mathias Soeken, Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Exact Synthesis of Majority-Inverter Graphs and Its Applications”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 36.11 (2017), pp. 1842–1855. DOI: 10.1109/TCAD.2017.2664059.

[Soe+22] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, and Giovanni De Micheli. “The EPFL Logic Synthesis Libraries”. In: *CoRR* (2022). DOI: 10.48550/ARXIV.1805.05121.

[Sol+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. “Combinatorial sketching for finite programs”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006.* Ed. by John Paul Shen and Margaret Martonosi. ACM, 2006, pp. 404–415. DOI: 10.1145/1168857.1168907.

[SS16] Friedrich Slivovsky and Stefan Szeider. “Soundness of Q-resolution with dependency schemes”. In: *Theor. Comput. Sci.* 612 (2016), pp. 83–101. DOI: 10.1016/J.TCS.2015.10.020.

[SS21a] André Schidler and Stefan Szeider. “SAT-based Decision Tree Learning for Large Data Sets”. In: *AAAI*. AAAI Press, 2021, pp. 3904–3912. DOI: 10.1609/AAAI.V35I5.16509.

[SS21b] Juraj Síč and Jan Strejček. “DQBDD: An Efficient BDD-Based DQBF Solver”. In: *Theory and Applications of Satisfiability Testing – SAT 2021*. Ed. by Chu-Min Li and Felip Manyà. Cham: Springer International Publishing, 2021, pp. 535–544. ISBN: 978-3-030-80223-3. DOI: 10.1007/978-3-030-80223-3_36.

[SS23] André Schidler and Stefan Szeider. “SAT-boosted Tabu Search for Coloring Massive Graphs”. In: *ACM J. Exp. Algorithmics* 28 (2023), 1.5:1–1.5:19. DOI: 10.1145/3603112.

[SS99] João P. Marques Silva and Karem A. Sakallah. “GRASP: A Search Algorithm for Propositional Satisfiability”. In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433.

[SW18] Christoph Scholl and Ralf Wimmer. “Dependency Quantified Boolean Formulas: An Overview of Solution Methods and Applications”. In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 3–16. ISBN: 978-3-319-94144-8. DOI: 10.1007/978-3-319-94144-8_1.

[Ten16] Leander Tentrup. “Non-prenex QBF Solving Using Abstraction”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 393–401. DOI: 10.1007/978-3-319-40970-2_24.

[Ten19] Leander Tentrup. “CAQE and QuAbS: Abstraction Based QBF Solvers”. In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 155–210. DOI: 10.3233/SAT190121.

[Tes+20] Eleonora Testa, Luca G. Amarù, Mathias Soeken, Alan Mishchenko, Patrick Vuillod, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Extending Boolean Methods for Scalable Logic Synthesis”. In: *IEEE Access* 8 (2020), pp. 226828–226844. DOI: 10.1109/ACCESS.2020.3045014.

[TR19] Leander Tentrup and Markus N. Rabe. “Clausal Abstraction for DQBF”. In: *Theory and Applications of Satisfiability Testing - SAT 2019*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 388–405. DOI: 10.1007/978-3-030-24258-9_27.

[Tse83] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28.

[UVS06] Christopher Umans, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. “Complexity of two-level logic minimization”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 25.7 (2006), pp. 1230–1246. DOI: 10.1109/TCAD.2005.855944.

[VGM15] Yakir Vizel, Arie Gurfinkel, and Sharad Malik. “Fast Interpolating BMC”. In: *CAV 2015*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 641–657. DOI: 10.1007/978-3-319-21690-4_43.

[Wil+08] Robert Wille, Hoang Minh Le, Gerhard W. Dueck, and Daniel Große. “Quantified Synthesis of Reversible Logic”. In: *DATE*. Ed. by Donatella Sciuto. ACM, 2008, pp. 1015–1020. DOI: 10.1109/DATE.2008.4484814.

[Wim+16a] Karina Wimmer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. “Skolem Functions for DQBF”. In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Ed. by Cyrille Artho, Axel Legay, and Doron Peled. Vol. 9938. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 395–411. ISBN: 978-3-319-46520-3. DOI: 10.1007/978-3-319-46520-3_25.

[Wim+16b] Ralf Wimmer, Christoph Scholl, Karina Wimmer, and Bernd Becker. “Dependency Schemes for DQBF”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 473–489. DOI: 10.1007/978-3-319-40970-2_29.

[Wim+17] Ralf Wimmer, Andreas Karrenbauer, Ruben Becker, Christoph Scholl, and Bernd Becker. “From DQBF to QBF by Dependency Elimination”. In: *Theory and Applications of Satisfiability Testing - SAT 2017, Proceedings*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 326–343. DOI: 10.1007/978-3-319-66263-3_21.

[Win96] Franz Winkler. *Polynomial Algorithms in Computer Algebra*. Texts & Monographs in Symbolic Computation. Springer, 1996. ISBN: 978-3-211-82759-8. DOI: 10.1007/978-3-7091-6571-3.

[WSB19] Ralf Wimmer, Christoph Scholl, and Bernd Becker. “The (D)QBF Preprocessor HQSpre - Underlying Theory and Its Implementation”. In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 3–52. DOI: 10.3233/SAT190115.

[Zha+01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. “Efficient Conflict Driven Learning in Boolean Satisfiability Solver”. In: *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001.* Ed. by Rolf Ernst. IEEE Computer Society, 2001, pp. 279–285. DOI: 10.1109/ICCAD.2001.968634.