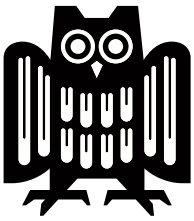


# Symbolic Reactive Synthesis

A dissertation submitted towards the degree Doctor of Natural Sciences (Dr. rer. nat.) of the Faculty of Mathematics and Computer Science of Saarland University

Leander T.J. Tentrup

Saarbrücken
2019

| | |
|---|---|
| Day of Colloquium | November 15th, 2019 |
| Dean of Faculty | Prof. Dr. Sebastian Hack |
| Chair of the Committee | Prof. Dr. Jan Reineke |
| Reviewers | Prof. Bernd Finkbeiner, Ph.D. |
| | Prof. Dr. Martina Seidl |
| | Prof. Dr. Helmut Seidl |
| Academic Assistant | Dr. Roland Leißa |

# Acknowledgements

I want to express my deepest gratitude to my advisor Bernd Finkbeiner. In addition to introducing me to the theory of reactive systems and the intriguing synthesis problem, he also let me pursue my research, which resulted in my endeavor into QBF solving.

I am thankful for the collaboration during and in-between the numerous coffee breaks by the members of the Reactive Systems Group at Saarland University: Norine Coenen, Rayna Dimitrova, Peter Faymonville, Michael Gerke, Christopher Hahn, Jesko Hecking-Harbusch, Jana Hofmann, Swen Jacobs, Felix Klein, Andrey Kupriyanov, Noemi Passing, Markus Rabe, Mouhammad Sakr, Christa Schäfer, Malte Schledjewski, Maximilian Schwenger, Hazem Torfah, Alexander Weinert, and Martin Zimmermann. I am glad I was able to work with Marvin Stenger on many exciting projects, from hyperproperties to monitoring. I thank the external reviewers Martina Seidl and Helmut Seidl for their time and effort, as well as their feedback.

I am grateful to the German Research Foundation (DFG) for supporting this work as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), under the project SpAGAT (FI 936/2-1), and as part of the Collaborative Research Centers "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223) and "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), as well as to the European Research Council (ERC) for supporting this work as part of the grant OSARES (No. 683300). I am thankful for the funding by the Saarbrücken Graduate School of Computer Science and the German Academic Scholarship Foundation.

This work would not have been possible without the support and faith of my family and friends. Special thanks to my parents, who have created the conditions for my studies through their support and encouragement.

Last, I owe special thanks to my wife, Lisa, for taking this journey together, for her assistance in difficult times, and her patience with me working late.

# Abstract

In this thesis, we develop symbolic algorithms for the synthesis of reactive systems. Synthesis, that is the task of deriving correct-by-construction implementations from formal specifications, has the potential to eliminate the need for the manual—and error-prone—programming task. The synthesis problem can be formulated as an infinite two-player game, where the system player has the objective to satisfy the specification against all possible actions of the environment player. The standard synthesis algorithms represent the underlying synthesis game explicitly and, thus, they scale poorly with respect to the size of the specification.

We provide an algorithmic framework to solve the synthesis problem symbolically. In contrast to the standard approaches, we use a succinct representation of the synthesis game, which leads to improved scalability in terms of the symbolically represented parameters. Our algorithm reduces the synthesis game to the satisfiability problem of quantified Boolean formulas (QBF) and dependency quantified Boolean formulas (DQBF). In the encodings, we use propositional quantification to succinctly represent different parts of the implementation, such as the state space and the transition function.

We develop highly optimized satisfiability algorithms for QBF and DQBF. Based on a counterexample-guided abstraction refinement (CEGAR) loop, our algorithms avoid an exponential blow-up by using the structure of the underlying symbolic encodings. Further, we extend the solving algorithms to extract certificates in the form of Boolean functions, from which we construct implementations for the synthesis problem. Our empirical evaluation shows that our symbolic approach significantly outperforms previous explicit synthesis algorithms with respect to scalability and solution quality.

# Zusammenfassung

In dieser Dissertation werden symbolische Algorithmen für die Synthese von reaktiven Systemen entwickelt. Synthese, d.h. die Aufgabe, aus formalen Spezifikationen korrekte Implementierungen abzuleiten, hat das Potenzial, die manuelle und fehleranfällige Programmierung überflüssig zu machen. Das Syntheseproblem kann als unendliches Zweispielerspiel verstanden werden, bei dem der Systemspieler das Ziel hat, die Spezifikation gegen alle möglichen Handlungen des Umgebungsspielers zu erfüllen. Die Standardsynthesealgorithmen stellen das zugrunde liegende Synthesespiel explizit dar und skalieren daher schlecht in Bezug auf die Größe der Spezifikation.

Diese Arbeit präsentiert einen algorithmischen Ansatz, der das Syntheseproblem symbolisch löst. Im Gegensatz zu den Standardansätzen wird eine kompakte Darstellung des Synthesespiels verwendet, die zu einer verbesserten Skalierbarkeit der symbolisch dargestellten Parameter führt. Der Algorithmus reduziert das Synthesespiel auf das Erfüllbarkeitsproblem von quantifizierten booleschen Formeln (QBF) und abhängigkeitsquantifizierten booleschen Formeln (DQBF). In den Kodierungen verwenden wir propositionale Quantifizierung, um verschiedene Teile der Implementierung, wie den Zustandsraum und die Übergangsfunktion, kompakt darzustellen.

Wir entwickeln hochoptimierte Erfüllbarkeitsalgorithmen für QBF und DQBF. Basierend auf einer gegenbeispielgeführten Abstraktionsverfeinerungsschleife (CEGAR) vermeiden diese Algorithmen ein exponentielles Blow-up, indem sie die Struktur der zugrunde liegenden symbolischen Kodierungen verwenden. Weiterhin werden die Lösungsalgorithmen um Zertifikate in Form von booleschen Funktionen erweitert, aus denen Implementierungen für das Syntheseproblem abgeleitet werden. Unsere empirische Auswertung zeigt, dass unser symbolischer Ansatz die bisherigen expliziten Synthesealgorithmen in Bezug auf Skalierbarkeit und Lösungsqualität deutlich übertrifft.

# Contents

# Chapter 1

# Introduction

In this thesis, we develop symbolic algorithms for the synthesis of reactive systems. A reactive system is a device that continuously interacts with an environment. Examples include embedded controllers, hardware circuits, and communication protocols. Synthesis, that is the task of deriving correct-by-construction implementations from formal specifications, has the potential to eliminate the need for the manual–and error-prone–programming task. In case the specification is *unrealizable*, that is, there are conflicting requirements that rule out the existence of any realizing implementation, synthesis algorithms can detect such situations early in the design phase and guide the refinement of the erroneous specification by providing counterexamples. The synthesis problem is one of the fundamental problems in the theory of reactive systems, with the first formulation dating back to Alonzo Church [Chu57] more than 60 years ago. Until now, most research has focused on synthesis methods based on explicit data structures.

The classic solution to the synthesis problem is based on work by Büchi and Landweber [BL69] and employs automata- and game-theory: Given a specification, e.g., linear-time temporal logic (LTL) [Pnu77], the first step translates this specification into an equivalent non-deterministic Büchi word automaton [VW94]. Afterward, the automaton is transformed into a deterministic parity tree automaton [Saf88; Pit07] that accepts those infinite trees that satisfy the specification. Deciding the emptiness problem of the tree automaton, by solving the underlying parity game, then solves the realizability problem. As observed by Kupferman and Vardi, the determinization procedure for non-deterministic Büchi word automata, also called *Safra construction*, "involves complicated data structures" [KV05] and, as a consequence, is not suitable for a symbolic implementation [KV05]. Even modern implementations of this approach, like LTLSYNT [MC18] and STRIX [MSL18], use explicit data structures up to and including the construction and subsequent solving of the resulting parity games.

**Symbolic Verification.** Compare and contrast this situation with the success of model checking, which now routinely handle systems of enormous size due to symbolic algorithms. The advent of modern verification started with the introduction of symbolic model checking algorithms [Bur+90] using reduced ordered binary decision diagrams

(BDDs) [Bry86] to represent the state-space of the underlying system. This work improves on the at this time common practice of traversing the state-graph of a system explicitly and the resulting poor scalability with respect to the size of the system, commonly known as the "state explosion problem" [Bur+90]. Efficient verification techniques [GSV14] from bounded model checking [Bie+99] and interpolation-based unbounded model checking [McM03] to inductive methods like IC3 [Bra11] and property direct reachability [EMB11] pushed the scalability of model checking even further. All those methods have in common that they have been enabled by the development of scalable propositional satisfiability (SAT) solvers.

A propositional formula is a formula containing only propositional variables, i.e., variables whose values can either be *true* **T** or *false* **F**, and connectives such as disjunction, conjunction, and exclusive or. Propositional satisfiability (SAT) is the problem to determine whether, for a given propositional formula, there is a satisfying assignment, that is, a function mapping variables to a Boolean value, such that the formula evaluates to true under this assignment. While the asymptotical complexity is not encouraging—propositional satisfiability is the prime NP-complete problem—and all known algorithms have exponential worst-case running time, SAT solvers routinely handle instances up to millions of variables. Investigating this discrepancy is an active field of research [IP01; CIP09; VW19], but there is the common belief that those solvers perform well due to the inherent structure of instances coming, for example, from model checking [New+14]. On the algorithmic side, conflict-driven clause learning (CDCL) [JS97; SS99] is the dominating satisfiability algorithm. As SAT solving is highly competitive, much time and engineering is spent towards optimizing every aspect of the implementation, including, but not limited to, the underlying data structures and various kinds of heuristics such as decision heuristics, clause deletion heuristics, and restart heuristics. On the other hand, the interface to a SAT solver is standardized[1] and easy to use, thus, it is an ideal building block not only for verification algorithms but also in many recent algorithms for quantified satisfiability [Jan+12; JM15b; RT15; THJ15; Jan+16; Ten16; HT18; LWJ18; Blo+18; TR19a].

**Symbolic Synthesis.** With the advent of Safraless decision procedures [KV05], *symbolic* synthesis methods started to gain more attention. The idea is to avoid Safra's determinization procedure by translating the specification into an equivalent universal co-Büchi automaton, whose language is then approximated in a sequence of deterministic safety automata, obtained by bounding the number of visits to rejecting states [FS07]. Examples for a symbolic representation of the state space of the approximated safety games are *antichains* [FJR11; Boh+12] and *BDDs* [Ehl12]. All of the synthesis approaches mentioned so far have the drawback that the size of the synthesized systems is often unnecessarily (and impractically) large (cf. [FJ12]) since the automata representing the specification often contain many more states than are needed by the implementation. In *bounded synthesis* [FS07; FS13], one can ensure that the synthesized system is the smallest possible realization of the specification by bounding the size of the to-be-synthesized

[1] The C API is called IPASIR and was first used in the SAT Race 2015 [Bal+16b].

implementation and by iteratively increasing the bound until a realizing implementation is found.

In the original formulation of bounded synthesis [FS07; SF07] and in many derived works [FS13; FJ12; KJB13b; KJB13a; Fin+18a; Coe+19] the constraint systems are built in a decidable first-order theory and solved using powerful SMT solvers. In the standard encoding, both the states of the synthesized system and its inputs are enumerated explicitly [FS13]. This thesis shows how *quantification* can be used to derive more succinct and, thereby, more symbolic representations of the bounded synthesis constraint system. As a target logic for the constraint system we use quantified Boolean formulas (QBF), that is propositional logic extended by (linear) propositional quantification, and dependency quantified Boolean formulas (DQBF), which allow branching quantification also know as *Henkin* quantifiers. The resulting reductions are landmarks on the spectrum of symbolic vs. explicit encodings.

All of our bounded synthesis encodings represent the synthesized system in terms of its transition function, which identifies the successor state using the current state and the input, and additionally in terms of an output function, which identifies the output signals using the current state and the input. Additionally, the encoding contains an annotation function, which relates the states of the system to the states of a universal automaton representing the specification.

In the SAT encoding of the transition function, a separate Boolean variable is used for every combination of a source state, an input signal, and a target state. The encoding is thus explicit in both the state and the input. In the QBF encoding, we quantify universally over the inputs, so that the encoding becomes symbolic in the inputs while staying explicit in the states. Quantifying universally over the states, just like over the input signals, is not possible in QBF because the states occur twice in the transition function, as a source, and as a target. Separate quantifiers over sources and targets would allow for models where, for example, the value of the output function differs, even though both the source state and the input are the same. In DQBF, we can avoid such artifacts and obtain a "fully symbolic" encoding in both the states and the input.

We evaluate the encodings systematically using benchmarks from the reactive synthesis competition (SYNTCOMP) [Jac+17b] and state-of-the-art solvers. Our empirical finding is that both the input-symbolic and state-symbolic encoding, perform better than the non-symbolic approach. This fits with our intuition that a more symbolic encoding provides opportunities for optimization in the solver.

To benefit from those more symbolic encodings, the choice of the underlying satisfiability algorithms for QBF and DQBF are crucial. Abstraction-based algorithms for quantifier elimination can exploit symbolic encodings to avoid an exponential blow-up. The first part of this thesis presents the *clausal abstraction* approach for solving the quantified satisfiability problems mentioned above. The clausal abstraction algorithm is inspired by the counterexample-guided abstraction refinement (CEGAR) [Cla+00] style of reasoning, which has also been successfully used in QBF solvers before [Jan+16; JM15b]. On a high level, the algorithm splits the given quantified problem into a sequence of propositional problems, one for each quantifier in the quantifier prefix, and instantiates a propositional SAT solver for each of them. Those SAT oracles solve the quantified problem by commu-

nicating assignments (representing examples and counterexamples) to their neighbors. Clausal abstraction has been very successful, winning the annual QBF competition QBFEVAL [PS19] since 2017, and has been used as a solving method for QBF encodings of various formal method problems [FFT17; FHH18; Fin+17a].

## 1.1 Quantified Satisfiability as a Building Block for Synthesis

QBF satisfiability has been repeatedly used to solve many problems in computer science, including but not limited to robotic planning [Rin07; Egl+17], ontology reasoning [Kon+09], model checking [DHK05; BM08; MSB13; Zha14], fault localization in hardware designs [SB07], circuit synthesis [JLH09], program synthesis [Sol+06], reactive synthesis [Fay+17; FFT17; CHR16; CLR17], satisfiability of hyperproperties [FHH18], solving Petri games [Fin15; Fin+17a], and debugging fault-tolerance specifications [FT14a; FT15]. QBF solving techniques have also been used to build custom and integrated synthesis procedures that exploit the knowledge of the underlying synthesis problem to synthesize the winning strategy for safety games [BKS14; Blo+14], debugging of circuits [Gas+14], and the simulation of axiomatic memory models [Coo+19]. In many cases, the binary truth/falsity answer from a QBF encoding of a synthesis problem is only part of the desired solver output. A certificate of the solving result, in the form of Boolean functions, typically directly corresponds to the solution of the original problem, like a realizing implementation in the case of reactive synthesis. We show how to certify the clausal abstraction algorithm by providing a method to build Boolean functions during solving.

The simplest form of QBF contains only a single quantifier alternation, that is, the quantifier prefix is of the form $\forall X \exists Y$. The satisfiability problem, which corresponds to a game where the existential player has complete information over the choice of the universal player, is $\Pi_2^P$-complete. Using more than one alternation, QBF satisfiability spans the complete polynomial hierarchy, where the satisfiability problem with unbounded quantifier alternations is PSpace-complete [SM73]. Using branching quantifiers such as Henkin quantifier, the satisfiability problem for the resulting logic, called dependency quantified Boolean formula (DQBF), becomes NExpTime-complete [PRA01]. Beyond the prefix, there is a second characterization of QBF solving methods based on the structure of the propositional formula: The standard solving format inherits a matrix representation called conjunctive normal form (CNF), that is, the propositional part consists of sets of clauses where every clause is a set of literals (which are variables or their negation). Every propositional formula can be transformed into CNF using auxiliary variables [Tse68]. To avoid those auxiliary variables and the resulting proof-theoretic weaknesses [JM17], more general structures have been used in the context of QBF solving, such as circuits and formulas in negation normal form (NNF).

The abstraction-based solving algorithm behind clausal abstraction is versatile as demonstrated in the first part of this thesis: it can handle QBFs from prenex conjunctive normal form to non-prenex and non-CNF, respectively, as well as branching quantifiers. An analysis of the algorithm in terms of proof theory reveals that clausal abstraction is closely related to *search-based* solving algorithms, i.e., the underlying proof systems in

both cases is a variant of $Q$-resolution [KKF95]. Solvers based on search assign variables in the order given by the quantifier prefix and progress by learning clauses and cubes for conflicts and solutions, respectively. In contrast, *expansion-based* solving methods eliminate universal quantifiers, which in the worst-case, continue until the formula is propositional. Search- and expansion-based solvers have incomparable proof systems [JM15a]. Thus, there are families of QBF formulas that have polynomial search-based refutations and only exponential expansion-based refutations, and vice versa. Due to the flexibility inherent in the abstraction, clausal abstraction can be extended to include refinements based on partial expansion. The resulting algorithm, a hybrid of search and expansion, has proof-theoretic advantages over both approaches individually, and this advantage also translates to the empirical solving performance.

## 1.2 Beyond Linear-time Specifications

The classic synthesis problem asks for an implementation in terms of a transition system or sequential circuit. The most prominent extension has been to distributed systems, called the *distributed synthesis* problem [PR90; KV01; FS05], where multiple transition systems are arranged in a *distributed architecture* which specifies the process topology and the synthesis problem asks for an implementation of every process such that the mutual interaction satisfies a global specification. In this setting, there is the intriguing question of how information is propagated through the system to achieve the joint goal. On the negative side, the complexity result is underwhelming: Already the simple architecture containing two processes with pairwise disjoint information from the environment, depicted in Figure 8.1a, is undecidable [PR90], which, comes from the high expressiveness of being able to represent a run of a Turing machine [PR90].

More recently, there has been work on temporal logic that includes strategic behavior, that is, in the extreme case, the synthesis problem itself. Examples include alternating-time temporal logic (ATL) [AHK97], strategy logic [CHP10], and Coordination Logic [FS10]. Coordination Logic has the convenient property of being able to (syntactical) express exactly those architectures in the distributed synthesis problem for which the realizability problem is decidable, that is, architectures without *information fork* [FS05]. Intuitively, an architecture contains an information fork if the environment can propagate disjoint bits of information to two distinct processes, such that the information given to the respective other process is not observable. On the other hand, many interesting examples of distributed systems contain such kinds of forks like consensus problems, and we will see many more in the second half of this thesis.

A related, although different kind of research direction is to characterize the *flow of information* through a distributed system. This provides an alternative characterization of architectures as constraining the information-flow between processes: A process reacts *consistently* if, and only if, for every pair of execution traces of the overall system, a change of its outputs on this pair is preceded by an observable difference, that is, a different valuation of one of its input propositions. More general, properties that relate *multiple* execution traces are known as *hyperproperties* [CS10]. Hyperproperties generalize

trace properties, i.e., sets of traces, to *sets of sets* of traces, and are of huge importance to the security community in the form of noninterference [GM82], observational determinism [ZM03], plausible deniability [BSG17; CCS17], and alike, but have also ties to classical formal methods problems like determining if an input to a hardware circuit influences an output [Fin+18b], distributed systems in the form of fault-tolerance [Fin+18a], symmetry in hardware designs [FRS15], or the verification of error-correcting codes [FRS15].

The temporal hyperlogic HyperLTL [Cla+14] is an extension of LTL that employs explicit trace quantification, i.e., it can quantify over multiple execution traces and then relate them using temporal operators. HyperLTL is able to express many hyperproperties of choice, for example safety hyperproperties, also called *hypersafety* properties, like distributivity mentioned above and liveness hyperproperties, also called *hyperliveness* properties, like plausible deniability ("For every execution trace there is a different one with the same (public) observations but different valuations of some secret/undisclosed variables") or generalized noninterference [McC88]. HyperLTL has been used to verify security properties in hardware designs [Cla+14; FRS15; FHT18] and to detect information-flow violations at runtime [Fin+17b; Fin+18b; HST19; Fin+19b]. Also, the satisfiability problem for HyperLTL has been studied extensively [FH16; FHS17; FHH18]. In this thesis, we investigate the realizability problem of HyperLTL and show that it subsumes many classic variants of the realizability problem such as distributed, fault-tolerant, and symmetric synthesis. While there are no gains concerning decidability results—given that it subsumes the distributed realizability variant—there is a strong motivation for investigating the HyperLTL synthesis problem: It is now possible to mix-and-match all those kinds of properties within a single, unifying framework. Consider, for example, the *dining cryptographers* problem [Cha85]: Three cryptographers $C_a$, $C_b$, and $C_c$ sit at a table in a restaurant having dinner, and either one of cryptographers or the NSA must pay for their meal. Is there a protocol where each cryptographer can find out whether it was a cryptographer who paid or the NSA, but cannot find out which cryptographer paid the bill? While being an easy to grasp problem thanks to the figurative description, the formal description of this protocol is more involved due to the mixture of linear properties, describing the protocol behavior and its assumptions, distributivity, a hypersafety property, and secrecy, a hyperliveness property. The search for a realizing protocol can be encoded as a single HyperLTL formula and can be synthesized efficiently using techniques presented in this thesis. We evaluate our HyperLTL synthesis algorithms on a set of case studies, including classic examples from the theory of distributed systems and cryptography like the CAP theorem and the dining cryptographers problem, as well as symmetries in hardware designs and error-correcting codes.

## 1.3 Contributions

**Symbolic Encodings of Bounded Synthesis.** In this thesis, we show how to derive more symbolic bounded synthesis constraints by a reduction to QBF and DQBF. We present a method to effectively solve those constraints, leading the way for improved scalability and solution quality compared to the original SMT formulation [FS07; SF07]. The QBF

and DQBF encodings were submitted to the annual QBF competition, helping diversify the benchmark set and spreading awareness of the synthesis problem for researchers working on solver development. The corresponding synthesis tool, called BoSy [FFT17], is publicly available in source code and has been used by students all over the world to solve synthesis tasks. BoSy won the reactive synthesis competition 2016 and 2017 in the sequential LTL synthesis track.

**HyperLTL Synthesis.** We were the first to analyze the realizability problem for HyperLTL. This thesis investigates the *expressivity* and *decidability* of the realizability problem and, further, gives efficient *semi-decision procedures* based on bounded synthesis [FS13]. On the expressiveness, we show that HyperLTL realizability subsumes many classical extensions of the LTL realizability problem like incomplete information [KV97], distributed synthesis [PR90; KV01; FS05], and fault-tolerant synthesis [DF09; FT15]. We analyze the decidability of the decision problem based on the quantifier prefix of a HyperLTL formula and show decidable subclasses of the otherwise undecidable class of universal HyperLTL formulas.

**QBF Solving.** We have developed a QBF solving methodology based on the counterexample-guided abstraction refinement (CEGAR) [Cla+00] algorithm. There are two outstanding properties: First, the algorithm is versatile and adaptable, as variants of the base algorithm apply to QBF and DQBF, prenex and non-prenex, as well as clausal and non-clausal formulas. Second, the implementations in the solvers CAQE and QuAbS show outstanding performance. This is partly due to the combination of search-based and expansion-based reasoning, as discussed in Chapter 3.

**DQBF Solving.** In contrast to previous work [FKB12], we show that search-based solving is feasible for DQBF. This result is based on two essential enhancements: As $Q$-resolution [BCJ14a] is incomplete for DQBF, we need an improved proof system. We characterize fragments for which an extension of $Q$-resolution called Fork Resolution [Rab17] is complete and give an extension of Fork Resolution that is complete for DQBF. Further, compared to [FKB12], we use a less coarse over-approximation during solving. Lastly, we show that our DQBF solver DCAQE has comparable performance to QBF solving on the symbolic bounded synthesis encodings, which is in stark contrast to other state-of-the DQBF solvers.

## 1.4 Publications

This thesis is based on the following peer-reviewed publications:

[FT14b] Bernd Finkbeiner and Leander Tentrup. „Fast DQBF Refutation“. In: *Proceedings of SAT*. Vol. 8561. LNCS. Springer, 2014, pp. 243–251. DOI: 10.1007/978-3-319-09284-3_19.

[FT15] Bernd Finkbeiner and Leander Tentrup. „Detecting Unrealizability of Distributed Fault-tolerant Systems“. In: *Logical Methods in Computer Science* 11.3 (2015). DOI: 10.2168/LMCS-11(3:12)2015.

[RT15] Markus N. Rabe and Leander Tentrup. „CAQE: A Certifying QBF Solver“. In: *Proceedings of FMCAD*. IEEE, 2015, pp. 136–143. DOI: 10.1109/FMCAD.2015.7542263.

[Ten16] Leander Tentrup. „Non-prenex QBF Solving Using Abstraction“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 393–401. DOI: 10.1007/978-3-319-40970-2_24.

[Fay+17] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. „Encodings of Bounded Synthesis“. In: *Proceedings of TACAS*. Vol. 10205. LNCS. 2017, pp. 354–370. DOI: 10.1007/978-3-662-54577-5_20.

[FFT17] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. „BoSy: An Experimentation Framework for Bounded Synthesis“. In: *Proceedings of CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 325–332. DOI: 10.1007/978-3-319-63390-9_17.

[Ten17] Leander Tentrup. „On Expansion and Resolution in CEGAR Based QBF Solving“. In: *Proceedings of CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 475–494. DOI: 10.1007/978-3-319-63390-9_25.

[Fin+18] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. „Synthesizing Reactive Systems from Hyperproperties“. In: *Proceedings of CAV*. Vol. 10981. LNCS. Springer, 2018, pp. 289–306. DOI: 10.1007/978-3-319-96145-3_16.

[HT18] Jesko Hecking-Harbusch and Leander Tentrup. „Solving QBF by Abstraction“. In: *Proceedings of GandALF*. Vol. 277. EPTCS. 2018, pp. 88–102. DOI: 10.4204/EPTCS.277.7.

[Coe+19] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. „Verifying Hyperliveness“. In: *Proceedings of CAV*. Vol. 11561. LNCS. Springer, 2019, pp. 121–139. DOI: 10.1007/978-3-030-25540-4_7.

[Fin+19] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. „Synthesis from Hyperproperties“. Accepted for publication in Acta Informatica. 2019.

[Ten19] Leander Tentrup. „CAQE and QuAbS: Abstraction Based QBF solvers“. Accepted for publication in JSAT. 2019.

[TR19a] Leander Tentrup and Markus N. Rabe. „Clausal Abstraction for DQBF“. In: *Proceedings of SAT*. Vol. 11628. LNCS. Springer, 2019, pp. 388–405. DOI: 10.1007/978-3-030-24258-9_27.

Further, the thesis contains material published in the following report:

[TR19b] Leander Tentrup and Markus N. Rabe. „Clausal Abstraction for DQBF (full version)“. In: *CoRR* abs/1808.08759 (2019). arXiv: 1808.08759. URL: http://arxiv.org/abs/1808.08759.

## 1.5 Structure of This Thesis

This thesis is structured in two parts. The former gives an in-depth discussion about various reasoning techniques regarding the satisfiability problem for QBF and DQBF. The latter provides algorithms for the reactive synthesis problem, which are influenced by the insights learned in the first part. Both parts are sufficiently self contained to be read independently although in their development they heavily influenced each other, for example, by applying similar techniques as in the case for bounded unrealizability [Ten13; FT14a; FT15] and bounded unsatisfiability (Chapter 5), using the certification capabilities developed in Section 2.4 to synthesize reactive systems using bounded synthesis [Fay+17], and employing techniques introduced for non-CNF QBF solving to improve safety game solvers in Section 7.2.1.

### 1.5.1 Part I: Quantified Satisfiability

In the first part, we tackle the quest to solve the satisfiability problem for propositional logic extended with quantification. We consider different types of quantifiers, namely linear and branching quantifiers, different structures such as conjunctive normal form, negation normal form, prenex and non-prenex forms. Further, we provide means to extract certificates while solving as well as formal correctness proofs. Also, we provide a proof-theoretic view of our QBF solving algorithm that provided valuable insights into the strength and weaknesses, leading to significant empirical performance improvement.

### 1.5.2 Part II: Reactive Synthesis

In the second part, we consider the synthesis problem for temporal logic. We consider the *bounded synthesis* [FS13] approach where the synthesis problem is reduced to the satisfiability problem of a suitable constraint system. We show that we can derive “more symbolic” constraint systems by employing linear and branching quantifiers. Our evaluation shows that the symbolic variants have better scaling behavior than the non-symbolic ones in terms of the size of the inputs given by the environment and the number of states of the to-be-synthesized system.

Afterward, we consider the synthesis problems for HyperLTL and study expressiveness and decidability of fragments based on the quantifier prefix. Further, we give semi-decision procedures for the search for realizing implementations and counterexamples, respectively. We conclude with a synthesis case-study using benchmarks ranging from symmetry in hardware designs to fault-tolerance as well as secrecy.

# Part I

# Quantified Satisfiability

# Chapter 2

# Clausal Abstraction

Clausal abstraction is a solving method for quantified Boolean formulas that was independently developed by Janota & Marques-Silva [JM15b][1] and Rabe & Tentrup [RT15]. While initially only applicable to QBFs in prenex conjunctive normal form, there have been extensions to QBFs in negation normal form [HT18], parallelization [Ten16], satisfiability modulo theories [BJ15], quantified stochastic Boolean satisfiability [LWJ18], and dependency quantified Boolean formulas [TR19a]. The underlying idea of clausal abstraction is to assign variables, where the assignment order is determined by the quantifier prefix until either all clauses are satisfied or there is a set of clauses that cannot be satisfied at the same time. The effect of assignments, i.e., whether they satisfy a clause, is abstracted into one bit of information per clause, and this information is communicated through the quantifier prefix. The fundamental data structure of the algorithm is an abstraction, a propositional formula for each maximal block of quantifiers, that, given the valuation of outer variables, generates candidate assignments for the variables bound at this quantifier block. In case this candidate is refuted by inner quantifiers, the returned counterexample is excluded in the abstraction. Thus, the clausal abstraction algorithm uses ideas of search-based solving [GMN09], and counterexample guided abstraction refinement (CEGAR) algorithms [Cla+00]. A proof-theoretic analysis of the clausal abstraction approach [Ten17] has shown that the refutation proofs correspond to the (level-ordered) $Q$-resolution calculus [KKF95]. The implementation of the clausal abstraction algorithm in the solver CAQE won the prenex CNF track in the annual QBF competition QBFEVAL [NPT06; PS19] 2017, 2018, and 2019. Further, it was awarded a medal in the FLoC Olympic Games 2018[2].

This chapter gives a complete overview over the clausal abstraction approach for QBF, and it is based on an article accepted for publication in the journal of satisfiability (JSAT) [Ten19], which is itself partly based on work published in the proceedings of FMCAD [RT15] and CAV [Ten17]. The remainder of this chapter is structured as follows. After presenting the necessary preliminaries in Section 2.1, we give the algorithmic details for the clausal abstraction algorithm, first for the one-alternation fragment of QBF

[1] which they called *clause selection*

[2] http://www.floc2018.org/floc-olympic-games/

in Section 2.2 followed by the generalization to quantified Boolean formulas with arbitrary many quantifiers in Section 2.3. In Section 2.4, we show how function extraction is realized, and in Section 2.5, we integrate partial expansion reasoning in the clausal abstraction approach. We give a detailed experimental evaluation of the algorithm implemented in the QBF solver CAQE in Section 2.6 and conclude with a summary given in Section 2.7.

**Related Work.** QBF solving techniques can be roughly characterized by search-based and expansion-based methods. Solvers based on search assign variables in the order given by the quantifier prefix and progress by learning clauses and cubes for conflicts and solutions, respectively. Expansion-based solving methods eliminate quantifiers by rewriting the formula into propositional form. On the algorithmic side, many recent solving methods [Jan+16; RT15; JM15b; Jan18a; Jan18b; Blo+18] employ a variant of the CEGAR [Cla+00] style of reasoning to avoid an exponential blowup.

*Search-based Solving.* Search-based solvers typically extend algorithms for the propositional satisfiability (SAT) problem to the richer logic. An early example of such an extension is the algorithms implemented in the solvers Quaffle [ZM02] and QUBE++ [GNT04]. The proof system underlying search-based solvers is $Q$-resolution [KKF95], which extends propositional resolution with universal reduction. A more recent solver is DepQBF [LB10; LE17], which features a variety of other extensions such as Skolem and Herbrand function extraction [Nie+12], incremental solving [LE14], and inprocessing [Lon+15]. Qute [PSS17] is a search-based solver that learns dependencies between variables during the execution. The clausal abstraction approach [RT15], respectively, clause selection [JM15b], can be characterized as search-based as they assign variables contained in quantifier blocks simultaneously using a SAT oracle. While the difference between the basic algorithms of clausal abstraction and clause selection is minor [JM15b; RT15], there are several algorithmic improvements described for clausal abstraction [RT15; Ten16] that make the implementation CAQE outperform the clause selection solver Qesto as shown in the evaluation in Section 2.6.

There are further extensions of search-based methods to quantified Boolean formulas beyond conjunctive normal form [ESW09; GIB09; Kli+10; PSS17]. These methods typically exploit the duality of propositional formulas in negation normal form. Further approaches include using antichains as the underlying data structure [Bri+11] and using the duality of negation normal form to enhance CNF solving [GSB13]. The clausal abstraction approach has been generalized to QBFs in negation normal form [HT18] and non-prenex formulas [Ten16]. cQESTO [Jan18a] is a recently introduced circuit solver based on a similar algorithm as presented in this thesis. The algorithm, however, differs in the way abstractions are built: We produce a "static" abstraction upfront and learn subformula valuations during solving, while cQESTO evaluates the circuit under the current variable assignments and re-encodes the resulting partial circuit using the Tseitin transformation in each refinement step. To our knowledge, cQESTO cannot produce certificates.

Recently, incremental determinization [RS16; Rab+18] has been proposed as a search-based algorithm whose propagation mechanism is based on Boolean functions instead of variable assignments.

*Expansion-based Solving.* For expansion-based methods, one can further distinguish into complete and partial expansion. Complete expansion eliminates all universal quantifiers and rewrites the QBF to an equisatisfiable propositional formula. Design choices include the order of elimination, rewriting, and the representation of propositional formulas. Examples for complete expansion solvers are QUBOS [AB02], QUANTOR [Bie04], NENOFEX [LB08], and AIGSOLVE [SP16]. DYNQBF [CW16] is a recent solver that traverses a tree decomposition of a QBF instance and uses dynamic programming in conjunction with BDDs to solve sub-problems.

Partial expansion tries to expand only a subset of the possible universal assignments to show unsatisfiability (and dually, satisfiability). RAREQS [Jan+16] is a solver based on partial expansion that has later been extended to include refinements with strategies [Jan18b]. The underlying proof system, $\forall$Exp+Res [JM15a], first builds a partial expansion of the QBF and then uses propositional resolution on the expanded matrix. Recently, an algorithm based on partial expansion called IJTIHAD [Blo+18] was proposed that uses only two competing SAT solvers, whereas RAREQS uses one per quantifier block in the prefix.

*Hybrid Approaches.* Hybrid approaches combine both search-based and expansion-based reasoning, with different levels of integration. The search-based solver GHOSTQ [Kli+10] incorporates partial expansion reasoning [Jan+16]. HERETIC [Blo+18] is a lightweight integration of IJTIHAD and DEPQBF. The clausal abstraction solver CAQE has been extended to include partial expansion reasoning [Ten17]. What makes the hybrid approaches theoretically appealing and in practice performant is the fact that the proof systems underlying search, $Q$-resolution, and partial expansion, $\forall$Exp+Res, are incomparable with respect to polynomial simulation [JM15a; BCJ15], that is, neither does $Q$-resolution subsume $\forall$Exp+Res nor vice versa. Hence, a solver that combines both types of reasoning has a potential advantage over both expansion and search-based solvers [Ten17].

*Preprocessing.* Whereas this section is only concerned with complete solving techniques for quantified Boolean formulas, there is a rich body of literature regarding QBF preprocessing techniques. Further, our experiments in Section 2.6 show that preprocessing is an integral part of the performance characteristics of modern clausal QBF solvers, and this applies to clausal abstraction as well.

Blocked clause elimination is a common preprocessing technique, implemented (among other preprocessing techniques) in the tool BLOQQER [BLS11]. HQSPRE [Wim+17] is a preprocessor for both QBF and DQBF. Both also use (incomplete) universal expansion as well as variable elimination using resolution as preprocessing techniques. Recently, a new preprocessor QRATPRE+ [LE18b] was introduced, which is based on the QRAT calculus [HSB14a].

*Certification and Function Extraction.* The need for providing solving witnesses beyond binary answers is a research question that started with the very first QBF solving algorithms. The solver SKIZZO [Ben04] is one of the earliest QBF solvers that included certification [Ben05b]. An early certification format was proposed by Jussila et al. [Jus+07] and implemented for the solvers QUAFFLE and SQUOLEM. Balabanov and Jiang [BJ12] showed

how to extract Skolem and Herbrand functions from term-resolution and $Q$-resolution proofs, respectively. The QBFCert framework [Nie+12] is an implementation of this approach for the search-based solver DepQBF [LE17]. There have been further improvements to the extraction algorithm [BLJ16] and extensions to handle long-distance resolution [ELW13; Bal+15]. As long as there were solvers with certification capabilities, there are attempts to provide a unified framework [Ben05a; Jus+07; SM09] with the most recent one, QRAT [HSB14a], being the most promising as it was already successfully applied to preprocessing [HSB17]. To overcome the problem of missing preprocessing in the context of certification, there has been work that combines certificates produced by solver and preprocessor [JGM13; Faz+17]. For non-CNF solvers, there have also been methods for extracting Skolem and Herbrand functions [GGB11; Bri+11].

## 2.1 Quantified Boolean Formulas

### 2.1.1 Syntax

A quantified Boolean formula (QBF) [BB09] is a propositional formula over a finite set of variables $\mathcal{V}$ with Boolean domain $\mathbb{B} = \{\mathbf{F}, \mathbf{T}\}$ and quantification over variables. The syntax is given by the grammar

$$\varphi \coloneqq v \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists v.\, \varphi,$$

where $v \in \mathcal{V}$. Let $\mathcal{B}(V)$ be the set of quantified Boolean formulas over variables $V$. We use the usual Boolean connectives *conjunction* $\varphi \wedge \psi \coloneqq \neg(\neg\varphi \vee \neg\psi)$, *implication* $\varphi \to \psi \coloneqq \neg\varphi \vee \psi$, *equivalence* $\varphi \leftrightarrow \psi \coloneqq (\varphi \to \psi) \wedge (\psi \to \varphi)$, and *exclusion* $\varphi \oplus \psi \coloneqq \neg(\varphi \leftrightarrow \psi)$. Universal quantification $\forall v.\, \varphi$ is defined as $\neg\exists v.\, \neg\varphi$.

We denote the universally and existentially quantified variables as *universals* and *existentials*, respectively. To improve readability, we lift the consecutive quantification over variables of the same type to the quantification over sets of variables and denote $\mathcal{Q}v_1.\, \mathcal{Q}v_2 \ldots \mathcal{Q}v_n.\, \varphi$ by $\mathcal{Q}V.\, \varphi$ for $V = \{v_1, \ldots, v_n\}$ and $\mathcal{Q} \in \{\forall, \exists\}$. We assume w.l.o.g. that every variable $v \in \mathcal{V}$ is quantified at most once. A quantifier block $\mathcal{Q}v.\, \varphi$ for $\mathcal{Q} \in \{\exists, \forall\}$ *binds* the variable $v$ in the *scope* $\varphi$. Variables that are not bound by a quantifier are called *free*. We refer to the set of free variables of formula $\varphi$ as $\mathit{free}(\varphi)$. A *closed* QBF is a formula without free variables. Closed QBFs are either true or false. Every QBF can be transformed into a closed QBF while maintaining satisfiability by prepending the formula with existential quantifiers that bind the free variables. A formula is in prenex form if the formula consists of a quantifier prefix followed by a propositional, i.e., quantifier-free, formula. Every QBF can be transformed into prenex form while maintaining satisfiability. For a $k > 0$, a formula $\varphi$ is in the $k$QBF fragment if it is closed, in prenex form, and has exactly $k - 1$ alternations between $\exists$ and $\forall$ quantifiers.

A literal $l$ is either a variable $v \in V$, or its negation $\neg v$. The complement of a literal $l$, written $\overline{l}$, is defined as $\overline{l} = \neg v$ if $l = v$, and $\overline{l} = v$ if $l = \neg v$. Given a literal $l = v$ or $l = \neg v$, we define $\mathit{var}(l) = v$. Given a set of literals $\{l_1, \ldots, l_n\}$, the disjunctive combination $(l_1 \vee \ldots \vee l_n)$ is called a clause and the conjunctive combination $(l_1 \wedge \ldots \wedge l_n)$ is called a cube.

A QBF is in PRENEX CONJUNCTIVE NORMAL FORM (PCNF) if its propositional formula is a conjunction over clauses, i.e., in conjunctive normal form (CNF). We call the propositional part of a QBF in PCNF the MATRIX and we use $C_i$ to refer to clause $i$ of the matrix where unambiguous. For convenience, we treat clauses and matrices as a set of literals and clauses, respectively, and use the usual set operations for their manipulation. When given matrices, we typically omit the $\wedge$ operator between clauses. Every QBF in prenex form can be transformed into an equisatisfiable formula in PCNF using the Tseitin transformation [Tse68] with a linear increase in the size of the formula and number of existential variables.

**Example 2.1.** The following quantified Boolean formula

$$\exists v, w.\, \forall x.\, \exists y, z.\, (w \vee x \vee y)(v \vee \overline{w})(x \vee \overline{y})(\overline{v} \vee z)(\overline{z} \vee \overline{x})$$

is in the 3QBF fragment and its propositional part is in conjunctive normal form.

A QBF is in NEGATION NORMAL FORM (NNF) if negation is only applied to variables, that is, a formula in NNF contains only conjunctions, disjunctions, and literals. Every QBF can be transformed into NNF by at most doubling the size of the formula and without introducing new variables as it is the case for the Tseitin transformation.

**Example 2.2.** The following quantified Boolean formula

$$\exists x.\, \forall v, w.\, \exists y.\, (x \vee v \vee (y \wedge w)) \wedge (\overline{x} \vee (v \wedge \overline{w}) \vee y) \wedge (\overline{v} \vee w \vee \overline{y})$$

has two quantifier alternations and its propositional formula is in negation normal form.

### 2.1.2 Boolean Assignments and Functions

Given a subset of variables $V \subseteq \mathcal{V}$, a BOOLEAN ASSIGNMENT of $V$ is a function $\alpha\colon V \to \mathbb{B}$ that maps each variable $v \in V$ to either true ($\mathbf{T}$) or false ($\mathbf{F}$). We write $\alpha_V$ when the domain of $\alpha$, written $\mathrm{dom}(\alpha)$, is not clear from the context. A PARTIAL ASSIGNMENT $\beta\colon V \to \mathbb{B}_\perp$, where $\mathbb{B}_\perp \coloneqq \mathbb{B} \cup \{\perp\}$, may additionally set variables $v \in V$ to an undefined value $\perp$. We use the notation $\alpha^+$ and $\alpha^-$ to denote the partial assignment that retains positive and negative variable assignments, respectively. It is defined as

$$\alpha^+(v) = \begin{cases} \alpha(v) & \text{if } \alpha(v) = \mathbf{T} \\ \perp & \text{otherwise} \end{cases} \quad \text{and} \quad \alpha^-(v) = \begin{cases} \alpha(v) & \text{if } \alpha(v) = \mathbf{F} \\ \perp & \text{otherwise} \end{cases}$$

for every $v \in \mathrm{dom}(\alpha)$. We use the replacement operator $\beta_V[\perp \mapsto b]$ for $b \in \mathbb{B}$ to denote the assignment where undefined is replaced by a default value $b$. It is defined as

$$\beta_V[\perp \mapsto b](v) \coloneqq \begin{cases} \beta_V(v) & \text{if } \beta_V(v) \neq \perp \\ b & \text{otherwise} \end{cases}$$

for every $v \in V$. To restrict the domain of an assignment $\alpha$ to a set of variables $V$, we write $\alpha|_V$. For two assignments $\alpha$ and $\alpha'$ with domains $V = \mathrm{dom}(\alpha)$ and $V' = \mathrm{dom}(\alpha')$, we define the combination $\alpha \sqcup \alpha' \colon V \cup V' \to \mathbb{B}$ as

$$(\alpha \sqcup \alpha')(v) = \begin{cases} \alpha'(v) & \text{if } v \in V' \\ \alpha(v) & \text{otherwise} \end{cases} .$$

Note that $\alpha'$ overrides $\alpha$ for every element $v \in V \cap V'$ in the intersection of their domains. If the domains of $\alpha$ and $\alpha'$ are disjoint, that is, $\mathrm{dom}(\alpha) \cap \mathrm{dom}(\alpha') = \varnothing$, we denote the combination by $\alpha \dot{\sqcup} \alpha'$. For two partial assignments $\beta_V$ and $\beta'_V$, we define the intersection operation $\beta_V \sqcap \beta'_V \colon V \to \mathbb{B}_\bot$ as

$$(\beta_V \sqcap \beta'_V)(v) = \begin{cases} \beta_V(v) & \text{if } \beta_V(v) = \beta'_V(v) \\ \bot & \text{otherwise} \end{cases} .$$

We define the *complement* $\overline{\alpha}$ to be $\overline{\alpha}(v) = \neg\alpha(v)$ for all $v \in \mathrm{dom}(\alpha)$. The complement of a partial assignment is defined analogously with $\neg\bot = \bot$. We use the notation $\varphi[\alpha]$ to replace variables $v \in \mathrm{dom}(\alpha)$ with their assignments $\alpha(v)$. We denote by $\alpha_V^b \coloneqq \{v \in V \mid \alpha_V(v) = b\}$ the subset of variables that are assigned to $b \in \mathbb{B}$, i.e., the preimage of $\alpha_V$ with respect to $b$. The *set of assignments* and the *set of partial assignments* of $V$ is denoted by $\mathcal{A}(V)$ and $\mathcal{A}_\bot(V)$, respectively.

A Boolean function $f \colon \mathcal{A}(V) \to \mathbb{B}$ maps *assignments* of $V$ to true or false. An assignment $\alpha_V$ over variables $V$ can be represented by the conjunctive formula $\bigwedge_{v \in \alpha_V^{\mathrm{T}}} v \wedge \bigwedge_{v \in \alpha_V^{\mathrm{F}}} \neg v$, that is, the only assignment over variables $V$ that satisfy this formula is the assignment $\alpha_V$. Similarly, Boolean functions can be represented by propositional formulas over the variables in their domain. Let $\varphi[f_v]$ be the formula where occurrences of $v$ are replaced by the propositional representation of $f_v$. It is defined as

$$\begin{aligned} x[f_v] &= \begin{cases} f_v & \text{if } v = x \\ x & \text{otherwise} \end{cases} \\ (\neg\varphi)[f_v] &= \neg(\varphi[f_v]) \\ (\varphi \vee \psi)[f_v] &= (\varphi[f_v]) \vee (\psi[f_v]) \\ (\exists x.\, \varphi)[f_v] &= \begin{cases} \varphi[f_v] & \text{if } v = x \\ \exists x.\, (\varphi[f_v]) & \text{otherwise} \end{cases} \end{aligned}$$

For example, let $\varphi = \forall x.\, \exists y.\, (x \vee \neg y) \wedge (\neg x \vee y)$ and let $f_y(x) = x$, then $\varphi[f_y] = \forall x.\, (x \vee \neg x) \wedge (\neg x \vee x)$. We use a function $g \colon \mathcal{A}(X) \to \mathcal{A}(Y)$ that maps assignments of $X$ to assignments of $Y$ to represent multiple Boolean functions and define the replacement operator $\varphi[g]$ accordingly. For example, given another Boolean function $f_{y'} \colon \mathcal{A}(\{x\}) \to \mathbb{B}$, the combination with $f_y$ is $g_{y,y'} \colon \mathcal{A}(\{x\}) \to \mathcal{A}(\{y, y'\})$ such that $g_{y,y'}(x) = \{y \mapsto f_y(x), y' \mapsto f_{y'}(x)\}$.

### 2.1.3 Semantics

We fix a set of variables $V \subseteq \mathcal{V}$. The satisfaction relation $\models \subset \mathcal{A}(V) \times \mathcal{B}(V)$ is defined as

$$
\begin{array}{lll}
\alpha \models v & \text{iff} & \alpha(v) = \mathbf{T} \\
\alpha \models \neg\varphi & \text{iff} & \alpha \not\models \varphi \\
\alpha \models \varphi \vee \psi & \text{iff} & \alpha \models \varphi \text{ or } \alpha \models \psi \\
\alpha \models \exists v.\, \varphi & \text{iff} & \text{there exists some } \alpha' \colon \mathcal{A}(\{v\}) \to \mathbb{B} \text{ such that } \alpha \dot\cup \alpha' \models \varphi
\end{array}
$$

QBF SATISFIABILITY is the problem to determine, for a given QBF $\Phi$, the existence of an assignment $\alpha$ for the free variables $\mathit{free}(\Phi)$ such that the relation $\models$ holds. In this case, we call $\alpha$ a satisfying assignment and say that $\alpha$ satisfies $\Phi$. If $\alpha \not\models \Phi$, we say that $\alpha$ falsifies $\Phi$. For a closed-form QBF $\Phi$, the QBF satisfiability problem is equivalent to the validity problem, which asks if all assignments satisfy $\Phi$, as the problem reduces to checking whether $\{\} \models \Phi$ where $\{\}$ denotes the empty assignment. For formulas in prenex form with propositional formula $\varphi$, the QBF satisfiability problem can be interpreted as a two-player game: Based on the order of quantifiers given by the quantifier prefix, the existential player $\exists$ chooses assignments of existential variables with the aim to satisfy $\varphi$, while the universal player $\forall$ chooses assignment of universal variables in order to falsify $\varphi$. The satisfiability game is *determined*, that is, for every QBF, either the existential player or the universal player has a winning strategy.

For satisfiable QBFs the winning strategy for the existential player is called a SKOLEM FUNCTION $f \colon \mathcal{A}(V_\forall) \to \mathcal{A}(V_\exists)$ which maps assignments of universal variables $V_\forall$ to assignments of existential variables $V_\exists$, such that $\varphi[f]$ is valid. For unsatisfiable QBFs, the winning strategies are defined dually, i.e., $f \colon \mathcal{A}(V_\exists) \to \mathcal{A}(V_\forall)$ such that $\varphi[f]$ is unsatisfiable, and are called HERBRAND FUNCTIONS . Intuitively, Skolem and Hebrand functions are well-formed if every assigned variable depends solely on its dependencies as given by the quantifier prefix. We formalize this intuition in the following using the concept of *dependencies* and *consistency*.

An existentially quantified variable $v$ *depends* on all universally quantified variables that are bound before $v$. A universally quantified variable $v$ *depends* on all existentially quantified variables bound prior to $v$ as well as the free variables. A free variable $v$ has no dependencies, i.e., can only be instantiated by constants. The set of dependencies of a variable $v \in V$ is denoted by $\mathit{dep}(v)$. For a set of variables $V$, we define $\mathit{dep}(V)$ as the union over the dependencies $\bigcup_{v \in V} \mathit{dep}(v)$.

A function $f_X$ is WELL-FORMED if the assignments are *consistent* with respect to the dependencies of $X$, i.e., for every $x \in X$ and every pair of assignments $\alpha_V$ and $\alpha'_V$ with $V = \mathit{dep}(X)$ and $\alpha_V|_{\mathit{dep}(x)} = \alpha'_V|_{\mathit{dep}(x)}$ it holds that $f_X(\alpha_V)(x) = f_X(\alpha'_V)(x)$. In other words, $f_X$ has to produce the same output for $x \in X$ if the dependencies of $x$ are the same.

## 2.2 Solving QBF with One Quantifier Alternation

We start the description of the clausal abstraction algorithm by considering only the one-alternation fragment of QBF, called 2QBF. In this fragment, the existential variables have

**Algorithm 2.1** Clausal Abstraction Algorithm for 2QBF

```
1:  procedure SOLVE_∀∃(∀X. ∃Y. φ)
2:      initialize abstractions θ_X and θ_Y with shared variables S = {s_i | C_i ∈ φ}
3:      loop
4:          match SAT(θ_X, {}) as
5:              Unsat(_) ⇒ return Sat
6:              Sat(α) ⇒
7:                  match SAT(θ_Y, α|_S) as
8:                      Unsat(_) ⇒ return Unsat(α|_X)          ▷ α|_X ⊭ ∃Y. φ
9:                      Sat(_) ⇒ θ_X ← θ_X ∧ ⋁_{s∈(α|_S)^T} s̄     ▷ refine θ_X
10:     end loop
11: end procedure
```

*complete information*, i.e., they depend on the complete set of universal variables. The reasons for choosing 2QBF as a starting point are manifold; it is in some sense the simplest extension of propositional logic that includes quantification and allows us to introduce the core ideas, notation, and terminology behind the clausal abstraction algorithm. After discussing the restricted fragment, we generalize the algorithm to arbitrary quantifier alternations in Section 2.3. For this section, we fix some 2QBF $\forall X.\, \exists Y.\, \varphi$ with universal variables $X$, existential variables $Y$, and matrix $\varphi$.

### 2.2.1 Algorithm

**Preliminaries.** We use a generic solving function $\textsc{sat}(\theta, \alpha)$ for propositional formula $\theta$ and assignment $\alpha$, that returns whether $\theta \wedge \alpha$ is satisfiable. In the positive case, it returns $\mathrm{Sat}(\alpha')$, where $\alpha'$ is a satisfying assignment of $\theta$ with $\alpha \sqsubseteq \alpha'$. In the negative case, it returns $\mathrm{Unsat}(\beta)$, where $\beta \sqsubseteq \alpha$ is a partial assignment such that $\theta \wedge \beta$ is unsatisfiable.

In the following algorithms, we make use of pattern matching on well-structured objects, such as the result of the call to SAT and the quantifier prefix of quantified Boolean formulas. For example, to determine the leading quantifier of some QBF $\Phi$, we write

```
match Φ as
    ∃X. Ψ ⇒ [...]      ▷ leading existential quantifier
    ∀X. Ψ ⇒ [...]      ▷ leading universal quantifier
```

Additionally, we allow wildcards, denoted by "_", in match arms.

**Overview.** The clausal abstraction algorithm is based on the idea of using two competing SAT solvers, one for the universal quantifier that tries to falsify clauses and one for the existential quantifier that has to satisfy the remaining clauses in the matrix. The algorithm $\textsc{solve}_{\forall\exists}$ is shown in Algorithm 2.1. After initializing the abstractions, which is detailed below, the algorithm repeatedly solves $\theta_X$ using a SAT solver. $\theta_X$ contains variables $X$ and satisfaction variables $S$, one variable $s_i \in S$ for every clause $C_i \in \varphi$ that represents whether this clause is satisfied by an assignment $\alpha_X$ of variables $X$. Every as-

signment $\alpha$ with $\alpha \vDash \theta_X$ is a combination of an assignment $\alpha|_X$ of variables $X$ and an assignment $\alpha|_S$ of variables $S$. In the following SAT call to $\theta_Y$, the assignment $\alpha|_S$ representing satisfied clauses is assumed. In case $\theta_Y[\alpha|_S]$ is satisfiable, we found a matching $Y$ assignment to the given $X$ assignment, thus, the abstraction $\theta_X$ is refined and the algorithm proceeds with the next iteration. The algorithm terminates, returning satisfiable and unsatisfiable, if the SAT call to $\theta_X$ and $\theta_Y$ is unsatisfiable, respectively. In the former case, we have depleted all universal assignments and in the latter case there is an assignment $\alpha_X$ such that there is no matching $Y$ assignment.

**Abstractions $\theta_X$ and $\theta_Y$.** The abstraction is the core data structure of the algorithm, representing, for each player, an over-approximation of the winning assignments and the resulting effect those assignments have on the satisfaction of clauses. The abstraction $\theta_Y$ represents the winning assignments $\alpha_Y$ of the existential player under the condition that a certain set of clauses is already satisfied by the prior universal assignment $\alpha_X$. Thus, $\theta_Y$ is satisfiable if, and only if, every clause in the matrix is satisfied, either by an assignment to $Y$ or by an assignment of the outer universal variables. For universal quantifier $\forall X$, the abstraction $\theta_X$ represents which clauses are satisfied with respect to an assignment to $X$. During the execution of the algorithm, we learn that the universal player cannot falsify $\varphi$ when a certain set of clauses is satisfied by $\alpha_X$, thus, we refine $\theta_X$ to make sure that one of the previously satisfied clauses is falsified, which eliminates losing assignments $\alpha_X$.

The interaction between $\theta_X$ and $\theta_Y$ is established by a common set of *clause satisfaction* variables $S$, one variable $s_i \in S$ for every clause $C_i \in \varphi$. Given an assignment $\alpha_X$ and some clause $C_i \in \varphi$, we guarantee that $s_i$ is assigned to true if $\alpha_X \vDash C_i|_X$. Thus, if $s_i$ is assigned to false, the existential quantifier has to satisfy clause $C_i$. We define the abstraction that implements those requirements for a clause $C_i \in \varphi$ as

$$clabs_{\forall X}(C_i) \coloneqq s_i \vee \neg C_i|_X = \bigwedge_{l \in C_i|_X} \overline{l} \vee s_i \quad \text{and} \tag{2.1}$$

$$clabs_{\exists Y}(C_i) \coloneqq s_i \vee C_i|_Y \tag{2.2}$$

for universal and existential quantifier, respectively. The clausal abstraction for the universal quantifier block $\forall X$ and the existential quantifier block $\exists Y$ is defined as

$$\theta_X \coloneqq \bigwedge_{C_i \in \varphi} clabs_{\forall X}(C_i) \qquad \text{and} \qquad \theta_Y \coloneqq \bigwedge_{C_i \in \varphi} clabs_{\exists Y}(C_i) \ . \tag{2.3}$$

Lastly, a refinement for $\theta_X$ ensures that from a set of clauses that was previously satisfied ($s_i$ set to true) one is falsified, thus we add the clause

$$\bigvee_{s \in \alpha_S^{\mathbf{T}}} \overline{s} \tag{2.4}$$

to the abstraction $\theta_X$. We conclude the description of the algorithm by a detailed example. In the following section, we show that the algorithm correctly determines the result of the satisfiability problem for 2QBF.

**Example 2.3.** Consider the following 2QBF

$$\forall x.\, \exists y, z.\, (x \vee z)(\overline{x} \vee \overline{y})(\overline{x} \vee y \vee z)(\overline{z} \vee \overline{x}).$$

By the definitions above, the resulting abstractions are

$$\theta_{\{x\}} = (s_1 \vee \overline{x})(s_2 \vee x)(s_3 \vee x)(s_4 \vee x) \text{ and}$$
$$\theta_{\{y,z\}} = (s_1 \vee z)(s_2 \vee \overline{y})(s_3 \vee y \vee z)(s_4 \vee \overline{z}) \ .$$

We show a possible execution of SOLVE$_{\forall\exists}$ on the example formula:

- SAT$(\theta_{\{x\}}, \{\}) = \mathsf{Sat}(\{x \mapsto \mathbf{F}, s_1 \mapsto \mathbf{F}, s_2 \mapsto \mathbf{T}, s_3 \mapsto \mathbf{T}, s_4 \mapsto \mathbf{T}\})$
- SAT$(\theta_{\{y,z\}}, \{s_1 \mapsto \mathbf{F}, s_2 \mapsto \mathbf{T}, s_3 \mapsto \mathbf{T}, s_4 \mapsto \mathbf{T}\}) = \mathsf{Sat}(\{z \mapsto \mathbf{T}, y \mapsto \mathbf{F}\})$
- $\theta'_{\{x\}} = \theta_{\{x\}} \wedge (\overline{s_2} \vee \overline{s_3} \vee \overline{s_4})$
- SAT$(\theta'_{\{x\}}, \{\}) = \mathsf{Sat}(\{x \mapsto \mathbf{T}, s_1 \mapsto \mathbf{T}, s_2 \mapsto \mathbf{F}, s_3 \mapsto \mathbf{F}, s_4 \mapsto \mathbf{F}\})$
- SAT$(\theta_{\{y,z\}}, \{s_1 \mapsto \mathbf{T}, s_2 \mapsto \mathbf{F}, s_3 \mapsto \mathbf{F}, s_4 \mapsto \mathbf{F}\}) = \mathsf{Unsat}$
- SOLVE$_{\forall\exists}$ returns $\mathsf{Unsat}(\{x \mapsto \mathbf{T}\})$

### 2.2.2 Correctness

The correctness argument relates variable assignments to assignments of the satisfaction variables $S$. We start by stating two properties over the abstractions $\theta_X$ and $\theta_Y$ that immediately follow from their definitions.

**Lemma 2.4.** *Let $\alpha$ be a satisfying assignment of $\theta_X$ and let $\alpha_S$ be some arbitrary assignment over variables $S$. It holds that*

1. $\alpha(s_i) = \mathbf{F} \Rightarrow \alpha|_X \not\models C_i|_X$ *for every clause* $C_i \in \varphi$ *and*
2. $\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} C_i|_Y$.

For termination, we need to argue that the main loop in Algorithm 2.1 cannot be executed infinitely often. We give an implicit ranking function, based on the following observations. First, the number of different refinements, i.e., clauses over $S$, is bounded by the number of variables in $S$. Second, during the execution of the algorithm, every refinement clause (line 9) is different, that is, it is impossible that two refinements are the same.

**Lemma 2.5.** *There are only finitely many different refinement clauses and the refinements during the execution of Algorithm 2.1 are pairwise different.*

*Proof.* The number of different refinement clauses is bounded by the number of subsets of $S$ by the definition in Equation 2.4, i.e., are at most $2^{|S|}$ different refinement clauses. Assume for contradiction that there is an execution of the algorithm that produces the same refinement clause $R$, thus, according to line 9 of Algorithm 2.1 there are two assignments $\alpha$ and $\alpha'$ such that $(\alpha|_S)^1 = (\alpha'|_S)^1$. It holds that $R = \bigvee_{s \in (\alpha|_S)^{\mathbf{T}}} \overline{s}$ and thus $\alpha' \not\models R$. As $\theta_X$ contains the clause $R$ after the refinement with $\alpha$ and $\alpha'$ satisfies $\theta_X$, we derive a contradiction. □

Given those lemmas, we can prove the correct termination for true formulas.

**Theorem 2.6.** *If* $\forall X.\, \exists Y.\, \varphi$ *is true, Algorithm 2.1 returns* Sat.

*Proof.* Let $\forall X.\, \exists Y.\, \varphi$ be true. By the QBF semantics, there is a Skolem function $f_Y : \mathcal{A}(X) \to \mathcal{A}(Y)$ such that $\varphi[f_Y]$ is valid. Let $\alpha_X$ and $\alpha_S$ be arbitrary assignments satisfying $\theta_X$ (line 4). By Lemma 2.4, it holds that

$$\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} C_i|_Y \quad \subseteq \quad \bigwedge_{\substack{C_i \in \varphi \\ \alpha_X \not\models C_i|_X}} C_i|_Y = \varphi[\alpha_X] \ .$$

Hence, $\alpha_Y \coloneqq f_Y(\alpha_X)$ is a satisfying assignment for $\theta_Y[\alpha_S]$ as it satisfies $\varphi[\alpha_X]$. The formula $\theta_Y[\alpha_S]$ in line 7 is, thus, always satisfiable and the return in line 8 is unreachable. Termination is guaranteed by Lemma 2.5. □

For the reverse direction, we need additional properties regarding the refinement operation that we state in the following. Let $\alpha_X$ be an assignment and let $\theta_X$ and $\theta'_X$ be the abstraction before and after the refinement with some assignment $\alpha_S$, respectively. We say that $\alpha_X$ is *excluded* from $\theta_X$ if $\theta'_X[\alpha_X]$ is unsatisfiable whereas $\theta_X[\alpha_X]$ is satisfiable.

**Lemma 2.7.** *If an assignment* $\alpha_X$ *is excluded from* $\theta_X$ *by a refinement with* $\alpha_S$*, it holds that* $\alpha_S(s_i) = \mathbf{T}$ *implies that* $\alpha_X \models C_i|_X$ *for every* $C_i \in \varphi$.

*Proof.* Let $\alpha_X$ and $\alpha_S$ be assignments such that $\alpha_X$ is excluded from $\theta_X$ by a refinement with $\alpha_S$, that is, $\theta_X[\alpha_X]$ is satisfiable and the refinement clause $\psi = \bigvee_{s \in \alpha_S^{\mathbf{T}}} \bar{s}$ (line 9 of Algorithm 2.1) excludes $\alpha_X$, i.e., $\theta'_X = \theta_X \wedge \psi$ and $\theta'_X[\alpha_X]$ is unsatisfiable. $\theta'_X$ entails $\psi' \coloneqq \bigvee_{s_i \in \alpha_S^{\mathbf{T}}} \neg C_i|_X$ (see the definition of the universal abstraction in Equation 2.1) and it holds that $\alpha_X \not\models \psi'$ (assuming otherwise would contradict that $\theta'_X[\alpha_X]$ is unsatisfiable). Thus, it holds that $\alpha_X \models \bigwedge_{s_i \in \alpha_S^1} C_i|_X$. □

**Theorem 2.8.** *If* $\forall X.\, \exists Y.\, \varphi$ *is false, Algorithm 2.1 returns* $\mathrm{Unsat}(\alpha_X)$ *where* $\varphi[\alpha_X]$ *is unsatisfiable.*

*Proof.* Let $\forall X.\, \exists Y.\, \varphi$ be false. By the QBF semantics, there exists some assignment $\alpha_X$ such that $\varphi[\alpha_X]$ is unsatisfiable. Let $\alpha_S$ be the assignment such that $\alpha_S(s_i) = \mathbf{T}$ if, and only if, $\alpha_X \models C_i|_X$ for every $C_i \in \varphi$. The combined assignment $\alpha_S \cup \alpha_X$ is a satisfying assignment for $\theta_X$ in line 4. It holds that $\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} C_i|_Y = \varphi[\alpha_X]$ by the definition of the existential abstraction. As $\varphi[\alpha_X]$ is unsatisfiable, $\theta_Y[\alpha_S]$ is unsatisfiable as well, leading to the return in line 8.

To conclude the proof, it remains to show that this $\alpha_X$ is not excluded by some refinement in line 9. Assume for contradiction that it is excluded by some assignment $\alpha_S$, i.e., by Lemma 2.7 for every $C_i \in \varphi$ it holds that $\alpha_S(s_i) = \mathbf{T} \Rightarrow \alpha_X \models C_i|_X$ which is equivalent to $\alpha_X \not\models C_i|_X \Rightarrow \alpha_S(s_i) = \mathbf{F}$. It holds that

$$\varphi[\alpha_X] = \bigwedge_{\substack{C_i \in \varphi \\ \alpha_X \not\models C_i|_X}} C_i|_Y \quad \subseteq \quad \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} C_i|_Y = \theta_Y[\alpha_S] \ ,$$

thus, from the unsatisfiability of $\varphi[\alpha_X]$ follows that $\theta_Y[\alpha_S]$ is unsatisfiable as well, contradicting the refinement of $\alpha_S$. As there are only finitely many different refinements, the query in line 4 eventually returns the assignment $\alpha_X$ or some other unsatisfying assignment. □

### 2.2.3 Optimizations

In this section, we investigate improvements to the algorithm stated in Algorithm 2.1. Those improvements fall in two categories. The first category is concerned with simplifying the propositional abstractions with the intention to improve the satisfiability check and the second category is concerned with potentially reducing the number of iterations of the algorithm.

**Abstraction Improvements.** In the case that a clause $C_i \in \varphi$ contains only existential quantified variables, $s_i$ can be assumed to be false. Thus, we can modify the definitions of $\mathit{clabs}_{\mathcal{Q}}$, given in Equation 2.1 and Equation 2.2 to $\mathit{clabs}_{\forall}(X, C_i) = \mathbf{T}$ and $\mathit{clabs}_{\exists}(Y, C_i) = C_i$ if $C_i|_Y = C_i$. Note that in this case, the variable $s_i$ does not appear in the abstraction.

Balabanov et al. [Bal+16a] describe two further simplifications for the universal abstraction:

- If some clause $C_i \in \varphi$ is a universal unit clause, i.e., $C|_X = \{l\}$ for some literal $l$ with $\mathit{var}(l) \in X$, then the shared variable $s_i$ can be replaced by the negation $\overline{l}$ of the literal.
- If there is a pair of clauses $C_i, C_j \in \varphi$ with $i \neq j$ such that those clauses are equal with respect to universal literals, i.e., $C_i|_X = C_j|_X$, then the same shared variable $s_i$ can be used for both clauses.

Lastly, one can use the knowledge of the objective of the universal quantifier to improve assignments $\alpha_S$. As the variables $S$ occur pure in the abstraction $\theta_X$, the SAT solver may set all of them to false initially. For SAT solver that support setting a default polarity on decisions, this can be used to improve the initial assignment. Alternatively, the problem could be reformulated as a maximum satisfiability (MaxSAT) optimization problem.

**Algorithmic Improvements.** Given assignments $\alpha_S$ and $\alpha_X$ from the SAT solver in line 4, $\alpha_S$ may not be optimal in the following sense: There can be some clause $C_i \in \varphi$ where $\alpha_S(s_i) = \mathbf{T}$ but the assignment $\alpha_X$ does not satisfy $C_i|_X$, i.e., $\alpha_X \vDash \neg C_i|_X$. This is due to the implication in the definition of $\mathit{clabs}_{\forall}$ in Equation 2.1. We circumvent this by applying a step after line 4 that optimizes $\alpha_S$ with respect to $\alpha_X$, i.e., we set $\alpha_S(s_i) = \mathbf{F}$ for every clause $C_i \in \varphi$ where $\alpha_X \vDash \neg C_i|_X$. This change is also compatible with the correctness proof in the previous section, especially Lemma 2.4 still holds after the $\alpha_S$ optimization.

Given satisfying assignments $\alpha_S$ and $\alpha_Y$ from the SAT solver in line 7, the assignment $\alpha_Y$ may satisfy clauses that are not required by the assignment $\alpha_S$, that is, the clause is already satisfied by the universal variable assignment represented by $\alpha_S$. This is the case if there is some clause $C_i \in \varphi$ with $\alpha_Y \vDash C_i|_Y$ and $\alpha_S(s_i) = \mathbf{T}$. We use this information to

**Algorithm 2.2** Clausal Abstraction Algorithm for QBF

1: **procedure** SOLVE($\Phi$)
2: initialize abstraction $\theta_X$ for every quantifier block $\mathcal{Q}X$ in $\Phi$
3: **match** $\Phi$ **as**
4: $\exists X.\, \Psi \Rightarrow$ **return** SOLVE$_\exists(X, \Psi, \{s_i \mapsto \mathbf{F} \mid C_i \in \varphi\})$
5: $\forall X.\, \Psi \Rightarrow$ **return** SOLVE$_\forall(X, \Psi, \{s_i \mapsto \mathbf{F} \mid C_i \in \varphi\})$
6: **end procedure**

improve the refinement clause in Equation 2.4: For every such clause $C_i$, we set $\alpha_S(s_i) = \mathbf{F}$, thus, reducing the number of literals in the refinement clause.

Another enhancement greedily flips variable assignments in $\alpha_Y$ if the resulting assignment satisfies strictly more clauses. Let $y \in Y$ be some existential variable. We can flip the value of $y$ if $\varphi[\alpha_Y \sqcup \{y \mapsto \neg\alpha_Y(y)\}] \subseteq \varphi[\alpha_Y]$. This greedy flipping may further improve the effect of the previous optimization.

Balabanov et al. [Bal+16a] noticed that under certain conditions, one can remove literals from the refinement clause in Equation 2.4. If there are two clauses $C_i$ and $C_j$ with $C_i|_X \subseteq C_j|_X$ and $\alpha_S(s_i) = \alpha_S(s_j) = \mathbf{T}$, then we can set $\alpha_S(s_j) = \mathbf{F}$, which removes $\overline{s_j}$ from the refinement clause. This is due to the implications $\overline{s_i} \rightarrow \neg C_i|_X$ and $\overline{s_j} \rightarrow \neg C_j|_X$ in the universal abstraction, $clabs_\forall$ in Equation 2.1, and the implication $\neg C_j|_X = \bigwedge_{l \in C_j|_X} \neg l \Rightarrow \bigwedge_{l \in C_i|_X} \neg l = \neg C_i|_X$ due to the fact that the literals in $C_j|_X$ are a superset of the literals in $C_i|_X$.

The algorithmic optimizations are crucial for the performance of the algorithm as they circumvent non-optimal assignments to satisfaction variables resulting from using implications in the definition of the abstraction (compared to the equivalences used in clause selection [JM15b]).

## 2.3 Solving QBF with Arbitrary Quantifier Alternations

We now generalize the 2QBF algorithm to QBFs with an arbitrary number of alternations by providing an algorithm that does recursion on the quantifier prefix. The main insight in this generalization is that the existential player now has a choice to either directly satisfy a clause or assume that an inner quantifier block will satisfy it. For this section, we fix some quantified Boolean formula $\Phi$ in closed prenex conjunctive normal form (PCNF) with matrix $\varphi$. We assume that $\Phi$ is universally reduced, that is, for every clause $C_i \in \varphi$ and every universal literal $l_\forall \in C_i$, there is an existential literal $l_\exists \in C_i$ that depends on $l_\forall$, formally $var(l_\forall) \in dep(var(l_\exists))$. If this property is violated for some clause $C_i$ and literal $l_\forall \in C_i$, then $l_\forall$ can be removed from $C_i$, which is called universal reduction [KKF95].

### 2.3.1 Algorithm

**Overview.** The overall approach of the algorithm is to construct a propositional formula $\theta_X$ for every quantifier block $\mathcal{Q}X$ that represents an over-approximation of the winning assignments $\alpha_X$ and the effect of those assignments on the matrix, that is, which clauses are satisfied and falsified for existential and universal quantifiers, respectively. The main algorithm SOLVE is depicted in Algorithm 2.2. It takes as an input a quantified Boolean formula $\Phi$, initializes the abstraction for every quantifier block of $\Phi$, and then returns the result of the call to $\text{SOLVE}_\exists$ or $\text{SOLVE}_\forall$, shown in Algorithm 2.3 and 2.4, depending on the type of the leading quantifier block. The algorithm $\text{SOLVE}_\mathcal{Q}(X, \Psi, \alpha_S)$ determines whether the quantified subformula $\mathcal{Q}X.\,\Psi$ is satisfiable under the condition that some clauses are already satisfied by assignments to variables bound at outer quantifiers (represented by $\alpha_S$ as discussed below).

The algorithms for quantified subformulas, $\text{SOLVE}_\mathcal{Q}$, determine candidate assignments to the variables bound at that quantifier that meet the quantifier's objective (to satisfy and falsify the formula for $\exists$ and $\forall$ quantifier, respectively), or give a reason why there is no such assignment. If a quantifier is able to provide a candidate assignment, it is recursively verified by proceeding to the inner quantified subformula. A *conflict* occurs when the current assignment of variables definitely violates some clause (existential conflict) or satisfies all clauses (universal conflict). In case of such a conflict, the reason for this conflict is excluded at an outer quantifier level by refining the corresponding abstraction.

**Abstraction $\theta$.** The formula $\theta$ represents, for every quantifier block, how the quantifier blocks's variables interact with the assignments of variables of other quantifier blocks. The algorithm guarantees that whenever a candidate assignment is generated, all variables bound at outer quantifier levels have a fixed assignment, and thus some (possibly empty) set of clauses is already satisfied. At an existential quantifier, the corresponding player then tries to satisfy more clauses with an assignment to the variables bound at this quantifier, while the universal player tries to find an assignment that make it harder to satisfy all clauses.

As in the case for 2QBF in the previous section, the interaction of abstractions is established by the clause satisfaction variables $S$ with the same semantics as before, i.e., given some quantifier block $\mathcal{Q}X$ and assignment $\alpha_V$ of outer variables $V$ (w.r.t. $\mathcal{Q}X$), for every clause $C_i \in \varphi$ the satisfaction variable $s_i \in S$ represents whether $C_i$ is satisfied by $\alpha_V$. This, however, is not enough for existential quantifiers as the existential player has the choice to either satisfy the clause or *assume* that the clause will be satisfied by an assignment of an inner quantifier. Thus, we add an additional set of variables $A$ for every existential quantifier block $\exists X$, called *assumption variables*, with the intended semantics that variable $a_i$ is set to false implies that the clause $C_i$ is satisfied at this quantifier level (either by an assignment to $X$ or an outer assignment $\alpha_V$ abstracted by $\alpha_S$).

We are now going to define the abstraction that implements this intuition. Fix some quantifier block $\mathcal{Q}X$. To define the abstraction for $\mathcal{Q}X$, we split a clause $C_i$ into three

parts,

$$C_i^< \coloneqq \{l \in C_i \mid l \text{ bound before } \mathcal{Q}X\},$$
$$C_i^= \coloneqq \{l \in C_i \mid var(l) \in X\}, \text{ and}$$
$$C_i^> \coloneqq \{l \in C_i \mid l \text{ bound after } \mathcal{Q}X\} \ .$$

By definition, it holds that $C_i = C_i^< \cup C_i^= \cup C_i^>$.

For existential quantifiers, a clause $C_i$ is encoded by a variable $s_i$ that represents whether the clause $C_i$ is *satisfied* by an assignment to variables outer to $Y$, the literals of the quantifiers's variables, and a variable $a_i$ that indicates whether the clause is *assumed* to be satisfied by an inner assignment. For existential quantifier $\exists X$, the clausal abstraction for clause $C_i \in \varphi$ is defined as

$$clabs_{\exists X}(C_i) = \begin{cases} s_i \vee C_i^= & \text{if } \exists X \text{ is the innermost quantifier} \\ s_i \vee C_i^= \vee a_i & \text{otherwise} \end{cases} \tag{2.5}$$

During the execution of the algorithm, the algorithm potentially visits each quantifier multiple times to generate candidate assignments and assumptions. If those assumptions turn out to be wrong, that is, the corresponding assignment is losing for the existential player, the abstraction is refined. Such a refinement is a clause that contains only assumption variables $A$ and represents sets of clauses that together cannot be satisfied by the inner quantifier.

The abstraction for universal quantifiers $\forall X$ is unchanged from the 2QBF algorithm, that is, we define

$$clabs_{\forall X}(C_i) = s_i \vee \neg C_i^= = \bigwedge_{l \in C_i^=} \overline{l} \vee s_i \ . \tag{2.6}$$

In contrast to existential quantifiers, universal quantifiers do not have separate sets of variables $S$ and $A$; to define the abstraction we use only satisfaction variables $S$. This is merely a minor simplification that exploits the formula structure of universal quantifiers. The universal quantifier cannot make assumptions on the inner quantifiers: Either a clause is falsified by some assignment to $X$ or it is not. Refinements are represented as clauses over literals from variables $S$.

The clausal abstraction $\theta_X$ for some quantifier block $\mathcal{Q}X$ is defined as the conjunction over the abstractions of clauses

$$\theta_X \coloneqq \bigwedge_{C_i \in \varphi} clabs_{\mathcal{Q}X}(C_i) \ . \tag{2.7}$$

**Algorithm for Existential Quantifiers.** The algorithm SOLVE$_\exists$ is shown in Algorithm 2.3. It decides whether the QBF $\exists X.\Phi$ is satisfiable under the assumption that the matrix $\varphi$ is restricted according to the assignment $\alpha_S$. The algorithm repeatedly generates candidate assignments by means of the abstraction $\theta_X$ (line 3). If the abstraction returns Unsat, there is no winning assignment for this quantifier, thus, the

**Algorithm 2.3** Algorithm for existentially quantified formulas

```
1:  procedure SOLVE∃(X, Φ, α_S)
2:      loop
3:          match ⟨SAT(θ_X, α_S),  Φ⟩ as        ▷ assume satisfied and falsified clauses
4:              ⟨Sat(α),  ∀Y.Ψ⟩ ⇒                                  ▷ Φ = ∀Y.Ψ
5:                  α'_S ← α_S ⊔ {s_i ↦ T | a_i ∈ α|_A^0}          ▷ update satisfied clauses
6:                  match SOLVE∀(Y, Ψ, α'_S) as                    ▷ recursive verification
7:                      Sat(β_S) ⇒ return Sat(β_S ⊓ α_S^+)
8:                      Unsat(β_S) ⇒ θ_X ← θ_X ∧ ⋁_{s_i∈β_S^0} ā_i   ▷ refine θ_X
9:              ⟨Sat(_),  _⟩ ⇒ return Sat(α_S^+)                   ▷ Φ is propositional
10:             ⟨Unsat(β_S),  _⟩ ⇒ return Unsat(β_S)
11:     end loop
12: end procedure
```

algorithm returns Unsat as well (line 10). Further, the reason for the negative result is given, represented by the assignment $\beta_S$, that indicates which clauses could not be satisfied simultaneously. If the abstraction returns Sat with assignment $\alpha$ we distinguish two cases. The first case is the base case of the recursion, that is, the inner formula is quantifier-free. The algorithm returns Sat together with the partial assignment $\alpha_S^+$ indicating which clauses have to be satisfied by outer quantifier such that the assignment $\alpha_X$ satisfies the matrix $\varphi$.

If the inner subformula is quantified, we split $\alpha$ into two parts $\alpha_A = \alpha|_A$ and $\alpha_X = \alpha|_X$. Then in line 5 we update $\alpha_S$ by marking those clauses as satisfied (set $s_i$ to $\mathbf{T}$) that $\alpha_X$ satisfies and continue with the recursive verification using SOLVE$_\forall$ (line 6) which, again, could either be Sat or Unsat. In the first case, the partial assignment $\beta_S$ (line 7) indicates the clauses that are required to be satisfied. Before returning, we adapt this witness by the operation $\beta_S \sqcap \alpha_S^+$ in line 7 which removes those clauses that are already satisfied by $\alpha_X$, i.e., clauses $C_i$ where $\alpha_S(s_i) = \mathbf{F}$ and $\alpha_A(a_i) = \mathbf{F}$. In the second case where the verification is unsuccessful, the abstraction $\theta_X$ is refined by enforcing that some clause from the previously unsatisfied clauses is satisfied, before continuing with the next iteration.

**Algorithm for Universal Quantifiers.** The algorithm SOLVE$_\forall$, shown in Algorithm 2.4, shares the same underlying concept and structure as SOLVE$_\exists$ and differs only in minor details that we discuss in the following. Due to the different abstractions, the algorithm only assumes already satisfied clauses (by assignments of outer variables), represented by $\alpha_S^+$, when generating the candidate assignment in line 3. This also means that there is no need to update $\alpha_S$ after line 4, as $\alpha|_S$ already represents all satisfied clauses due to the definition of the universal abstraction. Further, the base case is missing as it is guaranteed that every universal quantifier is followed by an existential quantifier (otherwise it can be removed by universal reduction). The refinement in line 7 states that one of the previously satisfied clauses has to be falsified, starting with the next iteration.

**Algorithm 2.4** Algorithm for universally quantified formulas

1: **procedure** $\textsc{solve}_\forall(X, \Phi, \alpha_S)$
2: **loop**
3: **match** $\langle \textsc{sat}(\theta_X, \alpha_S^+),\ \Phi\rangle$ **as** ▷ assume satisfied clauses only
4: $\langle \mathsf{Sat}(\alpha),\ \exists Y.\,\Psi\rangle \Rightarrow$ ▷ $\Phi = \exists Y.\,\Psi$
5: **match** $\textsc{solve}_\exists(Y, \Psi, \alpha|_S)$ **as** ▷ recursive verification
6: $\mathsf{Unsat}(\beta_S) \Rightarrow$ **return** $\mathsf{Unsat}(\beta_S)$
7: $\mathsf{Sat}(\beta_S) \Rightarrow \theta_X \leftarrow \theta_X \wedge \bigvee_{s_i \in \beta_S^1} \overline{s_i}$ ▷ refine $\theta_X$
8: $\langle \mathsf{Unsat}(\beta_S),\ \_\rangle \Rightarrow$ **return** $\mathsf{Sat}(\beta_S)$
9: **end loop**
10: **end procedure**

**Example 2.9.** Consider again the formula given in ➔ Example 2.1: ➔ Page 17

$$\exists v, w.\ \forall x.\ \exists y, z.\ (w \vee x \vee y)(v \vee \overline{w})(x \vee \overline{y})(\overline{v} \vee z)(\overline{z} \vee \overline{x})$$

We build the abstractions

$$\begin{aligned}\theta_{\{v,w\}} &= (s_1 \vee w \vee a_1)(s_2 \vee v \vee \overline{w} \vee a_2)(s_3 \vee a_3)(s_4 \vee \overline{v} \vee a_4)(s_5 \vee a_5),\\ \theta_{\{x\}} &= (s_1 \vee \overline{x})(s_3 \vee \overline{x})(s_5 \vee x), \text{ and}\\ \theta_{\{y,z\}} &= (s_1 \vee y)(s_2)(s_3 \vee \overline{y})(s_4 \vee z)(s_5 \vee \overline{z})\ .\end{aligned}$$

We give a possible execution of algorithm SOLVE. To improve readability, we use the propositional representation for assignments as cubes. Note that clause $C_2$ contains only variables of the outermost quantifier, thus, setting $a_2$ to true is a useless assumption. In Section 2.3.3 we discuss this (and other) improvements for the basic algorithm presented here, for now we just assume that the initial abstraction $\theta_{\{v,w\}}$ is $\theta_{\{v,w\}} \wedge \overline{a_2}$.

- $\textsc{solve}_\exists(\{v, w\}, \forall x.\ \exists y, z.\ \varphi, \overline{s_1}\overline{s_2}\overline{s_3}\overline{s_4}\overline{s_5})$
- $\textsc{sat}(\theta_{\{v,w\}}, \overline{s_1}\overline{s_2}\overline{s_3}\overline{s_4}\overline{s_5}) = \mathsf{Sat}(\overline{v}\,\overline{w}\,a_1\overline{a_2}a_3\overline{a_4}a_5)$
- $\alpha'_S = \overline{s_1}s_2\overline{s_3}s_4\overline{s_5}$
- $\textsc{solve}_\forall(\{x\}, \exists y, z.\ \varphi, \alpha'_S)$
  - $\textsc{sat}(\theta_{\{x\}}, s_2 s_4) = \mathsf{Sat}(x\, s_1 s_2 s_3 s_4 \overline{s_5})$
  - $\textsc{solve}_\exists(\{y, z\}, \varphi, s_1 s_2 s_3 s_4 \overline{s_5})$
    - $\textsc{sat}(\theta_{\{y,z\}}, s_1 s_2 s_3 s_4 \overline{s_5}) = \mathsf{Sat}(\overline{y}\,\overline{z})$
    - **return** $\mathsf{Sat}(s_1 s_2 s_3 s_4)$
  - $\theta'_{\{x\}} = \theta_{\{x\}} \wedge (\overline{s_1} \vee \overline{s_2} \vee \overline{s_3} \vee \overline{s_4})$
  - $\textsc{sat}(\theta_{\{x\}}, s_2 s_4) = \mathsf{Sat}(\overline{x}\,\overline{s_1}s_2\overline{s_3}s_4 s_5)$
  - $\textsc{solve}_\exists(\{y, z\}, \varphi, \overline{s_1}s_2\overline{s_3}s_4 s_5)$
    - $\textsc{sat}(\theta_{\{y,z\}}, \overline{s_1}s_2\overline{s_3}s_4 s_5) = \mathsf{Unsat}(\overline{s_1}\overline{s_3})$

        * **return** Unsat($\overline{s_1}\overline{s_3}$)
    - **return** Unsat($\overline{s_1}\overline{s_3}$)
- $\theta'_{\{v,w\}} = \theta_{v,w} \wedge (\overline{a_1} \vee \overline{a_3})$
- SAT($\theta'_{\{v,w\}}, \overline{s_1}\overline{s_2}\overline{s_3}\overline{s_4}\overline{s_5}$) = Sat($v\, w\, \overline{a_1}\overline{a_2}a_3a_4a_5$)
- $\alpha'_S = s_1s_2\overline{s_3}\overline{s_4}\overline{s_5}$
- SOLVE$_\forall$($\{x\}, \exists y, z.\, \varphi, \alpha'_S$)
    - SAT($\theta_{\{x\}}, s_1s_2$) = Sat($x\, s_1s_2s_3\overline{s_4}\overline{s_5}$)
    - SOLVE$_\exists$($\{y, z\}, \varphi, s_1s_2s_3\overline{s_4}\overline{s_5}$)
        * SAT($\theta_{\{y,z\}}, s_1s_2s_3\overline{s_4}\overline{s_5}$) = Unsat($\overline{s_4}\overline{s_5}$)
        * **return** Unsat($\overline{s_4}\overline{s_5}$)
    - **return** Unsat($\overline{s_4}\overline{s_5}$)
- $\theta''_{\{v,w\}} = \theta'_{v,w} \wedge (\overline{a_4} \vee \overline{a_5})$
- SAT($\theta''_{\{v,w\}}, \overline{s_1}\overline{s_2}\overline{s_3}\overline{s_4}\overline{s_5}$) = Unsat($\overline{s_1}\overline{s_2}\overline{s_3}\overline{s_4}\overline{s_5}$)
- **return** Unsat($\overline{s_1}\overline{s_2}\overline{s_3}\overline{s_4}\overline{s_5}$)

### 2.3.2 Correctness

The proof of correctness generalizes the arguments made in Section 2.2.2 to formulas with arbitrary prefixes. Thus, the correctness argument presented in this section is an inductive argument over the quantifier prefix.

A substantial part of the formal arguments relies on the relation between the abstractions and the quantified Boolean formula that we formalize in the following. Let $\mathcal{Q}X$ and $\alpha_S$ be some quantifier and an assignment of satisfaction variables, respectively. In combination, we can interpret them as a new QBF that starts with the quantifier block $\mathcal{Q}X$, removes all literals that are bound prior to $\mathcal{Q}X$, and has only the clauses that are marked as unsatisfied by $\alpha_S$. To formalize this intuition, we define an operator $\Phi|^{\mathcal{Q}X}_{\alpha_S}$ that restricts the matrix $\varphi$ in a QBF $\Phi$ to those clauses $C_i \in \varphi$ such that $\alpha_S(s_i) = \mathbf{F}$ and removes all leading quantifiers up to $\mathcal{Q}X$. In detail, the resulting QBF has the same quantifier prefix starting with $\mathcal{Q}X$ and the matrix $\{C_i^{\geq} \mid C_i \in \varphi \wedge \alpha_S(s_i) = \mathbf{F}\}$ where $C_i^{\geq}$ refers to quantifier block $\mathcal{Q}X$. Note that variables bound by outer quantifiers are removed from the matrix. As an example, consider the formula $\Phi_{ex} = \exists v, w.\, \forall x.\, \exists y, z.\, (w \vee x \vee y)(v \vee \overline{w})(x \vee \overline{y})(\overline{v} \vee z)(\overline{z} \vee \overline{x})$ from the previous example: The formula $\Phi_{ex}|^{\forall x}_{\overline{s_1}s_2\overline{s_3}s_4\overline{s_5}}$ is equal to $\forall x.\, \exists y, z.\, (x \vee y)(x \vee \overline{y})(\overline{z} \vee \overline{x})$.

We start by stating simple properties about the abstractions after assuming some assignment $\alpha_S$. Those are used in the induction proofs below.

**Lemma 2.10.** *Let $\Phi$ be a QBF with matrix $\varphi$ and let $\alpha_S$ be an assignment over variables $S$.*

1. *Let $\exists X$ be the innermost quantifier block. It holds that $\theta_X[\alpha_S] = \bigwedge_{s_i \in \alpha_S^0} C_i^{=}$ which is equisatisfiable to $\Phi|^{\exists X}_{\alpha_S}$.*

2. *Let $\exists X$ be a (non-innermost) quantifier block of $\Phi$. It holds that $\theta_X[\alpha_S] = \bigwedge_{s_i \in \alpha_S^0} (C_i^{=} \vee a_i)$.*
3. *Let $\forall X$ be a quantifier block of $\Phi$. It holds that $\theta_X[\alpha_S^+] = \bigwedge_{s_i \in \alpha_S^0} (s_i \vee \neg C_i^{=})$.*

*Proof.* Follows immediately from the definition of the abstraction $\theta_X$. □

Let $\mathcal{Q}X.\,\overline{\mathcal{Q}}Y$ be a quantifier alternation of $\Phi$. In the following proofs, we have to transform an assignment $\alpha_S$ of satisfaction variables (w.r.t. $\mathcal{Q}X$) to an assignment of satisfaction variables with respect to $\overline{\mathcal{Q}}Y$ by applying the effect of an assignment $\alpha_X$ to the variables $X$. Often, we will argue over the "optimal" assignment $\alpha_S^*$ of the satisfaction variables $S$ in the abstraction $\theta_X$ to relate $\Phi|_{\alpha_S^*}^{\overline{\mathcal{Q}}Y}$ with $(\Phi|_{\alpha_S}^{\mathcal{Q}X})[\alpha_X]$. The following lemma states this connection formally.

**Lemma 2.11.** *Let $\mathcal{Q}X.\,\overline{\mathcal{Q}}Y$ be a quantifier alternation of $\Phi$ and let $\alpha_X$ and $\alpha_S$ be assignments as defined before. Further, let $\alpha_S^*$ be defined such that $\alpha_S^*(s_i) = \mathbf{T}$ if, and only if, $\alpha_S(s_i) = \mathbf{T}$ or $\alpha_X \vDash C_i|_X$. It holds that $(\Phi|_{\alpha_S}^{\mathcal{Q}X})[\alpha_X] = \Phi|_{\alpha_S^*}^{\overline{\mathcal{Q}}Y}$.*

*Proof.* The quantified formulas $(\Phi|_{\alpha_S}^{\mathcal{Q}X})[\alpha_X]$ and $\Phi|_{\alpha_S^*}^{\overline{\mathcal{Q}}Y}$ have the same prefix (both starting with $\overline{\mathcal{Q}}Y$) and the same matrix

$$\underbrace{\bigwedge_{\substack{C_i \in \varphi \\ \alpha_S(s_i)=\mathbf{F} \wedge \alpha_X \not\vDash C_i|_X}} C_i^{>}}_{>\text{ w.r.t. } \mathcal{Q}X} = \underbrace{\bigwedge_{\substack{C_i \in \varphi \\ \alpha_S^*(s_i)=\mathbf{F}}} C_i^{\geq}}_{\geq\text{ w.r.t. } \overline{\mathcal{Q}}Y}\,. \qquad \square$$

We now have the necessary preconditions to state the inductive arguments formally. The following lemma states that SOLVE$_{\mathcal{Q}}$ returns Sat if the given QBF is satisfiable. Further, the returned witness represents the necessary condition for satisfiability in form of a partial assignment $\beta_S$. Recall that for some partial assignment $\beta$, the notation $\beta[\bot \mapsto b]$ describes the complete assignment where undefined values are set to $b \in \mathbb{B}$.

**Lemma 2.12.** *Let $\mathcal{Q}X.\,\Psi$ be a quantified subformula of a QBF $\Phi$ with matrix $\varphi$ and let $\alpha_S$ be an assignment of variables $S$. If $\Phi|_{\alpha_S}^{\mathcal{Q}X}$ is true SOLVE$_{\mathcal{Q}}(X, \Psi, \alpha_S)$ returns $\mathsf{Sat}(\beta_S)$ where $\beta_S \sqsubseteq \alpha_S^+$ and $\Phi|_{\beta_S[\bot \mapsto \mathbf{F}]}^{\mathcal{Q}X}$ is true.*

*Proof.* We prove the statement by structural induction over the quantifier prefix. The base case follows immediately by Lemma 2.10.1. For the induction step, we consider existential and universal quantification separately. For existential quantifier $\exists X$, there has to be a satisfying assignment $\alpha_X$ by the QBF semantics and we show that this assignment is a satisfying assignment for the abstraction $\theta_X$. Together with the optimal set of assumptions, we can use the induction hypothesis to build a witnessing partial assignment. Completeness follows from the fact that there are only finitely many different refinement clauses and the property that assignment $\alpha_X$ cannot be excluded by some refinement. For universal quantifier $\forall X$, every assignment $\alpha_X$ is satisfying, thus, we show that every satisfying assignment of the abstraction leads to a subsequent refinement. Thus, the

abstraction becomes unsatisfiable (under the given assumption $\alpha_S$) eventually, and the algorithm returns Sat with a witness satisfying the requirement. The detailed proof follows.

*Induction Base.* Let $\exists X.\, \varphi$ be the innermost quantifier of $\Phi$ and let $\alpha_S$ be such that $\Phi|_{\alpha_S}^{\exists X}$ is true. By Lemma 2.10.1, the truth of $\Phi|_{\alpha_S}^{\exists X}$ witnesses the satisfiability of $\theta_X[\alpha_S]$. Further, the algorithm SOLVE$_\exists$ returns $\mathrm{Sat}(\alpha_S^+)$ (line 9) and $\alpha_S^+[\bot \mapsto \mathbf{F}]$ is equivalent to $\alpha_S$.

*Induction Step ($\mathcal{Q} = \exists$).* Let $\exists X.\, \forall Y$ be an arbitrary quantifier alternation of $\Phi$ and let $\alpha_S$ be such that $\Phi|_{\alpha_S}^{\exists X}$ is true. By Lemma 2.10.2 it holds that

$$\theta_X[\alpha_S] = \bigwedge_{s_i \in \alpha_S^0} (C_i^= \vee a_i) \quad .$$

Since $\Phi|_{\alpha_S}^{\exists X}$ is true, there is a satisfying assignment $\alpha_X$ for the variables $X$ such that $(\Phi|_{\alpha_S}^{\exists X})[\alpha_X]$ (a QBF starting with quantifier $\forall Y$) is true. Define $\alpha_A^*$ as $\alpha_A^*(a_i) = \mathbf{F}$ if, and only if, $\alpha_X \vDash C_i^=$. Thus, $\alpha_A^*$ is the assignment with the smallest number of assumptions ($\alpha_A^*(a_i) = \mathbf{T}$) for the given assignment $\alpha_X$. The combined assignment $\alpha_X \dot\sqcup \alpha_A^*$ is a satisfying assignment of the initial abstraction $\theta_X[\alpha_S]$ by construction. We perform a case distinction on the returned assignment of the SAT solver in line 3.

- We assume that the SAT call in line 3 returns $\alpha_X \dot\sqcup \alpha_A^*$. Let $\alpha_S^*$ be the assignment constructed from $\alpha_S$ and $\alpha_A^*$ in line 5. By Lemma 2.11, it holds that $(\Phi|_{\alpha_S}^{\exists X})[\alpha_X] = \Phi|_{\alpha_S^*}^{\forall Y}$ is true. By induction hypothesis we deduce that SOLVE$_\forall$ returns $\mathrm{Sat}(\beta_S)$ where $\Phi|_{\beta_S[\bot \mapsto 0]}^{\forall Y}$ is true. Subsequently, SOLVE$_\exists$ returns $\mathrm{Sat}(\beta_S')$ (line 7), where $\beta_S' = \beta_S \sqcap \alpha_S^+$.

  As the algorithm returns $\mathrm{Sat}(\beta_S')$, it remains to show that $\Phi|_{\beta_S'[\bot \mapsto \mathbf{F}]}^{\exists X}$ is true. For every clause that is removed from $\beta_S$ by the intersection with $\alpha_S^+$, it holds that this clause is satisfied by the assignment $\alpha_X$: Assume $s_i \in S$ is removed by the intersection, that is, $\beta_S(s_i) = \mathbf{T}$ and $\alpha_S(s_i) = \mathbf{F}$. We know that $\beta_S \sqsubseteq \alpha_S^{*+} = (\alpha_S \sqcup \{s_i \mapsto 1 \mid a_i \in \alpha_A^{*0}\})^+$ by induction hypothesis and the construction of $\alpha_S^*$ in line 5. Hence, $\alpha_A^*(a_i) = \mathbf{F}$ and together with $\alpha_S(s_i) = \mathbf{F}$ we conclude that $\alpha_X \vDash C_i^=$ due to the definition of $clabs_{\exists X}$ in ➔ Equation 2.5.

➔ Page 27

- Assume that the SAT call in line 3 returns an assumptio $\alpha_A'$ different to $\alpha_A^*$. Either $\alpha_A'$ corresponds to $\alpha_X$ and is non-minimal, i.e., $\alpha_A^{*+} \sqsubseteq \alpha_A'^{+}$, or it corresponds to a different assignment $\alpha_X'$. The call to SOLVE$_\forall$ may either return Sat or a counterexample $\mathrm{Unsat}(\beta_S)$. We consider the latter case as in the former case SOLVE$_\exists$ also returns Sat and the same argumentation as in the previous case applies.

  The subsequent refinement in line 8 requires that one of the unsatisfied clauses $C_i$ with $\beta_S(s_i) = \mathbf{F}$ has to be satisfied in the next iteration and the corresponding refinement clause is $\psi \coloneqq \bigvee_{s_i \in \beta_S^0} \overline{a_i}$. By construction of $\alpha_A^*$ as the optimal assignment corresponding to $\alpha_X$, $\alpha_A^* \not\vDash \psi$ contradicts that $\alpha_X$ is a satisfying assignment of $\Phi|_{\alpha_S}^{\exists X}$. Hence, $\alpha_X \dot\sqcup \alpha_A^*$ is still a satisfying assignment for the refined abstraction $\theta_X'[\alpha_S]$. The refinement also reduces the number of $A$ assignments by at least 1 and, thus, brings us one step closer to termination.

*Induction Step ($\mathcal{Q} = \forall$).* Let $\forall X. \exists Y$ be a quantifier alternation of $\Phi$ and let $\alpha_S$ be such that $\Phi|_{\alpha_S}^{\forall X}$ is true. For every assignment $\alpha_X$, it holds that $(\Phi|_{\alpha_S}^{\forall X})[\alpha_X]$ (a QBF starting with quantifier $\exists Y$) is true. By Lemma 2.10.3 it holds that

$$\theta_X[\alpha_S^+] = \bigwedge_{s_i \in \alpha_S^0} (s_i \vee \neg C_i^=) \quad .$$

Thus, in order to set $s_i$ to false for some $i$, every literal $l \in C_i^=$ has to be assigned negatively. Fix some arbitrary assignment $\alpha_X$. Let $\alpha_S^*$ be the assignment with $\alpha_S^*(s_i) = \mathbf{T}$ if, and only if, $\alpha_S(s_i) = \mathbf{T}$ or $\alpha_X \vDash C_i^=$. Note that $\alpha_S^*$ is minimal with respect to the number of positively assigned $s_i$ corresponding to $\alpha_X$. For every $\alpha_S'$ returned from the SAT solver in line 3 (assuming $\alpha_X$ is fixed) it holds that ${\alpha_S^*}^+ \sqsubseteq {\alpha_S'}^+$ by the minimality of $\alpha_S^*$. By Lemma 2.11, it holds that $(\Phi|_{\alpha_S}^{\forall X})[\alpha_X] = \Phi|_{\alpha_S^*}^{\exists Y}$ is true and thereby $\Phi|_{\alpha_S'}^{\exists Y}$ is true as its matrix contains a subset of the clauses of $\Phi|_{\alpha_S^*}^{\exists Y}$. By induction hypothesis we deduce that SOLVE$_\exists$ returns $\mathrm{Sat}(\beta_S')$ where $\beta_S' \sqsubseteq \alpha_S'$ and $\Phi|_{\beta_S'[\bot \mapsto 0]}^{\exists Y}$ is true. The subsequent refinement in line 7 reduces the number of $S$ assignments, so the abstraction $\theta_X$ becomes unsatisfiable (under the assumption $\alpha_S$) eventually and the loop terminates with $\mathrm{Sat}(\beta_S)$ in line 8. Let $\theta_X'$ be the abstraction after the termination of the loop. $\beta_S \sqsubseteq \alpha_S^+$ holds as $\beta_S$ are the failed assumptions of the SAT call SAT$(\theta_X', \alpha_S^+)$.

It remains to show that $\Phi|_{\beta_S[\bot \mapsto \mathbf{F}]}^{\forall X}$ is true. Assume for contradiction that there is some $\alpha_X$ such that $(\Phi|_{\beta_S[\bot \mapsto \mathbf{F}]}^{\forall X})[\alpha_X]$ is false. We know that $\theta_X'[\alpha_X \dot\cup \beta_S]$ is unsatisfiable. Either the initial abstraction $\theta_X[\alpha_X \dot\cup \beta_S]$ was unsatisfiable, which leads to a contradiction due to Lemma 2.11, or the assignment $\alpha_X$ was excluded due to refinements. As the refinement only excludes $S$ assignments $\beta_S''$ such that $\Phi|_{\beta_S''[\bot \mapsto \mathbf{F}]}^{\exists Y}$ is true, this leads to a contradiction as well. □

The following lemma states the reverse direction, that the algorithm terminates with the correct result on false formulas. The arguments used in the proof are very similar to the one for true formulas, but the differences are enough to justify their inclusion.

**Lemma 2.13.** *Let $\mathcal{Q}X. \Psi$ be a quantified subformula of a QBF $\Phi$ with matrix $\varphi$ and let $\alpha_S$ be an assignment of variables $S$. If $\Phi|_{\alpha_S}^{\mathcal{Q}X}$ is false* SOLVE$_\mathcal{Q}(X, \Psi, \alpha_S)$ *returns* $\mathrm{Unsat}(\beta_S)$ *where $\beta_S \sqsubseteq \alpha_S^-$ and $\Phi|_{\beta_S[\bot \mapsto \mathbf{T}]}^{\mathcal{Q}X}$ is false.*

*Proof.* The structure of the proof is similar to the proof of Lemma 2.12, that is, a structural induction over the quantifier prefix. For existential quantifier $\exists X$, every assignment $\alpha_X$ leads to a false QBF. We can use the induction hypothesis for every assignment produced by the abstraction $\theta_X$ as the abstraction computes an under-approximation of the satisfied clauses with respect to $\alpha_X$. We show that the subsequent refinement excludes at least the given assignment, thus, the abstraction becomes unsatisfiable eventually (under the given assumption $\alpha_S$). It remains to show that the returned partial assignment satisfies is a witness for the falsity of the subformula. For universal quantifier $\forall X$, there is some assignment $\alpha_X$ that leads to a false QBF. We show that the algorithm eventually reaches this assignment (or another assignment that leads to unsatisfiability). Applying induction hypothesis leads to a witnessing partial assignment. The detailed proof follows.

*Induction Base.* Let $\exists X.\, \varphi$ be the innermost quantifier of $\Phi$ and let $\alpha_S$ be such that $\Phi|^{\exists X}_{\alpha_S}$ is false. By Lemma 2.10.1, $\theta_X[\alpha_S]$ is unsatisfiable. Let $\beta'_S$ be the failed assumptions from the call to $\textsc{sat}(\theta_X, \alpha_S)$, i.e., $\beta'_S \sqsubseteq \alpha_S^-$ and $\theta_X[\beta'_S]$ is unsatisfiable. Again by Lemma 2.10.1 it holds that $\Phi|^{\exists X}_{\beta'_S[\bot \mapsto \mathbf{T}]}$ is false which concludes the induction base as $\text{Unsat}(\beta'_S)$ is returned from $\textsc{solve}_\exists$.

*Induction Step ($\mathcal{Q} = \exists$).* Let $\exists X.\, \forall Y$ be a quantifier alternation of $\Phi$ and let $\alpha_S$ be such that $\Phi|^{\exists X}_{\alpha_S}$ is false. For every assignment $\alpha_X$, it holds that $(\Phi|^{\exists X}_{\alpha_S})[\alpha_X]$ is false. By Lemma 2.10.2 it holds that

$$\theta_X[\alpha_S] = \bigwedge_{s_i \in \alpha_S^0} (C_i^= \vee a_i) \quad .$$

The abstraction $\theta_X$ is initially satisfiable for every choice of $\alpha_S$ (every $a_i$ can be set to true)[3]. Let $\alpha$ be such a satisfying assignment of $\theta_X[\alpha_S]$. We define $\alpha_X \coloneqq \alpha|_X$ and $\alpha_A \coloneqq \alpha|_A$. By Lemma 2.10.2, $\alpha_X \not\models C_i^=$ implies that $\alpha_A(a_i) = \mathbf{T}$. We define the assignment with optimal assumptions $\alpha_A^*$ as $\alpha_A^*(a_i) = \mathbf{F}$ if, and only if, $\alpha_X \models C_i^=$. Note that $\alpha_X \dot\cup \alpha_A^*$ is a satisfying assignment of $\theta_X[\alpha_S]$. We show that even with optimal assumptions $\alpha_A^*$, the quantified subformula is unsatisfiable and the subsequent refinement step excludes at least assignment $\alpha_S \dot\cup \alpha_A$ from the abstraction $\theta_X$.

Let $\alpha'_S$ and $\alpha^*_S$ be the assignments after line 5 with respect to $\alpha_A$ and $\alpha_A^*$, respectively. From the construction, we know that ${\alpha_A}^- \sqsubseteq {\alpha_A^*}^-$, by the optimality of $\alpha_A^*$, and thereby ${\alpha'_S}^+ \sqsubseteq {\alpha^*_S}^+$. We deduce that $\Phi|^{\forall Y}_{\alpha'_S}$ is false, as the clauses in the matrix $\Phi|^{\forall Y}_{\alpha'_S}$ are a superset of those in the matrix of $\Phi|^{\forall Y}_{\alpha^*_S}$ which is equal to $(\Phi|^{\exists X}_{\alpha_S})[\alpha_X]$ by Lemma 2.11. By induction hypothesis, $\textsc{solve}_\forall$ with assignment $\alpha'_S$ returns $\text{Unsat}(\beta_S)$ such that $\beta_S \sqsubseteq {\alpha'_S}^-$ and $\Phi|^{\forall Y}_{\beta_S[\bot \mapsto \mathbf{T}]}$ is false. As $\beta_S^0 \subseteq {\alpha'_S}^0 = \{s_i \in S \mid \alpha_S(s_i) = \mathbf{F} \wedge \alpha_A(a_i) = \mathbf{T}\}$, the following refinement with clause $\bigvee_{s_i \in \beta_S^0} \overline{a_i}$ excludes assignment $\alpha_S \dot\cup \alpha_A$ from $\theta_X$. As there are only finitely many refinement clauses, the SAT call in line 3 eventually becomes unsatisfiable when assuming $\alpha_S$. Let $\theta'_X$ be the abstraction at this point and let $\beta'_S$ be the failed assumptions, i.e., $\beta'_S \sqsubseteq \alpha_S^-$.

Let $\alpha''_S = \beta'_S[\bot \mapsto \mathbf{T}]$. It remains to show that $\Phi|^{\exists X}_{\alpha''_S}$ is false. Assume for contradiction that there is some $\alpha_X$ such that $(\Phi|^{\exists X}_{\alpha''_S})[\alpha_X]$ is true. It holds that $\theta'_X[\alpha_X \dot\cup \alpha''_S]$ is unsatisfiable, whereas initially, $\theta_X[\alpha_X \dot\cup \alpha''_S]$ is satisfiable. Thus, the assignment $\alpha_X$ was excluded due to refinements. As the refinement only excludes assignments corresponding to some $S$ assignment $\beta''_S$ such that $\Phi|^{\forall Y}_{\beta''_S[\bot \mapsto \mathbf{T}]}$ is false, this contradicts our assumption.

*Induction Step ($\mathcal{Q} = \forall$).* Let $\forall X.\, \exists Y$ be a quantifier alternation of $\Phi$ and let $\alpha_S$ be such that $\Phi|^{\forall X}_{\alpha_S}$ is false, that is, there is an assignment $\alpha_X$ such that $(\Phi|^{\forall X}_{\alpha_S})[\alpha_X]$ is false. By Lemma 2.10.3 it holds that

$$\theta_X[\alpha_S^+] = \bigwedge_{s_i \in \alpha_S^0} (s_i \vee \neg C_i^=) \quad .$$

[3] In Section 2.3.3 we describe improvements of the abstraction.

$\theta_X[\alpha_S^+]$ is initially satisfiable. Let $\alpha$ be a satisfying assignment of $\theta_X[\alpha_S^+]$ and define $\alpha'_X \coloneqq \alpha|_X$ and $\alpha'_S = \alpha|_S$. Given $\alpha_X$ from above, we define the optimal corresponding assignment $\alpha_S^*$ as $\alpha_S^*(s_i) = \mathbf{T}$ if, and only if, $\alpha_S(s_i) = \mathbf{T}$ or $\alpha_X \vDash C_i^=$. Note that $\alpha_S$ and $\alpha_S^*$ correspond to quantifier $\forall X$ and $\exists Y$, respectively. If $\alpha'_S = \alpha_S^*$, the call to SOLVE$_\exists$ returns Unsat$(\beta_S)$ where $\beta_S \sqsubseteq \alpha_S^{*-}$ and $\Phi|^{\exists Y}_{\beta_S[\bot\mapsto\mathbf{T}]}$ is false by induction hypothesis as $(\Phi|^{\forall X}_{\alpha_S})[\alpha_X] = \Phi|^{\exists Y}_{\alpha_S^*}$ (Lemma 2.11) is false. Subsequently, SOLVE$_\forall$ returns Unsat$(\beta_S)$ (line 6). $\beta_S \sqsubseteq \alpha_S^-$ follows from $\alpha_S^{*-} \sqsubseteq \alpha_S^-$ due to the monotonicity of the abstraction: If $\alpha_S^*(s_i) = \mathbf{F}$, then $\alpha_S(s_i) = \mathbf{F}$.

Let $\alpha'_S \neq \alpha_S^*$ and assume that SOLVE$_\exists$ returns Sat$(\beta_S)$. Subsequently, $\theta_X$ is refined by adding the the clause $\psi \coloneqq \bigvee_{s\in\beta_S^1} \overline{s_i}$. Assume for contradiction that $\alpha_S^* \nvDash \psi$, i.e., that $\alpha_S^*$ is excluded by the refinement. Remember that $\alpha_S^*$ was constructed as the optimal assignment corresponding to $\alpha_X$. Hence, the exclusion contradicts that $\alpha_X$ is a witness that $\Phi|^{\forall X}_{\alpha_S}$ is false. Thus, $\alpha_X \dot\cup\, \alpha_S^*$ remains a satisfying assignment of the refined abstraction. The refinement reduced the number of $S$ assignments and, thus, some falsifying assignment $\alpha_X$ is reached eventually. □

Since the main algorithm SOLVE directly calls into SOLVE$_Q$, the following theorem follows immediately from Lemma 2.12 and 2.13.

**Theorem 2.14.** *SOLVE returns* Sat *if, and only if,* $\Phi$ *is true.*

### 2.3.3 Optimizations

In this section, we introduce optimizations for the basic algorithm presented in Section 2.3.1. We start with two optimizations already described in the initial paper describing clausal abstraction [RT15]. We then proceed to improvements of the abstraction followed by algorithmic improvements. Some of these optimizations are generalized from the 2QBF fragment in Section 2.2.3.

**Stronger Refinements.** An existential conflict for quantifier alternation $\exists X.\, \forall Y$ of QBF $\Phi$ is a partial assignment $\beta_S$ such that $\Phi|^{\forall Y}_{\beta_S[\bot\mapsto\mathbf{T}]}$ is false. Intuitively, $\beta_S$ represents a set of clauses $\mathcal{C} = \{C_i \mid s_i \in \beta_S^0\}$ that could not be satisfied by the inner quantifier, i.e., replacing the matrix of $\Phi$ by $\mathcal{C}^> = \{C_i^> \mid s_i \in \beta_S^0\}$ results in a false QBF (Lemma 2.13). Refinements for such a partial assignment (line 8 of Algorithm 2.3), thus, assert that one of these clauses has to be satisfied at quantifier $\exists X$ to prevent this situation.

In certain cases, we can strengthen the refinement by excluding a conjunction of "equivalent" clauses, that are clauses that can replace the original clause and would let to the same result. Let $\mathcal{C}$ be the representation of some existential conflict, let $C_i \in \mathcal{C}$ and let $\mathcal{C}'$ be $\mathcal{C} \setminus C_i$. If there is some $C_j \in \varphi$, such that $C_j^> \subseteq C_i^>$, then $\mathcal{C}' \cup C_j$ is an existential conflict as well. Thus, we change the refinement to exclude all equivalent existential conflicts by modifying it to

$$\bigvee_{s_i\in\beta_S^0} \bigwedge_{\substack{C_j\in\varphi \\ C_j^>\subseteq C_i^>}} \overline{a_j} \ . \tag{2.8}$$

→ Page 54

In → Section 3.2.1 we show that this improved refinement makes the underlying proof system exponentially more succinct.

The property described above is in some sense *static*, i.e., the sets of clauses such that $C_j^{>} \subseteq C_i^{>}$ can be computed once and can then be used throughout solving. However, it also ignores the difference between existential and universal variables. In the following, we develop an improved refinement that takes the quantification type into account. We show that only considering existential variables is unsound and derive a characterization how universal variables have to be taken into consideration. Given a clause $C$, we denote by $C|_\exists$ and $C|_\forall$ the projection of $C$ to existential and universal variables, respectively. Restricting the subset inclusion to existential variables only, i.e., $C_j^{>}|_\exists \subseteq C_i^{>}|_\exists$, lets us derive an unsound refinement clause in the following true QBF:

$$\exists x \forall y \exists z.\, (x \vee y \vee z)(y \vee \overline{z})(\overline{x} \vee \overline{y} \vee z) \quad .$$

Assume the partial assignment $\{s_1 \mapsto \mathbf{F}, s_2 \mapsto \bot, s_3 \mapsto \bot\}$ is returned at line 8 of Algorithm 2.3. Using $C_j^{>}|_\exists \subseteq C_i^{>}|_\exists$ in Equation 2.8, we get the refinement $\overline{a_1} \wedge \overline{a_3}$ as $(y \vee z)|_\exists \subseteq (\overline{y} \vee z)|_\exists$. This, however, makes the abstraction $\theta_{\{x\}}$ unsatisfiable (as it asserts that $x \wedge \overline{x}$) and, thus, results in the algorithm returning unsatisfiable as well. The problem is, that $C_1$ and $C_3$ contain the same universal variable $y$ in opposite polarities. Replacing one clause with the other in the underlying $Q$-resolution proof leads to universal tautology clauses and, thus, unsoundness. We refer the reader to Section 3.2.1 for more details.

Let $U = \bigcup_{s_i \in \beta_S^0} C_i^{>}|_\forall$ be the set of universal literals corresponding to $\beta_S$. The refinement

$$\bigvee_{s_i \in \beta_S^0} \quad \bigwedge_{\substack{C_j \in \varphi \\ C_j^{>}|_\exists \subseteq C_i^{>}|_\exists \wedge (C_j^{>}|_\forall \cup U) \text{ is not tautological}}} \overline{a_j} \tag{2.9}$$

is a sound generalization of Equation 2.8. However, it is not as efficient to implement as the property is now dependent on the conflict $\beta$ itself. We show soundness in Section 3.2.1, where we also explore the proof-theoretic expressibility.

**Tree-shaped Quantifier Prefix.** As a preprocessing, we apply the well known miniscoping rule

$$\forall X.\, \exists Y \exists Z.\, \varphi(X, Y) \wedge \psi(X, Z) \equiv (\forall X.\, \exists Y.\, \varphi(X, Y)) \wedge (\forall X.\, \exists Z.\, \psi(X, Z)) \quad ,$$

that is, at every existential quantifier block we search for a partitioning of the matrix into independent formulas. By applying this rule bottom-up, we get a tree-shaped quantifier prefix. Note, that this tree only branches after an existential quantifier, hence, we modify the algorithm to split the current entry according to the partitioning and solve every child individually. In → Section 4.3 we discuss this parallelization in more detail in the context of solving non-prenex formulas in negation normal form.

→ Page 86

**Abstraction Improvements.** We describe improvements to the way the abstractions are built, that is, reducing the number of satisfaction and assumption variables. These optimizations are similar to the ones described in Section 2.2.3. Fix some QBF $\Phi$. Let

$\exists X.\, \Psi$ be a quantified subformula of $\Phi$ and let $C_i$ some clause. If $C_i^<$ is empty, i.e., the clause contains no variable bound at some outer quantifier, then the assumption variable $s_i$ at this quantifier can be always assumed to be false. Further, if $C_i^>$ is empty, then $a_i$ can be assumed to be false and, thus, be removed. This requires a change to ➔ Algorithm 2.3, ➔ Page 28 though: In the return $\mathrm{Sat}(\beta_S \sqcap \alpha_S^+)$ in line 7 we have to add those clauses without assumption variable that are not satisfied by the current assignment, i.e., it has to change to $\mathrm{Sat}((\beta_S \sqcup \{s_i \mapsto \mathbf{T} \mid C_i^> = \varnothing \wedge \alpha_X \not\models C_i^=\}) \sqcap \alpha_S^+)$. Independent of the quantifier type, it is possible to omit building the abstraction for clauses with $C_i^= = \varnothing$ where the given quantifier has no influence on the satisfaction of the clause. Especially, we do not need to add the satisfaction and assumption variables initially. This is possible, since the updates to the satisfaction assignment $\alpha_S$ are monotone: If a clause is satisfied at some outer quantifier, it is guaranteed to be satisfied by every inner quantifier (see line 5 of Algorithm 2.3 and lines 3–4 of ➔ Algorithm 2.4). However, we may need to add them during solving in ➔ Page 29 case there is some refinement involving those variables.

We generalize the simplifications for the universal abstraction introduced for 2QBF in Section 2.2.3:

- If some clause $C_i \in \varphi$ is a universal unit clause, i.e., $C|_X = \{l\}$ for some literal $l$ with $var(l) \in X$, and there are no outer variables ($C_i^< = \varnothing$) then the shared variable $s_i$ can be replaced by the negation $\overline{l}$ of the literal.
- If there is a pair of clauses $C_i, C_j \in \varphi$ with $i \neq j$ such that those clauses are equal with respect to the variables bound at this quantifier, i.e., $C_i^\leq = C_j^\leq$, then the same shared variable $s_i$ can be used for both clauses.

The abstraction allows the existential quantifier to make assumptions on the satisfaction of clauses by inner quantifiers by means of the assignment $\alpha_A$. Lonsing et al. [LES16] proposed to check during solving whether assuming the current assignment makes the matrix unsatisfiable when treating all variables existentially. We can implement this check in the abstraction for some existential quantifier $\exists X.$ by adding the clause $(a_i \to C_i^>)$ for every clause $C_i$. Hence, assumptions made by the existential quantifier cannot lead to a unsatisfiable matrix, which, in theory should improve the solving by reducing the number of recursive calls.

**Algorithmic Improvements.** We recap generalizations of the algorithmic improvements described for the 2QBF algorithm in Section 2.2.3. Given some assignment $\alpha_X$ from the abstraction, we construct the corresponding "optimal" assignment of $\alpha_A$ (Algorithm 2.3) and $\alpha_S$ (Algorithm 2.4) as described by ➔ Lemma 2.11, respectively. For ➔ Page 31 the propositional case of existential quantifier $\exists X$, the same optimizations as discussed in Section 2.2.3 can be applied: We set $\alpha_S(s_i) = \mathbf{F}$ before line 9 if $\alpha_S(s_i) = \mathbf{T}$ and $\alpha_X \models C_i^=$. Further, we may change the assignment $\alpha_X$ if such a change satisfies strictly more clauses.

We also generalize the optimization of refinement clauses due to subsumed literals described in Section 2.2.3. Given a partial assignment $\beta_S$ representing a conflict in line 8 of SOLVE$_\exists$ (Algorithm 2.3). If there are two clauses $C_i$ and $C_j$ with $C_i^\leq \subseteq C_j^\leq$ and $\beta_S(s_i) = \beta_S(s_j) = \mathbf{F}$, then we can set $\beta_S(s_i) = \bot$, which removes $\overline{a_i}$ from the refinement clause.

Given a partial assignment $\beta_S$ representing a conflict in line 7 of SOLVE$_\forall$ (Algorithm 2.4). If there are two clauses $C_i$ and $C_j$ with $C_i^{\leq} \subseteq C_j^{\leq}$ and $\beta_S(s_i) = \beta_S(s_j) = \mathbf{T}$, then we can set $\beta_S(s_j) = \bot$, which removes $\overline{s_j}$ from the refinement clause.

The presented algorithms refine conflicts at the earliest point possible, e.g., if a universal quantifier returns Unsat$(\beta_S)$ (line 8 of Algorithm 2.3), the abstraction at the existential quantifier is refined immediately. In some cases, this refinement is not needed as the existential quantifier does not control any of the refined clauses, that is, for all $C_i \in \varphi$ with $s_i \in \beta_S^0$ it holds that $C_i^{=} = \varnothing$. The following SAT call in line 3 is unsatisfiable and $\beta_S$ is a possible failed assumption. Thus, the conflict is just propagated. As an example, consider the prefix $\exists x \forall v \exists y \forall w \exists z$ and a clauses $(x \vee \overline{v} \vee w \vee \overline{z})(x \vee \overline{v} \vee w \vee z)$. Given the assignment $\overline{x} v \overline{w}$, the quantifier $\exists z$ cannot satisfy both clauses simultaneously. The refinement at quantifier $\exists y$ produces the same conflict again as $y$ has no impact. We add a check to Algorithm 2.3 and Algorithm 2.4 whether a conflict $\beta_S$ can be propagated, thus, saving the cost of the refinement and the subsequent SAT call. This optimization was first described as part of the clause selection algorithm [JM15b].

## 2.4 Function Extraction

For quantified Boolean formulas, the solving result goes beyond the binary decision problem discussed in the previous sections. Especially when using QBF as a target for applications, the witnessing Boolean functions are of great importance. Using Skolem functions, one can directly construct realizing implementations for synthesis problems encoded to QBF [Fay+17; FFT17; BKS14; Blo+14]. And even in the negative case, the Herbrand functions may give valuable information about the underlying reason [HT18]. Another benefit of function extraction is the *certification* of the solving result, i.e., having a verifiable witness for the solving result. In this section, we present the function extraction approach for the clausal abstraction algorithm. In Chapter 3, we discuss another certification approach that is based on producing polynomially verifiable proofs in a calculus that models the execution of the clausal abstraction algorithm.

The function extraction is based on the correctness proof given in Section 2.3.2. Given a QBF $\Phi$, some quantifier block $\mathcal{Q}X$ of $\Phi$, and some assignment of satisfaction variables $\alpha_S$. Lemma 2.12 shows that there is an assignment to $\alpha_X$ such that the subformula $(\Phi|_{\alpha_S}^{\exists X})[\alpha_X]$ is true if $\Phi|_{\alpha_S}^{\exists X}$ is true. Dually, Lemma 2.13 states that an assignment to $\alpha_X$ exists such that the subformula $(\Phi|_{\alpha_S}^{\forall X})[\alpha_X]$ is false if $\Phi|_{\alpha_S}^{\forall X}$ is false. Thus, the function extraction amounts to logging the relevant results during the execution of the algorithm, that is after the successful verification of the candidate assignment. In the following, we determine the relevant information that is needed for the extraction, the data structure in which the information is stored, and an extraction algorithm that returns the Skolem and Herbrand functions, respectively.

**Recursion Tree.** The execution of the clausal abstraction algorithm can be represented as a tree, where the nodes represent quantifiers $\mathcal{Q}X$ and the edges determines the truth value and witnessing assignments $\alpha_X$. Formally, a node in the recursion tree is a pair

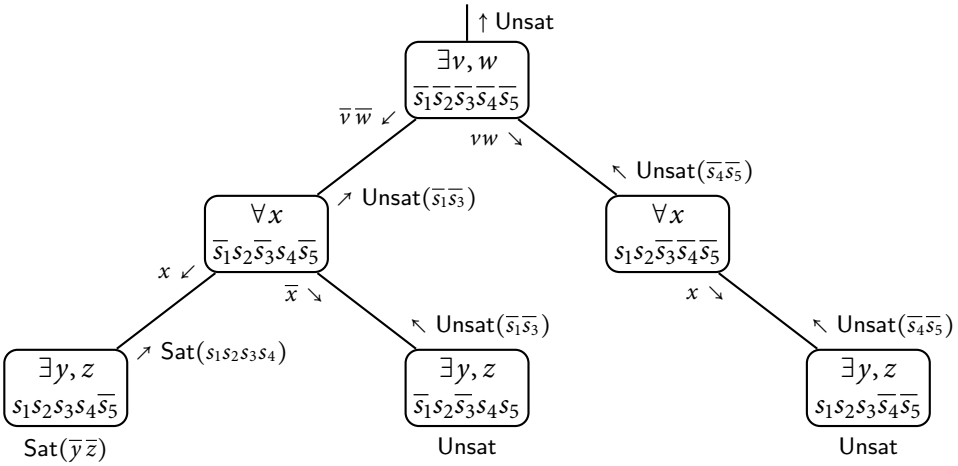



Figure 2.1: Recursion tree corresponding to the execution of SOLVE$_\mathcal{Q}$ on the formula $\exists v, w.\, \forall x.\, \exists y, z.\, (w \vee x \vee y)(v \vee \overline{w})(x \vee \overline{y})(\overline{v} \vee z)(\overline{z} \vee \overline{x})$ as shown in Example 2.9.

$\langle \mathcal{Q}X, \alpha_S \rangle$ and there is an edge from $\langle \mathcal{Q}X, \alpha_S \rangle$ to $\langle \overline{\mathcal{Q}}Y, \alpha'_S \rangle$ labeled with the candidate assignment $\alpha_X$ and the result $res(\beta_S)$ returned from SOLVE$_\mathcal{Q}$ if, and only if, (1) $\overline{\mathcal{Q}}Y$ is the quantifier block following $\mathcal{Q}X$, (2) $(\Phi|^{\mathcal{Q}X}_{\alpha_S})[\alpha_X] =^4 \Phi|^{\overline{\mathcal{Q}}Y}_{\alpha'_S}$, and (3) $res$ is the result of $\Phi|^{\overline{\mathcal{Q}}Y}_{\alpha'_S}$ where $\Phi|^{\overline{\mathcal{Q}}Y}_{\beta_S[\bot \mapsto \mathbf{F}]}$ is true if $res$ = Sat and $\Phi|^{\overline{\mathcal{Q}}Y}_{\beta_S[\bot \mapsto \mathbf{T}]}$ is false otherwise. The leaf nodes $\langle \exists X, \alpha_S \rangle$ are labeled with the result of the propositional formula $\Phi|^{\exists X}_{\alpha_S}$, that is, either Unsat or Sat($\alpha_X$). The root node for some formula $\Phi = \mathcal{Q}X.\, \Psi$ is the designated node $\langle \mathcal{Q}X, \{s_i \mapsto \mathbf{F} \mid C_i \in \varphi\} \rangle$. We depict such a recursion tree in Figure 2.1.

After the algorithm terminates, we use the recursion tree to extract the relevant information to build Skolem and Herbrand functions, respectively. Note that for true QBFs and existential nodes as well as false QBFs and universal nodes, the respective nodes have exactly one outgoing edge where the candidate assignment was verified recursively. Due to the correctness lemmata ➔ Lemma 2.12 ➔ Page 31 and ➔ Lemma 2.13 ➔ Page 33, only the labeling of the edges, i.e., the assignment $\alpha_X$ and the returned partial assignment $\beta_S$ are relevant. Thus, we store a list of these *verified candidates* as a sequence of pairs $\langle \beta_S, \alpha_X \rangle \in (\mathcal{A}_\bot(S) \times \mathcal{A}(X))$ for every quantifier block $\mathcal{Q}X$.

**Function Extraction.** We define a function $inv_{\mathcal{Q}X}\colon \mathcal{A}_\bot(S) \to \mathcal{B}(V)$ which, for a given quantifier block $\mathcal{Q}X$, maps an assignment $\beta_S$ to a propositional formula over variables $V$ bound by outer quantifiers (with respect to $\mathcal{Q}X$). Intuitively, $inv_{\mathcal{Q}X}(\beta_S)$ describes those assignments that lead to $\beta_S$ in the abstraction of quantifier block $\mathcal{Q}X$. We define $inv_{\mathcal{Q}X}$ as

$$inv_{\mathcal{Q}X}(\beta_S) \coloneqq \begin{cases} \bigwedge_{s_i \in \beta^1_S} C^<_i & \text{if } \mathcal{Q} = \exists \\ \bigwedge_{s_i \in \beta^0_S} \neg C^<_i & \text{otherwise} \end{cases}. \tag{2.10}$$

[4] The equality holds if we assume optimal assumptions w.r.t. $\alpha_X$ as discussed in Section 2.3.3 about algorithmic improvements.

Let $\langle \beta_S^1, \alpha_X^1 \rangle \dots \langle \beta_S^n, \alpha_X^n \rangle$ be the pairs of verified candidates corresponding to quantifier block $\mathcal{Q}X$ and let $x \in X$ be some variable, the function $f_x \colon \mathcal{A}(V) \to \mathbb{B}$ is defined as

$$f_x \coloneqq \bigvee_{i=1}^{n} \left( (\alpha_X^i(x) = 1) \wedge inv_{\mathcal{Q}X}(\beta_S^i) \wedge \bigwedge_{j<i} \neg inv_{\mathcal{Q}X}(\beta_S^j) \right) \quad . \tag{2.11}$$

The definition of $inv_{\mathcal{Q}X}$ allows that $f_x$ may depend on all variables bound at outer quantifiers, even those that are of the same quantifier type. By replacing those variables with their extracted functions, one can make sure that $f_x$ depends only on its dependencies $dep(x)$. The size of $f_x$, measured in terms of distinct subformulas, is linear in the number of pairs. The function $f_X \colon \mathcal{A}(V) \to \mathcal{A}(X)$ is defined as the union over all $f_x$ for $x \in X$, formally $f_X(\alpha_V) \coloneqq \bigsqcup_{x \in X} \{x \mapsto f_x(\alpha_V)\}$. The Skolem and Herband function are then defined as the union over the functions $f_X$ for every $\mathcal{Q}X$ where $\mathcal{Q} = \exists$ for Skolem functions and $\mathcal{Q} = \forall$ for Herbrand functions.

**Example 2.15.** We show the function extraction for our running example $\exists v, w. \forall x. \exists y, z. (w \vee x \vee y)(v \vee \overline{w})(x \vee \overline{y})(\overline{v} \vee z)(\overline{z} \vee \overline{x})$. From the recursion tree in Figure 2.1, we extract the sequence $\langle \overline{s_1} \overline{s_3}, \overline{x} \rangle \langle \overline{s_4} \overline{s_5}, x \rangle$ as described above. Applying the definition of $inv_{\forall x}$, we get

$$\begin{aligned} inv_{\forall x}(\overline{s_1} \overline{s_3}) &= \neg C_1^{<} \wedge \neg C_3^{<} = \neg w \quad \text{and} \\ inv_{\forall x}(\overline{s_4} \overline{s_5}) &= \neg C_4^{<} \wedge \neg C_5^{<} = v \quad . \end{aligned}$$

Thus, the Herbrand function $f_x$ is defined as

$$f_x(v, w) = inv_{\forall x}(\overline{s_4} \overline{s_5}) \wedge \neg inv_{\forall x}(\overline{s_1} \overline{s_3}) = v \wedge w \quad .$$

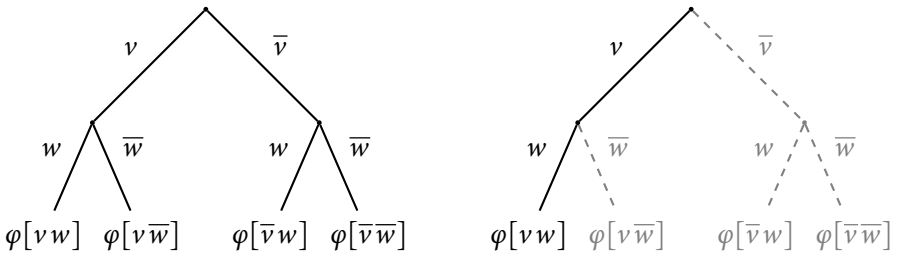
$f_x$ depends solely on its dependencies and is functionally correct as $\varphi[f_x]$ is equal to

$$\begin{aligned} & (w \vee (v \wedge w) \vee y)(v \vee \overline{w})((v \wedge w) \vee \overline{y})(\overline{v} \vee z)(\overline{z} \vee \overline{v} \vee \overline{w}) \\ = \; & (v)(w)(v \vee \overline{w})(\overline{v} \vee z)(\overline{z} \vee \overline{v} \vee \overline{w}) \\ = \; & (v)(w)(z)(\overline{z} \vee \overline{v} \vee \overline{w}) = \mathbf{F} \quad . \end{aligned}$$

**Theorem 2.16.** *Skolem and Herbrand functions generated by the clausal abstraction algorithm are correct.*

*Proof.* Let $\Phi$ be a true QBF over existential and universals variables $V_\exists$ and $V_\forall$, respectively, and let $f$ be the Skolem function as described above. It holds that $f = \bigsqcup_{v \in V_\exists} f_v$ is well-formed by construction. Assume that $f$ is not functionally correct. Thus, there is an assignment $\alpha_\forall$ of the universal variables $V_\forall$ such that $\alpha_\forall \vDash \neg\varphi[f]$. We show that $f$ and $\alpha_\forall$ together lead to a root-to-leaf path in recursion tree such that all clauses in the matrix are satisfied. In detail, we build this path by a traversal of the recursion tree where at every node we take the leftmost choice such that

- at an existential node $\langle \exists X, \alpha_S \rangle$, we take the unique edge labeled with Sat and




(a) A full expansion tree

(b) A partial expansion tree

Figure 2.2: A representation of full and partial expansion trees for formula $\forall v, w.\, \exists x, y.\, \varphi$, where $\varphi = (v \to x) \wedge (w \to y) \wedge (\overline{x} \vee \overline{y})$. The root-to-leaf paths represent a universal assignment $\alpha_{\{v,w\}}$ and the corresponding leaf node contains the propositional formula $\varphi[\alpha_{\{v,w\}}]$ expanded with $\alpha_{\{v,w\}}$. Both trees witness the unsatisfiability of $\forall v, w.\, \exists x, y.\, \varphi$.

- at an universal node $\langle \forall X, \alpha_S \rangle$, we take the leftmost edge labeled with $\mathrm{Sat}(\beta_S)$ such that the set of clauses in $\Phi|^{\forall X}_{\beta_S[\bot \mapsto \mathbf{F}]}$ is a superset of the clauses in $(\Phi|^{\forall X}_{\alpha_S})[\alpha_\forall|_X]$. Intuitively, the assignment $\alpha_\forall|_X$ satisfies more clauses than needed to show that the remaining subformula is true. This partial assignment $\beta_S$ would have excluded $\alpha_\forall|_X$ under the assumption $\alpha_S$ in the refinement step of SOLVE$_\forall$(.) Note, that such an edge has to exist and all outgoing edges are labeled with Sat as otherwise, the universal node would not return Sat itself.

By construction, such a path exists and it is consistent with the Skolem function $f$ due to Equation 2.11. Thus, $f$ produces an assignment corresponding to $\alpha_\forall$ that satisfies the matrix, contradicting $\alpha_\forall \vDash \neg\varphi[f]$. Analogously for false QBFs. □

## 2.5 Integrating Partial Expansion

In this section, we continue our quest started in Section 2.3.3 for improved refinements for existential quantifiers. Expansion-based solving methods are based on the idea that a universal quantifier $\forall x.\, \varphi$ can be rewritten as the conjunction $\varphi[x \mapsto \mathbf{F}] \wedge \varphi'[x \mapsto \mathbf{T}]$ where $x$ is eliminated by replacing it with $\mathbf{F}$ and $\mathbf{T}$ in the left and right conjunct, respectively, and by creating a copy of every variable in the right conjunct. By repeated application, a QBF can be transformed to a propositional formula. This type of *complete* expansion is for example implemented by the solvers QUBOS [AB02], QUANTOR [Bie04], and AIGSOLVE [SP16]. Consider, for example, the false QBF $\forall v, w.\, \exists x, y.\, (v \to x) \wedge (w \to y) \wedge (\overline{x} \vee \overline{y})$. Expanding $v$ and $w$ results into the unsatisfiable propositional formula $(x^{vw})(x^{v\overline{w}})(y^{vw})(y^{\overline{v}w})(\overline{x}^{vw} \vee \overline{y}^{vw})(\overline{x}^{\overline{v}w} \vee \overline{y}^{\overline{v}w})(\overline{x}^{v\overline{w}} \vee \overline{y}^{v\overline{w}})(\overline{x}^{\overline{v}\overline{w}} \vee \overline{y}^{\overline{v}\overline{w}})$. Here, we annotated variables $a$ with the assignment $\alpha$ of the universal variables, written $a^\alpha$. In Figure 2.2a we give a visual representation of the *full expansion tree*, that is, a tree whose root-to-leaf nodes represent all assignments $\alpha$ to universal variables.

Having to expand each and every universal variable and the resulting blow-up can

be, however, avoided in many cases by a method called *partial expansion*. The idea is that already a subset of universal assignments can rule out the existence of any Skolem function. Instantiating the universal assignment $\{v \mapsto \mathbf{T}, w \mapsto \mathbf{T}\}$ in our example above leads an unsatisfiable formula $(x^{vw})(y^{vw})(\overline{x}^{vw} \vee \overline{y}^{vw})$. Thus, there can be no Skolem function for $x$ and $y$ if there is no assignment satisfying the matrix on a single universal assignment. In Figure 2.2b we give a visual representation of the *partial expansion tree*, that is, an expansion tree that does not necessarily contain all assignments. The solvers RAReQS [Jan+16] and Ijtihad [Blo+18] base their reasoning on partial expansion. In Chapter 5 we show how to generalize this method for DQBF, which allows non-hierarchal dependencies, by using the notion of consistency of Skolem functions on a partial expansion tree.

We are now going to show how to integrate partial expansion into the clausal abstraction algorithm. This integration combines the results of the correctness proof given in Section 2.3.2 and the function extraction presented in the previous section. The key insight is, that if $\text{SOLVE}_\exists$ in ➔ Algorithm 2.3 determines that a quantified subformula $\Phi[\alpha'_S]$ is unsatisfiable, the witnessing Herbrand function corresponds to a *partial expansion tree* that can be used to strengthen the abstraction $\theta_X$. As we will show in Chapter 3, where we provide a proof-theoretic analysis, the resulting approach can be seen as a hybrid approach, enabling both expansion-based and $Q$-resolution-based, reasoning.

➔ Page 28

**Notation.** We start by providing necessary preliminaries and make the intuitive description given above more precise. For more details, we refer the reader to [JM15a]. A PARTIAL EXPANSION TREE for QBF $\Phi$ with $u$ universal quantifier blocks and matrix $\varphi$ is a rooted tree $\mathcal{T}$ such that every path $p_0 \xrightarrow{\alpha_1} p_1 \cdots \xrightarrow{\alpha_u} p_u$ in $\mathcal{T}$ from the root $p_0$ to some leaf $p_u$ has exactly $u$ edges and each edge $p_{i-1} \xrightarrow{\alpha_i} p_i$ is labeled with an assignment $\alpha_i$ to the universal variables at universal level $i$. Each path in $\mathcal{T}$ is uniquely defined by its labeling. Let $\mathcal{T}$ be a partial expansion tree and $P = p_0 \xrightarrow{\alpha_1} p_1 \cdots \xrightarrow{\alpha_u} p_u$ be a path from the root $p_0$ to some leaf $p_u$. For an existential variable $x$ we define $\mathit{expand\text{-}var}(P, x) = x^\alpha$ where $x^\alpha$ is a fresh variable and $\alpha = (\bigsqcup_{1 \le i \le u} \alpha_i)\,|_{dep(x)}$ is the universal assignment of the dependencies of $x$. For a propositional formula $\varphi$ define $\mathit{expand}(P, \varphi)$ as instantiating $\varphi$ with $\alpha_1, \ldots, \alpha_u$ and replacing every existential variable $x$ by $\mathit{expand\text{-}var}(P, x)$. We define $\mathit{expand}(\mathcal{T}, \Phi)$ as the conjunction of all $\mathit{expand}(P, \varphi)$ for each root-to-leaf path $P$ in $\mathcal{T}$.

**Expansion Refinement.** When the candidate verification algorithm returns $\text{Unsat}(\beta_S)$ in line 8 in Algorithm 2.3, we extract the partial expansion tree $\mathcal{T}$ that witnesses the unsatisfiability result. Extracting partial expansion trees during solving is closely related to function extraction. Given an existential node $\langle \exists X, \alpha_S \rangle$ in the recursion tree (see Section 2.4), we build the partial expansion tree by traversing the subtree of $\langle \exists X, \alpha_S \rangle$ and record every universal assignment $\alpha$ at an edge labeled with Unsat. In the recursion tree depicted in Figure 2.1 and root node $\langle \exists \{v, w\}, \{s_i \mapsto 0 \mid 1 \le i \le 5\} \rangle$, the extracted partial expansion tree $\mathcal{T}$ contains the paths $p_0 \xrightarrow{\{x \mapsto \mathbf{F}\}} p_1$ and $p_0 \xrightarrow{\{x \mapsto \mathbf{T}\}} p'_1$ from root $p_0$ to the leaves $p_1$ and $p'_1$.

Finally, given the partial expansion tree $\mathcal{T}$, we build the clausal abstraction for every clause in the expansion formula $expand(\mathcal{T}, \Phi)$. The resulting clauses are added to the abstraction $\theta_X$. Formally, after the clausal abstraction refinement in line 8, we update the abstraction by

$$\theta_X \leftarrow \theta_X \wedge \bigwedge_{C \in expand(\mathcal{T},\Phi)} clabs_{\exists X}(C) \ .$$

Correctness of this refinement follows from the soundness of the partial expansion, i.e., replacing the matrix $\varphi$ of some QBF $\Phi$ by $\varphi \wedge expand(\mathcal{T}, \Phi)$ preserves satisfiability for every expansion tree $\mathcal{T}$, and the correctness of the clausal abstraction. In the implementation, we can re-use the existent satisfaction variables $s_i$ of some clause $C_i$ for every corresponding expanded clause $C_i^\alpha$ as the literals bound by outer quantifier are equal, that is, $C_i^< = (C_i^\alpha)^<$.

## 2.6 Experimental Evaluation

We implemented the clausal abstraction algorithm in a tool called CAQE[5] (Clausal Abstraction for Quantifier Elimination) that takes as input a quantified Boolean formula encoded in the QDIMACS format. As the solver for the propositional abstractions, we used the SAT solver CryptoMiniSat [SNC09] version 5.0.1. We compare CAQE against publicly available QBF solvers that support the QDIMACS format, namely DepQBF [LE17] version 6.03, DynQBF [CW16] version 1.1.1, GhostQ [Kli+10] version 2017, Qesto [JM15b] version 1.0, Qute [PSS17] version 1.1, and RAReQS [Jan+16] version 1.1. For our experiments, we used a machine with a 3.6 GHz quad-core Intel Xeon (E3-1271 v3) processor and 32 GB of memory. The timeout and memout were set to 10 minutes and 8 GB, respectively. We use the prenex CNF benchmark set from the QBF competition QBFEVAL'18 [6]. As preprocessors, we used Bloqqer [BLS11] version 031, HQSPre [Wim+17] version 1.4, and QRATPre+ [LE18b] version 1.0. The cactus plot given in Figure 2.3 shows the number of solved instances for the best combination of preprocessor and solver. Detailed solving results are shown in Table 2.1. CAQE solves overall most instances, followed by RAReQS and Qesto. Further, all solvers solved significantly more instances when using HQSPre compared to Bloqqer. At the same time, the improvement due to HQSPre is much smaller for the solvers CAQE and RAReQS that are based on (partial) expansion than for the other solvers, possibly due to the more aggressive in expansion of universal variables in HQSPre compared to Bloqqer.

**Extended Refinements.** We discuss the effect of the stronger refinements given in Section 2.3.3 and the expansion refinement given in Section 2.5. There is a tradeoff between the *precision* of the abstraction and the cost of these satisfiability calls. The more precise an abstraction, the more losing assignments are excluded, i.e., higher precision can potentially reduce the number of propositional satisfiability calls. Both presented optimizations can potentially improve precision, but both of them may also increase the time

[5] Source code available at https://github.com/ltentrup/caqe
[6] Available at http://www.qbflib.org/qbfeval18.php

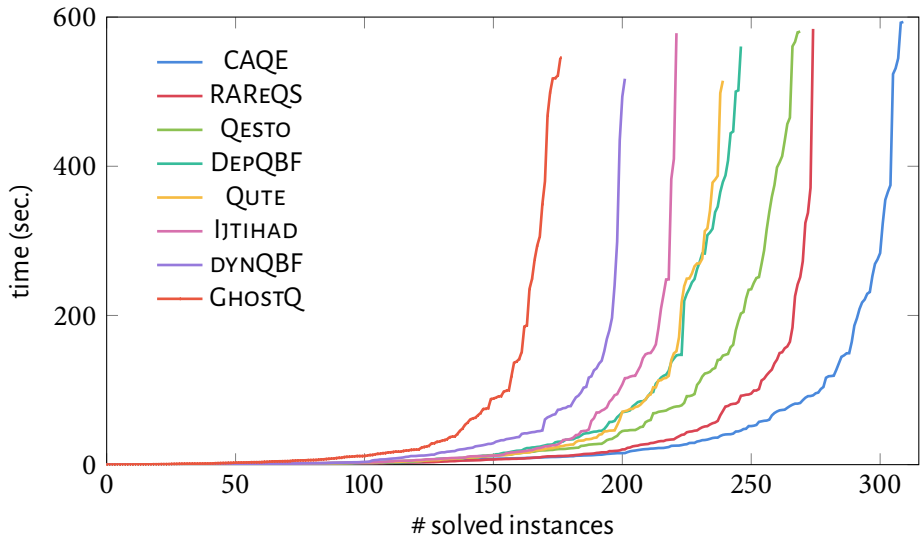



Figure 2.3: Cactus plot showing the number of solved instances on the prenex CNF benchmark set of QBFEVAL'18 using HQSPRE as preprocessor.

Table 2.1: Number of solved formulas by combinations of solvers and preprocessors on the prenex-CNF benchmark set of QBFEVAL'18. For every combination, we give the number of solved instances overall and broken down by result, that is, satisfiable and unsatisfiable.

| preprocessor | HQSPRE | | | BLOQQER | | | QRATPRE+ | | | none | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| solver | SOLVED | SAT | UNSAT | SOLVED | SAT | UNSAT | SOLVED | SAT | UNSAT | SOLVED | SAT | UNSAT |
| CAQE | **309** | **122** | **187** | **273** | **115** | **158** | **161** | 63 | **98** | 141 | 43 | 98 |
| RAREQS | 274 | 102 | 172 | 247 | 94 | 153 | 136 | 47 | 89 | 139 | 28 | **111** |
| QESTO | 269 | 108 | 161 | 196 | 89 | 107 | 127 | 52 | 75 | 98 | 29 | 69 |
| DEPQBF | 246 | 97 | 149 | 181 | 91 | 90 | 138 | **70** | 68 | 136 | 53 | 83 |
| QUTE | 239 | 79 | 160 | 159 | 58 | 101 | 116 | 40 | 76 | 94 | 17 | 77 |
| IJTIHAD | 201 | 75 | 146 | 198 | 74 | 124 | 125 | 39 | 86 | 131 | 23 | 108 |
| DYNQBF | 201 | 85 | 116 | 113 | 59 | 54 | 81 | 56 | 25 | 59 | 39 | 20 |
| GHOSTQ | – | – | – | – | – | – | – | – | – | **176** | **89** | 87 |

spent inside the SAT solver. Further, the relative performance of the optimizations depends on the benchmark set as well as the applied preprocessor. Thus, it is advisable to evaluate those optimizations in practice on a case-by-case basis. However, in our experiments, we found that the expansion refinement optimization vastly improves the number of solved instances independently of the preprocessor. Also, when comparing the running times directly, as done in the scatter plot depicted in Figure 2.4, the negative effect of the running time of the propositional SAT solver is reasonably small.

Regarding the stronger refinements, we found that the effect on instances preprocessed with HQSPRE is negligible. When using BLOQQER, however, the optimization im-

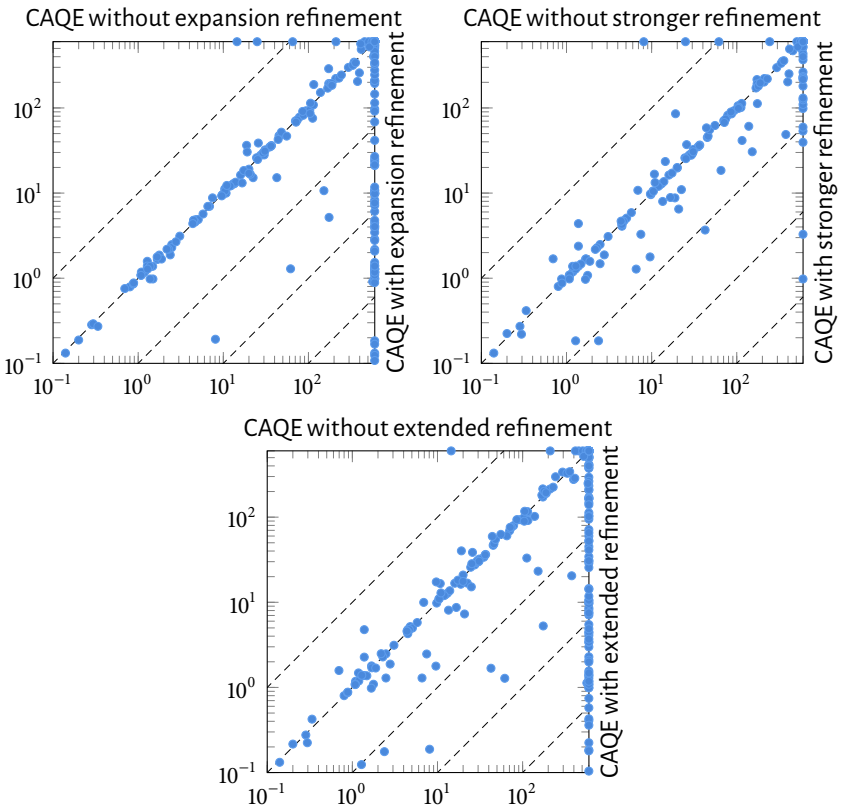



Figure 2.4: Scatter plot comparing the solving time (in sec.) of CAQE with and without extended refinements (expansion refinement and stronger refinement) and preprocessing using BLOQQER. Both axes have logarithmic scale.

proved the number of solved instances significantly. Further, the combination of both refinements, which we call extended refinement (which is also the default configuration used in the evaluation above), is the best performing variant of CAQE when using BLOQQER as the preprocessor. In our experiments, the combination performed better than any of the two refinements alone, indicating that they are in some sense orthogonal, as shown in the scatter plots in Figure 2.4.

**Algorithmic Choices.** In the following, we want to quantify the impact of the algorithmic choices described in this chapter. For this setup, we used a version of CAQE, which is close to the initial version of Section 2.3. Then, we enabled one of the algorithmic improvements mentioned in this chapter to evaluate their impact. The results are given in Table 2.2. The most impact in terms of solved instances has the expansion refinement, which can be explained by the corresponding improvement of the underlying proof system [Ten17]. The sum of additionally solved instances of the optimizations that are enabled by default (304) is smaller than the number of instances solved by CAQE (309), which hints at a positive synergy regarding the combination of individual optimizations.

Table 2.2: This table shows the impact of select algorithmic choices on a baseline version of CAQE using HQSPre as preprocessor. The baseline solves 229 instances on the prenex-CNF benchmark set of QBFEVAL'18. For every algorithmic choice, we give the difference of solved instances (Δ) compared to the baseline and detailed results (+) and (−).

| Algorithmic choice | default | described in | Δ | + | − |
|---|---|---|---|---|---|
| Expansion refinement | yes | Section 2.5 | +50 | 58 | 8 |
| Tree-shaped quantifier prefix | yes | Section 2.3.3 | +13 | 16 | 3 |
| Stronger refinement | yes | Section 2.3.3 | +6 | 7 | 1 |
| Sharing of abstraction literals | yes | Section 2.3.3 | +6 | 14 | 8 |
| Equivalence constraints in abstraction | no | [JM15b] | +3 | 5 | 2 |
| Backtracking over multiple quantifiers | no | Section 2.3.3 | −1 | 1 | 2 |
| Dropping redundant refinement literals | no | Section 2.3.3 | −1 | 6 | 7 |

## 2.7 Summary

In this section, we presented the clausal abstraction approach for solving quantified Boolean formulas. First, we presented the clausal abstraction algorithm for the one-alternation fragment of QBF in Section 2.2, followed by the generalization to QBF with arbitrary many alternations in Section 2.3. Among others, we included a detailed description of the basic algorithms, algorithmic improvements, and correctness proofs. The latter gave rise to the function extraction and the expansion-based refinements, as discussed in Section 2.4 and 2.5. Our experimental evaluation shows that the implementation of the clausal abstraction in the solver CAQE outperforms state-of-the-art solvers currently. As encouraging as those results are, they only are a preview of the potential of the clausal abstraction approach: Not everything described in Section 2.3.3 is fully implemented in CAQE[7], nor do we fully understand the interplay between the different optimizations yet, which, potentially, leaves much room for further solving improvements. In the following section, we take a proof-theoretic view and introduce a calculus for clausal abstraction. The comparison to existing proof systems reveals improvements that further improve the empirical solving performance.

[7] Currently, neither the non-static refinement from Equation 2.9 nor the matrix satisfiability preserving abstraction strengthening is implemented.

# Chapter 3

# A Proof System for Clausal Abstraction

In the previous chapter, we introduced the clausal abstraction algorithm for quantified Boolean formulas in prenex conjunctive normal form. We explained algorithmic details, proved the algorithm sound and complete, and detailed optimizations like the strong-unsat refinement. In this chapter, we focus on the proof-theoretical underpinnings. The motivation for this work was based on the following observation. The abstraction-based solvers RAReQS [Jan+12; Jan+16], Qesto [JM15b], and CAQE [RT15] share algorithmic similarities like working recursively over the structure of the quantifier prefix and using SAT solver to enumerate candidate solutions. However, instead of using partial expansions of the QBF as RAReQS does, the more recent approaches base their refinements on whether a set of clauses is satisfied or not. Despite those algorithmic similarities, the performance characteristics of the resulting solver in experimental evaluations are very different and in many cases orthogonal: While RAReQS tends to perform best on instances with a low number of quantifier alternations, Qesto and CAQE have an advantage in instances with many alternations [RT15].

Proof theory has been repeatedly used to improve the understanding of different solving techniques. For example, the proof calculus $\forall$Exp+Res [JM15a] has been developed to characterize aspects of expansion-based solving. The results of this section are three-fold. We introduce a proof system corresponding to clausal abstraction, which we call $\forall$Red+Res. The levelized nature of the clausal abstraction algorithm is reflected by the rules of $\forall$Red+Res, universal reduction and propositional resolution, which are applied to blocks of quantifiers. We show that this calculus is inherently different from $\forall$Exp+Res, explaining the empirical performance results. In fact, we show that $\forall$Red+Res is polynomially simulation equivalent to level-ordered $Q$-resolution [JM15a], which is the underlying proof calculus of search-based solvers. Then, we model the strong-unsat refinement as an additional proof rule in the $\forall$Red+Res calculus and show that the resulting proof calculus cannot by polynomially simulated by $\forall$Red+Res. Lastly, we show how to incorporate partial expansion, represented by the calculus $\forall$Exp+Res, as a new axiom rule in $\forall$Red+Res. The resulting calculus, $\forall$Red+$\forall$Exp+Res, is stronger than merely ap-

plying either calculi: We show that there is a family of formulas where both, ∀Red+Res and ∀Exp+Res have exponential refutations, whereas there exists a polynomial refutation in the ∀Red+∀Exp+Res calculus.

QBF proof theory is usually studied around the refutation of formulas, mainly inherited by methods developed around propositional satisfiability. To show the truth of a QBF, one merely has to proof the refutation of the negated formula. There is, however, also work on the theoretical foundations for true QBFs. The dual to $Q$-resolution, which applies *term*-resolution, is exponentially weaker than the negation and refutation method [JM17]. Beyond prior published results on the proof system of clausal abstraction [Ten17], we provide a proof system for true QBFs and show its equivalence to the dual of $Q$-resolution [LES16].

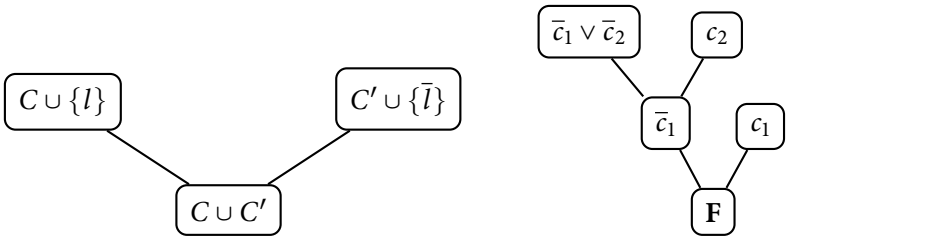
This chapter is based on work published in the proceedings of CAV [Ten17].

**Related Work.** $Q$-resolution [KKF95] is a variant of propositional refutation that is sound and refutation complete for QBF. There have been extensions proposed to $Q$-resolution, like long-distance resolution [ZM02] and universal resolution [Gel12], some of which are implemented in the QCDCL solver DepQBF [LB10; LE17]. Recently, there have also been extensions proposed that extend $Q$-resolution by more generalized axioms [LES16]. In some sense, the (∀**exp-res**) rule presented in Section 3.3 can be viewed as a new axiom rule for the ∀Red+Res calculus.

The ∀Exp+Res calculus [JM15a] was introduced to allow reasoning over expansion-based QBF solving, implemented by the QBF solver RAReQS [Jan+16]. The same desire motivated the work on ∀Red+Res, namely understanding the performance of the recently introduced QBF solvers CAQE [RT15] and Qesto [JM15b]. The incomparability of ∀Exp+Res and $Q$-resolution [JM15a; BCJ15] lead to the creation of stronger proof systems that unify those calculi, like IR-Calc [BCJ14b]. Further separation results, between variants of IR-Calc and variants of $Q$-resolution, were given in [BCJ15]. Those extensions, however, do not have accompanying implementations. This also applies to later work based on first-order resolution [Egl16].

There are two restrictions to $Q$-resolution studied in the literature, that is level-ordered and tree-like $Q$-resolution. Those restricted calculi were shown to be incomparable [MS16]. QCDCL based solvers usually exhibit level-ordered proofs (modulo unit-propagation) [Jan16]. It was shown that ∀Exp+Res polynomially simulates tree-like $Q$-resolution [JM15a]. A recent result [Bey+19] shows that DAG-like $Q$-resolution proofs of QBFs with an a priori fixed bound on the quantifier alternations can be efficiently transformed into an ∀Exp+Res proof. We showed that ∀Red+Res is polynomial simulation equivalent to level-ordered $Q$-resolution, which explains similar performance characteristics of the underlying solvers. Further, the strong-unsat rule presented in Section 3.2.1 can be viewed as a first step towards breaking the level-ordered restriction. The ∀Red+∀Exp+Res calculus polynomially simulates level-ordered and tree-like $Q$-resolution.

Quantified resolution asymmetric tautology (QRAT) is a proof calculus [HSB14a] introduced in the context of preprocessing and is able to express all preprocessing tech-




(a) Resolution rule (b) Resolution proof for $(c_1)(c_2)(\bar{c}_1 \vee \bar{c}_2)$

Figure 3.1: Visualization of the resolution rule as a graph.

niques implemented in the state-of-the-art preprocessor BLOQQER [HSB14b]. Recently, it was shown that it subsumes ∀Exp+Res [KS19].

While in the case of propositional SAT solving, symmetries and symmetry breaking are well-understood, there has been much less work on symmetries in QBF. Recently, Kauers and Seidl [KS18b] introduced a general framework for the characterization of symmetries in QBF. Further, they extended $Q$-resolution with a symmetry rule [KS18a].

## 3.1 Definitions

### 3.1.1 Resolution

Propositional RESOLUTION is a well-known method for refuting propositional formulas in conjunctive normal form (CNF). The resolution rule allows to *merge* two clauses that contain the same literal, but in opposite signs.

$$\frac{C \cup \{l\} \quad C' \cup \{\bar{l}\}}{C \cup C'} \text{ res}$$

Given a matrix $\varphi$, a RESOLUTION PROOF $\pi$ is a sequence of applications of the resolution rule where the clauses in $\varphi$ are the leaves. A propositional formula given as matrix $\varphi$ is unsatisfiable if, and only if, there is a resolution proof that derives the empty clause. We visualize resolution proofs by a graph where the nodes with indegree 0 are called the leaves and the unique node with outdegree 0 is called the root. We depict the graph representation of a resolution proof in Figure 3.1b. The *size* of a resolution proof is the number of nodes in the graph.

### 3.1.2 Proof Systems

We consider proof systems that can refute quantified Boolean formulas. To enable comparison between proof systems, one uses the concept of POLYNOMIAL SIMULATION . A proof system $P$ polynomially simulates ($p$-simulates) $P'$ if there is a polynomial $p$ such that for every formula $\Phi$ it holds that if there is a proof of $\Phi$ in $P'$ of size $n$, then there is a proof of $\Phi$ in $P$ whose size is less than $p(n)$. We call $P$ and $P'$ polynomial equivalent, if

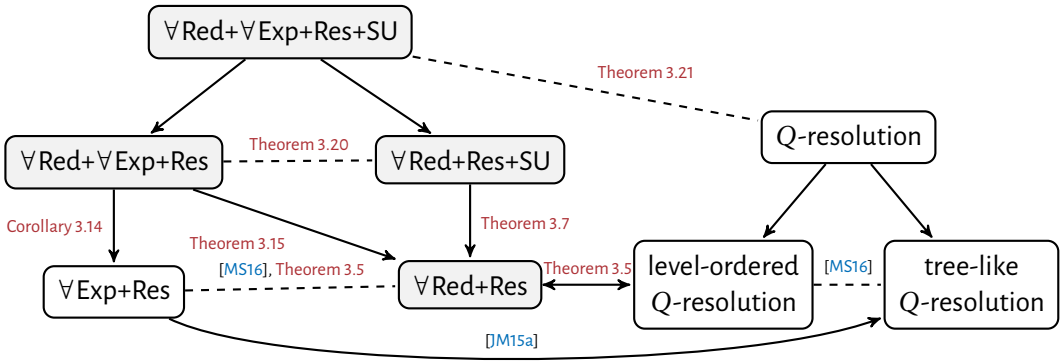



Figure 3.2: Overview of the proof systems and their relations. Solid arrows indicate $p$-simulation relation. Dashed lines indicate incomparability results. The gray boxes are the ones introduced in this section.

$P'$ additionally $p$-simulates $P$. A refutation based calculus (such as resolution) can proof validity by refuting the the negation of a valid formula.

Figure 3.2 gives an overview over the proof systems introduced in this chapter and their relation. An edge $P \to P'$ means that $P$ $p$-simulates $P'$ (transitive edges are omitted). A dashed line indicates incomparability results.

## 3.2 A Refutation Proof Calculus for Clausal Abstractions

Given a PCNF formula $\mathcal{Q}X_1 \ldots \mathcal{Q}X_n . \bigwedge_{1 \leq i \leq m} C_i$. We extend the notation $C_i^\circ$ (for $\circ \in \{<, \leq, =, \geq, >\}$) introduced in ➔ Section 2.3 by annotating the quantifier level $k$ explicitly. Given a clause $C_i$, we write $C_i^{=k}$ to denote the literals of $C_i$ that are bound at quantifier level $k$ ($1 \leq k \leq n$). As before, we use $C_i^{<k}$ and $C_i^{>k}$ to denote the literals bound before and after level $k$, respectively. Further, we define $C_i^{\leq 0} = C_i^{\geq n+1} = \varnothing$ for every $C_i \in \varphi$. We use $\mathcal{C}$ to denote a set of clauses and $\mathcal{Q}_k \in \{\exists, \forall\}$ to denote the quantification type of level $k$.

➔ Page 25

We start by defining the object on which our proof calculus ∀Red+Res is based on. A *proof object* $\mathcal{P}^k$ consists of a set of indices $\mathcal{P}$ where an index $i \in \mathcal{P}$ represents the $i$-th clause in the original matrix and $k$ denotes the $k$-th level of the quantifier hierarchy. We define an operation $lit(\mathcal{P}^k) = \bigcup_{i \in \mathcal{P}} C_i^{=k}$, that gives access to the literals of clauses contained in $\mathcal{P}^k$. The leaves in our proof system are singleton sets $\{i\}^z$ where $z$ is the maximum quantification level of all literals in clause $C_i$. The root of a refutation proof is the proof object $\mathcal{P}^0$ that represents the empty set, i.e., $lit(\mathcal{P}^0) = \varnothing$.

The rules of the proof system is given in Figure 3.3. It consists of three rules, an axiom rule (**init**) that generates leaves, a resolution rule (**res**), and a universal reduction rule (∀**red**). The latter two rules enable us to transform a premise that is related to quantifier level $k$ into a conclusion that is related to quantifier level $k-1$. The universal reduction rule and the resolution rule are used for universal and existential quantifier blocks, respectively.

$$\frac{\mathcal{P}_1^k \quad \mathcal{P}_2^k \quad \cdots \quad \mathcal{P}_j^k \quad \pi}{\left(\bigcup_{i\in\{1,\ldots,j\}} \mathcal{P}_i\right)^{k-1}}\ \textbf{res} \qquad \begin{array}{l} Q_k = \exists \\ \pi \text{ is a resolution refutation proof for } \bigwedge\limits_{1\leq i\leq j} lit(\mathcal{P}_i^k) \end{array}$$

$$\frac{\mathcal{P}^k}{\mathcal{P}^{k-1}}\ \forall\textbf{red} \qquad \begin{array}{l} Q_k = \forall \\ \forall l \in lit(\mathcal{P}^k).\, \overline{l} \notin lit(\mathcal{P}^k) \end{array}$$

$$\frac{}{\{i\}^k}\ \textbf{init} \qquad \begin{array}{l} 1 \leq i \leq m \\ C_i^{>k} = \varnothing \end{array}$$

Figure 3.3: The rules of the ∀Red+Res calculus.

**Resolution rule.** There is a close connection between (**res**) and the propositional resolution as (**res**) merges a set of proof objects $\mathcal{P}_i^k$ of level $k$ into a single proof object of level $k-1$. It does so by using a resolution proof for a propositional formula that is constructed from the premises $\mathcal{P}_i^k$. This propositional formula $\bigwedge_{1\leq i\leq j} lit(\mathcal{P}_i^k)$ contains *only* literals of level $k$. Intuitively, this rule can be interpreted as follows: A resolution proof over those clauses rules out any possible existential assignment at quantifier level $k$, thus, one of those clauses has to be satisfied at an earlier level.

**Universal reduction rule.** In contrast to (**res**), (∀**red**) works on single proof objects. It can be applied if level $k$ is universal and the premise does not encode a universal tautology, i.e., for every literal $l \in lit(\mathcal{P}^k)$, the negated literal $\overline{l}$ is not contained in $lit(\mathcal{P}^k)$.

**Graph representation.** A proof in the ∀Red+Res calculus can be represented as a directed acyclic graph (DAG). The nodes in the DAG are proof objects $\mathcal{P}^k$ and the edges represent applications of (**res**) and (∀**red**). The rule (**res**) is represented by a hyperedge that is labeled with the propositional resolution proof $\pi$. Edges representing the universal reduction can thus remain unlabeled without introducing ambiguity. The *size* of a ∀Red+Res proof is the number of nodes in the graph together with the number of inner (non-leaf, non-root) nodes of the containing propositional resolution proofs.

A *refutation* in the ∀Red+Res calculus is a proof that derives a proof object $\mathcal{P}^0$ at level 0. A proof for some $\mathcal{P}^k$ is a ∀Red+Res proof with root $\mathcal{P}^k$. Thus, a proof for $\mathcal{P}^k$ can also be viewed as a refutation for the formula $\mathcal{Q}X_{k+1}\ldots\mathcal{Q}X_n.\, \bigwedge_{i\in\mathcal{P}} C_i^{>k}$ starting with quantifier level $k+1$ and containing clauses represented by $\mathcal{P}$.

**Example 3.1.** Consider the following false QBF

$$\underbrace{\exists e_1}_{1}.\ \underbrace{\forall u_1}_{2}.\ \underbrace{\exists c_1, c_2}_{3}.\ \underbrace{(\overline{e}_1 \vee c_1)}_{C_1}\underbrace{(\overline{u}_1 \vee c_1)}_{C_2}\underbrace{(e_1 \vee c_2)}_{C_3}\underbrace{(u_1 \vee c_2)}_{C_4}\underbrace{(\overline{c}_1 \vee \overline{c}_2)}_{C_5}\ . \tag{3.1}$$

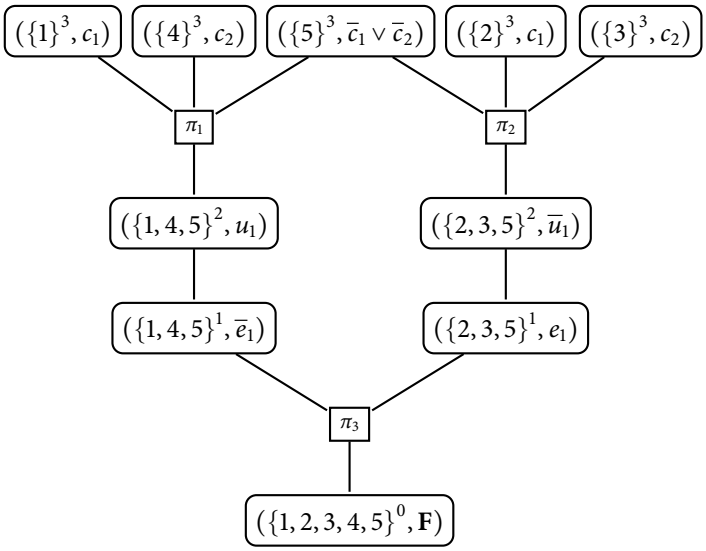



Figure 3.4: A ∀Red+Res refutation for the formula given in Equation 3.1.

The refutation in the ∀Red+Res calculus is given in Figure 3.4. In the nodes, we represent the proof objects $\mathcal{P}^k$ in the first component and the represented clause in the second component. The proof follows the structure of the quantifier prefix, i.e., it needs four levels to derive a refutation. The resolution proof $\pi_1$ for propositional formula

$$lit(\{1\}^3) \wedge lit(\{4\}^3) \wedge lit(\{5\}^3) \equiv (c_1)(c_2)(\bar{c}_1 \vee \bar{c}_2)$$

is depicted in Figure 3.1b.

*Remark* 3.2 (Correspondence to assignments of satisfaction variables $S$). The clauses indexed in a proof object $\mathcal{P}$ correspond to the partial assignments returned by Algorithm 2.3 and 2.4 in the case of unsatisfiability. This connection has already been used to extract Herbrand functions in Section 2.4.

In the following, we give a formal correctness argument and compare our calculus to established proof systems. A QBF proof system is *sound* if deriving a proof implies that the QBF is false and it is *refutational complete* if every false QBF has a proof.

**Theorem 3.3.** *∀Red+Res is sound and refutational complete for QBF.*

*Proof.* The completeness proof is carried out by induction over the quantifier prefix. For some quantified formula $\mathcal{Q}X$, we build proof objects for subformulas and combine them into a proof for $\mathcal{Q}X$.

*Induction base.* Let $\exists X.\, \varphi$ be a false QBF and $\varphi$ be propositional. Then (**res**) derives some $\mathcal{P}^0$, where $P$ is the set of indices of clauses contained in the resolution proof of $\varphi$, because resolution is complete for propositional formulas. Let $\forall X.\, \varphi$ be a false QBF and $\varphi$ be propositional. Picking an arbitrary (non-tautological) clause $C_i$ and applying (∀**red**) leads to $\{i\}^0$.

*Induction step.* Let $\exists X.\,\Phi$ be a false QBF, i.e., for all assignments $\alpha_X$ the QBF $\Phi[\alpha_X]$ is false. Hence, by induction hypothesis, there exists a $\forall$Red+Res proof for every $\Phi[\alpha_X]$. We transform those proofs in a way that they can be used to build a proof for $\Phi$. Let $P$ be a proof of $\Phi[\alpha_X]$. $P$ has a distinct root node (representing the empty set), that was derived using $(\forall\mathbf{red})$ as $\Phi[\alpha_X]$ starts with a universal quantifier. To embed $P$ in $\Phi$, we increment every level in $P$ by one, as $\Phi$ has one additional (existential) quantifier level. Then, instead of deriving the empty set, the former root node derives a proof object of the form $\mathcal{P}^1$. Let $N$ be the set of those former root nodes (one for each different $\alpha_X$). By construction, there exists a resolution proof $\pi$ such that the empty set can be derived by $(\mathbf{res})$ using $N$ (or a subset thereof). Assuming otherwise leads to the contradiction that some $\Phi[\alpha_X]$ is true.

Let $\forall X.\,\Phi$ be a false QBF, i.e., there is an assignment $\alpha_X$ such that the QBF $\Phi[\alpha_X]$ is false. Hence, by induction hypothesis, there exists a $\forall$Red+Res proof for $\Phi[\alpha_X]$. Applying $(\forall\mathbf{red})$ using $\alpha_X$ is a $\forall$Red+Res proof for $\Phi$.

For soundness it is enough to show that one cannot derive a clause using this calculus that changes the satisfiability. Let $\Phi = \mathcal{Q}X_1 \ldots \mathcal{Q}X_n.\,\bigwedge_{1\leq i\leq m} C_i$ be an arbitrary QBF. For every level $k$ and every $\mathcal{P}^k$ generated by the application of the $\forall$Red+Res calculus, we argue that $\Phi$ and $\mathcal{Q}X_1 \ldots \mathcal{Q}X_n.\,\bigwedge_{1\leq i\leq m} C_i \wedge \left(\bigvee_{i\in\mathcal{P}} C_i^{\leq k}\right)$ are equisatisfiable. Assume otherwise, then either $(\forall\mathbf{red})$ or $(\mathbf{res})$ have derived a $\mathcal{P}^k$ that would make $\Phi$ false. Again, by induction, one can show that if $(\forall\mathbf{red})$ derived a $\mathcal{P}^k$ that makes $\Phi$ false, the original premise $\mathcal{P}^{k+1}$ would have made $\Phi$ false; likewise, if $(\mathbf{res})$ derived a $\mathcal{P}^k$ that makes $\Phi$ false, the conjunction of the premises $\mathcal{P}_1^{k-1}, \mathcal{P}_2^{k-1}, \ldots, \mathcal{P}_j^{k-1}$ have made $\Phi$ false. □

The soundness proof shows that we can derive "summary" clauses for every proof object $\mathcal{P}^k$.

**Corollary 3.4.** *Given a proof object $\mathcal{P}^k$, adding the clause $C_i = \left(\bigvee_{i\in\mathcal{P}} C_i^{\leq k}\right)$ preserves satisfiability. Further, in a $\forall$Red+Res proof for the strengthened formula, one can replace $\mathcal{P}^k$ by $\{i\}^k$ (and one has to modify the subsequent proof accordingly).*

**Comparison to $Q$-resolution calculus.** $Q$-resolution [KKF95] is an extension of the (propositional) resolution rule to handle universal quantification. The universal reduction rule allows the removal of universal literal $u$ from a clause $C$ if no existential literal $l \in C$ depends on $u$. Resolution is only allowed on existential pivots. There is a further restriction on the applicability of the resolution rule, i.e., it is not allowed to produce tautology clauses. The definitions of $Q$-resolution proof and refutation are analogous to the propositional case.

There are two restricted classes of $Q$-resolution that are commonly considered, that is *level-ordered* and *tree-like* $Q$-resolution. A $Q$-resolution proof is level-ordered if resolution of an existential literal $l$ at level $k$ happens before every other existential literal with level $< k$. A $Q$-resolution proof is tree-like if the graph representing the proof has a tree shape.

We show that $\forall$Red+Res is polynomially equivalent to level-ordered $Q$-resolution, i.e., a proof in our calculus can be polynomially simulated in level-ordered $Q$-resolution

and vice versa. While this is straightforward from the definitions of both calculi, this is much less obvious if one compares the clausal abstraction algorithm, given in ➔ Page 25 ➔ Section 2.3, to QCDCL [ZM02].

**Theorem 3.5.** *$\forall$Red+Res and level-ordered $Q$-resolution are $p$-simulation equivalent.*

*Proof Sketch.* A $\forall$Red+Res proof can be transformed into a $Q$-resolution proof by replacing every node $\mathcal{P}^k$ by the clause $(\bigvee_{i\in\mathcal{P}} C_i^{\leq k})$ according to Corollary 3.4 and by replacing the hyper-edge labeled with $\pi$ by a graph representing the applications of the resolution rule. The resulting graph is a level-ordered $Q$-resolution proof: It derives the empty clause, contains the original clauses of the matrix as leaves and otherwise follows the levels of the $\forall$Red+Res proof. Note that the resulting $Q$-resolution proof does not contain tautologies: universal tautologies are checked in $(\forall\mathbf{red})$ and w.l.o.g. we can assume that resolution proofs $\pi$ in $(\mathbf{res})$ do not produce tautologies. Similarly, a level-ordered $Q$-resolution proof can be transformed into a $\forall$Red+Res proof by a step-wise transformation from leaves to the root. This way, one can track the clauses needed for constructing the proof objects $\mathcal{P}^k$ at every level $k$. □

Since the level-ordering constraint imposes an order on resolution and there are propositional formulas that have only exponentially larger refutations when an order is imposed [Goe92], level-ordered $Q$-resolution and $Q$-resolution have an exponential separation. Hence, also $\forall$Red+Res is in general exponentially weaker than unrestricted $Q$-resolution. In practice, and already noted by Janota and Marques-Silva [JM15a], solvers that are based on $Q$-resolution typically produce level-ordered $Q$-resolution proofs.

➔ Page 35 In ➔ Section 2.3.3, we presented an optimization that can generate new resolvents at level $k$ without recursion into deeper levels was described. We model this optimization as a new rule extending the $\forall$Red+Res calculus and show that this rule leads to an exponential separation.

### 3.2.1 Beyond Level-Orderedness

The goal of the *strong UNSAT refinement*, first described in the initial clausal abstraction paper [RT15], is to strengthen a certain type of refinements. The basic idea behind this optimization is that if the solver determines that, at an existential level $k$, a certain set of clauses $\mathcal{C}$ cannot be satisfied at the same time, then every alternative set of clauses $\mathcal{C}'$, that is equivalent with respect to the literals in levels $> k$, cannot be satisfied as well. We introduce the following proof rule that formalizes this intuition. We extend proof objects $\mathcal{P}^k$ such that they can additionally contain fresh literals, i.e., literals that were not part of the original QBF. Those literals are treated as they were bound at level $k$, i.e., they are contained in $lit(\mathcal{P}^k)$ and can thus be used in the premise of the rule $(\mathbf{res})$, but are not contained in the conclusion $\mathcal{P}^{k-1}$. Note that, adding literals to proof objects is merely syntactic sugar and can be desugared by introducing additional variables and clauses. An

example for the application of (**SU**) is given in the proof of Theorem 3.7 below.

$$\frac{(\mathcal{P} \cup \{i\})^k}{(\{a\} \cup \mathcal{P})^k \quad \{\overline{a}, j_1\}^k \quad \cdots \quad \{\overline{a}, j_n\}^k}\ \mathbf{SU} \qquad \begin{array}{l} Q_k = \exists, \\ C_j^{>k} \subseteq C_i^{>k} \text{ for all } j \in \{j_1, \ldots, j_n\}, \\ a \text{ fresh variable} \end{array}$$

**Theorem 3.6.** *The rule* (**SU**) *is sound.*

*Proof.* In a resolution proof at level $k$, one can derive the proof objects $(\mathcal{P} \cup \{j\})^k$ for $j \in \{j_1, \ldots, j_n\}$ using the conclusion of the rule (**SU**). If $\mathcal{P}$ contains a literal, one can derive a set of literal-free proof objects using resolution and the argumentation below holds for each of the elements in this set. Assume we have a proof for $(\mathcal{P} \cup \{i\})^k$ (premise), then the quantified formula $\forall X_{k+1} \ldots \mathcal{Q} X_n . \bigwedge_{i^* \in \mathcal{P}} C_{i^*}^{>k} \wedge C_i^{>k}$ is false. Thus, the QBF with the same quantifier prefix and matrix, extended by some clause $C_j^{>k}$ for $j \in \{j_1, \ldots, j_n\}$, is still false. Since every $C_j$ subsumes $C_i$ with respect to quantifier level greater than $k$ ($C_j^{>k} \subseteq C_i^{>k}$), the clause $C_i^{>k}$ is redundant and can be eliminated without changing satisfiability. Thus, the resulting quantified formula $\forall X_{k+1} \ldots \mathcal{Q} X_n . \bigwedge_{i^* \in \mathcal{P}} C_{i^*}^{>k} \wedge C_j^{>k}$ is false and there exists a ∀Red+Res proof for $(\mathcal{P} \cup \{j\})^k$. □

**Theorem 3.7.** *∀Red+Res does not p-simulate ∀Red+Res+SU.*

*Proof.* We use the family of formulas $\mathrm{CR}_n$ that was used to show that level-ordered $Q$-resolution cannot $p$-simulate ∀Exp+Res [JM15a]. We show that $\mathrm{CR}_n$ has a polynomial refutation in the ∀Red+Res+SU calculus, but has only exponential refutations without (**SU**). The latter follows from Theorem 3.5 and the results by Janota and Marques-Silva [JM15a].

The formula $\mathrm{CR}_n$ has the quantifier prefix $\exists x_{11}, \ldots, x_{nn} \forall z \exists a_1, \ldots, a_n, b_1, \ldots, b_n$ and the matrix is given by

$$\underbrace{\left(\bigvee_{i \in 1..n} \overline{a}_i\right)}_{A} \wedge \underbrace{\left(\bigvee_{i \in 1..n} \overline{b}_i\right)}_{B} \wedge \bigwedge_{i,j \in 1..n} \underbrace{(x_{ij} \vee z \vee a_i)}_{C_{ij}} \wedge \underbrace{(\overline{x}_{ij} \vee \overline{z} \vee b_j)}_{C_{\overline{ij}}} \ . \qquad (\mathrm{CR}_n)$$

One can interpret the constraints as selecting rows and columns in a matrix where $i$ selects the row and $j$ selects the column, e.g., for $n = 3$ it can be visualized as follows:

| | | | | | |
|---|---|---|---|---|---|
| $x_{11} \vee z \vee a_1$ | $\overline{x}_{11} \vee \overline{z} \vee b_1$ | $x_{12} \vee z \vee a_1$ | $\overline{x}_{12} \vee \overline{z} \vee b_2$ | $x_{13} \vee z \vee a_1$ | $\overline{x}_{13} \vee \overline{z} \vee b_3$ |
| $x_{21} \vee z \vee a_2$ | $\overline{x}_{21} \vee \overline{z} \vee b_1$ | $x_{22} \vee z \vee a_2$ | $\overline{x}_{22} \vee \overline{z} \vee b_2$ | $x_{23} \vee z \vee a_2$ | $\overline{x}_{23} \vee \overline{z} \vee b_3$ |
| $x_{31} \vee z \vee a_3$ | $\overline{x}_{31} \vee \overline{z} \vee b_1$ | $x_{32} \vee z \vee a_3$ | $\overline{x}_{32} \vee \overline{z} \vee b_2$ | $x_{33} \vee z \vee a_3$ | $\overline{x}_{33} \vee \overline{z} \vee b_3$ |

In the following, we give a ∀Red+Res+SU proof. A visualization of the proof steps is depicted in Figure 3.5. First, we derive the proof object $\mathcal{P}_{x1}^1 = \{i1 \mid i \in 1..n\}^1$ ($lit(\mathcal{P}_{x1}^1) = \bigvee_{i \in 1..n} x_{i1}$) by applying the resolution and reduction rule on the clauses $A$ and $C_{11}, C_{21}, \ldots, C_{n1}$. Likewise, we derive the proof object $\mathcal{P}_0^1 = \{\overline{1j} \mid j \in 1..n\}^1$ ($lit(\mathcal{P}_0^1) = \bigvee_{j \in 1..n} \overline{x}_{1j}$) using the clauses $B$ and , $C_{\overline{12}}, \ldots, C_{\overline{1n}}$. Applying the rule (**SU**) on $\mathcal{P}_0^1$ results in

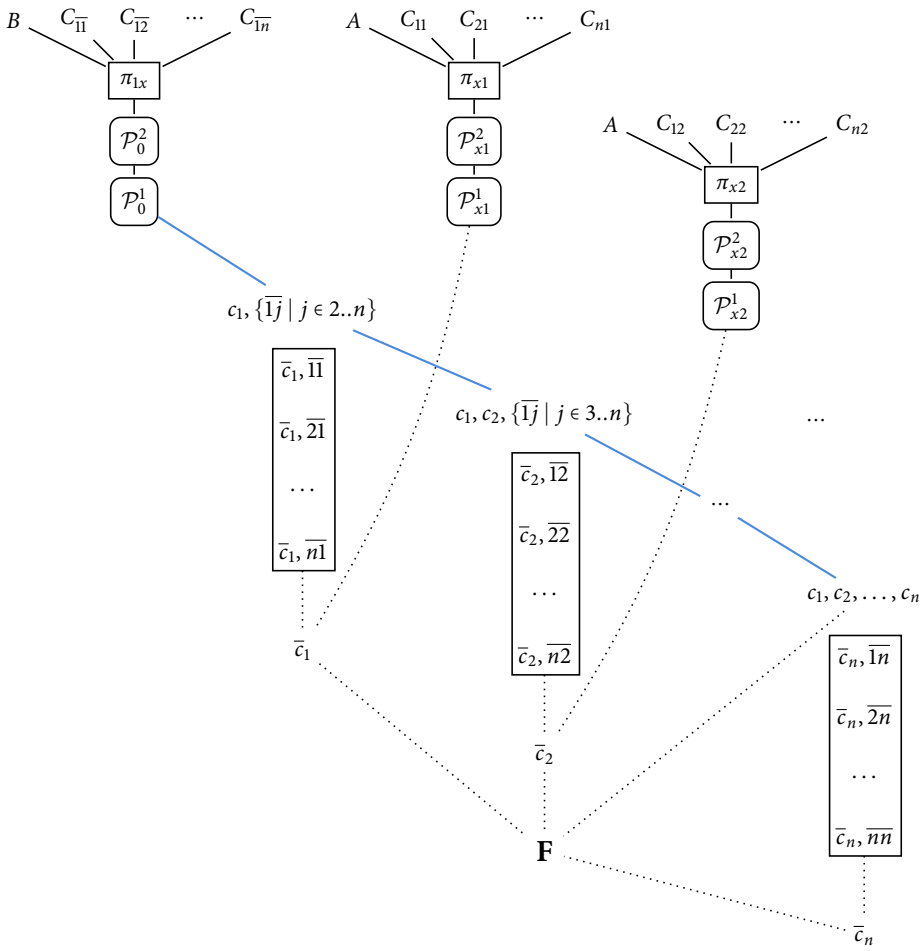



Figure 3.5: Schematic visualization of the proof of Theorem 3.7. The blue edges indicate applications of the (**SU**) rule and dotted edges denote propositional resolution steps.

$\mathcal{P}^1_1 = (\{c_1\} \cup \{\overline{1j} \mid j \in 2..n\})^1$ and $\{\overline{c}_1, \overline{11}\}^1, \{\overline{c}_1, \overline{21}\}^1, \ldots, \{\overline{c}_1, \overline{n1}\}^1$ where $c_1$ is a fresh variable. Further $n-1$ applications of the rule (**SU**) starting on $\mathcal{P}^1_1$ lead to $\mathcal{P}^1_n = \{c_j \mid j \in 1..n\}^1$ and the proof objects $\{\overline{c}_j, \overline{ij} \mid i, j \in 1..n\}^1$, where $c_j$ are fresh variables, as all clauses in a column are equivalent with respect to the inner quantifiers (contain $\overline{z} \vee b_j$).

Using $\mathcal{P}^1_{x1}$ and $\{\overline{c}_1, \overline{11}\}^1, \{\overline{c}_1, \overline{21}\}^1, \ldots, \{\overline{c}_1, \overline{n1}\}^1$ from the first (**SU**) application, we derive the singleton set $\{\overline{c}_1\}$ using $n$ resolution steps ($lit(\mathcal{P}^1_{x1}) = \bigvee_{i \in 1..n} x_{i1}$ and $lit(\{\overline{c}_1, \overline{i1}\}^1) = \{\overline{c}_1, \overline{x}_{i1}\}$). Analogously, one derives the singletons $\{\overline{c}_2\} \ldots \{\overline{c}_n\}$ and together with $\mathcal{P}^1_n = \{c_j \mid j \in 1..n\}$ the empty set is derived. Thus, there exists a resolution proof leading to a proof object $\mathcal{P}^0$. The size of the overall proof is polynomial in the size of the formula. □

Despite being stronger than plain ∀Red+Res, the extended calculus is still incomparable to ∀Exp+Res.

**Corollary 3.8.** *∀Red+Res+SU does not p-simulate ∀Exp+Res.*

*Proof.* We use a modification of formula ($\mathrm{CR}_n$), which we call $\mathrm{CR}'_n$ in the following. The single universal variable $z$ is replaced by a number of variables $z_{ij}$ for every pair $i, j \in 1..n$.

$$\left(\bigvee_{i\in 1..n} \overline{a}_i\right) \wedge \left(\bigvee_{i\in 1..n} \overline{b}_i\right) \wedge \bigwedge_{i,j\in 1..n} \underbrace{(x_{ij} \vee z_{ij} \vee a_i)}_{C_{ij}} \wedge \underbrace{(\overline{x}_{ij} \vee \overline{z}_{ij} \vee b_j)}_{C_{\overline{ij}}} \qquad (\mathrm{CR}'_n)$$

We argue that the rule (**SU**) is never applicable. Analogously to the proof of Theorem 3.7, we can the proof object $\mathcal{P}^1_0 = \{\overline{1j} \mid j \in 1..n\}^1$ ($lit(\mathcal{P}^1_0) = \bigvee_{j\in 1..n} \overline{x}_{1j}$). Where before, we could apply (**SU**) as the clauses $C_{\overline{11}}, C_{\overline{21}}, \ldots, C_{\overline{n1}}$ are equal w.r.t. the inner variables (in ($\mathrm{CR}_n$) these are $z$ and $b_1, \ldots, b_n$). In Equation $\mathrm{CR}'_n$ all of those clauses are different due to the variables $z_{11}, z_{21}, \ldots, z_{n1}$. Due to symmetry, this is the case for any derivable proof object $\mathcal{P}^1$.

Hence, the proof system is as strong as level-ordered $Q$-resolution which has an exponential refutation of $\mathrm{CR}_n$ while ∀Exp+Res has a polynomial refutation since the expansion tree has still only two branches [JM15a]. □

When compared to $Q$-resolution, the rule (**SU**) can be interpreted as a step towards breaking the level-ordered constraint inherent to ∀Red+Res. The calculus, however, is not as strong as $Q$-resolution.

**Corollary 3.9.** *∀Red+Res+SU does not p-simulate Q-resolution.*

*Proof.* The formula $\mathrm{CR}'_n$ from the previous proof has a polynomial (tree-like) $Q$-resolution proof. The proof for $\mathrm{CR}_n$ [MS16] can be adapted for $\mathrm{CR}'_n$. □

Both results follow from the fact that the rule (**SU**) as presented is not applicable to the formula $\mathrm{CR}'_n$. Where in $\mathrm{CR}_n$, the clauses $C_{\overline{ij}}$ are equal with respect to the inner quantifier when $j$ is fixed ($\overline{z} \vee b_j$), in $\mathrm{CR}'_n$ they are all different ($\overline{z}_{ij} \vee b_j$). This difference is only due to the universal variables $z_{ij}$. Thus, we propose a stronger version of the rule (**SU**), called (**SU**$^+$), that does the subset check only on the existential variables. For the universal literals, one additionally has to make sure that no resolvent produces a tautology (as it is the case in $\mathrm{CR}'_n$).

$$\frac{(\mathcal{P} \cup \{i\})^k}{(\{a\} \cup \mathcal{P})^k \quad \{\overline{a}, j_1\}^k \cdots \{\overline{a}, j_n\}^k}\ \mathbf{SU}^+$$

$Q_k = \exists$,
$C_j^{>k}|_\exists \subseteq C_i^{>k}|_\exists$ for all $j \in \{j_1, \ldots, j_n\}$,
$C_j|_\forall \cup \bigcup_{i\in\mathcal{P}} C_i^{>k}|_\forall$ is no tautology for all $j \in \{j_1, \ldots, j_n\}$,
$a$ fresh variable

**Theorem 3.10.** *The rule (**SU**$^+$) is sound.*

*Proof.* We adapt the soundness proof of (**SU**) given in Theorem 3.6. The only change that is needed is the argumentation why there is a ∀Red+Res proof for the newly created

clauses $C_j^k$ for $j \in \{j_1, \ldots, j_n\}$. Remember that the existential variables are a subset, i.e., $C_j^{>k}|_\exists \subseteq C_i^{>k}|_\exists$ which means that if there is a propositional resolution proof involving variables in $C_i^{>k}|_\exists$, there is a (potentially smaller) propositional resolution proof with the variables in $C_j^{>k}|_\exists$. The universal variables are eliminated by reduction, thus, we have to make sure that no universal tautology can be created with the other clauses in the proof object $\mathcal{P}$ which is guaranteed by the side condition of $(\mathbf{SU}^+)$. □

**Corollary 3.11.** *$\mathrm{CR}'_n$ has a polynomial refutation using $\forall$Red+Res+SU$^+$.*

*Proof.* The proof is the same as the proof for Theorem 3.7. $(\mathbf{SU}^+)$ is applicable as the $z_{ij}$ only appear positive in clauses $C_{ij}$ and thus, they cannot produce tautologies. □

## 3.3 Integrating Partial Expansion

### 3.3.1 The ∀Exp+Res Axiom Rule

The levelized nature of the proof system allows us to introduce additional rules that can reason about quantified subformulas. In the following, we introduce such a rule that allows us to use the ∀Exp+Res calculus [JM15a] within a ∀Red+Res proof. This models the partial-expansion refinement given in ➔ Section 2.5. In the following, we re-use notation introduced for the expansion-refinement.

➔ Page 41

We allow to use the expansion rule $(\forall\mathbf{exp\text{-}res})$ in every existential level of a ∀Red+Res proof tree. By $\mathcal{C}^{\geq k}$ we denote a set of clauses that only contain literals bound at level $\geq k$.

$$\frac{\mathcal{T} \quad \mathcal{C}^{\geq k} \quad \pi}{\mathcal{P}^{k-1}}\ \forall\mathbf{exp\text{-}res}$$

$Q_k = \exists$, $\pi$ is a resolution refutation of the expansion formula $expand(\mathcal{T}, \exists X_k.\, \forall X_{k+1} \cdots \exists X_m.\, \mathcal{C}^{\geq k})$
$\mathcal{P}^{k-1} = \{i \mid C_i \in \mathcal{C}\}^{k-1}$

The rule states that if there is a universal expansion of the quantified Boolean formula $\exists X_k.\, \forall X_{k+1} \cdots \exists X_m.\, \mathcal{C}^{\geq k}$ and a resolution refutation $\pi$ for this expansion, then there is no existential assignment that satisfies clauses $\mathcal{C}$ from level $k$. The size of the expansion rule is the sum of the size of the expansion tree and resolution proof [JM15a].

**Example 3.12.** We demonstrate the interplay between $(\forall\mathbf{exp\text{-}res})$ and the ∀Red+Res calculus on the following formula

$$\overbrace{\exists e_1}^{1}.\ \overbrace{\forall u_1}^{2}.\ \overbrace{\exists c_1, c_2}^{3}.\ \overbrace{\forall a}^{4}.\ \overbrace{\exists b}^{5}.\ \exists x.\ \overbrace{\forall z}^{6}.\ \overbrace{\exists t}^{7}.$$

$$\underbrace{(\overline{e}_1 \vee c_1)}_{1}\ \underbrace{(\overline{u}_1 \vee c_1)}_{2}\ \underbrace{(e_1 \vee c_2)}_{3}\ \underbrace{(u_1 \vee c_2)}_{4}\ \underbrace{(\overline{c}_1 \vee \overline{c}_2 \vee \overline{b} \vee \overline{a})}_{5}\ \underbrace{(z \vee t \vee b)}_{6}\ \underbrace{(\overline{z} \vee \overline{t})}_{7}\ \underbrace{(x \vee \overline{t})}_{8}\ \underbrace{(\overline{x} \vee t)}_{9}$$

To apply $(\forall\mathbf{exp\text{-}res})$, we use the clauses 5–9 from quantifier level 5, i.e., $\mathcal{C}^{\geq 5} = \{(\overline{b})(z \vee t \vee b)(\overline{z} \vee \overline{t})(x \vee \overline{t})(\overline{x} \vee t)\}$. The corresponding quantifier prefix is

$\exists b \exists x \forall z \exists t$. Using the complete expansion of $z$ ($\{z \to 0, z \to 1\}$) as the expansion tree $\mathcal{T}$, we get the following expansion formula

$$(\overline{b})(t^{\{z\to 0\}} \vee b)(x \vee \overline{t}^{\{z\to 0\}})(\overline{x} \vee t^{\{z\to 0\}})(\overline{t}^{\{z\to 1\}})(x \vee \overline{t}^{\{z\to 1\}})(\overline{x} \vee t^{\{z\to 1\}}) \ ,$$

which has a simple resolution proof $\pi$. The conclusion of ($\forall$**exp-res**) leads to the proof object $\{5, 6, 7, 8, 9\}^4$, but only clause 5 contains literals bound before quantification level 5. After a universal reduction, the proof continues as described in Example 3.1.

**Theorem 3.13.** *The* ($\forall$**exp-res**) *rule is sound.*

*Proof.* Assume otherwise, then one would be able to derive a proof object $\mathcal{P}^{k-1}$ that is part of a $\forall$Red+Res refutation proof for true QBF $\Phi$. Thus, the clause corresponding to $\mathcal{P}^{k-1}$ (cf. proof of Theorem 3.3) ($\bigvee_{i\in\mathcal{P}} C_i^{\leq k}$) made $\Phi$ false. However, the same clause can be derived directly by applying the expansion $\mathcal{T}$ to the original QBF, i.e., expanding universal variables beginning with quantification level $k+1$, and propositional resolution on the resulting expansion formula. Thus, this clause can be conjunctively added to the matrix without changing satisfiability, leading to a contradiction. □

The resulting proof system can be viewed as a unification of the CEGAR approaches for solving quantified Boolean formulas [Jan+16; JM15b; RT15]. As $\forall$Red+Res and $\forall$Exp+Res are incomparable due to Theorem 3.5 and [MS16], the inclusion of the rule ($\forall$**exp-res**) makes the resulting proof system $\forall$Red+$\forall$Exp+Res exponentially more succinct.

**Corollary 3.14.** *$\forall$Exp+Res does not $p$-simulate $\forall$Red+$\forall$Exp+Res.*

*Proof.* $\forall$Exp+Res does not $p$-simulate level-ordered $Q$-resolution [MS16]. □

More interestingly, the combination of both rules makes the proof system stronger than merely choosing between expansion and resolution proof upfront.

**Theorem 3.15.** *There is a family of quantified Boolean formulas that has polynomial refutation in $\forall$Red+$\forall$Exp+Res, but has only exponential refutations in $\forall$Red+Res and $\forall$Exp+Res.*

*Proof.* For this proof, we take two formulas that are hard for $Q$-resolution and $\forall$Exp+Res, respectively. We build a new family of formulas that has a polynomial refutation in $\forall$Red+$\forall$Exp+Res, but only exponential refutations in $\forall$Red+Res and $\forall$Exp+Res.

The first formula we consider is formula (2) form [JM15a], that we call $\mathrm{DAG}_n$ in the following:

$$\begin{aligned}&\exists e_1 \forall u_1 \exists c_1 c_2 \cdots \exists e_n \forall u_n \exists c_{2n-1} c_{2n}.\\ &(\bigvee_{i\in 1\ldots 2n} \overline{c}_i) \wedge \bigwedge_{i\in 1\ldots n} (\overline{e}_i \vee c_{2i-1}) \wedge (\overline{u}_i \vee c_{2i-1}) \wedge (e_i \vee c_{2i}) \wedge (u_i \vee c_{2i})\end{aligned} \qquad (\mathrm{DAG}_n)$$

It is known that $\mathrm{DAG}_n$ has a polynomial level-ordered $Q$-resolution proof and only exponential $\forall$Exp+Res proofs [JM15a]. As the second formula, we use the $\mathrm{QParity}_n$ formula [BCJ15]

$$\exists x_1 \cdots x_n \forall z \exists t_2 \cdots t_n.\, \mathrm{xor}(x_1, x_2, t_2) \wedge \bigwedge_{i\in 3\ldots n} \mathrm{xor}(t_{i-1}, x_i, t_i) \wedge (z \vee t_n) \wedge (\overline{z} \vee \overline{t}_n) \quad (\mathrm{QParity}_n)$$

where $\mathrm{xor}(o_1, o_2, o) = (\overline{o}_1 \vee \overline{o}_2 \vee \overline{o}) \wedge (o_1 \vee o_2 \vee \overline{o}) \wedge (\overline{o}_1 \vee o_2 \vee o) \wedge (o_1 \vee \overline{o}_2 \vee o)$ defines $o$ to be equal to $o_1 \oplus o_2$. $\mathrm{QParity}_n$ has a polynomial $\forall$Exp+Res refutation but only exponential $Q$-resolution refutations [BCJ15]. We construct the following formula

$$\exists e_1 \forall u_1 \exists c_1 c_2 \cdots \exists e_n \forall u_n \exists c_{2n-1} c_{2n}.\ \forall a \exists b.\ \exists x_1 \cdots x_n \forall z \exists t_2 \cdots t_n.$$
$$\bigwedge_{i \in 1 \ldots n} (\overline{e}_i \vee c_{2i-1}) \wedge (\overline{u}_i \vee c_{2i-1}) \wedge (e_i \vee c_{2i}) \wedge (u_i \vee c_{2i}) \wedge$$
$$(\overline{a} \vee \overline{b} \vee \bigvee_{i \in 1 \ldots 2n} \overline{c}_i) \wedge \mathrm{xor}(x_1, x_2, t_2) \wedge \bigwedge_{i \in 3 \ldots n} \mathrm{xor}(t_{i-1}, x_i, t_i) \wedge (z \vee t_n \vee b) \wedge (\overline{z} \vee \overline{t}_n)$$

Not that $a$ is pure in the formula above as it is only used such that the existential variable $b$ that connects both formulas does not collapse with the previous quantifier block. We argue in the following that this formula has a polynomial refutation in $\forall$Red+$\forall$Exp+Res. First, using ($\forall$**exp-res**) we can derive the proof object containing the clause $(\overline{a} \vee \bigvee_{i \in \{1 \ldots 2n\}} \overline{c}_i)$ using the expansion tree $\mathcal{T} = \{z \to 0, z \to 1\}$ and the clauses from the last row (analogue to Example 3.12). This proof is analogous to the $\forall$Exp+Res proof for QParity [BCJ15] and, thus, polynomial in size. After applying universal reduction, the proof object representing clause $(\bigvee_{i \in \{1 \ldots 2n\}} \overline{c}_i)$ can be derived. For the remaining formula, there is a polynomial and level-ordered resolution proof [JM15a], thus, the formula has a polynomial $\forall$Red+$\forall$Exp+Res proof.

There is no polynomial $Q$-resolution proof, because deriving $(\bigvee_{i \in \{1 \ldots 2n\}} \overline{c}_i)$ is exponential in $Q$-resolution. Likewise, there is no polynomial $\forall$Exp+Res proof as the formula after deriving this clause has only exponential $\forall$Exp+Res refutations. □

One question that remains open, is how the new proof system compares to unrestricted $Q$-resolution. We already know that the new proof system polynomially simulates both tree-like $Q$-resolution as well as level-ordered $Q$-resolution.

**Theorem 3.16.** *$\forall$Red+$\forall$Exp+Res does not p-simulate $Q$-resolution.*

*Proof Sketch.* We construct a formula that is hard for expansion and level-ordered $Q$-resolution, but easy for (unrestricted) $Q$-resolution. We have already seen in the proof of Theorem 3.15 that $\mathrm{DAG}_n$ is hard for $\forall$Exp+Res but easy for $Q$-resolution. However, the $Q$-resolution proof of $\mathrm{DAG}_n$ is level-ordered. Hence, we need an additional formula that is hard to refute for level-ordered $Q$-resolution. We use the modified pigeon hole formula from [Goe92] where unrestricted resolution has polynomial proofs and resolution proofs that are restricted to a certain variable ordering are exponential. Using universal quantification, one can impose an arbitrary order on a level-ordered $Q$-resolution proof, thus, there is a quantified Boolean formula which has only exponential level-ordered $Q$-resolution but has a polynomial $Q$-resolution proof. The disjunction of those two formulas gives the required witness. This formula is easy to refute for $Q$-resolution, but the first one is hard for $\forall$Exp+Res and the second is hard for level-ordered $Q$-resolution. □

### 3.3.2 Expanding Conflict Clauses

The applicability of the expansion rule introduced in the previous section is limited to be an axiom rule, that is, only using clauses of the original formula. In this section, we

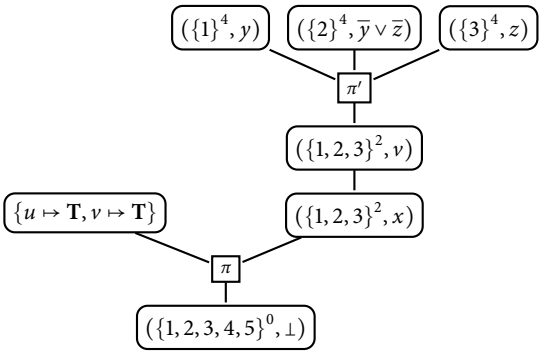



Figure 3.6: A refutation using the rule ($\forall$**exp-res**$^+$).

remove this limitation in the proof system by allowing universal expansion to also use clauses that are derived by the application of other proof rules. In order to allow the ($\forall$**exp-res**) to be applied as a non-axiom rule, we use the knowledge that proof object $\mathcal{P}^k$ for $\mathcal{Q}_k = \exists$ can be represented equivalently as clauses. In the Corollary 3.4, we have established that a proof object $\mathcal{P}^k$ corresponds to the derived clause $\bigvee_{i \in \mathcal{P}} C_i^{\leq k}$. Then, the only change to ($\forall$**exp-res**) is to allow an arbitrary number of already derived proof objects in the premise and we have to modify the expansion formula to include the corresponding conflict clauses. We call the resulting rule ($\forall$**exp-res**$^+$).

$$\frac{\mathcal{T} \quad \mathcal{C}^{\geq k} \quad \mathcal{P}_1^{k_1} \dots \mathcal{P}_j^{k_j} \quad \pi}{\mathcal{P}^{k-1}} \; \forall\textbf{exp-res}^+$$

$Q_k = \exists$, $\pi$ is a resolution refutation of the formula $expand(\mathcal{T}, \exists X_k. \forall X_{k+1} \dots \exists X_m. (\mathcal{C} \cup \{\bigvee_{i \in \mathcal{P}_1} C_i^{\leq k_1}, \dots \bigvee_{i \in \mathcal{P}_j} C_i^{\leq k_j}\})^{\geq k})$
$\mathcal{P}^{k-1} = \left(\{i \mid C_i \in \mathcal{C}\} \cup \bigcup_{j \in \{1,\dots,j\}} \mathcal{P}_j\right)^{k-1}$
$k_i > k$ for every $i \in \{1, \dots, j\}$

**Example 3.17.** Consider the formula

$$\underbrace{\forall u}_{1}. \underbrace{\exists x}_{2}. \underbrace{\forall v}_{3}. \underbrace{\exists y, z}_{4}. \underbrace{(v \vee y)}_{C_1} \underbrace{(\overline{y} \vee \overline{z})}_{C_2} \underbrace{(z \vee x)}_{C_3} \underbrace{(u \vee x)}_{C_4} \underbrace{(\overline{u} \vee \overline{x})}_{C_5} .$$

An application of the ($\forall$**exp-res**$^+$) is given in Figure 3.6.

**Corollary 3.18.** *The non-axiom* ($\forall$**exp-res**$^+$) *rule is sound.*

*Proof.* Follows from the soundness of the axiom ($\forall$**exp-res**) (Theorem 3.13) together with the soundness of the $\forall$Red+Res proof system (Theorem 3.3) and the correspondence of the proof objects with derived clauses in $Q$-resolution (Theorem 3.5). □

### 3.3.3 Dependency Schemes

A dependency scheme [SS09a] is a technique to detect *spurious* dependencies in a quantified Boolean formula and have been repeatedly used in QBF solvers [LB10; PSS17] as well

as preprocessors [Wim+15]. Clausal abstraction, however, is inherently unable to benefit from dependency schemes: After all, the algorithm recurses over the quantifier prefix by considering maximal blocks of quantifiers of the same type. When using partial expansion, a dependency scheme can be used to eliminate spurious dependencies before building the propositional expansion. The soundness of using dependency schemes in partial expansion was shown in [Bey+18].

Given a QBF $\Phi$, a dependency scheme $\mathcal{D} \subseteq \mathcal{V} \times \mathcal{V}$ is a binary relation between variables of $\Phi$, where $(v_1, v_2) \in \mathcal{D}$ indicates that $v_2$ depends on $v_1$. Pairs $(v_1, v_2)$ not included in $\mathcal{D}$ are considered *independent*. The *trivial* dependency scheme $\mathcal{D}^{\text{trv}}$ models the dependencies that are given by the quantifier prefix: A pair $(v_1, v_2)$ is included in $\mathcal{D}^{\text{trv}}$ if, and only if, $v_1$ and $v_2$ are bound by opposite quantifiers and $v_1$ is bound before $v_2$ in the quantifier prefix of $\Phi$. For some dependency scheme $\mathcal{D}$, we write $\mathcal{D}(v) = \{v' \in \mathcal{V} \mid (v', v) \in \mathcal{D}\}$ for some variable $v \in \mathcal{V}$ to denote the set of dependencies of $v$.

We parameterize the ($\forall$**exp-res**) rule by a dependency scheme $\mathcal{D}$ and adapt the expansion formula (Section 2.5): For an existential variable $x$, a root-to-leaf path $P$, and a dependency scheme $\mathcal{D}$ we define $\mathit{expand\text{-}var}_{\mathcal{D}}(P, x) = x^{\alpha}$ where $x^{\alpha}$ is a fresh variable and $\alpha = \left(\bigsqcup_{1 \le i \le u} \alpha_i\right)|_{\mathcal{D}(x)}$ is the universal assignment of the dependencies of $x$. In this way, we derive the definition of the expansion formula for paths $\mathit{expand}_{\mathcal{D}}(P, \varphi)$ and trees $\mathit{expand}_{\mathcal{D}}(\mathcal{T}, \varphi)$. We denote the resulting rule by ($\forall$**exp**$(\mathcal{D})$**-res**).

$$\frac{\mathcal{T} \quad \mathcal{C}^{\geq k} \quad \pi}{\mathcal{P}^{k-1}} \; \forall\textbf{exp}(\mathcal{D})\textbf{-res}$$

$Q_k = \exists$, $\pi$ is a resolution refutation of the expansion formula $\mathit{expand}_{\mathcal{D}}(\mathcal{T}, \exists X_k.\, \forall X_{k+1} \ldots \exists X_m.\, \mathcal{C}^{\geq k})$
$\mathcal{P}^{k-1} = \{i \mid C_i \in \mathcal{C}\}^{k-1}$

Note that in the original definition of the rule ($\forall$**exp-res**) we have implicitly used the trivial dependency scheme $\mathcal{D}^{\text{trv}}$, thus, ($\forall$**exp-res**) and ($\forall$**exp**$(\mathcal{D}^{\text{trv}})$**-res**) are equivalent. The soundness of ($\forall$**exp**$(\mathcal{D})$**-res**) depends on the dependency scheme $\mathcal{D}$. For example, the reflexive resolution path dependency scheme $\mathcal{D}^{\text{rrs}}$ [SS16], which is the most general dependency scheme that is known to be sound for $Q$-resolution, is sound for $\forall$Exp+Res [Bey+18] as well and, thus, the rule ($\forall$**exp**$(\mathcal{D}^{\text{rrs}})$**-res**) is sound.

**Corollary 3.19.** *The* ($\forall$**exp**$(\mathcal{D})$**-res**) *rule is sound for every dependency scheme* $\mathcal{D}$ *such that* $\forall$*Exp*$(\mathcal{D})$*-Res is sound.*

### 3.3.4 Comparison Between Extensions

We conclude this section by comparing the two extensions of the $\forall$Red+Res calculus.

**Theorem 3.20.** $\forall$*Red+*$\forall$*Exp+Res and* $\forall$*Red+Res+SU are incomparable.*

*Proof Sketch.* The family of formulas $\mathrm{CR}'_n$ separates $\forall$Red+$\forall$Exp+Res and $\forall$Red+Res+SU. Since the rule (**SU**) is not applicable, all $\forall$Red+Res proofs are exponential while there is a polynomial proof in $\forall$Red+$\forall$Exp+Res.

For the other direction we use a similar construction as the one used in the proof of Theorem 3.15. We use a combination of $\mathrm{CR}_n$ and $\mathrm{DAG}_n$ to construct a formula that has

$$\frac{(\mathcal{P}_1 \cup \mathcal{P}_1')^k \quad \cdots \quad (\mathcal{P}_j \cup \mathcal{P}_j')^k \quad \pi}{\left(\bigcup_{i\in\{1,\ldots,j\}} \mathcal{P}_i'\right)^{k-1}}\ \textbf{res} \qquad Q_k = \forall,\ \pi \text{ is a cube resolution proof for } \bigvee_{1\le i\le j} \bigwedge_{l\in lit(\mathcal{P}_i^k)} l$$

$$\frac{\mathcal{P}^k}{\mathcal{P}'^{k-1}}\ \exists\textbf{red} \qquad Q_k = \exists,\ \mathcal{P}' \subseteq \mathcal{P},\ \forall l \in lit((\mathcal{P}\setminus\mathcal{P}')^k).\, \bar{l} \notin lit((\mathcal{P}\setminus\mathcal{P}')^k)$$

$$\frac{}{\{1,\ldots,m\}^k}\ \textbf{init} \qquad \forall i \in \{1,\ldots,m\}.\, C_i^{>k} = \varnothing$$

Figure 3.7: The rules of the ∃Red+Res calculus.

only exponential refutations in ∀Red+∀Exp+Res, but a polynomial refutation using the strong-unsat rule. The formula $\mathrm{DAG}_n$ is used to generate the premise for the application of the strong-unsat rule to solve $\mathrm{CR}_n$. To generate this premise using the rule (∀**exp-res**) one needs an exponential proof. There is a polynomial proof for $\mathrm{DAG}_n$ in ∀Red+Res, but there is none for $\mathrm{CR}_n$, thus, ∀Red+∀Exp+Res has only exponential refutations. □

**Theorem 3.21.** *∀Red+∀Exp+Res+SU and Q-resolution are incomparable.*

*Proof.* Follows from the proof of Theorem 3.16 as the witnessing formula can be constructed such that the strong-unsat rule is not applicable. The other direction follows from the separation of $Q$-resolution and ∀Exp+Res by Beyersdorff et al. [BCJ15]. □

## 3.4 A Proof Calculus for Satisfiable Formulas

So far, we were only interested in *refutation proofs*, that are, proofs that a formula is false. We argued that this is enough to have a complete proof system as for true formulas we show the refutation of the negation. This, however, does not match the behavior of the clausal abstraction algorithm, which learns from counterexamples to universal choices in the same way as for existential choices. In this section, we present the proof system underlying the clausal abstraction algorithm on true formulas.

**Cube resolution.** Dual to the propositional resolution rule, the resolution rule can be applied to propositional formulas in disjunctive normal form to prove validity. For example, using the resolution rule twice, we derive that $u \vee v \vee (\overline{u} \wedge \overline{v})$ is valid.

**∃Red+Res.** The rules of the proof system for valid formulas, which we call ∃Red+Res, are presented in Figure 3.7. The axiom rule lets us derive the leaf of the proof, that is, the proof object containing the indices of every clauses. The *resolution* rule is applicable for universal quantifier and allows the merger of proof objects if the underlying cube resolution proof is valid. Lastly, existential reduction can be applied to existential quantifier

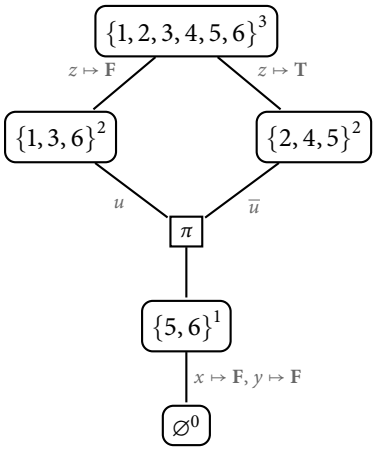



Figure 3.8: A ∃Red+Res proof for the formula given in Equation 3.2. The edges are annotated by the assignments in case of existential reduction and cubes in case of resolution.

and it removes the clauses satisfied by an assignment to the existential variables. The root of a ∃Red+Res proof is the proof object $\varnothing^0$.

**Example 3.22.** Consider the following true QBF

$$\underbrace{\exists x, y}_{1}.\ \underbrace{\forall u}_{2}.\ \underbrace{\exists z}_{3}.\ (\underbrace{u \vee \overline{z}}_{C_1})(\underbrace{\overline{u} \vee z}_{C_2})(\underbrace{x \vee u \vee \overline{z}}_{C_3})(\underbrace{y \vee \overline{u} \vee z}_{C_4})(\underbrace{\overline{x} \vee \overline{u} \vee \overline{z}}_{C_5})(\underbrace{\overline{y} \vee u \vee z}_{C_6}). \tag{3.2}$$

A proof in the ∃Red+Res calculus is given in Figure 3.8.

**Comparison to $Q$-Resolution calculus.** Analogously to ∀Red+Res, ∃Red+Res is polynomially equivalent to the dual of the level-ordered $Q$-resolution calculus, that is, the $Q$-resolution calculus that operates on cubes instead of clauses [LES16].

**Proposition 3.23.** *∃Red+Res and the dual of the level-ordered $Q$-resolution calculus are p-simulation equivalent.*

*Proof.* The proof is similar to the proof of Theorem 3.5. A ∃Red+Res proof can be transformed into a cube-based $Q$-resolution proof by a stepwise transformation from the root to the leaves: The root proof object $\varnothing^0$ corresponds to the empty cube. At an existential reduction, we add the corresponding existential assignment to the cube. The hyper-edge $\pi$ for a resolution proof in ∃Red+Res corresponds to a cube resolution proof in $Q$-resolution.

A level-ordered $Q$-resolution proof over cubes can be transformed into a ∃Red+Res proof by a transformation from leaves to the root by tracking the satisfied clauses. □

As the $Q$-resolution calculus over cubes is sound and complete for QBF [GNT06], the same follows for ∃Red+Res.

**Corollary 3.24.** *Given a QBF $\varphi$ in prenex conjunctive normal form. There is a $\exists$Red+Res proof for $\varphi$ if, and only if, $\varphi$ is true.*

We also know that the $Q$-resolution calculus over clauses is exponentially more succinct than the $Q$-resolution calculus over cubes [JM17]. In [JM17] the author proposed to solve true QBF by refuting their negation. In Chapter 4 we show another possible remedy by considering a generalized normal form, that is, negation normal form (NNF), and introducing an extension of the clausal abstraction algorithm that is applicable to NNF.

## 3.5 Summary

In this chapter, we have presented the QBF proof calculus $\forall$Red+Res that corresponds to the clausal abstraction algorithm. We defined two extensions of the $\forall$Red+Res calculus, the rule ($\mathbf{SU}$) that breaks the level-orderedness constraint inherent to $\forall$Red+Res and the axiom rule ($\forall\mathbf{exp\text{-}res}$) that allows the integration of partial expansion proofs. We showed that both extensions are orthogonal, and both make the resulting proof systems exponentially more succinct. Understanding the proof theory underlying the clausal abstraction approach helped us to understand theoretical strengths and weaknesses and explained empirical differences and similarities compared to existing solvers.

# Chapter 4

# Circuit Abstraction

The clausal abstraction algorithm presented in the previous sections was limited to formulas in prenex conjunctive normal form. Beyond conjunctive normal form (CNF), there have been many attempts to improve solving performance by going to more general formula representations, such as circuits [ESW09; GIB09; Kli+10; GB10]. These approaches close the gap in expressive power between universal and existential players in CNF [JM17] and often outperform CNF-based solvers on practical benchmarks. In this chapter, we present an extension of the clausal abstraction algorithm to QBFs in negation normal form (NNF). The algorithmic underpinnings are remarkably similar: The algorithm builds an abstraction for each quantifier block, communicates assignments between abstractions, and uses single-clause refinements. The construction of the abstraction, however, is quite involved, due to the more general propositional structure. The implementation of our NNF approach in the solver QuAbS is used in the reactive synthesis tool BoSy [FFT17], the Petri game solver Adam [Fin+17a], and the HyperLTL satisfiability solver MGHyper [FHH18]. Also, QuAbS won the prenex non-CNF track of QBFEVAL 2018 as well as 2019 and was awarded a medal in the FLoC Olympic Games in 2018[1].

This chapter is based on work published in the proceedings of SAT [Ten16] and GandALF [HT18], as well as an article accepted for publication in the journal of satisfiability (JSAT) [Ten19]. The circuit abstraction algorithm, including correctness proof, optimizations, and certification, is presented in Section 4.1. In Section 4.2, we provide an experimental evaluation of the solver QuAbS. Section 4.3 discusses the extension to formulas in non-prenex form.

## 4.1 Circuit Abstraction

A fundamental property of the PCNF game is that it is not dual for the two players: The existential player has to satisfy *all* clauses while the universal player tries to falsify some clause. This is especially visible in the underlying proof system: The refutation proof system is exponentially more succinct than the satisfaction proof system [JM17]. We propose

[1] http://www.floc2018.org/floc-olympic-games/

**Algorithm 4.1** Abstraction Algorithm for QBF in negation normal form.

1: **procedure** SOLVE($\Phi = \mathcal{Q}X.\,\Psi$)
2: initialize abstraction $\theta_Y$ and dual abstraction $\overline{\theta}_Y$ for every quantifier $\mathcal{Q}Y$ in $\Phi$
3: **return** SOLVE-NNF($\mathcal{Q}X, \Psi, \{s_i \mapsto \mathbf{F} \mid s_i \in S_X\}$)
4: **end procedure**

a generalization of the clausal abstraction algorithm to propositional formulas in negation normal form (NNF), making the game effectively dual.

Throughout this chapter, we use the notation established for quantified Boolean formulas in ➔ Section 2.1. For quantified Boolean formulas given in conjunctive normal form, we have presented a recursive refinement algorithm in Chapter 2, where the refinement is based on clauses. The underlying insight is that multiple variable assignments may lead to the satisfaction of the same clauses. Hence, instead of communicating assignments, the information on whether a clause is satisfied or not is communicated between quantifier blocks. Instead of excluding assignments one at a time, the clausal abstraction algorithm may exclude multiple assignments with a single refinement step. In the following, we propose a generalization to formulas in negation normal form, i.e., we base the communication on the satisfaction of individual subformulas. For this chapter, we assume an arbitrary (closed, prenex) QBF $\Phi = \mathcal{Q}X_1 \cdots \mathcal{Q}X_n.\,\varphi$ with quantifier prefix $\mathcal{Q}X_1 \cdots \mathcal{Q}X_n$ and propositional body $\varphi$ in NNF.

➔ Page 16

### 4.1.1 Algorithm

**Overview.** The algorithm for solving QBF in negation normal form is in large parts a staightforward extension of the existential CNF algorithm shown in ➔ Section 2.3. The algorithm SOLVE, depicted in Algorithm 4.1, initializes the abstractions and returns the result of SOLVE-NNF, shown in Algorithm 4.2. SOLVE-NNF determines candidate assignments to the variables bound at that quantifier, which is then verified recursively, or gives a reason why there is no such assignment. In the negative case, this reason is excluded at an outer quantifier.

➔ Page 25

Going from CNF to NNF makes the algorithm *more uniform* and—at the same time—*more complex*, where the uniformity comes from the quantifiers' duality and the complexity arises from the less restrictive normal form. Taking both into account leads us to the most significant algorithmic contribution, the use of a *dual* abstraction $\overline{\theta}_X$ in conjunction with the abstraction $\theta_X$ seen in previous algorithms. The dual abstraction, whose name indicates that it is the abstraction for negation of the current quantifier, elegantly solves two issues that already arose in the previous algorithms but were much easier to handle for CNF. First, consider again the optimization discussed in Section 2.3.3 that improves the returned witness in the propositional case. In CNF, this was done by setting satisfaction variables to false whenever the current assignment (of existential variables) satisfies a clause. In NNF, we use the dual abstraction to generate those partial assignments from complete assignments of the satisfaction variables using a technique inspired by dual propagation [GB10; GSB13; NPB14]. Second, in the CNF algorithm ➔ Algorithm 2.3,

➔ Page 28

we needed to project the partial assignment returned from the inner quantifier in case of a successful verification (line 8) as some of the clauses may be satisfied by the current assignment (of existential variables). In NNF, this is not merely a projection, but a transformation from one set of satisfaction variables to a (possibly) different set of satisfaction variables which can be efficiently implemented by the dual abstraction. Before going into details of algorithm SOLVE-NNF, we introduce the abstractions first.

**Example 4.1.** Consider again the QBF from Example 2.2 where the propositional formula $\varphi$ is in negation normal form:

$$\exists x.\, \forall v, w.\, \exists y.\, \underbrace{(x \vee v \vee \overbrace{(y \wedge w)}^{\psi_3})}_{\psi_2} \wedge \underbrace{(\overline{x} \vee \overbrace{(v \wedge \overline{w})}^{\psi_5} \vee y)}_{\psi_4} \wedge \underbrace{(\overline{v} \vee w \vee \overline{y})}_{\psi_6} \tag{4.1}$$

Throughout this section, we use the naming of the subformulas as indicated in above and name $\psi_1 = \varphi$. Note, that the formula is true as witnessed by the Skolem functions $x = \mathbf{T}$ and $y(v, w) = \overline{v} \vee w$.

**Abstraction $\theta$.** The abstraction $\theta_X$ is a propositional formula that represents, for every quantifier block $\mathcal{Q}X$, an over-approximation of the winning assignments $\alpha_X$ as well as the effect of the assignment $\alpha_X$ on the valuation of subformulas. The algorithm guarantees that whenever a candidate assignment $\alpha_X$ is generated using $\theta_X$, all variables bound at outer quantifiers have a fixed assignment, and, thus, the propositional formula $\varphi$ is partially evaluated.

To facilitate working with arbitrary Boolean formulas, we start with introducing additional notation. Let $\mathcal{B}$ be the set of Boolean formulas and let $sf(\psi) \subset \mathcal{B}$ and $dsf(\psi) \subset \mathcal{B}$ be the set of all subformulas of $\psi$ and the set of direct (or immediate) subformulas of $\psi$, respectively. Note that $\psi \in sf(\psi)$ but $\psi \notin dsf(\psi)$. For a propositional formula $\psi$, $type(\psi) \in \{lit, \vee, \wedge\}$ returns the Boolean connector if $\psi$ is not a literal. For example, given $\psi = (x \vee v \vee (y \wedge w))$, the set of all subformulas is $sf(\psi) = \{(x \vee v \vee (y \wedge w)), x, v, (y \wedge w), y, w\}$, the set of direct subformulas is $dsf(\psi) = \{x, v, (y \wedge w)\}$, and the Boolean connector is $type(\psi) = \vee$. For every subformula $\psi$, we denote by $\overline{\psi}$ the dual subformula, that is, the formula where every quantifier, Boolean connector, and literal is negated. It holds that $\overline{\psi}$ is in NNF and that $\neg\overline{\psi}$ is equivalent to $\psi$.

We will explain the abstraction for quantifier $\exists X$ as a transformation of the graph representation of propositional formulas. A propositional formula $\psi$ can be represented as a graph, where the nodes represent the Boolean connectives and the edges connect a formula with its direct subformulas. The leaves, i.e., terminal nodes, are the literals contained in $\psi$. Formally, the graph $\mathcal{G}_\varphi$ corresponding to some propositional formula $\varphi$ is a pair $\langle V, E \rangle$, where $V = sf(\varphi)$ is the set of vertices and $E = V \times V$ is the edge relation such that $(\psi_i, \psi_j) \in E$ if, and only if, $\psi_j \in dsf(\psi_i)$. Figure 4.1 depicts the graph corresponding to the propositional part of the QBF presented in Example 4.1.

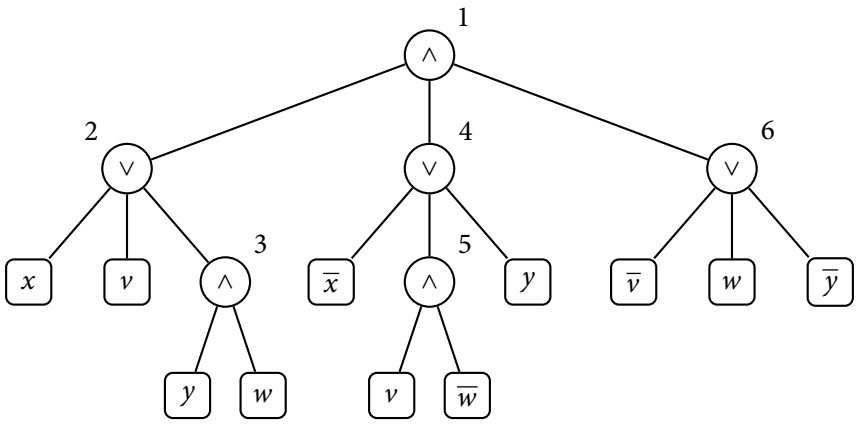



Figure 4.1: Visualization of the graph representation $\mathcal{G}_\varphi$ representation of $\varphi = (x \vee v \vee (y \wedge w)) \wedge (\overline{x} \vee (v \wedge \overline{w}) \vee y) \wedge (\overline{v} \vee w \vee \overline{y})$. The numbers on the non-terminal nodes represent the index $i$ for the corresponding subformula $\psi_i$ as shown in Example 4.1. To improve readability, some terminal nodes like $y$ and $w$ are drawn multiple times.

We define $\psi^\circ$ for $\circ \in \{<, \leq, =, \geq, >\}$ as the projection of $\psi$ onto variables bound by outer ($<$), current ($=$), or inner ($>$) quantifiers with respect to $\mathcal{Q}X$, respectively. If the projected formula does not contain a literal, we return undefined $\perp$. Formally, we define $\psi^\circ$ recursively (where we only recurse if the projection of a subformula is defined) as follows

$$\psi^\circ \coloneqq \begin{cases} \bigwedge\limits_{\substack{\psi_i \in dsf(\psi) \\ \psi_i^\circ \neq \perp}} \psi_i^\circ & \text{if } type(\psi) = \wedge \\ \bigvee\limits_{\substack{\psi_i \in dsf(\psi) \\ \psi_i^\circ \neq \perp}} \psi_i^\circ & \text{if } type(\psi) = \vee \\ \psi & \text{if } \psi \text{ is a literal } l \text{ and } var(l) \text{ is bound at a quantifier level satisfying } \circ \\ \perp & \text{otherwise} \end{cases}$$

Applying this definition on our running example in Equation 4.1, we get, for example, $\psi_2^{=} = x$, $\psi_4^{=} = \overline{x}$ for quantifier $\exists x$; $\psi_2^{\leq} = x \vee v \vee w$, $\psi_4^{\leq} = \overline{x} \vee (v \wedge \overline{w})$ for quantifier $\forall v, w$; and $\psi_1^{\leq} = \varphi$ for quantifier $\exists y$.

We use the same kind of variables as in clausal abstraction to establish the interaction between abstractions: The variables $X$ bound by the current quantifier and, additionally, the assumption and satisfaction variables $A$ and $S$, respectively. The satisfaction variable $s_i$ for some subformula $\psi_i \in sf(\varphi)$ represents the effect of variables $V$ bound at outer quantifier on $\psi_i$. To quantify this effect, we have to distinguish whether $\psi_i$ is a disjunctive ($type(\psi_i) = \vee$) or conjunctive ($type(\psi_i) = \wedge$) formula. In the disjunctive case, assigning $s_i$ to true implies that $\psi_i$ evaluates to true given the outer variable assignment $\alpha_V$. This is a straightforward generalization of the existential abstraction for clauses (see ➔ Page 25 ➔ Section 2.3). In case $\psi_i$ is conjunctive, a positive assignment of $s_i$ means that the conjunct is not yet falsified, that is, $\psi_i$ does not evaluate to false given the outer

variable assignment $\alpha_V$. We combine both cases by saying that $\psi_i$ is assigned *positively* with respect to the current quantifier. Since the valuation of the variables $X$ bound by the current quantifier has an influence on the valuation of subformulas as well, we use an assumption variable $a_i$ to represent the effect of the combined assignments $\alpha_X$ and $\alpha_S$. The intended semantics is that $a_i$ is set to false only if $\psi_i$ is assigned positively at this quantifier (by assignment $\alpha_X \dot{\cup} \alpha_V$).

Before formally defining the abstraction, we discuss the underlying derivation steps on Example 4.1.

**Example 4.2.** The abstraction $\theta_X$ quantifies the effect of valuations of variables $X$ on the satisfaction of subformulas. We derive the abstraction by transforming the graph representation of $\varphi$ and $\overline{\varphi}$ for existential and universal quantifiers, respectively. This transformation is visualized in Figure 4.2. As a first step, we remove all subformulas $\psi$ which are only influenced by inner quantifiers, i.e., every $\psi$ that is not contained in $\varphi^{\leq}$. For example, $\psi_6$ does not contain $x$, thus, the whole subformula is removed from $\varphi$ for quantifier $\exists x$.

Then, we replace all maximal subformulas with the property $\psi^{<} = \psi$ by satisfaction variables $s_i$ in a top-down way. Consider the innermost quantifier $\exists y$ and subformula $\psi_4$ with $dsf(\psi_4) = \{\overline{x}, \psi_5, y\}$. For the former two, $x$ and $\psi_5$, it holds that $x^{<} = x$ and $\psi_5^{\leq} = \psi_5$, thus, both are replaced with the satisfaction variable $s_4$.

In the innermost quantifier $\exists y$, this already adequately describes the abstraction, for every other quantifier we have to define the assumption variables. For example at quantifier $\exists x$, an assignment to $x$ can either satisfy $\psi_2$ or $\psi_4$, but not both, thus, the other formula is assumed to be satisfied by an inner quantifier. We define an assumption variable for every subformula $\psi_i \in sf(\varphi)$ such that there exist direct subformulas $\psi_j$ and $\psi_k$ such that $\psi_j = \psi_j^{\leq}$ and $\psi_k \neq \psi_k^{\leq}$. Intuitively, for these subformulas $\psi_i$, there is a direct influence by $\psi_j = \psi_j^{\leq}$ and the value of $\psi_i$ is not guaranteed to be determined after the current quantifier as there is some influence by inner variables $\psi_k \neq \psi_k^{\leq}$. This can be seen at our example at quantifier $\forall v, w$: We need to add an assumption variable to $\psi_3$ as $\overline{y} \in dsf(\psi_3)$ but not to $\psi_5$ as $\psi_5^{=} = \psi_5$. Lastly, given some quantifier alternation $\mathcal{Q}X.\, \overline{\mathcal{Q}}Y$, there is a one-to-one correspondence between the assumption variables $A_X$ of quantifier $\mathcal{Q}X$ and the satisfaction variables $S_Y$ of quantifier $Y$.

Using the intuition of the interface variables and the determinacy of subformulas, we are now going to define the abstraction formally. In this definition, we take advantage of the duality by only defining the abstraction for existential quantifiers. The abstraction for universal quantifiers is then the abstraction for the negated formula $\overline{\Phi}$. Let us fix some existential quantifier $\exists X$ and some subformula $\psi_i$ of $\varphi$. The abstraction $\theta_X$ is defined as

$$\theta_X = \begin{cases} enc(\varphi) & \exists X \text{ is the innermost quantifier} \\ \bigwedge_{\substack{\psi_i \in sf(\varphi) \\ \exists \psi_j, \psi_k \in dsf(\psi_i) \text{ with } \psi_j = \psi_j^{\leq} \wedge \psi_k \neq \psi_k^{\leq}}} a_i \vee enc(\psi_i) & \text{otherwise} \end{cases} \tag{4.2}$$

For the innermost quantifier $\exists X.\, \varphi$, we encode $\varphi$ using $enc$ defined below. In all other cases, we define the implication that setting an assumption variable $a_i$ to false is only

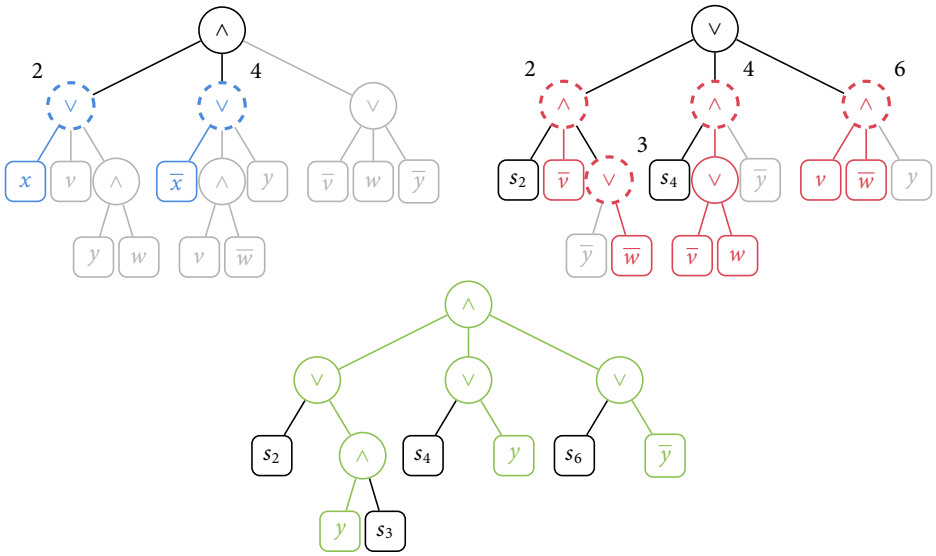



Figure 4.2: Abstraction for quantifiers $\exists x$, $\forall v, w$, and $\exists y$. The grayed out subformulas are only influenced by inner variables. The colored parts indicate continuous subformulas with $\psi = \psi^{\leq}$. The dashed subformulas indicate placement of assumption variables.

possible if the formula $enc(\psi_i)$ is satisfied. $enc(\psi_i)$ considers only subformulas of $\psi_i$ which do not contain inner variables and where outer variables are replaced by their respective satisfaction literals. Formally, the abstraction for $\psi_i$ is defined as

$$enc(\psi_i) = \begin{cases} \bigwedge\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j = \psi_j^{\leq}}} enc_{\psi_i}(\psi_j) & \text{if } type(\psi_i) = \wedge \\ \bigvee\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j = \psi_j^{\leq}}} enc_{\psi_i}(\psi_j) & \text{if } type(\psi_i) = \vee \end{cases} \tag{4.3}$$

where the direct subformulas $\psi_j$ of $\psi_i$ are transformed as follows

$$enc_{\psi_i}(\psi_j) = \begin{cases} \psi_j & \text{if } \psi_j = \psi_j^{=} \\ s_i & \text{if } \psi_j = \psi_j^{<} \\ enc(\psi_j) & \text{otherwise, i.e., } \psi_j = \psi_j^{\leq} \end{cases} \tag{4.4}$$

We carefully dissect the definitions in order to map them to the intuitions mentioned above. The function $enc(\psi_i)$ builds the abstraction for subformula $\psi_i$ depending on the Boolean connector $type(\psi_i) \in \{\wedge, \vee\}$. Further, $enc$ considers only those direct subformulas $\psi_j$ of $\psi_i$, which are solely influenced by the current or outer variables, i.e., $\psi_j = \psi_j^{\leq}$. The encoding of direct subformulas $enc_{\psi_i}(\psi_j)$ distinguishes three cases. If $\psi_j$ contains only variables $X$, that is, $\psi_j = \psi_j^{=}$, then the result of $enc_{\psi_i}(\psi_j) = \psi_j$ is the formula $\psi_j$ itself. If $\psi_j$ contains only outer variables, that is, $\psi_j = \psi_j^{<}$, then the result of $enc_{\psi_i}(\psi_j) = s_i$ is the satisfaction variable $s_i$. Finally, if $\psi_j$ contains both types of variables, we apply $enc$ on $\psi_j$.

**Algorithm 4.2** Algorithm for solving quantified formulas in NNF.

1: **procedure** SOLVE-NNF($\mathcal{Q}X, \Phi, \alpha_{S_X}$)
2: **loop**
3: **match** $\langle \text{SAT}(\theta_X, \alpha_{S_X}),\ \Phi \rangle$ **as** ▷ assume outer variable assignment
4: $\langle \text{Sat}(\alpha),\ \overline{\mathcal{Q}}Y.\Psi \rangle \Rightarrow$
5: $\alpha_{S_Y} \leftarrow \{s_i \mapsto \alpha(a_i) \mid a_i \in A_X\}$ ▷ update subformula valuation
6: **match** SOLVE-NNF($\overline{\mathcal{Q}}Y, \Psi, \alpha_{S_Y}$) **as** ▷ recursive verification
7: $\text{Sat}_{\mathcal{Q}}(\beta_{S_Y}) \Rightarrow \overline{\theta}_X \leftarrow \overline{\theta}_X \wedge \bigvee_{s_i \in \beta^0_{S_Y}} \overline{a}_i$ ▷ refine $\overline{\theta}_X$
8: **return** $\text{Sat}_{\mathcal{Q}}(\text{OPTIMIZE}(\alpha|_X, \alpha_{S_X}))$
9: $\text{Unsat}_{\mathcal{Q}}(\beta_{S_Y}) \Rightarrow \theta_X \leftarrow \theta_X \wedge \bigvee_{s_i \in \beta^1_{S_Y}} \overline{a}_i$ ▷ refine $\theta_X$
10: $\langle \text{Sat}(\alpha),\ \_ \rangle \Rightarrow$ **return** $\text{Sat}_{\mathcal{Q}}(\text{OPTIMIZE}(\alpha|_X, \alpha_{S_X}))$ ▷ propositional
11: $\langle \text{Unsat}(\beta_{S_X}),\ \_ \rangle \Rightarrow$ **return** $\text{Unsat}_{\mathcal{Q}}(\beta_{S_X})$
12: **end loop**
13: **end procedure**
14: **procedure** OPTIMIZE($\alpha_X, \alpha_{S_X}$)
15: **match** $\text{SAT}(\overline{\theta}_X, \alpha_X \sqcup \overline{\alpha}_{S_X})$ **as**
16: $\text{Unsat}(\beta) \Rightarrow$ **return** $\overline{\beta}|_{S_X}$ ▷ $\overline{\beta}|_{S_X} \sqsubseteq \alpha_{S_X}$
17: **end procedure**

The abstraction for a universal quantifier $\forall X$ and the dual abstraction $\overline{\theta}_X$ of quantifier $\exists X$ are both defined as the abstraction for $\exists X$ with respect to propositional formula $\overline{\varphi}$. As discussed above, satisfaction and assumption variables are not exposed for every subformula $\psi_i \in sf(\varphi)$. For the given abstraction, we define the set of interface variables for quantifier $\mathcal{Q}X$ as

$$A_X = \{a_i \mid \psi_i \in sf(\varphi) \wedge \exists \psi_j, \psi_k \in dsf(\psi_i).\, \psi_j = \psi_j^{\leq} \wedge \psi_k \neq \psi_k^{\leq}\} \text{ and}$$
$$S_X = \{s_i \mid \psi_i \in sf(\varphi) \wedge \exists \psi_j, \psi_k \in dsf(\psi_i).\, \psi_j = \psi_j^{<} \wedge \psi_k \neq \psi_k^{<}\}\ .$$

This means that for some quantifier alternation $\mathcal{Q}X.\,\overline{\mathcal{Q}}Y$ the sets $A_X$ and $S_Y$ represent the same subformulas, i.e., $a_i \in A_X$ if, and only if, $s_i \in S_Y$.

The algorithm makes progress by refining the abstraction during the execution of the algorithm. Such a refinement excludes wrong assumptions, i.e., assumptions corresponding to a losing assignment for the variables of the respective quantifier block. Given such a set of assumptions $L \subseteq A$, the refinement is represented by the clause

$$\bigvee_{a_i \in L} \overline{a}_i\ . \tag{4.5}$$

**Algorithm.** Algorithm 4.2 shows the recursive QBF solving algorithm SOLVE-NNF. It decides the problem whether the quantified subformula $\mathcal{Q}X.\,\Phi$ of $\Phi$ for $\mathcal{Q} \in \{\forall, \exists\}$ is satisfiable under the condition that the propositional formula $\varphi$ is partially evaluated according to the assignment $\alpha_S$ that abstracts the outer variable assignment. Note that due

to duality, the satisfiability and unsatisfiability are interpreted with respect to the current quantifier, that is, we define

$$\mathsf{Sat}_{\mathcal{Q}} = \begin{cases} \mathsf{Sat} & \text{if } \mathcal{Q} = \exists \\ \mathsf{Unsat} & \text{if } \mathcal{Q} = \forall \end{cases} \quad \text{and} \quad \mathsf{Unsat}_{\mathcal{Q}} = \begin{cases} \mathsf{Unsat} & \text{if } \mathcal{Q} = \exists \\ \mathsf{Sat} & \text{if } \mathcal{Q} = \forall \end{cases}.$$

For sake of simplicity, we base our explanation on existential quantifier in the following. The algorithm repeatedly generates candidate assignments by means of the abstraction $\theta_X$ (line 3). If the abstraction returns Unsat, there is no satisfiable assignment with respect to the assignment $\alpha_S$ of satisfaction variables, thus, the algorithm returns $\mathsf{Unsat}_{\mathcal{Q}}$ as well (line 11). Further, the reason for the unsatisfiability result is given, represented by the returned partial assignment $\beta_{S_X}$. If the abstraction returns Sat with assignments $\alpha_A$ and $\alpha_X$, we distinguish two cases. The first case is the base case of the recursion, that is, the inner formula is quantifier-free (line 10). The algorithm returns $\mathsf{Sat}_{\mathcal{Q}}$ and the partial assignment, generated by the algorithm OPTIMIZE, indicating which subformulas have to be positively assigned by outer quantifier such that the assignment $\alpha_X$ satisfies $\varphi$. Lastly, assume that the inner subformula is quantified. In this case, we compute the subformulas of $\varphi$ that the combination of $\alpha_X$ and $\alpha_S$ assign positively (line 5) and continue with the recursive verification. In the positive case, the partial assignment $\beta_{S_Y}$ (line 7) indicates the required positively assigned subformulas. As this witnesses the unsatisfiability of the negated formula, the dual abstraction $\overline{\theta}_X$ is refined with $\beta_{S_Y}$ before translating the assignment $\beta_{S_Y}$ to an assignment $\beta_{S_X}$ using OPTIMIZE in line 8. In case it is negative, the abstraction $\theta_X$ is refined by enforcing that some negatively assigned subformulas is assigned postively, before continuing with the next iteration.

The algorithm OPTIMIZE implements dual propagation. The dual abstraction $\overline{\theta}_X$ is a representation of the possible assignments of the negated quantifier $\overline{\mathcal{Q}}X$. Thus, assuming the positively verified assignments $\alpha_X$ and $\alpha_S$ (lines 8 and 10) lead to unsatisfiability of $\overline{\theta}_X$. Note, that we have to negate $\alpha_S$ due to the way the abstraction is built (a formal justification is given in Section 4.1.2). The return value $\beta$, that is, the failed assumptions projected onto variables $S_X$, represents a set of subformulas $\{\psi_i \mid s_i \in \beta^1_{S_X}\}$ that needs to be assigned positively such that the quantifier $\mathcal{Q}X$ has a satisfiable assignment.

→ Page 17

**Example 4.3.** Consider again the formula given in → Example 2.2:

$$\exists x.\, \forall v, w.\, \exists y.\, \underbrace{(x \vee v \vee \overbrace{(y \wedge w)}^{\psi_3})}_{\psi_2} \wedge \underbrace{(\overline{x} \vee \overbrace{(v \wedge \overline{w})}^{\psi_5} \vee y)}_{\psi_4} \wedge \underbrace{(\overline{v} \vee w \vee \overline{y})}_{\psi_6}$$

We give the abstractions as discussed in Example 4.2 in propositional form as

$$\begin{aligned} \theta_{\{x\}} &= (a_2 \vee x)(a_4 \vee \overline{x}), \\ \theta_{\{v,w\}} &= (a_2 \vee (s_2 \wedge \overline{v}))(a_3 \vee \overline{w})(a_4 \vee (s_4 \wedge (\overline{v} \vee w)))(a_6 \vee (v \wedge \overline{w})), \\ \overline{\theta}_{\{v,w\}} &= (a_2 \vee s_2 \vee v)(a_3 \vee w)(a_4 \vee s_4 \vee (v \wedge \overline{w}))(a_6 \vee \overline{v} \vee w), \\ \theta_{\{y\}} &= (s_2 \vee (y \wedge s_3))(s_4 \vee y)(s_6 \vee \overline{y}), \text{ and} \\ \overline{\theta}_{\{y\}} &= (s_2 \wedge (\overline{y} \vee s_3)) \vee (s_4 \wedge \overline{y}) \vee (s_6 \wedge y). \end{aligned}$$

We give a possible execution of algorithm SOLVE. To improve readability, we use the propositional representation for assignments.

- SOLVE-NNF($\exists x, \forall v, w. \exists y. \varphi, \{\}$)
- SAT($\theta_{\{x\}}$) = Sat($\overline{x}\, a_2 \overline{a_4}$)
- $\alpha_{S_{\{v,w\}}} = s_2 \overline{s_4}$
  - SOLVE-NNF($\forall v, w, \exists y. \varphi, \alpha_{S_{\{v,w\}}}$)
  - SAT($\theta_{\{v,w\}}, \alpha_{S_{\{v,w\}}}$) = Sat($\overline{v}\,\overline{w}\,\overline{a_2}\overline{a_3}a_4a_6$)
  - $\alpha_{S_{\{y\}}} = \overline{s_2}\overline{s_3}s_4s_6$
    - SOLVE-NNF($\exists y, \varphi, \alpha_{S_{\{y\}}}$)
    - SOLVE($\theta_{\{y\}}, \alpha_{S_{\{y\}}}$) = Unsat($\overline{s_2}\overline{s_3}$)
    - **return** Unsat($\overline{s_2}\overline{s_3}$)
  - $\overline{\theta}'_{\{v,w\}} = \overline{\theta}_{\{v,w\}} \wedge (\overline{a_2} \vee \overline{a_3}) = (s_2 \vee v \vee w)$ [...]
  - SOLVE($\overline{\theta}'_{\{vw\}}, \overline{v}\,\overline{w}\,\overline{\alpha}_{S_{\{v,w\}}}$) = Unsat($\overline{v}\,\overline{w}\,\overline{s_2}$)
  - **return** Unsat($s_2$)
- $\theta'_{\{x\}} = \theta_{\{x\}} \wedge \overline{a_2}$
- SAT($\theta'_{\{x\}}$) = Sat($x\,\overline{a_2}a_4$)
- $\alpha'_{S_{\{v,w\}}} = \overline{s_2}s_4$
  - SOLVE-NNF($\forall v, w, \exists y. \varphi, \alpha'_{S_{\{v,w\}}}$)
  - SAT($\theta_{\{v,w\}}, \alpha'_{S_{\{v,w\}}}$) = Sat($\overline{v}\,\overline{w}\,a_2\overline{a_3}\overline{a_4}a_6$)
  - $\alpha'_{S_{\{y\}}} = s_2\overline{s_3}\overline{s_4}s_6$
    - SOLVE-NNF($\exists y, \varphi, \alpha'_{S_{\{y\}}}$)
    - SOLVE($\theta_{\{y\}}, \alpha'_{S_{\{y\}}}$) = Sat($y$)
    - SOLVE($\overline{\theta}_{\{y\}}, y\,\overline{\alpha}'_{S_{\{y\}}}$) = Unsat($y\,\overline{s_2}\overline{s_6}$)
    - **return** Sat($s_2s_6$)
  - $\theta'_{\{v,w\}} = \theta_{\{v,w\}} \wedge (\overline{a_2} \vee \overline{a_6})$
  - SAT($\theta_{\{v,w\}}, \alpha'_{S_{\{v,w\}}}$) = Sat($v\,\overline{w}\,a_2\overline{a_3}a_4\overline{a_6}$)
  - $\alpha''_{S_{\{y\}}} = s_2\overline{s_3}s_4\overline{s_6}$
    - SOLVE-NNF($\exists y, \varphi, \alpha''_{S_{\{y\}}}$)
    - SOLVE($\theta_{\{y\}}, \alpha''_{S_{\{y\}}}$) = Sat($\overline{y}$)
    - SOLVE($\overline{\theta}_{\{y\}}, \overline{y}\,\overline{\alpha}''_{S_{\{y\}}}$) = Unsat($\overline{y}\,\overline{s_2}\overline{s_4}$)
    - **return** Sat($s_2s_4$)
  - $\theta''_{\{v,w\}} = \theta'_{\{v,w\}} \wedge (\overline{a_2} \vee \overline{a_4})$
  - SAT($\theta''_{\{v,w\}}, \alpha'_{S_{\{v,w\}}}$) = Unsat($\overline{s_2}$)
  - **return** Sat($\overline{s_2}$)
- SOLVE-NNF($\exists x, \forall v, w. \exists y. \varphi, \{\}$) returns Sat

### 4.1.2 Correctness

→ Page 30

The proof of correctness requires the same high level argumentation as the correctness proof for the prenex conjunctive normal form algorithm in → Section 2.3.2. The argumentation over the abstraction and negation normal form formulas is, however, much more sophisticated than the argumentation over clauses in a matrix. Thus, in this section, we give a rigorous argumentation for soundness and completeness, even though there is some repetition and overlap with Section 2.3.2. Remember, that we fixed a QBF $\Phi = \mathcal{Q}X_1 \cdots \mathcal{Q}X_n.\, \varphi$ with quantifier prefix $\mathcal{Q}X_1 \cdots \mathcal{Q}X_n$ and propositional body $\varphi$ in NNF. Further, we assume that $\psi_1, \ldots, \psi_m$ are the non-literal subformulas of $\varphi$.

Before going into detail, we outline the structure of this section. First, we establish a relation between assignments of satisfaction variables $\alpha_S$ and their effect on the QBF, analogously to Section 2.3.2. For some quantifier alternation $\mathcal{Q}X.\, \overline{\mathcal{Q}}Y$, we show how assignments $\alpha_{S_X}$ with respect to quantifier $\mathcal{Q}X$ are related to assignments $\alpha_{S_Y}$ w.r.t. quantifier $\overline{\mathcal{Q}}Y$. Afterward, we establish statements over the abstractions, the first (Lemma 4.7) covering the base case of the structural induction. Furthermore, we show that the abstractions $\theta_X$ and $\overline{\theta}_X$ are effectively dual, which leads to the correctness of the dual propagation in Lemma 4.9. The actual proof of correctness is carried out in Lemma 4.10.

**Duality in NNF representation.** To match the assignment of the satisfaction variables $\alpha_S$ with the corresponding valuation of the propositional formula $\varphi$, we define a partial function that maps subformulas of $\varphi$ to a Boolean valuation $\mathbb{B}$ or undefined $\perp$. We use the convention to write such subformula valuation functions as $\beta_\varphi \colon sf(\varphi) \to \mathbb{B}_\perp$, i.e., we index the partial function by a propositional formula. Then, similar to the correctness proof of clausal abstraction in Section 2.3.2, we define an operation $\Phi|_{\beta_\varphi}^{\mathcal{Q}X}$, for a QBF $\Phi$, quantifier $\mathcal{Q}X$, and subformula valuation $\beta_\varphi$, as the QBF with the same prefix as $\Phi$ with propositional formula $\varphi'$ resulting from replacing subformulas $\psi_i$ by their valuation $\beta_\varphi(\psi_i)$ if it is defined. Potentially occurring free variables, which were in the original QBF variables bound by outer quantifiers, are removed by this operation.

Formally, the propositional part of $\Phi|_{\beta_\varphi}^{\mathcal{Q}X}$ is defined as the partial evaluation of $\varphi$ according to the subformula valuation $\beta_\varphi$. Therefore, we use a partial evaluation function $parteval(\psi, \beta_\varphi)$ that maps a propositional formula $\psi$ and a subformula valuation $\beta_\varphi$ to a propositional formula. It is defined as

$$parteval(\psi, \beta_\varphi) = \begin{cases} \beta_\varphi(\psi) & \text{if } \beta_\varphi(\psi) \neq \perp \\ \bigwedge\limits_{\substack{\psi' \in dsf(\psi) \\ parteval(\psi', \beta_\varphi) \neq \perp}} parteval(\psi', \beta_\varphi) & \text{if } type(\psi) = \wedge \\ \bigvee\limits_{\substack{\psi' \in dsf(\psi) \\ parteval(\psi', \beta_\varphi) \neq \perp}} parteval(\psi', \beta_\varphi) & \text{if } type(\psi) = \vee \\ \psi & \text{if } type(\psi) = lit \text{ and } \psi \text{ is bound} \\ \perp & \text{otherwise} \end{cases}$$

Lastly, we need to define the subformula valuation function corresponding to some assignment of satisfaction variables $\alpha_S$. An assignment of satisfaction variables $\alpha_S$ represents the subformula valuation $\beta_\varphi \coloneqq \mathit{sfval}_\varphi(\alpha_S)$ where $\mathit{sfval}_\varphi(\alpha_S)$ is defined as

$$\mathit{sfval}_\varphi(\alpha_S)(\psi_i) = \begin{cases} \mathbf{T} & \text{if } s_i \in \mathrm{dom}(\alpha_S) \wedge (\mathit{type}(\psi_i) = \vee) \wedge \alpha_S(s_i) = \mathbf{T} \\ \mathbf{F} & \text{if } s_i \in \mathrm{dom}(\alpha_S) \wedge (\mathit{type}(\psi_i) = \wedge) \wedge \alpha_S(s_i) = \mathbf{F} \\ \bot & \text{otherwise} \end{cases} \tag{4.6}$$

Then, we define the shorthand notation $\Phi|^{\mathcal{Q}X}_{\alpha_S}$ as $\Phi|^{\mathcal{Q}X}_{\beta_\varphi}$, where $\beta_\varphi \coloneqq \mathit{sfval}_\varphi(\alpha_S)$.

Many times in this section, we will argue about *duality*. To make this reasoning precise, we begin with a formal justification using two lemmata. Recall that we denote by $\overline{\beta}$ the complement of the partial assignment $\beta$. The following lemma states that an assignment $\alpha_{S_X}$ for $\Phi$ corresponds to the negated assignment $\overline{\alpha}_{S_X}$ in the negated formula $\overline{\Phi}$.

**Lemma 4.4** (Duality)**.** *Let $\Phi$ be a QBF with propositional formula $\varphi$, let $\mathcal{Q}X$ be some quantifier of $\Phi$, and let $\alpha_{S_X}$ be an assignment to the satisfaction variables. $\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}}$ is true if, and only if, $\overline{\Phi}|^{\overline{\mathcal{Q}}X}_{\overline{\alpha}_{S_X}}$ is false.*

*Proof.* Let $\beta_\varphi \coloneqq \mathit{sfval}_\varphi(\alpha_{S_X})$ and let $\beta_{\overline{\varphi}} \coloneqq \mathit{sfval}_{\overline{\varphi}}(\overline{\alpha}_{S_X})$. It holds that $\overline{\beta}_\varphi = \beta_{\overline{\varphi}}$ by the definition of *sfval* in Equation 4.6. For every QBF $\Phi$ it holds that $\Phi$ is true if, and only if, $\overline{\Phi}$ is false. Together, this shows that $\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}}$ is true iff $\overline{\Phi}|^{\overline{\mathcal{Q}}X}_{\overline{\alpha}_{S_X}}$ is false. □

In the correctness proof below, we will argue over optimal assumption assignments for some quantifier $\mathcal{Q}X$, that is, assignments of assumption variables $\alpha^*_{A_X}$ that are minimal with respect to the number of assumptions $\alpha^*_{A_X}(a_i) = \mathbf{T}$. The following lemma establishes this form of reasoning for quantifier alternations by proving equisatisfiability between $(\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}})[\alpha_X]$ and $\Phi|^{\overline{\mathcal{Q}}Y}_{\alpha^*_{S_Y}}$ for some "optimal" $\alpha_{S_Y}$ constructed from $\alpha_{S_X}$ and $\alpha_X$ analogously to Lemma 2.11. In the proof, we argue over *consistency* of *complete* subformula assignments, which means that the assignment respects the propositional formula. A complete subformula assignment $\alpha_\varphi \colon \mathit{sf}(\varphi) \to \mathbb{B}$ is *consistent*, if and only if, for every (non-literal) subformula $\psi_i$ of $\varphi$ it holds that

$$\alpha_\varphi(\psi_i) = \begin{cases} \bigwedge_{\psi_j \in \mathit{dsf}(\psi_i)} \alpha_\varphi(\psi_j) & \text{if } \mathit{type}(\psi_i) = \wedge \\ \bigvee_{\psi_j \in \mathit{dsf}(\psi_i)} \alpha_\varphi(\psi_j) & \text{otherwise} \end{cases}$$

**Lemma 4.5.** *Let $\mathcal{Q}X.\, \overline{\mathcal{Q}}Y$ be a quantifier alternation of a QBF $\Phi$ with propositional formula $\varphi$ and let $\alpha_X$ and $\alpha_{S_X}$ be assignments. Further, let $\alpha^*_{S_Y}$ be defined such that $\alpha^*_{S_Y}(s_i) = \mathbf{F}$ if, and only if, $\alpha_X \dot{\cup}\, \alpha_{S_X} \vDash \mathit{enc}(\psi_i)$ (for quantifier $\mathcal{Q}X$). It holds that $(\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}})[\alpha_X]$ and $\Phi|^{\overline{\mathcal{Q}}Y}_{\alpha^*_{S_Y}}$ are equisatisfiable.*

*Proof.* We prove the statement for quantifier alternations of the form $\exists X.\, \forall Y$, the case $\forall X.\, \exists Y$ then follows by Lemma 4.4. The quantified formulas $(\Phi|^{\exists X}_{\alpha_{S_X}})[\alpha_X]$ and $\Phi|^{\forall Y}_{\alpha^*_{S_Y}}$

have the same quantifier prefix (starting with $\forall Y$) and equisatisfiable propositional formula. We show the latter by proving equality over the corresponding subformula assignments. Let $\beta_\varphi \coloneqq sfval_\varphi(\alpha_{S_X})$. We augment $\beta_\varphi$ with $\alpha_X$, that is, we define $\beta'_\varphi \coloneqq \beta_\varphi \sqcup \alpha_X$. Let $\beta_{\overline{\varphi}} = sfval_{\overline{\varphi}}(\alpha^*_{S_Y})$ be the subformula valuation corresponding to $\alpha^*_{S_Y}$. Note that $sfval_{\overline{\varphi}}(\alpha^*_{S_Y})$ is defined with respect to negated subformulas, that is, $\overline{\psi_i} \in \overline{\varphi}$ as it corresponds to a universal quantifier $\forall Y$ (and is thus equivalent to the existential quantifier $\exists Y$ over the dual propositional formula $\overline{\varphi}$). We show that the assignments $\beta'_\varphi$ and $\beta_{\overline{\varphi}}$ are dual with respect to the satisfaction variables for quantifier $\forall Y$. As the assignment $\alpha_X$ may propagate subformula valuations beyond the boundaries given by the satisfaction variables, we prove the following strengthening: For every complete and consistent extension $\alpha_\varphi$ of $\beta'_\varphi$ ($\beta'_\varphi \sqsubseteq \alpha_\varphi$) and every $s_i \in S_Y$ it holds that $\alpha_\varphi(\psi_i) = \overline{\beta}_{\overline{\varphi}}(\overline{\psi_i})$ if $\beta_{\overline{\varphi}}(\overline{\psi_i}) \neq \bot$.

Let $\beta_{\overline{\varphi}}(\overline{\psi_i}) = \mathbf{T}$ (analogous for $\beta_{\overline{\varphi}}(\overline{\psi_i}) = \mathbf{F}$). Then, by definition of $sfval_{\overline{\varphi}}(\alpha^*_{S_Y})(\overline{\psi_i})$ in Equation 4.6 it holds that $type(\overline{\psi_i}) = \vee$ and $\alpha^*_{S_Y}(\overline{s_i}) = \mathbf{T}$. By the definition of $\alpha^*_{S_Y}$, we know that $\alpha_X \dot{\sqcup} \alpha_{S_X} \not\models enc(\psi_i)$. By the definition of the abstraction $\theta_X$, for every $s_i \in S_Y$, there is a $\psi_j \in dsf(\psi_i)$ such that $\psi_j = \psi_j^<$, that is, $\psi_j$ is only influenced by outer variables (with respect to $X$). A recursive argument over $enc(\psi_i)$ shows that, $\alpha_X \dot{\sqcup} \alpha_{S_X} \not\models enc(\psi_i)$ implies that for every complete and consistent subformula valuation $\alpha_\varphi(\psi_i) = \mathbf{F}$ has to hold.

Further, for every complete and consistent extension $\alpha_{\overline{\varphi}}$ of $\beta_{\overline{\varphi}}$ with the same variable assignments as $\alpha_\varphi$ ($\alpha_{\overline{\varphi}}(v) = \alpha_\varphi(v)$ for every bound variable $v$), it holds that $\alpha_{\overline{\varphi}}(\overline{\psi_i}) = \overline{\alpha}_\varphi(\psi_i)$ by duality and the previous statement. □

If the assignments are not optimal, there is a monotonicity property on the satisfaction assignments stated below.

**Lemma 4.6** (Monotonicity of $\alpha_{S_X}$)**.** *Let $\mathcal{Q}X$ be a quantifier of QBF $\Phi$ and let $\alpha_{S_X}$ be an assignment such that $\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}}$ is winning for $\mathcal{Q}X$. For every $\alpha'_{S_X}$ with $\alpha^+_{S_X} \sqsubseteq {\alpha'_{S_X}}^+$ it holds that $\Phi|^{\mathcal{Q}X}_{\alpha'_{S_X}}$ is winning for $\mathcal{Q}X$.*

*Proof.* We prove the statement for $\exists X$, the universal case is analogous. Let $\alpha_{S_X}$ be given such that $\Phi|^{\exists X}_{\alpha_{S_X}}$ is winning for $\exists X$. Further, choose some arbitrary $\alpha'_{S_X}$ with $\alpha^+_{S_X} \sqsubseteq {\alpha'_{S_X}}^+$. The subformula valuations $\beta_\varphi$ and $\beta'_\varphi$ corresponding to $\alpha_{S_X}$ and $\alpha'_{S_X}$, respectively, are monotone as well: If $\beta_\varphi(\psi_i) = \mathbf{T}$ it follows that $\beta'_\varphi(\psi_i) = \mathbf{T}$ by definition in Equation 4.6. □

**Reasoning over abstractions $\theta_X$ and $\overline{\theta}_X$.** In this part, we focus on the two types of abstractions used in the algorithm. First, we have a formal statement regarding equisatisfiability of the innermost abstraction and the circuit representation for a given assignment of the satisfaction variables $\alpha_S$, similar to Lemma 2.10.1.

**Lemma 4.7.** *Let $\Phi$ be a QBF with propositional formula $\varphi$, let $\exists X$ be the innermost quantifier, and let $\alpha_{S_X}$ be an assignment over variables $S_X$. It holds that $\theta_X[\alpha_{S_X}]$ is equisatisfiable to $\Phi|^{\exists X}_{\alpha_{S_X}}$.*

*Proof.* As $\exists X$ is the innermost quantifier, all variables in $\varphi$ are either bound by $\exists X$ or by some outer quantifier. By definition in Equation 4.2, the abstraction is $\theta_X = enc(\varphi)$. Note

that $\varphi$ and $enc(\varphi)$ are identical up to subformulas $\psi$ of $\varphi$ with only outer influence ($\psi = \psi^{<}$), where $\psi$ is replaced in $enc(\varphi)$ by a satisfaction variable. Let $\alpha_{S_X}$ be some assignment over satisfaction variables $S_X$. By the definition of $enc(\varphi)$ (Equation 4.3), replacing $s_i$ with $\alpha_{S_X}(s_i)$ leads to formulas which are equal to $\mathbf{T}$ if $\psi_i$ is disjunctive and $\alpha_{S_X}(s_i) = \mathbf{T}$, and to $\mathbf{F}$ if $\psi_i$ is conjunctive and $\alpha_{S_X}(s_i) = \mathbf{F}$. Otherwise, the variable $s_i$ is just removed from the encoded formula $enc(\varphi)$. This matches the definition of the subformula valuation function $\beta_\varphi$ resulting from $\alpha_{S_X}$, thus,

$$\Phi|^{\exists X}_{\alpha_{S_X}} = \Phi|^{\exists X}_{\beta_\varphi} = enc(\varphi)[\alpha_{S_X}] = \theta_X[\alpha_{S_X}] \ ,$$

which we show by structural induction over $\varphi$. Let $\beta_\varphi$ be the subformula valuation corresponding to $\alpha_{S_X}$. We show that $\Phi|^{\exists X}_{\beta_\varphi}$ is equal to $\theta_X[\alpha_{S_X}]$. Let $\psi_i$ be an arbitrary non-literal subformula of $\varphi$. Further, let $\psi_j$ be an arbitrary direct subformula $\psi_j \in dsf(\psi_i)$. We perform a case distinction on $\psi_j$:

- Let $\psi_j = \psi_j^{=}$, thus, $\psi_j$ contains only variables $X$. The encoding of $\psi_j$ is equal in both cases, as $enc_{\psi_i}(\psi_j) = \psi_j$ and $\beta_\varphi(\psi_j) = \bot$.
- Let $\psi_j = \psi_j^{<}$, thus, $\psi_j$ contains only variables bound at outer quantifiers. Thus, $enc_{\psi_i}(\psi_j) = s_i$ and the subformula is replaced by a constant in $\Phi|^{\exists X}_{\beta_\varphi}$. Since we replace $s_i$ with $\alpha_{S_X}(s_i)$, we do a further case distinction on $type(\psi_i)$ and $\alpha_{S_X}(s_i)$.
  - Assume $type(\psi_i) = \wedge$ and $\alpha_{S_X}(s_i) = \mathbf{T}$, thus, assigning $s_i$ positively in $enc(\psi_i)$ has the same effect as removing $\psi_j$.
  - Assume $type(\psi_i) = \vee$ and $\alpha_{S_X}(s_i) = \mathbf{F}$, thus, assigning $s_i$ negatively in $enc(\psi_i)$ has the same effect as removing $\psi_j$.
  - Assume $type(\psi_i) = \wedge$ and $\alpha_{S_X}(s_i) = \mathbf{F}$, thus, assigning $s_i$ negatively in $enc(\psi_i)$ makes $enc(\psi_i)$ unsatisfiable. By definition in Equation 4.6, $\beta_\varphi(\psi_i) = \mathbf{F}$ as well.
  - Assume $type(\psi_i) = \vee$ and $\alpha_{S_X}(s_i) = \mathbf{T}$, thus, assigning $s_i$ positively in $enc(\psi_i)$ makes $enc(\psi_i)$ valid. By definition in Equation 4.6, $\beta_\varphi(\psi_i) = \mathbf{T}$ as well.

If neither of the base cases above applies, the claim follows by induction. □

The following two lemmata formalize the duality of the abstraction. These statements are used to argue over the dual abstraction. The former states that the abstraction is dual with respect to negation of the formula except for the satisfaction variables. It shows that the dual abstraction is unsatisfiable when assuming a satisfying assignment of the abstraction. The latter lemma shows the correctness of the dual propagation for the innermost quantifier.

**Lemma 4.8** (Duality of $\theta_X$)**.** *Let $\Phi$ be a QBF with propositional formula $\varphi$, let $\exists X$ be a quantifier of $\Phi$, and let $\alpha_X$ and $\alpha_{S_X}$ be assignments. It holds that*

$$enc(\psi_i)[\alpha_X \dot\cup \alpha_{S_X}] \leftrightarrow \neg enc(\overline{\psi_i})[\alpha_X \dot\cup \overline{\alpha}_{S_X}] \ .$$

*Proof.* As $\alpha_{S_X}$ abstracts the outer assignments as subformula valuations, $\alpha_{S_X}$ needs to be negated to represent the same assignments in $\overline{\theta}_X$. By structural induction, it is straightforward to show that $enc(\psi_i)$ and $enc(\overline{\psi_i})$ are dual with the exception of variables from $S$: A disjunction in $enc(\psi_i)$ is a conjunction in $enc(\overline{\psi_i})$ and vice versa, a literal $l$ of quantifier $\exists X$ in $enc(\psi_i)$ is negated $\neg l$ in $enc(\overline{\psi_i})$. Only satisfaction variables appear positively in both formulas. □

**Lemma 4.9.** *Let $\Phi$ be a QBF with propositional formula $\varphi$, let $\exists X$ be the innermost quantifier, and let $\alpha_X$ and $\alpha_{S_X}$ be satisfying assignments of $\theta_X$. It holds that $\overline{\theta}_X[\alpha_X \dot{\cup} \overline{\alpha}_{S_X}]$ is unsatisfiable. Let $\beta$ be some set of failed assumptions, that is, $\beta \sqsubseteq \alpha_X \dot{\cup} \overline{\alpha}_{S_X}$ and $\overline{\theta}_X[\beta]$ is unsatisfiable. Then, $\overline{\beta}|_{S_X} \sqsubseteq \alpha^+_{S_X}$ and $\Phi|^{\exists X}_{\overline{\beta}|_{S_X}[\bot \mapsto \mathbf{F}]}$ is true.*

*Proof.* By the definition of the abstractions, it holds that $\theta_X = enc(\varphi)$ and $\overline{\theta}_X = enc(\overline{\varphi})$. Lemma 4.8 shows that if $\alpha_X \dot{\cup} \alpha_{S_X}$ is a satisfying assignment of $enc(\varphi)$, the assignment $\alpha_X \dot{\cup} \overline{\alpha}_{S_X}$ falsifies $enc(\overline{\varphi})$. By definition of failed assumptions, $\beta \sqsubseteq \alpha_X \dot{\cup} \overline{\alpha}_{S_X}$, i.e., there is no $\alpha$ with $\beta \sqsubseteq \alpha$ that satisfies $\overline{\theta}_X$, hence, all $\alpha^*_{S_X}$ with $\overline{\beta}|_{S_X} \sqsubseteq \alpha^*_{S_X}$ satisfy $\theta_X[\alpha_X]$. Together with Lemma 4.7, this shows that $\Phi|^{\exists X}_{\overline{\beta}|_{S_X}[\bot \mapsto \mathbf{F}]}$ is true. □

**Correctness of SOLVE-NNF.** Finally, we can prove the correctness of the SOLVE-NNF algorithm. As in the case for CNF, we prove the correctness by induction over the quantifier prefix.

**Lemma 4.10.** *Let $\Phi$ be a QBF with propositional formula $\varphi$, let $\mathcal{Q}X.\,\Psi$ be a quantified subformula of $\Phi$, and let $\alpha_{S_X}$ be an assignment of the satisfaction variables $S_X$.*

- *If $\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}}$ is winning for $\mathcal{Q}X$, then SOLVE-NNF($\mathcal{Q}X$, $\Psi$, $\alpha_{S_X}$) returns $\mathsf{Sat}_{\mathcal{Q}}(\beta_{S_X})$ where $\beta_{S_X} \sqsubseteq \alpha^+_{S_X}$ and $\Phi|^{\mathcal{Q}X}_{\beta_{S_X}[\bot \mapsto \mathbf{F}]}$ is winning for $\mathcal{Q}X$.*
- *If $\Phi|^{\mathcal{Q}X}_{\alpha_{S_X}}$ is losing for $\mathcal{Q}X$, then SOLVE-NNF($\mathcal{Q}X$, $\Psi$, $\alpha_{S_X}$) returns $\mathsf{Unsat}_{\mathcal{Q}}(\beta_{S_X})$ where $\beta_{S_X} \sqsubseteq \alpha^-_{S_X}$ and $\Phi|^{\mathcal{Q}X}_{\beta_{S_X}[\bot \mapsto \mathbf{T}]}$ is losing for $\mathcal{Q}X$.*

*Proof.* We prove the statement by structural induction over the quantifier prefix. For this proof, we can restrict $\mathcal{Q}$ to $\exists$ as the universal case is completely dual (Lemma 4.4). The base case $\exists X$ distinguishes whether $\Phi|^{\exists X}_{\alpha_{S_X}}$ is true or false. In both cases, we use the equisatisfiability of $\Phi|^{\exists X}_{\alpha_{S_X}}$ and $\theta_X[\alpha_{S_X}]$ (Lemma 4.7). In case the formula is true, we additionally have to use the correctness of the dual propagation as established in Lemma 4.9. In the induction step, i.e., a quantifier alternation $\exists X.\,\forall Y$, we perform a case distinction on the value of $\Phi|^{\exists X}_{\alpha_{S_X}}$ as well. If it is true, there is a satisfying assignment $\alpha_X$ such that $\Phi|^{\exists X}_{\alpha_{S_X}}[\alpha_X]$ is losing for $\forall Y$. Applying induction hypothesis and dual propagation gives the required witness. In case the abstraction produces falsifying assignments, the subsequent refinement excludes them from the abstraction, hence, eventually a satisfying assignment is reached. If $\Phi|^{\exists X}_{\alpha_{S_X}}$ is false, every assignment $\alpha_X$ is winning for $\forall Y$ and, thus, leads to a refinement of the abstraction $\theta_X$. The abstraction becomes eventually unsatisfiable (under the assignment $\alpha_{S_X}$) and the failed assumption represents the required witness. The detailed proof follows.

*Induction Base.* Let $\exists X.\, \varphi$ be the innermost quantifier of $\Phi$ and let $\alpha_{S_X}$ be some assignment over $S_X$. We distinguish whether $\Phi|^{\exists X}_{\alpha_{S_X}}$ is true or false:

- Assume that $\Phi|^{\exists X}_{\alpha_{S_X}}$ is true. By Lemma 4.7, the truth of $\Phi|^{\exists X}_{\alpha_{S_X}}$ witnesses the satisfiability of $\theta_X[\alpha_{S_X}]$. By Lemma 4.9, the return value of SOLVE-NNF in line 10 meets the requirements.
- Assume that $\Phi|^{\exists X}_{\alpha_{S_X}}$ is false. By Lemma 4.7 it follows that $\theta_X[\alpha_{S_X}]$ is unsatisfiable. Thus, the algorithm returns $\mathrm{Unsat}(\beta_{S_X})$ where $\beta_{S_X}$ are the failed assumptions of $\mathrm{SAT}(\theta_X, \alpha_{S_X})$ which implies that $\beta_{S_X} \sqsubseteq \alpha^-_{S_X}$. As $\theta_X[\beta_{S_X}[\bot \mapsto \mathbf{T}]]$ is unsatisfiable, $\Phi|^{\exists X}_{\beta_{S_X}[\bot\mapsto\mathbf{T}]}$ is false by Lemma 4.7, thus, $\beta$ meets the requirements.

The base case for universal formulas $\forall X.\, \varphi$ follows from the existential cases by Lemma 4.4.

*Induction Step.* Let $\exists X.\, \forall Y$ be a quantifier alternation of $\Phi$ and let $\alpha_{S_X}$ be some assignment over $S_X$. We distinguish whether $\Phi|^{\exists X}_{\alpha_{S_X}}$ is true or false:

- Assume that $\Phi|^{\exists X}_{\alpha_{S_X}}$ is true. Thus, there is a satisfying assignment $\alpha_X$ for the variables $X$ such that $(\Phi|^{\exists X}_{\alpha_{S_X}})[\alpha_X]$ is true. We define the "optimal" assignment of the assumption variables $\alpha^*_{A_X}$, that is, the minimal assignment with respect to the number of assumptions ($\alpha^*_A(a_i) = \mathbf{T}$) for the given assignment $\alpha_X$, as

$$\alpha^*_{A_X}(a_i) = \begin{cases} \mathbf{F} & \text{if } \alpha_X \dot\cup\, \alpha_{S_X} \vDash enc(\psi_i) \\ \mathbf{T} & \text{otherwise} \end{cases} .$$

  The definition of the abstraction $\theta_X$ (Equation 4.2) is

$$\theta_X = \bigwedge_{a_i \in A_X} a_i \vee enc(\psi_i) \ .$$

  The combined assignment $\alpha_X \dot\cup\, \alpha^*_{A_X}$ is, thus, a satisfying assignment of the abstraction $\theta_X[\alpha_{S_X}]$ initially. We perform a case distinction on the returned assignment of the SAT solver in line 3.

  – We assume that the SAT call in line 3 returns $\alpha_X \dot\cup \alpha^*_{A_X}$. Let $\alpha^*_{S_Y}$ be the assignment constructed in line 5. By Lemma 4.5, it holds that $(\Phi|^{\exists X}_{\alpha_{S_X}})[\alpha_X] = \Phi|^{\forall Y}_{\alpha^*_{S_Y}}$ is true and, thus, losing for $\forall Y$. By induction hypothesis we deduce that SOLVE-NNF($\forall Y.\, \Psi, \alpha^*_{S_Y}$) returns $\mathrm{Sat}(\beta_{S_Y})$ with $\beta_{S_Y} \sqsubseteq \alpha^-_{S_Y}$ where $\Phi|^{\forall Y}_{\beta_{S_Y}[\bot\mapsto 1]}$ is true. Subsequently, the dual abstraction $\overline{\theta}_X$ is refined (line 7) and SOLVE-NNF returns $\mathrm{Sat}(\beta_{S_X})$ where $\beta_{S_X} = \textsc{optimize}(\alpha_X, \alpha_{S_X})$ (line 8).
    It remains to show that $\Phi|^{\exists X}_{\beta_{S_X}[\bot\mapsto\mathbf{F}]}$ is true and $\beta_{S_X} \sqsubseteq \alpha^+_{S_X}$. First, we show that $\overline{\theta}_X[\alpha_X \dot\cup\, \overline{\alpha}_{S_X}]$ is unsatisfiable. Initially, the dual abstraction is defined as

$$\overline{\theta}_X = \bigwedge_{a_i \in A_X} a_i \vee enc(\overline{\psi_i}) \ .$$

    The refinement clause for the dual abstraction is $\xi := \bigvee_{s_i \in \beta^0_{S_Y}} \overline{a_i}$ (line 7). As established by Lemma 4.8, for every $a_i \in A_X$ it holds that $enc(\psi_i)[\alpha_X \dot\cup$

$\alpha_{S_X}] \leftrightarrow \neg enc(\overline{\psi_i})[\alpha_X \dot\cup \overline{\alpha}_{S_X}]$. By the definition of $\theta_X$, for every $a_i \in \alpha^0_{A_X}$ it holds that $enc(\psi_i)[\alpha_X \dot\cup \alpha_{S_X}] = \mathbf{T}$. As $\alpha_{A_X}(a_i) = \alpha_{S_Y}(s_i)$ for every $a_i \in A_X$ and $\beta_{S_Y} \sqsubseteq \alpha^-_{S_Y}$ it follows that $enc(\overline{\psi_i})[\alpha_X \dot\cup \overline{\alpha}_{S_X}] = \mathbf{F}$ for every $s_i \in \beta^0_{S_Y}$. This shows that $\overline{\theta}_X[\alpha_X \dot\cup \overline{\alpha}_{S_X}]$ is unsatisfiable after the refinement $\xi$. Let $\beta$ be the failed assumptions. The returned assignment is $\beta_{S_X} = \overline{\beta}|_{S_X}$, thus $\beta_{S_X} \sqsubseteq \alpha^+_{S_X}$. For every $\alpha'_{S_X}$ with $\beta_{S_X} \sqsubseteq \alpha'_{S_X}$ it holds that $\overline{\theta}_X[\alpha_X \dot\cup \overline{\alpha}'_{S_X}]$ is unsatisfiable as it falsifies the refinement $\xi$. Thus, one can define a corresponding optimal $\alpha'_{A_X}$ that satisfies $\theta_X$ and for the resulting $\alpha'_{S_Y}$ it holds that $\Phi|^{\forall Y}_{\alpha'_{S_Y}}$ is true as $\beta_{S_Y} \sqsubseteq \alpha'^-_{S_Y}$. Hence, $\Phi|^{\exists X}_{\beta_{S_X}[\bot \mapsto \mathbf{F}]}$ is true.

– Assume that the SAT call in line 3 returns a different assumption $\alpha'_{A_X}$. Either $\alpha'_{A_X}$ corresponds to $\alpha_X$ and is non-minimal, i.e., ${\alpha^*_{A_X}}^+ \sqsubseteq {\alpha'_{A_X}}^+$, or it corresponds to a different assignment $\alpha'_X$. The call to SOLVE-NNF may either return Sat or a counterexample Unsat$(\beta_{S_Y})$ with $\beta_{S_Y} \sqsubseteq \alpha^+_{S_Y}$. We consider the latter case as in the former case SOLVE-NNF also returns Sat and the same argumentation as in the previous case applies.

  The subsequent refinement in line 9 requires that one of the not satisfied subformulas $\psi_i$ with $\beta_{S_Y}(s_i) = \alpha_{A_X}(a_i) = \mathbf{T}$ has to be satisfied in the next iteration and the corresponding refinement clause is $\xi \coloneqq \bigvee_{s_i \in \beta^1_{S_Y}} \overline{a_i}$. By construction of $\alpha^*_{A_X}$ as the minimal assignment corresponding to $\alpha_X$, $\alpha^*_{A_X} \not\models \xi$ contradicts that $\alpha_X$ is a satisfying assignment of $\Phi|^{\exists X}_{\alpha_{S_X}}$. Hence, $\alpha_X \dot\cup \alpha^*_{A_X}$ is still a satisfying assignment for the refined abstraction $\theta'_X[\alpha_{S_X}]$. The refinement also reduces the number of $A_X$ assignments by at least 1 and, thus, brings us one step closer to a satisfying assignment.

· Assume that $\Phi|^{\exists X}_{\alpha_{S_X}}$ is false. For every assignment $\alpha_X$, it holds that $(\Phi|^{\exists X}_{\alpha_{S_X}})[\alpha_X]$ is false. The abstraction $\theta_X$ is initially satisfiable for every choice of $\alpha_{S_X}$ (every $a_i$ can be set to true, see Equation 4.2). Let $\alpha$ be a such satisfying assignment of $\theta_X[\alpha_{S_X}]$. We define $\alpha_X \coloneqq \alpha|_X$ and $\alpha_{A_X} \coloneqq \alpha|_{A_X}$. By construction of $\theta_X$ (Equation 4.2), $\alpha_X \dot\cup \alpha_{S_X} \not\models enc(\psi_i)$ implies that $\alpha_{A_X}(a_i) = \mathbf{T}$. We define the assignment with optimal assumptions $\alpha^*_{A_X}$ as $\alpha^*_{A_X}(a_i) = \mathbf{F}$ if, and only if, $\alpha_X \sqcup \alpha_{S_X} \models enc(\psi_i)$. Note that $\alpha_X \dot\cup \alpha^*_{A_X}$ is a satisfying assignment of $\theta_X[\alpha_{S_X}]$. We show that even with optimal assumptions $\alpha^*_{A_X}$, the quantified subformula is unsatisfiable and the subsequent refinement step excludes assignment $\alpha_{A_X}$ from the abstraction $\theta_X$.

  Let $\alpha'_{S_Y}$ and $\alpha^*_{S_Y}$ be the assignments after line 5 with respect to $\alpha_{A_X}$ and $\alpha^*_{A_X}$, respectively. From the construction, we know that ${\alpha^*_{A_X}}^+ \sqsubseteq {\alpha_{A_X}}^+$, by the optimality of $\alpha^*_A$, and thereby ${\alpha^*_{S_Y}}^+ \sqsubseteq {\alpha'_{S_Y}}^+$. By Lemma 4.5, it holds that $(\Phi|^{\exists X}_{\alpha_{S_X}})[\alpha_X]$ and $\Phi|^{\forall Y}_{\alpha^*_{S_X}}$ are equisatisfiable and, thus, winning for $\forall$. By the monotonicity condition given in Lemma 4.6, it follows that $\Phi|^{\forall X}_{\alpha'_{S_X}}$ is false as well. By induction hypothesis, SOLVE-NNF$(\forall Y, \Psi, \alpha'_{S_X})$ returns Unsat$(\beta_{S_Y})$ such that $\beta_{S_Y} \sqsubseteq {\alpha'_{S_Y}}^+$ and $\Phi|^{\forall}_{\beta_{S_Y}[\bot \mapsto \mathbf{F}]}$ is false. As $\beta^1_{S_Y} \sqsubseteq {\alpha'_{S_Y}}^1 = \{a_i \in A_X \mid \alpha_A(a_i) = \mathbf{T}\}$, the following refinement with clause $\bigvee_{s_i \in \beta^1_S} \overline{a_i}$ excludes assignment $\alpha_{A_X}$ from $\theta_X$. As there are only finitely many

refinement clauses, the SAT call in line 3 eventually becomes unsatisfiable when assuming $\alpha_{S_X}$. Let $\theta'_X$ be the abstraction at this point and let $\beta'_{S_X}$ be the failed assumptions, i.e., $\beta'_{S_X} \sqsubseteq \alpha^+_{S_X}$.

Let $\alpha''_{S_X} = \beta'_{S_X}[\bot \mapsto \mathbf{T}]$. It remains to show that $\Phi|^{\exists X}_{\alpha''_{S_X}}$ is false. Assume for contradiction that there is some $\alpha_X$ such that $(\Phi|^{\exists}_{\alpha''_{S_X}})[\alpha_X]$ is true. It holds that $\theta'_X[\alpha_X \dot\sqcup \alpha''_{S_X}]$ is unsatisfiable, whereas $\theta_X[\alpha_X \dot\sqcup \alpha''_{S_X}]$ is satisfiable. Thus, the assignment $\alpha_X$ was excluded due to refinements. Let $\alpha''_{A_X}$ be the optimal assumption assignment corresponding to $\alpha_X$. As the refinement only excludes $A_X$ assignments corresponding to some $S_Y$ assignment $\beta''_{S_Y}$ such that $\Phi|^{\forall}_{\beta''_{S_Y}[\bot \mapsto \mathbf{F}]}$ is false, which contradicts our assumption.

The induction step for quantifier alternation $\forall X.\exists Y$ follows from $\exists X.\forall Y$ and Lemma 4.4. □

Since the main algorithm SOLVE directly calls into SOLVE-NNF, the following theorem follows immediately from Lemma 4.10.

**Theorem 4.11.** *SOLVE returns* Sat *if, and only if,* $\Phi$ *is true.*

### 4.1.3 Optimizations

In this section, we describe optimizations for the algorithm. Compared to CNF, there are less opportunities in the algorithm as the dual abstraction already takes care of generating and translating witnesses.

As shown in the last section, the satisfaction assignments $\alpha_S$ correspond to partial formula evaluations. In the same way as the CNF algorithm, the abstraction only builds an implication $a_i \vee enc(\psi_i)$, thus, assumption assignments $\alpha_A$ may not be optimal. Fix some quantifier $\mathcal{Q}X$. During the execution of the algorithm, we maintain the partial evaluation $\beta_\varphi$ of $\varphi$ under the current variable assignment $\alpha_V$ of variables bound at $X$ or at some outer quantifier and we use this evaluation to build optimal assignments. If $\beta_\varphi(\psi_i) = \mathbf{T}$ for some $a_i \in A_X$, then we set $\alpha_{A_X}(a_i)$ to $\mathbf{F}$.

Similar to the optimization described in Section 2.3.3, one can enhance the abstraction by only generating abstraction entries that are satisfiable with respect to the propositional formula, i.e., for every $a_\psi$ we add the constraint that $(a_\psi \to \psi)$.

Lastly, and already noted by other NNF approaches [Jan+16], subformulas $\psi \in sf(\varphi)$ do not need to be in negation normal form if $\psi$ is only influenced by variables of a single quantifier, that is, $\psi = \psi^=$. For example, the following formula $\forall x.\exists y,z.\, x \wedge (y \leftrightarrow z)$ can be solved with the algorithm presented above without modifications.

### 4.1.4 Function Extraction

The overall approach for function extraction algorithm is the same as the one described in Section 2.4. For every quantifier $\exists X$, we store a sequence of pairs $\langle \beta_{S_X}, \alpha_X \rangle \in (\mathcal{A}_\bot(S_X) \times \mathcal{A}(X))$ and these pairs can be obtained from the algorithm by the returned value $\beta_{S_X}$ after the dual abstraction optimization (lines 8 and 10). Next, we define the reverse

function of the abstraction $inv_{QX}\colon \mathcal{A}_\perp(S_X) \to \mathcal{B}(V)$ that maps an assignment $\beta_{S_X}$ to a propositional formula over variables $V$ bound by outer quantifiers (with respect to $X$). Intuitively, $inv_{QX}(\beta_{S_X})$ describes those assignments that lead to $\beta_{S_X}$ in the abstraction of quantifier $QX$. We define $inv_{QX}$ as

$$inv_{\exists X}(\beta_{S_X}) \coloneqq \bigwedge_{s_i \in \beta^1_{S_X}} outer(\psi_i) \tag{4.7}$$

where *outer* is defined as

$$outer(\psi_i) = \begin{cases} \bigwedge\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j^< = \psi_j}} \psi_j & \text{if } type(\psi_i) = \wedge \\ \bigvee\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j^< = \psi_j}} \psi_j & \text{otherwise} \end{cases}$$

→ Page 40

The definition of the extracted function $f_x$ for some $x \in X$ follows then by → Equation 2.11.

**Example 4.12.** We show the function extraction for our running example

$$\exists x.\, \forall v, w.\, \exists y.\, \underbrace{(x \vee v \vee \overbrace{(y \wedge w)}^{\psi_3})}_{\psi_2} \wedge \underbrace{(\overline{x} \vee \overbrace{(v \wedge \overline{w})}^{\psi_5} \vee y)}_{\psi_4} \wedge \underbrace{(\overline{v} \vee w \vee \overline{y})}_{\psi_6}$$

From the execution shown in Example 4.3, we extract the sequences $\langle \varnothing, x \rangle$ and $\langle s_2 s_6, y \rangle \langle s_2 s_4, \overline{y} \rangle$ as described above. The Skolem function for $x$ is the constant $x = \mathbf{T}$. Applying the definition of $inv_{\exists y}$, we get

$$\begin{aligned} inv_{\exists y}(s_2 s_6) &= (x \vee v) \wedge (\overline{v} \vee w) \quad \text{and} \\ inv_{\exists y}(s_2 s_4) &= (x \vee v) \wedge (\overline{x} \vee (v \wedge \overline{w})) \quad . \end{aligned}$$

Thus, the Skolem function $f_y$ is defined as

$$f_y(v, w) = inv_{\exists y}(s_2 s_6)[x \mapsto \mathbf{T}] = (x \vee v) \wedge (\overline{v} \vee w)[x \mapsto \mathbf{T}] = (\overline{v} \vee w) \quad .$$

$f_x$ and $f_y$ depend solely on its dependencies and are functionally correct as $\varphi[f_{\{x,y\}}] = ((v \wedge \overline{w}) \vee \overline{v} \vee w)(\overline{v} \vee w \vee (v \wedge \overline{w}))$ is a tautology.

## 4.2 Evaluation

We implemented the abstraction algorithm for negation normal form formulas in a solver called QuAbS[2] (Quantified Abstraction Solver) that takes as input a quantified Boolean formula encoded in the quantified circuit (QCIR) [QBF14] format. As the solver

[2] Source code available at https://github.com/ltentrup/quabs

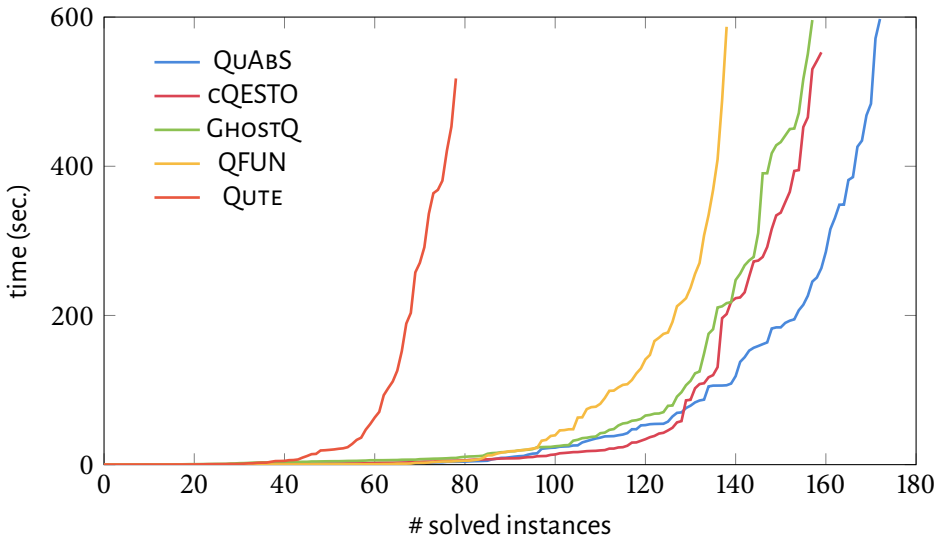



Figure 4.3: Cactus plot showing the number of solved instances on the QBFEVAL'18 benchmark set.

for the propositional abstractions, we used the SAT solver CryptoMiniSat [SNC09] version 5.0.1. We compare QuAbS against the publicly available QBF solvers that support the QCIR format, namely GhostQ [Kli+10] version 2017, QFUN [Jan18b] version 2018, cQESTO [Jan18a] version 2018, and Qute [PSS17] version 1.1. For our experiments, we used a machine with a 3.6 GHz quad-core Intel Xeon (E3-1271 v3) processor and 32 GB of memory. The timeout and memout were set to 10 minutes and 8 GB, respectively. We use the prenex non-CNF benchmark set from the QBF competition QBFEVAL'18. The results are shown in Figure 4.3. Despite being slower initially compared to cQESTO, QuAbS solves more instances overall.

**Function Extraction.** Enabling function extraction outputs a representation of the Skolem and Herbrand function, encoded as And-Inverter-Graph, after determining that the formula is satisfiable and unsatisfiable, respectively. In contrast to CNF solvers, we do not need to disable optimizations [Nie+12] nor preprocessing (as we do not use external preprocessors, and QuAbS uses only constant propagation as preprocessing technique). Thus, the impact of function extraction is small, as shown in Figure 4.4, which compares the running time of QuAbS with and without function extraction.

This makes QuAbS an ideal candidate for applications where solving witnesses are needed: QuAbS is used in the reactive synthesis tool BoSy [FFT17], which won the synthesis track in the reactive synthesis competition (SYNTCOMP) 2016 and 2017 [Jac+16; Jac+17a]. Further, it is also part of the Petri game solver Adam [Fin+17a] and the HyperLTL satisfiability solver MGHyper [FHH18].

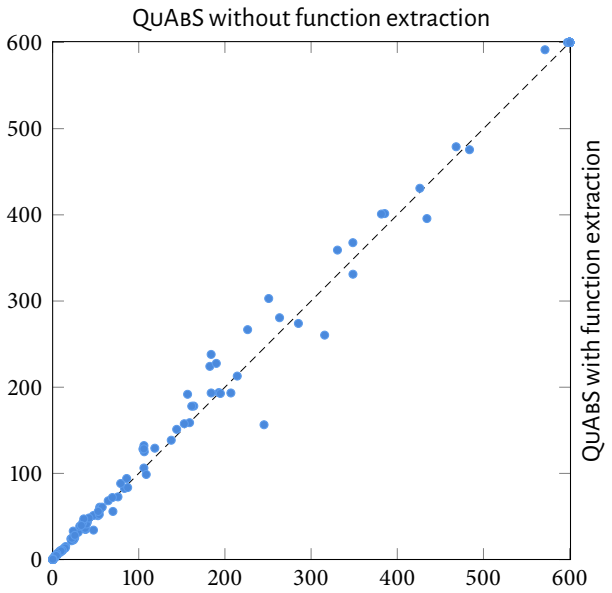



Figure 4.4: Scatter plot comparing the solving time (in sec.) of QuAbS with and without function extraction.

## 4.3 Solving Formulas in Non-Prenex Form

The algorithm presented in Section 4.1 assumes formulas given in prenex normal form where quantifiers are only allowed in the formula's prefix. While every QBF can be converted into prenex form, the task of prenexing a QBF is non-deterministic, and different prenexing strategies lead to different solving times [ESW09]. On the other hand, miniscoping can be used to translate prenex formulas into non-prenex form. We have observed [RT15] that this can be quite effective for splitting instances into independent parts on some benchmark families.

In this section, we extend the circuit abstraction algorithm to handle non-prenex QBFs in negation normal form. In addition to a linear quantifier prefix, the algorithm handles tree-shaped quantifier hierarchies where the quantifiers may appear under Boolean connectives that influence which quantifier needs to be considered for solving. For example, the formula $\exists x, y.\, x \vee \forall u.\, \overline{x} \wedge (y \leftrightarrow u)$, can be solved by only considering the top-level quantifier (using assignment $\{x \mapsto \mathbf{T}\}$). Further, for a branching node, i.e., a quantifier block which has multiple children, it is possible to solve the children independently. Our implementation exploits this independence by solving the different branches in parallel. We show the effectiveness of this approach on the reactive synthesis benchmark set.

**Related Work.** Some of the prior mentioned non-CNF solving approaches can be applied to non-prenex settings as well [ESW09; Kli+10; Jan+16]. There has been prior work on parallelization in QBF solving. PQSolve [FMS00] is an early example of a parallel DPLL

solver. Recent examples include the solver HordeQBF [BL16], that starts multiple instances of the solver DepQBF with different parameters, and MPIDepQBF [Jor+14], that relies on search space partitioning using assumptions. Da Mota et al. [MNS10] proposed methods to split a QBF at the top level and solve the resulting QBF instances in parallel by a sequential CNF algorithm. In contrast, our approach can handle branches at every node in the quantifier hierarchy, and our solving step is tightly integrated into the algorithm. We refer to [LS18] for more details on parallel solving approaches for QBF.

### 4.3.1 Algorithm

In addition to the definitions introduced in Section 4.1.1, we define $dqsf(\varphi) \subset \mathcal{B}$ as the direct quantified subformulas of $\varphi$, i.e., a quantifier $\mathcal{Q}X.\,\psi$ is in $dqsf(\varphi)$ if $\mathcal{Q}X.\,\psi$ is in the scope of $\varphi$ and there is no other quantifier $\mathcal{Q}Y.\,\psi'$ such that $\mathcal{Q}Y.\,\psi'$ is in the scope of $\varphi$ and $\mathcal{Q}X.\,\psi$ is in the scope of $\psi'$. For a QBF $\psi$, we extend the definition of $type(\psi) \in \{lit, \vee, \wedge, \mathcal{Q}\}$ to return the Boolean connector if $\psi$ is not a literal nor a quantifier. For example, given $\psi = \exists x.\,(\forall y.\,\exists z.\,(x \vee y \vee \neg x)) \vee (\forall y.\,(y \wedge x))$, it holds that $type(\psi) = \mathcal{Q}$, $dsf(\psi) = \{\forall y.\,\exists z.\,(x \vee y \vee \neg x)) \vee (\forall y.\,(y \wedge x)\}$, and $dqsf(\psi) = \{\forall y.\,\exists z.\,(x \vee y \vee \neg x), \forall y.\,(y \wedge x)\}$.

For this section, we assume w.l.o.g. that all quantifier blocks in the QBF are strictly alternating, even for quantifiers not in the prefix. That means that for every quantified formula $\mathcal{Q}X.\,\psi$, the quantifier type of all $\psi' \in dqsf(\mathcal{Q}X.\,\psi)$ is $\overline{\mathcal{Q}}$. Further, we assume that for every quantifier $\mathcal{Q}X.\,\psi$ with $type(\psi) = \{\vee, \wedge\}$ it holds that $\mathcal{Q} = \exists$ implies that $type(\psi) = \wedge$ and $\mathcal{Q} = \forall$ implies that $type(\psi) = \exists$. If this is not the case, one can apply the mini-scoping rules.

$$\begin{aligned}
&\forall X.\,\varphi \wedge \psi \equiv \forall X.\,\varphi \wedge \forall X.\,\psi \\
&\exists X.\,\varphi \vee \psi \equiv \exists X.\,\varphi \vee \exists X.\,\psi \\
&\forall X, Y.\,\varphi(X) \vee \psi(Y) \equiv \forall X.\,\varphi \vee \forall Y.\,\psi \\
&\exists X, Y.\,\varphi(X) \wedge \psi(Y) \equiv \exists X.\,\varphi \wedge \exists Y.\,\psi
\end{aligned}$$

The non-prenex algorithm, shown in Algorithm 4.3, is an extension of the prenex algorithm presented in ➔ Algorithm 4.2. The main difference is that due to the non-linear quantifier structure, the algorithm iterates over every quantified subformula. In the case that every recursive verification returns $\text{Sat}_Q$, the algorithm returns $\text{Sat}_Q$ after dual-abstraction optimization. If one of the recursive verifications turn out to be $\text{Unsat}_Q$, the given counterexample is excluded by refining the abstraction in line 17. To translate an assignment of assumption variables $\alpha_{A_X}$ into multiple assignments $\alpha_{S_Y}$, one for each quantifier $\overline{\mathcal{Q}}Y$, the sets $S_Y$ have to partition $A_X$.

➔ Page 73

**Example 4.13.** Given the following non-prenex formula

$$\exists x.\,(x \vee (\forall u.\,\exists z.\,((z \vee u) \wedge (\overline{z} \vee \overline{u}))) \wedge (\overline{x} \vee (\forall v.\,(v \vee (\overline{x} \wedge \overline{v}))) \quad . \tag{4.8}$$

**Algorithm 4.3** Non-prenex Abstraction Based Algorithm

1: **procedure** SOLVE-NNF-NON-PRENEX($\mathcal{Q}X, \Phi, \alpha_{S_X}$)
2: **loop**
3: **match** SAT($\theta_X, \alpha_{S_X}$) **as** ▷ assume outer variable assignment
4: Sat($\alpha$) ⇒
5: **if** $\Phi$ is propositional **then**
6: **return** $\mathsf{Sat}_{\mathcal{Q}}$(OPTIMIZE($\alpha|_X, \alpha_{S_X}$)) ▷ base case
7: **end if**
8: *sub-result* ← $\mathsf{Sat}_Q$
9: $\beta_{A_X} = \{\}$
10: **for** $\psi_i = \overline{\mathcal{Q}}Y.\,\Psi$ in $dqsf(\mathcal{Q}X.\,\Phi)$ **where** $\alpha(a_i) = \mathbf{T}$ **do**
11: $\alpha_{S_Y} \leftarrow \{s_i j \mapsto \alpha(a_j) \mid s_j \in S_Y\}$ ▷ update subformula valuation
12: **match** SOLVE-NNF-NON-PRENEX($\overline{\mathcal{Q}}Y, \Psi, \alpha_{S_Y}$) **as**
13: $\mathsf{Sat}_{\mathcal{Q}}(\beta_{S_Y})$ ⇒
14: $\beta_{A_X} \leftarrow \beta_{A_X} \sqcup \{a_i \mapsto \mathbf{F}\}$
15: $\sqcup \{a_j \mapsto \beta_{S_Y}(s_j) \mid s_j \in \mathrm{dom}(\beta_{S_Y})\}$
16: $\mathsf{Unsat}_{\mathcal{Q}}(\beta_{S_Y})$ ⇒
17: $\theta_X \leftarrow \theta_X \wedge \left(\overline{a_i} \vee \bigvee_{s_i j \in \beta^1_{S_Y}} \overline{a_j}\right)$ ▷ refine $\theta_X$
18: *sub-result* ← $\mathsf{Unsat}_Q$
19: **end for**
20: **if** *sub-result* = $\mathsf{Sat}_Q$ **then**
21: $\overline{\theta}_X \leftarrow \overline{\theta}_X \wedge \bigvee_{s_i \in \beta^0_{A_X}} \overline{a_i}$ ▷ refine $\overline{\theta}_X$
22: **return** $\mathsf{Sat}_{\mathcal{Q}}$(OPTIMIZE($\alpha|_X, \alpha_{S_X}$))
23: **end if**
24: Unsat($\beta_{S_X}$) ⇒ **return** $\mathsf{Unsat}_{\mathcal{Q}}(\beta_{S_X})$
25: **end loop**
26: **end procedure**

The corresponding graph representation and the subformula indices are given in Figure 4.5. Consider the following abstractions

$$\theta_{\{x\}} \coloneqq (a_3 \vee x) \wedge (a_9 \vee \overline{x}) \wedge (a_{12} \vee \overline{x})\ (a_3 \to a_4)(a_9 \to a_{10})$$
$$\theta_{\{u\}} \coloneqq (a_7 \vee \overline{u}) \wedge (a_8 \vee u)\ (a_5)$$
$$\theta_{\{v\}} \coloneqq (\overline{v} \wedge (s_{12} \vee v))$$
$$\theta_{\{z\}} \coloneqq (s_7 \vee z) \wedge (s_8 \vee \overline{z})$$

We give a possible execution of algorithm SOLVE-NNF-NON-PRENEX. To improve readability, we use the propositional representation for assignments.

- SOLVE-NNF-NON-PRENEX($\exists x, \psi_2, \{\}$)
- SAT($\theta_{\{x\}}$) = Sat($x\,\overline{a_3}\overline{a_4}a_9a_{10}a_{12}$)

- $\alpha_{S_{\{v\}}} = s_{12}$
  - SOLVE-NNF$(\forall v, \psi_{11}, \alpha_{S_{\{v\}}})$
  - SAT$(\theta_{\{v\}}, \alpha_{S_{\{v\}}}) = \mathsf{Sat}(\overline{v})$
  - **return** $\mathsf{Sat}_\forall(\textsc{optimize}(\overline{v}, s_{12})) \equiv \mathsf{Unsat}(s_{12})$
- $\theta'_{\{x\}} = \theta_{\{x\}} \wedge (\overline{a_{10}} \vee \overline{a_{12}})$
- SAT$(\theta'_{\{x\}}) = \mathsf{Sat}(\overline{x}\, a_3 a_4 \overline{a_9} \overline{a_{10}} \overline{a_{12}})$
- $\alpha_{S_{\{u\}}} = \{\}$
  - SOLVE-NNF-NON-PRENEX$(\forall u, \psi_5, \alpha_{S_{\{u\}}})$
  - SAT$(\theta_{\{u\}}, \alpha_{S_{\{u\}}}) = \mathsf{Sat}(\overline{u}\, a_5 \overline{a_7} a_8)$
  - $\alpha_{S_{\{z\}}} = \overline{s_7} s_8$
    - SOLVE-NNF-NON-PRENEX$(\exists z, \psi_6, \alpha_{S_{\{z\}}})$
    - SOLVE$(\theta_{\{z\}}, \alpha_{S_{\{y\}}}) = \mathsf{Sat}(z)$
    - **return** $\mathsf{Sat}_\exists(\textsc{optimize}(\overline{z}, \alpha_{S_{\{z\}}})) \equiv \mathsf{Sat}(s_8)$
  - $\theta'_{\{u\}} = \theta_{\{u\}} \wedge (\overline{a_5} \vee \overline{a_8})$
  - SAT$(\theta'_{\{u\}}, \alpha_{S_{\{u\}}}) = \mathsf{Sat}(u\, a_7 \overline{a_8})$
  - $\alpha'_{S_{\{z\}}} = s_7 \overline{s_8}$
    - SOLVE-NNF-NON-PRENEX$(\exists z, \varphi, \alpha'_{S_{\{z\}}})$
    - SOLVE$(\theta_{\{y\}}, \alpha'_{S_{\{y\}}}) = \mathsf{Sat}(\overline{z})$
    - **return** $\mathsf{Sat}_\exists(\textsc{optimize}(z, \alpha'_{S_{\{z\}}})) \equiv \mathsf{Sat}(s_7)$
  - $\theta''_{\{u\}} = \theta'_{\{u\}} \wedge (\overline{a_5} \vee \overline{a_7})$
  - SAT$(\theta''_{\{u\}}, \alpha_{S_{\{u\}}}) = \mathsf{Unsat}(\{\})$
  - **return** $\mathsf{Unsat}_\forall(\{\})$
- SOLVE-NNF-NON-PRENEX$(\exists x, \psi_2, \{\})$ returns Sat

**Abstraction $\theta$.** We adapt the construction of the abstraction $\theta$ from Section 4.3 to the non-prenex setting. We begin by identifying the differences between prenex and non-prenex solving and how it affects the constructions of the abstraction. A non-prenex quantifier introduces a sub-game, thus, for such a quantifier $\mathcal{Q}X.\,\psi$, we root the abstraction $\theta_X$ at formula $\psi$. At some quantifier $\mathcal{Q}X.\,\psi$, the respective player has the choice to *schedule* a subset of the quantified subformulas $dqsf(\psi)$ corresponding to player $\overline{\mathcal{Q}}$. For example, in the formula

$$\exists a.\Big((\forall x.\,[\ldots]) \wedge (\forall y.\,[\ldots] \vee \forall z.\,[\ldots])\Big)$$

the existential player has the choice to schedule either $\forall y$ or $\forall z$, but has to schedule $\forall x$.

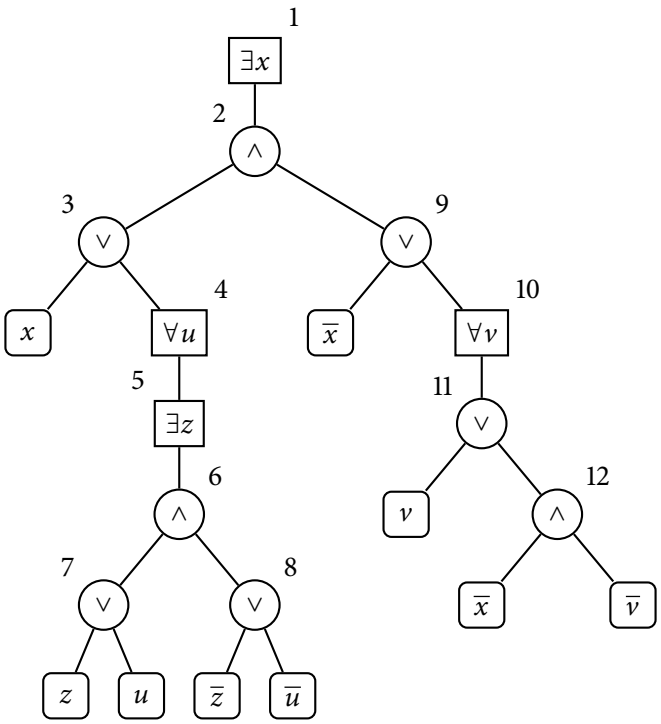



Figure 4.5: Visualization of the graph representation $\mathcal{G}_\varphi$ representation of $\varphi = \exists x.\,(x \vee (\forall u.\,\exists z.\,((z \vee u) \wedge (\overline{z} \vee \overline{u}))) \wedge (\overline{x} \vee (\forall v.\,(v \vee x)))$. The numbers on the non-terminal nodes represent the index $i$ for the corresponding subformula $\psi_i$. To improve readability, terminal nodes are drawn multiple times.

We implement those changes in the following. Fix some existential quantifier $\exists X.\,\psi$ and some subformula $\psi_i$ of $\psi$. The abstraction $\theta_X$ is defined as

$$\theta_X = \begin{cases} enc(\psi) & \text{if } \exists X \text{ is an innermost quantifier} \\ \bigwedge\limits_{\substack{\psi_i \in sf(\psi) \\ \exists \psi_j, \psi_k \in dsf(\psi_i) \text{ with } \psi_j = \psi_j^{\leq} \wedge \psi_k \neq \psi_k^{\leq}}} a_i \vee enc(\psi_i) & \text{otherwise} \end{cases} \tag{4.9}$$

Compared to the non-prenex definition in Equation 4.2, we change the root formula to be $\psi$ instead of $\varphi$ and there is the possibility to have multiple innermost quantifiers. The abstraction for $\psi_i$ is defined as

$$enc(\psi_i) = \begin{cases} \bigwedge\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j = \psi_j^{\leq}}} enc_{\psi_i}(\psi_j) & \text{if } type(\psi_i) = \wedge \\ \bigvee\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j = \psi_j^{\leq}}} enc_{\psi_i}(\psi_j) & \text{if } type(\psi_i) = \vee \\ enc(\psi_j) & \text{if } \psi_i = \mathcal{Q} Y.\,\psi_j \end{cases} \tag{4.10}$$

that is, for some inner quantifier $\mathcal{Q}Y.\,\psi_j$, we ignore the quantifier and directly continue building the abstraction for $\psi_j$. The encoding of subformulas $enc_{\psi_i}(\psi_j)$ stays the same (Equation 4.4).

Lastly, we have to encode the scheduling constraints for direct quantified subformulas $\overline{\mathcal{Q}}Y.\,\psi' \in dsf(\mathcal{Q}X.\,\psi)$. Thus, for every such formula $\psi_i \in dsf(\mathcal{Q}X.\,\psi)$, we search in the graph representation the earliest node $\psi_k$ on the path from $\psi_i$ to the root $\psi$ such that $type(\psi_k) \in \{\vee, \wedge\}$ and $a_k \in A_X$. If there is such a node, we add the constraint $(a_k \rightarrow enc_{\mathcal{Q}}(\psi_k))$ to $\theta_X$ and $(a_i)$ otherwise. $enc_{\mathcal{Q}}$ is closely related to $enc$, but only collects quantified formulas, that is,

$$enc_{\mathcal{Q}}(\psi_i) = \begin{cases} \bigwedge\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j = \psi_j^{\leq} \wedge \psi_j \neq \psi_j^{<} \wedge \psi_j \neq \psi_j^{=}}} enc_{\mathcal{Q}}(\psi_j) & \text{if } type(\psi_i) = \wedge \\ \bigvee\limits_{\substack{\psi_j \in dsf(\psi_i) \\ \psi_j = \psi_j^{\leq} \wedge \psi_j \neq \psi_j^{<} \wedge \psi_j \neq \psi_j^{=}}} enc_{\mathcal{Q}}(\psi_j) & \text{if } type(\psi_i) = \vee \\ a_j & \text{if } \psi_i = \mathcal{Q}Y.\,\psi_j \wedge \psi_i \in dqsf(\psi) \end{cases} \tag{4.11}$$

This gives the intended scheduling semantics: For a conjunction, all conjuncts with quantifier have to be scheduled while for a disjunction, only one of them are required.

### 4.3.2 Case Study: Parallelizing Bounded Synthesis

For our case study, we consider the *reactive synthesis* problem, i.e., the problem of synthesizing a finite-state controller from an $\omega$-regular specification. The details of the QBF encoding of this problem using the *bounded synthesis* approach [FS13] are given in → Section 7.3.2, thus, we only give a high level overview. The QBF query has a quantifier prefix of the form $\exists\forall\exists$. The variables in the top level existential correspond to a global constraint that cannot be split syntactically. However, the constraints regarding the inner quantifiers $\forall\exists$ are local to the state of the implementation, so one can derive a QBF with a top level existentially quantifier and $n$ independent $\forall\exists$ quantifiers below by using miniscoping rules, where $n$ is the number of states of the to be synthesized controller. This is merely a new observation and not particularly special for this kind of benchmark as we have made similar observations regarding competitive benchmark suites for CNF [RT15].

→ Page 147

We implemented Algorithm 4.3 in a prototype tool called pQuAbS (Parallel Quantified Abstraction Solver)[3] that accepts as input a QBF in the standard format QCIR [QBF14]. We use PicoSAT [Bie08] as the underlying SAT solver and the POSIX pthreads library for thread creation and synchronization. For every quantifier $\mathcal{Q}X$ that branches more than once, we create a thread for each child quantifier. The loop in line 10 is then implemented by passing $\alpha_{S_Y}$ to the subquantifier followed by notifying the corresponding thread. Before line 20, there is a barrier where we wait for all children to finish. For our experiments, we used a machine with a 3.6 GHz quad-core Intel Xeon (E3-1271 v3) processor and 32 GB of memory. The timeout was set to 10 minutes. The synthesis instances used in this case study where taken from the Acacia benchmark set [Boh+12].

[3] Available at https://www.react.uni-saarland.de/tools/quabs/

Table 4.1: Cumulated solving time of pQuAbS with respect to number of used threads. There are 443 instances in total.

| | 1 thread | 2 threads | 3 threads | 4 threads | prenex |
|---|---|---|---|---|---|
| # solved instances | 397 | 403 | 407 | 409 | 325 |
| cumulated solving time | 100% | 64.51% | 54.15% | 49.94% | - |

Table 4.2: Detailed solving results for example instances.

| instance | branching | 1 thread | 2 threads | 3 threads | 4 threads |
|---|---|---|---|---|---|
| ltl2dba-23 | 10 | 598.20 s<br>100% | 393.68 s<br>65.81% | 335.59 s<br>56.10% | 312.70 s<br>52.27% |
| ltl2dpa-12 | 15 | 521.35 s<br>100% | 302.13 s<br>57.95% | 233.98 s<br>44.88% | 202.27 s<br>38.80% |
| ltl2dba-05 | 4 | 476.12 s<br>100% | 359.40 s<br>75.49% | 331.87 s<br>69.70% | 322.59 s<br>67.75% |
| load-full-6 | 3 | 386.94 s<br>100% | 332.15 s<br>85.84% | 314.37 s<br>81.25% | 321.75 s<br>83.15% |
| ltl2dpa-11 | 18 | 252.61 s<br>100% | 143.13 s<br>56.67% | 107.54 s<br>42.57% | 92.43 s<br>36.59% |

Table 4.1 shows the overall results of our experiments. It depicts the number of solved instances and the cumulated solving times with respect to the number of threads used. For comparison, we also included the number of solved instances from the single threaded version of pQuAbS without miniscoping, i.e., linear prenex solving. One cannot expect linear speedup due to the non-parallelizable parts, like preprocessing and solving of the top-level existential quantifier, as well as the fact that the solving time of the children $\forall\exists$ quantifiers are not uniform.

Nevertheless, already using 2 threads, the speedup compared to single thread solving is more than 1.5 and using 4 threads reduces the solving time by a factor of 2 on average. Table 4.2 gives detailed results for select instances from the scatter plot of Figure 4.6. These examples are the two "outliers" *load-full-6* and *ltl2dba-05*, the hardest commonly solved instance *ltl2dba-23*, and two instances with close to optimal speedup (*ltl2dpa-12* and *ltl2dpa-11*).

## 4.4 Summary

In this section, we extended the clausal abstraction approach in two dimensions, lifting the CNF and prenex prerequisites assumed in Chapter 2. First, we considered quantified Boolean formulas in prenex negation normal form. We presented the new abstraction and algorithm in Section 4.1, where we also showed the correctness, algorithmic im-

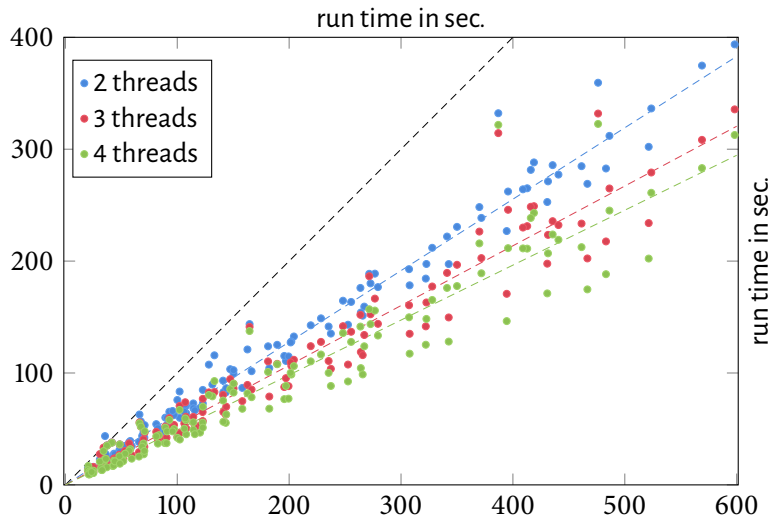



Figure 4.6: Scatter plot of solving times with multiple threads against single thread baseline. Here, we consider only instances with more than 1 second of solving time.

provements, and function extraction. Notably, we introduced the notion of a dual abstraction, which is a key factor in the algorithm. Our experiments show that the resulting algorithm performs better than competing solvers on the QBFEVAL'18 non-CNF benchmark set. Afterward, in Section 4.3, we lifted the prenex requirement by considering tree-shaped quantifier prefixes and showed how to exploit independence by parallelizing parts of the algorithm. Tree-shaped quantifiers can also be seen as the first step towards non-linear quantifiers as introduced by *branching quantifiers*, which we discuss in the following chapters.

# Chapter 5

# Fast DQBF Refutation

The synthesis problem, that is the search for an implementation given declarative specifications, is considered to be an extremely hard algorithmic problem. Depending on the underlying model, one common way to solve this problem is an encoding to a constraint problem asserting the existence of Boolean functions. For example, the synthesis of invariants, programs, or winning regions of (finite) games can all be expressed as the existence of a function $f\colon \mathbb{B}^m \to \mathbb{B}^n$ such that for all tuples of inputs $\vec{x}_1, \ldots, \vec{x}_k \in \mathbb{B}^m$ some relation $\varphi(\vec{x}_1, f(\vec{x}_1), \ldots, \vec{x}_k, f(\vec{x}_k))$ over function applications of $f$ is satisfied. While it is possible to specify these problems in SMT or in first-order logic, existing algorithms struggle to solve even simple instances of synthesis queries.

In order to develop a new algorithmic approach for synthesis problems, we focus on the simplest logic admitting the existential quantification over Boolean functions, dependency quantified Boolean formulas (DQBF). DQBF is an extension of QBF, which allows for non-linear dependencies between quantified variables. However, existing algorithms for DQBF perform poorly, in particular, on synthesis problems [Fay+17]. This is not surprising: Typical synthesis queries contain two or more function applications, i.e. are of the form $\exists f.\, \forall \vec{x}_1, \vec{x}_2.\, \varphi(\vec{x}_1, f(\vec{x}_1), \vec{x}_2, f(\vec{x}_2))$, and involve bit-vector variables, e.g. $\vec{x}_1, \vec{x}_2 \in \mathbb{B}^n$. The so far best-performing algorithm for DQBF needs to expand either $\vec{x}_1$ or $\vec{x}_2$ in order to reach a linear quantifier prefix, which can then be converted to a QBF [Git+15]. This means that they reduce to QBF formulas that are exponential in $n$.

Non-linear dependencies also occur naturally in verification problems for incomplete designs, such as the *partial equivalence checking* (PEC) problem [Git+13], where a partial circuit, with some parts left open as "black boxes", is compared against a reference circuit. In this chapter, we focus on the PEC problem as it is itself a *synthesis problem* ("does there exists an implementation of the black boxes such that the circuit becomes equivalent to its reference implementation?") and the encoding to DQBF is straightforward. The inputs to the circuit are modeled as universally quantified variables and the outputs of the black boxes as existentially quantified variables. Since the output of a black box should only depend on the inputs that are actually visible to the black box, we need to restrict the dependencies of the existentially quantified variables to subsets of the universally quantified variables.

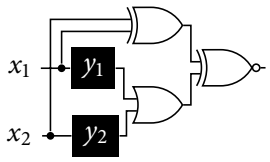



Figure 5.1: Example of a partial equivalence checking (PEC) problem. A partial design, consisting here of the two black boxes and the OR gate, is compared to the reference circuit above, here consisting of a single XOR gate. The output of the complete circuit is 1 iff the completion of the partial design and the reference circuit compute the same result.

There has been some success in extending standard techniques of QBF solving to DQBF [FKB12; Git+13; Frö+14; Git+15], including an extension of the clausal abstraction algorithm to DQBF which we present in Chapter 6. A much faster alternative to such precise methods is to *approximate* the dependencies, such that all dependencies become linear and DQBF thus simplify to QBF. For the PEC problem, an overapproximation of the dependencies is still useful to find errors (if the black box cannot be implemented with additional inputs, then it can, for sure, not be implemented according to the original design), but it significantly decreases the accuracy, because errors that result precisely from the incomparable dependencies of the black boxes are no longer detected. Consider, for example, the toy PEC problem shown in Figure 5.1, where we ask whether it is possible to implement the XOR gate at the top as an OR of the two black boxes below, which *each* only sees *one* of the two inputs $x_1$ and $x_2$. This is obviously not possible; however, the three overapproximating linearizations $\forall x_1 \forall x_2 \exists y_1 \exists y_2$, $\forall x_1 \exists y_1 \forall x_2 \exists y_2$ and $\forall x_1 \exists y_2 \forall x_2 \exists y_1$ all result in a positive answer, because an output that depends on both $x_1$ and $x_2$ can compute $x_1 \oplus x_2$, which gives the correct result, assuming that the other black box simply outputs constant 0.

In this chapter, we present an algorithm for DQBF that combines the efficiency of the QBF abstraction with the accuracy of the classic methods [FKB12; Git+13; Frö+14; Git+15]. We focus on the *refutation* of DQBF because this corresponds to the identification of errors in the PEC problem, or more general, inconsistencies in the specification of a synthesis problem. In contrast to QBF, where every false formula has a countermodel represented by a Herbrand function, this is no longer the case for DQBF [BCJ14a]. Another view on the QBF abstractions mentioned above is that they under-approximate the existence of a Herbrand countermodel. For example, the linear quantifier prefix $\forall x_1 \forall x_2 \exists y_1 \exists y_2$ represents the search for constant Herband functions $f_{x_1}$ and $f_{x_2}$. Our algorithm identifies situations in which *a set of Herbrand countermodels* is sufficient to rule out a satisfying assignment of the existentially quantified variables. In the PEC example from Figure 5.1, there are four possible constant Herbrand functions represented by the assignment $x_1x_2$, $x_1\overline{x_2}$, $\overline{x_1}x_2$, and $\overline{x_1}\overline{x_2}$. However, already three of them[1], $x_1x_2$, $x_1\overline{x_2}$, and $\overline{x_1}\overline{x_2}$, suffice to rule out a satisfying assignment for the existential variables: Since $y_1$ does not depend on $x_2$, its value must be the same for $x_1x_2$ and $x_1\overline{x_2}$; likewise, the value of $y_2$ must be the same for $x_1\overline{x_2}$ and $\overline{x_1}\overline{x_2}$. For $x_1x_2$, both $y_1$ and $y_2$ must be 0, because $1 \oplus 1 = 0$. However, if $y_1 = 0$ for $x_1x_2$, then $y_1 = 0$ also for $x_1\overline{x_2}$, which leads to a contradiction, because $y_2$ must be equal

[1] Using a better under-approximation reduces the number of required Herbrand functions in this example to just two.

to 1 for $x_1\overline{x_2}$ because $1 \oplus 0 = 1$ and, at the same time, equal to 0, because $0 \oplus 0 = 0$. In our algorithm, we specify the existence of such a set of Herbrand functions as a QBF formula. We iteratively increase the number of countermodels to be considered and terminate as soon as a satisfying assignment is ruled out.

This chapter is based on work published in the proceedings of SAT [FT14b].

**Related Work.** DQBF was first defined by Peterson and Reif [PR79] and gained more attention recently [BCJ12; Git+13; Frö+14; BCJ14a; Git+15; Wim+16; Rab17; Bey+18; TR19b]. The satisfiability problem for DQBF is NEXPTIME-complete [PRA01]. The first investigation of practical methods for DQBF solving is a DPLL-based approach due to Fröhlich et al. [FKB12]. The solver IDQ [Frö+14] uses an instantiation-based algorithm, which is based on the Inst-Gen calculus, a state-of-the-art decision procedure for the effectively propositional fragment of first-order logic (EPR), for which the satisfiability problem is also NEXPTIME-complete. HQS [Git+15] is an expansion based solver that expands universal variables until the resulting instance has a linear prefix and applies QBF solving afterward. The idea of partial expansions was used in previous work, e.g., there is a two-phase proof system for QBF that expands certain paths and then refutes the formula by propositional resolution [JM13]. The underlying proof system ∀Exp+Res is sound and complete for DQBF [Bey+18]. The verification of incompletely specified circuits has received significant attention (cf. [SB01; NSB07]); the connection between DQBF and the PEC problem was first pointed out by Gitina et al. [Git+13]. On a more general level, the verification of partial designs is related to the synthesis problems for reactive systems with incomplete information and for distributed systems (cf. [KV97; FS05]). In previous work, we have proposed an efficient method for disproving the existence of distributed realizations of specifications given in linear-time temporal logic (LTL) [FT14a] that bounds, similar to the approach of this chapter, the number of countermodels under consideration.

## 5.1 Dependency Quantified Boolean Formulas

Let $\mathcal{V}$ be a finite set of propositional variables. We use the convention to denote the set of all universal variables $\mathcal{X}$, an element $x \in \mathcal{X}$, and a subset $X \subseteq \mathcal{X}$ ($y \in Y \subseteq \mathcal{Y}$ for existential variables, respectively). The standard form of a DEPENDENCY QUANTIFIED BOOLEAN FORMULA (DQBF) (also called Skolem form [BCJ14a]) is

$$\forall x_1. \cdots \forall x_n. \exists y_1(H_1). \cdots \exists y_m(H_m). \varphi \ , \tag{5.1}$$

that is, formulas beginning with universal quantified variables followed by the existentially quantified *Henkin quantifiers* and the quantifier-free formula $\varphi$. A Henkin quantifier $\exists y(H)$ explicitly states the dependency for variable $y$ by its support set $H \subseteq \mathcal{X}$, which is the difference to QBF, where the preceding universal quantification determines the dependencies of an existential variable. For the quantifier-free part $\varphi$ we allow negation $\neg$, disjunction $\vee$, conjunction $\wedge$, implication $\rightarrow$, equivalence $\leftrightarrow$, exclusive or $\oplus$, and the abbreviations *true* $\mathbf{T}$ and *false* $\mathbf{F}$.

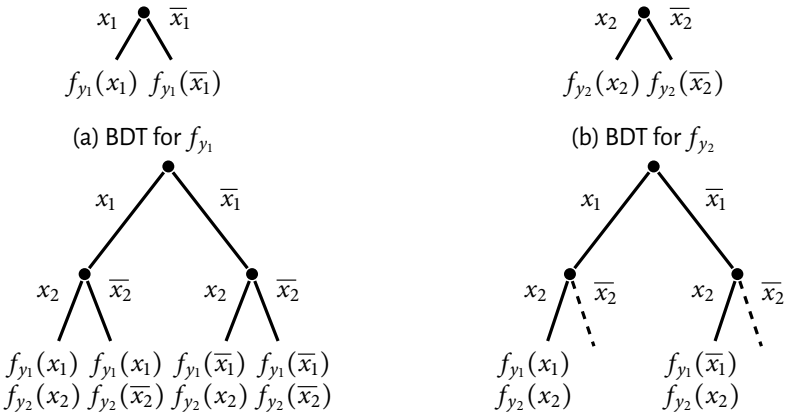
A DQBF formula $\Phi$ is satisfiable, if there exists a *Skolem function* $f_y$ for each existential variable $y \in \mathcal{Y}$, such that for all possible assignments of the universal variables $\mathcal{X}$, the Skolem functions evaluated on these assignments satisfy $\varphi$. We represent a function $f_y$ as a BINARY DECISION TREE (BDT) , where the branching of the tree represents the assignment of universal variables and $f_y$ serves as the labeling function for the leaves, see Figure 5.2a–c for examples of BDTs. As a notation for assignments, we use the cube representation defined before (see also Section 2.1). A root-to-leaf path of a BDT represents an assignment $\alpha_X$ where $X \subseteq \mathcal{X}$ is the set of universal variables that the BDT branches on. A MODEL $\mathcal{M}$ of a satisfiable DQBF formula $\Phi$ is a binary decision tree with branching $\mathcal{X}$ and leaf labeling $f_{\mathcal{Y}}$ such that (1) for every assignment $\alpha_{\mathcal{X}}$ represented by a root-to-leaf path in the tree it holds that $\alpha_{\mathcal{X}} \dot\cup f_{\mathcal{Y}}(\alpha_X) \vDash \varphi$ and (2) the labels of the leaves are *consistent* according to the dependencies of the existential variables, i.e., there exists a decomposition of the decision tree into individual Skolem functions $f_y$ for each $y \in \mathcal{Y}$. For example, the Skolem functions of a satisfiable DQBF formula

$$\forall x_1, x_2 . \exists y_1(x_1) . \exists y_2(x_2) . \varphi \tag{5.2}$$

are the Boolean functions $f_{y_1}$ and $f_{y_2}$, depicted in Figure 5.2a and b, respectively. Figure 5.2c shows the corresponding model, that is the composition of $f_{y_1}$ and $f_{y_2}$. In this representation, the incomparable dependencies become visible: Despite the branching of the tree by both variables $x_1$ and $x_2$, the results of the Skolem functions $f_{y_1}$ and $f_{y_2}$ must be equal on paths that cannot be distinguished according to the dependencies, e.g., as $y_2$ does not depend on $x_1$, the paths $x_1x_2$ and $\overline{x_1}x_2$ are indistinguishable for $f_{y_2}$ and the result on both paths is $f_{y_2}(x_2)$. A CANDIDATE MODEL of a DQBF is a BDT over all existential variables such that all existential assignments are *consistent*. A DQBF formula is *unsatisfiable* if there does not exist a model, i.e., for all *candidate models* always at least one path violates the quantifier-free formula $\varphi$.

**QBF Approximation.** Given a DQBF formula $\Phi$, a QBF formula $\Psi$ with the same quantifier-free part is an approximation of $\Phi$, written $\Phi \preceq \Psi$ if for all existential variables $y \in \mathcal{Y}$ it holds that $H_y \subseteq dep_\Psi(y)$, where $H_y$ is the support set of $y$ and $dep_\Psi(y) \subseteq \mathcal{X}$ is the dependency set of $y$ in the QBF formula $\Psi$. Given two QBF approximations $\Psi$ and $\Psi'$, we call $\Psi$ *stronger* than $\Psi'$, written $\Psi \preceq \Psi'$, if for all $y \in \mathcal{Y}$ it holds that $dep_\Psi(y) \subseteq dep_{\Psi'}(y)$ [Git+13]. In Equation 5.2, $y_1$ and $y_2$ have *incomparable* dependencies as neither $\{x_1\} \subseteq \{x_2\}$ nor $\{x_2\} \subseteq \{x_1\}$. Hence, in all strongest QBF approximations, that is $\forall x_1 \exists y_1 \forall x_2 \exists y_2$ and $\forall x_2 \exists y_2 \forall x_1 \exists y_1$, at least one existential variable has more dependencies than before. The resulting inaccuracy was already highlighted in the introduction on the PEC problem from Figure 5.1, which corresponds to the formula given in Equation 5.2 with quantifier-free part $\varphi = (y_1 \vee y_2) \leftrightarrow (x_1 \oplus x_2)$. All QBF abstractions of Equation 5.2 are satisfiable despite the DQBF formula being unsatisfiable.

**Variable Elimination.** The expansion based method for converting a DQBF formula $\Phi$ into a logically equivalent QBF formula $\Psi$ [BK06; BCJ14a] uses the idea of unrolling the binary decision tree, e.g., expanding the formula given in Equation 5.2 by $x_1$ results in the




(a) BDT for $f_{y_1}$

(b) BDT for $f_{y_2}$

(c) BDT representing the combination of $f_{y_1}$ and $f_{y_2}$

(d) Partial BDT for $f_{y_1}$ and $f_{y_2}$

Figure 5.2: The figure shows the binary decision trees for $f_{y_1}$ (a) and $f_{y_2}$ (b), their composition (c), and a partial model (d) for the PEC problem in Figure 5.1.

formula

$$\forall x_2.\, \exists y_2(x_2).\, \exists y_1().\, \exists y_1'().\, \varphi[x_1 \mapsto \mathbf{F}] \wedge \varphi'[x_1 \mapsto \mathbf{T}] \ , \tag{5.3}$$

where $\varphi'$ is the formula obtained from $\varphi$ by replacing all occurrences of $y_1$ by $y_1'$. In the expansion of $x_1$, only variable $y_1$ is duplicated to represent the different choices of the Skolem function $f_{y_1}$ on the paths that differ in the assignment of $x_1$. As $y_2$ does not depend on $x_1$, the variable $y_2$ is not duplicated. After the expansion of all universal variables, the resulting existential QBF formula can be solved by a SAT solver. Instead of expanding all quantifier, one can also expand universal variables until the resulting quantifier prefix is linear [Git+15] from which point on one can use a QBF solver.

## 5.2 Bounded Unsatisfiability

**Partial Models.** Instead of enumerating all constant Herbrand functions, which corresponds to expanding the whole binary decision tree, it is often possible to determine unsatisfiability with only a subset of the assignments to the universal variables. The restricted choice of a Skolem function based on its dependencies can also be defined as a *consistency condition* between universal assignments: Given two assignments $\alpha_{\mathcal{X}}, \alpha'_{\mathcal{X}} \in \mathcal{A}(\mathcal{X})$, different assignments of an existential variable $y \in \mathcal{Y}$ on $\alpha_{\mathcal{X}}$ and $\alpha'_{\mathcal{X}}$ are *consistent* if the assignments of $H_y$ on $\alpha_{\mathcal{X}}$ and $\alpha'_{\mathcal{X}}$ are different. We introduce the notion of partial models that are decision trees which contain only a subset of all universal assignments. Formally, a PARTIAL MODEL $\mathcal{P}$ of a DQBF formula $\Phi$ is a model over universal variables $\mathcal{X}$ were branches may be omitted. A PARTIAL CANDIDATE MODEL is defined analogously. As partial models are weaker than models, the existence of a partial model does not imply the existence of a model, but from the non-existence of a partial model follows the non-existence of a model.

**Lemma 5.1.** *Given a DQBF formula $\Phi$ and a set of assignments $P \subseteq \mathcal{A}(\mathcal{X})$. $\Phi$ is unsatisfiable if there does not exist a partial model over $P$.*

*Proof.* We only consider the case $P \neq \mathcal{A}(\mathcal{X})$, as for $P = \mathcal{A}(\mathcal{X})$ the definition of partial models and models coincide. Assume for contradiction that there does not exist a partial model over $P$ while $\Phi$ is satisfiable, i.e., there exists a model $\mathcal{M}$ of $\Phi$. We define a candidate partial model $\mathcal{P}$ by restricting $\mathcal{M}$ to the assignments $P$. From the definition of models, it follows that all assignments in $\mathcal{P}$ satisfy $\varphi$ and the labeling of the leaves are consistent on all pairs of assignments in $\mathcal{P}$. Hence, $\mathcal{P}$ is a partial model of $\Phi$, contradicting our assumption. □

**Bounded Unsatisfiability.** We turn the idea of non-existing partial models into the bounded unsatisfiability problem that limits the number of assignments under consideration to show that no partial model exists. For some $k \geq 1$, a DQBF formula $\Phi$ is *$k$-bounded unsatisfiable* if there exists a set of assignments $P \subseteq \mathcal{A}(\mathcal{X})$ with $|P| \leq k$ such that there does not exist a partial model over $P$.

**Theorem 5.2.** *A DQBF formula $\Phi$ is unsatisfiable iff it is $k$-bounded unsatisfiable for some $k \geq 1$.*

*Proof.* If a DQBF formula $\Phi$ is unsatisfiable, it follows immediately that it is $2^{|\mathcal{X}|}$-bounded unsatisfiable as $2^{|\mathcal{X}|}$ is the total number of assignments over $\mathcal{X}$. Assume that a DQBF formula $\Phi$ is $k$-bounded unsatisfiable, then there exists a set of assignments $P$ with $|P| \leq k$ such that $\Phi$ is unsatisfiable by Lemma 5.1. □

## 5.3 Encoding of Bounded Unsatisfiability in QBF

We give an encoding of the $k$-bounded unsatisfiability problem to QBF for a fixed bound $k \geq 1$. Before presenting the general encoding, we show the basic steps on the formula given in Equation 5.2 $\forall x_1, x_2.\, \exists y_1(x_1).\, \exists y_2(x_2).\, \varphi$. The formula is unsatisfiable iff for all candidate models, there exists at least one assignment to the universal such that the propositional formula $\varphi$ is unsatisfiable. Instead of expanding all four universal assignments, we restrict the binary decision tree to just two (but do not choose which one) and encode the search for the concrete assignment as QBF formula

$$\exists x_1^1, x_1^2, x_2^1, x_2^2.\, \forall y_1^1, y_1^2.\, \forall y_2^1, y_2^2.\, \neg\varphi^1 \vee \neg\varphi^2 \tag{5.4}$$

that asserts that either assignment violates $\varphi$. This, however, does not accurately represent the incomparable dependencies of $y_1$ and $y_2$. For the assignment depicted in Figure 5.2d, where only $x_1$ has a different assignment on the two assignments, $y_2^1$ and $y_2^2$ can have different assignment as well, despite the fact that $y_2$ does not depend on $x_1$. To fix this inaccuracy, we introduce a *consistency condition* that ensures the restricted choices across multiple universal assignments. For example, the consistency condition for $y_2$ in Equation 5.4 is $(y_2^1 \leftrightarrow y_2^2) \vee (x_2^1 \nleftrightarrow x_2^2)$, i.e., either the assignment of $y_2$ is equal on both assignments, or the assignment of the dependency $x_2$ is different. In the following, we describe the general encoding.

We build a QBF formula $bunsat(\Phi, k)$ that encodes the $k$-bounded unsatisfiability problem, i.e., for a given bound $k$, the satisfaction of $bunsat(\Phi, k)$ implies that $\Phi$ is unsatisfiable. In the encoding, we introduce $k$ copies of the existential and universal variables in the DQBF formula $\Phi$. Moreover, we specify a consistency condition that enforces that the universal variables can only act according to the assignment of the dependencies given by the support sets.

$$\begin{aligned} bunsat(\Phi, k) \coloneqq \exists & x_1^1, \ldots, x_m^1, x_1^2, \ldots, x_m^2, \ldots, x_m^k. \\ \forall & y_1^1, \ldots, y_n^1, y_1^2, \ldots, y_n^2, \ldots, y_n^k. \\ & consistent(\{y_1, \ldots, y_n\}, k) \rightarrow \bigvee_{1 \leq i \leq k} \neg \varphi^k \ , \end{aligned} \tag{5.5}$$

where $\varphi^k$ denotes the formula $\varphi$ for which every variable $v$ is replaced by $v^k$. The consistency condition is given by the formula

$$consistent(Y, k) \coloneqq \bigwedge_{y \in Y} \bigwedge_{(i,j) \in \{1,\ldots,k\}^2} \left( (y^i \leftrightarrow y^j) \vee \left( \bigvee_{x \in H_y} x^i \nleftrightarrow x^j \right) \right) . \tag{5.6}$$

**Lemma 5.3.** *Let $\Phi$ be a DQBF formula, $1 \leq k \leq 2^{|\mathcal{X}|}$ a bound, and let $P \subseteq 2^{\mathcal{X}}$ with $|P| = k$ be a set of assignments. A binary decision tree $\mathcal{P}$ over assignments $P$ is a candidate partial model if, and only if, $\mathcal{P}$ satisfies $consistent(\mathcal{Y}, k)$.*

*Proof.* Assume that $\mathcal{P}$ is a candidate partial model. Pick an arbitrary existential variable $y \in \mathcal{Y}$ with support set $H_y \subseteq \mathcal{X}$. Pick two different assignments $\alpha_i$ and $\alpha_j$ from $P$ (if $|P| = 1$, $consistent(\mathcal{Y}, k)$ is trivially satisfied). To be consistent, the valuation of $y$ on both assignments must be equal, or $\alpha_i|_{H_y} \neq \alpha_j|_{H_y}$. Hence $consistent(\mathcal{Y}, k)$ is satisfied.

Let $\mathcal{P}$ be an binary decision tree over $P$ such that the assignments of the existential variables $\mathcal{Y}$ satisfy $consistent(\mathcal{Y}, k)$. Let $y \in \mathcal{Y}$ be an arbitrary existential variable. Pick any combination $(i, j) \in \{1, \ldots, k\}^2$ of assignments. By the satisfaction of $consistent(\mathcal{Y}, k)$, it follows that either the valuation of $y$ is equal on both assignments, or the valuation of a variable from the support set $H_y$ is different. Thus, $\mathcal{P}$ is a candidate partial model. □

**Theorem 5.4.** *A DQBF formula $\Phi$ is unsatisfiable if, and only if, there exists a bound $k \geq 1$ such that the QBF formula $bunsat(\Phi, k)$ is satisfiable.*

*Proof.* Assume $\Phi$ is unsatisfiable, i.e., for all candidate models, there exists an assignment such that the propositional formula is unsatisfiable. Choose a bound $k$ of $2^{|\mathcal{X}|}$, i.e., we choose to expand the whole decision tree. In this case, the consistency condition models exactly the dependencies of the existential variables (Lemma 5.3). Thus, $bunsat(\Phi, k)$ considers every candidate model and from the unsatisfiability of $\Phi$ follows that for each such candidate, there exists an assignment that satisfies $\neg\varphi$, hence, $bunsat(\Phi, k)$ is satisfiable.

Assume there exists a $k \geq 1$ such that $bunsat(\Phi, k)$ is satisfiable. We choose the assignments $P \subseteq 2^{\mathcal{X}}$ from the assignment of the existential variables in $bunsat(\Phi, k)$. By

Lemma 5.3, the consistency condition allows for all candidate partial models and, for each candidate, one of the assignments satisfies $\neg\varphi$. Hence, it follows that $\Phi$ is $k$-bounded unsatisfiable and, by Theorem 5.2, that $\Phi$ is unsatisfiable. □

**Proposition 5.5.** *Let $\Phi$ be a DQBF formula. For some bound $k \geq 1$, the QBF formula bunsat$(\Phi, k)$ has $k \cdot |\mathcal{X}|$ existential and $k \cdot |\mathcal{Y}|$ universal variables, respectively, and the propositional part is of size $\mathcal{O}(|\mathcal{Y}| \cdot k^2 \cdot \max_{y \in \mathcal{Y}} |H_y| + k \cdot |\varphi|)$.*

*Proof.* The number of variables and the size of the unrolling of $\varphi$ follow directly from the definition given in Equation 5.5. It remains to be shown that the size of the consistency condition is in $\mathcal{O}(|\mathcal{Y}| \cdot k^2 \cdot \max_{y \in \mathcal{Y}} |H_y|$. Omitting symmetric cases in the consistency condition gives us $\binom{k}{2} \in \mathcal{O}(k^2)$ different sets $(i, j) \in \{1, \ldots, k\}^2$ and the remaining part follows from the definition given in Equation 5.6. □

**Combining with QBF Abstraction.** One critical observation in the QBF encoding in Equation 5.5 is that are pair of assignments that are not needed for proving the unsatisfiability, but for enforcing consistent labels in the partial model. In Equation 5.5, the universal variables depend on all existential variables which corresponds to the weakest QBF approximation of $\Phi$. By using a stronger QBF abstraction, the QBF formula itself takes care for a part of the consistency condition, which can decrease the bound needed to refute a DQBF formula.

**Example.** Consider again the PEC example from Figure 5.1. As we have seen, there exist two strongest QBF approximations, but both are satisfiable due to overapproximation. However, we prove unsatisfiability by using a strongest QBF abstraction together with a bound of two: The formula

$$\exists x_1^1, x_1^2.\, \forall y_1^1, y_1^2.\, \exists x_2^1, x_2^2.\, \forall y_2^1, y_2^2.\, ((y_2^1 \leftrightarrow y_2^2) \vee (x_1^1 \nleftrightarrow x_1^2)) \rightarrow (\neg\varphi^1 \vee \neg\varphi^2) \tag{5.7}$$

is satisfiable (choose arbitrary assignment $\alpha$ with $\alpha(x_1^1) \neq \alpha(x_1^2)$ and $\alpha(x_2^1) = \alpha(x_2^2)$). As the assignment of $y_2$ must be the same on both copies of universal assignments (otherwise it violates the consistency condition), it holds that either $\alpha(y_2) \neq \alpha(x_1^1)$ or $\alpha(y_2) \neq \alpha(x_1^2)$, hence the propositional formula is violated on either copy.

## 5.4 Experimental Results

We have implemented the bounded unsatisfiability method using a strongest QBF abstraction. In this section, we report on experiments carried out on a 2.6 GHz Opteron system. The QBF instances generated by our reduction are solved using a combination of the QBF preprocessor Bloqqer [BLS11] in version 031 and the QBF solver DepQBF [LB10] in version 2.0. As a base of comparison, we have also implemented an expansion-based DQBF solver using the BDD library CUDD in version 2.4.2[2].

[2] The source code of the benchmarks and tools are available at http://react.uni-saarland.de/tools/bunsat/

Table 5.1: Results of the bounded unsatisfiability method on PEC examples. The table shows the approximation ratio of the bounded-assignment prototype using a bound $\leq 2$ and the median running time relative to the expansion solver (timeout after 5min per instance). For every number of black boxes, we generated 1000 random instances.

| circuit | # black boxes | # unsat. | bound 1 | bound 2 | time (rel.) |
|---|---|---|---|---|---|
| multiplier | 1 | 950 | 100 % | 100 % | 27.5 % |
| | 3 | 927 | 97.6 % | 100 % | 22.2 % |
| | 5 | 924 | 87.6 % | 99.6 % | 17.9 % |
| | 7 | 912 | 67.9 % | 95.9 % | 13.8 % |
| | 9 | 870 | 30.9 % | 76.2 % | 16.2 % |
| adder | 1 | 962 | 100 % | 100 % | 10.8 % |
| | 3 | 959 | 100 % | 100 % | 9.0 % |
| | 5 | 959 | 99.9 % | 100 % | 8.9 % |
| | 7 | 951 | 99.5 % | 100 % | 7.0 % |
| | 9 | 957 | 98.5 % | 99.9 % | 6.8 % |
| multiplexer | 1 | 931 | 100 % | 100 % | 57.1 % |
| | 3 | 908 | 97.9 % | 99.8 % | 48.0 % |
| | 5 | 906 | 95.7 % | 98.5 % | 41.9 % |
| | 7 | 896 | 92.5 % | 97.0 % | 35.5 % |
| | 9 | 889 | 88.9 % | 94.7 % | 28.9 % |
| look-ahead | 1 | 999 | 100 % | 100 % | 4.5 % |
| | 3 | 997 | 98.2 % | 100 % | 3.4 % |
| | 5 | 996 | 97.1 % | 100 % | 3.3 % |
| | 7 | 996 | 94.2 % | 99.9 % | 0.7 % |
| | 9 | 986 | 84.4 % | 99.1 % | 0.8 % |

**Performance.** Table 5.1 shows the performance of our solver on several PEC benchmarks, including the arithmetic circuits *multiplier* (4-bit) and *adder* (32-bit), a 32-bit *look-ahead* arbiter implementation, and a 32-bit *multiplexer*. The PEC instances are created as follows: Starting with a circuit, we exchange a variable number of gates by black boxes and use one copy of the original circuit as the specification. Random faults are inserted by replacing precisely one gate with a gate of a different type. With only one exception (instances with more than 7 black boxes of the *multiplier* instances), more than 94 % of the instances were solved correctly with bound two, while the number of correctly solved instances by the QBF abstraction drops as low as 84.4 %. The running times in Table 5.1 are given relative to the running time of the expansion-based solver. Our solver outperforms the expansion-based solver significantly, especially on benchmarks with a large number of black boxes. For example, with 9 black boxes, the difference ranges from 37 % faster (*adder*) to more than 5 times faster (*lookahead*).

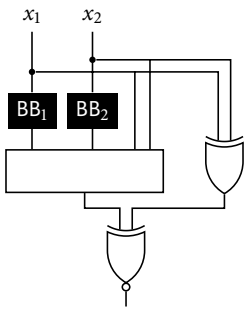



Figure 5.3: PEC problem corresponding to Table 5.2.

Table 5.2: Result of the XOR random function example. The table shows the approximation quality of the bounded-assignment prototype with respect to the number of black boxes.

| # black boxes | total | sat. | unsat. | bounded unsatisfiable | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1 | 2 | 3 | > 3 |
| 2 | 65536 | 32377 | 33159 | 22687 | 10472 | 0 | 0 |
| | | | 100 % | 68.4 % | 31.6 % | 0 % | 0 % |
| 3 | 50000 | 9273 | 40727 | 11257 | 26169 | 3115 | 186 |
| | | | 100 % | 27.6 % | 64.3 % | 7.6 % | 0.5 % |
| 4 | 50000 | 190 | 49810 | 5002 | 43781 | 1015 | 12 |
| | | | 100 % | 10.0 % | 87.9 % | 2.0 % | < 0.1% |

**Precision.** Table 5.1 indicates that an increase of the bound from 1, which corresponds to the plain QBF approximation, to 2 already results in a significant improvement of accuracy. Table 5.2 analyzes the impact of the bound on the approximation quality in more detail. Here, the benchmark is a circuit family from [Git+13], depicted for two black boxes in Figure 5.3. The circuit uses the XOR of the inputs as specification and a random Boolean function as implementation, where black boxes with pairwise different dependencies serve as inputs to this function. For two black boxes, we created all $65536 = 2^{16}$ instances of Boolean functions with four inputs. For more than two, we selected a random subset of 50 000 instances.

Table 5.2 shows an interesting correlation between the plain QBF approximation and the bounded-assignment method: The less effective the plain approximation, the more effective the bounded-assignment method. With an increasing number of black boxes, the number of solved instances by the plain QBF approximation (bound 1) decreases by more than a half with every black box. At the same time, the relative number of instances that are solved with a bound of at most two is always larger than 90 %. With a bound of at most three, nearly all unsatisfiable instances are detected (> 99 %). While the QBF approximation alone thus does not lead to satisfactory results, a comparatively small bound suffices to solve almost all instances.

## 5.5 Summary

We have presented a method for DQBF refutation that significantly outperforms a BDD-based DQBF solver based on variable expansion on PEC benchmarks. The bounded unsatisfiability method is based on an improved approximation of the DQBF formula encoded as a QBF, based on the search of multiple Herbrand functions and expressing dependency constraints as consistency conditions. Our experiments show that considering multiple Herbrand functions significantly improves accuracy, especially with an increasing number of black boxes. Compared to our expansion-based solver, the running time of our prototype implementation scales better with the number of black boxes. After the original formulation [FT14b], the bounded unsatisfiability method has been implemented by the expansion-based solver HQS [Git+15] and is used by default before the expansion solving loop.

# Chapter 6

# Clausal Abstraction for DQBF

We continue our investigation of DQBF solving methods. In this chapter, we lift the clausal abstraction algorithm to DQBF, which comes with two significant challenges: First, clausal abstraction is based on $Q$-resolution (see Section 3.2), and $Q$-resolution is sound but incomplete for DQBF [BCJ14a]. In particular, clauses may contain variables from incomparable quantifiers, so-called dependency forks [Rab17], which characterizes the reason for incompleteness. We address this problem using the *Fork Extension* [Rab17] proof rule, which allows us to split clauses with dependency fork into a set of clauses without dependency fork by introducing new variables. Second, clausal abstraction relies on the linear quantifier order of QBFs in prenex normal form. For DQBF, however, quantifiers can form an arbitrary partial order. When building a linear order by over-approximating the dependencies of existential variables and applying clausal abstraction naively, those variables may have *spurious dependencies*, i.e., they may only be able to satisfy the propositional part, if they depend on variables that are not allowed by the Henkin quantifiers. We show how to record consistency requirements, i.e., partial Skolem functions, that guarantee that existential variables solely depend on their stated dependencies.

In this chapter, we present the first abstraction based solving approach for DQBF. The algorithm successfully applies recent insight in solving quantified Boolean formulas: It is based on the versatile and award-winning clausal abstraction framework [RT15; JM15b; HT18; Ten16; Ten17] described in Chapter 2 and leverages progress in DQBF proof systems [Rab17]. Their integration in this work is non-trivial. To handle the non-linear dependencies, we use an over-approximation of the dependencies together with consistency requirements. Further, we turn clausal abstraction into an incremental algorithm that can accept new clauses and variables during solving. Our experiments show that our approach consistently outperforms first-order reasoning [Frö+14] on the DQBF benchmarks, and it is especially well-suited for the synthesis benchmark set [Fay+17] where expansion-based solvers fall short.

This chapter is based on work published in the proceedings of SAT [TR19a] and the corresponding technical report [TR19b].

**Related Work.** Fröhlich et al. [FKB12] proposed a first detailed solving algorithm for DQBF based on DPLL. They already encountered many challenges of lifting QBF algorithms to DQBF, like Skolem function consistency, replay of Skolem functions, forks in conflict clauses, but solved them differently. Their algorithm, called DQDPLL, has some similarities to our algorithm (in the same way that clausal abstraction and QDPLL share the same underlying proof system as shown in Chapter 3), but performs significantly worse [FKB12]. We highlight a few differences which we believe to be crucial:

1. Our algorithm tries to maintain as much order as possible. Placing universal nodes at the latest possible allows us to apply the cheaper QBF refinement method more often.
2. We learn consistency requirements only if they have been verified to satisfy the formula, while DQDPLL learns them on decisions. Consequently, in DQDPLL, learned Skolem functions become part of the clauses, thus, making conflict analysis more complicated and less effective as they may be undone during solving. We keep the consistency requirements distinct from the clauses, all learned clauses at existential quantifiers are thus valid during solving.
3. Skolem functions in DQDPLL are represented as clauses representing truth-table entries, thus, become quickly infeasible. In contrast, we use the certification mechanism introduced for QBF in Section 2.4.

Wimmer et al. [Wim+16] considered the problem of certifying Skolem functions produced by DQBF solvers. There has also been work on lifting QBF preprocessing techniques to DQBF [Wim+15; Wim+17].

## 6.1 Preliminaries

→ Page 16

For this chapter, we use the same notation as introduced for QBF in → Section 2.1. In the following, we state the additions needed to work with the more general logic.

We consider DQBF of the form

$$\forall x_1. \cdots \forall x_n. \exists y_1(H_1). \cdots \exists y_m(H_m). \varphi,$$

that is, DQBF begin with universal quantifiers followed by *Henkin quantifiers* and the quantifier-free part $\varphi$. Where unambiguous, we denote an existential variable by $y$, a set of existential variables by $Y$, and the set of all existential variables by $\mathcal{Y}$. For universal variables, we use $x$, $X$, and $\mathcal{X}$, respectively. A Henkin quantifier $\exists y(H)$ introduces a new variable $y$, like a normal quantifier, but also specifies a set $H \subseteq \mathcal{X}$ of *dependencies*. For this chapter, we assume that the propositional part $\varphi$ is given in conjunctive normal form (CNF). For literals $l$ of existential variables with dependency set $H$ we define $dep(l) = H$. For literals of universal variables we define $dep(l) = \{var(l)\}$. We lift the operator $dep$ to clauses by defining $dep(C) = \bigcup_{l \in C} dep(l)$.

**Definition 6.1** (Dependency Fork [Rab17][1])**.** A clause $C$ contains a *dependency fork* if there are two distinct existential variables $y$ and $y'$ with $\{y, y'\} \subseteq var(C)$ such that $y$ and $y'$ have incomparable dependencies, that is, $dep(y) \nsubseteq dep(y')$ and $dep(y') \nsubseteq dep(y)$.

**Relation to QBF in prenex form.** In QBF the dependencies of a variable are implicitly determined by the universal variables that occur before the quantifier in the quantifier prefix. This gives rise to the notion that QBF have a *linear* quantifier prefix, whereas DQBF allows for partially ordered quantifiers.

**Encoding Functions in DQBF.** One way to think about Henkin quantifiers is that they represent function applications—the existential variable they introduce is the result of applying the function to the variables in the dependency set. This allows us to easily encode the existential quantification over functions. For a detailed description of the encoding we refer to [Rab17].

## 6.2 A Resolution Style Proof System

In this section we recall the Fork Resolution proof system, which underlies the algorithm proposed in the following section. We also discuss a problem with the completeness of Fork Resolution and suggest two ways to overcome the problem. Fork Resolution consists of the well-known proof rules *resolution* and *universal reduction* and introduces a new proof rule called *Fork Extension* [Rab17].

*Resolution* allows us to merge two clauses as follows: Given two clauses $C_1 \vee v$ and $C_2 \vee \neg v$, we call $(C_1 \vee v) \otimes_v (C_2 \vee \neg v) = C_1 \vee C_2$ their *resolvent* with pivot $v$. The resolution rule states that $C_1 \vee v$ and $C_2 \vee \neg v$ imply their resolvent. *Universal reduction* allows us to drop universal variables from clauses when none of the existential variables in that clause may depend on them. Let $C$ be a clause, let $l \in C$ be a literal of a universal variable, and let $\bar{l} \notin C$. If for all existential variables $y$ in $C$ we have $var(l) \notin dep(y)$, universal reduction allows us to derive $C \setminus \{l\}$. *Fork Extension* allows us to split a clause $C_1 \vee C_2$ by introducing a fresh variable $y$. The dependency set of $y$ is defined as the intersection $dep(C_1) \cap dep(C_2)$ and represents that the question whether $C_1$ or $C_2$ satisfies the original clause needs to be resolved based on the information that is available to both of them. Fork Extension is usually only applied when $C_1$ and $C_2$ have incomparable dependencies ($dep(C_1) \nsubseteq dep(C_2)$ and $dep(C_1) \nsupseteq dep(C_2)$), as only then the dependency set of $y$ is smaller than those of $C_1$ and of $C_2$. The formal definition of the rule is

$$\frac{C_1 \cup C_2 \qquad y \text{ is fresh}}{\underbrace{\exists y(dep(C_1) \cap dep(C_2))}_{\text{quantifier prefix}} \qquad \underbrace{C_1 \cup \{y\} \wedge C_2 \cup \{\overline{y}\}}_{\text{matrix}}} \textbf{FEx}$$

[1] called *information fork* in the original formulation

**Example 6.2.** As an example of applying the Fork Extension, consider the quantifier prefix $\forall x_1, x_2.\, \exists y_1(x_1).\, \exists y_2(x_2)$ and clause $(\overline{x_1} \vee y_1 \vee y_2)$. Applying $(\mathbf{FEx})$ with the decomposition $C_1 = \{\overline{x_1}, y_1\}$ and $C_2 = \{y_2\}$ results in the clauses $(\overline{x_1} \vee y_1 \vee \boldsymbol{y_3})(\overline{\boldsymbol{y_3}} \vee y_2)$ where $y_3$ is a fresh existential variable with dependency set $dep(y_3) = \varnothing$ ($dep(C_1) = \{x_1\}$ and $dep(C_2) = \{x_2\}$).

Resolution is refutationally complete for propositional Boolean formulas. This means that for every propositional Boolean formula that is equivalent to false we can derive the empty clause using only resolution. In the same way, resolution with existential pivots and universal reduction (together they are called $Q$-resolution) are refutationally complete for QBF [KKF95]. For DQBF, however, $Q$-resolution is not sufficient—it was proven to be sound but incomplete [BCJ14a]. *Fork Resolution* addresses this problem by extending $Q$-resolution by the Fork Extension proof rule [Rab17].

Unfortunately, the proof of the completeness of Fork Resolution relied on a hidden assumption that we uncovered by implementing and testing the algorithm proposed in the following section. Consider the DQBF with prefix $\forall x_1, x_2, x_3.\, \exists y_1(x_1, x_2).\, \exists y_2(x_2, x_3).\, \exists y_3(x_1, x_3)$ and a clause $C = (y_1 \vee y_2 \vee y_3)$. Formally, $C$ is a *dependency fork* according to Definition 6.1, i.e., it contains variables with incomparable dependencies. However, we cannot apply $(\mathbf{FEx})$ because any split of the clause into two parts $C = C_1 \cup C_2$ satisfies either $dep(C_1) \subseteq dep(C_2)$ or $dep(C_1) \supseteq dep(C_2)$. Fork Extension therefore fails its purpose in this case to eliminate all dependency forks as required by the proof of completeness in [Rab17]. We say that dependency forks that Fork Extension cannot split with a literal with smaller dependency set have a *dependency cycle*. It is easy to extend the example above to a formula for which Fork Resolution is incomplete (see Section 6.2.1).

We see two ways to counter this problem. The first is to consider a normal form of DQBF that does not have dependency cycles. For example, we can restrict to DQBFs where every incomparable pair of dependency sets must have an empty intersection. This fragment does not admit any dependency cycles and it is NExpTime-complete: The formula used for the NExpTime-completeness proof of the deciding the satisfiability problem lies in this fragment [PRA01]. Thus, this fragment therefore could be used as a normal form of DQBF. It also guarantees that every dependency set that gets introduced through Fork Extension maintains this property (in fact, only variables with the empty dependency set can be created). In this way, the Fork Resolution proof system is indeed strong enough to serve as a proof system for DQBF. In fact, most applications already fall in this fragment. The synthesis problems introduced in the introduction that can be expressed as the existence of a function $f\colon \mathbb{B}^m \to \mathbb{B}^n$, such that for all tuples of inputs $\vec{x}_1, \ldots, \vec{x}_k \in \mathbb{B}^m$ some relation $\varphi(\vec{x}_1, f(\vec{x}_1), \ldots, \vec{x}_k, f(\vec{x}_k))$ over function applications of $f$ is satisfied, are contained in this fragment: By the typical translation into DQBF this results in a formula with pairwise disjoint dependency sets plus the Tseitin variables that may depend on every universal variable [Rab17]. In particular, we have never observed a dependency cycle in any of the available benchmark sets. In Section 6.2.2 we give a more concise definition of admissible DQBF fragments.

The second approach is to avoid this normal form and strengthen Fork Extension in

a way that allows us to break dependency cycles. The new rule Strong Fork Extension extends Fork Extension by the ability to add a set of universal literals $C_X$ to the two clauses that it produces. Intuitively, adding the literals $C_X$ restricts the Skolem function of $y$ to the case that all literals in $C_X$ are false. Hence $y$ does not need to explicitly depend on $dep(C_X)$. This allows us to remove $dep(C_X)$ from the dependency set of the freshly introduced variable $y$.

$$\frac{C_1 \cup C_2 \qquad y \text{ is fresh} \qquad C_X \text{ is a set of universal literals}}{\underbrace{\exists y((dep(C_1) \cap dep(C_2)) \setminus dep(C_X))}_{\text{quantifier prefix}} \qquad \underbrace{C_X \cup C_1 \cup \{y\} \;\wedge\; C_X \cup C_2 \cup \{\overline{y}\}}_{\text{matrix}}} \textbf{SFEx}$$

**Lemma 6.3.** *The Strong Fork Extension rule is sound.*

*Proof.* Given any Skolem function for the formula, we make a case split over the assignments to the universals: If a literal of $C_X$ is true, both produced clauses are true and the rule is trivially sound. If all literals of $C_X$ are false, the Strong Fork Extension is equivalent to Fork Extension [Rab17]. □

**Theorem 6.4.** *Strong Fork Resolution is sound and complete for DQBF.*

*Proof.* The proof of completeness of Fork Resolution assumed that any dependency fork can be split with Fork Extension, by introducing new literals with *smaller* dependency sets [Rab17]. Strong Fork Extension guarantees this property also for dependency cycles: We pick some universal variable $x$ of the dependency sets and split the original clause twice; once with $C_X = \{x\}$ and once with $C_X = \{\overline{x}\}$. This results in four clauses that together imply the original clause (such that the original clause can be dropped from the formula), and the two variables introduced have smaller dependency sets. The rest of the proof remains the same. □

### 6.2.1 Incompleteness Example

We give an example to demonstrate that Fork Resolution is incomplete for general DQBF. The example formula is an extension of the incompleteness examples used in [BJ12]. The formula is

$$\forall x_1, x_2, x_3.\, \exists y_1(x_1, x_2).\, \exists y_2(x_2, x_3).\, \exists y_3(x_1, x_3).\, (x_1 \wedge x_2 \wedge x_3) \leftrightarrow (y_1 \oplus y_2 \oplus y_3)$$

In CNF, the propositional part looks as follows:

$$\begin{aligned}
&(x_1 \vee x_1 \vee y_2 \vee \overline{y}_3)(x_1 \vee x_1 \vee y_3 \vee \overline{y}_2)(x_1 \vee y_2 \vee y_3 \vee \overline{x}_1)(x_2 \vee x_1 \vee y_2 \vee \overline{y}_3)\\
&(x_2 \vee x_1 \vee y_3 \vee \overline{y}_2)(x_2 \vee y_2 \vee y_3 \vee \overline{x}_1)(x_3 \vee x_1 \vee y_2 \vee \overline{y}_3)(x_3 \vee x_1 \vee y_3 \vee \overline{y}_2)\\
&(x_3 \vee y_2 \vee y_3 \vee \overline{x}_1)(x_1 \vee \overline{x}_1 \vee \overline{y}_2 \vee \overline{y}_3)(x_2 \vee \overline{x}_1 \vee \overline{y}_2 \vee \overline{y}_3)(x_3 \vee \overline{x}_1 \vee \overline{y}_2 \vee \overline{y}_3)\\
&(x_1 \vee y_2 \vee y_3 \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{y}_2 \vee \overline{y}_3)\\
&(y_2 \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_1 \vee \overline{y}_3)(y_3 \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_1 \vee \overline{y}_2)
\end{aligned}$$

It is easy to verify that the following statements hold:

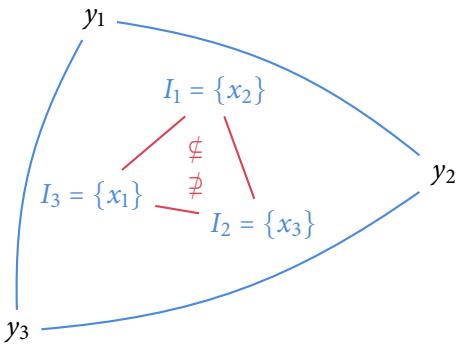



Figure 6.1: Visualization of the dependency cycle for clause $(y_1 \vee y_2 \vee y_3)$ with prefix $\forall x_1, x_2, x_3.\, \exists y_1(x_1, x_2).\, \exists y_2(x_2, x_3).\, \exists y_3(x_1, x_3)$.

- The formula is false.
- Universal reduction cannot be applied to any clause.
- All resolvents are tautologies.
- Fork Extension is not applicable.
- The formula does not contain the empty clause.

This means that Fork Resolution proof system is not strong enough to refute any DQBF.

### 6.2.2 Dependency Cycles

In the following, we formalize dependency cycles and show that they are the reason for incompleteness of Fork Extension. A clause $C$ contains a *dependency cycle* of length $k > 2$, if there is a subset $\{l_1, l_2, \ldots, l_k\} \subseteq C$ of existential literals such that the intersections of dependencies $I_i = dep(l_i) \cap dep(l_{i+1}) \neq \varnothing$ for all $1 \leq i \leq k$, with $l_{k+1} = l_1$ contain pairwise disjoint variables, i.e., $I_i \not\subseteq I_j$ and $I_j \not\subseteq I_i$ for each $i \neq j$. Figure 6.1 depicts a representation of the dependency cycle. Given a clause $C$, the *clause poset*, written $poset(C)$, is a partially ordered set $\langle P, \subseteq \rangle$ where $P \subseteq \mathcal{V}$ is the set of dependencies of existential literals in $C$, i.e., $\{dep(l) \mid l \in C \wedge l \text{ is existential}\}$. If $poset(C)$ contains more than one maximal element w.r.t. $\subseteq$, $C$ contains a *dependency fork*.

**Lemma 6.5.** *Fork Extension is applicable for clauses with dependency fork if, and only if, the clause does not contain a dependency cycle.*

*Proof.* Assume $C$ contains a dependency cycle, that is, there is a $k > 2$ and $\{l_1, l_2, \ldots, l_k\} \subseteq C$ of existential literals such that $I_i = dep(l_i) \cap dep(l_{i+1}) \neq \varnothing$ for all $1 \leq i \leq k$, with $l_{k+1} = l_1$ contain pairwise disjoint variables, i.e., $I_i \not\subseteq I_j$ and $I_j \not\subseteq I_i$ for each $i \neq j$. W.l.o.g. we assume that $\{l_1, l_2, \ldots, l_k\}$ are the only existential variables in $C$. Let $C_1 \cup C_2$ be an arbitrary split containing at least one existential variable. Then, either (**FEx**) is not applicable ($dep(C_1) \subseteq dep(C_2)$ or $dep(C_1) \supseteq dep(C_2)$) or applying it will lead to a clause with dependency cycle: Let $y$ be the fresh variable

with $dep(y) = dep(C_1) \cap dep(C_2)$. If $C_1$ contains a single existential literal $l_i$, then $dep(y) \cap dep(l_{i+1}) \neq \emptyset$ and $dep(y) \cap dep(l_{i-1}) \neq \emptyset$, i.e., the resulting clause has a dependency cycle of length $k$. If $C_1$ contains $j > 1$ existential literals, then both resulting clauses contain a dependency cycle of length $j+1$ and $k-j+1$.

Assume $C$ does not contain a dependency cycle, that is, there is a maximal element $H$ in $poset(C)$, such that there is a unique maximal element in the set of intersections with other maximal elements $H^* = \max_{\subseteq} \{H \cap H' \mid H' \text{ is a maximal element of } poset(C)\}$. We use the Fork Extension rule ($\mathbf{FEx}$) with $C_1 = \{l \in C \mid dep(l) \subseteq H, dep(l) \nsubseteq H^*\}$ and $C_2 = C \setminus C_1$. □

As the proof shows that any dependency cycle can be split into smaller dependency cycles, in the following we only need to argue that there does not exist a dependency cycle involving three existential variables.

In the following, we want to characterize fragments of DQBF (based on the quantifier prefix) that do not exhibit dependency cycles. A DQBF formula is in the *multi-linear* fragment of DQBF, if for all pairs of existential variables $y_1$ and $y_2$ with dependency sets $H_1$ and $H_2$ it holds that $H_1 \subseteq H_2$, $H_2 \subseteq H_1$, or $H_1 \cap H_2 = \emptyset$.

**Theorem 6.6.** *The multi-linear fragment of DQBF does not contain dependency cycles.*

*Proof.* Assume we have three variables $y_1$, $y_2$, and $y_3$ with dependency sets $H_1$, $H_2$, and $H_3$. Further, let $H_1 \cap H_2 \neq \emptyset$ and $H_2 \cap H_3 \neq \emptyset$, that is, $H_1 \subseteq H_2$ or $H_2 \subseteq H_1$, and $H_2 \subseteq H_3$ or $H_3 \subseteq H_2$, thus, there are 4 combinations:

- $H_1 \subseteq H_2$ and $H_2 \subseteq H_3$, thus $H_1 \subseteq H_3$ and $H_1 \cap H_2 \subseteq H_2 \cap H_3$
- $H_1 \subseteq H_2$ and $H_3 \subseteq H_2$, thus $H_1 \cap H_3 \subseteq H_1 \cap H_2$
- $H_2 \subseteq H_1$ and $H_2 \subseteq H_3$, thus $H_1 \cap H_2 = H_2 \cap H_3$
- $H_2 \subseteq H_1$ and $H_3 \subseteq H_2$, thus $H_3 \subseteq H_1$ and $H_2 \cap H_3 \subseteq H_1 \cap H_2$

which rules out any dependency cycle. □

**Corollary 6.7.** *Fork Extension is complete for the multi-linear fragment of DQBF.*

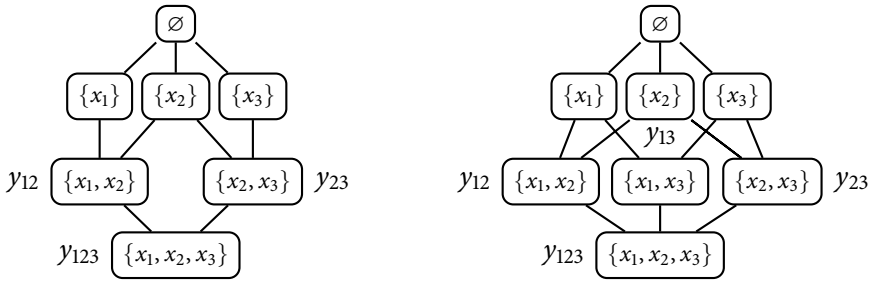
This characterization, however, is not tight in the following sense: There are many more quantifier prefixes that rule out the existence of dependency cycles. Consider and compare the two prefixes

$$\forall x_1, x_2, x_3.\, \exists y_{12}(x_1, x_2).\, \exists y_{23}(x_2, x_3).\, \exists y_{123}(x_1, x_2, x_3) \qquad \text{(planar)}$$

$$\forall x_1, x_2, x_3.\, \exists y_{12}(x_1, x_2).\, \exists y_{13}(x_1, x_3).\, \exists y_{23}(x_2, x_3).\, \exists y_{123}(x_1, x_2, x_3) \qquad \text{(non-planar)}$$

which are both not in the multi-linear fragment. The first prefix cannot produce dependency cycles while the latter one does. To derive a more concise characterization, we inspect the partial order underlying the dependency sets. In the following, we investigate the underlying reason using Hasse diagrams that represent the partial order defined by the quantifier prefix of a DQBF.

Given a DQBF $\Phi$, the DEPENDENCY LATTICE is the meet-semilattice $\langle \mathcal{H}, \subseteq \rangle$ where $\mathcal{H}$ contains all dependency sets of variables in $\Phi$ and additionally all intersections $H \cap H'$




(a) dependency lattice for (planar) (b) dependency lattice for (non-planar)

Figure 6.2: Visualizations of the dependency lattice as Hasse diagrams for (planar) and (non-planar).

for every $H, H' \in \mathcal{H}$. We depict the dependency lattice for our example prefixes as Hasse diagrams in Figure 6.2. When comparing those two representations, we observe that the left one admits a planar Hasse diagram, that is, there are no crossing edges, while the right one does not. A finite lattice has a planar Hasse diagram if, and only if, there is a *conjugate* partial order [Bir67], where a conjugate order is an order on the *incomparable* elements of the lattice. For the dependency lattice of (planar), we can verify that the relation $\prec_1$ given as $\{x_1\} \prec_1 \{x_2\}$, $\{x_1\} \prec_1 \{x_3\}$, $\{x_2\} \prec_1 \{x_3\}$, $\{x_1\} \prec_1 \{x_2, x_3\}$, $\{x_3\} \prec_1 \{x_1, x_2\}$, and $\{x_1, x_2\} \prec_1 \{x_2, x_3\}$ is such a conjugate order. Consider the analogous relation $\prec_2$ for (non-planar), built from $\prec_1$ with the additional entries $\{x_2\} \prec_2 \{x_1, x_3\}$, $\{x_1, x_2\} \prec_2 \{x_1, x_3\}$, $\{x_1, x_3\} \prec_2 \{x_2, x_3\}$. We observe that $\prec_2$ is not transitive: It holds that $\{x_2\} \prec_2 \{x_1, x_3\}$ and $\{x_1, x_3\} \prec_2 \{x_2, x_3\}$, while $\{x_2\} \not\prec_2 \{x_2, x_3\}$ as $\{x_2\} \subseteq \{x_2, x_3\}$.

**Theorem 6.8.** *Let $\Phi$ be a DQBF. If the dependency lattice of $\Phi$ can be represented as a planar Hasse diagram, then $\Phi$ does not contain dependency cycles.*

*Proof.* Let $\prec$ be the conjugate order for the incomparable elements in the dependency lattice of $\Phi$. Assume for contradiction that there is a dependency cycle, that is, a clause $C$ such that the $poset(C)$ contains 3 maximal elements $H_1$, $H_2$, and $H_3$ such that $I_{12} = H_1 \cap H_2 \neq \emptyset$, $I_{13} = H_1 \cap H_3 \neq \emptyset$, and $I_{23} = H_2 \cap H_3 \neq \emptyset$ are pairwise incomparable. W.l.o.g. assume that $H_1 \prec H_2 \prec H_3$, thus also $H_1 \prec H_3$ as $\prec$ is an order relation. Further, we know that $I_{ij} \not\prec H_i$, $I_{ij} \not\prec H_j$, $H_i \not\prec I_{ij}$ and $H_j \not\prec I_{ij}$ for each $(i, j) \in \{(1, 2), (1, 3), (2, 3)\}$ by construction of $I_{ij}$ as the intersection of $H_i$ and $H_j$. In the following, we we show that the incomparable elements of the dependency cycle cannot be ordered according to $\prec$.

- If $H_2 \prec I_{13}$, then we derive the contradiction that $H_1 \prec I_{13}$ due to $H_1 \prec H_2$.
- If $H_3 \prec I_{12}$, then we derive the contradiction that $H_2 \prec I_{12}$ due to $H_2 \prec H_3$.

Thus, we know that $I_{13} \prec H_2$ and $I_{12} \prec H_3$.

- Assume that $H_1 \prec I_{23}$, then $I_{23} \prec I_{12}$ and $I_{23} \prec I_{13}$ would immediately lead to the contradiction that $H_1 \prec I_{12}$ and $H_1 \prec I_{13}$, respectively. Thus $I_{12} \prec I_{23}$ and $I_{13} \prec I_{23}$.

- Assuming that $I_{13} \prec I_{12}$ leads to the contradiction $I_{13} \prec H_3$ due to $I_{12} \prec H_3$.
- Assuming that $I_{12} \prec I_{13}$ leads to the contradiction $I_{12} \prec H_2$ due to $I_{13} \prec H_2$.

Thus, $I_{23} \prec H_1$.

Finally, $I_{23} \prec H_1$ leads to the contradiction $I_{23} \prec H_2$ due to $H_1 \prec H_2$. □

The branching fragment of DQBF are those DQBF were the quantifier prefix can be represented as branching quantifier. The prefix (planar) can be equivalently stated as $\forall x_1. \begin{smallmatrix}\forall x_2.\exists y_{12}.\\ \forall x_3.\exists y_{23}.\end{smallmatrix} \exists y_{123}$.

**Corollary 6.9.** *Fork Extension is complete for the branching fragment of DQBF.*

*Proof.* The branching representation admits a natural conjugate order. □

## 6.3 Lifting Clausal Abstraction

In this section, we lift clausal abstraction to DQBF. We begin with a high level explanation of the algorithm for QBF and a discussion of the invariants that hold for QBF but are no longer valid for DQBF. For each of those we identify the underlying problem and show how we need to modify clausal abstraction. In the following subsections we then explain those extensions in detail. For the remainder of this section, we assume w.l.o.g. that we are given a DQBF $\Phi$ with matrix $\varphi$, that $\varphi$ does not contain clauses with dependency forks, and that every clause is universally reduced. If a formula contains dependency forks initially, they can be removed as described in Section 6.3.4.

The clausal abstraction algorithm assigns existential and universal variables, where the order of assignments is determined by the quantifier prefix, until all clauses in the matrix are satisfied or there is a conflict, i.e., a set of clauses that cannot be satisfied simultaneously. Those variable assignments are generated by propositional formulas, one for every quantifier, called *abstractions*. In case of a conflict, the reason for this conflict is excluded by refining the abstraction at an outer quantifier.

The assignment order is based on the quantifier prefix. Thus, for QBF it holds that an existential variable is only assigned if its dependencies are assigned. In DQBF, Henkin quantifiers allow us to introduce incomparable dependency sets, and hence, in general, there is no linear order of assignments. We thus weaken this invariant by requiring that for every existential variable $y$, all of its dependencies have to be assigned before assigning $y$. We ensure this by creating a graph-based data structure, which we call quantifier levels, described in Section 6.3.1. As an immediate consequence, and in contrast to QBF, an existential variable may be assigned different values depending on assignments to non-dependencies, and we call this phenomenon a *spurious dependency*. To eliminate those spurious dependencies, we enhance the certification approach of clausal abstraction (see Section 2.4) to build, incrementally, a constraint system that enforces that an existential variable only depends on its dependencies. These *consistency requirements* represent partial Skolem functions. Section 6.3.5 describes how the consistency requirements are derived, how they are integrated in the algorithm, and when they are invalidated.

We build an abstraction for every existential quantifier $\exists Y$, splitting every clause $C$ of the matrix into three parts, based on whether a literal $l \in C$ is (1) a dependency, (2) a literal of a variable in $Y$, or (3) neither of the two. Section 6.3.2 gives a formal description of the abstraction. As mentioned, all dependencies of $Y$ must be assigned before we query the abstraction of the quantifier $\exists Y$ for a candidate assignment of variables $Y$. From the perspective of this abstraction, assignments to non-$Y$ variables are equivalent when they satisfy the same set of clauses. Vice versa, the only information that matters for other abstractions is the set of clauses satisfied by variables $Y$ or their dependencies. The abstraction for $Y$ therefore defines a set of interface variables consisting of *satisfaction variables* and *assumption variables*, one for every clause $C$, where the satisfaction variable indicates whether the clause is satisfied by a dependency of $Y$ and the assumption variable indicates whether $C$ must still be satisfied by variables outside of $Y$. Conflicts are represented by a set of assumption variables that turned out to be unsatisfiable with respect to variables outside of $Y$. Refinements are clauses over those assumption variables, requiring that at least one of those contained clauses is satisfied by an assignment to $Y$.

Those refinements correspond to conflict clauses in search-based algorithms and can be formalized as derived clauses in the $Q$-resolution calculus (see Section 3.2). Since $Q$-resolution is incomplete for DQBF [BCJ14a] and the incompleteness can be characterized by clauses with dependency fork as discussed in the previous section, we check if a conflict clause derived by the algorithm contains such a fork. If this is the case, we *split* this clause into a set of clauses that are fork-free. As a byproduct, new existential variables are created. We show in Section 6.3.4 how clauses with dependency fork are split and how the clausal abstraction algorithm is extended to *incrementally* accept new clauses and variables.

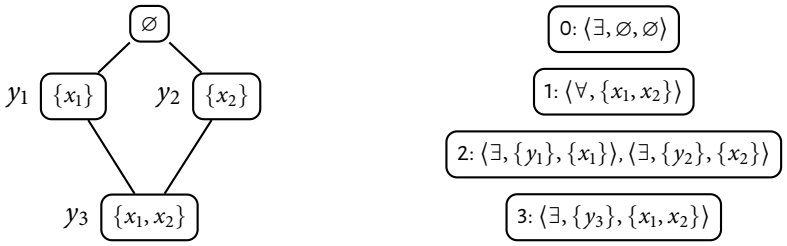
**Example 6.10.** We will use the following formula with the dependency sets $\{x_1\}$, $\{x_2\}$, and $\{x_1, x_2\}$ as a running example.

$$\forall x_1, x_2.\, \exists y_1(x_1).\, \exists y_2(x_2).\, \exists y_3(x_1, x_2).$$
$$\underbrace{(x_1 \vee \overline{x_2} \vee \overline{y_2} \vee y_3)}_{C_1}\ \underbrace{(\overline{x_1} \vee y_2 \vee y_3)}_{C_2}\ \underbrace{(\overline{y_1} \vee x_2 \vee \overline{y_3})}_{C_3}\ \underbrace{(y_1 \vee \overline{y_3})}_{C_4}\ \underbrace{(x_1 \vee y_1)}_{C_5}$$

### 6.3.1 Quantifier Levels

To lift clausal abstraction to DQBF, we need to deal with *partially* ordered dependency sets. Given a DQBF $\Phi$, the algorithm starts by building the dependency lattice as introduced in Section 6.2.2. Thus, it constructs the set $\mathcal{H}$ that contains all dependency sets of variables in $\Phi$ and, additionally, all $H \cap H'$ such that $H, H' \in \mathcal{H}$. For our running example, we have to add the empty dependency set, resulting in the dependency lattice depicted in Figure 6.3a. In addition to the dependency sets $\mathcal{H}$ and the edge relation $\subseteq$, we depict the existential variables next to their dependency sets.

**Quantifier Levels and Nodes.** We continue with building the data structure on which the algorithm operates. A *node* binds a variable of the DQBF. A universal node $\langle \forall, X \rangle$




(a) dependency lattice (b) quantifier levels

Figure 6.3: Dependency lattice and quantifier levels for the DQBF given in Example 6.10.

binds universal variables $X$ and an existential node $\langle \exists, Y, H \rangle$ binds existential variables $Y$ with dependency set $H$. Nodes are grouped together in *quantifier levels*, where each universal level contains exactly one universal node and existential levels may contain multiple existential nodes (which have all pairwise incomparable dependency sets). We index levels by natural numbers $i$, starting with 0. In Figure 6.3b we depict an example for the data structure obtained from the dependency lattice on its left. Before describing the construction of quantifier levels, we state their invariants. For some node $N$, let $bound_\forall(N)$ be the set of universal variables bound at $N$, i.e., the union of all $X$ where $\langle \forall, X \rangle$ is in a level with smaller index than node $N$. Let $bound_\exists(N)$ be the analogously defined set of bound existential variables. The set of bound variables is $bound(N) \coloneqq bound_\exists(N) \dot\cup bound_\forall(N)$.

**Proposition 6.11.** *The quantifier levels data structure has the following properties.*

1. *Every variable is bound excactly once, i.e., for every variable $v$ in $\Phi$, there is exactly one node $\langle \forall, X \rangle$ or $\langle \exists, Y, H \rangle$ such that $v \in X$ or $v \in Y$.*
2. *Every pair of nodes $\langle \exists, Y, H \rangle$ and $\langle \exists, Y', H' \rangle$ with $Y \neq Y'$ contained in an existential level have incomparable dependencies, i.e., $H \nsubseteq H'$ and $H \nsupseteq H'$.*
3. *For every pair of nodes $\langle \exists, Y_i, H_i \rangle$ and $\langle \exists, Y_j, H_j \rangle$ contained in existential levels $i$ and $j$ with $i < j$, it holds that either $H_i \subset H_j$ or $H_i$ and $H_j$ are incomparable.*
4. *For every existential node $\langle \exists, Y, H \rangle$ it holds that $H \subseteq bound_\forall(\langle \exists, Y, H \rangle)$.*
5. *There is a unique maximal $\langle \exists, Y, H \rangle$ with $H \supseteq H'$ for every $\langle \exists, Y', H' \rangle$.*

In the following, we describe the construction of quantifier levels from a dependency lattice. Every element of the dependency lattice $H \in \mathcal{H}$ is transformed into one existential node, $\langle \exists, Y, H \rangle$, where $Y$ is the set of existential variables with dependency set $H$, i.e. $dep(y) = H$ for all $y \in Y$. Some existential nodes (like the root node in our example) may be initially empty. The existential levels are obtained by an antichain decomposition of the dependency lattice (satisfying Proposition 6.11.2 and 6.11.3). If the dependency lattice does not contain a unique maximal element, we add an empty existential node $\langle \exists, \mathcal{X}, \varnothing \rangle$ (satisfying Proposition 6.11.5).

Universal variables are placed in the universal node just before the existential level they first appear in as a dependency. This is achieved by a top-down pass through the ex-

**Algorithm 6.1** Main solving algorithm that iterates over the quantifier levels

```
1:  procedure SOLVE(DQBF Φ)
2:      levels ← build quantifier levels
3:      initialize every node in levels, i.e., build abstraction θ, set entries ← []
4:      α_V ← {v ↦ F | v ∈ V}, lvl ← 0
5:      loop
6:          match SOLVELEVEL(lvl) as
7:              CandidateFound ⇒ lvl ← lvl + 1
8:              Conflict(jmpBackToLvl) ⇒ lvl ← jmpBackToLvl
9:              Result(res) ⇒ return res
10:     end loop
11: end procedure
```

istential quantifier levels, adding a universal level with node $N = \langle \forall, X \rangle$ before existential level with nodes $\langle \exists, Y_1, H_1 \rangle, \ldots, \langle \exists, Y_k, H_k \rangle$ such that $X = (\bigcup_{1 \le i \le k} H_i) \setminus bound_\forall(N)$ (satisfying Proposition 6.11.4).[2] Empty universal levels $\langle \forall, \varnothing \rangle$ are omitted. Level numbers follow the inverse order of the dependency sets, such that the "outer" quantifiers have smaller level numbers than the "inner" quantifiers; see Figure 6.3.

If the formula is a QBF, it holds that $bound_\forall(\langle \exists, Y, H \rangle) = H$. For QBF, this construction yields a strict alternation between universal and existential levels, but for DQBF existential levels can succeed each other, as shown in Figure 6.3.

**Algorithmic Overview.** The overall approach of the algorithm is to construct a propositional formula $\theta$ for every node, that represents which clauses it can satisfy (for existential nodes) or falsify (for universal nodes). We describe their initialization in detail below. In every iteration of the loop in algorithm Solve (Algorithm 6.1) the variable assignment $\alpha_V$ is extended (case CandidateFound), which we assume to be globally accessible, or node abstractions are refined by adding an additional clauses (case Conflict).

The nodes are responsible for determining candidate assignments to the variables bound at that node, or to give a reason why there is no such assignment. If a node is able to provide a candidate assignment, we proceed to the successor level (Algorithm 6.1, line 7). A conflict occurs when the algorithm determines that the current assignment $\alpha_V$ definitely violates the formula (unsat conflict) or satisfies it (sat conflict). When conflicts are inspected (explained in Section 6.3.3), they indicate a level that tells the main loop how far we have to jump back (Algorithm 6.1, line 8). The last alternative in the main loop is that we have found a result, which allows us to terminate (line 9).

[2] It is possible to introduce all universal variables upfront, but this negatively affects solving performance. Equality is not always possible, as shown by the formula $\forall x_1, x_2, x_3 \exists y_1(x_1) \exists y_2(x_2) \exists y_3(x_2, x_3)$.

### 6.3.2 Initialization of the abstractions $\theta$

The formula $\theta$ for each node represents how the node's variables interact with the assignments on other levels. The algorithm guarantees that whenever we generate a candidate assignment for a node, all variables on outer (=smaller) levels have a fixed assignment, and thus some set of clauses is satisfied already. Existential nodes then try to satisfy more clauses with their assignment, while universal nodes try to find an assignment that makes it harder to satisfy all clauses. An existential variable $y$ may not only depend on assignments of its dependencies, but also on assignments of existential variables with strict smaller dependency as they are in a strictly smaller level (see Section 6.3.1) and thus are assigned before $y$. We call this the extended dependency set, written $exdep(y)$, and it is defined as $dep(y) \dot\cup \{y' \in \mathcal{Y} \mid dep(y') \subset dep(y)\}$. For a set $Y \subseteq \mathcal{Y}$, we define $exdep(Y) = \bigcup_{y \in Y} exdep(y)$.

The interaction of abstractions is established by a common set of *clause satisfaction* variables $S$, one variable $s_i \in S$ for every clause $C_i \in \varphi$. Given some existential node $\langle \exists, Y, H \rangle$ with extended dependency set $D = exdep(Y)$ and assignment $\alpha_V$ of outer variables $V$ (w.r.t. $\exists Y$, i.e., $V = bound(\langle \exists, Y, H \rangle)$). For every clause $C_i \in \varphi$ it holds that if $s_i$ is assigned to true, one of its dependencies has satisfied the clause, that is, $\alpha_V \vDash C_i|_D$. Thus, an assignment of the satisfaction variables $\alpha_S$ is an abstraction of the concrete variable assignment $\alpha_V$ as multiple assignments could led to the same satisfied clauses.

For universal quantifiers, this abstraction is sufficient as the universal player tries to satisfy as few clauses as possible. For existential quantifiers, however, the existential player can choose to either satisfy the clause directly or *assume* that the clause will be satisfied by an inner quantifier. Thus, we add an additional type of variables $A$, called *assumption variables*, with the intended semantics that $a_i$ is set to false at some existential quantifier $\exists Y$ implies that the clause $C_i$ is satisfied at this quantifier (either by an assignment $\alpha_Y$ to variables $Y$ of the current node or an assignment of dependencies represented by an assignment $\alpha_S$ to the satisfaction variables $S$), formally, $\alpha_V \dot\cup \alpha_Y \vDash C_i|_{D \dot\cup Y}$ if $a_i$ is false.

We continue by defining the abstraction that implements this intuition. Formally, for every node $\langle \exists, Y, H \rangle$ and every clause $C_i$, we define $C_i^< := \{l \in C_i \mid var(l) \in exdep(Y)\}$ as the set of literals on which the current node may depend, $C_i^= := \{l \in C_i \mid var(l) \in Y\}$ as the the set of literals which the current node binds, and $C_i^> := \{l \in C_i \mid var(l) \notin exdep(Y) \cup Y\}$ as the set of literals on which the current node may not depend. By definition, it holds that $C_i = C_i^< \dot\cup C_i^= \dot\cup C_i^>$. The clausal abstraction $\theta_Y$ for this node is defined as $\bigwedge_{C_i \in \varphi} (a_i \vee s_i \vee C_i^=)$. Note, that $s_i$ and $a_i$ are omitted if $C_i^< = \varnothing$ and $C_i^> = \varnothing$, respectively.

Over time, the algorithm calls each node potentially many times for candidate assignments, and it adds new clauses learnt from refinements. The new clauses for existential nodes will only contain literals from assumption variables $L \subseteq A$, representing sets of clauses that together cannot be satisfied by the inner levels. The refinement $\bigvee_{a_i \in L} \overline{a_i}$ ensures that some clause $C_i$ with $a_i \in L$ is satisfied at this node.

Universal nodes $\langle \forall, X \rangle$ have the objective to falsify clause. We define the abstraction $\theta_X$ for this node as $\bigwedge_{C_i \in \varphi} (s_i \vee \neg C_i^=) = \bigwedge_{C_i \in \varphi} \left( s_i \vee \bigwedge_{l \in C_i^=} \overline{l} \right)$. Observe that univer-

**Algorithm 6.2** Algorithm for solving a quantifier level by iterating over its nodes

```
1:  procedure SolveLevel(lvl)
2:      if lvl is universal then return Solve∀(levels[lvl])
3:      end if
4:      for each node n in levels[lvl] do
5:          if Solve∃(n) = Conflict(jmpBackToLvl) then
6:              return Conflict(jmpBackToLvl)
7:          end if
8:      end for
9:      return CandidateFound
10: end procedure
```

sal nodes do not have separate sets of variables $A$ and $S$, but just one copy $S$. This is just a minor simplification, exploiting the formula structure of universal nodes. Note that $s_i$ set to false implies that $\alpha_X$ falsifies the literals in the clause, that is, $\alpha_X \vDash \neg C_i^{=}$. Refinements are represented as clauses $\bigvee_{s_i \in L} \overline{s_i}$ over literals in $S$.

In our running example, clauses 3–5 $(\overline{y_1} \vee x_2 \vee \overline{y_3})(y_1 \vee \overline{y_3})(x_1 \vee y_1)$ are represented at node $\langle \exists, \{y_1\}, \{x_1\} \rangle$ by clauses $(a_3 \vee \overline{y_1})(a_4 \vee y_1)(y_1)$. Note especially, that $x_2 \notin exdep(y_1) = \{x_1\}$, thus there is no $s$ variable in the first encoded clause, despite $x_2$ being assigned earlier in the algorithm (Figure 6.3).

### 6.3.3 Solving Levels and Nodes

SolveLevel in Algorithm 6.2 directly calls Solve$_\forall$ or Solve$_\exists$ on all the nodes contained in the given level *lvl*. For existential levels, if any node returns a conflict, the level returns that conflict (Algorithm 6.2, line 6).

We process universal and existential nodes with the two procedures shown in Algorithm 6.3. The SAT solvers generate a candidate assignment to the variables (lines 4 and 15) of that node, which is then used to extend the (global) assignment $\alpha_V$ (lines 7 and 18). In case the SAT solver returns Unsat, the unsat core represents a set of clauses that cannot be satisfied (for existential nodes) or falsified (for universal nodes). The unsat core is then used to refine an outer node (lines 6 and 17) and we proceed with the level returned by Refine.

**Solving Existential Nodes.** There are some differences in the handling of existential and universal nodes that we look into now. The linear ordering of the levels in our data structure means that there may be a variable assigned that an existential node must not depend on. We therefore need to *project* the assignment $\alpha_V$ to those variables in the node's dependency set. We define a function $prj_\exists : 2^{\mathcal{Y}} \times \mathcal{A}(V) \to \mathcal{A}(S)$ that maps variable assignments $\alpha_V$ to assignments of satisfaction variables $S$ such that $s_i$ is set to true if, and only if, some literal $l \in C_i^{<}$ is assigned positively by $\alpha_V$. Thus, the projection function

**Algorithm 6.3** Algorithms for solving existential and universal nodes

```
1:  procedure SOLVE∃(N ≡ ⟨∃, Y, H⟩)
2:      β_Y ← CHECKCONSISTENCY(α_V)
3:      α_S ← prj∃(Y, α_V)
4:      match SAT(θ_Y, β_Y ⊔ α_S) as
5:          Unsat(β) ⇒
6:              return REFINE(unsat, β|_S, N)
7:          Sat(α) ⇒ update α_V with α|_Y
8:              if node is max. element then
9:                  return REFINE(sat, α_S, N)
10:             end if
11:             return CandidateFound
12: end procedure
13: procedure SOLVE∀(N ≡ ⟨∀, X⟩)
14:     β_S ← prj∀(X, α_V)
15:     match SAT(θ_X, β_S) as
16:         Unsat(β) ⇒
17:             return REFINE(sat, β|_S, N)
18:         Sat(α) ⇒ update α_V with α|_X
19:             return CandidateFound
20: end procedure
```

only considers actual dependencies of $\langle \exists, Y, H \rangle$:

$$prj_\exists(Y, \alpha_V)(s_i) = \begin{cases} \mathbf{T} & \text{if } \alpha_V \vDash C_i^< \\ \mathbf{F} & \text{otherwise} \end{cases}$$

For our running example, at node $\langle \exists, \{y_2\}, \{x_2\} \rangle$, the projection for the first clause $C_1 = (x_1 \vee \overline{x_2} \vee \overline{y_2} \vee y_3)$ is $prj_\exists(\{y_2\}, \overline{x_1}\boldsymbol{x_2})(s_1) = prj_\exists(\{y_2\}, x_1\boldsymbol{x_2})(s_1) = \mathbf{F}$ and $prj_\exists(\{y_2\}, \overline{x_1}\boldsymbol{\overline{x_2}})(s_1) = prj_\exists(\{y_2\}, x_1\boldsymbol{\overline{x_2}})(s_1) = \mathbf{T}$ because $C_1^< = (\overline{x_2})$.

If the SAT solver returns a candidate assignment at the maximal existential node (i.e., the node on innermost level), we know that all clauses have been satisfied, and we have therefore refuted the candidate assignment of some universal node. This is handled by calling REFINE in line 9. For existential nodes we additionally have to check for consistency, which we discuss in Section 6.3.5 (called in line 2).

**Solving Universal Nodes.** Similar to the projection for existential nodes, we need an (almost symmetric) projection for universal nodes (line 14). It has to differ slightly from $prj_\exists$, because we use just one set of variables $S$ for universal nodes. A universal quantifier cannot falsify the clause if it is already satisfied.

$$prj_\forall(X, \alpha_V)(s_i) = \begin{cases} \mathbf{T} & \text{if } \alpha_V \vDash C_i|_{bound\langle\forall, X\rangle} \\ \bot & \text{otherwise} \end{cases}$$

### 6.3.4 Refinement

Algorithm REFINE in Algorithm 6.4 is called whenever there is a conflict, i.e. whenever it is clear that $\alpha_V$ satisfies the formula (*sat* conflict) or violates it (*unsat* conflict). In case there is an unsat conflict at an existential node, we build the (universally reduced) conflict clause from $\beta_S$ (as explained in Section 3.2) in line 3. If the clause is fork-free, we can apply the standard refinement for clausal abstraction (see also Section 2.3) with the exception that we need to find the unique refinement node first (line 10). This backward search

**Algorithm 6.4** Refinement algorithm that eliminates dependency forks

```
1:  procedure Refine(res, β_S, node)
2:      if res = unsat then
3:          C_conflict = ⋃_{s_i ∈ β_S^F} C_i|_{bound(node)}        ▷ C_conflict is universally reduced
4:          if C_conflict contains dependency fork then
5:              fork elimination ⇒ add clauses and variables, update abstractions θ
6:              resetConsistency for all nodes
7:              return Conflict(lvl = 0)
8:          end if
9:      end if
10:     if next ← DetermineRefinementNode(res, β_S, node.level) then
11:         return Conflict(next.level)
12:     else                                                    ▷ conflict at outermost ∃/∀ node
13:         return Result(res)
14:     end if
15: end procedure
```

over the quantifier levels is shown in Algorithm 6.5. For an *unsat* conflict, we traverse the levels backwards until we find an existential node that binds a variable contained in the conflict clause.

Because the conflict clause is fork-free, the target node of the traversal is unique. For a *sat* conflict, we do the same for universal nodes but the uniqueness comes from the fact that universal levels are singletons. We then add the refinement clause to the SAT solver at the corresponding node (Algorithm 6.5, lines 6 and 12) and proceed. For *sat* conflicts, we have to additionally learn consistency requirements at existential nodes (line 18) that make sure that the node produces the same result if the assignment (restricted to the dependencies of that node) repeats. In case the conflict propagated beyond the root node, we terminate with the given result.

**Fork Extension.** In case that the conflict clause contains a fork, we apply Fork Extension as described in Section 6.2. After applying Fork Extension, we encode the newly created clauses and variables within their respective nodes. We update the abstractions with those fresh variables and clauses as for the initial abstraction discussed in Section 6.3.2. Additionally, we reset learned Skolem functions as they may be invalidated by the refinement (Algorithm 6.4, line 6).

### 6.3.5 Consistency Requirements

The algorithm described so far produces correct refutations in case the DQBF is false. For positive results, the *consistency* of Skolem functions of *incomparable* existential variables may be violated. Consider for example the formula $\forall x_1 \forall x_2.\, \exists y_1(x_1).\, \exists y_2(x_2).\, \exists y_3(x_1, x_2).\, \varphi$ and assume that for the assignment $\overline{x_1}\overline{x_2}$, there is a corresponding satisfying assignment $\overline{y_1}\overline{y_2}\overline{y_3}$. If the next assignment is $\overline{x_1}\boldsymbol{x_2}$,

**Algorithm 6.5** Backward search algorithm to determine refinement node

```
1:  procedure DetermineRefinementNode(res, β_S, lvl)
2:      while lvl ≥ 0 do
3:          if res = unsat and lvl is existential then
4:              for node ⟨∃, Y, H⟩ in levels[lvl] do        ▷ check if Y is in conflict clause
5:                  if there exists some s_i ∈ β_S^F such that C_i|_Y ≠ ∅ then
6:                      θ_Y ← θ_Y ∧ ⋁_{s_i ∈ β_S^F} ā_i                    ▷ refine abstraction
7:                      return ⟨∃, Y, H⟩
8:                  end if
9:              end for
10:         else if res = sat and lvl is universal with node ⟨∀, X⟩ then
11:             if there exists some s_i ∈ β_S^T such that C_i|_X ≠ ∅ then
12:                 θ_X ← θ_X ∧ ⋁_{s_i ∈ β_S^T} s̄_i                        ▷ refine abstraction
13:                 return ⟨∀, X⟩
14:             end if
15:         else if res = sat and lvl is existential then    ▷ add consistency requirements
16:             for N = ⟨∃, Y, H⟩ in levels[lvl] do
17:                 if H ⊂ bound_∀(N) then
18:                     learnEntry(N)
19:                 end if
20:                 set β_S(s_i) = ⊥ if α_Y ⊨ C_i and β_S(s_i) = T for all s_i ∈ S
21:             end for
22:         end if
23:         lvl ← lvl − 1
24:     end while
25:     return res
26: end procedure
```

then the assignment to $y_1$ has to be the same as before ($y_1 \to \mathbf{F}$) as the value of its sole dependency $x_1$ is unchanged.

We enhance the certification capabilities of clausal abstraction (see Section 2.4) to build consistency requirements that represent partial Skolem functions in our algorithm during solving. We incrementally build a list of *entries*, where the first component in an entry is a propositional formula over the dependencies and the second component is the corresponding assignment $\alpha_Y$. Before generating a candidate assignment in SOLVE$_\exists$, we call CHECKCONSISTENCY (Algorithm 6.6) to check if the assignment $\alpha_Y$ for the given assignment $\alpha_V$ of dependencies is already determined, by iterating through the learned entries (Algorithm 6.6, lines 2–3). If it is the case, we get an assignment $\alpha_Y$ that is then assumed for the candidate generation. Note that in this case, the SAT call in line 4 of SOLVE$_\exists$ is guaranteed to return Sat (we already verified this assignment, otherwise it would not have been learned). Further, consistency requirements are only needed for existential nodes $\langle \exists, Y, H \rangle$ with $H \subset bound_\forall(\langle \exists, Y, H \rangle)$, i.e., that observe an over-approximation

**Algorithm 6.6** Algorithms for handling consistency requirements

```
1:  procedure checkConsistency(α_Y)
2:      for (cond, α_Y) in entries do
3:          if SAT(cond, α_Y) is Sat then return α_Y
4:          end if
5:      end for
6:      return empty assignment
7:  end procedure
8:  procedure resetConsistency
9:      entries ← [ ]
10:     reset learned clauses at universal nodes
11: end procedure
12: procedure learnEntry(node ≡ ⟨∃, Y, H⟩)
13:     let α_S and α_Y be from line 7 of Algorithm 6.3.
14:     entries.push((⋀_{s_i ∈ α_S^T} C_i^<, α_Y))
15: end procedure
```

of their dependency set. For those nodes, the consistency requirements enforce that whenever two assignments of the dependencies are equal, the assignment of $\alpha_Y$ returns the same value as well. We call resetConsistency (Algorithm 6.6) to reset the consistency requirements in case we applied Fork Extension (Algorithm 6.4, line 6) as the new clauses may affect already learned parts of the function. We, further, have to reset the clauses learned at universal nodes (Algorithm 6.6, line 10).

We *learn* a new consistency requirement by calling learnEntry (Algorithm 6.6) on the backward search on sat conflicts, that is in line 18 in Algorithm 6.5. When we determine the refinement node for sat conflicts, we call learnEntry in every existential node $\langle \exists, Y, H \rangle$ with $H \subset bound_\forall(\langle \exists, Y, H \rangle)$ on the path to that node. In our example, when the base case of $\langle \exists, \{y_3\}, \{x_1, x_2\} \rangle$ returns (all clauses are satisfied, line 9 in Algorithm 6.3), we add consistency requirements at nodes $\langle \exists, \{y_2\}, \{x_2\} \rangle$ and $\langle \exists, \{y_1\}, \{x_1\} \rangle$ before refining at $\langle \forall, \{x_1, x_2\} \rangle$. In addition to the learning, we modify the witnesses by projecting away entries that are satisfied by some assignment $\alpha_Y$ for some existential node $\langle \exists, Y, H \rangle$ in line 20.

### 6.3.6 Example

We consider a possible execution of the presented algorithm on our running example. For the sake of readability, we combine unimportant steps and focus on the interesting cases. Assume the following initial assignment $\alpha_1 = x_1\overline{x_2}\,\overline{y_1}\,\overline{y_2}$ before node $N_{max} \equiv \langle \exists, \{y_3\}, \{x_1, x_2\} \rangle$. The result of projecting function $prj_\exists(\{y_3\}, \alpha_1)$ is $s_1\overline{s_2}s_3\overline{s_4}s_5$ and the SAT solver (Algorithm 6.3, line 4) returns Unsat$(\alpha_1')$ with core $\alpha_1' = \overline{s_2}\,\overline{s_4}$ as there is impossible to satisfy both clauses $(s_2 \vee y_3)$ and $(s_4 \vee \overline{y_3})$ of the abstraction. The refinement algorithm (Algorithm 6.4) builds the conflict clause $C_{conflict} = C_2|_{bound(N_3)} \cup C_4|_{bound(N_3)} = (\overline{x_1} \vee y_2 \vee y_1)$ at line 3 which contains a dependency fork between $y_1$ and $y_2$. We have

already seen in Example 6.2 that the fork can be eliminated resulting in fresh variable $y_4$ with $dep(y_4) = \varnothing$ and the clauses 6 and 7 $(\overline{x_1} \vee y_1 \vee y_4)(\overline{y_4} \vee y_2)$.

Now, the root node contains variable $y_4$, for which we assume assignment $\{y_4 \mapsto \mathbf{T}\}$. For the same universal assignment as before ($x_1\overline{x_2}$), the assignment of $y_2$ has to change to $\{y_2 \mapsto \mathbf{T}\}$ due to the newly added clause 7, leading to $\alpha_2 = x_1\overline{x_2}\,\overline{y_1}y_2y_4$ before node $N_{max}$. The only unsatisfied clause is $C_4$ which can be satisfied using $\{y_3 \mapsto \mathbf{F}\}$, leading to the base case (Algorithm 6.3, line 9). During refinement, we learn Skolem function entries $(x_1 \wedge y_4, \overline{y_1})$ and $(\overline{x_2} \wedge y_4, y_2)$ at nodes $\langle \exists, \{y_1\}, \{x_1\}\rangle$ and $\langle \exists, \{y_2\}, \{x_2\}\rangle$ as $prj_\exists(\{y_1\}, x_1\overline{x_2})$ and $prj_\exists(\{y_2\}, x_1\overline{x_2})$ assign $s_1, s_5, s_6$ and $s_1, s_6$ positively, respectively.

For the following universal assignment $\overline{x_1}\overline{x_2}$, the value of $y_2$ is already determined by the consistency requirements (Algorithm 6.3, line 2) to be positive. There is a continuation of the algorithm without further unsat conflict, determining that the instance is true.

## 6.4 Correctness

In this section, we give a formal correctness proof of the algorithm. For soundness, the algorithm has to guarantee that existential variables are assigned consistently, that is for an existential variable $y$ with dependency $dep(y)$ it holds that $f_y(\alpha) = f_y(\alpha')$ if $\alpha|_{dep(y)} = \alpha'|_{dep(y)}$ for every $\alpha$ and $\alpha'$. Our algorithm maintains this property at every point during the execution by a combination of over-approximation and consistency requirements. Completeness relies on the fact that the underlying proof system is refutationally complete for DQBF. Progress is guaranteed as there are only finitely many different conflict clauses and, thus, only finitely many Skolem function resets. We start by giving the correctness arguments for the base case and state theorems over the structure of the abstractions. Then, we split the actual correctness proof into two theorems that argue inductively over the structure of the quantifier levels. The argumentation in this section roughly follows the structure of the correctness proof for the QBF algorithm given in Section 2.3.2.

The first lemma states the base case, i.e., that the abstraction for the maximal element is equisatisfiable to replacing the assignment of the bound variables $\alpha_V$ in the matrix $\varphi$.

**Lemma 6.12.** *Let $\langle \exists, Y, H\rangle$ be the existential node corresponding to the unique maximal element and let $\alpha_V$ be some assignment with $V = bound(\langle \exists, Y, H\rangle)$. Then, the SAT call (line 4) of SOLVE$_\exists(\langle \exists, Y, H\rangle)$ returns* Sat *if, and only if, $\varphi[\alpha_V]$ is satisfiable.*

*Proof.* The abstraction $\theta_Y$ for the maximal element does not contain assumption literals, i.e., it has the form $\theta_Y = \bigwedge_{C_i \in \varphi} s_i \vee C_i^=$. By definition of $\alpha_S = prj_\exists(Y, \alpha_V)$, it holds that $\theta_Y[\alpha_S] = \bigwedge_{\substack{C_i \in \varphi \\ \alpha_V \not\models C_i^<}} C_i^= = \varphi[\alpha_V]$. □

Additionally, for non-maximal nodes, we state two lemmata that describe the effect of assignments of satisfaction variables on the existential and universal abstractions.

**Lemma 6.13.** *Let $\langle \exists, Y, H \rangle$ be an existential node and let $\alpha_S$ be some assignment of the satisfaction variables. It holds that $\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} (C_i^{=} \vee a_i)$.*

*Proof.* The abstraction $\theta_Y$ for the existential node $\langle \exists, Y, H \rangle$ has the form $\theta_Y = \bigwedge_{C_i \in \varphi} (a_i \vee s_i \vee C_i^{=})$. It follows immediately that $\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} (C_i^{=} \vee a_i)$. □

**Lemma 6.14.** *Let $\langle \forall, X \rangle$ be an universal node and let $\beta_S$ be some positive assignment of the satisfaction variables (i.e., a partial assignment containing only positive values). It holds that $\theta_X[\beta_S] = \bigwedge_{C_i \in \varphi, \beta_S(s_i) \neq \mathbf{T}} (s_i \vee \neg C_i^{=})$.*

*Proof.* The abstraction $\theta_X$ for the a universal node $\langle \forall, X \rangle$ has the form $\theta_X = \bigwedge_{C_i \in \varphi} (s_i \vee \neg C_i^{=})$. It follows immediately that $\theta_X[\beta_S] = \bigwedge_{C_i \in \varphi, \beta_S(s_i) \neq \mathbf{T}} (s_i \vee \neg C_i^{=})$. □

The following lemmata state that refinements are correct, i.e., that the clause contained in the refinement is satisfied, respectively, falsified.

**Lemma 6.15.** *Let $\langle \exists, Y, H \rangle$ be some existential node and let $\alpha_V$ be some assignment with $V = bound(\langle \exists, Y, H \rangle)$. Let $\alpha$ be the assignment after a satisfiable call to the abstraction $\theta_Y$ (line 4 of SOLVE$_\exists$($\langle \exists, Y, H \rangle$)). For every clause $C_i \in \varphi$ it holds that $a_i \mapsto \mathbf{F}$ implies that $\alpha_Y \dot\cup \alpha_V \vDash C_i$.*

*Proof.* Follows by the abstraction definitions and the projection functions. □

**Lemma 6.16.** *Let $\langle \forall, X \rangle$ be some universal node and let $\alpha_V$ be some assignment with $V = bound(\langle \forall, X \rangle)$. Let $\alpha$ be the assignment after a satisfiable call to the abstraction $\theta_X$ (line 15 of SOLVE$_\forall$($\langle \forall, X \rangle$)). For every clause $C_i \in \varphi$ it holds that $s_i \mapsto \mathbf{F}$ implies that $\alpha_X \dot\cup \alpha_V \nvDash C_i$.*

*Proof.* Follows by the abstraction definitions and the projection functions. □

The proof of correctness is an inductive argument over the quantifier levels. Fix some level *lvl* and an assignment of the variables bound before *lvl*, the algorithm determines the result of the DQBF where the prior bound variables are replaced by this assignment. The algorithm, further, determines a subset of the satisfied clauses as a witness for the outer levels.

We define an operator $\Phi|_{\alpha_S}^{lvl}$ that restricts the matrix $\varphi$ in a DQBF $\Phi$ to those clauses $C_i \in \varphi$ such that $\alpha_S(s_i) = \mathbf{F}$, i.e., the resulting DQBF has the same quantifier prefix from quantifier level *lvl* onwards with matrix $\varphi' \coloneqq \{C_i^{\geq} \mid C_i \in \varphi \wedge \alpha_S(s_i) = \mathbf{F}\}$. Variables that are bound by a smaller quantifier level than *lvl* are removed from the matrix. Intuitively, the operator removes clauses marked as satisfied by $\alpha_S$.

**Lemma 6.17.** *Let $\Phi$ be a DQBF with matrix $\varphi$, let lvl be a quantifier level, and let $\alpha_V$ be an assignment of variables bound prior to lvl. If $\Phi[\alpha_V]$ is true SolveLevel(lvl) produces a sat conflict with partial assignment $\beta_S$ such that $\Phi|_{\beta_S[\bot \mapsto \mathbf{F}]}^{lvl}$ is true.*

*Proof.* We prove the statement by induction over the quantifier levels.

*Induction Base.* Let $lvl$ be the quantifier level with the unique maximal node $N_{max} = \langle \exists, Y, H \rangle$ (see Proposition 6.11.5) and let $\alpha_V$ be such that $\Phi[\alpha_V]$ is true. By Lemma 6.12, the truth of $\Phi[\alpha_V]$ witnesses the satisfiability of $\theta_Y[\alpha_S]$ where $\alpha_S = prj(Y, \alpha_V)$. As $N_{max}$ is maximal, the algorithm SOLVE$_\exists$ calls REFINE($sat$, $\alpha_S$, $N_{max}$) and $\alpha_S[\bot \mapsto \mathbf{F}]$ is equivalent to $\alpha_S$ satisfying the second condition due to the definition of $prj_\exists$.

*Induction Step ($\mathcal{Q} = \exists$).* Let $lvl$ be an existential quantifier level and let $\alpha_V$ be such that $\Phi[\alpha_V]$ is true. Let $\langle \exists, Y, H \rangle$ be an arbitrary existential node in $lvl$. Further, let $\alpha_S = prj_\exists(Y, \alpha_V)$. By Lemma 6.13 it holds that

$$\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^{\mathbf{F}}} (C_i^{=} \vee a_i) \quad .$$

Since $\Phi[\alpha_V]$ and thereby $\Phi|^{lvl}_{\alpha_S}$ is true, there is a satisfying assignment $\alpha_Y$ for the variables $Y$ such that $\Phi|^{lvl}_{\alpha_S}[\alpha_Y]$ is true. Define $\alpha_A^*$ as $\alpha_A^*(a_i) = \mathbf{F}$ if, and only if, $\alpha_V \dot\cup \alpha_Y \vDash C_i^{\leq}$. Thus, $\alpha_A^*$ is the minimal assignment with respect to the number of assumptions ($\alpha_A^*(a_i) = \mathbf{T}$) for the given assignment $\alpha_Y$. The combined assignment $\alpha_X \dot\cup \alpha_A^*$ is a satisfying assignment of the initial abstraction $\theta_Y[\alpha_S]$ by construction. Thus, for every node $N$ in $lvl$, SOLVE$_\exists$ returns CandidateFound and the algorithm continues to the next quantifier level. We perform a case distinction on the assignments created on $lvl$, i.e., returned by the SAT solver in line 7. As $\Phi[\alpha_V]$ is true, there is a satisfying assignment $\alpha_Y^*$ for every node $\langle \exists, Y, H \rangle$ in $lvl$. Let $\alpha_Y^\cup$ be the combined assignment of every existential node in this level.

Assume that the SAT solver in line 7 returns this assignment. Thus, $\Phi[\alpha_V \dot\cup \alpha_Y^\cup]$ is true. By induction hypothesis we deduce that the next level produces a sat conflict with partial assignment $\beta_S$ such that $\Phi|^{lvl+1}_{\beta_S[\bot \mapsto \mathbf{F}]}$ is true, i.e., the assignment $\beta_S$ represents those clauses that need to be satisfied such that $\Phi$ is true. Since $lvl$ is existential, this witness is propagated. During this propagation, we adapt the witness by removing entries satisfied by the assignment $\alpha_Y^\cup$ of this quantifier level (line 20).

Assume that the SAT solver in line 7 returns a different assignment. If the assignment is still satisfying $\Phi$, the next level returns a sat conflict and the same argumentation as above applies. In the case the next level returns a unsat conflict with witness $\beta_S'$ there are 3 possibilities:

1. The conflict does not contain variables of any existential node, which immediately contradicts that $\Phi[\alpha_V]$ is true.
2. The conflict contains variables of a single existential node $\langle \exists, Y, H \rangle$. The subsequent refinement in line 6 of Algorithm 6.4 requires that one of the not satisfied clauses $C_i$ with $\beta_S'(s_i) = \mathbf{F}$ has to be satisfied in the next iteration and the corresponding refinement clause is $\psi \coloneqq \bigvee_{s_i \in \beta_S'^{\mathbf{F}}} \overline{a_i}$. By construction of $\alpha_A^*$ as the minimal assignment corresponding to $\alpha_Y$, $\alpha_A^* \nvDash \psi$ contradicts that $\alpha_Y$ is a satisfying assignment of $\Phi[\alpha_V]$. Hence, $\alpha_Y \dot\cup \alpha_A^*$ is still a satisfying assignment for the refined abstraction $\theta_Y'[\alpha_S]$. The refinement also reduces the number of $A$ assignments by at least 1 and, thus, brings us one step closer to termination.
3. The conflict contains variables of more than one existential node, thus, the conflict clause $C_{conflict}$ in line 3 of Algorithm 6.4 contains a dependency fork. It holds

that $\alpha_V \dot\cup \alpha_Y^\cup \not\models C_{conflict}$ as established in Section 3.2, where $\alpha_Y^\cup$ is the combined assignment of *lvl*. Applying Fork Extension gives us new clauses without dependency fork. There exists an assignment $\alpha'_V$ with $\alpha_V \sqsubseteq \alpha'_V$ such that the new DQBF $\Phi[\alpha'_V]$ is still true (where $\alpha'_V$ is the assignment $\alpha_V$ plus added assignment for the new variables due to Fork Extension). Unlike before, $\alpha_Y^\cup$ does no longer satisfy the abstraction, thus, a different assignment is produced.

In all possible cases, eventually, the satisfying assignment is reached.

*Induction Step ($\mathcal{Q} = \forall$).* Let *lvl* be a universal quantifier level with the singleton node $\langle \forall, X\rangle$ and let $\alpha_V$ be such that $\Phi[\alpha_V]$ is true. Further, let $\beta_S = prj_\forall(X, \alpha_V)$. For every assignment $\alpha_X$, it holds that $\Phi[\alpha_V \dot\cup \alpha_X]$ is true. By Lemma 6.14 it holds that

$$\theta_X[\beta_S] = \bigwedge_{C_i \in \varphi, \beta_S(s_i) \neq \mathbf{T}} (s_i \vee \neg C_i^=) \quad .$$

Thus, in order to set $s_i$ to false for some $i$, every literal $l \in C_i^=$ has to be assigned negatively. Fix some arbitrary assignment $\alpha_X$. By induction hypothesis, the following level produces a sat conflict with partial assignment $\beta'_S$ such that $\Phi|^{lvl+1}_{\beta'_S[\bot\mapsto\mathbf{F}]}$ is true. The subsequent refinement in line 12 of Algorithm 6.4 reduces the number of $S$ assignments, thus, eventually, the abstraction $\theta_X[\beta_S]$ becomes unsatisfiable. Let $\theta'_X$ be the abstraction at this point and let $\beta^*_S$ be the failed assumptions. $\beta^*_S \sqsubseteq \beta^+_S$ holds as $\beta^*_S$ are the failed assumptions of the SAT call $\textsc{sat}(\theta'_X, \beta_S)$.

It remains to show that $\Phi[\alpha_V]|^{lvl}_{\beta^*_S[\bot\mapsto\mathbf{F}]}$ is true. Assume for contradiction that there is some $\alpha_X$ such that $(\Phi[\alpha_V]|^{lvl}_{\beta^*_S[\bot\mapsto\mathbf{F}]})[\alpha_X]$ is false. Let $\beta_S = prj_\forall(X, \alpha_V)$ We know that $\theta'_X[\alpha_X \dot\cup \beta_S]$ is unsatisfiable. Thus, the assignment $\alpha_X$ was excluded due to refinements. As the refinement only excludes $S$ assignments $\beta''_S$ such that $\Phi|^{lvl+1}_{\beta''_S[\bot\mapsto\mathbf{F}]}$ is true, this leads to a contradiction. □

**Lemma 6.18.** *Let $\Phi$ be a DQBF with matrix $\varphi$, let lvl be a quantifier level, and let $\alpha_V$ be an assignment of variables bound prior to lvl. If $\Phi[\alpha_V]$ is false SolveLevel(lvl) produces a unsat conflict with partial assignment $\beta_S$ such that $\Phi|^{lvl}_{\beta_S[\bot\mapsto\mathbf{T}]}$ is false.*

*Proof.* We prove the statement by induction over the quantifier levels.

*Induction Base.* Let *lvl* be the quantifier level with the unique maximal node $N_{max} = \langle \exists, Y, H\rangle$ (see Proposition 6.11.5) and let $\alpha_V$ be such that $\Phi[\alpha_V]$ is false. By Lemma 6.12, $\theta_Y[\alpha_S]$ is unsatisfiable where $\alpha_S = prj(Y, \alpha_V)$. Let $\beta'_S$ be the failed assumptions from the sat call to $\textsc{sat}(\theta_Y, \alpha_S)$, i.e., $\beta'_S \sqsubseteq \alpha_S$ and $\theta_Y[\beta'_S]$ is unsatisfiable. Due to the definition of the abstraction, $\Phi|^{lvl}_{\beta'_S[\bot\mapsto\mathbf{T}]}$ is false.

*Induction Step ($\mathcal{Q} = \exists$).* Let *lvl* be an existential quantifier level and let $\alpha_V$ be such that $\Phi[\alpha_V]$ is false. Let $\langle \exists, Y, H\rangle$ be an arbitrary existential node in *lvl*. Further, let $\alpha_S = prj_\exists(Y, \alpha_V)$. By Lemma 6.13 it holds that

$$\theta_Y[\alpha_S] = \bigwedge_{s_i \in \alpha_S^\mathbf{F}} (C_i^= \vee a_i) \quad .$$

As $\Phi[\alpha_V]$ is false, every assignment of the existential level is false as well. There are two possible executions:

- Assume that all existential nodes generate a candidate assignment, then we can apply the induction hypothesis to deduce that the next level produces a unsat conflict with partial assignment $\beta'_S$ such that $\Phi|^{lvl+1}_{\beta'_S[\bot\mapsto\mathbf{T}]}$ is false. The refinement with witness $\beta'_S$ has three possibilities:
  1. The conflict does not contain variables of any existential node, that is, the algorithm produces the partial assignment $\beta'_S$.
  2. The conflict contains variables of a single existential node $\langle\exists, Y, H\rangle$. The subsequent refinement in line 6 of Algorithm 6.4 requires that one of the not satisfied clauses $C_i$ with $\beta'_S(s_i) = \mathbf{F}$ has to be satisfied in the next iteration and the corresponding refinement clause is $\psi \coloneqq \bigvee_{s_i\in\beta'_S{}^{\mathbf{F}}} \overline{a_i}$. The refinement reduces the number of $A$ assignments by at least 1 and, thus, brings us one step closer to termination.
  3. The conflict contains variables of more than one existential node, thus, the conflict clause $C_{conflict}$ in line 3 of Algorithm 6.4 contains a dependency fork. It holds that $\alpha_V \dot\cup \alpha^{\cup}_Y \not\models C_{conflict}$, where $\alpha^{\cup}_Y$ is the combined assignment of $lvl$. Applying Fork Extension gives us new clauses without dependency fork. In the following, a different assignment is produced.

  In the latter two cases, we make progress towards termination.
- Assume that one of the existential nodes $\langle\exists, Y, H\rangle$ produce a unsat conflict. Let $\theta'_Y$ be the abstraction at this point and let $\beta'_S$ be the failed assumptions, i.e., $\beta'_S \sqsubseteq \alpha_S$. Let $\alpha''_S = \beta'_S[\bot \mapsto \mathbf{T}]$. It remains to show that $\Phi|^{lvl}_{\alpha''_S}$ is false. Assume for contradiction that there is some $\alpha_Y$ such that $\Phi|^{lvl}_{\alpha''_S}[\alpha_Y]$ is true. It holds that $\theta'_Y[\alpha_Y \dot\cup \alpha''_S]$ is unsatisfiable, whereas initially, $\theta_Y[\alpha_Y \dot\cup \alpha''_S]$ is satisfiable. Thus, the assignment $\alpha_Y$ was excluded due to refinements. As the refinement only excludes assignments corresponding to some $S$ assignment $\beta^*_S$ such that $\Phi|^{lvl}_{\beta^*_S[\bot\mapsto\mathbf{T}]}$ is false, this contradicts our assumption.

*Induction Step ($\mathcal{Q} = \forall$).* Let $lvl$ be a universal quantifier level with the singleton node $\langle\forall, Y\rangle$ and let $\alpha_V$ be such that $\Phi[\alpha_V]$ is false. Further, let $\beta_S = prj_\forall(X, \alpha_V)$. There is some assignment $\alpha_X$ such that that $\Phi[\alpha_V \dot\cup \alpha_X]$ is false. By Lemma 6.14 it holds that

$$\theta_X[\beta_S] = \bigwedge_{C_i\in\varphi, \beta_S(s_i)\neq\mathbf{T}} (s_i \vee \neg C^{=}_i) \ .$$

$\theta_X[\beta_S]$ is initially satisfiable by construction. Given $\alpha_X$ from , we define the optimal corresponding assignment $\beta^*_S$ as $\beta^*_S(s_i) = \mathbf{T}$ if, and only if, either $\beta_S(s_i) = \mathbf{T}$ or $\beta_X \models C^{=}_i$. Assume that the SAT solver in line 15 of Algorithm 6.3 returns the assignment $\alpha_X$. Thus, by induction hypothesis, the next level produces a unsat conflict with partial assignment $\beta'_S$ such that $\Phi|^{lvl+1}_{\beta'_S[\bot\mapsto\mathbf{T}]}$ is false.

Assume that the SAT solver in line 15 of Algorithm 6.3 returns a different assignment $\alpha'_X$. If $\Phi[\alpha_V \dot\cup \alpha'_X]$ is false, the same argumentation as above applies. If this is not the case,

Table 6.1: Number of instances solved within 10 min. For every solver, we give the number of solved instances overall (#) and broken down by satisfiable (**T**), unsatisfiable (**F**), and uniquely solved instances (∗).

| Benchmark | # | dCAQE | | | | iDQ | | | | HQS | | | | iProver | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | # | **T** | **F** | ∗ | # | **T** | **F** | ∗ | # | **T** | **F** | ∗ | # | **T** | **F** | ∗ |
| PEC1 [FT14b] | 1000 | **839** | 7 | **832** | **224** | 37 | 0 | 37 | 0 | 636 | **10** | 626 | 32 | 71 | 0 | 71 | 0 |
| PEC2 [Git+15] | 720 | 342 | 71 | 271 | 12 | 214 | 45 | 169 | 0 | **401** | **104** | **297** | **60** | 288 | 60 | 228 | 0 |
| BoSy [Fay+17] | 1216 | **1006** | **389** | **617** | **66** | 924 | 335 | 589 | 2 | 735 | 231 | 504 | 0 | 946 | 370 | 576 | 20 |
| | 2936 | **2187** | | | | 1175 | | | | 1772 | | | | 1305 | | | |

the next level produces a sat conflict with partial assignment $\beta'_S$ such that $\Phi|^{lvl+1}_{\beta'_S[\perp\mapsto\mathbf{F}]}$ is true. Subsequently, $\theta_X$ is refined by adding the the clause $\psi \coloneqq \bigvee_{s_i\in\beta'_S \mathrm{T}} \overline{s_i}$. By construction of $\beta^*_S$ as the optimal assignment corresponding to $\alpha_X$, we deduce that $\beta^*_S \not\models \psi$ contradicts that $\alpha_X$ is a witness that $\Phi[\alpha_V]$ is false. Thus, $\alpha_X \dot\cup \beta^*_S$ remains a satisfying assignment of the refined abstraction. The refinement reduced the number of $S$ assignments and, thus, the falsifying assignment $\alpha_X$ is reached eventually. □

From Lemma 6.17 and 6.18 we can conclude the faithful replication of the DQBF semantics by the lifted clausal abstraction algorithm.

**Theorem 6.19.** *Solve(Φ) returns sat if, and only if, Φ is satisfiable.*

## 6.5 Evaluation

We compare our prototype implementation, called dCAQE[3], against the publicly available DQBF solvers, iDQ [Frö+14], HQS [Git+15], and iProver [Kor08]. We ran the experiments on machines with a 3.6 GHz quad-core Xeon (E3-1271 v3) processor with 32 GB of memory. Timeout and memout were set to 10 minutes and 8 GB, respectively. We used the DQBF preprocessor HQSPre [Wim+17] for every solver except HQS. We evaluate our solver on the DQBF case studies regarding reactive synthesis [Fay+17] (described in Section 7.3) and the partial equivalence checking problem (PEC), already introduced in Chapter 5, using the benchmark sets PEC1 [FT14b] and PEC2 [Git+15].

The second case study (BoSy) considers the problem of synthesizing sequential circuits from specifications given in linear-time temporal logic (LTL) [Fay+17]. The benchmarks were created using the tool BoSy [FFT17] and the LTL benchmarks from the Reactive Synthesis Competition [Jac+16; Jac+17a]. Each formula encodes the existence of a sequential circuit that satisfies the LTL specification.

The results are presented in Table 6.1. The PEC instances contain over-proportionally many unsatisfiable instances, and we conjecture that the differences between dCAQE and iDQ/iProver can be explained by the effectiveness of the resolution-based refutations that dCAQE is based on. HQS performs well on those benchmarks as well, which

[3] Available at `https://github.com/ltentrup/caqe`.

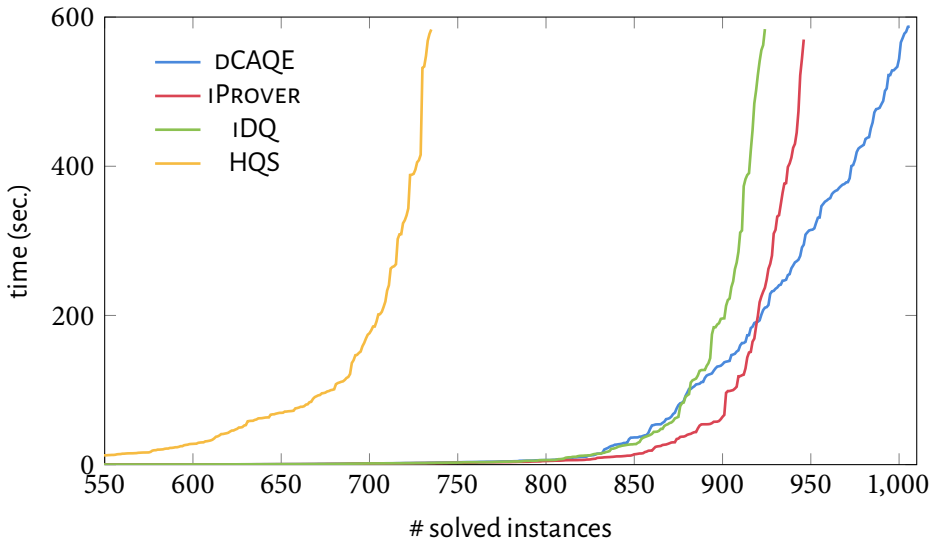



Figure 6.4: Number of solved instances within 10 minutes among the 1216 instances from the BoSy benchmark set.

could be due to the fact that it implements the fast refutation technique presented in Chapter 5 that was introduced alongside the benchmark set PEC1. The reactive synthesis benchmark set is were DCAQE excels. The benchmark set contains many easily solvable benchmarks, indicated by the high number of instances that are commonly solved by all solvers. However, there are also a fair amount of hard instances, and DCAQE solves significantly more of those than any other solver. Further, we can see the effect mentioned in the introduction of the infeasibility of expansion-based methods, as shown by the result of HQS. The cactus plot given in Figure 6.4 shows that DCAQE makes more progress, especially with a more substantial runtime where the other solvers solve very few instances after 100s. These results give rise to the hope that the scalability of more expressive synthesis approaches [FT15; Fin+18a; Coe+19] can be improved by employing DQBF solving.

## 6.6 Summary

We lifted the clausal abstraction algorithm to DQBF. This algorithm is the first to use the new Fork Resolution proof system, and it significantly increases the performance of DQBF solving on synthesis benchmarks. In particular, in the light of the past attempts to define search algorithms [FKB12] (which are closely related to clausal abstraction) for DQBF, this is a surprising success. It appears that the Fork Extension proof rule was the missing piece in the puzzle to build search/abstraction algorithms for DQBF.

# Part II

# Reactive Synthesis

# Chapter 7

# Synthesizing Reactive Systems

In the previous part, we showed how to solve the quantified satisfiability problem for linear and branching quantifiers. We now turn our attention towards applying those insights to solve the realizability problem for reactive systems using specifications in linear-time temporal logic. There has been a recent surge of new algorithms and tools for the synthesis of reactive systems from temporal specifications [Job+07; Ehl11; Boh+12; FS13; FT14a; MC18; MSL18]. Roughly, these approaches can be classified into two categories: *game-based synthesis*, whose origins date back to the seminal paper by Büchi and Landweber [BL69], translates the specification into a deterministic automaton and subsequently determines the winner in a game played on the state graph of this automaton; *Safraless synthesis* [KV05] avoids the transformation of the specification into an equivalent deterministic automaton via Safra's determinization procedure by approximating the specification in a sequence of deterministic safety automata. A special case of the latter category is *bounded synthesis* [SF07; FS13], which constructs a constraint system that characterizes all implementations, up to a fixed bound on the size of the implementation, that satisfy the specification. While in game-based methods the synthesized implementations are often unnecessarily (and impractically) large (cf. [FJ12]), due to the fact that the deterministic automaton often contains many more states than are needed by the implementation, in bounded synthesis, one can ensure that the synthesized implementation is the smallest possible realization of the specification by iteratively increasing the bound.

In the original formulation [FS07; SF07] and many derived works [FS13; FJ12; KJB13b; KJB13a; Fin+18a; Coe+19] the constraint systems are built in a decidable first-order theory and solved using powerful SMT solver. In the standard encoding, both the states of the synthesized implementation and its inputs are enumerated explicitly [FS13]. In this section, we investigate whether the scalability of bounded synthesis can be improved by using propositional quantification, thus, effectively making the constraint system more "symbolic".

We reduce the bounded synthesis problem of linear-time temporal logic (LTL) to constraint systems given as Boolean formulas (SAT), quantified Boolean formulas (QBF), and dependency quantified Boolean formulas (DQBF). The reductions are landmarks on the spectrum of symbolic vs. explicit encodings. All encodings represent the synthesized im-

plementation in terms of its transition function, which identifies the successor state using the current state and the input, and additionally, in terms of an output function, which identifies the output signals using the current state and the input. Furthermore, the encoding contains an annotation function, which relates the states of the implementation to the states of a universal automaton representing the specification.

In the SAT encoding of the transition function, a separate Boolean variable is used for every combination of a source state, an input signal, and a target state. The encoding is thus explicit in both the state and the input. In the QBF encoding, we quantify universally over the inputs, so that the encoding becomes symbolic in the inputs while staying explicit in the states. Quantifying universally over the states, just like over the input signals, is not possible in QBF because the states occur twice in the transition function, as source and as target. Separate quantifiers over sources and targets would allow for models where, for example, the value of the output function differs, even though both the source state and the input are the same. In DQBF, we can avoid such artifacts and obtain a "fully symbolic" encoding in both the states and the input.

We evaluate the encodings systematically using benchmarks from the reactive synthesis competition (SYNTCOMP) [Jac+17b] and state-of-the-art solvers. Our empirical finding is that both the input-symbolic and state-symbolic encoding, perform better than the non-symbolic approach. This fits with our intuition that a more symbolic encoding provides opportunities for optimization in the solver. It turns out that the DQBF solver DCAQE form Chapter 6 is crucial to the performance of the state-symbolic encoding: There is even evidence that the DQBF approach scales better with the number of states then the QBF encoding.

This section is based on work published in the proceedings of TACAS [Fay+17] and CAV [FFT17].

## 7.1 Preliminaries

Let AP be a finite set of atomic propositions and let $\Sigma = 2^{\mathsf{AP}}$ be the corresponding alphabet. An infinite word $\sigma \in \Sigma^\omega$ is an infinite sequence of elements of $\Sigma$. Finite words $\sigma \in \Sigma^*$ are finite sequences. The length of $\sigma = \sigma_0\sigma_1\cdots\sigma_n \in \Sigma^*$ is $|\sigma| = n+1$. For infinite $\sigma \in \Sigma^\omega$, we define $|\sigma| = \infty$. Given some set $A \subseteq \mathsf{AP}$ of atomic propositions, we use $\boldsymbol{a} \in 2^A$ to denote an assignment of propositions, where $a \in \boldsymbol{a}$ and $a \notin \boldsymbol{a}$ means that $a \in A$ is assigned true and false, respectively.

### 7.1.1 Linear-time Temporal Logic

Linear-time temporal logic (LTL) [Pnu77] is a commonly used specification language for linear-time properties. The grammar of LTL is given by

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi,$$

where $p \in \mathsf{AP}$ is an atomic proposition. We use the standard Boolean abbreviations $\mathit{false} \coloneqq \neg\mathit{true}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, and $\varphi \leftrightarrow \psi = (\varphi \rightarrow$

$\psi) \wedge (\psi \rightarrow \varphi)$. In addition to the temporal operators next $\bigcirc\varphi$ and until $\varphi \,\mathcal{U}\, \psi$ we use the derived operators release $\varphi \,\mathcal{R}\, \psi = \neg(\neg\varphi \,\mathcal{U}\, \neg\psi)$, eventually $\Diamond\varphi \equiv true\,\mathcal{U}\,\varphi$, globally $\Box\varphi \equiv \neg\Diamond\neg\varphi$, and weak until $\varphi \,\mathcal{W}\, \psi \equiv \Box\varphi \vee (\varphi\mathcal{U}\psi)$. We define the satisfaction relation $\sigma, i \vDash \varphi$ for $\sigma \in \Sigma^\omega$ and $i \geq 0$ as

$$
\begin{array}{lll}
\sigma, i \vDash true & & \\
\sigma, i \vDash p & \text{iff} & p \in \sigma_i \\
\sigma, i \vDash \neg\varphi & \text{iff} & \sigma, i \nvDash \varphi \\
\sigma, i \vDash \varphi \vee \psi & \text{iff} & \sigma, i \vDash \varphi \text{ or } \sigma, i \vDash \psi \\
\sigma, i \vDash \bigcirc\varphi & \text{iff} & \sigma, i+1 \vDash \varphi \\
\sigma, i \vDash \varphi \,\mathcal{U}\, \psi & \text{iff} & \text{there is some } j \geq i \text{ with } \sigma, j \vDash \psi \\
 & & \text{and for all } k \text{ with } i \leq k < j \text{ it holds that } \sigma, k \vDash \varphi
\end{array}
$$

We write $\sigma \vDash \varphi$ as a shorthand for $\sigma, 0 \vDash \varphi$. The language of $\varphi$, written $\mathcal{L}(\varphi)$, is the set of infinite words that satisfy $\varphi$, that is, $\mathcal{L}(\varphi) = \{\sigma \in \Sigma^\omega \mid \sigma \vDash \varphi\}$.

### 7.1.2 Automata

A UNIVERSAL CO-BÜCHI AUTOMATON $\mathcal{A}$ over a finite alphabet $\Sigma$ is a tuple $\langle Q, q_0, \delta, F\rangle$, where $Q$ is a finite set of states, $q_0 \in Q$ the designated initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is the set of rejecting states. Given an infinite word $\sigma \in \Sigma^\omega$, a run of $\sigma$ on $\mathcal{A}$ is a finite or infinite path $q_0 q_1 q_2 \cdots \in (Q^* \cup Q^\omega)$ that respects the transition relation, i.e., for all $i \geq 0$ with $i+1 < |\sigma|$ it holds that $(q_i, \sigma_i, q_{i+1}) \in \delta$. A run is accepting, if it contains only finitely many rejecting states, i.e., either the run is finite or there exists an $i \geq 0$ such that for all $j \geq i$ it holds that $q_j \notin F$. $\mathcal{A}$ accepts a word $\sigma$, if *all* runs of $\sigma$ on $\mathcal{A}$ are accepting. The language of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set $\{\sigma \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \sigma\}$.

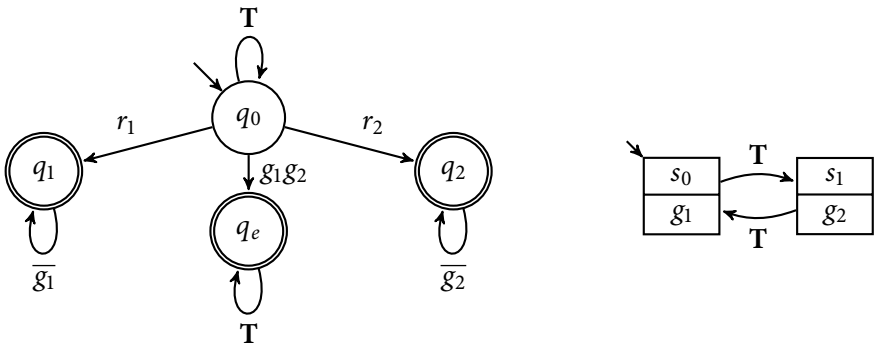
We represent automata as directed graphs with vertex set $Q$ and a symbolic representation of the transition relation $\delta$ as propositional formulas $\mathbb{B}(\mathsf{AP})$. The rejecting states in $F$ are marked by double lines.

**Example 7.1.** Consider the LTL formula $\psi = \Box(r_1 \rightarrow \bigcirc\Diamond g_1) \wedge \Box(r_2 \rightarrow \bigcirc\Diamond g_2) \wedge \Box\neg(g_1 \wedge g_2)$. Whenever there is a request $r_i$, the corresponding grant $g_i$ must be set eventually. Further, it is disallowed to set both grants simultaneously. The universal co-Büchi automaton $\mathcal{A}_\psi$ that accepts the same language as $\psi$ is shown in Figure 7.1a.

**Proposition 7.2** ([KV05])**.** *Given an LTL formula $\varphi$, there is a universal co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{O}(2^{|\varphi|})$ states that accepts the language $\mathcal{L}(\varphi)$.*

### 7.1.3 Transition Systems as a Model of Reactive Systems

We use transition systems as a model of computation for reactive systems. Transition systems consume sequences over an input alphabet by transforming their internal state in every step. Let $I$ be a finite set of input propositions and let $\Upsilon = 2^I$ be the corresponding finite alphabet. A $\Upsilon$-TRANSITION SYSTEM is a tuple $\langle S, s_0, \tau\rangle$, where $S$ is a finite set




(a) Universal co-Büchi automaton $\mathcal{A}_\psi$ (b) Transition system $\mathcal{S}_{arb}$

Figure 7.1: A specification automaton over inputs $r_1, r_2$ and outputs $g_1, g_2$ and a realizing transition system.

of states, $s_0 \in S$ is the designated initial state, and $\tau\colon S \times \Upsilon \to S$ is the transition function. We write $s \xrightarrow{v} s'$ or $(s, v, s') \in \tau$ if $\tau(s, v) = s'$. We generalize the transition function to sequences over $\Upsilon$ by defining $\tau^*\colon \Upsilon^* \to S$ recursively as $\tau^*(\epsilon) = s_0$ and $\tau^*(v_0 \cdots v_{n-1} v_n) = \tau(\tau^*(v_0 \cdots v_{n-1}), v_n)$ for $v_0 \cdots v_{n-1} v_n \in \Upsilon^+$.

We define the output behavior of a transition system with respect to a labeling function. Let $O$ be a finite set of output propositions and let $\Gamma = 2^O$ be the corresponding finite alphabet. A $\Gamma$-labeled $\Upsilon$-transition system $\mathcal{S}$ is a tuple $\langle S, s_0, \tau, l \rangle$ where, $\langle S, s_0, \tau \rangle$ is a $\Upsilon$-transition system and $l$ is a labeling function. We distinguish between two types of labeling, state-labeled or Moore transition systems with labeling function $l\colon S \to \Gamma$ and transition-labeled or Mealy transition systems with labeling function $l\colon S \times \Upsilon \to \Gamma$. Given an infinite word $v = v_0 v_1 \cdots \in \Upsilon^\omega$, the transition system produces an infinite sequence of outputs $\gamma = \gamma_0 \gamma_1 \cdots \in \Gamma^\omega$, such that

$$\gamma_i = \begin{cases} l(\tau^*(v_0 \cdots v_{i-1})) & \text{if } \mathcal{S} \text{ is state-labeled} \\ l(\tau^*(v_0 \cdots v_{i-1}), v_i) & \text{if } \mathcal{S} \text{ is transition-labeled} \end{cases}$$

for every $i \geq 0$. The resulting *trace* $\rho$ is $(v_0 \cup \gamma_0)(v_1 \cup \gamma_1) \cdots \in (2^{I \cup O})^\omega$. The set of traces generated by $\mathcal{S}$ is denoted by $traces(\mathcal{S})$. A transition system $\mathcal{S}$ satisfies an LTL formula $\varphi$, if, and only if, $traces(\mathcal{S}) \vDash \varphi$.

**Example 7.3.** Figure 7.1b depicts the two-state (state-labeled) transition system $\mathcal{S}_{arb} = \langle \{s_0, s_1\}, s_0, \tau \rangle$ with $\tau(s_0, \boldsymbol{i}) = s_1$ and $\tau(s_1, \boldsymbol{i}) = s_0$ for every $\boldsymbol{i} \in 2^I$ as well as $l(s_0) = \{g_1\}$ and $l(s_1) = \{g_2\}$. The set of traces is $traces(\mathcal{S}) = (\{g_1\}\{g_2\})^\omega \cup (2^{\{i_1, i_2\}})^\omega$.

### 7.1.4 Strategies and Trees

A strategy $f\colon (2^I)^* \to 2^O$ maps sequences of input valuations $2^I$ to an output valuation $2^O$. The behavior of a strategy $f\colon (2^I)^* \to 2^O$ is characterized by an infinite tree that

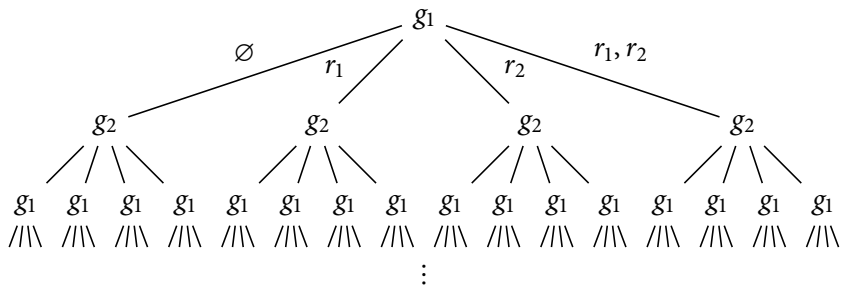



Figure 7.2: Visualization of the strategy induced by the transition system from Figure 7.1b.

branches by the valuations of $I$ and whose nodes $N \in (2^I)^*$ are labeled with the strategic choice $f(N)$. For an infinite word $v = v_0 v_1 v_2 \cdots \in (2^I)^\omega$, the corresponding labeled path is defined as $(f(\epsilon) \cup v_0)(f(v_0) \cup v_1)(f(v_0 v_1) \cup v_2) \cdots \in (2^{I \cup O})^\omega$. We lift the set containment operator $\in$ to the containment of a labeled path $\sigma = \sigma_0 \sigma_1 \sigma_2 \cdots \in (2^{I \cup O})^\omega$ in a strategy tree induced by $f \colon (2^I)^* \to 2^O$, i.e., $\sigma \in f$ if, and only if, $f(\epsilon) = \sigma_0 \cap O$ and $f((\sigma_0 \cap I) \cdots (\sigma_i \cap I)) = \sigma_{i+1} \cap O$ for all $i \geq 0$. We define the satisfaction of an LTL formula $\varphi$ (over atomic propositions $I \cup O$) on strategy $f$, written $f \vDash \varphi$, as $\{\sigma \mid \sigma \in f\} \vDash \varphi$. Thus, a strategy $f$ is a model of $\varphi$ if the set of labeled paths of $f$ is a model of $\varphi$.

A transition system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ GENERATES the strategy $f$ if $f(v) = l(\tau^*(v))$ for every $v \in (2^I)^*$. A strategy $f$ is called FINITE-STATE if there exists a transition system that generates $f$. The strategy generated by the transition system $\mathcal{S}_{arb}$ from Example 7.3 is depicted in Figure 7.2.

## 7.2 Safraless Synthesis

The game-based approach to reactive synthesis, dating back to Büchi and Landweber's seminal 1969 paper [BL69], first translates the specification into an equivalent nondeterministic Büchi word automaton [VW94]. Afterward, the automaton is transformed into a deterministic parity tree automaton [Saf88; Pit07] that accepts those infinite trees that satisfy the specification. Deciding the emptiness problem of the tree automaton, by solving the underlying parity game, then solves the realizability problem. Examples of synthesis tools implementing this approach are LTLSYNT [MC18] and STRIX [MSL18].

Safraless decision procedures [KV05] avoid the transformation of the specification into an equivalent deterministic automaton via Safra's determinization procedure. Instead, the specification is first translated into an equivalent universal co-Büchi automaton, whose language is then approximated in a sequence of deterministic safety automata, obtained by bounding the number of visits to rejecting states [FS07]. Synthesis tools employing this approach are, e.g., UNBEAST [Ehl11], ACACIA+ [Boh+12], and recent versions of BOSY [FFT17].

Bounded synthesis [SF07] limits not only the number of visits to rejecting states, but also the number of states of the synthesized system itself. As a result, the bounded synthesis problem can be represented as a decidable constraint system, even in set-

tings where the classical synthesis problem is undecidable, such as the synthesis of asynchronous and distributed systems (cf. [FS13]). There have been several proposals for encodings of bounded synthesis. The first encoding [FS07; SF07] was based on first-order logic modulo finite integer arithmetic. Improvements to the original encoding include the representation of transition systems that are not necessarily input-preserving, and, hence, often significantly smaller [FS13], the lazy generation of the constraints from model checking runs [FJ12], and specification rewriting and modular solving [KJB13b]. Recently, a SAT-based encoding was proposed [SHY15]. Another SAT-based encoding [FK16] bounds, in addition to the number of states, also the number of loops. A QBF-based encoding has been used in the related problem of solving Petri games [Fin15]. Petri games can be used to solve specific distributed synthesis problems. They have, however, a significantly simpler winning condition than the games resulting from LTL specifications.

### 7.2.1 Safety Game Reduction

Before we consider the bounded synthesis problem, we briefly describe the Safraless safety game reduction method [FS07], for which we show that insights from solving QBF in negation normal form given in Chapter 4 can lead to improvement over the state-of-the-art SAT-based safety game solver.

**Infinite Games.** Infinite games are a convenient way to represent the realizability problem. Based on a finite graph structure, called arena, the environment and system player move ad infinitum a token by choosing outgoing transitions. For our description, we will use standard game notation, e.g., player 0 typically represents the system player while player 1 represents the environment.

An ARENA $Ar = \langle V, V_0, V_1, E \rangle$ is a finite graph where $V$ is a finite set of vertices, partitioned into vertices $V_0 \subseteq V$ belonging to player 0 and $V_1 \subseteq V$ belonging to player 1, as well as an edge relation $E \subseteq V \times V$. A PLAY is an infinite sequence $\rho$ satisfying $(\rho_i, \rho_{i+1}) \in E$ for all $i \geq 0$. An INFINITE GAME $\mathcal{G} = \langle Ar, Win \rangle$ consists of an arena, on which the game is played, and a winning condition $Win \subseteq V^\omega$ for player 0. A SAFETY GAME $\mathcal{G} = \langle Ar, Bad \rangle$ with $Bad \subseteq V$ is an infinite game with the winning condition that no play visits a *bad state* $v \in Bad$, i.e., the associated winning condition is $Win = \{\rho \in V^\omega \mid \rho_i \notin Bad \text{ for all } i \geq 0\}$.

**Reduction.** Given a universal co-Büchi automaton $\mathcal{A} = \langle Q, q_0, \delta, F \rangle$ over alphabet $\Sigma = 2^{I \cup O}$ and a bound $k > 0$, we recap the reduction to safety games [FS07] where a winning strategy for player 0 (from the initial vertex) implies that system player has a realizing strategy. The underlying concept of the reduction is that the game represents all runs of the automaton, that is, determines the universal automaton. We define the underlying arena $Ar_{\mathcal{A}} = \langle V, V_0, V_1, E \rangle$ as follows. The state space for player 1, $V_1 : Q \to \{0, \ldots, k\}$, is a representation of the powerset of $Q$, where $V(q) = 0$ and $V(q) = 1$ means that $q \in Q$ is not reached and reached, respectively. For rejecting states, we additionally store the number of recurring visits. Player 0 reacts on actions chosen by player 1, thus, we encode those actions in the state space, i.e., $V_0 = \{(v, \boldsymbol{i}) \mid v \in V_1, \boldsymbol{i} \in 2^I\}$. Thus, $V = V_0 \cup V_1$. The

edge relation $E = V \times V$ is defined such that for every $(v_0, \boldsymbol{i}) \in V_0$ and $v_1 \in V_1$ it holds that $(v_1, (v_0, \boldsymbol{i})) \in E$ if, and only if, $v_0 = v_1$ and $((v_0, \boldsymbol{i}), v_1) \in E$ if, and only if

$$
\begin{aligned}
&\exists \boldsymbol{o} \in 2^O. \forall q' \in Q. \\
&\quad v_1(q') = 0 \leftrightarrow \forall q \in Q. v_0(q) > 0 \rightarrow q' \notin \delta(q, \boldsymbol{i}, \boldsymbol{o}, q') && \text{(unreach)} \\
&\wedge\, v_1(q') = v_0(q') \leftrightarrow q' \notin F \wedge \exists q \in Q. v_0(q) > 0 \wedge q' \in \delta(q, \boldsymbol{i}, \boldsymbol{o}, q') && \text{(reach)} \\
&\wedge\, v_1(q') = v_0(q') + 1 \leftrightarrow q' \in F \wedge \exists q \in Q. v_0(q) > 0 \wedge q' \in \delta(q, \boldsymbol{i}, \boldsymbol{o}, q'). && \text{(reject)}
\end{aligned}
$$

In other words, (reach) states $q'$ is only unreachable iff there is no transition $q \xrightarrow{\boldsymbol{i} \cup \boldsymbol{o}} q'$ from any reachable automaton state $q$, (reach) states that $q'$ is only reachable iff there is a transition $q \xrightarrow{\boldsymbol{i} \cup \boldsymbol{o}} q'$ from some reachable automaton state $q$, and lastly (reject) implements the rejecting counter. We denote the initial vertex by $v_{init} \in V_1$, where $v_{init}(q_0) = 1$ and for all $q \in Q \setminus \{q_0\}$ it holds that $v_{init}(q) = 0$. The set of bad states is $Bad = \{v \in V_1 \mid \exists q \in F. v(q) = k\}$.

**Example 7.4.** Figure 7.3 depicts the safety game resulting from applying the reduction to the automaton from our earlier example (Figure 7.1a). In this visualization, we use $\langle q_0^a, q_1^b, q_2^c, q_e^d \rangle$ for $a, b, c, d \in \{0, 1, 2\}$ to denote the state $\{q_0 \mapsto a, q_1 \mapsto b, q_2 \mapsto c, q_e \mapsto d\}$. To improve readability, we omit unreachable states $q^0$ and use $q$ instead of $q^1$. This view approximates the search space of synthesis tools like Unbeast [Ehl11] and Acacia+ [Boh+12].

**Lemma 7.5** ([FS07])**.** *If player 0 has a winning strategy for $Ar_{\mathcal{A}}$ from initial vertex $v_{init}$ then there is a realizing strategy for $\mathcal{A}$.*

The representation of the winning strategy can be improved by considering *sparse* strategies, i.e., positional strategies that are only defined for the reachable part of the state space [EM12].

**Symbolic Safety Games.** While the safety game reduction has the benefit of a polynomial decision procedure, the state space is exponential in the number of input propositions. To avoid this blow-up, we instead use a *symbolic* game representation. A SYMBOLIC SAFETY GAME is a tuple $\mathcal{G} = \langle X, X_{init}, T, P \rangle$ where $X$ is a finite set of state variables, $X_{init} \in \mathbb{B}(X)$ is the initial state constraint, $T \in \mathbb{B}(X \cup I \cup O \cup X')$ represents the transition relation and $P: \mathbb{B}(X)$ represents the states on which the property holds.

Given a universal co-Büchi automaton $\mathcal{A} = \langle Q, q_0, \delta, F \rangle$ over alphabet $\Sigma = 2^{I \cup O}$ and a bound $k > 0$, we build a safety game where a winning strategy for the system player implies the existence of a realizing strategy. We use $\delta_q^{\mathbb{B}} \in \mathbb{B}(Q, I, O)$ to denote the state, input, and output valuations that lead to $q$ as a Boolean formula. We define a symbolic safety game $\mathcal{G}_{\mathcal{A}}$ where $X = \{q \in Q\} \cup \{q^1, \ldots, q^k \mid q \in F\}$ is the state space, $X_{init} = q_0 \wedge \bigwedge_{q \in Q \setminus \{q_0\}} \neg q$ is the initial state, $T = \bigwedge_{q' \in Q} q' \leftrightarrow \delta_q^{\mathbb{B}} \wedge \bigwedge_{\substack{1 < i \leq k \\ q^{i'} \in F}} q^{i'} \leftrightarrow \delta_{q^{i-1}}^{\mathbb{B}}$ is the transition function, and $P = \neg \bigvee_{q \in F} q^k$ is the safety property. The underlying idea is the same as for the non-symbolic safety game.

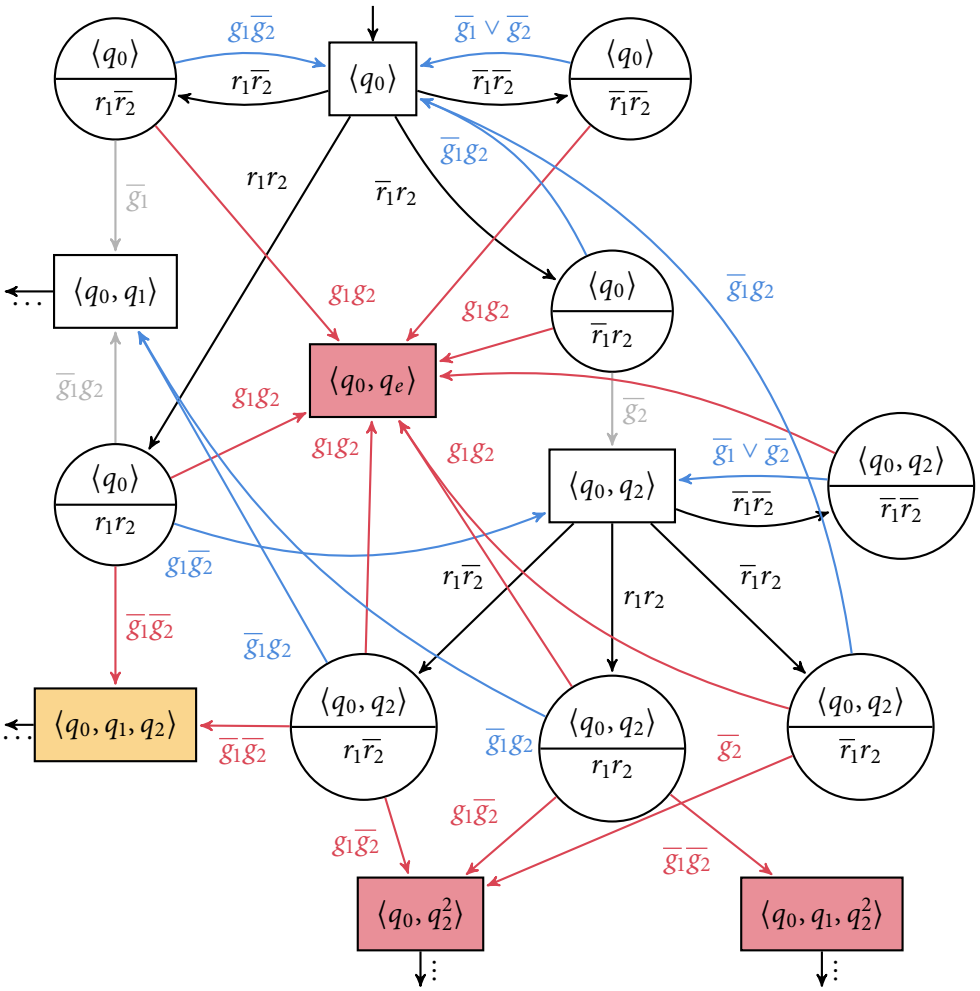



Figure 7.3: Safety game resulting from the automaton given in Figure 7.1a and bound $k = 2$. Red and orange states denote bad states and losing states for player 0, respectively. Red and blue edges denote losing and winning moves of player 0, respectively.

**Corollary 7.6.** *If the system player wins the game $\mathcal{G}_\mathcal{A}$ then there is a realizing strategy for $\mathcal{A}$.*

The solution to a symbolic safety game, that is, the existence of a *winning region* $w\colon \mathbb{B}(X)$ containing the initial state(s), implying the property $P$ and being invariant under the transition relation, can be expressed by the DQBF formula [BKS14]

$$\begin{aligned}&\forall \vec{x}.\, \exists w.\, \forall \vec{i}.\, \exists \vec{o}.\\ &\forall \vec{x}'.\, \exists w'.\end{aligned} \quad (X_{init} \Rightarrow w) \wedge (w \Rightarrow P) \wedge ((\vec{x} = \vec{x}') \Rightarrow (w = w')) \wedge (w \wedge T \Rightarrow w')$$

A direct solving approach starts with the states that are initially safe and then computes the smallest fixpoint over the enforceable predecessor that can be computed using the QBF formula (with free variables $\vec{x}$) $pred(F) = \forall \vec{i}.\, \exists \vec{o}, \vec{x}'.\, T(\vec{x}, \vec{i}, \vec{o}, \vec{x}') \wedge F(\vec{x}')$. We

Table 7.1: Result of our prototype safety game solver on the safety benchmarks from the reactive synthesis competition (SYNTCOMP).

| benchmark family | DEMIURGE | prototype |
|---|---|---|
| amba | 141 | **147** |
| genbuf | 37 | **45** |
| circuit-equiv | 94 | **100** |
| ltl2dpa | 16 | **22** |
| cycle-shed | 11 | **12** |
| driver | 31 | **36** |
| hwmcc | 140 | **145** |
| ltl2aig | 26 | **29** |
| matrix-mult | 20 | **21** |
| total | 516 | **557** |

refer to Bloem et al. [BKS14] for more details on an effective implementation using competing SAT solvers.

For the remainder of this section, we describe an optimization based on knowledge of the abstraction for the circuit abstraction algorithm given in Section 4.1.1. As a motivating example, consider the quantified formula $\forall X \exists y.\ y \leftrightarrow \varphi(X)$. Depending on $\varphi$, refinements purely based on universal variable assignments may need exponentially (in $|X|$) many refinements to determine the result. This is for example the case for $\varphi(X) = \bigoplus_{x \in X} x$. On the other hand, we have seen that the circuit abstraction algorithm only communicates subformula valuations, in our example, the value of $\varphi(X)$, resulting in a constant number of refinements (namely 2). This abstraction method can be integrated into the SAT-based algorithms [BKS14]. We implemented this idea in a prototype safety game solver and compare it to the state-of-the-art SAT-based solver DEMIURGE [BKS14] in Table 7.1. Notably, we see improvement across all benchmark sets and not only the circuit equivalence benchmarks.

### 7.2.2 Bounded Synthesis

Bounded synthesis [FS13] is a synthesis procedure for LTL specifications that produces size-optimal transition systems. A given LTL formula $\varphi$ is translated into a universal co-Büchi automaton $\mathcal{A}$ that accepts the language $\mathcal{L}(\varphi)$. A transition system $\mathcal{S}$ realizes specification $\varphi$ if, and only if, every trace generated by $\mathcal{S}$ is in the language $\mathcal{L}(\varphi)$. $\mathcal{S}$ is accepted by $\mathcal{A}$ if every path of the unique run graph, that is the product of $\mathcal{S}$ and $\mathcal{A}$, has only finitely many visits to rejecting states. This acceptance is witnessed by a bounded annotation on this product.

The bounded synthesis approach is to synthesize a transition system of bounded size $n$, by solving a constraint system that asserts the existence of a transition system and labeling function of $\mathcal{S}$ as well as a valid annotation. In this section, we discuss how to con-

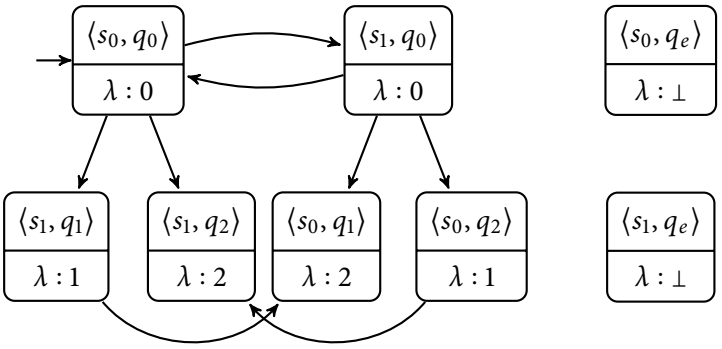



Figure 7.4: Run graph of the automaton $\mathcal{A}_\psi$ and the two-state transition system $\mathcal{S}_{arb}$ from the earlier example (Figure 7.1). The bottom node part displays a valid $\lambda$ annotation of the run graph.

struct a formula that represents that a *given* annotation is correct. We will use this formula as a building block for different bounded synthesis constraint systems in Section 7.3.

The product of a transition system $\mathcal{S} = \langle S, s, \tau, l \rangle$ and a universal co-Büchi automaton $\mathcal{A} = \langle Q, q_0, \delta, F \rangle$ is a RUN GRAPH $\mathcal{G} = \langle V, E \rangle$, where $V = S \times Q$ is the set of vertices and $E \subseteq V \times V$ is the edge relation with

$$\big((s, q), (s', q')\big) \in E \quad \text{iff} \quad \exists \boldsymbol{i} \in 2^I.\, \tau(s, \boldsymbol{i}) = s' \text{ and } (q, \boldsymbol{i} \cup l(s, \boldsymbol{i}), q') \in \delta \ . \qquad \text{(rel)}$$

A vertex $v = (s, q) \in V$ is rejecting, if and only if, $q \in F$. A run is a path starting in the initial vertex $(s_0, q_0)$. A run is *accepting* if it is finite or contains only finitely many rejecting states. A run graph is accepting if every run is accepting.

**Lemma 7.7** ([FS13])**.** *Let $\mathcal{S}$ and $\mathcal{A}$ be a transition system and universal co-Büchi automata over the same alphabet. If the run graph $\mathcal{S} \times \mathcal{A}$ is accepting, then $\mathcal{S}$ is accepted by $\mathcal{A}$.*

A witness of an accepting run graph is an ANNOTATION $\lambda$, a function $V \to \mathbb{N} \cup \{\bot\}$ that maps nodes from the run graph to either unreachable $\bot$ or a natural number $k$. An annotation is valid if it satisfies the following conditions:

- the initial vertex $v_0 = (s_0, q_0)$ is labeled by a natural number ($\lambda(v_0) \neq \bot$), and
- if a vertex $v \in V$ is annotated with a natural number ($\lambda(v) = k \neq \bot$) then every successor vertex $v' \in V$ with $(v, v') \in E$ is annotated with a greater number, which needs to be strictly greater if $v'$ is rejecting. That is, $\lambda(v') \trianglerighteq k$ where $\trianglerighteq \ := \ >$ if $v'$ is rejecting and $\geq$ otherwise.

**Example 7.8.** Figure 7.4 shows the run graph of $\mathcal{A}_\psi$ and $\mathcal{S}_{arb}$ from our earlier example (Figure 7.1). Additionally, a valid annotation $\lambda$ is provided in the second component of every node. One can verify that the annotation is correct by checking every edge individually. For example, the annotation has to increase from $\langle s_0, q_0 \rangle \to \langle s_1, q_2 \rangle$ and from $\langle s_0, q_2 \rangle \to \langle s_1, q_2 \rangle$ as $q_2$ is rejecting. As $\lambda(\langle s_0, q_0 \rangle) = 0$ and $\lambda(\langle s_0, q_2 \rangle) = 1$, it holds that $\lambda(\langle s_1, q_2 \rangle)$ must be at least 2.

More formally, we encode the check for a valid annotation as two constraints

$$\lambda(v_0) \neq \bot \tag{init}$$
$$\forall v, v' \in V. (\lambda(v) \neq \bot \wedge (v, v') \in E) \rightarrow (\lambda(v') \neq \bot \wedge \lambda(v') \trianglerighteq \lambda(v)) \tag{succ}$$

form which we derive encodings for various theories like propositional logic, (dependency) quantified Boolean formulas, and satisfiability modulo theories in Section 7.3.

**Theorem 7.9** ([FS13]). *Let $\mathcal{S}$ and $\mathcal{A}$ be a transition system and universal co-Büchi automata over the same alphabet. If $\lambda$ is a valid annotation of the run graph $\mathcal{S} \times \mathcal{A}$, then $\mathcal{S}$ is accepted by $\mathcal{A}$.*

## 7.3 Encodings of Bounded Synthesis

In this section, we explore different encodings of the constraint system given above. The de facto standard and most obvious candidate is an encoding to the SMTLIB [BFT17] format which conveniently can express quantification over Boolean and finite data-types (such as the state-space), as well as the ordering constraints of the $\lambda$ annotation. The original formulation of bounded synthesis [FS07; SF07] and many other [FS13; KJB13b; FJ12] uses the SMT encoding. This has the disadvantage of not being able to use state-of-the-art solvers for related domains like SAT, QBF, and DQBF. In this section, we explore specialized encodings to these domains. To derive a baseline, we use a purely propositional encoding of the constraints (init) and (succ) by expanding all quantifiers, especially also those contained in (rel), and encoding the ordering constraints as binary arithmetic. Afterward, we show how the signals controlled by the environment (input propositions $I$) can be handled symbolically by universal propositional quantification, leading to an exponentially more succinct constraint system. Furthermore, we derive encodings that handle the state-space of automaton $\mathcal{A}$ and transition system $\mathcal{S}$ symbolically by an encoding to DQBF. In the following section, we then evaluate those different encodings using state-of-the-art satisfiability solvers.

### 7.3.1 SAT: The Basic Encoding

Given $\mathcal{S}$, $\mathcal{A}$, and $\lambda$, we want to derive a propositional constraint that is satisfiable if, and only if, the annotation is valid. First, by the characterization above, we know that we can verify the annotation by local checks, i.e., we have to consider only one step in the product graph. To derive a propositional encoding, we encode $\mathcal{S}$, $\mathcal{A}$, and $\lambda$:

- $\mathcal{S} = \langle S, s, \tau, l \rangle$. We represent the transition function $\tau$ by one variable $\tau_{s,\boldsymbol{i},s'}$ representing a transition form $s$ to $s'$ with $\boldsymbol{i} \in 2^I$ and the labeling function $l$ by one variable $l_{o,s,\boldsymbol{i}}$ for every output proposition $o \in O$, state $s \in S$, and input valuation $\boldsymbol{i} \in 2^I$. Given $s, s' \in S$, $\boldsymbol{i} \in 2^I$ and $o \in O$, it holds that (1) $\tau_{s,\boldsymbol{i},s'}$ is true if, and only if, $\tau(s, \boldsymbol{i}) = s'$, and (2) $l_{o,s,\boldsymbol{i}}$ is true if, and only if, $o \in l(s, \boldsymbol{i})$. Note that we assume transition-labeled transition systems. The corresponding encoding for state-labeled transition systems only differs in the encoding of the labeling function, i.e., $l_{o,s}$ instead of $l_{o,s,\boldsymbol{i}}$.

- $\mathcal{A} = \langle Q, q_0, \delta, F \rangle$. We represent $\delta : (Q \times 2^{I \cup O} \times Q)$ as propositional formulas $\delta^{\mathbb{B}}_{q,\boldsymbol{i},q'}$ over the output variables $O$. That is, an assignment $\boldsymbol{o} \in 2^O$ satisfies $\delta^{\mathbb{B}}_{q,\boldsymbol{i},q'}$ iff $(q, \boldsymbol{i} \cup \boldsymbol{o}, q') \in \delta$.

- $\lambda \colon S \times Q \to \mathbb{N} \cup \{\bot\}$. We first split the annotation $\lambda$ into two parts: The first part $\lambda^{\mathbb{B}} : S \times Q \to \mathbb{B}$ represents the reachability constraint and the second part $\lambda^{\#} : S \times Q \to \mathbb{N}$ represents the bound. For every $s \in S$ and $q \in Q$ we introduce variables $\lambda^{\mathbb{B}}_{s,q}$ that we assign to true iff the state pair is reachable from the initial state pair and a bit vector $\lambda^{\#}_{s,q}$ that represents the binary encoding of the value $\lambda(s, q)$.

Using the encodings of $\mathcal{S}$, $\mathcal{A}$, and $\lambda$, we derive the following constraint system.

$$\exists_{s \in S, q \in Q} \lambda^{\mathbb{B}}_{s,q}, \lambda^{\#}_{s,q} . \exists_{s,s' \in S, \boldsymbol{i} \in 2^I} \tau_{s,\boldsymbol{i},s'} . \exists_{o \in O, s \in S, \boldsymbol{i} \in 2^I} l_{o,s,\boldsymbol{i}} .$$

$$\bigwedge_{s \in S} \bigwedge_{\boldsymbol{i} \in 2^I} \bigvee_{s' \in S} \tau_{s,\boldsymbol{i},s'} \qquad (\tau\text{-complete})$$

$$\lambda^{\mathbb{B}}_{s_0,q_0} \qquad \text{(init)}$$

$$\bigwedge_{q \in Q} \bigwedge_{s \in S} \left( \lambda^{\mathbb{B}}_{s,q} \to \bigwedge_{q' \in Q} \bigwedge_{\boldsymbol{i} \in 2^I} \left( \delta^{\mathbb{B}}_{q,\boldsymbol{i},q'}[o \mapsto l_{o,s,\boldsymbol{i}}] \to \bigwedge_{s' \in S} \left( \tau_{s,\boldsymbol{i},s'} \to \lambda^{\mathbb{B}}_{s',q'} \wedge \lambda^{\#}_{s',q'} \trianglerighteq \lambda^{\#}_{s,q} \right) \right) \right) \qquad \text{(succ)}$$

In addition to (init) and (succ), we have a constraint that enforces that for every state and every input combination, there is at least one successor as otherwise the transition system would not be complete and the constraint (succ) can be trivially satisfied. We use the notation $\delta^{\mathbb{B}}_{q,\boldsymbol{i},q'}[o \mapsto l_{o,s,\boldsymbol{i}}]$ to replace every occurrence of $o \in O$ by the variable representing the labeling $l_{o,s,\boldsymbol{i}}$. Note, that we do not enforce that $\tau$ is deterministic. This opens up the possibility to optimize the non-determinism using other design constraints by using a weighted satisfiability problem, such as MaxSAT. One can, for example, try to maximize the non-determinism by optimizing the number of positive assignments $\tau_{s,\boldsymbol{i},s'}$, or minimize the number of $\lambda^{\mathbb{B}}$ assignments to minimize the number of reachable states in the run graph.

**Proposition 7.10.** *If the propositional constraint system is satisfiable for a given $\mathcal{A}$, then there exists a transition system $\mathcal{S}$ and a valid $\lambda$ annotation such that $\mathcal{S}$ is accepted by $\mathcal{A}$.*

*Proof.* The propositional encoding is a straightforward unrolling of the constraints (init) and (succ). □

**Proposition 7.11.** *The size of the constraint system is in $\mathcal{O}(nm^2 \cdot 2^{|I|} \cdot (\max_{q,q' \in Q, \boldsymbol{i} \in 2^i} |\delta^{\mathbb{B}}_{q,i,q'}| + nb))$ and the number of variables is in $\mathcal{O}(n(mb + 2^{|I|} \cdot (n + |O|)))$, where $n = |S|$, $m = |Q|$, and $b$ is the number of bits for the binary encoding of $\lambda^{\#}$.*

*Proof.* Let $n = |S|$ and $m = |Q|$. The number of variables is in

$$\begin{aligned}
&\mathcal{O}(\underbrace{nm}_{\lambda} + \underbrace{nmb}_{\lambda^{\#}} + \underbrace{n^2 \cdot 2^{|I|}}_{\tau} + \underbrace{n \cdot |O| \cdot 2^{|I|}}_{l}) \\
&= \mathcal{O}(n(mb + n \cdot 2^{|I|} + |O| \cdot 2^{|I|})) \\
&= \mathcal{O}(n(mb + 2^{|I|} \cdot (n + |O|)))
\end{aligned}$$

The size of the constraint system is

$$\begin{aligned}
&\mathcal{O}(\underbrace{n^2 \cdot 2^{|I|}}_{(\tau\text{-complete})} + \underbrace{1}_{(\text{init})} + \underbrace{nm \cdot m \cdot 2^{|I|} \cdot (\max_{q,q' \in Q, \boldsymbol{i} \in 2^i} |\delta^{\mathbb{B}}_{q,i,q'}| + nb))}_{(\text{succ})} \\
&= \mathcal{O}(nm^2 \cdot 2^{|I|} \cdot (\max_{q,q' \in Q, \boldsymbol{i} \in 2^i} |\delta^{\mathbb{B}}_{q,i,q'}| + nb))
\end{aligned}$$

□

Since we only quantify existentially over propositional variables, the encoding can be solved by a SAT solver. The synthesized transition system can be directly extracted from the satisfying assignment of the solver.

### 7.3.2 QBF: The Input-Symbolic Encoding

One immediate drawback of the encoding above is the explicit handling of the inputs in the existential quantifiers representing the transition function $\tau$ and the labeling $l$, which introduces several variables for each possible input $\boldsymbol{i} \in 2^I$. This leads to a constraint system that is exponential in the number of inputs, both in the size of the constraints and in the number of variables. Also, since all variables are quantified on the same level, some of the inherent structure of the problem is lost and the solver will have to assign a value to each propositional variable, which may lead to non-minimal solutions of $\tau$ and $l$ due to unnecessary interdependencies.

By adding a universal quantification over the input variables, we obtain a quantified Boolean formula (QBF) and avoid this exponential blow-up. In this encoding, the variables representing the $\lambda$ annotation remain in the outer existential quantifier—they cannot depend on the input. We then universally quantify over the valuations of the input propositions $I$ (interpreted as variables in this encoding) before we existentially quantify over the remaining variables.

By the semantics of QBF, the innermost quantified variables, representing the transition function $\tau$ and labeling function $l$ of $\mathcal{S}$, can be seen as Boolean functions (Skolem functions) whose domain is the set of assignments to $I$. Indicating the dependency on the inputs in the quantifier hierarchy, we can now drop the indices $\boldsymbol{i}$ from the variables $\tau_{s,\boldsymbol{i},s'}$ and $l_{o,s,\boldsymbol{i}}$. Further, we now represent $\delta : (Q \times 2^{I \cup O} \times Q)$ as propositional formulas $\delta^{\mathbb{B}}_{q,q'}$ over the inputs $I$ and outputs $O$ with the following property: An assignment $\boldsymbol{i} \cup \boldsymbol{o}$ satisfies $\delta^{\mathbb{B}}_{q,q'}$ iff $(q, \boldsymbol{i} \cup \boldsymbol{o}, q') \in \delta$. We obtain the following formula for the input-symbolic encoding. (The gray box highlights the changes in the quantifier prefix compared to the

previous encoding.)

$$\exists_{s\in S, q\in Q} \lambda^{\mathbb{B}}_{s,q}, \lambda^{\#}_{s,q}.\ \forall \vec{i}.\ \exists_{s,s'\in S} \tau_{s,s'}.\ \exists_{o\in O, s\in S} l_{o,s}.$$

$$\bigwedge_{s\in S} \bigvee_{s'\in S} \tau_{s,s'} \qquad (\tau\text{-complete})$$

$$\lambda^{\mathbb{B}}_{s_0,q_0} \qquad (\text{init})$$

$$\bigwedge_{q\in Q} \bigwedge_{s\in S} \left( \lambda^{\mathbb{B}}_{s,q} \rightarrow \bigwedge_{q'\in Q} \left( \delta^{\mathbb{B}}_{q,q'}[o \mapsto l_{o,s}] \rightarrow \bigwedge_{s'\in S} \left( \tau_{s,s'} \rightarrow \lambda^{\mathbb{B}}_{s',q'} \wedge \lambda^{\#}_{s',q'} \trianglerighteq \lambda^{\#}_{s,q} \right) \right) \right) \qquad (\text{succ})$$

**Proposition 7.12.** *If the input-symbolic constraint system is satisfiable for a given $\mathcal{A}$, then there exists a transition system $\mathcal{S}$ and a valid $\lambda$ annotation such that $\mathcal{S}$ is accepted by $\mathcal{A}$.*

*Proof.* By expanding the universal quantification, we derive the propositional encoding. □

**Proposition 7.13.** *Let $n = |S|$, $m = |Q|$, and let $b$ be the number of bits for the binary encoding of $\lambda^{\#}$. The size of the input-symbolic constraint system is in $\mathcal{O}(nm^2(\max_{q,q'\in Q} |\delta^{\mathbb{B}}_{q,q'}| + nb))$. The number of existential and universal variables is in $\mathcal{O}(n(mb + n + |O|))$ and $\mathcal{O}(|I|)$, respectively.*

*Proof.* Let $n = |S|$ and $m = |Q|$. The number of universal variables is in $\mathcal{O}(|I|)$. The number of existential variables is in

$$\begin{aligned} &\mathcal{O}(\underbrace{nm}_{\lambda} + \underbrace{nmb}_{\lambda^{\#}} + \underbrace{n^2}_{\tau} + \underbrace{n \cdot |O|}_{l}) \\ &= \mathcal{O}(n(mb + n + |O|)) \end{aligned}$$

The size of the constraint system is

$$\begin{aligned} &\mathcal{O}(\underbrace{n^2}_{(\tau\text{-complete})} + \underbrace{1}_{(\text{init})} + \underbrace{nm \cdot m \cdot (\max_{q,q'\in Q} |\delta^{\mathbb{B}}_{q,q'}| + n \cdot b))}_{(\text{succ})} \\ &= \mathcal{O}(nm^2(\max_{q,q'\in Q} |\delta^{\mathbb{B}}_{q,q'}| + n \cdot b)) \end{aligned}$$ □

The input-symbolic encoding is exponentially smaller (in $|I|$) than the basic encoding and enables the solver to exploit the dependency between $I$ and the transition function $\tau$. An additional property of this encoding that we use in the implementation is the following: If we fix the values of the $\lambda$ annotation, the resulting 2QBF query represents all transition systems that are possible with respect to the $\lambda$ annotation. Since the outermost variables are existentially quantified, their assignments (in case the formula is satisfiable) can be extracted easily, even from non-certifying QBF solvers. For synthesis, we thus employ a two-step approach. We first solve the complete encoding and, if the formula was satisfiable, extract the assignment of the annotation variables $\lambda^{\mathbb{B}}_{s,q}$ and $\lambda^{\#}_{s,q}$. In the second step we instantiate the formula by the satisfiable $\lambda$ annotation and solve the

remaining formula with a certifying solver to generate Boolean functions for the inner existential variables. Those can be then translated into a realizing transition system.

We have already exploited another property of this encoding for the experiments in Section 4.3.2, namely that the constraints for the innermost quantifier alternation are local to the state $s \in S$ of the transition system and, thus, can be split syntactically using the miniscoping rules.

### 7.3.3 DQBF: The State- and Input-Symbolic Encoding

The previous encoding shows how to describe the functional dependency between the inputs $I$ and the transition function $\tau$ and outputs $o$ as a quantifier alternation. The reactive synthesis problem, however, contains more functional dependencies that we can exploit.

In the following, we describe an encoding that asserts the existence of a symbolic transition system. A SYMBOLIC TRANSITION SYSTEM $\mathcal{S} = \langle X, X_{init}, T, L \rangle$, where $X$ is a finite set of state-bits, $X_{init} \in \mathbb{B}(X)$ represents the initial state, $T \in \mathbb{B}(X \cup I \cup O \cup X')$ denotes the transition relation, and $L\colon O \to \mathbb{B}(X \cup I)$ is the labeling function. W.l.o.g. we fix $X_{init} = \bigwedge_{x \in \vec{x}} \overline{x}$. Every symbolic transition system $\mathcal{S}$ can be transformed into an equivalent transition system $\mathcal{S}$ where the size of $\mathcal{S}$ is $|\mathcal{S}| = \mathcal{O}(2^{|X|})$.

Since all variables depend on the state, we no longer have propositional variables. Instead, we use branching quantification, i.e., an encoding to dependency quantified Boolean formulas (DQBF).

$$
\begin{aligned}
&\forall \vec{x}. \exists_{q \in Q} \lambda^{\mathbb{B}}_q, \lambda^{\#}_q. \forall \vec{i}. \exists_{x' \in \vec{x}'} \tau_{x'}. \exists_{o \in O} l_o. \\
&\forall \vec{x}'. \exists_{q \in Q} \lambda'^{\mathbb{B}}_q, \lambda'^{\#}_q. \\
&\quad (\vec{x} = \vec{x}') \to \bigwedge_{q \in Q} \left( (\lambda^{\mathbb{B}}_q = \lambda'^{\mathbb{B}}_q) \wedge (\lambda^{\#}_q = \lambda'^{\#}_q) \right) && \text{(func)} \\
&\quad X_{init} \to \lambda^{\mathbb{B}}_{q_0} && \text{(init)} \\
&\quad \bigwedge_{q \in Q} \left( \lambda^{\mathbb{B}}_q \to \bigwedge_{q' \in Q} \left( \delta^{\mathbb{B}}_{q,q'}[o \mapsto l_o] \wedge \bigwedge_{x' \in \vec{x}'} (\tau_{x'} \leftrightarrow x') \to \lambda'^{\mathbb{B}}_{q'} \wedge \lambda'^{\#}_{q'} \trianglerighteq \lambda^{\#}_q \right) \right) && \text{(succ)}
\end{aligned}
$$

The branching quantification introduces two copies of the annotation, one depending on the current state $\vec{x}$ and one depending on the successor state $\vec{x}'$. To ensure that both copies represent the same function, we add the constraint (func).

**Proposition 7.14.** *If the state-symbolic constraint system is satisfiable for a given $\mathcal{A}$, then there exists a symbolic transition system $\mathcal{S}$ and a valid $\lambda$ annotation such that $\mathcal{S}$ is accepted by $\mathcal{A}$.*

**Proposition 7.15.** *Let $n = |X|$, $m = |Q|$, and let $b$ be the number of bits for the binary encoding of $\lambda^{\#}$. The size of the state-symbolic constraint system is in $\mathcal{O}(m^2 \cdot (\max_{q,q' \in Q} |\delta^{\mathbb{B}}_{q,q'}| + n + b))$. The number of existential and universal variables is in $\mathcal{O}(n + mb + |O|)$ and $\mathcal{O}(n + |I|)$, respectively.*

*Proof.* Let $n = |X|$ and $m = |Q|$. The number of universal variables is in $\mathcal{O}(n + |I|)$. The number of existential variables is in

$$\begin{aligned}&\mathcal{O}(\underbrace{m}_{\lambda} + \underbrace{mb}_{\lambda^\#} + \underbrace{n}_{\tau} + \underbrace{|O|}_{l})\\ &= \mathcal{O}(mb + n + |O|)\end{aligned}$$

The size of the constraint system is

$$\begin{aligned}&\mathcal{O}(\underbrace{n + mb}_{\text{(func)}} + \underbrace{n}_{\text{(init)}} + \underbrace{m \cdot m \cdot (\max_{q,q' \in Q} |\delta^{\mathbb{B}}_{q,q'}| + n + b))}_{\text{(succ)}}\\ &= \mathcal{O}(m^2 \cdot (\max_{q,q' \in Q} |\delta^{\mathbb{B}}_{q,q'}| + n + b))\end{aligned}$$

□

Note that when comparing to the previous encodings $n$ is only logarithmic in the size of the non-symbolic transition system.

**Encoding the states of the specification automaton.** The last dependency that we consider here is the dependency on the state space of the specification automaton. As a precondition, we need a symbolic representation $\mathscr{A} = \langle Y, Y_{init}, \Delta, R \rangle$ of a universal co-Büchi automaton over atomic propositions $I \cup O$, where $Y$ is a set of variables whose valuations represent the state space, $Y_{init} \in \mathbb{B}(Y)$ is a propositional formula representing the initial state, $\Delta \in \mathbb{B}(Y \cup I \cup O \cup Y')$ is the transition relation ($\vec{y} \cup \boldsymbol{i} \cup \boldsymbol{o} \cup \vec{y}'$ satisfies $\Delta$ iff $\vec{y} \xrightarrow{\boldsymbol{i} \cup \boldsymbol{o}} \vec{y}'$), and $R \in \mathbb{B}(Q)$ is a formula representing the rejecting states.

$$\begin{aligned}
&\forall \vec{x} \quad \forall \vec{y}.\, \exists \lambda^{\mathbb{B}}, \lambda^{\#}.\\
&\qquad \forall \vec{i}.\, \exists_{x' \in \vec{x}'} \tau_{x'}.\, \exists_{o \in O} l_o.\\
&\forall \vec{x}' \forall \vec{y}'.\, \exists \lambda'^{\mathbb{B}}, \lambda'^{\#}.\\
&\quad (\vec{x} = \vec{x}') \wedge (\vec{y} = \vec{y}') \rightarrow (\lambda^{\mathbb{B}} = \lambda'^{\mathbb{B}}) \wedge (\lambda^{\#} = \lambda'^{\#}) && \text{(func)}\\
&\quad (X_{init} \wedge Y_{init}) \rightarrow \lambda^{\mathbb{B}} && \text{(init)}\\
&\quad \lambda^{\mathbb{B}} \wedge \Delta[o \mapsto l_o] \wedge \bigwedge_{x' \in \vec{x}'} (\tau_{x'} \leftrightarrow x') \rightarrow \lambda'^{\mathbb{B}} \wedge \lambda'^{\#} \trianglerighteq \lambda^{\#} && \text{(succ)}
\end{aligned}$$

**Proposition 7.16.** *If the symbolic constraint system is satisfiable for a given $\mathscr{A}$, then there exists a symbolic transition system $\mathscr{S}$ and a valid $\lambda$ annotation such that $\mathscr{S}$ is accepted by $\mathscr{A}$.*

**Theorem 7.17.** *Let $n = |X|$, $m = |Y|$, and let $b$ be the number of bits for the binary encoding of $\lambda^\#$. The size of the symbolic constraint system is in $\mathcal{O}(n + m + b + |\Delta|)$. The number of existential and universal variables is in $\mathcal{O}(n + b + |O|)$ and $\mathcal{O}(n + m + |I|)$, respectively.*

*Proof.* Let $n = |X|$ and $m = |Y|$. The number of universal variables is in $\mathcal{O}(n + m + |I|)$. The number of existential variables is in

$$\begin{aligned}&\mathcal{O}(\underbrace{1}_{\lambda} + \underbrace{b}_{\lambda^\#} + \underbrace{n}_{\tau} + \underbrace{|O|}_{l})\\ &= \mathcal{O}(n + b + |O|)\end{aligned}$$

Table 7.2: The table compares the encodings with respect to the number of variables and the size of the constraint system. We indicate the number of states of the (non-symbolic) transition system and the (non-symbolic) automaton by $n$ and $m$, respectively. $b$ denotes the number of bits in the binary encoding of $\lambda^{\#}$.

| | # existentials | # universals | constraint size |
|---|---|---|---|
| basic | $n(mb + 2^{\lvert I\rvert} \cdot (n + \lvert O\rvert))$ | - | $nm^2 \cdot 2^{\lvert I\rvert} \cdot (\max_{q,q' \in Q, \boldsymbol{i} \in 2^I} \lvert\delta^{\mathbb{B}}_{q,q'}\rvert + nb)$ |
| input-symbolic | $n(mb + n + \lvert O\rvert)$ | $\lvert I\rvert$ | $nm^2(\max_{q,q' \in Q} \lvert\delta^{\mathbb{B}}_{q,q'}\rvert + nb)$ |
| state-symbolic | $\log n + mb + \lvert O\rvert$ | $\log n + \lvert I\rvert$ | $m^2(\max_{q,q' \in Q} \lvert\delta^{\mathbb{B}}_{q,q'}\rvert + \log n + b)$ |
| symbolic | $\log n + b + \lvert O\rvert$ | $\log nm + \lvert I\rvert$ | $\log nm + b + \lvert\Delta\rvert$ |

The size of the constraint system is

$$\mathcal{O}(\underbrace{n + m + b}_{\text{(func)}} + \underbrace{n + m}_{\text{(init)}} + \underbrace{\lvert\Delta\rvert + n + b}_{\text{(succ)}})$$
$$= \mathcal{O}(n + m + b + \lvert\Delta\rvert) \qquad \square$$

### 7.3.4 Comparison

Table 7.2 compares the sizes of the presented encodings. From the basic propositional encoding, we developed more symbolic encodings by making dependencies explicit and employing quantification over Boolean functions. This conciseness, however, comes with the price of higher solving complexity. In the following section we study this tradeoff empirically.

## 7.4 Experimental Evaluation

### 7.4.1 Implementation

We implemented the encodings described in this section in a tool called BoSy[1]. By using standard formats for solving, automaton conversion, and solution representation, the implementation is highly flexible with respect to tools used for specific tasks such as LTL to automaton conversion and solving. An overview of the tool architecture is given in Figure 7.5.

The LTL to automaton conversion is provided by the tools Spot [Dur+16] and ltl3ba [Bab+12]. We reduce the number of $\lambda^{\#}$ annotations and the number of bits in the encoding by only using them for automaton states within a rejecting strongly connected component, as proposed in [Ehl12]. BoSy searches for a system implementation and a counter-strategy for the environment in parallel. An exponential search strategy is employed for the bound on the size of the transition system. In synthesis mode, we apply as a post-processing step circuit minimization provided by ABC [BM10].

[1] The tool is available at https://github.com/reactive-systems/bosy

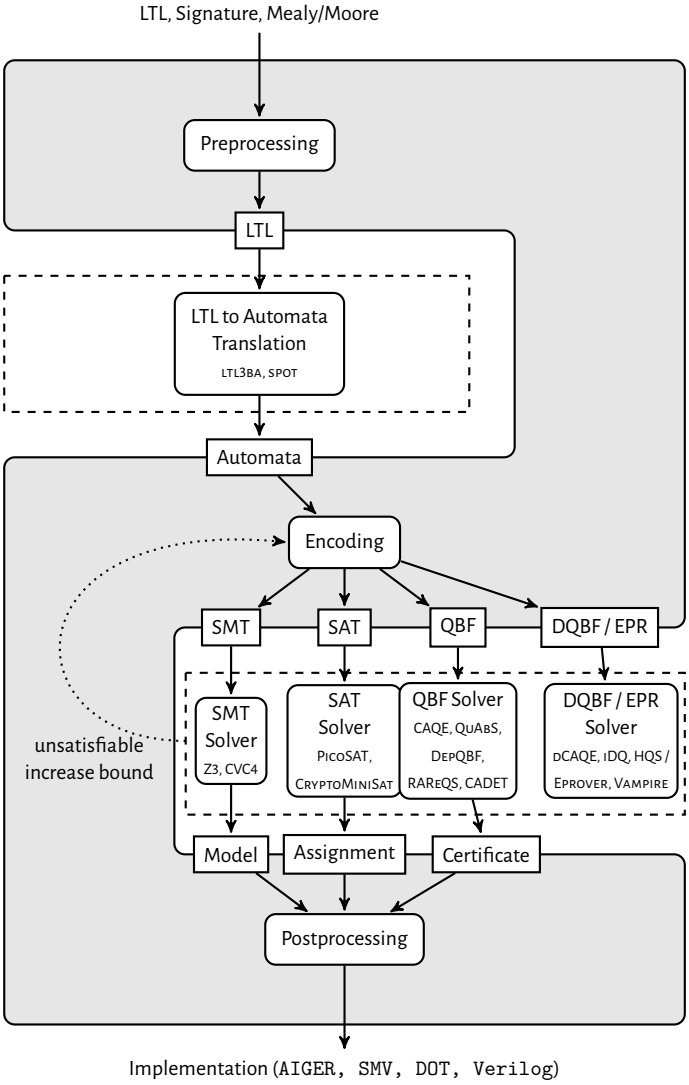



Figure 7.5: Tool Architecture of BoSy

For solving the non-symbolic encoding, we translate the propositional query to the DIMACS file format and solve it using the CryptoMiniSat [SNC09] SAT solver in version 5. The satisfying assignment is used to construct the realizing transition system.

The input-symbolic encoding is translated to the QDIMACS file format and is solved by a combination of the QBF preprocessor Bloqqer [BLS11] and the QBF solver CAQE described in Chapter 2. The solution extraction is implemented in two steps. For satisfiable queries, we first derive a top level ($\lambda$) assignment [SK14] and instantiate the QBF query using this assignment which results in a 2QBF query that represents transition systems that satisfy the specification. This is then solved using a certifying QBF solver, such as QuAbS described in Chapter 4, CADET [RS16], or DepQBF [LB10; Nie+12]. The resulting

Table 7.3: Implementation matrix

| | basic | input-symbolic | state-symbolic | symbolic |
|---|---|---|---|---|
| fragment | SAT | QBF | DQBF | DQBF |
| state-/transition-labeled | ●/● | ●/● | ●/● | ●/● |
| solution extraction | ● | ● | ○ | ○ |

Skolem functions, represented as AIGER circuit, are transformed into a representation of the transition system.

The symbolic encodings are translated to the DQDIMACS file format and solved by a DQBF solver such as DCAQE described in Chapter 6, IDQ [Frö+14], and HQS [Git+15]. Due to limited solver support, we have not implemented solution extraction.

For comparison, we also implemented an SMT version using the classical encoding [FS13]. We also tested the state-symbolic and symbolic encoding with state-of-the-art EPR solvers, but the solving times were not competitive. Table 7.3 gives an overview over the capabilities of the implemented encodings.

Internally, we use a circuit representation of the synthesized transition system. From this representation, it is possible to translate the implementation into an AIGER circuit as required by the SYNTCOMP rules, to an SMV model for model checking, to a graphical representation using the DOT format, and to a Verilog module description.

### 7.4.2 Setup & Benchmarks

For our experiments, we used a machine with a 3.6 GHz quad-core Intel Xeon (E3-1271 v3) processor and 32 GB of memory. The timeout and memout were set to 1 hour and 30 GB, respectively. We use the LTL benchmark sets from the reactive synthesis competition (SYNTCOMP) [Jac+16] from the year 2019. In total, the benchmark suite of SYNTCOMP 2019 consists of 429 benchmarks. We compare against the game-based solvers LTLSYNT [MC18] and STRIX [MSL18].

### 7.4.3 Realizability

**Bounded Synthesis Encodings.** In Figure 7.6 we depict the result of the BOSY variants

- BOSY-CAQE using the input-symbolic (QBF) encoding and the solver CAQE (Chapter 2) with preprocessor BLOQQER [BLS11],
- BOSY-DCAQE using the state-symbolic (DQBF) encoding and the solver DCAQE (Chapter 6) with preprocessor HQSPRE [Wim+17],
- BOSY-CRYPTOMINISAT using the basic (SAT) encoding and the solver CRYPTOMINISAT [SNC09],
- BOSY-Z3 using the SMT encoding [FS13] and the solver Z3 [MB08], and
- BOSY-IDQ using state-symbolic (DQBF) encoding and the solver IDQ [Frö+14] with preprocessor HQSPRE [Wim+17].

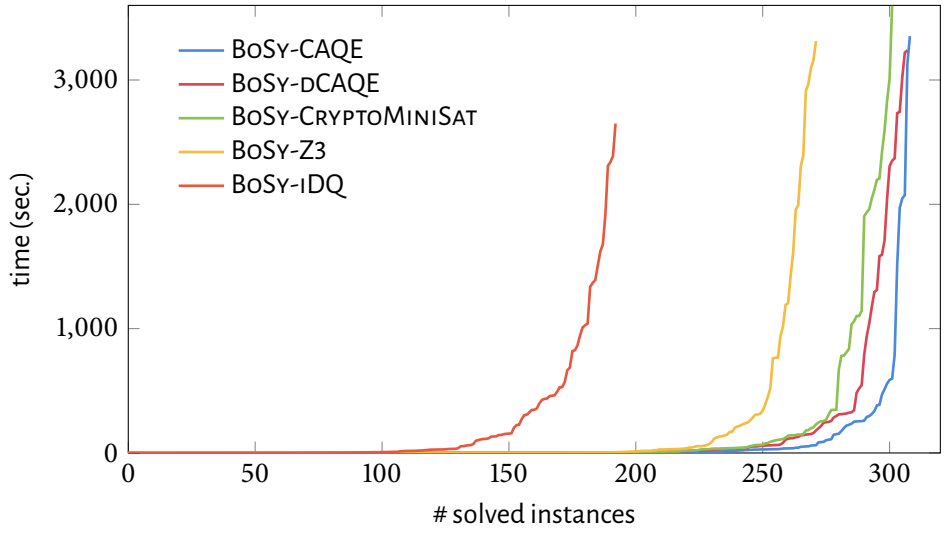



Figure 7.6: Number of solved instances within 1 hour among the 429 instances from SYNTCOMP 2019.

From the cactus plot, we observe that all encodings presented in Section 7.3 with the respective best performing solver outperform the SMT-based baseline. This is in sharp contrast to our previous experiments [Fay+17], where the DQBF encoding did not perform well. This performance increase can be directly attributed to the DQBF solver dCAQE, discussed in Chapter 6, as the comparison to the same encoding with a different DQBF solver iDQ in Figure 7.6 shows. In Table 7.4, we give more detailed results on parameterized benchmarks. While overall in most cases slower than the variant using CAQE, there is some evidence that dCAQE may scale better with respect to the number of states: BoSy-dCAQE is able to solve the 4bit arbiter variants without spurious grants (instances "full_arbiter_4" and "full_arbiter_enc_4") which no other bounded synthesis tool is able to (they are only able to solve the 3bit variants). Overall, those results show that not only is DQBF solving now feasible for solving the realizability problem, but it is also competitive when compared to state-of-the-art SAT and QBF solvers. Further, there is evidence that a symbolic representation of the state space could let to improved scalability of tools based on bounded synthesis.

**Game-based Tools.** We compare an implementation of the symbolic safety game reduction from Section 7.2.1, called BoSy-game, against the game-based tools ltlsynt in version 2.8.1 and Strix from SYNTCOMP 2018. The results are shown in Figure 7.7 and include, for comparison, the best performing BoSy variant using CAQE and the bounded synthesis tool Party [KJB13a]. BoSy-game and ltlsynt use the same tool (spot) for the conversion of LTL formulas to automata but use different solving methods afterward. This conversion is the bottleneck for those tools; we see that in Figure 7.7 they solve exactly the same amount of instances (although there is some variance on the level of individual formulas). In contrast, Strix uses a different conversion mechanism that exploits

Table 7.4: Experimental results of selected scalable instances. Reported is the number of solved instances $k$ per benchmark class and the cumulative solving time $t$ (in seconds).

| | | BoSy-dCAQE | | BoSy-CAQE | | BoSy-CMSat | | BoSy-Z3 | |
|---|---|---|---|---|---|---|---|---|---|
| instance | # | $k$ | sum $t$ | $k$ | sum $t$ | $k$ | sum $t$ | $k$ | sum $t$ |
| amba_decomp_arbiter | 6 | 3 | 965.9 | 3 | 598.0 | 2 | 80.6 | 2 | 1.3 |
| amba_decomp_encode | 6 | 4 | 18.5 | 6 | 316.5 | 4 | 117,7 | 2 | 0.9 |
| amba_decomp_lock | 6 | 5 | 318.3 | 5 | 93.4 | 3 | 1112.2 | 1 | 7.1 |
| collector | 24 | 20 | 3110.0 | 19 | 461.9 | 21 | 7882.8 | 14 | 344.9 |
| detector | 12 | 9 | 872.4 | 9 | 756.3 | 8 | 2226.8 | 7 | 1161.1 |
| full_arbiter | 15 | 8 | 5815.8 | 6 | 17.9 | 6 | 17.0 | 6 | 186.5 |
| load_balancer | 12 | 9 | 3236.7 | 9 | 119.0 | 9 | 1950.0 | 8 | 74.5 |
| ltl2dba | 47 | 17 | 2434.1 | 16 | 783.4 | 16 | 2479.0 | 11 | 370.3 |
| prioritized_arbiter | 14 | 7 | 736.4 | 7 | 370.8 | 6 | 284.9 | 5 | 321.0 |
| round_robin_arbiter | 12 | 8 | 344.4 | 9 | 843.6 | 9 | 102.1 | 8 | 365.3 |
| simple_arbiter | 17 | 10 | 249.2 | 10 | 333.2 | 9 | 524.3 | 9 | 1379.7 |

the syntactic structure of formulas and is able to solve the realizability problem for more instances. To test the conjecture that the LTL handling leads to significantly improved performance, we implemented a variant of BoSy-game, called BoSy-decompose, that tries to decompose the LTL specification into the syntactic categories *presets* (concern only the initial state), *invariants*, *safety*, and *liveness*. Furthermore, it tries to split the formula into a set of assumptions $A$ and guarantees $G$. This approach is similar to *safety-first synthesis* [SS09b] and GR(1) synthesis [KP10; PPS06], and thus, sound but incomplete for general LTL as it assumes that the liveness assumptions have no influence on the satisfiability of the safety guarantees. However, it is fast and can solve many realizable specifications, such as the AMBA bus controller and the generalized buffer examples. If the incomplete check does not determine a solution, we apply a complete method based on the decomposition of safety and liveness given in [Ehl12]. As can be seen in Figure 7.7, the resulting prototype implementation solves nearly all benchmarks (411 out of 429) of the benchmark set.

### 7.4.4 Synthesis

To evaluate the different encodings in terms of their solutions to the synthesis problem and to compare them to other competing tools, we measure the size of the provided solutions. In line with the rules of SYNTCOMP, the synthesized transition system is encoded as an AIGER circuit. The size of the result is measured in terms of the number of AND gates.

First, we compare in the scatter plot of Figure 7.8 the propositional, non-symbolic

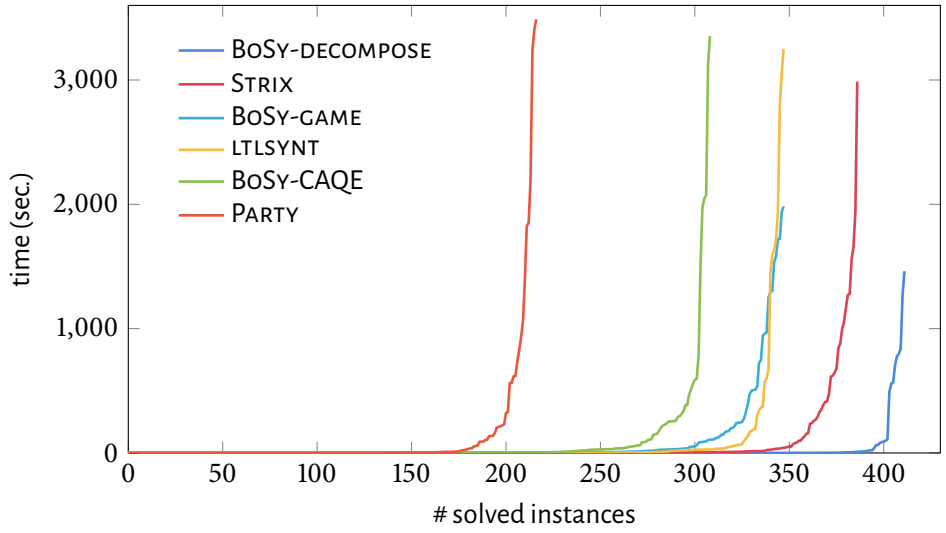



Figure 7.7: Number of solved instances within 1 hour among the 429 instances from SYNTCOMP 2019.

encoding to the input-symbolic encoding. Since most points are below the diagonal and, thus, the size of the implementation is smaller than their counterparts, the input-symbolic solutions are better in size compared to the non-symbolic encoding. We thus observe that the ability to universally quantify over the inputs and extract the transition system from the functional descriptions leads to advantages in terms of the size of the solution strategies.

In Figure 7.9, we compare our input-symbolic encoding against two competing tools. On the left, we observe that the solution sizes of our input-symbolic encoding are significantly better (observe the log-log scale) than the solutions provided by ltlsynt, sometimes with 3 orders of magnitude. The reason for the size difference is that the strategies of ltlsynt may depend on the current state of the specification automaton, as they are extracted from the resulting safety game. When comparing to Strix, there are cases where Strix produces smaller implementations when measuring the number of AND gates (but never smaller than BoSy in the number of latches). Still, there is a slight advantage of BoSy, with some instances that are two orders of magnitude smaller compared to the solutions produced by Strix.

## 7.5 Summary

We have investigated the reactive synthesis problem for linear specifications given as LTL formulas. Based on the safraless decision procedures, we introduced two methods to solve the synthesis problem. A reduction to symbolic safety games and the bounded synthesis approach. For the safety game reduction, we showed how the state-of-the-art SAT-based solution algorithm can be improved based on the insights from the circuit abstrac-

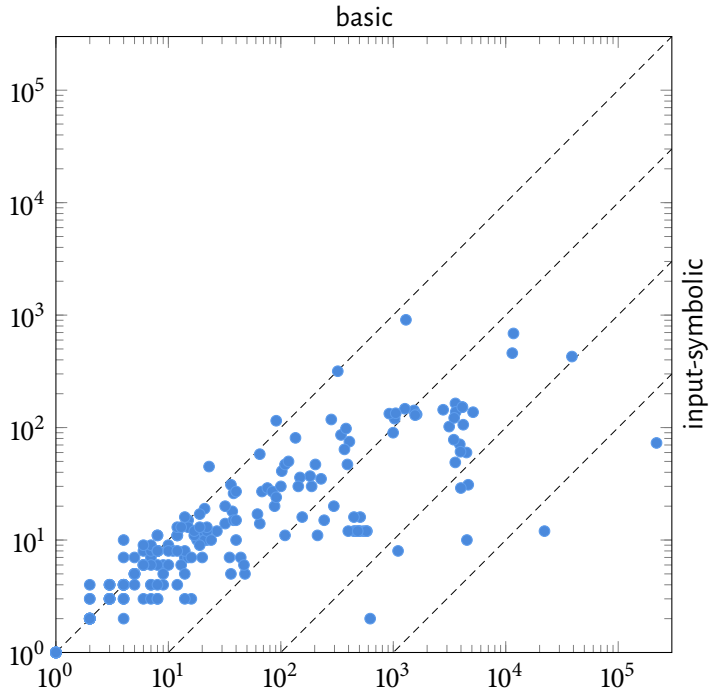



Figure 7.8: Scatter plot comparing the size of the synthesized strategies between the basic (Section 7.3.1) and input-symbolic (Section 7.3.2) encoding using CryptoMiniSat and CAQE, respectively. Both axes have logarithmic scale.

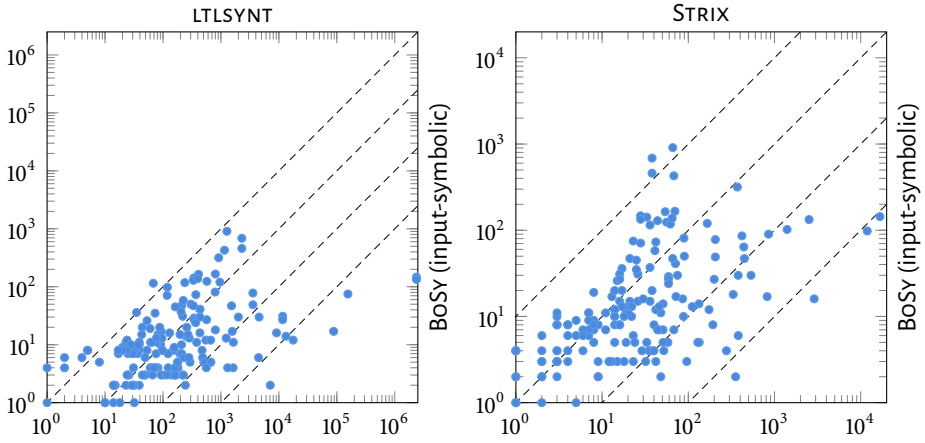



Figure 7.9: Scatter plot comparing the size of the synthesized strategies of BoSy-CAQE, ltlsynt, and Strix. Both axes have logarithmic scale.

tion approach (Section 4.1.1). We have revisited the bounded synthesis problem [FS13] and presented alternative encodings into Boolean formulas (SAT), quantified Boolean formulas (QBF), and dependency-quantified Boolean formulas (DQBF). Our evaluation

shows that the approaches based on quantified satisfiability clearly dominate the SAT approach, and also previous approaches to bounded synthesis—both in terms of the number of instances solved and in the size of the solutions.

In the following sections, we consider a more expressive specification language, that can express many prior extensions of the LTL realizability problem. In more detail, we will investigate *hyperproperties* in the form of the temporal logic HyperLTL, an extension of LTL by quantification over execution traces.

# Chapter 8

# Synthesis From Hyperproperties

Generalization is a fundamental aspect of computer science. Instead of examining similar yet seemingly unrelated research questions in isolation, the task is to find a suitable overarching framework. In reactive synthesis, there is plenty of work extending the monolithic (single-process) synthesis from $\omega$-regular languages, introduced in the previous section, to different settings, such as incomplete information [KV97], distributed systems [PR90; KV01; FS05], fault-tolerance [DF09; FT14a; FT15], reasoning over knowledge [Fag+95], to name a few. Hyperproperties [CS10] are such a generalization that promises the unification of many such extensions into a single framework. Hyperproperties extend trace properties, i.e., *sets of traces*, to properties over sets of traces, i.e., *sets of sets of traces*.

While decidability results cannot be obtained from this unification, synthesis from hyperproperties is in general undecidable [Fin+18a] even for restricted classes- The advantage is that having a semi-decision procedure for the synthesis from hyperproperties allows us to synthesize solutions for any of the prior studied extensions and, further, consider multiple extensions in conjunction. The result is that we can specify properties well beyond the expressiveness of earlier synthesis approaches that remain at the same time concise as we can pick the best representation at hand. The dining cryptographers from the introduction is such an example: The secrecy requirements can be conveniently represented using *knowledge* "no one knows whether cryptographer $C_a$ has paid" while the information available (and hidden) can be described as a *distributed architecture* where every cryptographer is an entity that has inputs (shared secrets) and produces outputs.

HyperLTL [Cla+14] is a temporal hyperlogic that extends LTL by introducing trace quantifiers. The model checking [Cla+14; FRS15; FHT18], synthesis [Fin+18a; Coe+19], satisfiability [FH16; FHS17; FHH18], and runtime verification [Fin+17b; Fin+18b; HST19; Fin+19b] problems have been studied for HyperLTL. HyperLTL is able to express all the requirements needed for the dining cryptographers example, thus, we can synthesize a realizing protocol using a single HyperLTL formula.

When considering the *output complexity* of reactive synthesis from hyperproperties, which is a more apt comparison to model checking [FS13], we show that the results from model checking of hyperproperties translate to synthesis as well: In the same way that

model checking of (alternation-free) HyperLTL has the same complexity as LTL model checking, the output complexity of (alternation-free) HyperLTL synthesis has the same complexity as the output complexity of LTL synthesis.

This chapter is based on work published in the proceedings of CAV [Fin+18a] and an article submitted to Acta Informatica [Fin+19a].

## 8.1 Temporal Hyperproperties

In this section, we develop the notion of a temporal hyperproperties and we recap HyperLTL, an extension of LTL using quantification over execution traces.

**HyperLTL.** HyperLTL [Cla+14] is a temporal logic for specifying hyperproperties. It extends LTL (see Section 7.1.1) by quantification over trace variables $\pi$ and a method to link atomic propositions to specific traces. The set of trace variables is $\mathcal{V}$. Formulas in HyperLTL are given by the grammar

$$\varphi ::= \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi$$
$$\psi ::= \mathit{true} \mid a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi$$

where $a \in \mathrm{AP}$ and $\pi \in \mathcal{V}$. For the quantifier-free part, we allow the same abbreviations as introduced for LTL in Section 7.1.1. To denote that two traces $\pi$ and $\pi'$ are equal w.r.t. propositions $A \subseteq \mathrm{AP}$, we write $\pi =_A \pi'$ as a shorthand for $\bigwedge_{a \in A} (a_\pi \leftrightarrow a_{\pi'})$.

The semantics is given by the satisfaction relation $\vDash_T$ over a set of traces $T \subseteq \Sigma^\omega$. We define an assignment $\Pi : \mathcal{V} \to \Sigma^\omega$ that maps trace variables to traces. $\Pi[\pi \mapsto t]$ updates $\Pi$ by assigning variable $\pi$ to trace $t$.

$$\begin{array}{lll}
\Pi, i \vDash_T \mathit{true} & & \\
\Pi, i \vDash_T a_\pi & \text{iff} & a \in \Pi(\pi)[i] \\
\Pi, i \vDash_T \neg\varphi & \text{iff} & \Pi, i \nvDash_T \varphi \\
\Pi, i \vDash_T \varphi \vee \psi & \text{iff} & \Pi, i \vDash_T \varphi \text{ or } \Pi, i \vDash_T \psi \\
\Pi, i \vDash_T \bigcirc\varphi & \text{iff} & \Pi, i+1 \vDash_T \varphi \\
\Pi, i \vDash_T \varphi \,\mathcal{U}\, \psi & \text{iff} & \exists j \geq i.\, \Pi, j \vDash_T \psi \wedge \forall i \leq k < j.\, \Pi, k \vDash_T \varphi \\
\Pi, i \vDash_T \exists \pi.\, \varphi & \text{iff} & \text{there is some } t \in T \text{ such that } \Pi[\pi \mapsto t], i \vDash_T \varphi \\
\Pi, i \vDash_T \forall \pi.\, \varphi & \text{iff} & \text{for all } t \in T \text{ it holds that } \Pi[\pi \mapsto t], i \vDash_T \varphi
\end{array}$$

We write $T \vDash \varphi$ for $\{\} \vDash_T \varphi$ where $\{\}$ denotes the empty assignment. Two HyperLTL formulas $\varphi$ and $\psi$ are equivalent, written $\varphi \equiv \psi$ if they have the same models. A HyperLTL formula $\varphi$ is denoted satisfiable if there is a set of traces $T$ which satisfies $\varphi$, i.e., $T \vDash \varphi$. The satisfiability problem is undecidable for general HyperLTL formulas but becomes decidable if we renounce $\forall^*\exists^*$ formulas which alternate the quantifier from $\forall$ to $\exists$ [FH16]. For an LTL formula $\varphi$, we denote by $\varphi[\pi]$ the quantifier-free HyperLTL formula where every proposition $a$ is replaced by $a_\pi$.

Every hyperproperty is an intersection of a hypersafety and a hyperliveness property [CS10]. A *hypersafety* property is one where there is a finite set of finite traces that is a

bad prefix, i.e., that cannot be extended into a (possibly infinite) set of (possibly infinite) traces that satisfies the hypersafety property. A *hyperliveness* property is a hyperproperty where every finite set of finite traces can be extended to a possibly infinite set of infinite traces such that the resulting trace set satisfies the hyperliveness property.

*(In)dependence* is a hyperproperty that we will use multiple times in this section, thus, we define the following syntactic sugar. Given two disjoint subsets of atomic propositions $C \subseteq \mathsf{AP}$ and $A \subseteq \mathsf{AP}$, we define independence as the following HyperLTL formula

$$D^{\pi,\pi'}_{A \mapsto C} \coloneqq \left( \bigvee_{a \in A} (a_\pi \nleftrightarrow a_{\pi'}) \right) \mathcal{R} \left( \bigwedge_{c \in C} (c_\pi \leftrightarrow c_{\pi'}) \right) \quad , \tag{8.1}$$

which requires that the valuations of propositions $C$ on traces $\pi$ and $\pi'$ have to be equal until and including the point in time where there is a difference in the valuation of some proposition in $A$. Prefacing universal quantification, that is, the formula $\forall \pi \forall \pi' . D^{\pi,\pi'}_{A \mapsto C}$ guarantees that every proposition $c \in C$ solely depends on propositions in $A$.

## 8.2 On the Expressiveness of Temporal Hyperproperties

In this section, we introduce the realizability problem for HyperLTL and compare its expressiveness to various previous extensions of the LTL realizability problem.

**Definition 8.1** (HyperLTL Realizability)**.** A HyperLTL formula $\varphi$ over atomic propositions $\mathsf{AP} = I \cup O$ is realizable if, and only if, there is a strategy $f \colon (2^I)^* \to 2^O$ that satisfies $\varphi$.

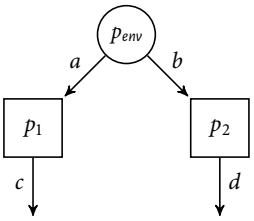
The fragment of HyperLTL with only a single, universal quantifier $\forall \pi . \varphi$ is equivalent to the LTL realizability problem of $\varphi$. With two universal quantifiers, one can express relations between traces in the execution tree, thus, one can express the LTL realizability problem with restricted information flow like incomplete information [KV97], distributed synthesis [PR90; KV01; FS05], and fault-tolerant synthesis [DF09; FT15].

**Incomplete Information.** The realizability problem with incomplete information [KV97] is a tuple $\langle \varphi, I, O, H \rangle$, where $\varphi$ is an LTL formula, $I$ is a set of input propositions, $O$ is a set of output propositions, and $H \subseteq I$ is a set of hidden inputs not observable by the system. Thus, a realizing strategy $f \colon (2^{I \setminus H})^* \to 2^O$ has a computation tree that only branches by $I \setminus H$. In order to evaluate $\varphi$, which may include propositions $H$, the computation tree is *widened* [KV97] by $H$. In HyperLTL, we can verify that a strategy $f' \colon (2^I)^* \to 2^O$ has the same output-behavior as a $H$-widened strategy $f$ by checking $f' \vDash \forall \pi \forall \pi' . D^{\pi,\pi'}_{I \setminus H \mapsto O}$.

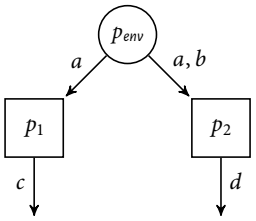
**Theorem 8.2.** *The HyperLTL realizability problem subsumes the LTL realizability with incomplete information problem.*

*Proof.* Given $\langle \varphi, I, O, H \rangle$, the following HyperLTL formula over inputs $I$ and outputs $O$ is equirealizable:

$$\forall \pi \forall \pi' . \varphi[\pi] \wedge D^{\pi,\pi'}_{I \setminus H \mapsto O} \qquad \square$$




(a) An architecture of two processes that specify process $p_1$ to produce $c$ from $a$ and $p_2$ to produce $d$ from $b$.




(b) The same architecture as on the left, where the inputs of process $p_2$ are changed to $a$ and $b$.

Figure 8.1: Distributed architectures

**Distributed Synthesis.** The distributed synthesis problem was introduced by Pnueli and Rosner [PR90] and introduces the concept of *architectures* as a constraint on the information flow. An architecture is a set of processes $P$, with distinct environment process $p_{env} \in P$, such that the processes produce outputs synchronously, but each process bases its decision only on the history of valuation of inputs that it observes.

Formally, a distributed architecture $A$ is a tuple $\langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ where $P$ is a finite set of processes with distinguished environment process $p_{env} \in P$. The functions $\mathcal{I}: P \to 2^{\mathrm{AP}}$ and $\mathcal{O}: P \to 2^{\mathrm{AP}}$ define the inputs and outputs of processes. While processes may share the same inputs (in case of broadcasting), the outputs of processes must be pairwise disjoint, i.e., for all $p \neq p' \in P$ it holds that $\mathcal{O}(p) \cap \mathcal{O}(p') = \varnothing$. W.l.o.g. we assume that $\mathcal{I}(p_{env}) = \varnothing$. We denote by $P^- = P \setminus \{p_{env}\}$ the set of processes excluding the environment process.

The distributed realizability problem for architectures without *information forks* [FS05] is decidable. Intuitively, an information fork is a situation where two distinct processes $p, p' \in P$ receive environment inputs $I$ and $I'$ (may be transitive through other processes) such that both observe inputs that the other process does not observe, i.e., there exist $i \in I$ and $i' \in I'$ such that $i \notin I'$ and $i' \notin I$. We depict two example architectures in Figure 8.1. The architecture in Figure 8.1a contains an information fork while the architecture in Figure 8.1b does not. Furthermore, the processes in Figure 8.1b can be ordered linearly according to the subset relation on the inputs.

**Theorem 8.3.** *The HyperLTL realizability problem subsumes the distributed LTL realizability problem.*

*Proof.* Given a distributed realizability problem $\langle \varphi, A \rangle$, the following HyperLTL formula over inputs $\mathcal{O}(p_{env})$ for $P^-$ and outputs $\bigcup_{p \in P^-} \mathcal{O}(p)$ is equirealizable:

$$\forall \pi \forall \pi' . \varphi[\pi] \wedge \bigwedge_{p \in P^-} D^{\pi,\pi'}_{\mathcal{I}(p) \mapsto \mathcal{O}(p)} \qquad \square$$

**Asynchronous Distributed Synthesis.** The asynchronous system model [SF06] is a generalization of the synchronous model discussed previously. In this model, we have a global scheduler, controlled by the environment, that decides when and which processes

are scheduled. The resulting distributed realizability problem is already undecidable for LTL specifications and systems with more than one process [SF06].

**Theorem 8.4.** *The HyperLTL realizability problem subsumes the asynchronous distributed LTL realizability problem.*

*Proof.* Let $A = \langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ be a distributed architecture. To model scheduling, we introduce an additional set $Sched = \{sched_p \mid p \in P^-\}$ of atomic propositions. The valuation of $sched_p$ indicates whether system process $p$ is currently scheduled or not. A process $p \in P^-$ may observe whether it is scheduled or not, that is, it may depend on $sched_p$. The environment can decide at every step which processes to schedule. When a process is not scheduled, its output behavior does not change [FS13]. As the scheduling is controlled by the environment, we assume that every process is infinitely often scheduled, as otherwise, the environment wins by simply not scheduling any process.

Given an asynchronous distributed realizability problem $\langle \varphi, A \rangle$, the following HyperLTL formula over inputs $\mathcal{O}(p_{env}) \cup Sched$ and outputs $\bigcup_{p \in P^-} \mathcal{O}_p$ is equirealizable:

$$\forall \pi \forall \pi'. \left( \bigwedge_{p \in P^-} \square\lozenge sched_p[\pi] \right) \rightarrow \varphi[\pi] \wedge \bigwedge_{p \in P^-} D^{\pi,\pi'}_{(\mathcal{I}(p) \cup \{sched_p\}) \mapsto \mathcal{O}(p)}$$
$$\wedge \square \bigwedge_{p \in P^-} \neg sched_p[\pi] \rightarrow \left( \bigwedge_{o \in \mathcal{O}(p)} o_\pi \leftrightarrow \bigcirc o_\pi \right) \qquad \square$$

**Symmetric Synthesis.** A special case of distributed synthesis is symmetric synthesis [EF17], which, additionally to distributivity, requires that all system processes act exactly the same if they are given the same inputs. Formally, symmetric synthesis requires a symmetric architecture $\langle P, p_{env}, \mathcal{I}, \mathcal{O} \rangle$ where for each process $p \in P^-$, $|\mathcal{I}(p)| = n$ and $|\mathcal{O}(p)| = m$ for some $n, m \in \mathbb{N}$. We assume an implicit ordering of inputs and output per process and use the notation $\mathcal{I}(p)_j$ and $\mathcal{O}(p)_j$ to access the $j$-th input and output of process $p \in P^-$, respectively. Then, we can express the symmetry constraint as an LTL formula

$$\bigwedge_{p,p' \in P^-} \left( \bigvee_{1 \le j \le n} \mathcal{I}(p)_j \nleftrightarrow \mathcal{I}(p')_j \right) \mathcal{R} \left( \bigwedge_{1 \le j \le m} \mathcal{O}(p)_j \leftrightarrow \mathcal{O}(p')_j \right) . \qquad \text{(sym)}$$

**Theorem 8.5.** *The HyperLTL realizability problem subsumes the symmetric (distributed) LTL realizability problem.*

*Proof.* Given a symmetric realizability problem over architecture $A$ and specifications $\varphi_1, \ldots, \varphi_k$ for the $k$ processes the following HyperLTL formula over inputs $\mathcal{O}(p_{env})$ for $P^-$ and outputs $\bigcup_{p \in P^-} \mathcal{O}(p)$ is equirealizable:

$$\forall \pi \forall \pi'. \bigwedge_{1 \le i \le k} \varphi_i[\pi] \wedge \bigwedge_{p \in P^-} D^{\pi,\pi'}_{\mathcal{I}(p) \mapsto \mathcal{O}(p)} \wedge \mathrm{sym}[\pi] \qquad \square$$

**Fault-tolerant Synthesis.** We consider another extension to the distributed synthesis problem where we incorporate the possibility that communication *between* processes may be subject to faults, such as Byzantine faults [FT14a; FT15]. In the distributed synthesis formulation above, communication from some process $p$ to $p'$ was encoded as an atomic proposition $a$ such that $a \in \mathcal{O}(p)$ and $a \in \mathcal{I}(p')$. In the fault-tolerance encoding, we split this connection into a sending part $a_s \in \mathcal{O}(p)$ and a receiving part $a_r \in \mathcal{I}(p')$ where $a_r \in \mathcal{O}(p_{env})$ is a proposition controlled by the environment. To relate $a_s$ and $a_r$, we add the assumption $\square(a_s \leftrightarrow a_r)$ to the LTL specification. This encoding uses more atomic propositions and additional LTL constraints but is otherwise equivalent to the one presented before.

This increased flexibility, that is, being able to specify communication using temporal logic, allows us to express unreliable communication. For example, using the assumption $\square(a_s \leftrightarrow \bigcirc a_r)$ specifies a delay of one time step on the receiver, $\square a_r$ specifies a stuck-at-one fault, and $\mathbf{T}$ specifies a Byzantine fault where the environment takes over the communication. This alone is not enough though: If a process gets such a specification it knows which receiving propositions present actual values and which one is subject to a fault. Thus, the processes are challenged in *multiple architectures*, where each architecture may have a different set of communication faults as well as specifications: Depending on the type of failure, the overall system may only be expected to satisfy a weaker property then the original, non-faulty one.

Formally, the fault-tolerant realizability problem is a tuple $\langle A, \varphi_1, \dots, \varphi_n \rangle$, where $A$ is a distributed architecture with the property that every process receives only environment inputs, i.e., $\mathcal{I}(p) \subseteq \mathcal{O}(p_{env})$ for all $p \in P^-$, and $\varphi_1, \dots, \varphi_n$ are LTL formulas. For Byzantine fault-tolerance, $\varphi_i = \bigwedge_{(s,r)\in R_i} \square(s \leftrightarrow r) \rightarrow \psi_i$ where $R_i \subseteq O \times I$ are the non-faulty communication of architecture $i$ and $\psi_i$ is the LTL specification that should be ensured.

As an example, consider the architecture

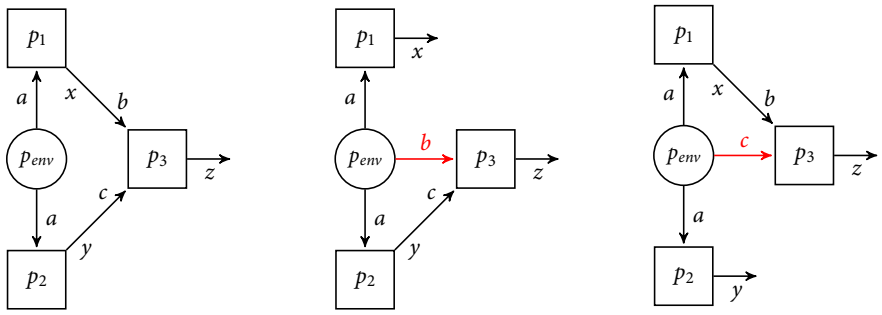
$$\langle \{p_{env}, p_1, p_2, p_3\}, p_{env}, \{p_1 \mapsto \{a\}, p_2 \mapsto \{a\}, p_3 \mapsto \{b, c\}\}, \\ \{p_{env} \mapsto \{a, b, c\}, p_1 \mapsto \{x\}, p_2 \mapsto \{y\}, p_3 \mapsto \{z\}\}\rangle \tag{8.2}$$

with specifications $\varphi_1 = \square((x \leftrightarrow b) \wedge (y \leftrightarrow c)) \rightarrow \psi$, $\varphi_2 = \square(y \leftrightarrow c) \rightarrow \psi$, and $\varphi_3 = \square(x \leftrightarrow b) \rightarrow \psi$. This example specification asserts that $\psi$ holds in all three architectures depicted in Figure 8.2, i.e., if either $p_1 \xrightarrow{x} p_3$ or $p_2 \xrightarrow{y} p_3$ fails, but not both of them. Hence, process $p_3$ cannot know whether the information given via propositions $b$ or $c$ is correct.

**Theorem 8.6.** *The HyperLTL realizability problem subsumes the fault-tolerant LTL realizability problem.*

*Proof.* Given $\langle A, \varphi_1, \dots, \varphi_n \rangle$, the following HyperLTL formula over inputs $\mathcal{O}(p_{env})$ and outputs $\bigcup_{p\in P^-} \mathcal{O}_p$ is equirealizable:

$$\forall \pi \forall \pi'. \bigwedge_{1\le i\le n} \varphi_i[\pi] \wedge \bigwedge_{p\in P^-} D^{\pi,\pi'}_{\mathcal{I}(p)\mapsto\mathcal{O}(p)} \qquad \square$$




(a) non-faulty (b) $p_1 \nrightarrow p_3$ (c) $p_2 \nrightarrow p_3$

Figure 8.2: Visual interpretation of a fault-tolerance specification: On the left is the original (non-faulty) architecture where the communication between $p_1 \rightarrow p_3$ and $p_2 \rightarrow p_3$ is intact. The two architectures on the right represent the case where either $p_1 \nrightarrow p_3$ or $p_2 \nrightarrow p_3$. In this case, the receiving propositions $b$ and $c$, respectively, are controlled by the environment. In fault-tolerant synthesis, we search for strategies for processes $p_1$, $p_2$, and $p_3$ such that the specification is satisfied in all architectures.

Table 8.1: Complexity of the HyperLTL realizability problem for the fragments discussed in Section 8.3.

| $\exists^*$ | $\forall^1$ | incompl. inform. $\forall^*$ | $\exists^*\forall^1$ | linear $\forall^*$ | $\exists^*\forall^{>1}$ | $\forall^*$ | $\forall^*\exists^*$ |
|---|---|---|---|---|---|---|---|
| PSPACE-complete | 2EXPTIME-complete | | 3EXPTIME | non-elem. | undecidable | | |

## 8.3 Deciding HyperLTL Realizability

In this section, we identify fragments of HyperLTL for which the realizability problem is decidable. The results are summarized in Table 8.1.

We base our investigation on the structure of the quantifier prefix of the HyperLTL formulas. We call a HyperLTL formula $\varphi$ (quantifier) *alternation-free* if the quantifier prefix consists solely of either universal or existential quantifiers. We denote the corresponding fragments as the (universal) $\forall^*$ and the (existential) $\exists^*$ fragment, respectively. A HyperLTL formula is in the $\exists^*\forall^*$ fragment, if it starts with arbitrarily many existential quantifiers, followed by arbitrarily many universal quantifiers. Analogously for the $\forall^*\exists^*$ fragment. For a given natural number $n$, we refer to a bounded number of quantifiers with $\forall^n$, respectively $\exists^n$. The $\forall^1$ realizability problem is equivalent to the LTL realizability problem.

### 8.3.1 $\exists^*$ Fragment

The realizability problem for existential HyperLTL is PSPACE-complete. This can be shown by a reduction of the realizability problem to the satisfiability problem for bounded one-alternating $\exists^*\forall^2$HyperLTL [FH16], i.e., finding a trace set $T$ such that $T \vDash \varphi$. For completeness, we recap the proof given in [Fin+18a; Fin+19a].

**Lemma 8.7** ([Fin+18a]). *An existential HyperLTL formula $\varphi = \exists \pi_1 \cdots \exists \pi_n.\ \psi$ is realizable if, and only if, $\varphi_{sat} \coloneqq \exists \pi_1 \cdots \exists \pi_n.\ \forall \pi \forall \pi'.\ \psi \wedge D^{\pi,\pi'}_{I \mapsto O}$ is satisfiable.*

*Proof.* Assume $f \colon (2^I)^* \to 2^O$ realizes $\varphi$, that is $f \vDash \varphi$. Let $T = \mathit{traces}(f)$ be the set of traces generated by $f$. It holds that $T \vDash \varphi$ and $T \vDash \forall \pi, \pi'.\ D^{\pi,\pi'}_{I \mapsto O}$. Therefore, $T$ witnesses the satisfiability of $\varphi_{sat}$. For the reverse direction, assume that $\varphi_{sat}$ is satisfiable. Let $S$ be a set of traces that satisfies $\varphi_{sat}$. We construct a strategy $f \colon (2^I)^* \to 2^O$ as

$$f(\sigma) = \begin{cases} w_{|\sigma|} \cap O & \text{if } \sigma \text{ is a prefix of some } w|_I \text{ with } w \in S \\ \varnothing & \text{otherwise} \end{cases}$$

where $w|_I$ denotes the trace restricted to $I$, meaning that $w_i \cap I$ for all $i \geq 0$. Note that if there are multiple candidates $w \in S$, then $w_{|\sigma|} \cap O$ is equal across all of them due to the required determinism $\forall \pi \forall \pi'.\ D^{\pi,\pi'}_{I \mapsto O}$. By construction, all traces in $S$ are contained in $f$, and together with $S \vDash \varphi$, it holds that $f \vDash \varphi$ as the sets of sets of traces satisfying the existential formula $\varphi$ are upward closed. □

**Theorem 8.8** ([Fin+18a]). *Realizability of existential HyperLTL specifications is PSpace-complete.*

*Proof.* Given an existential HyperLTL formula, we gave a linear reduction to the satisfiability of the $\exists^* \forall^2$ fragment in Lemma 8.7. The satisfiability problem for a bounded number of universal quantifiers is in PSpace [FH16]. Hardness follows from LTL satisfiability [SC85], which is equivalent to the $\exists^1$ fragment. □

### 8.3.2 $\forall^*$ Fragment

In the following, we will use the *distributed synthesis* problem defined above, i.e., the problem whether there is an implementation of processes in a distributed architecture (cf. Figure 8.1) that satisfies an LTL formula.

**Corollary 8.9** ([Fin+18a]). *The synthesis problem for universal HyperLTL becomes undecidable as soon as we have more than one universal quantifier.*

*Proof.* Follows from Theorem 8.3 and the undecidability of distributed synthesis for LTL [PR90]. □

It tuns out that the reduction works the other way, too: Using the idea of *collapsing* quantifiers [Fin+18a], it is possible to check whether the HyperLTL realizability problem can be reduced to the distributed synthesis problem and, thus, one can characterize a decidable fragment of universal HyperLTL. Given a universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n.\ \psi$, we define the COLLAPSED formula of $\psi$ as $\mathit{collapse}(\psi) \coloneqq \psi[\pi_1 \mapsto \pi][\pi_2 \mapsto \pi] \ldots [\pi_n \mapsto \pi]$ where $\psi[\pi_i \mapsto \pi]$ replaces all occurrences of $\pi_i$ in $\psi$ with $\pi$. Further, we define $\mathit{collapse}(\varphi) = \forall \pi.\ \mathit{collapse}(\psi)$. While there are formulas that are equivalent to their collapsed form, e.g., $\forall \pi \forall \pi'.\ \square a_\pi \wedge \square b_{\pi'}$, this does not hold in general as shown by the formula $\forall \pi.\ \forall \pi'.\ \square(a_\pi \leftrightarrow a_{\pi'})$. If a HyperLTL formula is not equivalent to its collapsed form, then this formula is not expressible using one universal quantifier.

**Lemma 8.10** ([Fin+18a]). *Either $\varphi \equiv collapse(\varphi)$ or $\varphi$ has no equivalent $\forall^1$ formula.*

Note that equivalence of universal HyperLTL formulas can be checked in ExpSpace [FH16]. Using this, we can recap the definition the *linear* $\forall^*$ fragment [Fin+18a], a subclass of universal formulas for which the realizability problem is decidable: A universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n . \psi$ is in the LINEAR FRAGMENT of $\forall^*$ if and only if, for all output propositions $o_i \in O$ there is a subset of input propositions $J_i \subseteq I$ such that

$$\forall \pi_1 \cdots \forall \pi_n . \psi \wedge D^{\pi_1,\pi_2}_{I \mapsto O} \equiv \forall \pi \forall \pi' . collapse(\psi) \wedge \bigwedge_{o_i \in O} D^{\pi,\pi'}_{J_i \mapsto \{o_i\}}$$

and $J_i \subseteq J_{i+1}$ for all $i$. The constraint $D^{\pi_1,\pi_2}_{I \mapsto O}$ is added to $\varphi$ to enforce input-determinism for the equivalence check. This is no restriction for the realizability problem as strategies are input-deterministic.

**Example 8.11** ([Fin+18a]). An example of a formula in the linear fragment of $\forall^*$ is $\varphi = \forall \pi \forall \pi' . \psi$ with $\psi = D^{\pi,\pi'}_{\{a\} \mapsto \{c\}} \wedge \square(c_\pi \leftrightarrow d_\pi) \wedge \square(b_\pi \leftrightarrow \bigcirc e_\pi)$, $I = \{a, b\}$, and $O = \{c, d, e\}$. The corresponding formula asserting input-determinism is $\varphi_{det} = \forall \pi \forall \pi' . \psi \wedge D^{\pi,\pi'}_{I \mapsto O}$. One possible choice of $J$'s is $\{a, b\}$ for $c$, $\{a\}$ for $d$ and $\{a, b\}$ for $e$. Note, that one can use either $\{a, b\}$ or $\{a\}$ for $c$ as $\forall \pi \forall \pi' . D^{\pi,\pi'}_{\{a\} \mapsto \{d\}} \wedge (c_\pi \leftrightarrow d_\pi)$ implies $\forall \pi \forall \pi' . D^{\pi,\pi'}_{\{a\} \mapsto \{c\}}$. However, the apparent alternative $\{b\}$ for $e$ would yield an undecidable architecture with information fork. It holds that $\varphi_{det}$ and $\forall \pi \forall \pi' . collapse(\psi) \wedge D^{\pi,\pi'}_{\{a,b\} \mapsto \{c\}} \wedge D^{\pi,\pi'}_{\{a\} \mapsto \{d\}} \wedge D^{\pi,\pi'}_{\{a,b\} \mapsto \{e\}}$ are equivalent and, thus, that $\varphi$ is in the linear fragment.

**Theorem 8.12** ([Fin+18a]). *The realizability of the linear $\forall^*$ fragment of HyperLTL can be decided in non-elementary time.*

From this observation, we can derive further fragments of HyperLTL. In the following, we define the *incomplete information* fragment of $\forall^*$ HyperLTL and show that the decision problem is 2ExpTime-complete, i.e., no harder than LTL. This fragment includes properties like observational determinism [ZM03], which can be expressed in HyperLTL [Cla+14] as

$$\forall \pi \forall \pi' . \square(I^{obs}_\pi = I^{obs}_{\pi'}) \rightarrow \square(O^{obs}_\pi = O^{obs}_{\pi'})$$

stating that, for every pair of traces, if the observable inputs are the same, then the observable outputs must be same as well. A universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n . \psi$ is in the INCOMPLETE INFORMATION FRAGMENT of $\forall^*$ if and only if, there is a subset of input propositions $H \subseteq I$ such that

$$\forall \pi_1 \cdots \forall \pi_n . \psi \wedge D^{\pi_1,\pi_2}_{I \mapsto O} \equiv \forall \pi \forall \pi' . collapse(\psi) \wedge D^{\pi,\pi'}_{I \setminus H \mapsto O} \ .$$

**Theorem 8.13.** *The decision problem of formulas in the incomplete information fragment is 2ExpTime-complete.*

*Proof.* Given a universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n . \psi$ in the incomplete information fragment. Due to the proof of Theorem 8.2, the decision problem of $\forall \pi \forall \pi' . \mathit{collapse}(\psi) \wedge D^{\pi,\pi'}_{I \setminus H \mapsto O}$ can be reduced to the decision problem of synthesis under incomplete information [KV97] whose decision problem is in 2ExpTime. Hardness follows from LTL realizability. □

### 8.3.3 $\exists^* \forall^1$ Fragment

In this fragment, we consider arbitrary many existential trace quantifiers followed by a single universal trace quantifier. This fragment turns out to be still decidable. We solve the realizability problem for this fragment by reducing it to a decidable fragment of the distributed realizability problem for LTL.

**Theorem 8.14.** *Realizability of $\exists^* \forall^1$ HyperLTL specifications is decidable.*

*Proof.* Let $\varphi$ be $\exists \pi_1 \cdots \exists \pi_n . \forall \pi' . \psi$. We reduce the realizability problem of $\varphi$ to the distributed realizability problem for LTL. Intuitively, we use a two-process distributed architecture where the first process $p$ is supposed to produce the traces for the leading existential quantification and the second process $p'$ represents the realizing strategy. The architecture is depicted in Figure 8.3.

For every existential trace quantifier $\pi$, we introduce a copy of every atomic proposition for the distributed realizability problem, written $a^\pi$ for $a \in \mathit{AP}$. We use the same notation for sets of atomic propositions, e.g., $I^\pi = \{i^\pi \mid i \in I\}$. Process $p$ has no inputs, thus, produces only a single trace, and it controls the outputs $\bigcup_{1 \leq i \leq n} (I^{\pi_i} \cup O^{\pi_i})$. Using an appropriate valuation of its outputs, process $p$ selects the paths in the strategy tree corresponding to the existential trace quantifiers $\exists \pi_1 \cdots \exists \pi_n$. Thus, those output propositions of process $p$ have to encode an actual path in the strategy tree produced by $p'$. To ensure this, we add the LTL constraint $\square(I^{\pi_i} = I) \rightarrow \square(O^{\pi_i} = O)$ that asserts that if the inputs correspond to some path in the strategy tree, the outputs on those paths have to be the same. The resulting architecture $A_\varphi$ is

$$\langle \{p_{env}, p, p'\}, p_{env}, \{p \mapsto \varnothing, p' \mapsto I\}, \{p_{env} \mapsto I, p \mapsto \bigcup_{1 \leq i \leq n} (I^{\pi_i} \cup O^{\pi_i}), p' \mapsto O\} \rangle \ .$$

It is easy to verify that $A_\varphi$ does not contain an information fork, thus the realizability problem is decidable [FS05]. The LTL specification $\theta$ is $\psi \wedge \bigwedge_{1 \leq i \leq n} \square(I^{\pi_i} = I) \rightarrow \square(O^{\pi_i} = O)$ where we replace every $a_\pi$ by $a^\pi$ for existential traces and $a_{\pi'}$ to $a$ in $\psi$. The implementation of process $p'$ (if it exists) is a realizing strategy for the HyperLTL formula (process $p$ producing witnesses for the $\exists$ quantifiers): Assume that there are realizing strategies for $\langle A_\varphi, \theta \rangle$, i.e., $f_p \colon (2^\varnothing)^* \rightarrow 2^{\bigcup_{1 \leq i \leq n} (I^{\pi_i} \cup O^{\pi_i})}$ and $f_{p'} \colon (2^I)^* \rightarrow 2^O$. $f_{p'}$ is a realizing strategy for $\varphi$ as well: By the HyperLTL semantics, we have to show that there is a trace assignment $\Pi \colon \mathcal{V}_\exists \rightarrow \mathit{traces}(f_{p'})$ such that for all $t \in \mathit{traces}(f_{p'})$ it holds that $\Pi[\pi' \rightarrow t] \vDash_{\mathit{traces}(f_{p'})} \psi$. We define $\Pi$ in the following. Note that $\mathit{traces}(f_p)$ is a singleton set and let $t_p \in \left(2^{\bigcup_{1 \leq i \leq n} (I^{\pi_i} \cup O^{\pi_i})}\right)^\omega$ be the corresponding trace. For every

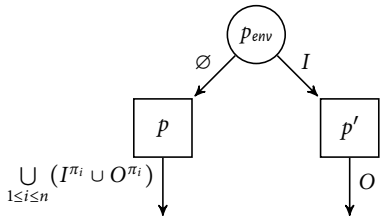



Figure 8.3: Visualization of the architecture used in the $\exists^* \forall^1$ reduction in the proof of Theorem 8.14.

$\pi_i \in \{\pi_1, \dots, \pi_n\}$, we define $\Pi(\pi_i) = t_p|_{I^{\pi_i}}$ where we replace $a^{\pi_i}$ by $a$ for every $a \in \text{AP}$. This, together with $\theta$ shows that $\psi$ holds for every chosen path $t \in \mathit{traces}(f_{p'})$ for $\pi'$.

Conversely, a model for $\varphi$ can be used as an implementation of $p$ and $p'$: Let $f\colon (2^I)^* \to 2^O$ be a realizing strategy of $\varphi$. We use $f$ as a strategy for $p'$. We construct the single trace produced by $p$ using the existential trace assignment $\Pi\colon \mathcal{V}_\exists \to \mathit{traces}(f)$. Let $t_1, \dots, t_n \in \mathit{traces}(f)$ be the corresponding traces. We construct a single trace $t_p$ by replacing propositions $a \in \text{AP}$ by $a^{\pi_i}$ for every $t_i$ and the subsequent union of the resulting traces (which now have pairwise disjoint propositions). Due to the construction, $f_p$ satisfies $\bigwedge_{1 \le i \le n} \square(I^{\pi_i} = I) \to \square(O^{\pi_i} = O)$ and thus, the distributed architecture satisfies $\theta$. Hence, the distributed synthesis problem $\langle A_\varphi, \theta \rangle$ has a solution if, and only if, $\varphi$ is realizable. □

### 8.3.4 $\forall^* \exists^*$ Fragment

To complete the characterization of decidability results, we briefly state the result for formulas in the $\forall^* \exists^*$ fragment. Whereas the $\exists^* \forall^1$ fragment remains decidable, the realizability problem of $\forall^* \exists^*$ turns out to be undecidable even when restricted to only one quantifier of both sorts ($\forall^1 \exists^1$). The undecidability proof uses a reduction from Post's Correspondence Problem (PCP).

**Theorem 8.15** ([Fin+18a]). *Realizability of $\forall^* \exists^*$ HyperLTL is undecidable.*

## 8.4 Summary

In this section, we investigated temporal hyperproperties, which are properties relating multiple observation traces. We showed that the realizability problem for the temporal hyperlogic HyperLTL subsumes many earlier extensions of the LTL realizability problem, including realizability under incomplete information, distributed realizability, and symmetric synthesis. Further, we analyzed the complexity of the decision problem for (unbounded) synthesis based on the structure of the quantifier prefix. In the following section, we introduce a semi-decision procedure for realizability and unrealizability.

# Chapter 9

# Bounded Synthesis from Hyperproperties

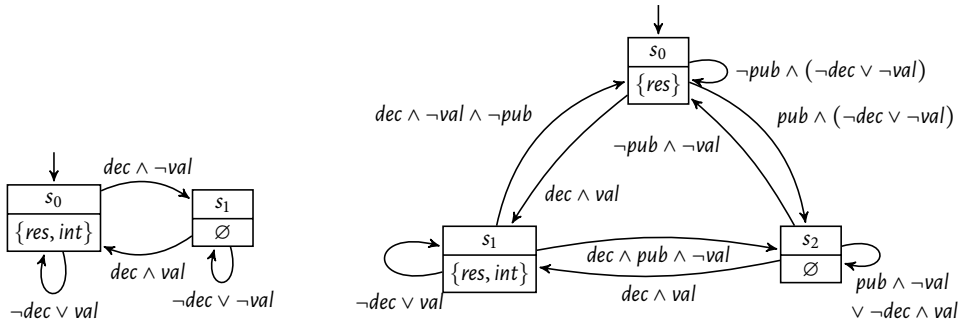
In the previous section, we have seen that HyperLTL is an immensely expressive specification language—at the cost of giving up decidability. In this section, we provide an alternative view of the synthesis procedure that is motivated by considering the *output complexity* as pioneered by Finkbeiner and Schewe [FS13]. In this view, we measure the complexity of the synthesis procedure in the size of the system to-be-synthesized as it is done in model checking. In this model, we derive complexity results that are analogous to model checking HyperLTL: In the same way that the model checking of universal HyperLTL has the same asymptotical complexity as the corresponding problem for LTL, that is, NLogSpace-complete [FRS15], we show that the realizability problem for universal HyperLTL is in the same complexity class as LTL [FS13], namely in NP.

Using the bounded synthesis [FS07; SF07; FS13; Fay+17] approach discussed in Section 7.2.2, we derive a semi-decision procedure for universal HyperLTL in Section 9.1. Based on the bounded unrealizability method [Ten13; FT14a; FT15], we derive a method to detect unrealizability of universal HyperLTL specifications in Section 9.2. Lastly, we consider the synthesis problem for HyperLTL with quantifier alternations in Section 9.3.

This chapter is based on work published in the proceedings of CAV [Fin+18a; Coe+19] and an article submitted to Acta Informatica [Fin+19a].

## 9.1 Synthesis from Universal HyperLTL

**Overview.** We first sketch the synthesis procedure and then proceed with a description of the intermediate steps. Let $\varphi$ be a universal HyperLTL formula $\forall \pi_1 \cdots \forall \pi_n . \psi$. We build the automaton $\mathcal{A}_\psi$ whose language is the set of $n$-tuples of traces that satisfy $\psi$. We then define the acceptance of a transition system $\mathcal{S}$ on $\mathcal{A}_\psi$ by means of the self-composition of $\mathcal{S}$. Lastly, we encode the existence of a transition system accepted by $\mathcal{A}_\psi$ as a constraint system.




(a) Synthesized transition system from LTL formula in Equation 9.1.

(b) Synthesized transition system from the conjunction of LTL specification in Equation 9.1 and HyperLTL specification in Equation 9.2.

Figure 9.1: Synthesized state-labeled transition systems based on the specification given in Example 9.1.

**Example 9.1.** Throughout this section, we will use the following (simplified) running example. Assume we want to synthesize a system that keeps decisions secret until it is allowed to publish. Thus, our system has three input signals *decision*, indicating whether a decision was made, the secret *value*, and a signal to *publish* results. Furthermore, our system has two outputs, an undisclosed output *internal* that stores the value of the last decision, and a public output *result* that indicates the result. No information about decisions should be inferred until publication. To specify the functionality, we propose the LTL specification

$$\begin{aligned} &\square(decision \rightarrow (value \leftrightarrow \bigcirc internal)) \\ \wedge\, &\square(\neg decision \rightarrow (internal \leftrightarrow \bigcirc internal)) \\ \wedge\, &\square(publish \rightarrow \bigcirc(internal \leftrightarrow result)) \ . \end{aligned} \tag{9.1}$$

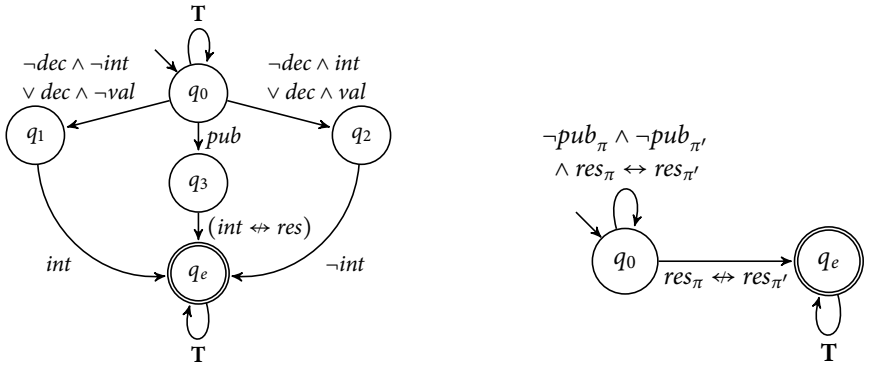
The solution produced by the LTL synthesis tool BoSy (Section 7.4.1), shown in Figure 9.1a, clearly violates our intention that results should be secret until publish: Whenever a decision is made, the output *result* changes as well.

We formalize the property that no information about the decision can be inferred from *result* until publication as the HyperLTL formula

$$\forall \pi \forall \pi' . \, (publish_\pi \vee publish_{\pi'}) \, \mathcal{R} (result_\pi \leftrightarrow result_{\pi'}) \ . \tag{9.2}$$

It asserts that for every pair of traces, the *result* signals have to be the same until (if ever) there is a *publish* signal on either trace. The universal co-Büchi automata for the LTL and HyperLTL specifications are depicted in Figure 9.2.

A solution satisfying both, the functional specification and the hyperproperty, is shown in Figure 9.1b. The system switches states whenever there is a decision with a different value than before and only exposes the decision in case there is a prior publish command.




(a) Automaton accepting the language defined by the LTL formula in Equation 9.1.

(b) Automaton accepting the language defined by the HyperLTL formula in Equation 9.2.

Figure 9.2: Universal co-Büchi automata recognizing the languages from Example 9.1.

Before discussing our synthesis approach, we proceed with discussing the model checking of universal HyperLTL using self-composition.

**Self-composition.** The model checking of universal HyperLTL formulas [FRS15] is based on self-composition [BDR11]. Let $prj_i$ be the projection to the $i$-th element of a tuple. Let $zip$ denote the usual function that maps an $n$-tuple of sequences to a single sequence of $n$-tuples, for example, $zip([1,2,3],[4,5,6]) = [(1,4),(2,5),(3,6)]$, and let $unzip$ denote its inverse. The $n$-fold self-composition of $\mathcal{S} = \langle S, s_0, \tau, l\rangle$, written $\mathcal{S}^n$, is defined as $\langle S^n, s_0^n, \vec{\tau}, \vec{l}\rangle$ where $\vec{\tau}\colon S^n \times (2^I)^n \to S^n$ and $\vec{l}\colon S^n \to (2^O)^n$ are defined such that for all $\vec{s}, \vec{s}' \in S^n$, $\vec{\boldsymbol{i}} \in (2^I)^n$, and $\vec{\boldsymbol{o}} \in (2^O)^n$ we have that $\vec{\tau}(\vec{s}, \vec{\boldsymbol{i}}) = \vec{s}'$ and $\vec{l}(\vec{s}) = \vec{\boldsymbol{o}}$ iff for all $1 \leq i \leq n$, it holds that $\tau(prj_i(\vec{s}), prj_i(\vec{\boldsymbol{i}})) = prj_i(\vec{s}')$ and $l(prj_i(\vec{s})) = prj_i(\vec{\boldsymbol{o}})$. The set of traces generated by $\mathcal{S}^n$ is $traces(\mathcal{S}^n) = \{zip(t_1, \dots, t_n) \mid t_1, \dots, t_n \in traces(\mathcal{S})\}$

For a quantifier-free HyperLTL formula $\psi$, we construct the universal co-Büchi automaton $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi)$ is the set of infinite words $\sigma$ such that $unzip(\sigma) \vDash \psi$, i.e., the tuple of traces satisfies $\psi$ (see for example Figure 9.2b). We get this automaton by dualizing the non-deterministic Büchi automaton for $\neg\psi$ [KV05; FS13], i.e., changing the branching from non-deterministic to universal and the acceptance condition from Büchi to co-Büchi. Hence, $\mathcal{S}$ satisfies a universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n.\, \psi$ if the traces generated by self-composition $\mathcal{S}^n$ are a subset of $\mathcal{L}(\mathcal{A}_\psi)$.

**Lemma 9.2.** *A transition system $\mathcal{S}$ satisfies the universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n.\, \psi$, if, and only if, the run graph of $\mathcal{S}^n$ on $\mathcal{A}_\psi$ is accepting.*

*Proof.* $\mathcal{S} \vDash \varphi$ if, and only if, $\mathcal{S}^n$ is accepted by $\mathcal{A}_\psi$ [Cla+14]. The correctness of the run graph is established in Lemma 7.7. □

**Synthesis.** Let $\mathcal{S} = \langle S, s_0, \tau, l\rangle$ and $\mathcal{A}_\psi = \langle Q, q_0, \delta, F\rangle$. We encode the synthesis problem as a constraint system in a decidable first-oder theory. Therefore, we use uninterpreted function symbols to encode the transition system and the annotation. For the

transition system, those functions are the transition function $\tau\colon S \times 2^I \to S$ and the labeling function $l : S \to 2^O$. The annotation is split into two parts, a reachability constraint $\lambda^{\mathbb{B}}\colon S^n \times Q \to \mathbb{B}$ indicating whether a vertex in the run graph is reachable and a counter $\lambda^{\#}\colon S^n \times Q \to \mathbb{N}$ that maps every reachable vertex to the maximal number of rejecting vertices visited by any path starting in the initial vertex. The resulting constraint asserts that there is a transition system with an accepting run graph:

$$\forall \vec{s}, \vec{s}' \in S^n.\ \forall q, q' \in Q.\ \forall \vec{\boldsymbol{i}} \in (2^I)^n.$$
$$\lambda^{\mathbb{B}}((s_0)^n, q_0) \wedge$$
$$\left(\lambda^{\mathbb{B}}(\vec{s}, q) \wedge \vec{\tau}(\vec{s}, \vec{\boldsymbol{i}}) = \vec{s}' \wedge (q, \vec{\boldsymbol{i}} \cup \vec{l}(\vec{s}), q') \in \delta\right) \to \lambda^{\mathbb{B}}(\vec{s}', q') \wedge \lambda^{\#}(\vec{s}', q') \trianglerighteq \lambda^{\#}(\vec{s}, q)$$

where $\trianglerighteq$ is $>$ if $q' \in F$ and $\geq$ otherwise.

**Theorem 9.3.** *The constraint system is satisfiable with bound $b = |S|$ if, and only if, there is a transition system $\mathcal{S}$ of size $b$ that realizes the HyperLTL formula.*

*Proof.* If the constraint system is satisfiable, the satisfying interpretation of $\lambda^{\#}$ and $\lambda^{\mathbb{B}}$ represent a valid annotation for the run graph $\mathcal{S}^n \times \mathcal{A}_\psi$, where $\mathcal{S}$ is constructed from satisfying interpretations of $\tau$ and $l$. For the reverse direction, note that for every transition system $\mathcal{S}$ with $|S| = b$, there is an interpretation of $\tau$ and $l$ that represents this transition system, i.e., an unsatisfiable constraint system rules out any transition system of size $b$. Further, there is an upper bound on the number of $\lambda^{\#}$ when the size of $\mathcal{S}$ is fixed [FS13]. □

We extract a realizing implementation by asking the satisfiability solver to generate a model for the uninterpreted functions that encode the transition system.

**Theorem 9.4.** *Let $\varphi$ be a universal HyperLTL formula. Deciding whether there exists a realizing transition system $\mathcal{S}$ of size $b = |S|$ is NP-complete.*

**Theorem 9.5.** *The constraint system after removing syntactic sugar, i.e., the quantifications over finite domains, is polynomial in $b$. Further, it is propositional up to the* difference *constraints $X - Y \leq c$ for some bound $c \in \{0, 1\}$. Deciding propositional logic with difference constraints is NP-complete [NO05]. Hardness follows from NP-hardness of the corresponding problem for LTL [FS13].*

## 9.2 Bounded Unrealizability

While the previous section was concerned with the existence of (small) system strategies, we now shift our focus on environment strategies. In the case of LTL, Chapter 7 already gave a satisfying answer as the realizability problem is dual. Already for universal HyperLTL this is no longer the case: To show unrealizability, the environment player has to produce, with the knowledge of the system player's strategy, a *set of traces* such that the quantifier-free formula is violated. On the other hand, unrealizability is of great importance: Especially for distributed architectures and complex specifications such as fault-tolerance, the search for environment counter-strategies becomes more important as it

indicates problems early in the design process. In earlier work, we introduced the concept of *counterexamples to realizability* [Ten13; FT14a; FT15] for distributed realizability and developed methods to derive small counterexamples automatically.

We adapt the definition of counterexamples to realizability for LTL [FT14a] to HyperLTL in the following. Let $\varphi$ be a universal HyperLTL formula $\forall \pi_1 \cdots \forall \pi_n. \psi$ over inputs $I$ and outputs $O$, a *counterexample to realizability* is a set of input traces $\mathcal{P} \subseteq (2^I)^\omega$ such that for every strategy $f: (2^I)^* \to 2^O$ the labeled traces $\mathcal{P}^f \subseteq (2^{I \cup O})^\omega$ satisfy $\neg\varphi = \exists \pi_1 \cdots \exists \pi_n. \neg\psi$.

**Proposition 9.6.** *A universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n. \psi$ is unrealizable if, and only if, there is a counterexample $\mathcal{P}$ to realizability.*

*Proof.* Let $\mathcal{P}$ be a counterexample to realizability. Assume for contradiction that $\varphi$ is realizable by a strategy $f$. By definition of $\mathcal{P}$, we know that $\mathcal{P}^f \vDash \exists \pi_1 \cdots \exists \pi_n. \neg\psi$. This means that there exists an assignment $\Pi_{\mathcal{P}}: \mathcal{V} \to \mathcal{P}^f$ with $\Pi_{\mathcal{P}}, 0 \vDash_{\mathcal{P}^f} \neg\psi$, which is equivalent to $\Pi_{\mathcal{P}} \nvDash_{\mathcal{P}^f} \psi$. Therefore, not all assignments $\Pi : \mathcal{V} \to \mathcal{P}^f$ satisfy $\Pi, 0 \vDash_{\mathcal{P}^f} \psi$, which implies that $\mathcal{P}^f \nvDash \forall \pi_1 \cdots \forall \pi_n. \psi = \varphi$. Hence, $f \nvDash \varphi$, which concludes the contradiction.

Let $\varphi$ be unrealizable. We show that the set $\mathcal{P} = (2^I)^\omega$ is a counterexample to realizability. Let $f: (2^I)^* \to 2^O$ be an arbitrary strategy, and let $\mathcal{P}^f$ be the corresponding set of labeled traces. From the unrealizability of $\varphi$, we now that $f \nvDash \forall \pi_1 \cdots \forall \pi_n. \psi$. Thus, there exists a trace assignment $\Pi_{\mathcal{P}}: \mathcal{V} \to \mathcal{P}^f$ with $\Pi_{\mathcal{P}}, 0 \vDash_{\mathcal{P}^f} \neg\psi$, which is equivalent to $\mathcal{P} \vDash \exists \pi_1 \cdots \exists \pi_n. \neg\psi$. □

Despite being independent of strategy trees, there are in many cases finite representations of $\mathcal{P}$. Consider, for example, the unrealizable specification $\varphi_1 = \forall \pi \forall \pi'. \Diamond(i_\pi \leftrightarrow i_{\pi'})$, where the set $\mathcal{P}_1 = \{\varnothing^\omega, \{i\}^\omega\}$ is a counterexample to realizability. As a second example, consider $\varphi_2 = \forall \pi \forall \pi'. \square(o_\pi \leftrightarrow o_{\pi'}) \wedge \square(i_\pi \leftrightarrow \bigcirc o_\pi)$ with conflicting requirements on $o$. $\mathcal{P}_1$ is a counterexample to realizability for $\varphi_2$ as well: By choosing a different valuation of $i$ in the first step of $\mathcal{P}_1$, the system is forced to either react with different valuations of $o$ (violating the first conjunct), or not correctly repeating the initial value of $i$ (violating the second conjunct).

There are, however, already linear specifications where the set of counterexample traces is not finite and depends on the strategy tree [FT15]. For example, the specification

$$\forall \pi. \Diamond(o_\pi \leftrightarrow \bigcirc i_\pi) \tag{9.3}$$

is unrealizable as the system cannot predict future values of the environment. There is no finite set of traces witnessing this: For every finite set of traces, there is a strategy tree such that $\Diamond(o_\pi \leftrightarrow \bigcirc i_\pi)$ holds on every such trace. On the other hand, there is a simple *counterexample strategy*, that is a strategy that observes output sequences and produces inputs, depicted in Figure 9.3. In this example, the counterexample strategy inverts the outputs given by the system, thus it is guaranteed that $\square(o \leftrightarrow \bigcirc i)$ for every system strategy.

We combine those two approaches, selecting counterexample traces and using strategic behavior. A $k$-counterexample strategy for $\forall^n$HyperLTL observes $k$ output sequences and produces $k$ inputs, where $k$ is a new parameter. We require that $k$ is at

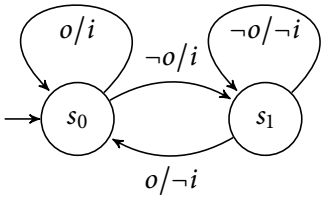



Figure 9.3: Counterexample strategy for the LTL formula given in Equation 9.3.

least the number of universal quantifiers $n$. The counterexample strategy is winning if (1) either the traces given by the system player do not correspond to a strategy, or (2) the body of the HyperLTL formula is violated for any $n$ subset of $k$ traces. Regarding property (1), consider the two traces where the system player produces different outputs initially. Clearly, those two traces cannot be generated by any system strategy since the initial state (root labeling) is fixed.

We reduce the search for a $k$-counterexample strategy to LTL synthesis. For every atomic proposition $a \in \text{AP}$, we produce $k$ copies $a^1, \dots, a^k$. We use the same notation for sets of atomic propositions, e.g., $I^j = \{i^j \mid i \in I\}$ for $1 \leq j \leq k$. The search for a $k$-counterexample strategy can be reduced to LTL synthesis using $k$-tuple input propositions $O^k$, $k$-tuple output propositions $I^k$, and the formula

$$\mathit{strategic}(I^k, O^k) \rightarrow \bigvee_{P \subseteq \{1,\dots,k\} \text{ with } |P|=n} \neg\psi[P] \ ,$$

where $\psi[P]$ denotes the replacement of $a_{\pi_i}$ by the $P_i$th position of the combined input/output $k$-tuple. The formula $\mathit{strategic}(I^k, O^k)$ enforces that the behavior of the system player is strategic and is defined as

$$\bigwedge_{1 \leq j_1 < j_2 \leq k} \left( \bigvee_{i \in I} (i^{j_1} \leftrightarrow i^{j_2}) \right) \mathcal{R} \left( \bigwedge_{o \in O} (o^{j_1} \leftrightarrow o^{j_2}) \right) \ .$$

This is an instance of the formula (sym) given in Section 8.2.

**Theorem 9.7.** *A universal HyperLTL formula $\varphi = \forall \pi_1 \cdots \forall \pi_n . \psi$ is unrealizable if there is a $k$-counterexample strategy for some $k \geq n$.*

*Proof.* Fix $\varphi$ and let $f_{cex} \colon (2^{O^k})^+ \rightarrow 2^{I^k}$ be a $k$-counterexample strategy. Assume for contradiction that $f \colon (2^I)^* \rightarrow 2^O$ is a strategy realizing $\varphi$. Let $f^k \colon (2^{I^k})^* \rightarrow 2^{O^k}$ be the strategy that represents the $k$-fold self-composition of $f$ (adapting atomic propositions as described earlier). By combining $f$ and $f_{cex}$, we get an infinite sequence $t \in (2^{I^k \cup O^k})^\omega$: $t_0 = f(\epsilon) \cup f_{cex}(f(\epsilon))$, $t_1 = f(f_{cex}(f(\epsilon))) \cup f_{cex}(f(f_{cex}(f(\epsilon))))$, ... This sequence represents a $k$-tuple $cex_k = (I \cup O)^k$. As $f^k$ satisfies $\mathit{strategic}(I^k, O^k)$, there is a $n$-tuple $cex_n$ build from elements of $cex_k$ such that for the corresponding trace assignment $\Pi$ it holds that $\Pi, 0 \vDash_{traces(f)} \neg\psi$. This contradicts our assumption that $f$ is a realizing strategy. □

## 9.3 Synthesis from HyperLTL with Quantifier Alternations

The alternation-free fragment of HyperLTL does not cover all hyperproperties. For example, universal HyperLTL cannot express the secrecy property that given a trace of the system, there has to be an alternative trace with the same valuation of observable variables while having a different valuation of the undisclosed, that is, secret variables. In HyperLTL, this property can be expressed with a formula that has a $\forall\exists$ quantifier alternation, where the existential trace quantifier selects a suitable alternative trace. More generally, those properties can be characterized as hyperliveness [CS10]. In this section, we show how to reduce existential trace quantification to strategic choice that approximates the HyperLTL semantics where the $\exists$-player has access to the complete trace assignment, i.e., infinite traces. To cope with this incompleteness, we introduce a bounded *lookahead* as a variant of prophecy. This section is based on work published in the proceedings of CAV [Coe+19].

The model checking problem of $\forall\pi.\,\exists\pi'.\,\psi$ can be viewed as a game between the $\forall$-player and the $\exists$-player [Coe+19]. For every trace $\pi$ chosen by the $\forall$-player by a traversal of the state space of the transition system, the $\exists$-player has to build a trace $\pi'$ such that the pair $\langle\pi, \pi'\rangle$ satisfies $\psi$. We use an approximation of this game by limiting the $\exists$-player to *strategic* choice, i.e., she can only act on the finite prefix observed instead of the complete knowledge of $\pi$. While the existence of a winning strategy for the $\exists$-player implies that $\forall\pi.\,\exists\pi'.\,\psi$ is satisfiable, the converse is not true as a satisfying/matching choice of $\pi'$ given $\pi$ may depend on a position in the future.

This game-theoretic approach provides an opportunity for the user to reduce the complexity of the model checking problem [Coe+19]: If the user provides a strategy for the $\exists$-player, then the problem reduces to the cheaper model checking problem for universal properties. We show that such strategies can also be constructed automatically using synthesis. Beyond model checking, the game-theoretic approach also provides a method for the synthesis of systems that satisfy a conjunction of hypersafety and hyperliveness properties. Here, we do not only synthesize the strategy but also construct the system itself, i.e., the game graph on which the model checking game is played. While the synthesis from $\forall^*\exists^*$ hyperproperties is known to be undecidable in general, we show that the game-theoretic approach can naturally be integrated into the bounded synthesis approach presented in Section 9.1, which checks for the existence of a correct system up to a bound on the number of states.

In our game-theoretic view on the model checking problem for $\forall^*\exists^*$HyperLTL, the $\exists$-player has an infinite lookahead. There is some work on *finite* lookahead on trace languages [KZ15]. We use the idea of finite lookahead as an approximation to construct existential strategies and give a novel synthesis construction for strategies with delay based on bounded synthesis [FS13].

### 9.3.1 Model Checking with Synthesized Strategies

**Preliminaries.** For tuples $\vec{x} \in X^n$ and $\vec{y} \in X^m$ over set $X$, we use $\vec{x} \cdot \vec{y} \in X^{n+m}$ to denote the concatenation of $\vec{x}$ and $\vec{y}$. Given a function $f\colon X \to Y$ and a tuple $\vec{x} \in X^n$, we define

by $f \circ \vec{x} \in Y^n$ the tuple $(f(\vec{x}_0), \dots, f(\vec{x}_{n-1}))$.

Given a $\Gamma$-labeled $\Upsilon$-transition system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ and a $\Gamma'$-labeled $\Upsilon'$-transition system $\mathcal{S}' = \langle S', s_0', \tau', l' \rangle$, we define the CROSS-PRODUCT $\mathcal{S} \times \mathcal{S}' = \langle S \times S', (s_0, s_0'), \tau'', l'' \rangle$ as the $\Gamma \times \Gamma'$-labeled $\Upsilon \times \Upsilon'$-transition system where $\tau''((s,s'),(v,v')) = (\tau(s,v), \tau'(s',v'))$ and $l''((s,s')) = (l(s), l'(s'))$.

Let $\mathcal{S}^* = \langle S^*, s_0^*, \tau^*, l^* \rangle$ be the transition system generating strategy $f$, then the COMPOSITION $\mathcal{S} \parallel f = \mathcal{S} \parallel \mathcal{S}^*$ is the transition system $\langle S \times S^*, (s_0, s_0^*), \tau^{\parallel}, l^{\parallel} \rangle$ where $\tau^{\parallel} \colon (S \times S^*) \times \Upsilon^* \to (S \times S^*)$ is defined as $\tau^{\parallel}((s,s^*),v^*) = (\tau(s, l^*(s^*)), \tau^*(s^*, v^*))$ and $l^{\parallel} \colon (S \times S^*) \to \Gamma$ is defined as $l^{\parallel}(s,s^*) = l(s)$ for every $s \in S$, $s^* \in S^*$, and $v^* \in \Upsilon^*$. Intuitively, the resulting transition system $\mathcal{S}^*$ produces the inputs for $\mathcal{S}$. We use the notation $zip(\psi, \pi_1, \dots, \pi_n)$ for some HyperLTL formula $\psi$ to combine the trace variables $\pi_1, \dots, \pi_n$ (occurring free in $\psi$) into a fresh trace variable $\pi^*$.

Our semi-decision procedure is based on the following substitution theorem.

**Theorem 9.8** (Strategic Substitution [Coe+19])**.** *Let $\mathcal{S}$ be a $\Upsilon$-transition system.*

- *It holds that $\mathcal{S} \vDash \forall \pi_1 \cdots \forall \pi_n. \exists \pi_1' \cdots \exists \pi_m'. \psi$ if there is a strategy $f \colon (\Upsilon^n)^* \to \Upsilon^m$ such that $\mathcal{S}^n \times (\mathcal{S}^m \parallel f) \vDash \forall \pi^*. zip(\psi, \pi_1, \dots \pi_n, \pi_1', \dots, \pi_m')$.*
- *It holds that $\mathcal{S} \vDash \exists \pi_1 \cdots \exists \pi_m. \forall \pi_1' \cdots \forall \pi_n'. \psi$ if there is a strategy $f \colon \varnothing^* \to \Upsilon^m$ such that $(\mathcal{S}^m \parallel f) \times \mathcal{S}^n \vDash \forall \pi^*. zip(\psi, \pi_1, \dots \pi_m, \pi_1', \dots, \pi_n')$.*

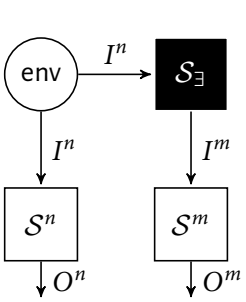
**Model Checking.** We first consider the model checking problem of $\forall^n \exists^m$HyperLTL, where we search for a strategy $f_\exists \colon (\Upsilon^n)^* \mapsto \Upsilon^m$ that produces a witness for the existential trace quantifiers. For a given HyperLTL formula of the form $\forall^n \exists^m \psi$ and a transition system $\mathcal{S}$, we search for a transition system $\mathcal{S}_\exists = \langle X, x_0, \mu, l_\exists \rangle$, where $X$ is a set of states, $x_0 \in X$ is the designated initial state, $\mu \colon X \times \Upsilon^n \to X$ is the transition function, and $l_\exists \colon X \to \Upsilon^m$ is the labeling function, such that $\mathcal{S}^n \times (\mathcal{S}^m \parallel \mathcal{S}_\exists) \vDash zip(\psi)$[1].

**Theorem 9.9.** *The strategy realizability problem for $\forall^* \exists^*$ formulas is 2ExpTime-complete.*

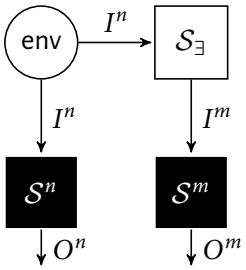
*Proof.* Given a transition system $\mathcal{S}$ and a formula $\forall^n \exists^m \psi$. We reduce the strategy synthesis problem to the problem of synthesizing a distributed reactive system with a single black-box process. The architecture is given in Figure 9.4a and the specification is given by $\psi$. This problem is decidable [FS05] and can be solved in 2ExpTime. The lower bound follows from the LTL realizability problem [PR89]. □

The decidability result implies that there is an upper bound on the size of $\mathcal{S}_\exists$ that is doubly exponential in $\psi$. Thus, the bounded synthesis approach [FS13] can be used to search for increasingly larger implementations, until a solution is found or the maximal bound is reached, yielding an efficient decision procedure for the strategy synthesis problem. In the following, we describe this approach in detail.

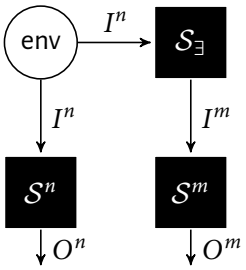
[1] We focus on $\forall^n \exists^m$HyperLTL, however, for formulas of the form $\exists^m \forall^n \psi$ the problem only differs in the input of $\mathcal{S}_\exists$ as the generated strategy is $f_\exists \colon \varnothing^* \mapsto \Upsilon^m$.




(a) The strategy synthesis problem with given implementation




(b) The implementation synthesis problem with given strategy




(c) The synthesis problem for strategy and implementation

Figure 9.4: Representation of the different synthesis problems as a distributed synthesis problem, i.e., the problem of synthesizing implementations of the black-box processes such that the joint combination with the white-box processes (which have a given implementation), satisfy the specification.

**Bounded Synthesis of Strategies.** We transform the synthesis problem into a constraint satisfaction problem, where we leave the representation of the strategy uninterpreted and challenge the solver to provide an interpretation. Given a HyperLTL formula $\forall^n \exists^m \psi$ where $\psi$ is quantifier-free, the model checking is based on the product of the $n$-fold self composition of the transition system $\mathcal{S}$, the $m$-fold self-composition of $\mathcal{S}$ where the strategy $\mathcal{S}_\exists$ controls the inputs, and the universal co-Büchi automaton $\mathcal{A}_\psi$ representing the language $\mathcal{L}(\psi)$ of $\psi$.

In more detail, the algorithm searches for a transition system $\mathcal{S}_\exists = \langle X, x_0, \mu, l_\exists \rangle$ such that the run graph of $\mathcal{S}^n$, $\mathcal{S}^m \,||\, \mathcal{S}_\exists$, and $\mathcal{A}_\psi$, written $\mathcal{S}^n \times (\mathcal{S}^m \,||\, \mathcal{S}_\exists) \times \mathcal{A}_\psi$, is accepting. Formally, given a $\Gamma$-labeled $\Upsilon$-transition system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ and a universal co-Büchi automaton $\mathcal{A}_\psi = \langle Q, q_0, \delta, F \rangle$, where $\delta \colon Q \times \Upsilon^{n+m} \times \Gamma^{n+m} \to 2^Q$, the run graph $\mathcal{S}^n \times (\mathcal{S}^m \,||\, \mathcal{S}_\exists) \times \mathcal{A}_\psi$ is the directed graph $(V, E)$, with the set of vertices $V = S^n \times S^m \times X \times Q$, initial vertex $v_{init} = (s_0^n, s_0^m, x_0, q_0)$ and the edge relation $E \subseteq V \times V$ satisfying $((\vec{s}_n, \vec{s}_m, x, q), (\vec{s}'_n, \vec{s}'_m, x', q')) \in E$ if, and only if

$$\exists \vec{v} \in \Upsilon^n. \quad \left(\vec{s}_n \xrightarrow[\tau_n]{\vec{v}} \vec{s}'_n\right) \wedge \left(\vec{s}_m \xrightarrow[\tau_m]{l_\exists(x)} \vec{s}'_m\right) \wedge \left(x \xrightarrow[\mu]{\vec{v}} x'\right) \\ \wedge q' \in \delta(q, \vec{v} \cdot l_\exists(x), l_n(\vec{s}_n) \cdot l_m(\vec{s}_m)) \ .$$

**Lemma 9.10.** *Given $\mathcal{S}$, $\mathcal{S}_\exists$, and a HyperLTL formula $\forall^n \exists^m \psi$ where $\psi$ is quantifier-free. Let $\mathcal{A}_\psi$ be the universal co-Büchi automaton for $\psi$. If the run graph $\mathcal{S}^n \times (\mathcal{S}^m \,||\, \mathcal{S}_\exists) \times \mathcal{A}_\psi$ is accepting, then $\mathcal{S} \vDash \forall^n \exists^m \psi$.*

*Proof.* Follows from Theorem 9.8 and the fact that $\mathcal{A}_\psi$ represents $\mathcal{L}(\psi)$. □

We encode the search for $\mathcal{S}_\exists$ and the annotation $\lambda$ as an SMT constraint system. Therefore, we use uninterpreted function symbols to encode $\mathcal{S}_\exists$ and $\lambda$. A transition system $\mathcal{S}$ is represented in the constraint system by two functions, the transition function

$\tau\colon S \times \Upsilon \to S$ and the labeling function $l\colon S \to \Gamma$. As before, the annotation is split into reachability $\lambda^{\mathbb{B}}\colon V \to \mathbb{B}$ and rejecting counter $\lambda^{\#}\colon V \to \mathbb{N}$. The resulting constraint asserts that there is a transition system $\mathcal{S}_\exists$, i.e., an interpretation for $\mu$ and $l_\exists$ such that the resulting run graph is accepting. Note, that the functions representing the system $\mathcal{S}$ ($\tau\colon S \times \Upsilon \to S$ and $l\colon S \to \Gamma$) are given, that is, they are interpreted.

$$
\begin{aligned}
&\forall \vec{v} \in \Upsilon^n.\ \forall \vec{s}_n, \vec{s}'_n \in S^n.\ \forall \vec{s}_m, \vec{s}'_m \in S^m.\ \forall q, q' \in Q.\ \forall x, x' \in X.\\
&\lambda^{\mathbb{B}}(s_0^n, s_0^m, x_0, q_0) \wedge\\
&\Big(\lambda^{\mathbb{B}}(\vec{s}_n, \vec{s}_m, x, q) \wedge q' \in \delta(q, (\vec{v} \cdot l_\exists(x)), (l \circ (\vec{s}_n \cdot \vec{s}_m))) \wedge x' = \mu(x, \vec{v})\\
&\quad \wedge \vec{s}'_n = \tau_n(\vec{s}_n, \vec{v}) \wedge \vec{s}'_m = \tau_m(\vec{s}_m, l_\exists(x))\Big)\\
&\Rightarrow \lambda^{\mathbb{B}}(\vec{s}'_n, \vec{s}'_m, x', q') \wedge \lambda^{\mathbb{N}}(\vec{s}_n, \vec{s}_m, x, q) \trianglerighteq \lambda^{\mathbb{N}}(\vec{s}'_n, \vec{s}'_m, x', q')
\end{aligned}
$$

where $\trianglerighteq$ is $>$ if $q' \in F$ and $\geq$ otherwise. The *bounded synthesis algorithm* increases the bound of the strategy $\mathcal{S}_\exists$ until either the constraints system becomes satisfiable, or a given upper bound is reached. In the case the constraint system is satisfiable, we can extract interpretations for the functions $\mu$ and $l_\exists$ using a solver that is able to produce models. These functions then represent the synthesized transition system $\mathcal{S}_\exists$.

**Corollary 9.11.** *Given $\mathcal{S}$ and a HyperLTL formula $\forall^*\exists^*\psi$ where $\psi$ is quantifier-free. If the constraint system is satisfiable for some bound on the size of $\mathcal{S}_\exists$ then $\mathcal{S} \vDash \forall^*\exists^*\psi$.*

*Proof.* Follows immediately by Lemma 9.10. □

As the decision problem is decidable, we know that there is an upper bound on the size of a realizing strategy transition system $\mathcal{S}_\exists$ and, thus, the bounded synthesis approach is a decision procedure for the strategy realizability problem.

**Corollary 9.12.** *The bounded synthesis algorithm decides the strategy realizability problem for $\forall^*\exists^*$ HyperLTL.*

*Proof.* The existence of such an upper bound follows from Theorem 9.9. □

**Approximating Prophecy.** We introduce a new parameter to the strategy synthesis problem to approximate the information about the future that can be captured using prophecy variables [Coe+19]. This bound represents a constant *lookahead* into future choices made by the environment. In other words, for a given $k \geq 0$, the strategy $\mathcal{S}_\exists$ is allowed to depend on choices of the $\forall$-player in the next $k$ steps. There are formulas, however, were even bounded lookahead is not enough. In the formula

$$\forall \pi.\ \exists \pi'.\ (o_{\pi'} \to i_\pi\, \mathcal{W}(i'_\pi \wedge \neg i_\pi)) \wedge (\neg o_{\pi'} \to i_\pi\, \mathcal{W}(\neg i'_\pi \wedge \neg i_\pi))$$

the environment player (controlling $i$ and $i'$) can delay the satisfaction of the right hand side of the weak until ad infinitum. While constant lookahead is only an approximation of infinite clairvoyance, it suffices for many practical situations as shown by prior case studies [FRS15; DAr+17].

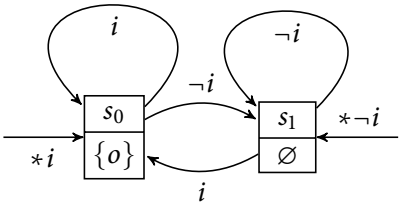



Figure 9.5: Visualization of a 2-lookahead transition system that satisfies the LTL formula given in Equation 9.3.

We present a solution to synthesize transition systems with constant lookahead for $k \geq 0$ using bounded synthesis. To simplify the presentation, we present the stand-alone problem with respect to a specification given as a universal co-Büchi automaton. The integration into the constraint system for the $\forall^*\exists^*$ HyperLTL synthesis as presented in the previous section is then straightforward. First, we present an extension to the transition system model that incorporates the notion of constant lookahead. The idea of this extension is to replace the initial state $s_0$ by a function $\mathit{init}\colon \Upsilon^k \to S$ that maps input sequences of length $k$ to some state. Thus, the transition system observes the first $k$ inputs, chooses some initial state based on those inputs, and then progresses with the same pace as the input sequence. Next, we define the run graph of such a system $\mathcal{S}_k = \langle S, \mathit{init}, \tau, l\rangle$ and an automaton $\mathcal{A} = \langle Q, q_0, \delta, F\rangle$, where $\delta\colon Q \times \Upsilon \times \Gamma \to Q$, as the directed graph $(V, E)$ with the set of vertices $V = S \times Q \times \Upsilon^k$, the initial vertices $(s, q_0, \vec{v}) \in V$ such that $s = \mathit{init}(\vec{v})$ for every $\vec{v} \in \Upsilon^k$, and the edge relation $E \subseteq V \times V$ satisfying $((s, q, v_1v_2\cdots v_k), (s', q', v_1'v_2'\cdots v_k')) \in E$ if, and only if

$$\exists v_{k+1} \in \Upsilon.\, s \xrightarrow{v_{k+1}} s' \wedge q' \in \delta(q, v_1, l(s)) \wedge \bigwedge_{1 \leq i \leq k} v_i' = v_{i+1} \ .$$

Thus, the run graph contains a prefix of the input sequence of length $k$.

**Lemma 9.13.** *Given a universal co-Büchi automaton $\mathcal{A}$ and a $k$-lookahead transition system $\mathcal{S}_k$. $\mathcal{S}_k \vDash \mathcal{A}$ if, and only if, the run graph $\mathcal{S}_k \times \mathcal{A}$ is accepting.*

Finally, synthesis amounts to solving the following constraint system:

$$\begin{aligned}
&\exists \lambda^{\mathbb{B}}\colon S \times Q \times \Upsilon^k \to \mathbb{B}.\ \exists \lambda^{\mathbb{N}}\colon S \times Q \times \Upsilon^k \to \mathbb{N}.\\
&\exists \mathit{init}\colon \Upsilon^k \to S.\ \exists \tau\colon S \times \Upsilon \to S.\ \exists l\colon S \to \Gamma.\\
&(\forall \vec{v} \in \Upsilon^k.\, \lambda^{\mathbb{B}}(\mathit{init}(\vec{v}), q_0, \vec{v})) \wedge\\
&\forall v_1v_2\cdots v_{k+1} \in \Upsilon^{k+1}.\ \forall s, s' \in S.\ \forall q, q' \in Q.\\
&\big(\lambda^{\mathbb{B}}(s, q, v_1\cdots v_k) \wedge s' = \tau(s, v_{k+1}) \wedge q' \in \delta(q, v_1, l(s))\big)\\
&\Rightarrow \lambda^{\mathbb{B}}(s', q', v_2\cdots v_{k+1}) \wedge \lambda^{\mathbb{N}}(s, q, v_1\cdots v_k) \trianglerighteq \lambda^{\mathbb{N}}(s', q', v_2\cdots v_{k+1})
\end{aligned}$$

**Corollary 9.14.** *Given some $k \geq 0$, if the constraint system is satisfiable for some bound on the size of $\mathcal{S}_k$ then $\mathcal{S}_k \vDash \mathcal{A}$.*

### 9.3.2 Synthesis with Quantifier Alternations

We now build on the introduced techniques to solve the *synthesis* problem for HyperLTL with quantifier alternation, that is, we search for implementations that satisfy the given properties. In Section 8.2, we solved the synthesis problem for $\exists^*\forall^*$ HyperLTL by a reduction to the distributed synthesis problem. We present an alternative synthesis procedure that (1) introduces the necessary concepts for the synthesis of the $\forall^*\exists^*$ fragment and that (2) strictly decomposes the choice of the existential trace quantifier from the implementation.

Fix a formula of the form $\exists^m \forall^n . \psi$. We again reduce the verification problem to the problem of determining whether a run graph is accepting. As the existential quantifiers do not depend on the universal ones, there is no future dependency and thus no need for prophecy variables or bounded lookahead. Formally, $\mathcal{S}_\exists$ is a tuple $\langle X, x_0, \mu, l_\exists \rangle$ such that $X$ is a set of states, $x_0 \in X$ is the designated initial state, $\mu \colon X \to X$ is the transition function, and $l_\exists \colon X \to \Upsilon^m$ is the labeling function. $\mathcal{S}_\exists$ produces one infinite sequence in $(\Upsilon^m)^\omega$, without having any knowledge about the behavior of the universally quantified traces. The run graph is then $(\mathcal{S}^m \,||\, \mathcal{S}_\exists) \times \mathcal{S}^n \times \mathcal{A}_\psi$. The constraint system is built analogously to Section 9.3.1, with the difference that the representation of the system $\mathcal{S}$ is now also uninterpreted. In the resulting SMT constraint system, we have two bounds, one for the size of the implementation $\mathcal{S}$ and one for the size of $\mathcal{S}_\exists$.

**Corollary 9.15.** *The bounded synthesis algorithm decides the realizability problem for $\exists^*\forall^1$ HyperLTL and is a semi-decision procedure for $\exists^*\forall^{>1}$ HyperLTL.*

The synthesis problem for formulas in the $\forall^*\exists^*$ HyperLTL fragment uses the same reduction to a constraint system as the strategy synthesis in Section 9.3.1, with the only difference that the transition system $\mathcal{S}$ itself is uninterpreted. In the resulting SMT constraint systems, we have three bounds, the size of the implementation $\mathcal{S}$, the size of the strategy $\mathcal{S}_\exists$, and the lookahead $k$.

**Corollary 9.16.** *Given a HyperLTL formula $\forall^n \exists^m . \psi$ where $\psi$ is quantifier-free. $\forall^n \exists^m . \psi$ is realizable if the SMT constraint system corresponding to the run graph $\mathcal{S}^n \times (\mathcal{S}^m \,||\, \mathcal{S}_\exists) \times \mathcal{A}_\psi$ is satisfiable for some bounds on $\mathcal{S}$, $\mathcal{S}_\exists$, and lookahead $k$.*

## 9.4 Experimental Evaluation

We implemented a prototype synthesis tool, called BoSyHyper[2], for HyperLTL based on the bounded synthesis algorithms described in Section 9.1 and 9.3. Furthermore, we implemented the search for counterexamples proposed in Section 9.2.

We base our implementation on the LTL synthesis tool BoSy described in Section 7.4.1. For efficiency, we split the specifications into two parts, a part containing the linear (LTL) specification, and a part containing the hyperproperty given as HyperLTL formula. Consequently, we build two constraint systems, one using the standard bounded

[2] BoSyHyper is available at https://www.react.uni-saarland.de/tools/bosy/

synthesis approach discuessed in Section 7.3 and one using the approach described in Section 9.1. Before solving, those constraints are combined into a single SMT query. This results in a much more concise constraint system compared to the one where the complete specification is interpreted as a HyperLTL formula. For solving the SMT queries, we use the solver Z3 [MB08]. We continue by describing the benchmarks used in our experiments.

**Symmetric mutual exclusion.** Our first example demonstrates the ability to specify symmetry in HyperLTL for a simple mutual exclusion protocol. Let $r_1$ and $r_2$ be input signals representing mutually exclusive *requests* to a critical section and $g_1/g_2$ the respective grants to enter the section. Every request should be answered eventually $\square(r_i \rightarrow \diamondsuit g_i)$ for $i \in \{1,2\}$, but not at the same time $\square\neg(g_1 \wedge g_2)$. The minimal LTL solution is depicted in Figure 9.6a. It is well known that no mutex protocol can ensure perfect symmetry [MP95], thus when adding the symmetry constraint specified by the HyperLTL formula

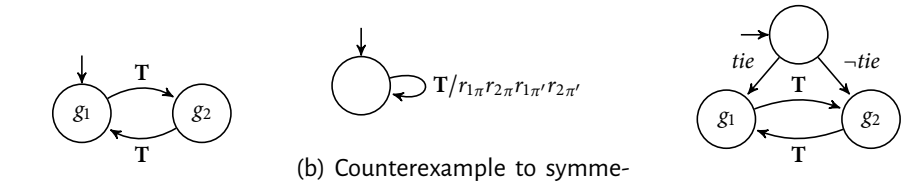
$$\forall\pi\exists\pi'.\square\Big((g_{1_\pi} \leftrightarrow g_{2_{\pi'}}) \wedge (g_{2_\pi} \leftrightarrow g_{1_{\pi'}}) \wedge (r_{1_\pi} \leftrightarrow r_{2_{\pi'}}) \wedge (r_{2_\pi} \leftrightarrow r_{1_{\pi'}})\Big) \tag{9.4}$$

the formula becomes unrealizable. The formula asserts that for every execution, there is an alternative execution were $g_1$ and $g_2$ as well as $r_1$ and $r_2$ are mirrored. An alternative formulation can be derived by *universal strengthening*, that is reducing the $\forall\exists$ formula to a $\forall^2$ formula by encoding the knowledge of the existential trace selection into the formula. For the universal strengthening $\forall\pi\forall\pi'.(r_{1_\pi} \nleftrightarrow r_{2_{\pi'}})\,\mathcal{R}(g_{1_\pi} \leftrightarrow g_{2_{\pi'}})$ our tool produces the counterexample shown in Figure 9.6b. By adding another input signal *tie*, that breaks the symmetry in case of simultaneous requests, the specification becomes realizable with the witnessing transition system given in Figure 9.6c. The universal strengthening with tie breaker is $\forall\pi\forall\pi'.((r_{1_\pi} \nleftrightarrow r_{2_{\pi'}}) \vee (tie_\pi \nleftrightarrow \neg tie_{\pi'}))\,\mathcal{R}(g_{1_\pi} \leftrightarrow g_{2_{\pi'}})$. Compare and contrast this formulation to the one with alternation: We neither have to specify the witnessing traces manually nor do we have to modify the symmetry constraint when adding the symmetry breaker *tie*. In formulation with alternation, the HyperLTL specification stays as is, in fact, it does not even mention *tie*. Detailed solving results for the different variants are given in Table 9.1. We further evaluated the same properties on a version that forbids spurious grants. Here, the universal strengthening is stronger than needed, resulting in a larger solution than the one synthesized from Equation 9.4. We verified that the transition system resulting from the strengthening by model checking the synthesized solution with the $\forall\exists$ formula from Equation 9.4.

Beyond symmetry, we can specify properties over the *scheduling*, like the question whether the scheduling introduces artificial waiting times. To show that this is not the case, we have to provide, for every execution, an alternative execution trace where always at most one request is active:

$$\forall\pi\exists\pi'.\square((g_{1_\pi} \leftrightarrow g_{2_{\pi'}}) \wedge (g_{2_\pi} \leftrightarrow g_{1_{\pi'}}) \wedge \neg(r_{1_{\pi'}} \wedge r_{2_{\pi'}}))$$

In other words, we defend our scheduling by providing a wait-free alternative path.




(a) Non-symmetric solution (b) Counterexample to symmetry (c) Symmetry breaking solution

Figure 9.6: Synthesized solution of the mutual exclusion protocols.

Table 9.1: Results of the mutex benchmarks. When no hyperproperty is given, only the LTL part is used. A given implementation is denoted by "(states)".

| Instance | Hyperproperty | $\lvert\mathcal{S}\rvert$ | $\lvert\mathcal{S}_\exists\rvert$ | Time[s] |
|---|---|---|---|---|
| Mutex | — | 2 | – | $< 1$ |
| | symmetry ($\forall^2$-strengthened) | 3 | – | 3.3 |
| | symmetry ($\forall\exists$) | 3 | 1 | 3.4 |
| Mutex w/o spurious grants | — | 3 | – | $< 1$ |
| | symmetry ($\forall^2$-strengthened) | 5 | – | 423 |
| | symmetry ($\forall\exists$) | (5) | 1 | 1.2 |
| | symmetry ($\forall\exists$) | 3 | 1 | 3.9 |
| | wait-free ($\forall\exists$) | 3 | 3 | 46 |
| | symmetry ($\forall\exists$) + wait-free ($\forall\exists$) | 3 | 1+3 | 840 |

**Dining Cryptographers.** Recap the *dining cryptographers* problem: Three cryptographers $C_a$, $C_b$, and $C_c$ sit at a table in a restaurant having dinner and either one of the cryptographers or, alternatively, the NSA must pay for their meal. Is there a protocol where each cryptographer can find out whether it was a cryptographer who paid or the NSA, but cannot find out which cryptographer paid the bill?

In order to do so, every cryptographer $C$ has to be able to plausibly deny that she has paid, that is, for every possible execution of the protocol there has to be an alternative execution with the same observations despite the fact that $C$ has not paid. We formalize the deniability for cryptographer $C_a$ ($C_b$ and $C_c$ are dual) as

$$\forall\pi.\,\exists\pi'.\,\square(\neg paid_{a\pi'} \wedge (out_{a\pi} \leftrightarrow out_{a\pi'}) \wedge (out_{b\pi} \leftrightarrow out_{b\pi'}) \wedge (out_{c\pi} \leftrightarrow out_{c\pi'})).$$

The setting is also *distributed* with four system processes: The three cryptographers ($C_a$, $C_b$, and $C_c$), where each cryptographer shares a secret bit with each other (denoted $s_{ab}$ for the shared secret of $C_a$ and $C_b$). The fourth entity is the process that receives the output from the cryptographers ($out_a$, $out_b$, and $out_c$) and computes the result whether one of them has paid the bill (output $paid_{group}$). The architecture is given in Figure 9.7.

We now formalize this as a HyperLTL synthesis problem. The set of atomic proposi-

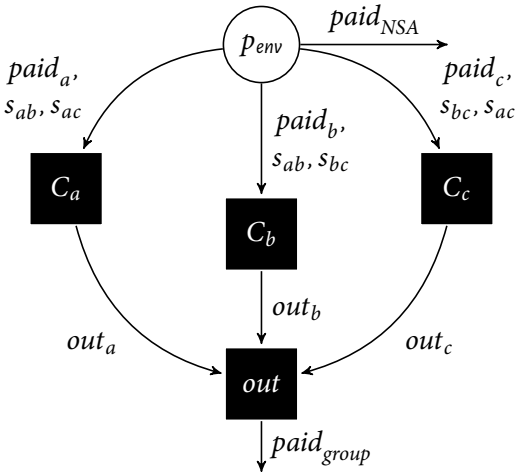



Figure 9.7: The architecture of the dining cryptographers problem with three cryptographers.

tions is partitioned into system and environment outputs given as

$$O = \{out_a, out_b, out_c, paid_{group}\} \text{ and}$$
$$I = \{paid_{NSA}, paid_a, paid_b, paid_c, s_{ab}, s_{ac}, s_{bc}\} \ .$$

The functional (LTL) requirements are simple, assuming that exactly one of $paid_{NSA}$, $paid_a$, $paid_b$, and $paid_c$ is true, the output of $paid_{group}$ is the negation of $paid_{NSA}$. As an LTL formula this can be written as

$$\square exactly\text{-}one(paid_a, paid_b, paid_c, paid_{NSA}) \rightarrow \square(paid_{group} \leftrightarrow \neg paid_{NSA}) \ .$$

The distributed architecture is encoded as a conjunction of HyperLTL formulas ensuring independence of non-observable inputs, i.e.,

$$\forall \pi \forall \pi' . D^{\pi,\pi'}_{\{paid_a, s_{ab}, s_{ac}\} \mapsto \{out_a\}} \wedge D^{\pi,\pi'}_{\{paid_b, s_{ab}, s_{bc}\} \mapsto \{out_b\}} \wedge D^{\pi,\pi'}_{\{paid_c, s_{ac}, s_{bc}\} \mapsto \{out_c\}} \wedge D^{\pi,\pi'}_{\{out_a, out_b, out_c\} \mapsto \{paid_{group}\}}$$

In a previous formulation [Fin+18a] we used a universal strengthening were the indistinguishable executions that witnessed the required deniability were explicitly enumerated in order to encode the synthesis problem in $\forall^*$HyperLTL, e.g., for $C_a$ and $C_b$:

$$\begin{aligned}\forall \pi \forall \pi' . \square ( & (paid_{a\pi} \wedge \neg paid_{a\pi'}) \wedge (\neg paid_{b\pi} \wedge paid_{b\pi'}) \\ & \wedge (s_{ab\pi} \not\leftrightarrow s_{ab\pi'}) \wedge (s_{bc\pi} \leftrightarrow s_{bc\pi'}) \wedge (s_{ac\pi} \leftrightarrow s_{ac\pi'}) \\ & \rightarrow (out_{a\pi} \leftrightarrow out_{a\pi'}) \wedge (out_{b\pi} \leftrightarrow out_{b\pi'})) \ .\end{aligned}$$

Neither LTL synthesis nor its distributed variant can express the combination of those requirements. Our HyperLTL synthesis tool BoSyHyper is able to synthesize a realizing protocol automatically. A closer look in the implementation reveals, that the tool has synthesized the XOR scheme presented in the original solution [Cha85].

Table 9.2: Results of the dining cryptographers benchmarks. A given implementation is denoted by "(states)".

| Instance | Hyperproperty | $\lvert\mathcal{S}\rvert$ | $\lvert\mathcal{S}_\exists\rvert$ | Time[s] |
|---|---|---|---|---|
| Dining Cryptographers | distributed ($\forall^2$) | 1 | – | 44 |
| | distr. ($\forall^2$) + deniability ($\forall^2$-strength.) | 1 | – | 61 |
| | distributed ($\forall^2$) + deniability ($\forall\exists$) | (1) | 1 | 1.2 |
| | distributed ($\forall^2$) + deniability ($\forall\exists$) | 1 | 1 | 12.6 |

**Distributed and fault-tolerant systems.** In Section 8.3 we presented a reduction of arbitrary distributed architectures to HyperLTL. As an example for our evaluation, consider a setting with two processes, one for *encoding* input signals and one for *decoding*. Both processes can be synthesized simultaneously using a single HyperLTL specification. The (linear) correctness condition states that the decoded signal is always equal to the inputs given to the encoder. Furthermore, the encoder and decoder should solely depend on the inputs and the encoded signal, respectively. Additionally, we can specify desired properties about the encoding like fault-tolerance [FT15] or Hamming distance of code words [FRS15]. An example solution for 2 input bits and 3 encoded bits is shown in Figure 9.8. For the encoding, we required that for every change in the input, two encoding bits change. The synthesized solution uses a parity bit as the third encoded bit and the encoding and decoding parts are strictly independent. Detailed solving results are reported in Table 9.3 where $i$-$j$-$x$ means $i$ input bits, $j$ encoded bits, and $x$ represents the property. The property is either tolerance against a single Byzantine signal failure or a guaranteed Hamming distance of code words.

**CAP Theorem.** The CAP Theorem due to Brewer [Bre00] states that it is impossible to design a distributed system that provides Consistency, Availability, and Partition tolerance (CAP) simultaneously. This example has been considered before [FT15] to evaluate a technique that could automatically detect unrealizability. However, when we drop either Consistency, Availability, or Partition tolerance, the corresponding instances (AP, CP, and CA) become realizable, which the previous work was not able to prove. We show that our implementation can show both, unrealizability of CAP and realizability of AP, CP, and CA.

We recap the formal encoding [FT15] using HyperLTL. There are two differences, we allow for cyclic architectures and we show the Mealy version (the Moore version needs 3 consecutive $\bigcirc$ before the output, due to the delayed flow of information). We assume there is a fixed number $n$ of nodes, that every node implements the same service, and that the architecture is fully connected. We use the variables $req_i$ and $out_i$ to denote input and output of node $i$, respectively. The consistency and availability requirements are encoded as the LTL formulas $\square(\bigwedge_{1\leq i<n} out_i \leftrightarrow out_{i+1})$ and $\square((\bigvee_{1\leq i\leq n} req_i) \leftrightarrow (\bigvee_{1\leq i\leq n} out_i))$. The partition tolerance is modeled in a way that there is always at most

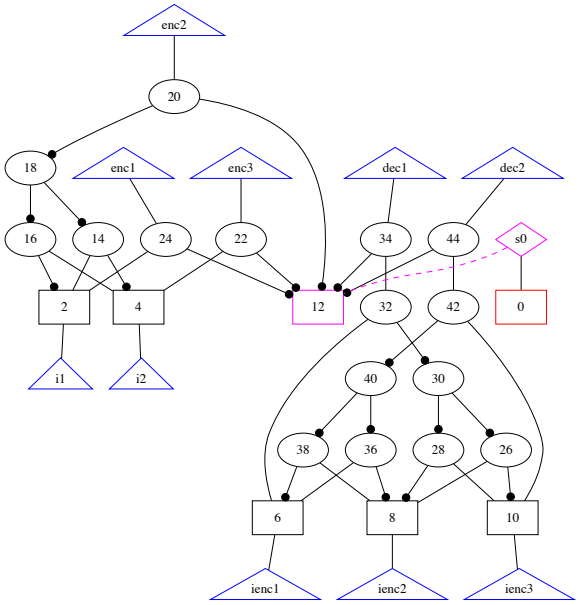



Figure 9.8: Representation of the solution for an encoder with 2 input bits and 3 encoded bits as And-Inverter-Graph. The solution is produced by BoSyHyper where the specification is given as a single HyperLTL formula specifying both, the encoder and the decoder, as well as the distributivity constraints. Note that although BoSyHyper produces a global implementation, the implementation is actually distributed as decoder and encoder do not share gates.

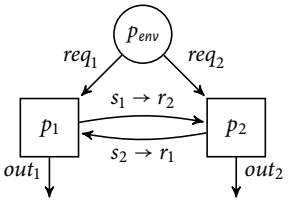



Figure 9.9: Visualization of the architecture for the CAP-2 instance.

one node partitioned from the rest of the system. For two nodes, we get the LTL formula

$$(\square(r_2 \leftrightarrow s_1) \vee \square(r_1 \leftrightarrow s_2)) \rightarrow \square((out_1 \leftrightarrow out_2) \wedge ((req_1 \vee req_2) \leftrightarrow (out_1 \vee out_2)))$$

and additionally the HyperLTL constraint $\forall \pi \forall \pi'. D^{\pi,\pi'}_{\{req_1,r_2\} \mapsto \{s_1,out_1\}} \wedge D^{\pi,\pi'}_{\{req_2,r_1\} \mapsto \{s_2,out_2\}}$. The corresponding architecture is shown in Figure 9.9. The results are given in Table 9.3.

**Long-term information flow.** Previous work on model-checking hyperproperties [FRS15] found that an implementation for the commonly used *I2C* bus protocol

could remember input values ad infinitum. For example, it could not be verified that information given to the implementation eventually leaves it, i.e., is forgotten. This is especially unfortunate in high security contexts. We consider a simple bus protocol which is inspired by the widely used *I2C* protocol. Our example protocol has the inputs *send* for initiating a transmission, *in* for the value that should be transferred, and an *ack*nowledgment bit indicating successful transmission. The bus master waits in an *idle* state until a *send* is received. Afterward, it transmits a header sequence, followed by the value of *in*, waits for an acknowledgement and then indicates *success* or *failure* to the sender before returning to the idle state. We specify the property that the *in*put has no influence on the *data* that is send, which is obviously violated (instance NI1). As a second property, we check that this information leak cannot happen arbitrary long (NI2) for which there is a realizing implementation.

We give the LTL specification used in the benchmarks in the following

$$
\begin{aligned}
&\mathit{idle}\\
&\wedge \square((\mathit{idle} \wedge \neg \mathit{send}) \rightarrow \bigcirc \mathit{idle})\\
&\wedge \square((\mathit{idle} \wedge \mathit{send}) \rightarrow (\bigcirc \mathit{start} \wedge \bigcirc_2 \mathit{transmit} \wedge (\bigcirc_2 \mathit{data} \leftrightarrow \mathit{in})))\\
&\wedge \square(\mathit{transmit} \rightarrow \bigcirc \mathit{waitForAck})\\
&\wedge \square((\mathit{waitForAck} \wedge \mathit{ack}) \leftrightarrow \bigcirc \mathit{success})\\
&\wedge \square((\mathit{waitForAck} \wedge \neg \mathit{ack}) \leftrightarrow \bigcirc \mathit{failure})\\
&\wedge \square(\mathit{success} \rightarrow \bigcirc \mathit{idle})\\
&\wedge \square(\mathit{failure} \rightarrow \bigcirc \mathit{idle})\\
&\wedge \square(\mathit{mutex}(\mathit{idle}, \mathit{start}, \mathit{transmit}, \mathit{waitForAck}, \mathit{success}, \mathit{failure}))
\end{aligned}
$$

Additionally, we used the following HyperLTL specifications

$$\forall \pi \forall \pi' . \square((\mathit{send}_\pi \leftrightarrow \mathit{send}_{\pi'}) \wedge (\mathit{ack}_\pi \leftrightarrow \mathit{ack}_{\pi'})) \rightarrow \square(\mathit{data}_\pi \leftrightarrow \mathit{data}_{\pi'}) \tag{NI1}$$

$$\forall \pi \forall \pi' . \square((\mathit{send}_\pi \leftrightarrow \mathit{send}_{\pi'}) \wedge (\mathit{ack}_\pi \leftrightarrow \mathit{ack}_{\pi'})) \rightarrow (\lozenge\square(\mathit{in}_\pi \leftrightarrow \mathit{in}_{\pi'}) \rightarrow \lozenge\square(\mathit{data}_\pi \leftrightarrow \mathit{data}_{\pi'})) \tag{NI2}$$

**Results.** Table 9.3 reports on the results of the $\forall^*$HyperLTL benchmarks. We distinguish between state-labeled (*Moore*) and transition-labeled (*Mealy*) transition systems. Note that the counterexample strategies use the opposite transition system, i.e., a Mealy system strategy corresponds to a state-labeled (Moore) environment strategy. Typically, Mealy strategies are more compact, i.e., need smaller transition systems and this is confirmed by our experiments. BoSyHyper is able to solve most of the examples, providing realizing implementations or counterexamples. Regrading the unrealizable benchmarks we observe that usually two simultaneously generated paths ($k = 2$) are enough with the exception of the encoder example. Overall the results are encouraging showing that we can solve a variety of instances with non-trivial information flow.

Table 9.3: Results of BoSyHyper on the $\forall^*$HyperLTL benchmarks sets described in Section 6.5. They ran on a machine with a dual-core Core i7, 3.3 GHz, and 16 GB memory.

| Benchmark | Instance | Result | States | | Time[sec.] | |
|---|---|---|---|---|---|---|
| | | | Moore | Mealy | Moore | Mealy |
| Encoder/Decoder | 1-2-hamming-2 | realizable | 4 | 1 | 1.6 | 1.3 |
| | 1-2-fault-tolerant | unrealizable ($k = 2$) | 1 | - | 54.9 | - |
| | 1-3-fault-tolerant | realizable | 4 | 1 | 151.7 | 1.7 |
| | 2-2-hamming-2 | unrealizable ($k = 3$) | - | 1 | - | 10.6 |
| | 2-3-hamming-2 | realizable | 16 | 1 | $> 1\,h$ | 1.5 |
| | 2-3-hamming-3 | unrealizable ($k = 3$) | - | 1 | - | 126.7 |
| CAP Theorem | cap-2-non-dist. | realizable | 8 | 1 | 7.0 | 1.3 |
| | cap-2 | unrealizable ($k = 2$) | 1 | - | 1 823.9 | - |
| | ca-2 | realizable | - | 1 | - | 4.4 |
| | ca-3 | realizable | - | 1 | - | 15.0 |
| | cp-2 | realizable | 1 | 1 | 1.8 | 1.6 |
| | cp-3 | realizable | 1 | 1 | 3.2 | 10.6 |
| | ap-2 | realizable | - | 1 | - | 2.0 |
| | ap-3 | realizable | - | 1 | - | 43.4 |
| Bus Protocol | NI1 | unrealizable ($k = 2$) | 1 | 1 | 75.2 | 69.6 |
| | NI2 | realizable | 8 | 8 | 24.1 | 33.9 |

## 9.5 Summary

In this chapter, we presented semi-decision procedures for the realizability problem of HyperLTL. We gave algorithms that can efficiently show the realizability and unrealizability of universal HyperLTL. By reducing existential trace quantification to strategic choice and bounded lookahead, we reduced the realizability problem with quantifier-alternations to the alternation-free fragment. Using case studies whose properties go well beyond state-of-the-art synthesis tools, we showed that we can synthesize implementations with non-trivial information flow automatically.

# Chapter 10

# Conclusions & Outlook

We have developed symbolic methods for the synthesis problem of reactive systems. Based on a reduction to the satisfiability problem of QBF and DQBF, we have constructed a more succinct encoding of the bounded synthesis approach. Using the solving algorithms introduced in the first part of this thesis, we were able to show that our method significantly improves over the previous bounded synthesis encoding based on a reduction to a decidable first-order theory. Further, we showed that, in addition to producing state-space optimal solutions, the measurement in terms of the size of the transition function is often significantly smaller than state-of-the-art synthesis tools.

Further research questions include using a more modular approach to handling LTL formulas: Frequently, it is not even possible to construct an encoding as the automaton conversion from a single, monolithic LTL formula time out. The encodings can be improved, too: For example, symmetry breaking constraints have been successful in many approaches that use SAT solving and may significantly improve solver performance. Further, a more specialized encoding to conjunctive normal form may improve performance and may obviate the need for preprocessing.

This thesis has established DQBF solving as a viable alternative to QBF solving for the bounded synthesis from LTL. In theory, we can also use DQBF as a target logic for symbolic encodings of the HyperLTL synthesis problem. This, however, needs to go hand-in-hand with a suitable solving algorithm and implementation; the current prototypical clausal abstraction solver does not always perform better than the SMT encoding.

This thesis also contributes to the theory and practice of solving quantified Boolean formulas. We have shown that the underlying concept of the clausal abstraction algorithm can be applied to a variety of different quantified formulas: prenex and non-prenex, clausal and non-clausal, as well as linear and branching quantification. On the theory side, we have shown how to combine $Q$-resolution and $\forall$Exp+Res to a unified proof system that combines the strength of both of them. This result leads the way to an outstanding performance of the implementations in the annual QBF competition.

The implementation of clausal abstraction would benefit from heuristics that guide which of the extensions to use for a given formula. For example, it is known that the number of alternations of the quantifier prefix influences the decision between search-based

and expansion-based solving, both in theory [Bey+19] and practice [LE18a]. Further, supporting the QRAT proof format should be possible with the benefit of being able to certify the complete solving pipeline, from preprocessing to solving.

# Bibliography


[AB02] Abdelwaheb Ayari and David A. Basin. „QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers“. In: *Proceedings of FMCAD*. Vol. 2517. LNCS. Springer, 2002, pp. 187–201. DOI: 10.1007/3-540-36126-X_12.

[AHK97] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. „Alternating-time Temporal Logic“. In: *Proceedings of FOCS*. IEEE Computer Society, 1997, pp. 100–109. DOI: 10.1109/SFCS.1997.646098.

[Bab+12] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. „LTL to Büchi Automata Translation: Fast and More Deterministic“. In: *Proceedings of TACAS*. Vol. 7214. LNCS. Springer, 2012, pp. 95–109. DOI: 10.1007/978-3-642-28756-5_8.

[Bal+15] Valeriy Balabanov, Jie-Hong Roland Jiang, Mikolas Janota, and Magdalena Widl. „Efficient Extraction of QBF (Counter)models from Long-Distance Resolution Proofs“. In: *Proceedings of AAAI*. AAAI Press, 2015, pp. 3694–3701. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9419.

[Bal+16a] Valeriy Balabanov, Jie-Hong Roland Jiang, Christoph Scholl, Alan Mishchenko, and Robert K. Brayton. „2QBF: Challenges and Solutions“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 453–469. DOI: 10.1007/978-3-319-40970-2_28.

[Bal+16b] 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.

[BB09] Hans Kleine Büning and Uwe Bubeck. „Theory of Quantified Boolean Formulas“. In: *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 735–760. DOI: 10.3233/978-1-58603-929-5-735.

[BCJ12] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong Roland Jiang. „Henkin Quantifiers and Boolean Formulae“. In: *Proceedings of SAT*. Vol. 7317. LNCS. Springer, 2012, pp. 129–142. DOI: 10.1007/978-3-642-31612-8_11.

[BCJ14a] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. „Henkin quantifiers and Boolean formulae: A certification perspective of DQBF“. In: *Theor. Comput. Sci.* 523 (2014), pp. 86–100. DOI: 10.1016/j.tcs.2013.12.020.

[BCJ14b] Olaf Beyersdorff, Leroy Chew, and Mikolas Janota. „On Unification of QBF Resolution-Based Calculi“. In: *Proceedings of MFCS*. Vol. 8635. LNCS. Springer, 2014, pp. 81–93. DOI: 10.1007/978-3-662-44465-8_8.

[BCJ15] Olaf Beyersdorff, Leroy Chew, and Mikolás Janota. „Proof Complexity of Resolution-based QBF Calculi“. In: *Proceedings of STACS*. Vol. 30. LIPIcs. Schloss Dagstuhl – LZI, 2015, pp. 76–89. DOI: 10.4230/LIPIcs.STACS.2015.76.

[BDR11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. „Secure information flow by self-composition“. In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252. DOI: 10.1017/S0960129511000193.

[Ben04] Marco Benedetti. „Evaluating QBFs via Symbolic Skolemization“. In: *Proceedings of LPAR*. Vol. 3452. LNCS. Springer, 2004, pp. 285–300. DOI: 10.1007/978-3-540-32275-7_20.

[Ben05a] Marco Benedetti. „Extracting Certificates from Quantified Boolean Formulas“. In: *Proceedings of IJCAI*. Professional Book Center, 2005, pp. 47–53. URL: http://ijcai.org/Proceedings/05/Papers/0985.pdf.

[Ben05b] Marco Benedetti. „sKizzo: A Suite to Evaluate and Certify QBFs“. In: *Proceedings of CADE-20*. Vol. 3632. LNCS. Springer, 2005, pp. 369–376. DOI: 10.1007/11532231_27.

[Bey+18] Olaf Beyersdorff, Joshua Blinkhorn, Leroy Chew, Renate Schmidt, and Martin Suda. „Reinterpreting Dependency Schemes: Soundness Meets Incompleteness in DQBF“. In: *J. Autom. Reasoning* (2018), pp. 1–27. DOI: 10.1007/s10817-018-9482-4.

[Bey+19] Olaf Beyersdorff, Leroy Chew, Judith Clymo, and Meena Mahajan. „Short Proofs in QBF Expansion“. In: *Proceedings of SAT*. Vol. 11628. LNCS. Springer, 2019, pp. 19–35. DOI: 10.1007/978-3-030-24258-9_2.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, 2017. URL: http://www.SMT-LIB.org.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. „Symbolic Model Checking without BDDs“. In: *Proceedings of TACAS*. Vol. 1579. LNCS. Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[Bie04] Armin Biere. „Resolve and Expand“. In: *Proceedings of SAT*. Vol. 3542. LNCS. Springer, 2004, pp. 59–70. DOI: 10.1007/11527695_5.

[Bie08] Armin Biere. „PicoSAT Essentials“. In: *JSAT* 4.2-4 (2008), pp. 75–97. URL: https://satassociation.org/jsat/index.php/jsat/article/view/45.

[Bir67] Garrett Birkhoff. *Lattice theory*. eng. American Math. Soc., 1967.

[BJ12] Valeriy Balabanov and Jie-Hong R. Jiang. „Unified QBF certification and its applications“. In: *Formal Methods in System Design* 41.1 (2012), pp. 45–65. DOI: 10.1007/s10703-012-0152-6.

[BJ15] Nikolaj Bjørner and Mikolás Janota. „Playing with Quantified Satisfaction“. In: *Proceedings of LPAR*. Vol. 35. EPiC Series in Computing. EasyChair, 2015, pp. 15–27.

[BK06] Uwe Bubeck and Hans Kleine Büning. „Dependency Quantified Horn Formulas: Models and Complexity“. In: *Proceedings of SAT*. Vol. 4121. LNCS. Springer, 2006, pp. 198–211. DOI: 10.1007/11814948_21.

[BKS14] Roderick Bloem, Robert Könighofer, and Martina Seidl. „SAT-Based Synthesis Methods for Safety Specs“. In: *Proceedings of VMCAI*. Vol. 8318. LNCS. Springer, 2014, pp. 1–20. DOI: 10.1007/978-3-642-54013-4_1.

[BL16] Tomás Balyo and Florian Lonsing. „HordeQBF: A Modular and Massively Parallel QBF Solver“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 531–538. DOI: 10.1007/978-3-319-40970-2_33.

[BL69] J. Richard Büchi and Lawrence H. Landweber. „Solving Sequential Conditions by Finite-State Strategies“. English. In: *Transactions of the American Mathematical Society* 138 (1969). ISSN: 00029947. URL: http://www.jstor.org/stable/1994916.

[BLJ16] Valeriy Balabanov, Shuo-Ren Lin, and Jie-Hong R. Jiang. „Flexibility and Optimization of QBF Skolem-Herbrand Certificates“. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 35.9 (2016), pp. 1557–1568. DOI: 10.1109/TCAD.2015.2512906.

[Blo+14] Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. „SAT-based methods for circuit synthesis“. In: *Proceedings of FMCAD*. IEEE, 2014, pp. 31–34. DOI: 10.1109/FMCAD.2014.6987592.

[Blo+18] Roderick Bloem, Nicolas Braud-Santoni, Vedad Hadzic, Uwe Egly, Florian Lonsing, and Martina Seidl. „Expansion-Based QBF Solving Without Recursion“. In: *Proceedings of FMCAD*. IEEE, 2018, pp. 1–10. DOI: 10.23919/FMCAD.2018.8603004.

[BLS11] Armin Biere, Florian Lonsing, and Martina Seidl. „Blocked Clause Elimination for QBF“. In: *Proceedings of CADE*. Vol. 6803. LNCS. Springer, 2011, pp. 101–115. DOI: 10.1007/978-3-642-22438-6_10.

[BM08] Marco Benedetti and Hratch Mangassarian. „QBF-Based Formal Verification: Experience and Perspectives“. In: *JSAT* 5.1-4 (2008), pp. 133–191. URL: https://satassociation.org/jsat/index.php/jsat/article/view/62.

[BM10] Robert K. Brayton and Alan Mishchenko. „ABC: An Academic Industrial-Strength Verification Tool“. In: *Proceedings of CAV*. Vol. 6174. LNCS. Springer, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[Boh+12] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. „Acacia+, a Tool for LTL Synthesis“. In: *Proceedings of CAV*. Vol. 7358. LNCS. Springer, 2012, pp. 652–657. DOI: 10.1007/978-3-642-31424-7_45.

[Bra11] Aaron R. Bradley. „SAT-Based Model Checking without Unrolling“. In: *Proceedings of VMCAI*. Vol. 6538. LNCS. Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[Bre00] Eric A. Brewer. „Towards robust distributed systems (abstract)“. In: *Proceedings of PODC*. ACM, 2000, p. 7. DOI: 10.1145/343477.343502.

[Bri+11] Thomas Brihaye, Véronique Bruyère, Laurent Doyen, Marc Ducobu, and Jean-François Raskin. „Antichain-Based QBF Solving“. In: *Proceedings of ATVA*. Vol. 6996. LNCS. Springer, 2011, pp. 183–197. DOI: 10.1007/978-3-642-24372-1_14.

[Bry86] Randal E. Bryant. „Graph-Based Algorithms for Boolean Function Manipulation“. In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[BSG17] Vincent Bindschaedler, Reza Shokri, and Carl A. Gunter. „Plausible Deniability for Privacy-Preserving Data Synthesis“. In: *PVLDB* 10.5 (2017), pp. 481–492. DOI: 10.14778/3055540.3055542. URL: http://www.vldb.org/pvldb/vol10/p481-bindschaedler.pdf.

[Bur+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. „Symbolic Model Checking: $10^20$ States and Beyond“. In: *Proceedings of LICS*. IEEE Computer Society, 1990, pp. 428–439. DOI: 10.1109/LICS.1990.113767.

[CCS17] Anrin Chakraborti, Chen Chen, and Radu Sion. „DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries“. In: *PoPETs* 2017.3 (2017), p. 179. DOI: 10.1515/popets-2017-0035.

[Cha85] David Chaum. „Security Without Identification: Transaction Systems to Make Big Brother Obsolete“. In: *Commun. ACM* 28.10 (1985), pp. 1030–1044. DOI: 10.1145/4372.4373.

[CHP10] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. „Strategy logic“. In: *Inf. Comput.* 208.6 (2010), pp. 677–693. DOI: 10.1016/j.ic.2009.07.004.

[CHR16] Chih-Hong Cheng, Yassine Hamza, and Harald Ruess. „Structural Synthesis for GXW Specifications“. In: *Proceedings of CAV*. Vol. 9779. LNCS. Springer, 2016, pp. 95–117. DOI: 10.1007/978-3-319-41528-4_6.

[Chu57] Alonzo Church. „Application of Recursive Arithmetic to the Problem of Circuit Synthesis“. In: *Summaries of the Summer Institute of Symbolic Logic* 1 (1957), pp. 3–50.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. „The Complexity of Satisfiability of Small Depth Circuits“. In: *Proceedings of IWPEC*. Vol. 5917. LNCS. Springer, 2009, pp. 75–85. DOI: 10.1007/978-3-642-11269-0_6.

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. „Counterexample-Guided Abstraction Refinement“. In: *Proceedings of CAV*. Vol. 1855. LNCS. Springer, 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[Cla+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. „Temporal Logics for Hyperproperties“. In: *Proceedings of POST*. Vol. 8414. LNCS. Springer, 2014, pp. 265–284. DOI: 10.1007/978-3-642-54792-8_15.

[CLR17] Chih-Hong Cheng, Edward A. Lee, and Harald Ruess. „autoCode4: Structural Controller Synthesis“. In: *Proceedings of TACAS*. Vol. 10205. LNCS. 2017, pp. 398–404. DOI: 10.1007/978-3-662-54577-5_23.

[Coe+19] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. „Verifying Hyperliveness“. In: *Proceedings of CAV*. Vol. 11561. LNCS. Springer, 2019, pp. 121–139. DOI: 10.1007/978-3-030-25540-4_7.

[Coo+19] Simon Cooksey, Sarah Harris, Mark Batty, Radu Grigore, and Mikolás Janota. „PrideMM: A Solver for Relaxed Memory Models“. In: *CoRR* abs/1901.00428 (2019). arXiv: 1901.00428. URL: http://arxiv.org/abs/1901.00428.

[CS10] Michael R. Clarkson and Fred B. Schneider. „Hyperproperties“. In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210. DOI: 10.3233/JCS-2009-0393.

[CW16] Günther Charwat and Stefan Woltran. „Dynamic Programming-based QBF Solving“. In: *Proceedings of QBF@SAT*. Vol. 1719. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 27–40.

[DAr+17] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. „Is Your Software on Dope? - Formal Analysis of Surreptitiously "enhanced" Programs“. In: *Proceedings of ESOP*. Vol. 10201. LNCS. Springer, 2017, pp. 83–110. DOI: 10.1007/978-3-662-54434-1_4.

[DF09] Rayna Dimitrova and Bernd Finkbeiner. „Synthesis of Fault-Tolerant Distributed Systems“. In: *Proceedings of ATVA*. Vol. 5799. LNCS. Springer, 2009, pp. 321–336. DOI: 10.1007/978-3-642-04761-9_24.

[DHK05] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. „Bounded Model Checking with QBF“. In: *Proceedings of SAT*. Vol. 3569. LNCS. Springer, 2005, pp. 408–414. DOI: 10.1007/11499107_32.

[Dur+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. „Spot 2.0 - A Framework for LTL and $\omega$-Automata Manipulation“. In: *Proceedings of ATVA*. Vol. 9938. LNCS. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8.

[EF17] Rüdiger Ehlers and Bernd Finkbeiner. „Symmetric Synthesis“. In: *Proceedings of IARCS*. Vol. 93. LIPIcs. Schloss Dagstuhl – LZI, 2017, 26:1–26:13. DOI: 10.4230/LIPIcs.FSTTCS.2017.26.

[Egl+17] Uwe Egly, Martin Kronegger, Florian Lonsing, and Andreas Pfandler. „Conformant planning as a case study of incremental QBF solving“. In: *Ann. Math. Artif. Intell.* 80.1 (2017), pp. 21–45. DOI: 10.1007/s10472-016-9501-2.

[Egl16] Uwe Egly. „On Stronger Calculi for QBFs“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 419–434. DOI: 10.1007/978-3-319-40970-2_26.

[Ehl11] Rüdiger Ehlers. „Unbeast: Symbolic Bounded Synthesis“. In: *Proceedings of TACAS*. Vol. 6605. LNCS. Springer, 2011, pp. 272–275. DOI: 10.1007/978-3-642-19835-9_25.

[Ehl12] Rüdiger Ehlers. „Symbolic bounded synthesis“. In: *Formal Methods in System Design* 40.2 (2012), pp. 232–262. DOI: 10.1007/s10703-011-0137-x.

[ELW13] Uwe Egly, Florian Lonsing, and Magdalena Widl. „Long-Distance Resolution: Proof Generation and Strategy Extraction in Search-Based QBF Solving“. In: *Proceedings of LPAR*. Vol. 8312. LNCS. Springer, 2013, pp. 291–308. DOI: 10.1007/978-3-642-45221-5_21.

[EM12] Rüdiger Ehlers and Daniela Moldovan. „Sparse Positional Strategies for Safety Games“. In: *Proceedings of SYNT*. Vol. 84. EPTCS. 2012, pp. 1–16. DOI: 10.4204/EPTCS.84.1.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. „Efficient implementation of property directed reachability“. In: *Proceedings of FMCAD*. FMCAD Inc., 2011, pp. 125–134. URL: http://dl.acm.org/citation.cfm?id=2157675.

[ESW09] Uwe Egly, Martina Seidl, and Stefan Woltran. „A solver for QBFs in negation normal form“. In: *Constraints* 14.1 (2009), pp. 38–79. DOI: 10.1007/s10601-008-9055-y.

[Fag+95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[Fay+17] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. „Encodings of Bounded Synthesis“. In: *Proceedings of TACAS*. Vol. 10205. LNCS. 2017, pp. 354–370. DOI: 10.1007/978-3-662-54577-5_20.

[Faz+17] Katalin Fazekas, Marijn J. H. Heule, Martina Seidl, and Armin Biere. „Skolem Function Continuation for Quantified Boolean Formulas“. In: *Proceedings of TAP*. Vol. 10375. LNCS. Springer, 2017, pp. 129–138. DOI: 10.1007/978-3-319-61467-0_8.

[FFT17] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. „BoSy: An Experimentation Framework for Bounded Synthesis“. In: *Proceedings of CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 325–332. DOI: 10.1007/978-3-319-63390-9_17.

[FH16] Bernd Finkbeiner and Christopher Hahn. „Deciding Hyperproperties“. In: *Proceedings of CONCUR*. Vol. 59. LIPIcs. Schloss Dagstuhl – LZI, 2016, 13:1–13:14. DOI: 10.4230/LIPIcs.CONCUR.2016.13.

[FHH18] Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. „MGHyper: Checking Satisfiability of HyperLTL Formulas Beyond the $\exists^*\forall^*$ Fragment“. In: *Proceedings of ATVA*. Vol. 11138. LNCS. Springer, 2018, pp. 521–527. DOI: 10.1007/978-3-030-01090-4_31.

[FHS17] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. „EAHyper: Satisfiability, Implication, and Equivalence Checking of Hyperproperties“. In: *Proceedings of CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 564–570. DOI: 10.1007/978-3-319-63390-9_29.

[FHT18] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. „Model Checking Quantitative Hyperproperties“. In: *Proceedings of CAV*. Vol. 10981. LNCS. Springer, 2018, pp. 144–163. DOI: 10.1007/978-3-319-96145-3_8.

[Fin+17a] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. „Symbolic vs. Bounded Synthesis for Petri Games“. In: *Proceedings of SYNT@CAV*. Vol. 260. EPTCS. 2017, pp. 23–43. DOI: 10.4204/EPTCS.260.5.

[Fin+17b] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. „Monitoring Hyperproperties“. In: *Proceedings of RV*. Vol. 10548. LNCS. Springer, 2017, pp. 190–207. DOI: 10.1007/978-3-319-67531-2_12.

[Fin+18a] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. „Synthesizing Reactive Systems from Hyperproperties“. In: *Proceedings of CAV*. Vol. 10981. LNCS. Springer, 2018, pp. 289–306. DOI: 10.1007/978-3-319-96145-3_16.

[Fin+18b] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. „RVHyper: A Runtime Verification Tool for Temporal Hyperproperties“. In: *Proceedings of TACAS*. Vol. 10806. LNCS. Springer, 2018, pp. 194–200. DOI: 10.1007/978-3-319-89963-3_11.

[Fin+19a] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. „Synthesis from Hyperproperties“. Accepted for publication in Acta Informatica. 2019.

[Fin+19b] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. „Monitoring Hyperproperties“. In: *Formal Methods in System Design* (2019), pp. 1–28. DOI: 10.1007/s10703-019-00334-z.

[Fin15] Bernd Finkbeiner. „Bounded Synthesis for Petri Games“. In: *Symposium on Correct System Design*. Vol. 9360. LNCS. Springer, 2015, pp. 223–237. DOI: 10.1007/978-3-319-23506-6_15.

[FJ12] Bernd Finkbeiner and Swen Jacobs. „Lazy Synthesis“. In: *Proceedings of VMCAI*. Vol. 7148. LNCS. Springer, 2012, pp. 219–234. DOI: 10.1007/978-3-642-27940-9_15.

[FJR11] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. „Antichains and compositional algorithms for LTL synthesis“. In: *Formal Methods in System Design* 39.3 (2011), pp. 261–296. DOI: 10.1007/s10703-011-0115-3.

[FK16] Bernd Finkbeiner and Felix Klein. „Bounded Cycle Synthesis“. In: *Proceedings of CAV*. Vol. 9779. LNCS. Springer, 2016, pp. 118–135. DOI: 10.1007/978-3-319-41528-4_7.

[FKB12] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. „A DPLL Algorithm for Solving DQBF“. In: *Proceedings of POS@SAT*. 2012.

[FMS00] Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. „A Distributed Algorithm to Evaluate Quantified Boolean Formulae“. In: *Proceedings of AAAI*. AAAI Press / The MIT Press, 2000, pp. 285–290. URL: http://www.aaai.org/Library/AAAI/2000/aaai00-044.php.

[Frö+14] Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. „iDQ: Instantiation-Based DQBF Solving“. In: *Proceedings of POS@SAT*. Vol. 27. EPiC Series in Computing. EasyChair, 2014, pp. 103–116. URL: http://www.easychair.org/publications/paper/187054.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. „Algorithms for Model Checking HyperLTL and HyperCTL*“. In: *Proceedings of CAV*. Vol. 9206. LNCS. Springer, 2015, pp. 30–48. DOI: 10.1007/978-3-319-21690-4_3.

[FS05] Bernd Finkbeiner and Sven Schewe. „Uniform Distributed Synthesis“. In: *Proceedings of LICS*. IEEE Computer Society, 2005, pp. 321–330. DOI: 10.1109/LICS.2005.53.

[FS07] Bernd Finkbeiner and Sven Schewe. „SMT-Based Synthesis of Distributed Systems“. In: *Proceedings of AFM*. 2007.

[FS10] Bernd Finkbeiner and Sven Schewe. „Coordination Logic“. In: *Proceedings of CSL*. Vol. 6247. LNCS. Springer, 2010, pp. 305–319. DOI: 10.1007/978-3-642-15205-4_25.

[FS13] Bernd Finkbeiner and Sven Schewe. „Bounded synthesis“. In: *STTT* 15.5-6 (2013), pp. 519–539. DOI: 10.1007/s10009-012-0228-z.

[FT14a] Bernd Finkbeiner and Leander Tentrup. „Detecting Unrealizable Specifications of Distributed Systems“. In: *Proceedings of TACAS*. Vol. 8413. LNCS. Springer, 2014, pp. 78–92. DOI: 10.1007/978-3-642-54862-8_6.

[FT14b] Bernd Finkbeiner and Leander Tentrup. „Fast DQBF Refutation“. In: *Proceedings of SAT*. Vol. 8561. LNCS. Springer, 2014, pp. 243–251. DOI: 10.1007/978-3-319-09284-3_19.

[FT15] Bernd Finkbeiner and Leander Tentrup. „Detecting Unrealizability of Distributed Fault-tolerant Systems“. In: *Logical Methods in Computer Science* 11.3 (2015). DOI: 10.2168/LMCS-11(3:12)2015.

[Gas+14] Adrià Gascón, Pramod Subramanyan, Bruno Dutertre, Ashish Tiwari, Dejan Jovanovic, and Sharad Malik. „Template-based circuit understanding“. In: *Proceedings of FMCAD*. IEEE, 2014, pp. 83–90. DOI: 10.1109/FMCAD.2014.6987599.

[GB10] Alexandra Goultiaeva and Fahiem Bacchus. „Exploiting QBF Duality on a Circuit Representation“. In: *Proceedings of AAAI*. AAAI Press, 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1791.

[Gel12] Allen Van Gelder. „Contributions to the Theory of Practical Quantified Boolean Formula Solving“. In: *Proceedings of CP*. Vol. 7514. LNCS. Springer, 2012, pp. 647–663. DOI: 10.1007/978-3-642-33558-7_47.

[GGB11] Alexandra Goultiaeva, Allen Van Gelder, and Fahiem Bacchus. „A Uniform Approach for Generating Proofs and Strategies for Both True and False QBF Formulas“. In: *Proceedings of IJCAI*. IJCAI/AAAI, 2011, pp. 546–553. DOI: 10.5591/978-1-57735-516-8/IJCAI11-099.

[GIB09] Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. „Beyond CNF: A Circuit-Based QBF Solver“. In: *Proceedings of SAT*. Vol. 5584. LNCS. Springer, 2009, pp. 412–426. DOI: 10.1007/978-3-642-02777-2_38.

[Git+13] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. „Equivalence checking of partial designs using dependency quantified Boolean formulae“. In: *Proceedings of ICCD*. 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 DATE*. ACM, 2015, pp. 1617–1622. URL: http://dl.acm.org/citation.cfm?id=2757188.

[GM82] Joseph A. Goguen and José Meseguer. „Security Policies and Security Models“. In: *Proceedings of S&P*. IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[GMN09] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. „Reasoning with Quantified Boolean Formulas“. In: *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 761–780. DOI: 10.3233/978-1-58603-929-5-761.

[GNT04] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. „QuBE++: An Efficient QBF Solver“. In: *Proceedings of FMCAD*. Vol. 3312. LNCS. Springer, 2004, pp. 201–213. DOI: 10.1007/978-3-540-30494-4_15.

[GNT06] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. „Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas“. In: *J. Artif. Intell. Res.* 26 (2006), pp. 371–416. DOI: 10.1613/jair.1959.

[Goe92] Andreas Goerdt. „Davis-Putnam Resolution versus Unrestricted Resolution“. In: *Ann. Math. Artif. Intell.* 6.1-3 (1992), pp. 169–184. DOI: 10.1007/BF01531027.

[GSB13] Alexandra Goultiaeva, Martina Seidl, and Armin Biere. „Bridging the gap between dual propagation and CNF-based QBF solving“. In: *Proceedings of DATE*. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 811–814. DOI: 10.7873/DATE.2013.172.

[GSV14] Orna Grumberg, Sharon Shoham, and Yakir Vizel. „SAT-based Model Checking: Interpolation, IC3, and Beyond“. In: *Software Systems Safety*. Vol. 36. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2014, pp. 17–41. DOI: 10.3233/978-1-61499-385-8-17.

[HSB14a] Marijn Heule, Martina Seidl, and Armin Biere. „A Unified Proof System for QBF Preprocessing“. In: *Proceedings of IJCAR*. Vol. 8562. LNCS. Springer, 2014, pp. 91–106. DOI: 10.1007/978-3-319-08587-6_7.

[HSB14b] Marijn Heule, Martina Seidl, and Armin Biere. „Efficient extraction of Skolem functions from QRAT proofs“. In: *Proceedings of FMCAD*. IEEE, 2014, pp. 107–114. DOI: 10.1109/FMCAD.2014.6987602.

[HSB17] Marijn J. H. Heule, Martina Seidl, and Armin Biere. „Solution Validation and Extraction for QBF Preprocessing“. In: *J. Autom. Reasoning* 58.1 (2017), pp. 97–125. DOI: 10.1007/s10817-016-9390-4.

[HST19] Christopher Hahn, Marvin Stenger, and Leander Tentrup. „Constraint-Based Monitoring of Hyperproperties“. In: *Proceedings of TACAS*. Vol. 11428. LNCS. Springer, 2019, pp. 115–131. DOI: 10.1007/978-3-030-17465-1_7.

[HT18] Jesko Hecking-Harbusch and Leander Tentrup. „Solving QBF by Abstraction“. In: *Proceedings of GandALF*. Vol. 277. EPTCS. 2018, pp. 88–102. DOI: 10.4204/EPTCS.277.7.

[IP01] Russell Impagliazzo and Ramamohan Paturi. „On the Complexity of k-SAT“. In: *J. Comput. Syst. Sci.* 62.2 (2001), pp. 367–375. DOI: 10.1006/jcss.2000.1727.

[Jac+16] Swen Jacobs, Roderick Bloem, Romain Brenguier, Ayrat Khalimov, Felix Klein, Robert Könighofer, Jens Kreber, Alexander Legg, Nina Narodytska, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. „The 3rd Reactive Synthesis Competition (SYNTCOMP 2016): Benchmarks, Participants & Results“. In: *Proceedings of SYNT@CAV*. Vol. 229. EPTCS. 2016, pp. 149–177. DOI: 10.4204/EPTCS.229.12.

[Jac+17a] Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur, and Leander Tentrup. „The 4th Reactive Synthesis Competition (SYNTCOMP 2017): Benchmarks, Participants & Results“. In: *Proceedings of SYNT@CAV*. Vol. 260. EPTCS. 2017, pp. 116–143. DOI: 10.4204/EPTCS.260.10.

[Jac+17b] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. „The first reactive synthesis competition (SYNTCOMP 2014)“. In: *STTT* 19.3 (2017), pp. 367–390. DOI: 10.1007/s10009-016-0416-3.

[Jan+12] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. „Solving QBF with Counterexample Guided Refinement“. In: *Proceedings of SAT*. Vol. 7317. LNCS. Springer, 2012, pp. 114–128. DOI: 10.1007/978-3-642-31612-8_10.

[Jan+16] Mikolás Janota, William Klieber, Joao 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.

[Jan16] Mikolás Janota. „On Q-Resolution and CDCL QBF Solving“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 402–418. DOI: 10.1007/978-3-319-40970-2_25.

[Jan18a] Mikolás Janota. „Circuit-Based Search Space Pruning in QBF“. In: *Proceedings of SAT*. Vol. 10929. LNCS. Springer, 2018, pp. 187–198. DOI: 10.1007/978-3-319-94144-8_12.

[Jan18b] Mikolás Janota. „Towards Generalization in QBF Solving via Machine Learning“. In: *Proceedings of AAAI*. AAAI Press, 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16945.

[JGM13] Mikolás Janota, Radu Grigore, and João Marques-Silva. „On QBF Proofs and Preprocessing“. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Vol. 8312. LNCS. Springer, 2013, pp. 473–489. DOI: 10.1007/978-3-642-45221-5_32.

[JLH09] Jie-Hong Roland Jiang, Hsuan-Po Lin, and Wei-Lun Hung. „Interpolating functions from large Boolean relations“. In: *Proceedings of ICCAD*. ACM, 2009, pp. 779–784. DOI: 10.1145/1687399.1687544.

[JM13] Mikolás Janota and João Marques-Silva. „On Propositional QBF Expansions and Q-Resolution“. In: *Proceedings of SAT*. Vol. 7962. LNCS. Springer, 2013, pp. 67–82. DOI: 10.1007/978-3-642-39071-5_7.

[JM15a] Mikolás Janota and Joao 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.

[JM15b] Mikolás Janota and Joao Marques-Silva. „Solving QBF by Clause Selection“. In: *Proceedings of IJCAI*. AAAI Press, 2015, pp. 325–331.

[JM17] Mikolás Janota and João Marques-Silva. „An Achilles' Heel of Term-Resolution“. In: *Proceedings of EPIA*. Vol. 10423. LNCS. Springer, 2017, pp. 670–680. DOI: 10.1007/978-3-319-65340-2_55.

[Job+07] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. „Anzu: A Tool for Property Synthesis“. In: *Proceedings of CAV*. Vol. 4590. LNCS. Springer, 2007, pp. 258–262. DOI: 10.1007/978-3-540-73368-3_29.

[Jor+14] Charles Jordan, Lukasz Kaiser, Florian Lonsing, and Martina Seidl. „MPIDepQBF: Towards Parallel QBF Solving without Knowledge Sharing“. In: *Proceedings of SAT*. Vol. 8561. LNCS. Springer, 2014, pp. 430–437. DOI: 10.1007/978-3-319-09284-3_32.

[JS97] Roberto J. Bayardo Jr. and Robert Schrag. „Using CSP Look-Back Techniques to Solve Real-World SAT Instances“. In: *Proceedings of AAAI*. AAAI Press / The MIT Press, 1997, pp. 203–208. URL: http://www.aaai.org/Library/AAAI/1997/aaai97-032.php.

[Jus+07] Toni Jussila, Armin Biere, Carsten Sinz, Daniel Kröning, and Christoph M. Wintersteiger. „A First Step Towards a Unified Proof Checker for QBF“. In: *Proceedings of SAT*. Vol. 4501. LNCS. Springer, 2007, pp. 201–214. DOI: 10.1007/978-3-540-72788-0_21.

[KJB13a] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. „PARTY Parameterized Synthesis of Token Rings“. In: *Proceedings of CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 928–933. DOI: 10.1007/978-3-642-39799-8_66.

[KJB13b] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. „Towards Efficient Parameterized Synthesis“. In: *Proceedings of VMCAI*. Vol. 7737. LNCS. Springer, 2013, pp. 108–127. DOI: 10.1007/978-3-642-35873-9_9.

[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.

[Kli+10] William Klieber, Samir Sapra, Sicun Gao, and Edmund M. Clarke. „A Non-prenex, Non-clausal QBF Solver with Game-State Learning“. In: *Proceedings of SAT*. Vol. 6175. LNCS. Springer, 2010, pp. 128–142. DOI: 10.1007/978-3-642-14186-7_12.

[Kon+09] Roman Kontchakov, Luca Pulina, Ulrike Sattler, Thomas Schneider, Petra Selmer, Frank Wolter, and Michael Zakharyaschev. „Minimal Module Extraction from DL-Lite Ontologies Using QBF Solvers“. In: *Proceedings of IJCAI*. 2009, pp. 836–841. URL: http://ijcai.org/Proceedings/09/Papers/143.pdf.

[Kor08] Konstantin Korovin. „iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description)“. In: *Proceedings of IJCAR*. Vol. 5195. LNCS. Springer, 2008, pp. 292–298. DOI: 10.1007/978-3-540-71070-7_24.

[KP10] Uri Klein and Amir Pnueli. „Revisiting Synthesis of GR(1) Specifications“. In: *Proceedings of HVC*. Vol. 6504. LNCS. Springer, 2010, pp. 161–181. DOI: 10.1007/978-3-642-19583-9_16.

[KS18a] Manuel Kauers and Martina Seidl. „Short proofs for some symmetric Quantified Boolean Formulas“. In: *Inf. Process. Lett.* 140 (2018), pp. 4–7. DOI: 10.1016/j.ipl.2018.07.009.

[KS18b] Manuel Kauers and Martina Seidl. „Symmetries of Quantified Boolean Formulas“. In: *Proceedings of SAT*. Vol. 10929. LNCS. Springer, 2018, pp. 199–216. DOI: 10.1007/978-3-319-94144-8_13.

[KS19] Benjamin Kiesl and Martina Seidl. „QRAT Polynomially Simulates ∀ \text -Exp+Res“. In: *Proceedings of SAT*. Vol. 11628. LNCS. Springer, 2019, pp. 193–202. DOI: 10.1007/978-3-030-24258-9_13.

[KV01] Orna Kupferman and Moshe Y. Vardi. „Synthesizing Distributed Systems“. In: *Proceedings of LICS*. IEEE Computer Society, 2001, pp. 389–398. DOI: 10.1109/LICS.2001.932514.

[KV05] Orna Kupferman and Moshe Y. Vardi. „Safraless Decision Procedures“. In: *Proceedings of FOCS*. IEEE Computer Society, 2005, pp. 531–542. DOI: 10.1109/SFCS.2005.66.

[KV97] Orna Kupferman and Moshe Y. Vardi. „Synthesis with incomplete information“. In: *ICTL*. 1997.

[KZ15] Felix Klein and Martin Zimmermann. „How Much Lookahead is Needed to Win Infinite Games?“ In: *Proceedings of ICALP*. Vol. 9135. LNCS. Springer, 2015, pp. 452–463. DOI: 10.1007/978-3-662-47666-6_36.

[LB08] Florian Lonsing and Armin Biere. „Nenofex: Expanding NNF for QBF Solving“. In: *Proceedings of SAT*. Vol. 4996. LNCS. Springer, 2008, pp. 196–210. DOI: 10.1007/978-3-540-79719-7_19.

[LB10] Florian Lonsing and Armin Biere. „DepQBF: A Dependency-Aware QBF Solver“. In: *JSAT* 7.2-3 (2010), pp. 71–76. URL: https://satassociation.org/jsat/index.php/jsat/article/view/84.

[LE14] Florian Lonsing and Uwe Egly. „Incremental QBF Solving by DepQBF“. In: *Proceedings of ICMS*. Vol. 8592. LNCS. Springer, 2014, pp. 307–314. DOI: 10.1007/978-3-662-44199-2_48.

[LE17] Florian Lonsing and Uwe Egly. „DepQBF 6.0: A Search-Based QBF Solver Beyond Traditional QCDCL“. In: *Proceedings of CADE*. Vol. 10395. LNCS. Springer, 2017, pp. 371–384. DOI: 10.1007/978-3-319-63046-5_23.

[LE18a] Florian Lonsing and Uwe Egly. „Evaluating QBF Solvers: Quantifier Alternations Matter“. In: *Proceedings of CP*. Vol. 11008. LNCS. Springer, 2018, pp. 276–294. DOI: 10.1007/978-3-319-98334-9_19.

[LE18b] Florian Lonsing and Uwe Egly. „QRAT+: Generalizing QRAT by a More Powerful QBF Redundancy Property“. In: *Proceedings of IJCAR*. Vol. 10900. LNCS. Springer, 2018, pp. 161–177. DOI: 10.1007/978-3-319-94205-6_12.

[LES16] Florian Lonsing, Uwe Egly, and Martina Seidl. „Q-Resolution with Generalized Axioms“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 435–452. DOI: 10.1007/978-3-319-40970-2_27.

[Lon+15] Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. „Enhancing Search-Based QBF Solving by Dynamic Blocked Clause Elimination“. In: *Proceedings of LPAR*. Vol. 9450. LNCS. Springer, 2015, pp. 418–433. DOI: 10.1007/978-3-662-48899-7_29.

[LS18] Florian Lonsing and Martina Seidl. „Parallel Solving of Quantified Boolean Formulas“. In: *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 101–139. DOI: 10.1007/978-3-319-63516-3_4.

[LWJ18] Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R. Jiang. „Solving Exist-Random Quantified Stochastic Boolean Satisfiability via Clause Selection“. In: *Proceedings of IJCAI*. ijcai.org, 2018, pp. 1339–1345. DOI: 10.24963/ijcai.2018/186.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. „Z3: An Efficient SMT Solver“. In: *Proceedings of TACAS*. Vol. 4963. LNCS. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MC18] Thibaud Michaud and Maximilien Colange. „Reactive Synthesis from LTL Specification with Spot“. In: *Proceedings of SYNT@CAV*. 2018.

[McC88] Daryl McCullough. „Noninterference and the composability of security properties“. In: *Proceedings of S&P*. IEEE Computer Society, 1988, pp. 177–186. DOI: 10.1109/SECPRI.1988.8110.

[McM03] Kenneth L. McMillan. „Interpolation and SAT-Based Model Checking“. In: *Proceedings of CAV*. Vol. 2725. LNCS. Springer, 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1.

[MNS10] Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. „A new parallel architecture for QBF tools“. In: *Proceedings of HPCS*. IEEE, 2010, pp. 324–330. DOI: 10.1109/HPCS.2010.5547114.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995. ISBN: 978-0-387-94459-3.

[MS16] Meena Mahajan and Anil Shukla. „Level-ordered Q-resolution and tree-like Q-resolution are incomparable“. In: *Inf. Process. Lett.* 116.3 (2016), pp. 256–258. DOI: 10.1016/j.ipl.2015.11.017.

[MSB13] Christian Miller, Christoph Scholl, and Bernd Becker. „Proving QBF-hardness in Bounded Model Checking for Incomplete Designs“. In: *Proceedings of MTV*. IEEE Computer Society, 2013, pp. 23–28. DOI: 10.1109/MTV.2013.11.

[MSL18] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. „Strix: Explicit Reactive Synthesis Strikes Back!“ In: *Proceedings of CAV*. Vol. 10981. LNCS. Springer, 2018, pp. 578–586. DOI: 10.1007/978-3-319-96145-3_31.

[New+14] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. „Impact of Community Structure on SAT Solver Performance“. In: *Proceedings of SAT*. Vol. 8561. LNCS. Springer, 2014, pp. 252–268. DOI: 10.1007/978-3-319-09284-3_20.

[Nie+12] Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. „Resolution-Based Certificate Extraction for QBF - (Tool Presentation)“. In: *Proceedings of SAT*. Vol. 7317. LNCS. Springer, 2012, pp. 430–435. DOI: 10.1007/978-3-642-31612-8_33.

[NO05] Robert Nieuwenhuis and Albert Oliveras. „DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic“. In: *Proceedings of CAV*. Vol. 3576. LNCS. Springer, 2005, pp. 321–334. DOI: 10.1007/11513988_33.

[NPB14] Aina Niemetz, Mathias Preiner, and Armin Biere. „Turbo-charging Lemmas on demand with don't care reasoning“. In: *Proceedings of FMCAD*. IEEE, 2014, pp. 179–186. DOI: 10.1109/FMCAD.2014.6987611.

[NPT06] Massimo Narizzano, Luca Pulina, and Armando Tacchella. „The QBFEVAL Web Portal“. In: *Proceedings of JELIA*. Vol. 4160. LNCS. Springer, 2006, pp. 494–497. DOI: 10.1007/11853886_45.

[NSB07] Tobias Nopper, Christoph Scholl, and Bernd Becker. „Computation of minimal counterexamples by using black box techniques and symbolic methods“. In: *Proceedings of ICCAD*. IEEE Computer Society, 2007, pp. 273–280. DOI: 10.1109/ICCAD.2007.4397277.

[Pit07] Nir Piterman. „From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata“. In: *Logical Methods in Computer Science* 3.3 (2007). DOI: 10.2168/LMCS-3(3:5)2007.

[Pnu77] Amir Pnueli. „The Temporal Logic of Programs“. In: *Proceedings of FOCS*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. „Synthesis of Reactive(1) Designs". In: *Proceedings of VMCAI*. Vol. 3855. LNCS. Springer, 2006, pp. 364–380. DOI: 10.1007/11609773_24.

[PR79] Gary L. Peterson and John H. Reif. „Multiple-Person Alternation". In: *Proceedings of FOCS*. IEEE Computer Society, 1979, pp. 348–363. DOI: 10.1109/SFCS.1979.25.

[PR89] Amir Pnueli and Roni Rosner. „On the Synthesis of a Reactive Module". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 179–190. DOI: 10.1145/75277.75293.

[PR90] Amir Pnueli and Roni Rosner. „Distributed Reactive Systems Are Hard to Synthesize". In: *Proceedings of FOCS*. IEEE Computer Society, 1990, pp. 746–757. DOI: 10.1109/FSCS.1990.89597.

[PRA01] Gary Peterson, John Reif, and Syed Azhar. „Lower Bounds for Multiplayer Non-Cooperative Games of Incomplete Information". In: *Computers and Mathematics with Applications* 41 (2001), pp. 957–992.

[PS19] Luca Pulina and Martina Seidl. „The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17)". In: *Artif. Intell.* 274 (2019), pp. 224–248. ISSN: 0004-3702. DOI: 10.1016/j.artint.2019.04.002.

[PSS17] Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. „Dependency Learning for QBF". In: *Proceedings of SAT*. Vol. 10491. LNCS. Springer, 2017, pp. 298–313. DOI: 10.1007/978-3-319-66263-3_19.

[QBF14] QBF Gallery 2014. *QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas*. Tech. rep. 2014. URL: http://qbf.satisfiability.org/gallery/qcir-gallery14.pdf.

[Rab+18] Markus N. Rabe, Leander Tentrup, Cameron Rasmussen, and Sanjit A. Seshia. „Understanding and Extending Incremental Determinization for 2QBF". In: *Proceedings of CAV*. Vol. 10982. LNCS. Springer, 2018, pp. 256–274. DOI: 10.1007/978-3-319-96142-2_17.

[Rab17] Markus N. Rabe. „A Resolution-Style Proof System for DQBF". In: *Proceedings of SAT*. Vol. 10491. LNCS. Springer, 2017, pp. 314–325. DOI: 10.1007/978-3-319-66263-3_20.

[Rin07] Jussi Rintanen. „Asymptotically Optimal Encodings of Conformant Planning in QBF". In: *Proceedings of AAAI*. AAAI Press, 2007, pp. 1045–1050. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-166.php.

[RS16] Markus N. Rabe and Sanjit A. Seshia. „Incremental Determinization". In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 375–392. DOI: 10.1007/978-3-319-40970-2_23.

[RT15] Markus N. Rabe and Leander Tentrup. „CAQE: A Certifying QBF Solver“. In: *Proceedings of FMCAD*. IEEE, 2015, pp. 136–143. DOI: 10.1109/FMCAD.2015.7542263.

[Saf88] Shmuel Safra. „On the Complexity of omega-Automata“. In: *Proceedings of FOCS*. IEEE Computer Society, 1988, pp. 319–327. DOI: 10.1109/SFCS.1988.21948.

[SB01] Christoph Scholl and Bernd Becker. „Checking Equivalence for Partial Implementations“. In: *Proceedings of DAC*. ACM, 2001, pp. 238–243. DOI: 10.1145/378239.378471.

[SB07] Stefan Staber and Roderick Bloem. „Fault Localization and Correction with QBF“. In: *Proceedings of SAT*. Vol. 4501. LNCS. Springer, 2007, pp. 355–368. DOI: 10.1007/978-3-540-72788-0_34.

[SC85] A. Prasad Sistla and Edmund M. Clarke. „The Complexity of Propositional Linear Temporal Logics“. In: *J. ACM* 32.3 (1985), pp. 733–749. DOI: 10.1145/3828.3837.

[SF06] Sven Schewe and Bernd Finkbeiner. „Synthesis of Asynchronous Systems“. In: *Proceedings of LOPSTR*. Vol. 4407. LNCS. Springer, 2006, pp. 127–142. DOI: 10.1007/978-3-540-71410-1_10.

[SF07] Sven Schewe and Bernd Finkbeiner. „Bounded Synthesis“. In: *Proceedings of ATVA*. Vol. 4762. LNCS. Springer, 2007, pp. 474–488. DOI: 10.1007/978-3-540-75596-8_33.

[SHY15] Masaya Shimakawa, Shigeki Hagihara, and Naoki Yonezaki. „Reducing Bounded Realizability Analysis to Reachability Checking“. In: *Proceedings of RP*. Vol. 9328. LNCS. Springer, 2015, pp. 140–152. DOI: 10.1007/978-3-319-24537-9_13.

[SK14] Martina Seidl and Robert Könighofer. „Partial witnesses from preprocessed quantified Boolean formulas“. In: *Proceedings of DATE*. European Design and Automation Association, 2014, pp. 1–6. DOI: 10.7873/DATE.2014.162.

[SM09] Igor Stéphan and Benoit Da Mota. „A Unified Framework for Certificate and Compilation for QBF“. In: *Proceedings of ICLA*. Vol. 5378. LNCS. Springer, 2009, pp. 210–223. DOI: 10.1007/978-3-540-92701-3_15.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. „Word Problems Requiring Exponential Time: Preliminary Report“. In: *Proceedings of STOC*. ACM, 1973, pp. 1–9. DOI: 10.1145/800125.804029.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. „Extending SAT Solvers to Cryptographic Problems“. In: *Proceedings of SAT*. Vol. 5584. LNCS. Springer, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[Sol+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. „Combinatorial sketching for finite programs“. In: *Proceedings of ASPLOS*. ACM, 2006, pp. 404–415. DOI: 10.1145/1168857.1168907.

[SP16] Christoph Scholl and Florian Pigorsch. „The QBF Solver AIGSolve“. In: *Proceedings of QBF@SAT*. Vol. 1719. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 55–62. URL: http://ceur-ws.org/Vol-1719/paper5.pdf.

[SS09a] Marko Samer and Stefan Szeider. „Backdoor Sets of Quantified Boolean Formulas“. In: *J. Autom. Reasoning* 42.1 (2009), pp. 77–97. DOI: 10.1007/s10817-008-9114-5.

[SS09b] Saqib Sohail and Fabio Somenzi. „Safety first: A two-stage algorithm for LTL games“. In: *Proceedings of FMCAD*. IEEE, 2009, pp. 77–84. DOI: 10.1109/FMCAD.2009.5351138.

[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.

[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.

[Ten13] Leander Tentrup. „Detecting Unrealizable Specifications of Distributed Systems“. Master's Thesis. Saarland University, Dec. 2013.

[Ten16] Leander Tentrup. „Non-prenex QBF Solving Using Abstraction“. In: *Proceedings of SAT*. Vol. 9710. LNCS. Springer, 2016, pp. 393–401. DOI: 10.1007/978-3-319-40970-2_24.

[Ten17] Leander Tentrup. „On Expansion and Resolution in CEGAR Based QBF Solving“. In: *Proceedings of CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 475–494. DOI: 10.1007/978-3-319-63390-9_25.

[Ten19] Leander Tentrup. „CAQE and QuAbS: Abstraction Based QBF solvers“. Accepted for publication in JSAT. 2019.

[THJ15] Kuan-Hua Tu, Tzu-Chien Hsu, and Jie-Hong R. Jiang. „QELL: QBF Reasoning with Extended Clause Learning and Levelized SAT Solving“. In: *Proceedings of SAT*. Vol. 9340. LNCS. Springer, 2015, pp. 343–359. DOI: 10.1007/978-3-319-24318-4_25.

[TR19a] Leander Tentrup and Markus N. Rabe. „Clausal Abstraction for DQBF“. In: *Proceedings of SAT*. Vol. 11628. LNCS. Springer, 2019, pp. 388–405. DOI: 10.1007/978-3-030-24258-9_27.

[TR19b] Leander Tentrup and Markus N. Rabe. „Clausal Abstraction for DQBF (full version)“. In: *CoRR* abs/1808.08759 (2019). arXiv: 1808.08759. URL: http://arxiv.org/abs/1808.08759.

[Tse68] Grigori S Tseitin. „On the complexity of derivation in propositional calculus“. In: *Studies in constructive mathematics and mathematical logic* 2.115-125 (1968), pp. 10–13.

[VW19] Nikhil Vyas and Ryan Williams. „On Super Strong ETH“. In: *Proceedings of SAT*. Vol. 11628. LNCS. Springer, 2019, pp. 406–423. DOI: 10.1007/978-3-030-24258-9_28.

[VW94] Moshe Y. Vardi and Pierre Wolper. „Reasoning About Infinite Computations“. In: *Inf. Comput.* 115.1 (1994), pp. 1–37. DOI: 10.1006/inco.1994.1092.

[Wim+15] Ralf Wimmer, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker. „Preprocessing for DQBF“. In: *Proceedings of SAT*. Vol. 9340. LNCS. Springer, 2015, pp. 173–190. DOI: 10.1007/978-3-319-24318-4_13.

[Wim+16] Karina Wimmer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. „Skolem Functions for DQBF“. In: *Proceedings of ATVA*. Vol. 9938. LNCS. 2016, pp. 395–411. DOI: 10.1007/978-3-319-46520-3_25.

[Wim+17] Ralf Wimmer, Sven Reimer, Paolo Marin, and Bernd Becker. „HQSpre - An Effective Preprocessor for QBF and DQBF“. In: *Proceedings of TACAS*. Vol. 10205. LNCS. 2017, pp. 373–390. DOI: 10.1007/978-3-662-54577-5_21.

[Zha14] Wenhui Zhang. „QBF Encoding of Temporal Properties and QBF-Based Verification“. In: *Proceedings of IJCAR*. Vol. 8562. LNCS. Springer, 2014, pp. 224–239. DOI: 10.1007/978-3-319-08587-6_16.

[ZM02] Lintao Zhang and Sharad Malik. „Conflict driven learning in a quantified Boolean Satisfiability solver“. In: *Proceedings of ICCAD*. ACM / IEEE Computer Society, 2002, pp. 442–449. DOI: 10.1145/774572.774637.

[ZM03] Steve Zdancewic and Andrew C. Myers. „Observational Determinism for Concurrent Program Security“. In: *Proceedings of CSFW*. IEEE Computer Society, 2003, p. 29. DOI: 10.1109/CSFW.2003.1212703.

# Index