JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Dipl.-Inf.
Andreas Fröhlich

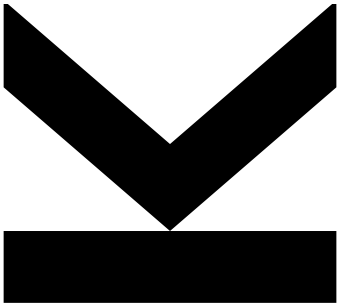
Angefertigt am
Institut für Formale Modelle und Verifikation

Erstbeurteiler
Univ.-Prof. Dr.
Armin Biere

Zweitbeurteiler
Univ.-Prof. Dr.
Christoph Scholl

März 2016

Theoretical and Practical Aspects of Bit-Vector Reasoning



Dissertation
zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften
im Doktoratsstudium der
Technischen Wissenschaften

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Satisfiability Modulo Theories (SMT) is a broad field of research and an important topic for many practical applications. In this thesis, we focus on theories of bit-vectors as used, e.g., in hardware and software verification, but also in many other areas. In particular, we discuss satisfiability of bit-vector logics and related problems. The satisfiability problem of propositional formulas (SAT) is a well-known NP-complete problem. In the past, satisfiability for quantifier-free bit-vector logics was often assumed to be NP-complete as well. We show that several earlier complexity results for bit-vector logics only hold if a unary encoding for scalars is used. Instead, complexity for many decision problems grows significantly, if a more succinct logarithmic encoding is used for representation. As one of our central results, we prove that satisfiability of quantifier-free bit-vector formulas turns out to be NEXPTIME-complete. We also show that similar results can be obtained for satisfiability of quantified bit-vector logics with uninterpreted functions (2-NEXPTIME-complete), and can even be extended to multi-logarithmic encodings. For the latter case, we give a very general ν-NEXPTIME-completeness result, when considering bit-vector logics with ν-logarithmic scalar encodings. We further analyze how the choice of operators affects the expressiveness of certain bit-vector logics and can lead to specific fragments of quantifier-free bit-vector logics that are PSPACE-complete or NP-complete. On the practical side, this implies that the bit-blasting followed by the use of a CDCL (conflict driven clause learning) SAT solver, which is the common approach in state-of-the-art bit-vector solvers, can be exponential. Our complexity results directly point to several possibilities for new solving approaches, proposing reductions to model checking (for the PSPACE-complete fragment), to EPR (for the general, NEXPTIME-complete class), or by applying SLS (stochastic local search) directly on the theory level. We also develop two algorithms for solving Dependency Quantified Boolean Formulas (DQBF), a further NEXPTIME-complete problem, either using a DPLL based algorithm or an instantiation based approach, similar to the one applied in EPR solving.

Zusammenfassung

Erfüllbarkeit Modulo einer Theorie (SMT) ist ein breites Forschungsgebiet mit vielen praktische Anwendungen. Diese Dissertation setzt ihren Fokus auf Theorien über Bitvektoren, wie sie zum Beispiel in der Hardware und Software Verfikation - aber auch in vielen anderen Bereichen - Verwendung finden. Insbesondere diskutieren wir die Erfüllbarkeit von Bitvektor-Logiken und verwandte Probleme. Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist ein sehr bekanntes NP-vollständiges Problem. Bisher wurde oft angenommen, dass das Erfüllbarkeitsproblem für quantorenfreie Bitvektor-Logiken ebenfalls NP-vollständig sei. Wir zeigen, dass viele frühere Komplexitäts-Resultate für Bitvektor-Logiken nur dann gelten, wenn enthaltene Skalare unär kodiert werden. Im Gegensatz dazu steigt die Komplexität vieler Entscheidungsprobleme drastisch, wenn eine kompaktere logarithmische Kodierung zur Darstellung verwendet wird. Unter anderem beweisen wir, dass das Erfüllbarkeitsproblem für quantorenfreie Bitvektor Formeln NEXPTIME-vollständig ist. Des Weiteren zeigen wir, dass vergleichbare Resultate auch für das Erfüllbarkeitsproblem von quantifizierten Bitvektor-Logiken mit uninterpretierten Funktionen (2-NEXPTIME-vollständig), sowie für Logiken mit multi-logarithmisch kodierten Skalaren gelten. Für den zweiten Fall zeigen wir ν-NEXPTIME-Vollständigkeit, wenn eine ν-logarithmische Kodierung für Skalare verwendet wird. Wir analysieren zudem, wie sich die Wahl der Operatoren auf die Komplexität von verschiedenen Bitvektor-Logiken auswirkt und wie sie dazu führen kann, dass bestimmte Fragmente von quantorenfreien Bitvektor-Logiken PSPACE-vollständig oder NP-vollständig werden. Für die Praxis bedeuten unsere Resultate, dass der übliche Ansatz von Bitvektor Algorithmen, der "bit-blasting" mit der Verwendung von CDCL (konfliktgesteuerte Klauseln lernende) SAT Algorithmen kombiniert, exponentiell sein kann. Unsere Resultate eröffnen direkt mehrere neue Ansätze, z.B. durch Modellprüfung (für das PSPACE-vollständige Fragment), Übersetzung nach EPR (für die allgemeine, NEXPTIME-vollständige Klasse), oder Anwendung eines SLS (stochastische lokale Suche) Algorithmus direkt auf der Theorie-Repräsentation. Wir entwickeln zudem zwei Algorithmen um DQBF zu lösen, ein weiteres NEXPTIME-vollständiges Problem; entweder mit einem DPLL-basierten Algorithmus, oder durch einen auf Instanziierung basierenden Ansatz, ähnlich dem für EPR.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Acknowledgements

I would like to thank my parents, Roswitha and Wolfgang, for being supportive with me throughout my whole life—thanks for everything. This thesis is dedicated to them. I am also very thankful to my partner, Kathrin, for bearing with me when I decided to move to Linz, and for encouraging me to take the chance of working at Microsoft Research in Cambridge. Further thanks go to all my colleagues and co-authors—it was a joy working with you. Finally, I would like to thank Armin, for being the best supervisor any student could possibly hope for.

"In individuals, insanity is rare—but in groups,
parties, nations and epochs, it is the rule."

- Friedrich Wilhelm Nietzsche,
Beyond Good and Evil, 1886

Contents

Chapter 1

Introduction

This thesis is about *bit-vectors* in the widest sense. We will mainly deal with *satisfiability* of bit-vector formulas, which is a special case of SMT and an extension of SAT. We will discuss the importance of bit-vectors as well as approaches of finding solutions to bit-vector formulas and related problems, such as SAT, QBF, DQBF, and EPR. We will also deal with questions related to computational complexity of these problem classes. Looking at both, the theoretical side as well as practical applicability, we will, furthermore, show the connection between theoretical complexity and practical solving approaches. In this introduction, we first provide an informal overview as well as background information on the specific topics that will be discussed in the later chapters. The first part of this chapter, therefore, is mainly for the reader who is not familiar with the topics presented in this thesis. The second part then provides an outline of the remaining part of this work, giving a brief summary of the content of each chapter, as well as the main thread connecting each of the individual contributions. The reader with a deeper interest in the topic should feel referred to the *Handbook of Satisfiability* [31], which contains the probably most complete overview on the topic of satisfiability and related problems.

A further extensive discussion of satisfiability can also be found as a section in Volume 4B of Donald Knuth's "The Art of Computer Programming" [146]. In a preface to the corresponding section, Knuth states the following:

> Wow—Section 7.2.2.2 has turned out to be the longest section, by far, in *The Art of Computer Programming*. The SAT problem is evidently a killer app because it is key to the solution of so many other problems.

In the following, we will present our contributions to this topic.

1.1 Background

The question of satisfiability is one of the most essential problems in computer science. The satisfiability problem can be roughly characterized by the following

informal description: Given a (syntactically valid) formula over a well-defined logic, the answer to the satisfiability problem consists of deciding whether there exist values for the variables in that formula, so that the semantic statement of the formula is true. If such values for the variables exist, the formula is called *satisfiable*, otherwise it is said to be *unsatisfiable*.

Obviously, a formal description requires a definition of all components, such as "formula", "variable", and "true statements". This is done at multiple occasions for several logics in later chapters of this thesis, e.g., in Chapter 3, Chapter 5, or, most detailed, in Chapter 4. Thus, for the moment, we will stick to the more abstract level given by the informal definition, and extend it by giving some concrete examples. A detailed formal introduction is also given in [31].

1.1.1 Propositional Satisfiability

The most elementary version of satisfiability is given by the satisfiability problem of propositional logic, usually referred to as the "SAT problem" or just "SAT". Synonymously to propositional logic, also the term Boolean logic is used. In a propositional (or Boolean) formula, all variables and all composite expressions correspond to truth values. A truth value can either be *true* or *false*. Alternative symbols for *true* and *false* are also 1 and 0, or $\top$ and $\bot$, respectively.

Compound expressions are defined by using Boolean connectives, such as *logical and* ($\wedge$, also called *conjunction*), *logical or* ($\vee$, also called *disjunction*), or *negation* ($\neg$). Additional connectives exist, but can always be replaced by the previous ones (e.g., see [31]). While this is already true for only $\wedge$ and $\neg$ (e.g., using an and-inverter graph representation [159]), all three operators are used for the common *conjunctive normal form* (CNF) representation of a SAT formula. A formula is said to be in CNF, if it is a conjunction of *clauses*, with a clause being defined as a disjunction of *literals*, and a literal being defined as a variable or its negation. If all clauses in a formula are further restricted to contain exactly k literals, it is also said to be in k-CNF, and the corresponding decision problem is referred to as k-SAT. Using the well-known Tseitin-transformation, an arbitrary SAT formula can be translated into an equisatisfiable formula in CNF (even into a formula in k-CNF, for $k \geqslant 3$), with only polynomial growth in formula size [221]. More sophisticated approaches exist, e.g., the Plaisted-Greenbaum transformation [193]. Aside from several other advantages, CNF representations allow to apply *resolution*: Whenever clauses $(x \vee l_{1,1} \vee \cdots \vee l_{k_1,1})$ and $(\neg x \vee l_{1,2}, \cdots \vee l_{k_2,2})$ are part of a formula, adding a new clause $(l_{1,1} \vee \cdots \vee l_{k_1,1} \vee l_{1,2} \vee \cdots \vee l_{k_2,2})$, the so-called resolvent, results in an equivalent formula [80]. For propositional logic, iteratively applying the resolution rule is even guaranteed to refute every unsatisfiable formula [80]. Resolution is a key ingredient of most modern SAT solvers (e.g., [26, 184, 3, 5, 27]) and is also discussed for certain quantified formulas in Chapter 5.

Although the underlying logic is very simple, deciding SAT is already very difficult, regarding computational complexity. In particular, SAT was the first prob-

lem that was shown to be NP-complete [73]. The concept of NP-completeness is an essential one in complexity theory, defining sets of problems that have a certain "difficulty" regarding the computational effort required to solve them. For example, the complexity class NP is defined as the set of problems that allow the verification of a solution in polynomial time. Alternatively, the concept of Turing machines [222] can be used for the characterization of NP. In particular, NP corresponds to the set of problems that can be decided in polynomial time by a non-deterministic Turing machine. On the other hand, a problem is said to be NP-hard, if all other problems in NP can be polynomially reduced to that problem [142]. The set of NP-complete problems is exactly the set of problems that are contained in NP and that are NP-hard at the same time.

In 1971, Cook proved that any computational problem which can be solved by a non-deterministic Turing machine in polynomial time (i.e., any problem in NP) can also be encoded into a polynomial-sized SAT problem [73]. He showed this by giving a SAT encoding for any Turing machine instance. This implies NP-hardness of SAT and, with NP-inclusion being trivial to show (non-deterministically guess the truth values for all variables and check whether the formula evaluates to true), SAT was known to be NP-complete [73]. A similar result was also obtained independently by Levin, roughly at the same time, therefore, leading to the name *Cook-Levin theorem*. We will revisit the proof by Cook in Chapter 12, where a similar result for the more complex logics of multi-logarithmic encoded bit-vector formulas is shown.

NP-completeness of SAT is of huge importance when trying to practically solve it. In contrast to the hypothetical device of a Turing machine, actual computer architectures are strictly deterministic (leaving aside the concept of quantum computers[1]). Similarly to NP, the class P is said to be the set of problems that can be solved by a deterministic Turing machine in polynomial time. At the present moment, it is not known whether P $=$ NP and the answer to that question is one of the biggest unsolved problems in theoretical computer science. Nevertheless, the common belief among most experts is that P $\neq$ NP [107].

1.1.2 SAT Algorithms

Assuming P $\neq$ NP, it is implied that SAT, in the general case, cannot be solved in polynomial time by a program running on an actual computer, but instead needs superpolynomial time. For example, the naive approach of enumerating all possible combinations of truth values for the variables would require evaluating 2^n different combinations, with n being the number of variables in a given formula. While there are algorithms with smaller bounds on complexity, all those approaches still have exponential upper bounds. It is not clear whether SAT algorithms with subexponential worst case runtime exist. Actually, the *exponential time hypothesis* suggests that this is not the case, and that every algorithm to solve SAT requires an

[1] The relation between BQP (bounded error quantum polynomial time) and NP is unknown.

```
1 // input: formula F in CNF and a partial assignment beta
2 procedure DPLL(F, β)
3   if (F(β) = 1) // all clauses evaluate to true
4     return 1;
5   if (F(β) = 0) // at least one clause evaluates to false
6     return 0;
7   if (F contains a unit clause l) // unit propagation
8     return DPLL(F, β ∪ l ← 1);
9   x = pickVar() // select a variable to branch
10  return DPLL(F, β ∪ x ← 1) ∨ DPLL(F, β ∪ x ← 0);
```

Figure 1.1: Pseudo-code for a basic DPLL procedure. Evaluation of formulas, unit propagation, and the notion of partial assignments are discussed, e.g., in Chapter 5.

exponential runtime in the worst case [133].

At first, this might seem very demotivating for SAT research. Indeed, it is easy to see that the naive approach of enumerating all possible values will fail already for very small formulas, e.g., with 50 variables. Theoretical algorithms with smaller upper bounds can solve slightly larger formulas, but will also quickly reach their limits due to the exponential complexity. Furthermore, this would not even change with growing computational power. As a consequence, practically solving SAT formulas was widely believed to be computationally intractable, for a long time. However, this belief turned out to be too pessimistic. Instead, modern SAT solvers, meanwhile, are very efficient in dealing with practical problems, solving formulas with up to 10^7 variables. So, actually, SAT solvers should not work in theory; but they *do* work in practice [224].

Those modern solvers usually are heuristic solvers. No upper bounds on the runtime can be given and, for complexity reasons, they obviously are not able to solve *all* formulas. Nevertheless, it turns out that *practical instances*, coming from *structured formulas* and corresponding to *industrial problems*, or certain *combinatorial benchmarks*, can often be solved efficiently. The same is true for particular classes of *random benchmarks*, generated by using specific distributions regarding clauses and literals. It seems that those kind of benchmarks which actually show up in practice correspond to "easy" formulas and not to those that produce worst case results. Understanding why this is the case is still ongoing research as well as part of many discussions within the SAT community [224]. Improving the performance of actual SAT solvers, in the light of those circumstances, becomes even more challenging and more intriguing at the same time.

Most algorithms require the input formula to be in CNF. Until the last decade, people usually distinguished between two main kinds of approaches of SAT solving: DPLL based approaches and SLS ones. DPLL is short for Davis, Putnam, Logemann, and Loveland, the inventors of the original procedure [78], as an ex-

```
1 // input: formula F
2 procedure SLS(F)
3   α = init(); // generate an initial assignment
4   while (F(α) = 0) // if F is not satisfied yet
5     x = pickVar(); // select a variable ...
6     α(x) = ¬α(x); //... and change its value
7   return α
```

Figure 1.2: Pseudo-code for a typical SLS algorithm. Details and examples of a concrete implementation are given, e.g., in Chapter 10.

tension to the previous (resolution-based) DP algorithm [80]. The key idea behind the DPLL procedure is splitting the search space by *branching* on the two possible values of a variable and applying a simple deterministic inference rule called *unit propagation*. The pseudo-code for DPLL is given in Figure 1.1. An extension of the traditional DPLL procedure to certain quantified formulas, which we call DQDPLL, is defined in Chapter 5.

A new paradigm, introduced in the context of the solver Grasp [170], was the concept of *conflict-driven clause learning* (CDCL). Originally, this was considered to represent an extension of DPLL, by learning from conflicting variable assignments that occur during search. With emerging of more frequent and more sophisticated *restart techniques* over the last decade (e.g., [25, 223, 129, 196, 4, 30]), however, it is now often argued that modern CDCL solvers are meanwhile closer to some *guided resolution* approach than to the original DPLL one [4, 30]. In general, CDCL solvers are considered to be the state-of-the-art for SAT solving, in particular for the important case of *application instances*. Aside from clause learning and restart techniques, this is usually also attributed to sophisticated *variable selection strategies* [175, 25, 29]. Note that the application track of SAT competitions [12, 20] is dominated by CDCL solvers (e.g., [26, 184, 3, 5, 27]). CDCL solvers for SAT are not directly addressed in this thesis, but are found at the core of most state-of-the-art bit-vector solvers, such as used in Chapter 6, Chapter 8, and Chapter 10. They are further found in bounded model checkers, which are addressed in Chapter 8 as well. A direct extension for CDCL, in the context of our DQDPLL architecture, is presented in Chapter 5. Similarly, in Chapter 7, we use a CDCL solver as the core of an instantiation-based solver, for the same class of quantified formulas. The same kind of procedure is also applied in iProver, which is used for experiments in Chapter 6. We also mention related contributions to improving CDCL for SAT, in Chapter 11.

SLS is short for *stochastic local search*. This approach assigns values to all variables in a formula, i.e., samples the search space, and then modifies the current state by trying to find some *local* improvement in the sense of (hopefully) getting closer to a solution. This is usually done by using some probabilistic heuristic,

hence leading to the term *stochastic*. In contrast to DPLL (and CDCL) based methods, this kind of SLS algorithms are inherently incomplete, i.e., they cannot prove unsatisfiability of a formula. The pseudo-code for a basic SLS procedure is given in Figure 1.2. Classical SLS algorithms for SAT (e.g., [162, 15, 14, 163]) turned out to be particularly efficient for *random benchmarks* and *combinatorial benchmarks*, but usually perform bad on *application benchmarks*. This is often attributed to the fact that, due to their local nature, they are not able to make use of the particular structure inherent in industrial instances. In Chapter 10, we present an extension of the classical SLS approach for SAT to the more complex case of bit-vector logics, which turns out to be efficient on application instances as well. The state-of-the-art of SLS for SMT and possible directions for future work are further discussed in Chapter 12. We also mention related contributions to improving SLS for SAT in Chapter 11.

1.1.3 Quantification and Dependencies

A natural extension of propositional satisfiability is given by the introduction of quantifiers for variables. This leads to *Quantified Boolean Formulas* (QBF) or *Dependency Quantified Boolean Formulas* (DQBF). We distinguish between existential quantification and universal quantification, denoted by the symbols $\exists$ and $\forall$, respectively. When talking about SAT in the context of quantification, it is implicitly assumed that all variables in a propositional formula are existentially quantified. Note that, according to our original definition, a propositional formula is said to be satisfiable if and only if there *exist* values for all variables, so that the formula evaluates to true.

The actual extension, therefore, is found in the introduction of universal quantification. Universal quantification over a variable means that a certain statement, i.e., a specific subformula, should be true for both values that the particular variable can take. Whether certain other variables are allowed to take distinct values in the two subformulas depends on the dependency constraints of the formula. A formal definition using the concept of *assignment trees* is given, e.g., in Chapter 5. Dependency constraints can be defined implicitly by quantification order, as in the case of QBF, or explicitly using a functional representation, as done in DQBF. Both, QBF and DQBF, will play a role in several chapters, such as Chapters 2-5 and Chapter 7. Again, a more formal definition will be given in the specific chapters, e.g., most detailed in Chapter 5 and Chapter 7.

Note that, similar to SAT, its quantified extensions are considered to be prototypical problems for certain complexity classes. In particular, QBF is PSPACE-complete [185] and DQBF is NEXPTIME-complete [187, 188]. This will be important for several new complexity results presented in Chapters 2-4.

Practical solvers for QBF exist, e.g., [164, 24, 22, 165, 118, 135], all based on different kinds of approaches. For the even more complex case of DQBF, the situation is different. Our approach presented in Chapter 5 was the very first implementation of a DQBF solver, and our implementation of the algorithm in Chapter 7

is still the only publicly available solver at the time of this writing. However, several non-publicly available solvers now exist [111, 94].

A further extension of SAT, which will appear at several places throughout this thesis, is the class of *Effectively Propositional Logic* (EPR) formulas, also known as the Bernays-Schönfinkel class [23, 161]. Aside from quantification, EPR also allows arbitrary domains for the range of variables and the use of predicates, i.e., functions that map from the specific variable domain to the set of Boolean values. Quantification in EPR, however, is restricted in the sense that all variables are universally quantified and all predicates are implicitly assumed to be existentially quantified. EPR is a decidable fragment of first-order logic and it is NEXPTIME-complete [161]. Formally, EPR is usually not defined bottom-up (as an extension of SAT), but top-down (as a restriction of first-order logic), as a set of first-order formulas that have an $\exists^*\forall^*$ quantifier prefix and do not contain any function symbols (e.g. [161]). EPR will play a role in Chapter 6 and Chapter 7.

1.1.4 Satisfiability Modulo Theories

An even more general direction of extending propositional satisfiability is the one of *Satisfiability Modulo Theories* (SMT) [17]. The field of SMT considers the topic of propositional satisfiability in combination with some underlying background theory. Atomic elements of the propositional structure of a formula are no longer restricted to be Boolean variables, but can itself be compound subformulas over some background logic, mapping input variables from the specific theory to a truth value. SMT is more general than quantification, in the sense that quantification may also be part of a background theory. There are many different SMT logics in literature [17, 82] or, e.g., in the SMT-LIB [18]. Examples are the theory of arrays (with or without extensionality), lists, bit-vectors of fixed or non-fixed size, linear or non-linear arithmetic over floating point numbers, integer numbers, real numbers, and many more. Variations and combinations thereof are possible as well—quantified versions and quantifier-free ones. The SMT-LIB alone lists a set of 29 different logics [18], but possible logics are not restricted to this enumeration.

The most important logics, in the context of this thesis, concern theories of fixed size bit-vector formulas. Bit-vector representations play an important role in many practical applications of computer science, most prominently in hard- and software verification, but increasingly also in other scientific areas. While we discuss quantified bit-vector formulas at several occasions, e.g., in Chapter 2 and Chapter 4, the focus will be on quantifier-free ones.

Note that, compared to other theories, bit-vector logics are also closest to propositional logic, in the sense that bit-vector expressions can always be directly translated into a Boolean formula by using the circuit representation of a certain bit-vector operation, as realized in computer hardware. This is also the approach that is used by most state-of-the-art bit-vector solvers: The bit-vector formula is first encoded into a propositional formula (also called *bit-blasting*), and then handed over to a SAT solver. Due to this strong connection, improving SAT solvers and

bit-vector solvers often goes hand in hand and it is natural to also consider SAT solvers, when trying to improve bit-vector solving.

We explicitly deal with bit-vector logics from two different perspectives: Analyzing theoretical complexity, and efficiently solving formulas in practice. For example, in Chapters 2-4, we will provide new complexity results. Note that even quantifier-free bit-vector logics turn out to be NEXPTIME-complete, in the general case. In Chapter 12, we further extend those results, showing that bit-vector logics also provide prototypical representations for a whole hierarchy of complexity classes. Approaches to practically solve quantifier-free bit-vector formulas can then be found in Chapter 6, Chapter 8, and Chapter 10.

1.2 Outline

In this thesis, we approach the topic of bit-vectors from several directions, providing various theoretical results as well as practical solving approaches. The first part of this work is about theoretical properties of bit-vector logics. To really understand a computational problem and the difficulties that might arise with it, it is essential to know its complexity. Aside from being of interest on its own, knowing the complexity of a problem class, then again, can often help to find practical solving approaches. The second part of this work is more related to the practical aspects of solving bit-vector formulas. This, however, does not necessarily imply directly solving bit-vector formulas. Actually, the larger part of this work deals with algorithms for other problems, e.g., the development of DQBF solvers, or reencoding of bit-vector formulas into SMV or EPR. Practically, solving bit-vector formulas, in the first place, comes down to providing a decision procedure. This, however, is a very general concept. For example, we can either give a decision procedure that is directly operating on a bit-vector representation, or we can translate the decision problem for a bit-vector formula into a different kind of problem for which we already have an efficient decision procedure. The latter approach is what usually happens in practice, since state-of-the-art bit-vector solvers mainly rely on bit-blasting (i.e., translating the bit-vector formula to a SAT formula) and then calling a SAT solver (e.g., see [47, 50, 104, 81, 87]). This already shows why improving state-of-the-art SAT solvers can and often will directly affect bit-vector solvers as well. Apart from this, our previous complexity results help us identify other possible target logics.

The main contributions of this work are given in Chapters 2–10, each representing a paper which contains novel research and has been published. Apart from Chapter 9, which is a benchmark description, all papers were peer-reviewed during the publication process. Publication status of the individual papers, co-authors, publisher, and related conference or workshop are given at the beginning of each chapter as well as in the bibliography at the end of this thesis.

Note that all chapters might contain small modifications compared to the original publication, mainly of aesthetic nature. Those modifications include, e.g.,

fixing of typos, changes in layout due to differences in stylesheets, and renumbering citations due to the shared bibliography at the end of this work. Moreover, all occurrences of "in this paper" have been replaced by "in this chapter", as was also done for similar terms. However, note that none of those changes affects the content. Some small bugs are addressed in the next section, after the outline, and before the main part of this work. We will now provide a short motivation for each topic of research in the individual papers represented by Chapters 2–10 and describe its main contribution in a few words. After this, a brief summary of the remainder of this work will be given.

Chapter 2 [151]. In the past, the decision problem for quantifier-free bit-vector formulas has often been assumed to be NP-complete. In Chapter 2, we show that this assumption does not hold in general and that the problem actually becomes NEXPTIME-complete, as soon as we allow logarithmic encoding of bit-widths. This is shown by giving a reduction from DQBF to QF_BV. On the practical side, this implies that bit-blasting can result in an exponentially larger SAT representation. We also give several other complexity results for related problems, discussing in detail the difference between unary and binary encodings. For example, UFBV, the quantified case with uninterpreted functions, is shown to be 2-NEXPTIME-complete.

Chapter 3 [101]. The work from Chapter 2 is extended in Chapter 3 by restricting the usage of certain operators in bit-vector logics. We end up with three very simplistic classes of bit-vector formulas, showing that each of it is a prototypical characterization of a decision problem which is complete for a certain class. In particular, it is sufficient to consider bit-vector formulas with bitwise operations, equality, and different restrictions on the shift operator. Depending on whether shifts by arbitrary constants are allowed, shifts by only the constant 1 are allowed, or no shifts are allowed, the resulting logic turns out to be complete for NEXPTIME, PSPACE, or NP, respectively.

Chapter 4 [153]. We then continue to analyze the expressiveness of individual operators in Chapter 4, as well as the influence of certain other extensions or restrictions to bit-vector formulas. This chapter corresponds to a journal paper, restructuring the previous results from Chapter 2 and Chapter 3, together with more elegant proofs. Additionally, the chapter also contains new complexity results for several modifications of the earlier logics, and proofs for the equal expressiveness of certain operators. Altogether, this part of our work provides the currently most complete overview on the complexity of common bit-vector logics. Chapter 4 comprises the final part of our earlier work that is dealing with theoretical properties of bit-vectors. Some new results will be presented in Chapter 12.

Chapter 5 [99]. As one of the results from Chapter 2, we know that the decision problem for quantifier-free bit-vector formulas is NEXPTIME-complete. Complexity theory, therefore, tells us that bit-blasting is exponential, in the general case.

However, we also know that polynomial translations to other NEXPTIME-hard problems exist. With DQBF being a prototypical NEXPTIME-complete problem, efficiently solving DQBF instances is an interesting topic on its own but, at the same time, might be useful to solve quantifier-free bit-vector formulas. In Chapter 5, we therefore present a so-called DQDPLL algorithm for DQBF, extending the successful DPLL [78] and QDPLL [61] architectures for SAT and QBF, respectively. At the point of writing the corresponding paper, no other DQBF solver existed and, therefore, our approach was also the first DQBF solver developed.

Chapter 6 [150]. Effectively Propositional Logic (EPR) is another well-known NEXPTIME-complete problem. In contrast to DQBF, efficient solvers for EPR already exist, with IPROVER [147] being the most notable one. While our results from Chapter 2 imply that a polynomial translation from quantifier-free bit-vector formulas to EPR has to exist, giving this kind of reduction is far from being trivial. In Chapter 6, we propose a possible translation and present its concrete implementation in our tool BV2EPR. Furthermore, we evaluate the performance of IPROVER on the resulting formulas. As expected, our results show that the exponential growth that comes with bit-blasting causes difficulties for state-of-the-art SMT solvers in connection with larger bit-widths. This confirms the relevance of the theoretical results from the previous chapters. While still being exponential in the general case, this effect is less dramatic when using IPROVER on the corresponding EPR representation, due to its lazy instantiation only considering relevant parts of a formula.

Chapter 7 [102]. We again address the challenge from Chapter 5, aiming to construct an efficient DQBF solver. Inspired by the efficiency of IPROVER on our benchmarks in Chapter 6 and also on certain QBF instances [202], we analyze the theoretical behaviour of IPROVER on the class of DQBF in Chapter 7. We use those insights to develop an instantion-based DQBF solver IDQ, explicitly tailored towards the structure of DQBF, which is more restricted compared to that of EPR. We also include an experimental evaluation, confirming the benefit of our techniques on DQBF instances, but sometimes even outperforming more specialized QBF solvers on QBF benchmarks. Aside from being efficient on practical benchmarks, IDQ also represents the first publicly available DQBF solver.

Chapter 8 [100]. While bit-blasting can be exponential, encoding a bit-vector formula into another NEXPTIME-complete logic cannot necessarily prevent the exponential blow-up that is inherent in that complexity class. Practically, we also witnessed this in Chapter 6. However, using our previous complexity results from Chapter 3, we know that certain restricted sets of bit-vector formulas actually are PSPACE-complete. While bit-blasting might still cause an exponential growth in formula size, we know that more efficient solving approaches must exist. For example, encoding a bit-vector formula into a model checking problem could yield this kind of algorithm. In Chapter 8, we present a direct translation from a PSPACE-complete class of bit-vector formulas to the SMV format [172]. We then run an

experimental evaluation, showing that model checkers on the SMV representation can outperform state-of-the-art SMT solvers on the original bit-vector representation by orders of magnitude on this kind of benchmarks.

Chapter 9 [152]. Generating challenging benchmarks is essential for improving any solver. As discussed in the beginning, the probably most straightforward way on improving bit-vector solvers is by improving the underlying SAT solvers. The SAT competition [12, 20] offers a possibility for state-of-the-art SAT solvers competing with each other on a huge variety of different benchmarks. In Chapter 9, we describe two sets of benchmarks which we obtained by bit-blasting certain bit-vector formulas. The first set of benchmarks was generated in order to verify the correctness of various reductions between different bit-vector operators, as described in Chapter 4. Due to the expressiveness of the given operators, this directly implies bit-blasting of the SMT formulas to be exponential, thus, generating arbitrary large and challenging benchmarks for SAT solvers. The second set of benchmarks are those from the restricted class of bit-vector formulas used in Chapter 8, being guaranteed to be in the PSPACE fragment but still producing exponentially large SAT formulas when bit-blasted. This allows SAT solvers to check for opportunities to leverage the inherent structure in this kind of benchmarks, possibly improving also on other benchmarks that require similar techniques.[2]

Chapter 10 [98]. With SLS being a very general concept, it cannot only be applied to SAT formulas, but also to more general problems. While SLS solvers for SAT usually perform very well on random and combinatorial formulas, their performance on applicational instances tends to be rather bad. This is often attributed to the fact that SLS solvers cannot make use of the structure that is contained in applicational instances. In Chapter 10, we therefore propose to use SLS directly on the theory level of bit-vector formulas, presenting the very first true SLS algorithm for SMT. We give a large experimental evaluation, showing that our approach outperforms SLS solvers for SAT by several orders of magnitude and, at the same time, represents an efficient approach of solving bit-vector formulas without bit-blasting or re-encoding.

Remaining Part. Starting with Chapter 11, the remainder of this work consists of three more chapters. Since all referenced papers were the result of joint work with other researchers, Chapter 11 will give details on the author's individual contribution to each publication. Furthermore, some of the previous results will be revisited in Chapter 12. In this context, we will also discuss additional related work, which did not exist at the time of publication of the individual papers, or which actually extended our research. We also propose different possibilities for future work for several topics that have been discussed in this work. Aside from this, Chapter 12 also contains very general new results, extending the complexity

[2] As a side effect, our benchmarks also revealed a bug in at least one of the solvers participating in the SAT competition 2013.

results from Chapters 2–4. Finally, a conclusion is given in Chapter 13.

1.3 Bugs

As in every other area of research, there is no bulletproof way to avoid bugs. A peer reviewed process, as formed the basis for all publications being part of this work, usually provides a high level of confidence in the correctness of the overall results—still, some less crucial details might sometimes be overlooked. All of the following chapters have been reworked in minor ways, compared to the original publication. Apart from formatting issues due to different LaTeX templates, this includes typos or grammatical mistakes, whenever realized by the author. Not affecting the content, none of those changes will be explicitly mentioned.

There were, however, two occasions where a bug actually affected minor parts of the content. Those bugs have been fixed in this work and will explicitly be addressed in the following. Note that all overall results still remain correct.

Chapter 8 [100]. When preparing a camera-ready version of [100], an off-by-one error occurred in the compile script, for reading some of the columns from the experimental results. This was due to adding another solver to the data base in the meantime. The bug was actually fixed before the corresponding workshop but, unfortunately, the wrong version can still be found on the web in addition to the correct one. Naturally, the correct plots were used in Chapter 8. Note that, as stated in [100], the complete results were also available on our webpage at all times [60].

Chapter 10 [98]. In the published version of [98], there was a bug in the example which we used to illustrate the behaviour of our algorithm. When evaluating possible moves in the constraint $x + 1 = y - 1$, we checked the neighbourhood of the evaluated terms $x + 1$ and $y - 1$. What our SLS algorithm would actually do, is checking the neighbourhood of x and y, leading to a different search path in order to find the solution. Since the resulting path would, however, be less illustrative, a different example is used for this thesis, in Chapter 10. Note that this does not affect the correctness of the proposed algorithm. Apart from this, a sign error occurred in [98], when specifying the score function for bit-vector expressions containing $\leqslant$. Instead of $t_2|_\alpha - t_1|_\alpha$, the score function of inequalities now correctly states $t_1|_\alpha - t_2|_\alpha$, in Chapter 10 of this thesis.

Aside from this, we are not aware of any other bugs in our existing work. Even though we hope that this is not the case, the reader should feel encouraged to let us know if one encountered further issues.

Chapter 2

On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width

Published. In Proceedings 10th International Workshop on Satisfiability Modulo Theories (SMT 2012), Affiliated to IJCAR 2012, EPiC Series, volume 20, pages 44–56, EasyChair 2013 [151].

Authors. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere.

Abstract. Bit-precise reasoning is important for many practical applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. From the theoretical point of view, only few results on the complexity of fixed-size bit-vector logics have been published. In this chapter, we show that some of these results only hold if unary encoding on the bit-width of bit-vectors is used. We then consider fixed-size bit-vector logics with binary encoded bit-width and establish new complexity results. Our proofs show that binary encoding adds more expressiveness to bit-vector logics, e.g., it makes fixed-size bit-vector logic even without uninterpreted functions nor quantification NEXPTIME-complete. We also show that, under certain restrictions, the increase of complexity when using binary encoding can be avoided.

2.1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Syntax and semantics of *fixed-size bit-vector logics* do not differ much in the literature [74, 19, 35, 96, 51]. Concrete formats for specifying bit-vector problems also exist, like the SMT-LIB format or the BTOR format [48].

Working with *non-fixed-size* bit-vectors has been considered, for instance, in [35, 6] and more recently in [208], but will not be further discussed in this chapter. Most industrial applications (and examples in the SMT-LIB) have fixed bit-width. We investigate the *complexity* of solving *fixed-size bit-vector formulas*. Some papers propose such complexity results, e.g., in [19], the authors consider quantifier-free bit-vector logic, and give an argument for NP-*hardness* of its satisfiability problem. In [51], a sublogic of the previous one is claimed to be NP-*complete*. In [234, 233], the *quantified* case is addressed, and the satisfiability of this logic with uninterpreted functions is proven to be NEXPTIME-*complete*. The proof holds only if we assume that the bit-widths of the bit-vectors in the input formula are written/encoded in *unary* form. We are not aware of any work that investigates how the particular encoding of the bit-widths in the input affects complexity (as an exception, see [71, Page 239, Footnote 3]). In practice, a more natural and exponentially more succinct *logarithmic* encoding is used, such as in the SMT-LIB, the BTOR, and the Z3 format. We investigate how complexity varies if we consider either a unary or a logarithmic, w.l.o.g., *binary encoding*.

In practice, state-of-the-art bit-vector solvers rely on rewriting and bit-blasting. The latter is defined as the process of translating a bit-vector (or word-level) description into a bit-level circuit, as in hardware synthesis. The result can then be checked by a (propositional) SAT solver. We give an example, why bit-blasting is not polynomial in general. Consider checking commutativity of bit-vector addition for two bit-vectors of size one million. Written to a file this formula in SMT2 syntax can be encoded with 138 bytes:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(assert (distinct (bvadd x y) (bvadd y x)))
```

Using Boolector [48] with rewriting optimizations switched off (except for structural hashing), bit-blasting produces a circuit of size 103 MB in AIGER format. Tseitin transformation results in a CNF in DIMACS format of size 1 GB. A bit-width of 10 million can be represented by two more bytes in the SMT2 input, but could not be bit-blasted anymore with our tool-flow (due to integer overflow). As this example shows, checking bit-vector logics through bit-blasting cannot be considered to be a polynomial reduction, which also disqualifies bit-blasting as a sound way to prove that the decision problem for (quantifier-free) bit-vector logics is in NP. We show that deciding bit-vector logics, even without quantifiers, is much harder: it is NEXPTIME-complete.

Informally speaking, we show that moving from unary to binary encoding for bit-widths increases complexity *exponentially*, and that binary encoding has at least as much expressive power as quantification. However, we give a sufficient condition for bit-vector problems to remain in the "lower" complexity class, when moving from unary to binary encoding. We call them *bit-width bounded* problems. For such problems, it does not matter whether bit-width is encoded unary or binary. We also discuss some concrete examples from SMT-LIB.

2.2 Preliminaries

We assume the common syntax for (fixed-size) *bit-vector formulas*, cf. SMT-LIB and [74, 19, 35, 96, 51, 48]. Every bit-vector possesses a bit-width n, either explicit or implicit, where n is a natural number, $n \geqslant 1$. We denote a bit-vector constant with $c^{[n]}$, where c is a natural number, $0 \leqslant c < 2^n$. A variable is denoted with $x^{[n]}$, where x is an identifier. Let us note that no explicit bit-width belongs to bit-vector operators, and, therefore, the bit-width of a compound term is *implicit*, i.e., can be calculated. Let $t^{[n]}$ denote the fact that the bit-vector term t is of bit-width n. We even omit an explicit bit-width if it can be deduced from the context.

In our proofs, we use the following *bit-vector operators*: *indexing* ($t^{[n]}[i]$, $0 \leqslant i < n$), *bitwise negation* ($\sim t^{[n]}$), *bitwise and* (${t_1}^{[n]} \& {t_2}^{[n]}$), *bitwise or* (${t_1}^{[n]} \mid {t_2}^{[n]}$), *shift left* (${t_1}^{[n]} \ll {t_2}^{[n]}$), *logical shift right* (${t_1}^{[n]} \gg_{\mathbf{u}} {t_2}^{[n]}$), *addition* (${t_1}^{[n]} + {t_2}^{[n]}$), *multiplication* (${t_1}^{[n]} \cdot {t_2}^{[n]}$), *unsigned division* (${t_1}^{[n]} /_{\mathbf{u}} {t_2}^{[n]}$), and *equality* (${t_1}^{[n]} = {t_2}^{[n]}$). Including other common operations (e.g., slicing, concatenation, extensions, arithmetic right shift, signed arithmetic and relational operators, rotations, etc.) does not destroy the validity of our subsequent propositions, since they all can be bit-blasted polynomially in the bit-width of their operands. *Uninterpreted functions* will also be considered. They have an explicit bit-width for the result type. The application of such a function is written as $f^{[n]}(t_1, \ldots, t_m)$, where f is an identifier, and ${t_1}^{[n_1]}, \ldots, {t_m}^{[n_m]}$ are terms.

Let QF_BV1 and QF_BV2 denote the logics of quantifier-free bit-vectors with unary and binary encoded bit-width, respectively (both without uninterpreted functions). As mentioned before, we prove that the complexity of deciding QF_BV2 is exponentially higher than deciding QF_BV1. This fact is, of course, due to the more succinct encoding. The logics we get by adding *uninterpreted functions* to these logics are denoted by QF_UFBV1 and QF_UFBV2. Uninterpreted functions are powerful tools for abstraction and, e.g., can be used to formalize reads on arrays. When *quantification* is introduced, we get the logics BV1 and BV2, if uninterpreted functions are prohibited. If they are allowed, we get UFBV1 and UFBV2. These latter logics are expressive enough, for instance, to formalize reads and writes on arrays with quantified indices.[1]

2.3 Complexity

In this section, we discuss the complexity of deciding the bit-vector logics defined so far. We first summarize our results and then give more detailed proofs for the new non-trivial ones. The results are also summarized in a tabular form in Appendix 2.6.2.

First, consider *unary encoding* of bit-widths. Without uninterpreted functions nor quantification, i.e., for QF_BV1, the following complexity result can be shown

[1] Let us emphasize again that among all these logics the ones with binary encoding correspond to the logics QF_BV, QF_UFBV, BV, and UFBV used by the SMT community, e.g., in SMT-LIB.

(for partial results and related work see also [19] and [51]):

Proposition 2.1. Q*F_BV*1 *is* NP-*complete*[2]

Proof. By bit-blasting, QF_BV1 can be polynomially reduced to *Boolean formulas*, for which the satisfiability problem (SAT) is NP-complete. The other direction follows from the fact that Boolean formulas are actually QF_BV1 formulas with terms of bit-width 1. □

Adding uninterpreted functions to QF_BV1 does not increase complexity:

Proposition 2.2. Q*F_UFBV*1 *is* NP-*complete.*

Proof. In a formula, uninterpreted functions can be eliminated by replacing each occurrence with a new bit-vector variable and adding (at most quadratic many) Ackermann constraints (e.g., [158, Chapter 3.3.1]). Therefore, QF_UFBV1 can be polynomially translated to QF_BV1. The other direction directly follows from the fact that QF_BV1 $\subset$ QF_UFBV1. □

Adding quantifiers to QF_BV1 yields the following complexity (see also [71]):

Proposition 2.3. B*V*1 *is* PSPACE-*complete.*

Proof. By bit-blasting, BV1 can be polynomially reduced to *Quantified Boolean Formulas* (QBF), which is PSPACE-complete. The other direction directly follows from the fact that QBF $\subset$ BV1 (following the same argument as in Prop. 2.1). □

Adding quantifiers to QF_UFBV1 increases complexity exponentially:

Proposition 2.4 (see [233])**.** U*FBV*1 *is* NEXPTIME-*complete.*

Proof. Effectively Propositional Logic (EPR), being NEXPTIME-complete, can be polynomially reduced to UFBV1 [233, Theorem 7]. For completing the other direction, apply the reduction in [233, Theorem 7] combined with the bit-blasting of the bit-vector operations. □

Our main contribution is to give complexity results for the more common logarithmic, w.l.o.g., *binary encoding*. Even without uninterpreted functions nor quantification, i.e., for QF_BV2, we obtain the same complexity as for UFBV1.

Proposition 2.5. Q*F_BV*2 *is* NEXPTIME-*complete.*

Proof. It is obvious that QF_BV2 $\in$ NEXPTIME, since a QF_BV2 formula can be translated exponentially to QF_BV1 $\in$ NP (Prop. 2.1), by a simple unary re-encoding of all bit-widths. The proof that QF_BV2 is NEXPTIME-hard is more complex and is given in Section 2.3.1. □

Adding uninterpreted functions to QF_BV2 does not increase complexity, again

[2] This kind of result is often called unary NP-completeness [105].

using Ackermann constraints, as in the proof for Prop. 2.2:

Proposition 2.6. Q*F_UFBV*2 *is* NEXPTIME*-complete.*

However, adding quantifiers to QF_UFBV2 increases complexity exponentially:

Proposition 2.7. U*FBV*2 *is* 2*-NExpTime-complete.*

Proof. Similar to the proof of Prop. 2.5, a UFBV2 formula can be exponentially translated to UFBV1 $\in$ NEXPTIME (Prop. 2.4), simply by re-encoding all the bit-widths to unary. It is more difficult to prove that UFBV2 is 2-NExpTime-hard, which we show in Section 2.3.2. □

Note that deciding QF_BV2 has the same complexity as UFBV1. Thus, starting with QF_BV1, re-encoding bit-widths to binary gives the same expressive power, in a precise complexity theoretical sense, as introducing both, uninterpreted functions and quantification. Thus, it is important to differentiate between unary and binary encoding of bit-widths in bit-vector logics. Our results show that binary encoding is at least as expressive as quantification, while only the latter has been considered in [234, 233].

2.3.1 QF_BV2 is NEXPTIME-hard

In order to prove that QF_BV2 is NEXPTIME-hard, we pick a NEXPTIME-hard problem and, then, we reduce it to QF_BV2. Let us choose the satisfiability problem of *Dependency Quantified Boolean Formulas* (DQBF), which has been shown to be NEXPTIME-complete [7].

In DQBF, quantifiers are not forced to be totally ordered. Instead, a partial order is explicitly expressed in the form $e(u_1, \ldots, u_m)$, stating that an existential variable e depends on the universal variables $u_1, \ldots, u_m$, where $m \geqslant 0$. Given an existential variable e, we will use $Deps(e)$ to denote the set of universal variables that e depends on. A more formal definition can be found in [7]. W.l.o.g., we can assume that a DQBF instance is in conjunctive normal form.

In the proof, we are going to apply bitmasks of the form

$$\overbrace{\underbrace{0\ldots0}_{2^i}\underbrace{1\ldots1}_{2^i}\ldots\underbrace{0\ldots0}_{2^i}\underbrace{1\ldots1}_{2^i}}^{2^n}$$

Given $n \geqslant 1$ and i, with $0 \leqslant i < n$, we denote such a bitmask with M_i^n. Note that these bitmasks correspond to the *binary magic numbers* [97] (see also [231, Chapter 7]), and, thus, can be arithmetically calculated in the following way (actually as sum of a geometric series):

$$M_i^n := \frac{2^{(2^n)} - 1}{2^{(2^i)} + 1}$$

In order to reformulate this definition in terms of bit-vectors, the numerator can be written as $\sim 0^{[2^n]}$, and $2^{(2^i)}$ as $1 \ll (1 \ll i)$, which results in the following bit-vector expression:

$$M_i^n \quad := \quad \sim 0^{[2^n]} \,/_{\mathbf{u}}\, \big((1 \ll (1 \ll i)) + 1\big) \tag{2.1}$$

Theorem 2.8. D*QBF can be (polynomially) reduced to* Q*F_BV*2*.*

Proof. The basic idea is to use bit-vector logic to encode function tables in an exponentially more succinct way, which then allows to characterize independence of an existential variable from a particular universal variable polynomially.

More precisely, we will use binary magic numbers, as constructed in Equation (2.1), to create a certain set of fully-specified exponential-size bit-vectors by using a polynomial expression, due to binary encoding. We will then formally point out the well-known fact that those bit-vectors correspond exactly to the set of *all assignments*. We can then use a polynomial-size bit-vector formula for *cofactoring Skolem-functions* in order to express independency constraints.

First, we describe the reduction (cf. an example in Appendix 2.6.1), then show that the reduction is polynomial, and, finally, that it is correct.

The reduction. Let $\phi := Q.m$ be a DQBF, consisting of a quantifier prefix Q and a Boolean CNF formula m called the *matrix* of ϕ. Let $u_0, \ldots, u_{k-1}$ denote all the universal variables that occur in ϕ. Translate ϕ to a QF_BV2 formula Φ by eliminating the quantifier prefix and translating the matrix as follows:

Step 1. Replace Boolean constants 0 and 1 with $0^{[2^k]}$ and $\sim 0^{[2^k]}$, respectively, and replace logical connectives with corresponding bitwise bit-vector operators ($\vee$, $\wedge$, $\neg$, with $|$, $\&$, $\sim$, respectively). Let Φ' denote the formula generated so far. Extend it to the formula $\Big(\Phi' = \sim 0^{[2^k]}\Big)$.

Step 2. For each u_i,

1. translate (all the occurrences of) u_i to a new bit-vector variable ${U_i}^{[2^k]}$;
2. in order to assign the appropriate bitmask of Equation (2.1) to U_i, add the following equation (i.e., conjunct it with the current formula):

$$U_i \quad = \quad M_i^k \tag{2.2}$$

For an optimization, see Remark 2.9 further down.

Step 3. For each existential variable e depending on universal variables $Deps(e)$, with $Deps(e) \subseteq \{u_0, \ldots, u_{k-1}\}$,

1. translate (all the occurrences of) e to a new bit-vector variable $E^{[2^k]}$;
2. for each $u_i \notin Deps(e)$, add the following equation:

$$(E \& U_i) \quad = \quad \Big(\big(E \gg_{\mathbf{u}} (1 \ll i)\big) \& U_i\Big) \tag{2.3}$$

As it is going to be detailed in the rest of the proof, the latter equations enforce the corresponding bits of $E^{[2^k]}$ to satisfy the dependency scheme of ϕ. More precisely, Equation (2.3) makes sure that the positive and negative cofactors of the Skolem-function, representing e with respect to an independent variable u_i, have the same value.

Polynomiality. Let us recall that all the bit-widths are encoded *binary* in the formula Φ. Thus, exponential bit-widths (2^k) are encoded into linear many (k) bits. We now show that each reduction step results in polynomial growth of the formula size.

Step 1 may introduce additional bit-vector constants to the formula. Their bit-width is 2^k, therefore, the resulting formula is bounded quadratically in the input size. *Step 2* adds k variables $U_i^{[2^k]}$ for the original universal variables, as well as k equations as restrictions. The bit-widths of added variables and constants is 2^k. Thus, the size of the added constraints is bounded quadratically in the input size. *Step 3* adds one bit-vector variable $E^{[2^k]}$ and at most k constraints for each existential variable. Thus, the size is bounded cubically in the input size.

Correctness. We show that the original formula ϕ and the result Φ of the translation are equisatisfiable. Consider one bit-vector variable U_i introduced in Step 2. In the following, we formalize the well-known fact that all the U_is correspond exactly to *all assignments*. By construction, all bits of U_i are fixed to some constant value. Additionally, for every bit-vector index $b_m \in [0, 2^k - 1]$ there exists a bit-vector index $b_n \in [0, 2^k - 1]$ such that

$$U_i[b_m] \neq U_i[b_n] \text{ and} \tag{2.4a}$$
$$U_j[b_m] = U_j[b_n],\ \forall j \neq i. \tag{2.4b}$$

Actually, let us define b_n in the following way (considering the 0th bit the least significant):

$$b_n \quad := \quad \begin{cases} b_m - 2^i & \text{if } U_i[b_m] = 0 \\ b_m + 2^i & \text{if } U_i[b_m] = 1 \end{cases}$$

By defining b_n this way, Equation (2.4a) and (2.4b) both hold, which can be seen as follows. Let $R(c, l)$ be the bit-vector of length l with each bit set to the Boolean constant c. Equation (2.4a) holds, since, due to construction, U_i consists of 2^{k-1-i} concatenated bit-vector fragments $0 \ldots 01 \ldots 1 = R(0, 2^i)R(1, 2^i)$ (with both 2^i zeros and 2^i ones). Therefore, it is easy to see that $U_i[b_m] \neq U_i[b_m - 2^i]$ holds if $U_i[b_m] = 0$, and $U_i[b_m] \neq U_i[b_m + 2^i]$ holds if $U_i[b_m] = 1$. With a similar argument, we can show that Equation (2.4b) holds: $U_j[b_m] = U_j[b_m - 2^i]$ ($U_j[b_m] = U_j[b_m + 2^i]$) if $U_j[b_m] = 0$ ($U_j[b_m] = 1$), since $b_m - 2^i$ ($b_m + 2^i$) is located either still in the same half or already in a concatenated copy of a $R(0, 2^j)R(1, 2^j)$ fragment, if $j \neq i$.

Now consider all possible assignments to the universal variables of our original DQBF-formula ϕ. For a given assignment $\alpha \in \{0,1\}^k$, the existence of such a previously defined b_n for every U_i and b_m allows us to iteratively find a b_α such that $(U_0[b_\alpha], \ldots, U_{k-1}[b_\alpha]) = \alpha$. Thus, we have a *bijective mapping* of every *universal assignment* α in ϕ to a *bit-vector index* b_α in Φ.

In Step 3, we first replace each existential variable e with a new bit-vector variable E, which can take $2^{(2^k)}$ different values. The value of each individual bit $E[b_\alpha]$ corresponds to the value e takes under a given assignment $\alpha \in \{0,1\}^k$ to the universal variables. Note that, with no further restrictions, there is no connection between the different bits in E and, therefore, the vector represents an arbitrary Skolem-function for an existential variable e. It may have different values for all universal assignments and thus would allow e to depend on all universal variables.

If, however, e does not depend on a universal variable u_i, we add the constraint of Equation (2.3). In DQBF, independence can be formalized in the following way: e does not depend on u_i if e has to take the same value in the case of all pairs of universal assignments $\alpha, \beta \in \{0,1\}^k$ where $\alpha[j] = \beta[j]$ for all $j \neq i$. Exactly this is enforced by our constraint. We have already shown that for α we have a corresponding bit-vector index b_α, and we have defined how we can construct a bit-vector index b_β for β. Our constraint for independence ensures that $E[b_\alpha] = E[b_\beta]$.

Step 1 ensures that all logical connectives and all Boolean constants are consistent for each bit-vector index, i.e., for each universal assignment, and that the matrix of ϕ evaluates to 1 for each universal assignment. □

Remark 2.9. Using Equation (2.1) as part of Equation (2.2) seems to require the use of *division*, which, however, can easily be eliminated by rewriting Equation (2.2) to

$$\Big(U_i \cdot \big((1 \ll (1 \ll i)) + 1\big)\Big) \quad = \quad \sim 0^{[2^k]}$$

Multiplication in this equation can then be eliminated by rewriting it as follows:

$$\Big((U_i \ll (1 \ll i)) + U_i\Big) \quad = \quad \sim 0^{[2^k]}$$

2.3.2 UFBV2 is 2-NExpTime-hard

In order to prove that UFBV2 is 2-NExpTime-hard, we pick a 2-NExpTime-hard problem and then, we reduce it to UFBV2. We can find such a problem among the so-called *domino tiling* problems [65]. Let us first define what a domino system is, and then we specify a 2-NExpTime-hard problem on such systems.

Definition 2.10 (Domino System)**.** A domino system is a tuple $\langle T, H, V, n\rangle$, where

- T is a finite set of *tile types*, in our case, $T = [0, k-1]$, where $k \geqslant 1$;
- $H, V \subseteq T \times T$ are the horizontal and vertical matching conditions, respectively;

- $n \geqslant 1$, encoded *unary*.

Let us note that the above definition differs (but not substantially) from the classical one in [65], in the sense that we use sub-sequential natural numbers for identifying tiles, as it is common in recent papers. Similar to [171] and [183], the size factor n, encoded *unary*, is part of the input. However, while a start tile α and a terminal tile ω are usually used, in our case, w.l.o.g., the starting tile is denoted by 0 and the terminal tile by $k-1$.

There are different domino tiling problems examined in the literature. In [65], a classical tiling problem is introduced, namely the *square tiling problem*, which can be defined as follows.

Definition 2.11 (Square Tiling)**.** Given a domino system $\langle T, H, V, n\rangle$, an $f(n)$-square tiling is a mapping $\lambda : [0, f(n)-1] \times [0, f(n)-1] \mapsto T$, such that

- the first row starts with the start tile: $\lambda(0,0) = 0$
- the last row ends with the terminal tile: $\lambda(f(n)-1, f(n)-1) = k-1$
- all horizontal matching conditions hold:
$$\big(\lambda(i,j), \lambda(i,j+1)\big) \in H,\ \forall i,j\ .\ 0 \leqslant i < f(n),\ 0 \leqslant j < f(n)-1$$
- all vertical matching conditions hold:
$$\big(\lambda(i,j), \lambda(i+1,j)\big) \in V,\ \forall i,j\ .\ 0 \leqslant i < f(n)-1,\ 0 \leqslant j < f(n)$$

In [65], a general theorem on the complexity of domino tiling problems is proved:

Theorem 2.12 (from [65])**.** *The $f(n)$-square tiling problem is* NTIME$\,(f(n))$*-complete.*

Since, for completing our proof on UFBV2, we need a 2-NExpTime-hard problem, let us emphasize the following easy corollary:

Corollary 2.13. *The $2^{(2^n)}$-square tiling problem is 2-NExpTime-complete.*

Theorem 2.14. *The $2^{(2^n)}$-square tiling problem can be (polynomially) reduced to* U*FBV*2*.*

Proof. Given a domino system $\langle T = [0, k-1], H, V, n\rangle$, let us introduce the following notations which we intend to use in the resulting UFBV2 formula.

- Represent each tile in T with the corresponding bit-vector of bit-width l, with $l := \lceil \log k \rceil$.
- Represent the horizontal and vertical matching conditions with the uninterpreted functions (predicates) $h^{[1]}(t_1{}^{[l]}, t_2{}^{[l]})$ and $v^{[1]}(t_1{}^{[l]}, t_2{}^{[l]})$, respectively.

- Represent the tiling with an uninterpreted function $\lambda^{[l]}(i^{[2^n]}, j^{[2^n]})$. Intuitively, λ represents the type of the tile in the cell at the row index i and column index j. Note that the bit-width of i and j is exponential in the size of the domino system, but due to *binary encoding* it can represented polynomially.

The resulting UFBV2 formula is the following:

$$\begin{aligned}
&\lambda(0,0) = 0 \quad \wedge \quad \lambda\Big(2^{(2^n)} - 1, 2^{(2^n)} - 1\Big) = k - 1 \\
&\wedge \bigwedge_{(t_1,t_2)\in H} h(t_1,t_2) \quad \wedge \bigwedge_{(t_1,t_2)\in V} v(t_1,t_2) \\
&\wedge \quad \forall i,j \left(\begin{array}{c} \Big(\; j < 2^{(2^n)} - 1 \;\Rightarrow\; h\big(\lambda(i,j), \lambda(i,j+1)\big) \; \Big) \\ \wedge \\ \Big(\; i < 2^{(2^n)} - 1 \;\Rightarrow\; v\big(\lambda(i,j), \lambda(i+1,j)\big) \; \Big) \end{array} \right)
\end{aligned}$$

This formula contains four kinds of constants. Three can be encoded directly ($0^{[2^n]}$, $0^{[l]}$, and $(k-1)^{[l]}$). However, the constant $2^{(2^n)} - 1$ has to be treated in a special way, in order to avoid double exponential size. Instead, it is written in the following form: $\sim 0^{[2^n]}$. The size of the resulting formula, due to *binary encoding* of the bit-width, is polynomial in the size of the domino system. □

2.4 Problems Bounded in Bit-Width

We are going to introduce a sufficient condition for bit-vector problems to remain in the "lower" complexity class, when re-encoding bit-width from unary to binary. This condition tries to capture the bounded nature of bit-width in certain bit-vector problems.

In any bit-vector formula, there has to be at least one term with explicit specification of its bit-width. In the logics we are dealing with, only a variable, a constant, or an uninterpreted function can have *explicit bit-width*. Given a bit-vector formula Φ, let us denote the *maximal explicit bit-width* in Φ with $max_{\mathsf{bw}}(\Phi)$. Furthermore, let $cnt_{\mathsf{bw}}(\Phi)$ denote *the number of terms with explicit bit-width* in Φ.

Definition 2.15 (Bit-Width Bounded Formula Set)**.** An infinite set S of bit-vector formulas is *(polynomially) bit-width bounded*, if there exists a polynomial function $p : \mathbb{N} \mapsto \mathbb{N}$ such that $\forall \Phi \in S.\ max_{\mathsf{bw}}(\Phi) \leqslant p(cnt_{\mathsf{bw}}(\Phi))$.

Proposition 2.16. *Given a bit-width bounded set S of formulas with binary encoded bit-width, any $\Phi \in S$ grows polynomially when re-encoding the bit-widths to unary.*

Proof. Let Φ' denote the formula obtained through re-encoding bit-widths in Φ to *unary*. For the size of Φ' the following upper bound can be shown:

$$|\Phi'| \leqslant cnt_{\mathsf{bw}}(\Phi) \cdot max_{\mathsf{bw}}(\Phi) + c.$$

Note that $cnt_{\mathsf{bw}}(\Phi) \cdot max_{\mathsf{bw}}(\Phi)$ is an upper bound on the sum over the sizes of all the terms with *explicit bit-width* in Φ'. The constant c represents the size of the rest of the formula. Since S is *bit-width bounded*, it holds that

$$|\Phi'| \leqslant cnt_{\mathsf{bw}}(\Phi) \cdot max_{\mathsf{bw}}(\Phi) + c \leqslant cnt_{\mathsf{bw}}(\Phi) \cdot p(cnt_{\mathsf{bw}}(\Phi)) + c \leqslant |\Phi| \cdot p(|\Phi|) + c,$$

where p is a polynomial function. Therefore, the size of Φ' is *polynomial* in the size of Φ. □

By applying this proposition to the logics of Section 2.2 we get:

Corollary 2.17. *Let us assume a bit-width bounded set S of bit-vector formulas. If $S \subseteq$ QF_UFBV2 (and even if $S \subseteq$ QF_BV2), then $S \in$ NP. If $S \subseteq$ BV2, then $S \in$ PSPACE. If $S \subseteq$ UFBV2, then $S \in$ NEXPTIME.*

2.4.1 Benchmark Problems

In this section, we discuss concrete SMT-LIB benchmark problems, and whether they are bit-width bounded. Since in SMT-LIB bit-widths are encoded logarithmically and quantification on bit-vectors is not (yet) addressed, we have picked benchmarks from `QF_BV`, which can be considered as QF_BV2 formulas.

Consider the benchmark set `QF_BV/brummayerbiere2/umulov2bw`b, representing instances of an *unsigned multiplication overflow detection* equivalence checking problem, that are parameterized by the bit-width of unsigned multiplicands (b). We show that the set of these benchmarks, with $b \in \mathbb{N}$, is *bit-width bounded*, and therefore is in NP. This problem checks that a certain (unsigned) overflow detection unit, defined in [201], gives the same result as the following condition: if the $b/2$ most significant bits of the multiplicands are zero, then no overflow occurs. It requires $2 \cdot (b-2)$ variables and a fixed number of constants to formalize the overflow detection unit, as detailed in [201]. The rest of the formula contains only a fixed number of variables and constants. The maximal bit-width in the formula is b. Therefore, the (maximal explicit) bit-width is linearly bounded in the number of variables and constants.

The benchmark class `QF_BV/brummayerbiere3/mulhs`b represents instances of computing the *high-order half of product* problem, parameterized by the bit-width of unsigned multiplicands (b). In this problem, the high-order $b/2$ bits of the product are computed, following an algorithm detailed in [231, Page 132]. The maximal bit-width is b and the number of variables and constants to formalize this problem is fixed, i.e., independent of b. Therefore, the (maximal explicit) bit-width is *not bounded* in the number of variables and constants.

The family `QF_BV/bruttomesso/lfsr/lfsr`t_b_n formalizes the behaviour of a *linear feedback shift register* [51]. Since, by construction, the bit-width (b) and the number (n) of registers do not correlate, and only n variables are used, this benchmark problem is *not bit-width bounded*.

2.5 Conclusion

We discussed complexity of deciding various quantified and quantifier-free fixed-size bit-vector logics. In contrast to existing literature, where usually it is not distinguished between unary or binary encoding of the bit-width, we argued that it is important to make this distinction. Our new results apply to the much more natural binary encoding, as it is also used in standard formats, e.g. in the SMT-LIB format. We proved that deciding QF_BV2 is NEXPTIME-complete, which is the same complexity as for deciding UFBV1. This shows that binary encoding for bit-widths has at least as much expressive power as quantification does. We also proved that UFBV2 is 2-NExpTime-complete. The complexity of deciding BV2 remains unclear. While it is easy to show EXPSPACE-inclusion for BV2 by bit-blasting to an exponential-size QBF, and NEXPTIME-hardness follows directly from QF_BV2 $\subset$ BV2, it is not clear whether BV2 is complete for any of these classes. We also showed that, under certain conditions on bit-width, the increase of complexity that comes with a binary encoding can be avoided. Finally, we gave examples of benchmark problems that do or do not fulfill this condition. As future work, it might be interesting to consider our results in the context of parametrized complexity [85]. Our theoretical results give an argument for using more powerful solving techniques. Currently, the most common approach used in state-of-the-art SMT solvers for bit-vectors is based on simple rewriting, bit-blasting, and SAT solving. We have shown this can possibly produce exponentially larger formulas when a logarithmic encoding is used as an input. Possible candidates are techniques used in EPR and/or (D)QBF solvers (see e.g. [99, 147]).

2.6 Appendix

2.6.1 Example: A Reduction of DQBF to QF_BV2

Consider the following DQBF:

$$\begin{aligned}\forall u_0, u_1, u_2 \exists x(u_0), y(u_1, u_2) \,.\, & (x \vee y \vee \neg u_0 \vee \neg u_1) \wedge \\ & (x \vee \neg y \vee u_0 \vee \neg u_1 \vee \neg u_2) \wedge \\ & (x \vee \neg y \vee \neg u_0 \vee \neg u_1 \vee u_2) \wedge \\ & (\neg x \vee y \vee \neg u_0 \vee \neg u_2) \wedge \\ & (\neg x \vee \neg y \vee u_0 \vee u_1 \vee \neg u_2)\end{aligned}$$

This DQBF is *unsatisfiable*. Let us note that by adding one more dependency for y, or even by making x and y dependent on all u_is, the resulting QBF becomes satisfiable.

Using the reduction in Section 2.3.1, this formula is translated to the following

QF_BV2 formula:

$$\Big((X \mid Y \mid\!\sim U_0 \mid\!\sim U_1)\&(X \mid\!\sim Y \mid U_0 \mid\!\sim U_1 \mid\!\sim U_2)\&(X \mid\!\sim Y \mid\!\sim U_0 \mid\!\sim U_1 \mid U_2)$$
$$\&(\sim X \mid Y \mid\!\sim U_0 \mid\!\sim U_2)\&(\sim X \mid\!\sim Y \mid U_0 \mid U_1 \mid\!\sim U_2)\Big) = \sim 0^{[8]} \;\wedge$$
$$\bigwedge_{i \in \{0,1,2\}} \Big(\big((U_i \ll (1 \ll i)) + U_i\big) \;=\; \sim 0^{[8]}\Big) \;\wedge$$
$$\begin{aligned}(X\&U_1) \;&=\; \big((X \gg_{\mathbf{u}} (1 \ll 1))\&U_1\big) \;\wedge\\ (X\&U_2) \;&=\; \big((X \gg_{\mathbf{u}} (1 \ll 2))\&U_2\big) \;\wedge\\ (Y\&U_0) \;&=\; \big((Y \gg_{\mathbf{u}} (1 \ll 0))\&U_0\big)\end{aligned} \tag{2.5}$$

Note that $M_0^3 = 55_{16}{}^{[8]} = 01010101_2{}^{[8]}$, $M_1^3 = 33_{16}{}^{[8]} = 00110011_2{}^{[8]}$, and $M_2^3 = 0\mathrm{F}_{16}{}^{[8]} = 00001111_2{}^{[8]}$, where "$\cdot_{16}$" and "$\cdot_2$" denotes hexadecimal and binary encoding of the binary magic numbers, respectively.

In the following, let us show that the formula (2.5) is also unsatisfiable. First, we show how the bits of X get restricted by the constraints introduced above. Let us denote the originally unrestricted bits of X with $x_7, x_6, \ldots, x_0$. Since the bit-vectors

$$(X\&U_1) \;=\; \big(0, 0, X[5], X[4], 0, 0, X[1], X[0]\big)$$

and

$$\big((X \gg_{\mathbf{u}} (1 \ll 1))\&U_1\big) \;=\; \big(0, 0, X[7], X[6], 0, 0, X[3], X[2]\big)$$

are forced to be equal, some bits of X should coincide, as follows:

$$X \;:=\; \big(x_5, x_4, x_5, x_4, x_1, x_0, x_1, x_0\big)$$

Furthermore, considering also the equation of

$$(X\&U_2) \;=\; \big(0, 0, 0, 0, X[3], X[2], X[1], X[0]\big)$$

and

$$\big((X \gg_{\mathbf{u}} (1 \ll 2))\&U_2\big) \;=\; \big(0, 0, 0, 0, X[7], X[6], X[5], X[4]\big)$$

results in

$$X \;:=\; \big(x_1, x_0, x_1, x_0, x_1, x_0, x_1, x_0\big)$$

In a similar fashion, the bits of Y are constrained as follows:

$$Y \;:=\; \big(y_6, y_6, y_4, y_4, y_2, y_2, y_0, y_0\big)$$

In order to show that the formula (2.5) is unsatisfiable, let us evaluate the "clauses" in the formula:

$$\begin{aligned}(X \mid Y \mid\!\sim U_0 \mid\!\sim U_1) \;&=\; \big(\;1\;,\;1\;,\;1\;,\; x_0 \vee y_4\;,\;1\;,\;1\;,\;1\;,\; x_0 \vee y_0\;\big)\\ (X \mid\!\sim Y \mid U_0 \mid\!\sim U_1 \mid\!\sim U_2) \;&=\; \big(\;1\;,\;1\;,\;1\;,\;1\;,\;1\;,\;1\;,\; x_1 \vee \neg y_0\;,\;1\;\big)\\ (X \mid\!\sim Y \mid\!\sim U_0 \mid\!\sim U_1 \mid U_2) \;&=\; \big(\;1\;,\;1\;,\;1\;,\; x_0 \vee \neg y_4\;,\;1\;,\;1\;,\;1\;,\;1\;\big)\\ (\sim X \mid Y \mid\!\sim U_0 \mid\!\sim U_2) \;&=\; \big(\;1\;,\;1\;,\;1\;,\;1\;,\;1\;,\; \neg x_0 \vee y_2\;,\;1\;,\; \neg x_0 \vee y_0\;\big)\\ (\sim X \mid\!\sim Y \mid U_0 \mid U_1 \mid\!\sim U_2) \;&=\; \big(\;1\;,\;1\;,\;1\;,\;1\;,\; \neg x_1 \vee \neg y_2\;,\;1\;,\;1\;,\;1\;\big)\end{aligned}$$

By applying *bitwise and* to them, we get the bit-vector represented by the formula (2.5):

$$\begin{pmatrix} 1 \\ 1 \\ 1 \\ (x_0 \vee \neg y_4) \wedge (x_0 \vee y_4) \\ \neg x_1 \vee \neg y_2 \\ \neg x_0 \vee y_2 \\ x_1 \vee \neg y_0 \\ (x_0 \vee y_0) \wedge (\neg x_0 \vee y_0) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ x_0 \\ \neg x_1 \vee \neg y_2 \\ \neg x_0 \vee y_2 \\ x_1 \vee \neg y_0 \\ y_0 \end{pmatrix}$$

In order to check if every bits of this bit-vector can evaluate to 1, it is sufficient to try to satisfy the set of the above (propositional) clauses. It is easy to see that this clause set is unsatisfiable, since by unit propagation x_1 and y_2 must be 1, which contradicts with the clause $\neg x_1 \vee \neg y_2$.

2.6.2 Table: Completeness Results for Bit-Vector Logics

		quantifiers			
		no		*yes*	
		uninterpreted functions		uninterpreted functions	
		no	*yes*	*no*	*yes*
encoding	*unary*	NP	NP	PSPACE	NEXPTIME
	binary	NEXPTIME	NEXPTIME	?	2-NEXPTIME

Chapter 3

More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding

Published. In Proceedings 8th International Computer Science Symposium in Russia (CSR 2013), Lecture Notes in Computer Science (LNCS) volume 7913, pages 378–390, Springer 2013 [101].

Authors. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere.

Abstract. Bit-precise reasoning is important for many practical applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. From the theoretical point of view, only few results on the complexity of fixed-size bit-vector logics have been published. Most of these results only hold if unary encoding on the bit-width of bit-vectors is used. In previous work [151], we showed that binary encoding adds more expressiveness to bit-vector logics, e.g. it makes fixed-size bit-vector logic without uninterpreted functions nor quantification NEXPTIME-complete. In this chapter, we look at the quantifier-free case again and propose two new results. While it is enough to consider logics with *bitwise operations*, *equality*, and *shift by constant* to derive NEXPTIME-completeness, we show that the logic becomes PSPACE-complete if, instead of *shift by constant*, only *shift by 1* is permitted, and even NP-complete if *no shifts* are allowed at all.

3.1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and

software verification. Examples of state-of-the-art SMT solvers with support for bit-precise reasoning are Boolector, MathSAT, STP, Z3, and Yices.

Syntax and semantics of *fixed-size bit-vector logics* do not differ much in the literature [74, 19, 35, 51, 96]. Concrete formats for specifying bit-vector problems also exist, e.g., the SMT-LIB format [18] or the BTOR format [48]. Working with *non-fixed-size* bit-vectors has been considered, for instance, in [35, 6], and more recently in [209], but is not the focus of this chapter. Most industrial applications (and examples in the SMT-LIB) have fixed bit-width.

We investigate the *complexity* of solving *fixed-size bit-vector formulas*. Some papers propose such complexity results. For example, in [19], the authors consider quantifier-free bit-vector logic and give an argument for the NP-*hardness* of its satisfiability problem. In [51], a sublogic of the previous one is claimed to be NP-*complete*. Interestingly, in [52], there is a claim about the full quantifier-free bit-vector logic without uninterpreted functions (QF_BV) being NP-complete, however, the proposed decision procedure confirms this claim only if the bit-widths of the bit-vectors in the input formula are written/encoded in *unary* form. In [234, 233], the *quantified* case is addressed, and the satisfiability problem of this logic with uninterpreted functions (UFBV) is proved to be NEXPTIME-*complete*. Again, the proof only holds if we assume unary encoded bit-widths. In practice, a more natural and exponentially more succinct *logarithmic* encoding is used, such as in the SMT-LIB, the BTOR, and the Z3 format.

In previous work [151], we already investigated how complexity varies if we consider either a unary or a logarithmic, w.l.o.g., *binary encoding*. Apart from this, we are not aware of any work that investigates how the particular encoding of the bit-widths in the input affects complexity (as an exception, see [71, Page 239, Footnote 3]). Table 3.1 summarizes the completeness results we obtained in [151].

		quantifiers			
		no		*yes*	
		uninterpreted functions		uninterpreted functions	
		no	*yes*	*no*	*yes*
encoding	*unary*	NP	NP	PSPACE	NEXPTIME
	binary	NEXPTIME	NEXPTIME	?	2-NEXPTIME

Table 3.1: Completeness results for various bit-vector logics and encodings [151].

In this chapter, we revisit QF_BV2, the quantifier-free case with binary encoding and without uninterpreted functions. We then put certain restrictions on the operations we use (in particular on the *shift* operation). As a result, we obtain two new sublogics, QF_BV2$_{\ll 1}$ and QF_BV2$_{bw}$, which we show to be PSPACE-complete and NP-complete, respectively.

3.2 Motivation

In practice, state-of-the-art bit-vector solvers rely on rewriting and bit-blasting. The latter is defined as the process of translating a bit-vector (or word-level) description into a bit-level circuit, as in hardware synthesis. The result can then be checked by a (propositional) SAT solver. In [151], we gave the following example (in SMT2 syntax) to point out that bit-blasting is not polynomial in general. It checks commutativity of adding two bit-vectors of bit-width 1000000:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(assert (distinct (bvadd x y) (bvadd y x)))
```

Bit-blasting such formulas generates huge circuits, which shows that checking bit-vector logics through bit-blasting cannot be considered to be a polynomial reduction. This also disqualifies bit-blasting as a sound way to argue that the decision problem for (quantifier-free) bit-vector logics is in NP. We actually proved in [151], that deciding bit-vector logics, even without quantifiers, is much harder. It turned out to be NEXPTIME-complete in the general case.

However, in [151] we then also defined a class of *bit-width bounded problems* and showed that under certain restrictions on the bit-widths this growth in complexity can be avoided and the problem remains in NP.

In this chapter, we give a more detailed classification of quantifier-free fixed-size bit-vector logics by investigating how complexity varies when we restrict the operations that can be used in a bit-vector formula. We establish two new complexity results for restricted bit-vector logics and bring together our previous results in [151] with work on linear arithmetic on non-fixed-size bit-vectors [209, 208] and work on the reduction of bit-widths [140, 141]. The formula in the given example only contains bitwise operations, equality, and addition. Solving this kind of formulas turns out to be PSPACE-complete.

3.3 Definitions

We assume the usual syntax for (quantifier-free) bit-vector logics, with a restricted set of bit-vector operations: bitwise operations, equality, and (left) shift by constant.

Definition 3.1 (Term)**.** A bit-vector term t of bit-width n ($n \in \mathbb{N}$, $n \geqslant 1$) is denoted by $t^{[n]}$. A term is defined inductively as follows:

	term	condition	bit-width
bit-vector constant:	$c^{[n]}$	$c \in \mathbb{N}, 0 \leqslant c < 2^n$	n
bit-vector variable:	$x^{[n]}$	x is an identifier	n

bitwise negation:	$\sim t^{[n]}$	$t^{[n]}$ is a term	n
bitwise and/or/xor: $\bullet \in \{\&, \mid, \oplus\}$	$\left(t_1{}^{[n]} \bullet t_2{}^{[n]}\right)$	$t_1{}^{[n]}$ and $t_2{}^{[n]}$ are terms	n
equality:	$\left(t_1{}^{[n]} = t_2{}^{[n]}\right)$	$t_1{}^{[n]}$ and $t_2{}^{[n]}$ are terms	1
shift by constant:	$\left(t^{[n]} \ll c^{[n]}\right)$	$t^{[n]}$ is a term, $c^{[n]}$ is a constant	n

We also define how to measure the size of bit-vector expressions:

Definition 3.2 (Size)**.** The size of a bit-vector term $t^{[n]}$ is denoted by $|t^{[n]}|$ and is defined inductively as follows:

	term	size
natural number:	$enc(n)$	$\lceil \log_2 (n+1) \rceil + 1$
bit-vector constant:	$\lvert c^{[n]} \rvert$	$enc(c) + enc(n)$
bit-vector variable:	$\lvert x^{[n]} \rvert$	$1 + enc(n)$
bitwise negation:	$\lvert \sim t^{[n]} \rvert$	$1 + \lvert t^{[n]} \rvert$
binary operations: $\bullet \in \{\&, \mid, \oplus, =, \ll\}$	$\lvert \left(t_1{}^{[n]} \bullet t_2{}^{[n]}\right) \rvert$	$1 + \lvert t_1{}^{[n]} \rvert + \lvert t_2{}^{[n]} \rvert$

A bit-vector term $t^{[1]}$ is also called a *bit-vector formula*. We say that a bit-vector formula is in *flat form* if it does not contain nested equalities. It is easy to see that any bit-vector formula can be translated to this form with only linear growth in the number of variables. In the rest of the chapter, we may omit parentheses in a formula for the sake of readability.

Let Φ be a bit-vector formula and α an assignment to the variables in Φ. We use the notation $\alpha(\Phi)$ to denote the evaluation of Φ under α, with $\alpha(\Phi) \in \{0, 1\}$. α satisfies Φ if and only if $\alpha(\Phi) = 1$. We define three different bit-vector logics:

- QF_BV2$_{\ll c}$:
  bitwise operations, equality, and shift by any constant are allowed
- QF_BV2$_{\ll 1}$:
  bitwise operations, equality, and shift by only $c = 1$ are allowed
- QF_BV2$_{bw}$:
  only bitwise operations and equality are allowed

Obviously, QF_BV2$_{bw}$ $\subseteq$ QF_BV2$_{\ll 1}$ $\subseteq$ QF_BV2$_{\ll c}$. In Section 3.4, we investigate the complexity of the satisfiability problem for these logics:

- QF_BV2$_{\ll c}$ is NEXPTIME-complete.

- QF_BV2$_{\ll 1}$ is PSPACE-complete.
- QF_BV2$_{bw}$ is NP-complete.

Adding uninterpreted functions does not change expressiveness of these logics, since in the quantifier-free case, uninterpreted functions can always be replaced by new variables. To guarantee functional consistency, Ackermann constraints have to be added to the formula. However, even in the worst case, the number of Ackermann constraints is only quadratic in the number of function instances. W.l.o.g., we therefore do not explicitly deal with uninterpreted functions.

3.4 Complexity Results

Theorem 3.3. Q*F_BV*2$_{\ll c}$ *is* NEXPTIME*-complete.*

Proof. The theorem directly follows from our previous work in [151]. We informally defined QF_BV2 as the quantifier-free bit-vector logic that uses the common bit-vector operations as defined for example in SMT-LIB, including bitwise operations, equality, shifts, addition, multiplication, concatenation, slicing, etc., and then showed that QF_BV2 is NEXPTIME-complete.

Obviously, QF_BV2$_{\ll c} \subseteq$ QF_BV2 and therefore, QF_BV2$_{\ll c} \in$ NEXPTIME. To show the NEXPTIME-hardness of QF_BV2, we gave a (polynomial) reduction from DQBF (which is NEXPTIME-complete [187]) to QF_BV2. Since we only used *bitwise operations*, *equality*, and *shift*[1] *by constant* in our reduction, we also immediately get the NEXPTIME-hardness of QF_BV2$_{\ll c}$. □

Theorem 3.4. Q*F_BV*2$_{\ll 1}$ *is* PSPACE*-complete.*

Proof. In Lemma 3.5, we give a (polynomial) reduction from QBF (which is PSPACE-complete) to QF_BV2$_{\ll 1}$. This shows PSPACE-hardness of QF_BV2$_{\ll 1}$. In Lemma 3.6, we then prove that QF_BV2$_{\ll 1} \in$ PSPACE by giving a translation from QF_BV2$_{\ll 1}$ to (polynomial sized) sequential circuits. As pointed out, e.g., in [194], the symbolic reachability problem is PSPACE-complete as well. □

Lemma 3.5. Q*BF can be (polynomially) reduced to* Q*F_BV*2$_{\ll 1}$.

Proof. To show the PSPACE-hardness of QF_BV2$_{\ll 1}$, we give a polynomial reduction from QBF similar to the one from DQBF to QF_BV2 that we proposed in [151]. For our reduction, we again use the so-called *binary magic numbers* (or magic masks in [145, page 141]). Appendix 3.7.2 demonstrates how the reduction works.

Given $m, n \in \mathbb{N}$ with $0 \leqslant m < n$, a binary magic number can be written in the

[1] Note, logical right shifts were used in the proof in [151]. However, by applying negated bit masks throughout the proof, all right shifts can be rewritten as left shifts.

following form:

$$\textit{binmagic}\,(2^m,\ 2^n) = \overbrace{\underbrace{0\ldots0}_{2^m}\,\underbrace{1\ldots1}_{2^m}\ldots\underbrace{0\ldots0}_{2^m}\,\underbrace{1\ldots1}_{2^m}}^{2^n}$$

Note that in [151], we used *shift by constant* to construct the binary magic numbers, as done in the literature [145]. This is not permitted in QF_BV2$_{\ll 1}$. Therefore, we give an alternative construction using only *bitwise operations*, *equality*, and *shift by 1*:

Given $n > 0$, for all m, $0 \leqslant m < n$, add the following equation to the formula:

$$b'_m{}^{[2^n]} = \left(\bigwedge_{0\leqslant i<m} b_i{}^{[2^n]}\right) \oplus b_m{}^{[2^n]}$$

Consider all the bit-vector variables $b_0{}^{[2^n]}, \ldots, b_{n-1}{}^{[2^n]}$ as column vectors in a matrix $B^{[2^n \times n]}$ and all the bit-vector variables $b'_0{}^{[2^n]}, \ldots, b'_{n-1}{}^{[2^n]}$ as column vectors in a matrix $B'^{[2^n \times n]}$. If each row of B is interpreted as a number c in binary representation, with $0 \leqslant c < 2^n$, the corresponding row of B' is equal to $c + 1$.

Now, again for all m, $0 \leqslant m < n$, add another constraint:

$$b'_m{}^{[2^n]} = b_m{}^{[2^n]} \ll 1^{[2^n]}$$

Together with the previous n equations, those n constraints force the rows of B to represent an enumeration of all binary numbers $0 \leqslant c < 2^n$. Therefore, the columns of B, i.e., the individual bit-vectors $b_0{}^{[2^n]}, \ldots, b_{n-1}{}^{[2^n]}$, exactly define the binary magic numbers: $\textit{binmagic}\,(2^m,\ 2^n) := b_m{}^{[2^n]}$.

Obviously, all b'_m, for $0 \leqslant m < n$, can be eliminated and the two sets of constraints can be replaced by a single set of constraints:

$$\left(\bigwedge_{0\leqslant i<m} b_i{}^{[2^n]}\right) \oplus b_m{}^{[2^n]} = b_m{}^{[2^n]} \ll 1^{[2^n]}$$

Now let $\phi := Q.M$ denote a QBF with quantifier prefix Q and matrix M. Since ϕ is a QBF (in contrast to DQBF in [151]), we know that Q defines a total order on the universal variables. We now assume the universal variables $u_0, \ldots, u_{n-1}$ of ϕ are ordered according to their appearance in Q, with u_0 and u_{n-1} being the innermost and outermost variable, respectively.

Translate ϕ to a QF_BV2$_{\ll 1}$ formula Φ by eliminating the quantifier prefix and translating the matrix as follows:

Step 1. Replace Boolean constants 0 and 1 with $0^{[2^n]}$ and $\sim 0^{[2^n]}$, respectively, and replace logical connectives with corresponding bitwise bit-vector operations (e.g., $\wedge$ with $\&$). Let Φ' denote the formula generated so far. Extend it to the formula $\left(\Phi' = \sim 0^{[2^k]}\right)$.

Step 2. For each universal variable $u_m \in \{u_0, \ldots, u_{n-1}\}$,

1. translate (all the occurrences of) u_m to a new bit-vector variable ${U_m}^{[2^n]}$;
2. in order to assign a binary magic number to ${U_m}^{[2^n]}$, add the following equation (i.e., conjunct it with the current formula):

$$ {U_m}^{[2^n]} \quad = \quad \mathit{binmagic}\,(2^m,\ 2^n) $$

Step 3. For an existential variable e, depending on $Deps(e) = \{u_m, \ldots, u_{n-1}\}$, with u_m being the innermost universal variable that e depends on,

1. translate (all the occurrences of) e to a new bit-vector variable $E^{[2^n]}$;
2. if $Deps(e) = \varnothing$ add the following equation:

$$ (E \& \sim 1) \quad = \quad (E \ll 1) \tag{3.1} $$

otherwise, if $m \neq 0$ add the two equations:

$$ U'_m \quad = \quad \sim\big((U_m \ll 1) \oplus U_m\big) \tag{3.2} $$

$$ (E \& U'_m) \quad = \quad \big((E \ll 1) \& U'_m\big) \tag{3.3} $$

Note that we omitted the bit-widths in the last equations to improve readability. Each bit position of Φ corresponds to the evaluation of ϕ under a specific assignment to the universal variables $u_0, \ldots, u_{n-1}$, and, by construction of the corresponding vectors ${U_0}^{[2^n]}, \ldots, {U_{n-1}}^{[2^n]}$, all possible assignments are considered. Equation (3.2) creates a bit-vector ${U'_m}^{[2^n]}$ for which each bit equals to 1 if and only if the corresponding universal variable changes its value from one universal assignment to the next.

Of course, Equation (3.2) does not have to be added multiple times, if several existential variables depend on the same universal variable. Equation (3.3) (or Equation (3.1)) ensures that the corresponding bits of $E^{[2^n]}$ satisfy the dependency scheme of ϕ by only allowing the value of e to change if an outer universal variable takes a different value. If $m = 0$, i.e., if e depends on all universal variables, Equation (3.2) evaluates to ${U'_0}^{[2^n]} = 0$, and, as a consequence, Equation (3.3) simplifies to $true$. Due to this, no constraints need to be added for $m = 0$. A similar approach used for translating QBF to Symbolic Model Verification (SMV) can be found in [84]. See also [194] for a translation from QBF to sequential circuits. □

Lemma 3.6. *QF_BV2$_{\ll 1}$ can be (polynomially) reduced to sequential circuits.*

Proof. In [209, 208], the authors give a translation from quantifier-free Presburger arithmetic with bitwise operations (QFPAbit) to sequential circuits. We can adopt

their approach in order to construct a translation for QF_BV2$_{\ll 1}$. The main difference between QFPAbit and QF_BV2$_{\ll 1}$ is the fact that bit-vectors of arbitrary, non-fixed, size are allowed in QFPAbit while all bit-vectors contained in QF_BV2$_{\ll 1}$ have a fixed bit-width.

Given $\Phi \in$ QF_BV2$_{\ll 1}$ in flat form. Let $x^{[n]}, y^{[n]}$ denote bit-vector variables, $c^{[n]}$ a bit-vector constant, and ${t_1}^{[n]}, {t_2}^{[n]}$ bit-vector terms only containing bit-vector variables and bitwise operations. Following [209, 208], we additionally assume, w.l.o.g., that Φ only consists of three types of expressions: ${t_1}^{[n]} = {t_2}^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \ll 1^{[n]}$, since any QF_BV2$_{\ll 1}$ formula can be written like this with only a linear growth in the number of original variables.

We encode each equality in Φ separately into an atomic sequential circuit. Compared to [209, 208], two modifications are needed. First, we need to give a translation for $x = y \ll 1$ to sequential circuits. This can be done, for example, by using the sequential circuit for $x = 2 \cdot y$ in QFPAbit. However, a direct translation can also easily be constructed.

The second modification relates to dealing with *fixed-size* bit-vectors. Let n be the bit-width of all bit-vectors in a given equality. We extend each atomic sequential circuit to include a counter (circuit). The counter initially is set to 0 and is incremented by 1 in each clock cycle up to a value of n.

When the counter reaches a value of n, it does not change anymore and the output of the atomic sequential circuit is set to the same value as the output in the previous cycle. A counter like this can be realized with $\lceil \log_2(n) \rceil$ gates, i.e. polynomially in the size of Φ. In contrast to the implementation described in [208], we assume that the input streams for all variables start with the least significant bit. However, as already pointed out by the authors in [208], their choice was arbitrary and it is no more complicated to construct the circuits the other way round.

Finally, after constructing atomic circuits, their outputs are combined by logical gates following the Boolean structure of Φ, in the same way as for unbounded bit-width in [209, 208]. Due to adding counters, we ensure that for every input stream x_i, only the first n_i bits of x_i influence the result of the whole circuit. □

For the proof of Theorem 3.9, we need the following definition and a corresponding lemma from [151]:

Definition 3.7 (Bit-Width Bounded Formula Set [151])**.** Given a formula Φ, we denote the *maximal bit-width* in Φ with $max_{\mathsf{bw}}(\Phi)$. An infinite set S of bit-vector formulas is *(polynomially) bit-width bounded*, if there exists a polynomial function $p : \mathbb{N} \mapsto \mathbb{N}$ such that $\forall \Phi \in S.\ max_{\mathsf{bw}}(\Phi) \leqslant p(|\Phi|)$.

Lemma 3.8 ([151])**.** *$S \in$ NP for any bit-width bounded formula set $S \subseteq$ QF_BV2.*

Theorem 3.9. QF_BV2_{bw} *is* NP*-complete.*

Proof. Since *Boolean Formulas* are a subset of QF_BV2$_{bw}$, NP-hardness follows directly. To show that QF_BV2$_{bw} \in$ NP, we give a reduction from QF_BV2$_{bw}$ to a *bit-width bounded* set of formulas. The claim then follows from Lemma 3.8.

Given a formula $\Phi \in$ QF_BV2$_{bw}$ in flat form. If Φ contains any constants $c^{[n]} \neq 0^{[n]}$, we remove those constants in a (polynomial) preprocessing step. Let ${c_{max}}^{[n]} = b_{k-1} \ldots b_1 b_0$ be the largest constant in Φ, denoted in binary representation with $b_{k-1} = 1$ and arbitrary bits $b_{k-2}, \ldots, b_0$. We now replace each equality ${t_1}^{[m]} = {t_2}^{[m]}$ in Φ with

$$({t_{1,k'-1}}^{[1]} = {t_{2,k'-1}}^{[1]}) \& \ldots \& ({t_{1,0}}^{[1]} = {t_{2,0}}^{[1]})$$

where $k' = min\{m, k\}$, and, if $m > k$, we additionally add

$$\&({t_{1,hi}}^{[m-k]} = {t_{2,hi}}^{[m-k]})$$

For $0 \leqslant i < k$, we use $({t_{1,i}}^{[1]} = {t_{2,i}}^{[1]})$ to express the *i*th row of the original equality. All occurrences of a variable $x^{[m]}$ are replaced with a new variable ${x_i}^{[1]}$. All occurrences of a constant $c^{[m]}$ are replaced with $0^{[1]}$ if the *i*th bit of the constant is 0, and by $\sim 0^{[1]}$ otherwise.

In a similar way, if $m > k$, $({t_{1,hi}}^{[m-k]} = {t_{2,hi}}^{[m-k]})$ represents the remaining $(m-k)$ rows of the original equality corresponding to the most significant bits. All occurrences of a variable $x^{[m]}$ are replaced with a new variable ${x_{hi}}^{[m-k]}$ and all occurrences of a constant $c^{[m]}$ are replaced with $0^{[m-k]}$. Since this preprocessing step is logarithmic in the value of c_{max}, it is polynomial in $|\Phi|$. W.l.o.g., we now assume that Φ does not contain any bit-vector constants different from $0^{[n]}$.

We now construct a formula Φ' by reducing the bit-widths of all bit-vector terms in Φ. Each term $t^{[n]}$ in Φ, with bit-width n, is replaced with a term $t^{[n']}$, with $n' := min\{n, |\Phi|\}$. Apart from this, Φ' is exactly the same as Φ. As a consequence, $max_{\mathsf{bw}}(\Phi') \leqslant |\Phi|$. The set of formulas constructed in this way is bit-width bounded according to Definition 3.7. To complete our proof, we now have to show that the proposed reduction is sound, i.e., out of every satisfying assignment to the bit-vector variables ${x_1}^{[n_1]}, \ldots, {x_k}^{[n_k]}$ for Φ, we can also construct a satisfying assignment to ${x_1}^{[n'_1]}, \ldots, {x_k}^{[n'_k]}$ for Φ' and vice versa.

It is easy to see that whenever we have a satisfying assignment α' for Φ', we can construct a satisfying assignment α for Φ. This can be done by simply setting all additional bits of all bit-vector variables to the same value as the most significant bit of the corresponding original vector, i.e., by performing a signed extension. Since all equalities still evaluate to the same value under the extended assignment, $\alpha(F) = \alpha'(F')$ for all equalities F and F' of Φ and Φ'), respectively. As a direct consequence, $\alpha(\Phi) = \alpha'(\Phi) = 1$. The other direction needs slightly more reasoning. Given α, with $\alpha(\Phi) = 1$, we need to construct α', with $\alpha'(\Phi') = 1$. Again, we want to ensure that $\alpha'(F') = \alpha(F)$ for all equalities F and F' in Φ and Φ'. respectively.

In each variable ${x_i}^{[n_i]}$, $i \in \{1, \ldots, k\}$, we are going to select some of the bits. For each equality F with $\alpha(F) = 0$, we select a bit-index as a witness for its evaluation. If $\alpha(F) = 1$, we select an arbitrary bit-index. We then mark the selected bit-index in all bit-vector variables contained in F, as well as in all other

bit-vector variables of the same bit-width. Having done this for all equalities, we end up with sets M_i of selected bit-indices, for all $i \in \{1, \ldots, k\}$, where

$$|M_i| \leqslant min\{n_i, |\Phi|\}$$
$$M_i = M_j \qquad \forall j \in \{1, \ldots, k\} \text{ with } n_i = n_j$$

The selected indices contain a witness for the evaluation of each equality. We now add arbitrary further bit-indices, again selecting the same indices in bit-vector variables of the same bit-width, until $|M_i| = min\{n_i, |\Phi|\}\ \forall i \in \{1, \ldots, k\}$. Finally, we can directly construct α' using the selected indices and get $\alpha'(\Phi') = \alpha(\Phi) = 1$ because of the fact that we included a witness for every equality in our index-selection process. Note, that we only had to choose a specific witness for the case that $\alpha(F) = 0$. For $\alpha(F) = 1$, we were able to choose an arbitrary bit-index because every satisfied equality will trivially still be satisfied when only a subset of all bit-indices is considered. □

Remark 3.10. A similar proof can be found in [140, 141]. While the focus of [140, 141] lies on improving the practical efficiency of SMT solvers by reducing the bit-width of a given formula before bit-blasting, the author does not investigate its influence on the complexity of a given problem class. In fact, the author claims that bit-vector theories with common operators are NP-complete. As we have already shown in [151], this only holds if unary encoding on the bit-widths is used. However, unary encoding leads to the fact that the given class of formulas remains NP-complete, independent of whether a reduction of the bit-width is possible. While the arguments on bit-width reduction given in [140, 141] still hold for binary encoded bit-vector formulas when only bitwise operators are used, our proof considers the complexity of the problem class.

3.5 Discussion

The complexity results given in Section 3.4 provide some insight in where the expressiveness of bit-vector logics with binary encoding comes from. While we assume bitwise operations and equality naturally being part of a bit-vector logic, if and to what extent we allow shifts directly determines its complexity. Shifts, in a certain way, allow different bits of a bit-vector to interact with each other. Whether we allow no interaction, interaction between neighbouring bits, or interaction between arbitrary bits is crucial to the expressiveness of bit-vector logics and the complexity of their decision problem.

Additionally, we directly get classifications for various other bit-vector operations: for example, we still remain in PSPACE if we add *linear modular arithmetic* to QF_BV2$_{\ll 1}$. This can be seen by replacing expressions $x^{[n]} = y^{[n]} + z^{[n]}$ by

$$\left(x^{[n]} = y^{[n]} \oplus z^{[n]} \oplus {c_{in}}^{[n]}\right) \,\&\, \left({c_{in}}^{[n]} = {c_{out}}^{[n]} \ll 1^{[n]}\right) \,\&$$
$$\left({c_{out}}^{[n]} = \left(x^{[n]} \& y^{[n]}\right) \mid \left({c_{in}}^{[n]} \& y^{[n]}\right) \mid \left(x^{[n]} \& {c_{in}}^{[n]}\right)\right)$$

with new variables ${c_{in}}^{[n]}, {c_{out}}^{[n]}$, and by splitting multiplication by constant into several multiplications by 2 (or shifts by 1), similar to [209, 208]. However, this is not surprising since QFPAbit is already known to be PSPACE-complete [208].

More interestingly, we can also extend QF_BV2$_{\ll 1}$ (or QFPAbit) by *indexing* (denoted by $x^{[n]}[i]$) without growth in complexity. The counter we introduced in our translation from QF_BV2$_{\ll 1}$ to sequential circuits can be used to return the value at a specific bit-index of a bit-vector. Extending QF_BV2$_{\ll 1}$ with additional relational operators like e.g., *unsigned less than* (denoted by $x^{[n]} <_{\mathbf{u}} y^{[n]}$) does not increase complexity, either. For instance, the above expression can be replaced by checking whether $x - y < 0$ holds, which can simply be done by constructing an adder for $x^{[n]} + \left(\sim y^{[n]} + 1^{[n]}\right)$, as shown above, and then check whether overflow occurs, i.e., $\left(y^{[n]} \neq 0^{[n]}\right) \,\&\, \left({c_{out}}^{[n]}[n-1] = 0^{[1]}\right)$.

On the other hand, *slicing* (denoted by $x^{[n]}\,[i:j]$) cannot be added without growth in complexity. To prove this, consider

$$\left(x^{[n]}\,[n-1:c] = y^{[n]}\,[n-c-1:0]\right) \,\&\, \left(x^{[n]}\,[c-1:0] = 0^{[c]}\right),$$

which is equivalent to

$$x^{[n]} = (y^{[n]} \ll c^{[n]}),$$

and shows that *slicing* can be used to express shift by constant. Therefore, the resulting logic becomes NEXPTIME-complete. The same result holds for general *multiplication*. We can use

$$x^{[n]} = (y^{[n]} \cdot 2^{c[n]})$$

to replace shift by constant and use exponentiation by squaring to calculate $2^{c[n]}$ with $\lceil \log_2(n) \rceil$ multiplications.

Note that those results only hold for fixed-size bit-vector logics. For example, allowing *multiplication* (in combination with *addition*) makes non-fixed-size bit-vector logics undecidable [79]. We are not aware of any complexity results concerning non-fixed-size bit-vector logics with *slicing* or *shift by constant*.

3.6 Conclusion

In this chapter, we discussed the complexity of fixed-size bit-vector logics with binary encoding on numbers. In contrast to existing literature, except for our previous work [151], where usually it is not distinguished between unary or binary encoding, we argued that it is important to make this distinction. Our results apply to the actually much more natural binary encoding as it is also used in standard formats, e.g., in the SMT-LIB format. In previous work [151], we already showed the quantifier-free case of those bit-vector logics to be NEXPTIME-complete. We now extended our previous work by analyzing the quantifier-free case in more detail and gave two new complexity results.

In particular, we showed that the complexity of deciding quantifier-free bit-vector logics with bitwise operations and equality depends on whether we allow *shift by constant* (QF_BV2$_{\ll c}$), *shift by 1* (QF_BV2$_{\ll 1}$), or *no shifts at all* (QF_BV2$_{bw}$). While deciding QF_BV2$_{\ll c}$ still remains NEXPTIME-complete, we proved that QF_BV2$_{\ll 1}$ is PSPACE-complete, and QF_BV2$_{bw}$ even becomes NP-complete. In addition to the already previously proposed concept of bit-width boundedness, this gives an alternative way to avoid the increase in complexity that comes with binary encoding in the general case. To be more specific for practical logics, we then looked at the effect some other common operations have on this complexity results. We discussed why logics with *addition*, *multiplication by constant*, *indexing*, and *relational operations* still can be decided in PSPACE, and showed that allowing *general multiplication* or *slicing* already causes NEXPTIME-completeness.

On the one hand, our theoretical results give an argument for using more powerful solving techniques when dealing with bit-vector logics. Currently the most common approach used in state-of-the-art SMT solvers for bit-vectors is based on simple rewriting, bit-blasting, and SAT solving. We have shown this can possibly produce exponentially larger formulas when a logarithmic encoding is used in the input. As already argued in [151], possible candidates for the general case are techniques used in EPR and/or DQBF solvers (see e.g. [99, 147]).

On the other hand, we described various logics that remain in lower complexity classes. For QF_BV2$_{bw}$ this shows the importance of bit-width reduction as proposed in [140, 141] before bit-blasting. For formulas in QF_BV2$_{\ll 1}$ or one of the related classes, only using *shift by 1*, *addition*, *multiplication by constant*, and *indexing*, techniques used in state-of-the-art QBF solvers [165] or symbolic model checking on sequential circuits [194] might be of interest.

3.7 Appendix

3.7.1 Table: Completeness Results for Fixed-Size and Non-Fixed-Size Logics

	fixed-size	*non-fixed-size*
QF_BV2:	NEXPTIME [151]	undecidable [79]
QF_BV2$_{\ll c}$:		?
QF_BV2$_{\ll 1}$:	PSPACE [$\star$]	PSPACE [209, 208]
QF_BV2$_{bw}$:	NP [$\star$]	NP [$\star$] [2]
Presburger arithmetic:	?	NP

($\star$ shown in this chapter)

3.7.2 Example: A Reduction of QBF to QF_BV$2_{\ll 1}$

Consider the following QBF:

$$\begin{aligned}\exists x \forall u_2 \exists y \forall u_1 u_0 \exists z \,.\, & (u_2 \vee u_1 \vee \neg z) \wedge \\ & (u_2 \vee \neg x \vee y) \wedge \\ & (u_0 \vee \neg x \vee \neg z) \wedge \\ & (u_1 \vee \neg y \vee z) \wedge \\ & (u_0 \vee \neg u_1 \vee z)\end{aligned}$$

This QBF is *satisfiable*, and is translated to the following QF_BV$2_{\ll 1}$ formula:

$$\begin{aligned}&\Big((U_2 \mid U_1 \mid\!\sim Z) \& (U_2 \mid\!\sim X \mid Y) \& (U_0 \mid\!\sim X \mid\!\sim Z) \& \\ &(U_1 \mid\!\sim Y \mid Z) \& (U_0 \mid\!\sim U_1 \mid Z)\Big) = \sim 0^{[8]} \wedge \\ &\bigwedge_{m \in \{0,1,2\}} \left(\left(\bigwedge_{0 \leqslant i < m} U_i \right) \oplus U_m \;=\; U_m \ll 1 \right) \wedge \\ &(X \,\&\sim 1) \;=\; (X \ll 1) \wedge \\ &\big(U_2' \;= \sim\big((U_2 \ll 1) \oplus U_2\big)\big) \;\wedge\; \big((Y \& U_2') \;=\; \big((Y \ll 1) \& U_2'\big)\big)\end{aligned} \tag{3.4}$$

Let us note that we omit the bit-widths for the sake of readability, and also because all the bit-vector variables are of bit-width 8.

In the following, let us show that this formula is also satisfiable. Note that $U_0 = 01010101_2{}^{[8]}$, $U_1 = 00110011_2{}^{[8]}$, and $U_2 = 00001111_2{}^{[8]}$. The following table gives some insight into the process of generating these binary magic numbers:

$1 \oplus U_0$	$=$	$U_0 \ll 1$	$\rightarrow$	U_0	$U_0 \oplus U_1$	$=$	$U_1 \ll 1$	$\rightarrow$	U_1
$\neg U_{0,7}$		$U_{0,6}$		0	$U_{1,7}$		$U_{1,6}$		0
$\neg U_{0,6}$		$U_{0,5}$		1	$\neg U_{1,6}$		$U_{1,5}$		0
$\neg U_{0,5}$		$U_{0,4}$		0	$U_{1,5}$		$U_{1,4}$		1
$\neg U_{0,4}$		$U_{0,3}$		1	$\neg U_{1,4}$		$U_{1,3}$		1
$\neg U_{0,3}$		$U_{0,2}$		0	$U_{1,3}$		$U_{1,2}$		0
$\neg U_{0,2}$		$U_{0,1}$		1	$\neg U_{1,2}$		$U_{1,1}$		0
$\neg U_{0,1}$		$U_{0,0}$		0	$U_{1,1}$		$U_{1,0}$		1
$\neg U_{0,0}$		0		1	$\neg U_{1,0}$		0		1

[2] Although we did not point this out explicitly, it is easy to check that the proof we gave for the specific fixed-sized bit-vector logic in Theorem 3.9 still holds for the corresponding non-fixed-size one if we set all bit-widths in Φ' to $n' := |\Phi|$.

$(U_0 \wedge U_1) \oplus U_2$	$=$	$U_2 \ll 1$	$\rightarrow$	U_2
$U_{2,7}$		$U_{2,6}$		0
$U_{2,6}$		$U_{2,5}$		0
$U_{2,5}$		$U_{2,4}$		0
$\neg U_{2,4}$		$U_{2,3}$		0
$U_{2,3}$		$U_{2,2}$		1
$U_{2,2}$		$U_{2,1}$		1
$U_{2,1}$		$U_{2,1}$		1
$\neg U_{2,0}$		0		1

First, we show how the bits of X get restricted by the constraints introduced above. Let us denote the originally unrestricted bits of X with $x_7, x_6, \ldots, x_0$. Since the bit-vectors

$$\begin{aligned}(X \& \sim 1) &= (x_7, x_6, x_5, x_4, x_3, x_2, x_1, 0)\\ (X \ll 1) &= (x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0)\end{aligned}$$

are forced to be equal, all bits of X have to be equal:

$$X := (x_0, x_0, x_0, x_0, x_0, x_0, x_0, x_0)$$

Similarly we get the constraints on Y:

$$U_2' := \sim((U_2 \ll 1) \oplus U_2) = 11101110$$

and, therefore,

$$\begin{aligned}(Y \& U_2') &= (y_7, y_6, y_5, 0, y_3, y_2, y_1, 0)\\ (Y \ll 1) \& U_2') &= (y_6, y_5, y_4, 0, y_2, y_1, y_0, 0)\end{aligned}$$

which are forced to be equal. Then we put restrictions on individual bits of Y:

$$Y := (y_4, y_4, y_4, y_4, y_0, y_0, y_0, y_0)$$

Finally, Z is not restricted in any way since u_0 is the innermost universal variable that z depends on, i.e., z depends on all universal variables.

$$Z := (z_7, z_6, z_5, z_4, z_3, z_2, z_1, z_0)$$

In order to show that the formula (3.4) is satisfiable, let us evaluate the "clauses" in the formula:

$$\begin{aligned}(U_2 \mid U_1 \mid\sim Z) &= (\neg z_7, \neg z_6, 1, 1, 1, 1, 1, 1)\\ (U_2 \mid\sim X \mid Y) &= (\neg x_0 \vee y_4, \neg x_0 \vee y_4, \neg x_0 \vee y_4, \neg x_0 \vee y_4, 1, 1, 1, 1)\\ (U_0 \mid\sim X \mid\sim Z) &= (\neg x_0 \vee \neg z_7, 1, \neg x_0 \vee \neg z_5, 1, \neg x_0 \vee \neg z_3, 1, \neg x_0 \vee \neg z_1, 1)\\ (U_1 \mid\sim Y \mid Z) &= (\neg y_4 \vee z_7, \neg y_4 \vee z_6, 1, 1, \neg y_0 \vee z_4, \neg y_0 \vee z_3, 1, 1)\\ (U_0 \mid\sim U_1 \mid Z) &= (1, 1, z_5, 1, 1, 1, z_1, 1)\end{aligned}$$

By applying *bitwise and* to them, we get the following bit-vector:

$$\Phi' = \begin{pmatrix} \neg z_7 \wedge (\neg x_0 \vee y_4) \wedge (\neg x_0 \vee \neg z_7) \wedge (\neg y_4 \vee z_7) \\ \neg z_6 \wedge (\neg x_0 \vee y_4) \wedge (\neg y_4 \vee z_6) \\ (\neg x_0 \vee y_4) \wedge (\neg x_0 \vee \neg z_5) \wedge z_5 \\ \neg x_0 \vee y_4 \\ (\neg x_0 \vee \neg z_3) \wedge (\neg y_0 \vee z_4) \\ \neg y_0 \vee z_3 \\ (\neg x_0 \vee \neg z_1) \wedge z_1 \\ 1 \end{pmatrix}$$

In order to check if $\Phi' = \sim 0^{[8]}$ is satisfiable, one can check the satisfiability of the following simplified clause set:

$$\{\neg z_7, \neg x_0, \neg y_4, \neg z_6, z_5, \neg y_0 \vee z_4, \neg y_0 \vee z_3, z_1\}$$

This can be satisfied, e.g., by setting

$$\begin{aligned} z_7 = x_0 = y_4 = z_6 = y_0 = 0 \\ z_5 = z_1 = 1 \end{aligned}$$

Therefore,

$$\begin{array}{llllll} U_0 & = 01010101_2{}^{[8]}, & U_1 & = 00110011_2{}^{[8]}, & U_2 & = 00001111_2{}^{[8]}, \\ X & = 00000000_2{}^{[8]}, & Y & = 00000000_2{}^{[8]}, & Z & = 00111111_2{}^{[8]} \end{array}$$

is a possible solution of the bit-vector formula (3.4).

Chapter 4

Complexity of Fixed-Size Bit-Vector Logics

Published. In the Journal of Theory of Computing Systems (TOCS'15), Springer 2015, DOI 10.1007/s00224-015-9653-1 [153].

Authors. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere.

Abstract. Bit-precise reasoning is important for many practical applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. From the theoretical point of view, only few results on the complexity of fixed-size bit-vector logics have been published. Some of these results only hold if unary encoding on the bit-width of bit-vectors is used. In our previous work [151], we have already shown that binary encoding adds more expressiveness to various fixed-size bit-vector logics with and without quantification. In a follow-up work [101], we then gave additional complexity results for several fragments of the quantifier-free case. In this chapter, we revisit our complexity results from [101, 151] and go into more detail when specifying the underlying logics and presenting the proofs. We give a better insight in where the additional expressiveness of binary encoding comes from. In order to do this, we bring together our previous work and propose several new complexity results for new fragments and extensions of earlier bit-vector logics. We also discuss the expressiveness of various bit-vector operations in more detail. Altogether, we provide the currently most complete overview on the complexity of common bit-vector logics.

4.1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for

bit-precise reasoning are Boolector [47], MathSAT [50], STP [104], Z3 [81], and Yices [87].

The theory of *fixed-size bit-vector logics* is investigated in several scientific works [19, 35, 51, 74, 96], and even concrete formats for specifying such bit-vector problems exist, e.g., the SMT-LIB format [18] or the BTOR format [48]. Working with *non-fixed-size* bit-vectors has been considered, for instance, in [6, 35], and, more recently, in [208, 209], but is not further discussed in this chapter. Most industrial applications (and examples in the SMT-LIB [1]) have fixed bit-width.

We investigate the *complexity* of solving *fixed-size bit-vector formulas*. Some papers propose such complexity results, e.g., in [19], the authors consider the common quantifier-free bit-vector logic and give an argument for NP-*hardness* of its satisfiability problem. In [51], a sublogic of the previous one is claimed to be NP-*complete*. Interestingly, in [52], there is a claim about the full quantifier-free logic being NP-complete, however the proposed decision procedure justifies this claim only if the bit-widths of the bit-vectors in the input formula are written/encoded in *unary* format. In [233, 234], the *quantified* case is addressed, and the satisfiability problem for this logic with uninterpreted functions is proved to be NEXPTIME-*complete*. However, the proof, similarly to the decision procedure in [52], only holds if we assume unary encoded bit-widths.

Parts of our chapter already appeared as previous work [101, 151]. Apart from this, we are not aware of any work that investigates how the encoding of the bit-widths in the input affects complexity (as an exception, see [71, Page 239, Footnote 3]). In practice, the more natural and exponentially more succinct *logarithmic* encoding is used, such as in the SMT-LIB [18] or the BTOR [48] format. We investigate how complexity varies if we consider either a unary or a binary encoding. Note that binary encoding, throughout the whole chapter, can be replaced with any other logarithmic encoding.

The present chapter extends our previous work in several ways. After giving a motivation for the use of binary encoded bit-vector logics in Section 4.2, we specify various fixed-size bit-vector logics in detail (Section 4.3). While our previous chapters were referring to the common syntax and semantics used in other works, e.g., [19, 35, 48, 51, 74, 96], but was never fully specified from the theoretical point of view, we now want to provide self-contained descriptions for the bit-vector logics that we are considering. Therefore, we introduce syntax and semantics for fixed-size bit-vector logics containing all common bit-vector operations as used in the SMT-LIB format.

After these preliminary definitions, we give a short overview of the existing complexity results for bit-vector logics with unary encoding in Section 4.4. We then introduce the concept of scalar-boundedness for bit-vector logics with binary encoding in Section 4.5 and give improved versions of our complexity proofs for quantifier-free bit-vector logics in Section 4.6. Although our previous proofs from [101, 151] are still valid, we modified and restructured our work to present

[1] `http://www.smtlib.org/`

those proofs in a clearer, easier-to-read, way. In Section 4.7, we look at the expressiveness of various bit-vector operations and analyze whether they can be used to extend some of the previously defined fragments or to give an alternative characterization of a given class. We then revisit the quantified case in Section 4.8 and give new complexity results for fragments with restrictions on operations and the bit-widths of universal variables. Also, we provide a new complexity result for quantifier-free logics extended with non-recursive macros, which are allowed, for example, in the SMT-LIB format.

Finally, we discuss practical considerations of our results in Section 4.9, giving a brief overview of related practical work which we presented in [100, 150], and explain how our theoretic contributions can help to improve practical SMT solving. We then conclude in Section 4.10. The Appendix contains examples that make some definitions and proofs easier to understand.

4.2 Motivation

In practice, state-of-the-art bit-vector solvers rely on rewriting and bit-blasting. The latter is defined as the process of translating a bit-vector description (also called *word-level* description) into a combinatorial circuit, as in hardware synthesis. The result can then be checked by a (propositional) SAT solver.

Usually, numbers contained in a bit-vector description (e.g., the bit-widths of bit-vector variables) are encoded in a logarithmic way. When translating the original description into a circuit, all numbers are effectively replaced by their unary encoding. Bit-blasting can therefore lead to an exponential growth, if the numbers are not logarithmic in the original description size.

To illustrate this effect on a practical example, consider the following bit-vector formula in SMT-LIB syntax [18]:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(declare-fun z () (_ BitVec 1000000))
(assert (= z (bvadd x y)))
(assert (= z (bvshl x (_ bv1 1000000))))
(assert (distinct x y))
```

The first line defines the logic to be the one of quantifier-free bit-vectors. The following three lines introduce bit-vector variables x, y, and z of bit-width one million. The last three lines enforce some constraints between the variables. Basically, the formula verifies that, for an arbitrary bit-vector x of bit-width one million, there exists no bit-vector $y \neq x$ with $x + y = x \ll 1$.

Written to a file, this formula can be encoded with 217 bytes. Using the SMT solver Boolector (even with all rewritings switched on), bit-blasting produces a circuit of size 129 MB, encoded in the actually rather compact AIGER format. Tseitin transformation results in a CNF in DIMACS format of size 843 MB. A

bit-width of 10 million bits can be represented by four more bytes in the original SMT-LIB input, but could not be bit-blasted anymore with our tool-flow (due to integer overflow). As this example illustrates, checking satisfiability of bit-vector formulas through bit-blasting can suffer dramatically from the exponential growth caused by the implicit unary re-encoding of the numbers.

Obviously, its exponential nature also disqualifies bit-blasting as a sound way to prove that the satisfiability problem for (quantifier-free) bit-vector logics is in NP. In [151], we showed that deciding bit-vector logics, even without quantifiers, is much harder. It turned out to be NEXPTIME-complete. Informally speaking, we showed that moving from unary to binary encoding for bit-widths increases complexity *exponentially* and that binary encoding has at least as much expressive power as quantification. However, in [101, 151], we also proposed certain restrictions for bit-vector problems to remain in a "lower" complexity class, when moving from unary to binary encoding.

These theoretical insights as well as later practical results from [100, 150] give reason to look into bit-vector logics more closely and to provide a comprehensive framework for dealing with complexity of bit-vector logics, particularly combined with the use of a binary encoding.

4.3 Preliminaries

$\mathbb{N}$ denotes the set of natural numbers $\{0, 1, 2, \dots\}$, while $\mathbb{N}^+$ denotes $\mathbb{N}\backslash\{0\}$ and $\mathbb{B} := \{0, 1\}$ is the Boolean domain, thus, truth values *false* and *true* are represented by 0 and 1, respectively. Given $n \in \mathbb{N}^+$, let $\mathsf{L}n$ denote the *ceiling of the logarithm of n base 2*: $\mathsf{L}n := \lceil \log_2 n \rceil$.

4.3.1 SAT, QBF, and DQBF

Let V be a set of Boolean variables. *Boolean formulas* over V are defined inductively as follows: (i) x is a Boolean formula where $x \in V$; (ii) $\neg\phi_0$, $(\phi_0 \wedge \phi_1)$, $(\phi_0 \vee \phi_1)$, $(\phi_0 \Rightarrow \phi_1)$, and $(\phi_0 \Leftrightarrow \phi_1)$ are Boolean formulas where ϕ_0, ϕ_1 are Boolean formulas. A Boolean formula ϕ is satisfiable if and only if there exists an assignment $\alpha : V \mapsto \mathbb{B}$ to the variables, such that ϕ evaluates to 1 under α. The Boolean satisfiability problem (SAT) is NP-complete.

The class of *Quantified Boolean Formulas* (QBF) is obtained by adding quantifiers to Boolean formulas. Each QBF ψ can be written in prenex normal form, i.e., as a closed formula $Q.\phi$ where Q is a *quantifier prefix* $\exists V_0 \forall V_1 \exists V_2 \dots \forall V_{m-1} \exists V_m$, the V_is are pairwise disjoint sets of variables, and ϕ is a Boolean formula, which is called the *matrix* of ψ. A variable $v \in V_i$ depends on a variable $v' \in V_j$ if and only if $i > j$. This defines a total order on the variables of ψ. A QBF is satisfiable if and only if there exist Skolem functions for its existential variables to make the formula evaluate to 1. The satisfiability problem for QBF is PSPACE-complete [185, 216].

Instead of using totally ordered quantifiers, it is also possible to extend Boolean formulas with *Henkin quantifiers* [122]. Henkin quantifiers specify variable depen-

dencies explicitly instead of using implicit dependencies defined by the quantifier order. This allows to define more general dependency constraints, only requiring a partial order. Adding Henkin quantifiers to Boolean formulas results in the class of *Dependency Quantified Boolean Formulas* (DQBF), as first defined in [187]. Again, a DQBF can always be expressed in prenex normal form, i.e., as a closed formula $Q'.\phi$, where Q' is a *quantifier prefix*

$$\forall u_1, \ldots, u_m \exists e_1(u_{1,1}, \ldots, u_{1,m_1}), \ldots, e_n(u_{n,1}, \ldots, u_{n,m_n})$$

where each $u_{i,j}$ is a universally quantified variable, $m_i \in \mathbb{N}$, and the *matrix* ϕ is a Boolean formula. In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given, and not implicitly defined by the order of the prefix (in contrast to QBF). The more general quantifier order makes DQBF more powerful than QBF and allows more succinct encodings. A DQBF is satisfiable if and only if there exist Skolem functions for its existential variables to make the formula evaluate to 1. In DQBF, the arguments for Skolem functions of an existential variable are exactly the universal variables that are explicitly specified in its Henkin quantifier. The satisfiability problem for DQBF is NEXPTIME-complete [188, 187]. Although we did not formally specify the dependencies of universal variables, this can be done by the use of Herbrand functions [9].

Throughout this chapter, we use SAT, QBF, and DQBF to give reductions from or to certain bit-vector logics, showing inclusion or hardness for the corresponding complexity class, respectively. While SAT and QBF are considered to be prototypical complete problems for their complexity classes, DQBF is used less frequently. Another NEXPTIME-complete logic used in reductions in the context of unary encoded bit-vector logics [233] is *Effectively Propositional Logic* (EPR) [161]. However, due to its simplicity, we consider DQBF to be a better choice for our purposes.

4.3.2 Circuits

We distinguish between two kind of circuits: *combinatorial circuits* and *sequential circuits*. For both kinds of circuits, we stick closely to the definitions in [208]:

A combinatorial circuit with n_i inputs and n_o outputs is a finite acyclic directed graph with exactly n_i vertices of in-degree zero and n_o vertices of out-degree zero. All vertices of a non-zero in-degree have a logical function assigned to them and are called gates. All vertices of in-degree one represent a NOT-gate and vertices of greater in-degrees are either AND- or OR-gates. Given boolean values for the inputs, each gate can be evaluated in the natural way according to the logical function it represents. As already noted in the introduction, this kind of representation of a bit-vector formula is created during bit-blasting. For every combinatorial circuit, a corresponding set of n_o SAT formulas with n_i variables can be constructed naturally.

A (clocked) sequential circuit SC consists of a combinatorial circuit C and a set of D-type flip-flops. The data input of each flip-flop is connected to a unique output of C and the Q-output of each flip-flop is connected to a unique input of C. Such a backward-connected output-input pair will be denoted as a state variable. The circuit is assumed to work in clock pulses. In every clock pulse, it takes the values of its inputs and computes the output values. Via the flip-flops these values are routed back to the inputs for the use in the next clock cycle. Inputs of C that do not receive their value from an output through a flip-flop will be called the inputs of the sequential circuit SC and outputs of C that do not pass their value to an input of a flip-flop will be called the outputs of the sequential circuit SC.

All the state variables are assumed to be provided with initial values stored in the flip-flops before the first clock cycle. The input variables need to be provided values from outside the system at every clock cycle and the output variables produce a new output at every clock cycle. A sequential circuit can be used to recognize languages. A word $w \in (\{0,1\}^{n_i})^+$ is said to be accepted by a sequential circuit SC with one output o, if and only if the value of o is 1 after the last clock cycle when w is given as input, one letter each clock cycle.

Symbolic model checking for sequential circuits refers to the problem of checking whether the language for a given sequential circuit is empty. It is known to be PSPACE-complete [194, 198, 205].

4.3.3 Fixed-Size Bit-Vector Logics

A *bit-vector*, or word, is a sequence of bits, i.e., Boolean values. Such a sequence may be either infinite or of a fixed size $n \in \mathbb{N}^+$, where n is called the *bit-width* of the bit-vector. While *non-fixed-size* bit-vectors have been considered for example in [6, 35, 208, 209], working with *fixed-size* bit-vectors is the focus of this chapter.

Let D_n denote the set of all bit-vectors of bit-width n. Given $d \in D_n$, the ith bit of d is denoted by $d[i]$, where $i \in \mathbb{N}$ and $i < n$. Using vector notation, d is written as $\big(d[n-1], \ldots, d[1], d[0]\big)$, i.e., the most significant bit standing on the left-hand side and the least significant bit on the right-hand side. Sometimes, we omit parentheses and commas.

Syntax and semantics of fixed-size bit-vector logics do not differ much in the literature [19, 35, 51, 74, 96]. Concrete formats for specifying bit-vector problems also exist, e.g., the SMT-LIB format [18] or the BTOR format [48]. In the subsequent sections, we give the necessary definitions, in a more general way than in the works cited above, in order to propose a uniform and general framework using any set of bit-vector operations.

Syntax

The main objective of this section is to define *bit-vector formulas*. As it turns out in Definition 4.2 and 4.3, such a formula, informally speaking, is a combination of bit-vector operations on some atomic elements, each of which can be represented

either as a bit-vector or an integer, which we call a *scalar*. Let us emphasize that scalars in formulas are *not* represented as bit-vectors. Note that the bit-width of a bit-vector is also a scalar.

A bit-vector operator symbol (or *operator* for short) represents an operation that takes some bit-vector operands and scalar operands, and computes a single bit-vector. Given an arbitrary *operator set*, one has to specify syntactic rules for using the operators. Definition 4.1 of a *signature* captures these rules by providing three properties for each operator: (1) An operator is given an *arity*, which is a pair of numbers that specify the number of bit-vector operands and the number of scalar operands, respectively. For instance, the arithmetic operator *addition* has 2 bit-vector and 0 scalar operands, while *extraction* has 1 bit-vector and 2 scalar operands. (2) Since there usually exist restrictions on what kind of operands are legal to use with an operator, a signature has to specify a *condition* on the bit-widths and scalar values of operands. For instance, the operands of *addition* must be of the same bit-width; the scalar operands i, j of *extraction* must be less than the bit-width of the bit-vector operand and $i \geqslant j$. (3) A *bit-width* of the resulting bit-vector is assigned to each legal combination of bit-widths and scalar values of operands.

Definition 4.1 (Signature)**.** A signature for an operator set Op is defined as a set $\Sigma_{Op} := \{\langle arity_o, cond_o, wid_o\rangle \mid o \in Op\}$, where

- $arity_o \in \mathbb{N} \times \mathbb{N}$;
- $cond_o : (\mathbb{N}^+)^k \times \mathbb{N}^l \mapsto \mathbb{B}$ where $\langle k, l\rangle := arity_o$;
- $wid_o : Par_o \mapsto \mathbb{N}^+$ where
  $$Par_o := \big\{p \in (\mathbb{N}^+)^k \times \mathbb{N}^l \mid \langle k, l\rangle := arity_o,\ cond_o(p)\big\}.$$

Table 4.1 shows the set of the most common operators provided by the SMT-LIB format [18] and the literature [19, 35, 51, 74, 96], such as bitwise operators (*negation*, *and*, *or*, *xor*, etc.), relational operators (*equality*, *unsigned/signed less than*, *unsigned/signed less than or equal*, etc.), arithmetic operators (*addition*, *subtraction*, *multiplication*, *unsigned/signed division*, *unsigned/signed remainder*, etc.), shifts (*left shift*, *logical/arithmetic right shift*), *extraction*, *concatenation*, *zero/sign extension*, etc. Let $\mathbf{Op}$ denote the *common operator set* given in Table 4.1. $\mathbf{Op}$ includes all bit-vector operators used in the SMT-LIB providing a collection of the most common bit-vector operators in software and hardware verification; other frameworks, like Boolector and Z3, provide additional useful operators, e.g., reduction operators and overflow operators. Let $\mathbf{\Sigma_{Op}}$ denote the *common signature* for $\mathbf{Op}$. Note that Table 4.1 specifies some of the syntactic properties provided by $\mathbf{\Sigma_{Op}}$ in an implicit way: the arity is completely, the condition is partly implicit.

	operation	condition	bit-width	alternative syntax
negation:	*bvnot* $\left(t^{[n]}\right)$		n	$\sim t^{[n]}$
and:	*bvand* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \& {t_2}^{[n]}\right)$
or:	*bvor* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \mid {t_2}^{[n]}\right)$
xor:	*bvxor* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \oplus {t_2}^{[n]}\right)$
nand:	*bvnand* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	
nor:	*bvnor* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	
xnor:	*bvxnor* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	
if-then-else:	*ite* $\left({t_1}^{[1]}, {t_2}^{[n]}, {t_3}^{[n]}\right)$		n	
equality:	*bvcomp* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	$\left({t_1}^{[n]} = {t_2}^{[n]}\right)$
unsigned (u.) less than:	*bvult* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	$\left({t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}\right)$
u. less than or equal:	*bvule* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
u. greater than:	*bvugt* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
u. greater than or equal:	*bvuge* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
signed (s.) less than:	*bvslt* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
s. less than or equal:	*bvsle* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
s. greater than:	*bvsgt* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
s. greater than or equal:	*bvsge* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		1	
shift left:	*bvshl* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \ll {t_2}^{[n]}\right)$
logical shift right:	*bvlshr* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \gg_{\mathbf{u}} {t_2}^{[n]}\right)$
arithmetic shift right:	*bvashr* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \gg_{\mathbf{s}} {t_2}^{[n]}\right)$
extraction:	*extract* $\left(t^{[n]}, i, j\right)$	$n > i \geqslant j$	$i - j + 1$	$t^{[n]}\,[i:j]$
concatenation:	*concat* $\left({t_1}^{[m]}, {t_2}^{[n]}\right)$		$m + n$	$\left({t_1}^{[m]} \circ {t_2}^{[n]}\right)$
zero extend:	*zero_extend* $\left(t^{[n]}, i\right)$		$n + i$	$ext_{\mathbf{u}}\left(t^{[n]}, i\right)$
sign extend:	*sign_extend* $\left(t^{[n]}, i\right)$		$n + i$	
rotate left:	*rotate_left* $\left(t^{[n]}, i\right)$	$n > i \geqslant 0$	n	
rotate right:	*rotate_right* $\left(t^{[n]}, i\right)$	$n > i \geqslant 0$	n	

continued on next page

	continued from previous page			
repeat:	*repeat* $\left(t^{[n]}, i\right)$	$i > 0$	$n \cdot i$	
unary minus:	*bvneg* $\left(t^{[n]}\right)$		n	$-t^{[n]}$
addition:	*bvadd* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} + {t_2}^{[n]}\right)$
subtraction:	*bvsub* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} - {t_2}^{[n]}\right)$
multiplication:	*bvmul* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} \cdot {t_2}^{[n]}\right)$
unsigned division:	*bvudiv* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	$\left({t_1}^{[n]} /_{\mathbf{u}} {t_2}^{[n]}\right)$
u. remainder:	*bvurem* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	
signed division:	*bvsdiv* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	
s. remainder with rounding to 0:	*bvsrem* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	
s. remainder with rounding to $-\infty$:	*bvsmod* $\left({t_1}^{[n]}, {t_2}^{[n]}\right)$		n	

Table 4.1: Syntax (signature) for common bit-vector operators

The simplest bit-vector expressions, or *terms*, are the variables and constants, as Definition 4.2 shows. Operators can be applied to bit-vector terms which obey the syntactic rules given by the signature of the operator set. While operators have a priori fixed syntax and semantics, uninterpreted functions can be introduced on demand.

Definition 4.2 (Term)**.** A bit-vector term t of bit-width $n \in \mathbb{N}^+$ is denoted by $t^{[n]}$. A term over a signature Σ_{Op} is defined inductively as follows:

	term	condition	bit-width
constant:	$c^{[n]}$	$c \in \mathbb{N}, 0 \leqslant c < 2^n$	n
variable:	$x^{[n]}$	x is an identifier	n
operation:	$o({t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]}, i_1, \ldots, i_l)$	$o \in Op$, $\langle k, l\rangle := arity_o$ ${t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]}$ are terms $i_1, \ldots, i_l \in \mathbb{N}$ $cond_o(n_1, \ldots, n_k, i_1, \ldots, i_l)$	$wid_o(n_1, \ldots, i_l)$
uninterpreted function:	$f^{[n]}({t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]})$	f is an identifier, $k \in \mathbb{N}$ ${t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]}$ are terms	n

Let us emphasize that, in a term, bit-widths are specified explicitly only for constants, variables, and uninterpreted functions. In all other cases, the bit-width

is implicit, i.e., it can be derived from the bit-widths of the operands of operations. In the following, we may omit explicit bit-widths and parentheses if they can be concluded from the context.

Definition 4.3 (Formula)**.** A *bit-vector formula* is an expression $Q.t^{[1]}$, where $t^{[1]}$ is a bit-vector term, Q is a *quantifier prefix* $Q_0 {x_0}^{[n_0]} Q_1 {x_1}^{[n_1]} \dots Q_k {x_k}^{[n_k]}$, each $Q_i \in \{\forall, \exists\}$, and each ${x_i}^{[n_i]}$ is a bit-vector variable. We call t the *matrix* of the formula.

If only existential quantifiers appear in a formula, we may omit the quantifier prefix and refer to this kind of formula as a *quantifier-free* one. In the same way, we refer to a formula as being *quantified*, if it contains universal quantifiers.

W.l.o.g., we can assume that variables and uninterpreted functions are identified by their unique names. In a formula, therefore, each variable and each uninterpreted function must be used in a consistent way, regarding its bit-width and the bit-widths of its arguments.

In the literature, most of the approaches distinguish between a bit-vector level and a Boolean level within a bit-vector formula, by allowing only *relational operators* (i.e., operators with result of bit-width 1) at the Boolean level [19, 49, 51, 74, 96]. Note that, in our definitions, there is no such explicit distinction. Therefore, for example, relational operators are allowed to be embedded in concatenations or arithmetic operations. However, by introducing the so-called *flat form* in Definition 4.8, the same separation of a Boolean level and a bit-vector level can be made in any bit-vector formula over $\mathbf{\Sigma_{Op}}$, assuming the common interpretation of $\mathbf{\Sigma_{Op}}$, defined in Section 4.3.3.

Semantics

Given a signature Σ_{Op} and an operator $o \in Op$ where $\langle k, l \rangle := arity_o$, each $p := (n_1, \dots, n_k, i_1, \dots, i_l) \in Par_o$ can be mapped to a set of possible operands (bit-vectors and scalars) and also to a set of possible results (bit-vectors). These two sets, called the domain and the range of p, are defined as follows:

$$\begin{aligned} Dom_o(p) &:= D_{n_1} \times \dots \times D_{n_k} \times \{i_1\} \times \dots \times \{i_l\} \\ Range_o(p) &:= D_{wid_o(p)} \end{aligned}$$

In order to evaluate a term or formula, it is first necessary to interpret all the operators we use (Definition 4.4), and then to assign domain elements to free variables and to interpret uninterpreted functions (Definition 4.5).

Definition 4.4 (Interpretation)**.** An interpretation of a signature Σ_{Op} is defined as a set $\widehat{Op}$ of functions, consisting of an $\widehat{o}$ for each $o \in Op$, such that

$$\widehat{o} : \bigcup_{p \in Par_o} Dom_o(p) \mapsto \bigcup_{p \in Par_o} Range_o(p)$$

where

$$\forall p \in Par_o, d \in Dom_o(p) \,.\, \widehat{o}(d) \in Range_o(p)$$

Let $\widehat{\mathbf{Op}}$ denote the common interpretation of $\mathbf{\Sigma_{Op}}$, detailed in Table 4.2, based on [51, 58, 96] and the SMT-LIB. Note that Table 4.2 uses a notation that is introduced by the following definitions.

Definition 4.5 (Model)**.** $M := \langle \alpha, \widehat{F} \rangle$ is a model for a formula Φ where

- α is an assignment, i.e., it assigns an element of D_n to each free variable $x^{[n]}$ in Φ;
- $\widehat{F}$ is a set of interpretations $\widehat{f} : D_{n_1} \times \cdots \times D_{n_k} \mapsto D_n$ of all uninterpreted functions $f^{[n]}\left(t_1{}^{[n_1]}, \ldots, t_k{}^{[n_k]}\right)$ in Φ.

To facilitate the presentation, similar to [51, 96], we define an auxiliary bijective meta-function $nat_n : D_n \mapsto [0, 2^n - 1]$. Given a bit-vector $d \in D_n$, $nat_n(d) := \sum_{i=0}^{n-1} 2^i d[i]$. We also introduce the inverse meta-function $bv_n := nat_n^{-1}$.

Definition 4.6 (Evaluation)**.** Given a signature Σ_{Op}, a formula Φ over Σ_{Op}, an interpretation $\widehat{Op}$ of Σ_{Op}, and a model $M := \langle \alpha, \widehat{F} \rangle$ for Φ, Φ can be evaluated to either 0 or 1, by using the inductive definition of the evaluation function $[\![\cdot]\!]_M^{\widehat{Op}}$, as follows:

constant:	$\left[\!\left[c^{[n]}\right]\!\right]_M^{\widehat{Op}} := bv_n\left(c\right)$
variable:	$\left[\!\left[x^{[n]}\right]\!\right]_M^{\widehat{Op}} := \alpha(x)$
operation:	$\left[\!\left[o\left(t_1{}^{[n_1]}, \ldots, t_k{}^{[n_k]}, i_1, \ldots, i_l\right)\right]\!\right]_M^{\widehat{Op}} := \widehat{o}\left(\left[\!\left[t_1{}^{[n_1]}\right]\!\right]_M^{\widehat{Op}}, \ldots, \left[\!\left[t_k{}^{[n_k]}\right]\!\right]_M^{\widehat{Op}}, i_1, \ldots, i_l\right)$
uninterpreted function:	$\left[\!\left[f^{[n]}\left(t_1{}^{[n_1]}, \ldots, t_k{}^{[n_k]}\right)\right]\!\right]_M^{\widehat{Op}} := \widehat{f}\left(\left[\!\left[t_1{}^{[n_1]}\right]\!\right]_M^{\widehat{Op}}, \ldots, \left[\!\left[t_k{}^{[n_k]}\right]\!\right]_M^{\widehat{Op}}\right)$
quantifiers:	$\left[\!\left[\forall x^{[n]}.\Phi\right]\!\right]_M^{\widehat{Op}} := \bigwedge_{d \in D_n} [\![\Phi]\!]^{\widehat{Op}}_{\langle \alpha \cup \{x^{[n]} \mapsto d\}, \widehat{F} \rangle}$ $\left[\!\left[\exists x^{[n]}.\Phi\right]\!\right]_M^{\widehat{Op}} := \bigvee_{d \in D_n} [\![\Phi]\!]^{\widehat{Op}}_{\langle \alpha \cup \{x^{[n]} \mapsto d\}, \widehat{F} \rangle}$

As mentioned before, the common interpretation $\widehat{\mathbf{Op}}$ is given in Table 4.2. In the table, we omit the interpretation and the model for evaluation. Furthermore, we use two abbreviations:

$$
\begin{array}{rcl}
\mathit{msb}\left(t^{[n]}\right) & := & [\![t]\!][n-1] \\
\mathit{abs}\left(t^{[n]}\right) & := & \begin{cases} -t & \text{if } \mathit{msb}\,(t) \\ t & \text{otherwise} \end{cases}
\end{array}
$$

bvnot:	$[\![\sim t^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i\,(\neg\,[\![t]\!][i])\right)$
bvand:	$[\![t_1{}^{[n]} \& t_2{}^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i\,([\![t_1]\!][i] \wedge [\![t_2]\!][i])\right)$
bvor:	$[\![t_1{}^{[n]} \mid t_2{}^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i\,([\![t_1]\!][i] \vee [\![t_2]\!][i])\right)$
bvxor:	$[\![t_1{}^{[n]} \oplus t_2{}^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i\,(\neg\,[\![t_1]\!][i] \Leftrightarrow [\![t_2]\!][i])\right)$
bvnand:	$[\![\mathit{bvnand}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_1{}^{[n]} \& t_2{}^{[n]})]\!]$
bvnor:	$[\![\mathit{bvnor}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_1{}^{[n]} \mid t_2{}^{[n]})]\!]$
bvxnor:	$[\![\mathit{bvxnor}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_1{}^{[n]} \oplus t_2{}^{[n]})]\!]$
ite:	$[\![\mathit{ite}\,(t_1{}^{[1]}, t_2{}^{[n]}, t_3{}^{[n]})]\!] := \begin{cases} [\![t_2]\!] & \text{if } [\![t_1]\!] \\ [\![t_3]\!] & \text{otherwise} \end{cases}$
bvcomp:	$[\![t_1{}^{[n]} = t_2{}^{[n]}]\!] := bv_1\,(nat_n\,([\![t_1]\!]) = nat_n\,([\![t_2]\!]))$
bvult:	$[\![t_1{}^{[n]} <_{\mathbf{u}} t_2{}^{[n]}]\!] := bv_1\,(nat_n\,([\![t_1]\!]) < nat_n\,([\![t_2]\!]))$
bvule:	$[\![\mathit{bvule}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_2 <_{\mathbf{u}} t_1)]\!]$
bvugt:	$[\![\mathit{bvugt}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![t_2 <_{\mathbf{u}} t_1]\!]$
bvuge:	$[\![\mathit{bvuge}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\mathit{bvule}\,(t_2, t_1)]\!]$
bvslt:	$[\![\mathit{bvslt}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := bv_1\begin{pmatrix} (\mathit{msb}\,(t_1) \wedge \neg \mathit{msb}\,(t_2)) \vee \\ ((\mathit{msb}\,(t_1) \Leftrightarrow \mathit{msb}\,(t_2)) \wedge [\![t_1 <_{\mathbf{u}} t_2]\!]) \end{pmatrix}$
bvsle:	$[\![\mathit{bvsle}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim \mathit{bvslt}\,(t_2, t_1)]\!]$
bvsgt:	$[\![\mathit{bvsgt}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\mathit{bvslt}\,(t_2, t_1)]\!]$
bvsge:	$[\![\mathit{bvsge}\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\mathit{bvsle}\,(t_2, t_1)]\!]$
bvshl:	$[\![t_1{}^{[n]} \ll t_2{}^{[n]}]\!] := bv_n\left(nat_n\,([\![t_1]\!]) \cdot 2^k \mod 2^n\right)$ where $k := nat_n\,([\![t_2]\!])$
bvlshr:	$[\![t_1{}^{[n]} \gg_{\mathbf{u}} t_2{}^{[n]}]\!] := bv_n\left(\lfloor nat_n\,([\![t_1]\!])/2^k \rfloor\right)$ where $k := nat_n\,([\![t_2]\!])$
bvashr:	$[\![t_1{}^{[n]} \gg_{\mathbf{s}} t_2{}^{[n]}]\!] := \begin{cases} [\![\sim (\sim t_1 \gg_{\mathbf{u}} t_2)]\!] & \text{if } \mathit{msb}\,(t_1) \\ [\![t_1 \gg_{\mathbf{u}} t_2]\!] & \text{otherwise} \end{cases}$
extract:	$[\![t^{[n]}\,[i:j]]\!] := bv_{i-j+1}\left(\lfloor nat_n\,([\![t]\!])/2^j \rfloor \mod 2^i\right)$
concat:	$[\![t_1{}^{[m]} \circ t_2{}^{[n]}]\!] := bv_{m+n}\,(2^n nat_m\,([\![t_1]\!]) + nat_n\,([\![t_2]\!]))$
zero_extend:	$[\![\mathit{ext}_{\mathbf{u}}\,(t^{[n]}, i)]\!] := bv_{n+i}\,(nat_n\,([\![t]\!]))$

continued on next page

continued from previous page

sign_extend:	$\left[\!\left[\mathit{sign\_extend}\left(t^{[n]}, i\right)\right]\!\right]$ $:= \begin{cases} bv_{n+i}\left(2^{n+i} - 2^n + nat_n\left(\left[\!\left[t\right]\!\right]\right)\right) & \text{if } \mathit{msb}\,(t) \\ \left[\!\left[\mathit{ext}_{\mathbf{u}}\left(t^{[n]}, i\right)\right]\!\right] & \text{otherwise} \end{cases}$
rotate_left:	$\left[\!\left[\mathit{rotate\_left}\left(t^{[n]}, i\right)\right]\!\right]$ $:= \begin{cases} \left[\!\left[t\right]\!\right] & \text{if } n=1 \vee i=0 \\ \left[\!\left[t\left[n-i-1:0\right] \circ t\left[n-1:n-i\right]\right]\!\right] & \text{otherwise} \end{cases}$
rotate_right:	$\left[\!\left[\mathit{rotate\_right}\left(t^{[n]}, i\right)\right]\!\right]$ $:= \begin{cases} \left[\!\left[t\right]\!\right] & \text{if } n = 1 \vee i = 0 \\ \left[\!\left[t\left[i-1:0\right] \circ t\left[n-1:i\right]\right]\!\right] & \text{otherwise} \end{cases}$
repeat:	$\left[\!\left[\mathit{repeat}\left(t^{[n]}, i\right)\right]\!\right] := \begin{cases} \left[\!\left[t\right]\!\right] & \text{if } i = 1 \\ \left[\!\left[t \circ \mathit{repeat}\,(t, i-1)\right]\!\right] & \text{otherwise} \end{cases}$
bvneg:	$\left[\!\left[-t^{[n]}\right]\!\right] := bv_n\left(2^n - nat_n\left(\left[\!\left[t\right]\!\right]\right)\right)$
bvadd:	$\left[\!\left[{t_1}^{[n]} + {t_2}^{[n]}\right]\!\right] := bv_n\left(nat_n\left(\left[\!\left[t_1\right]\!\right]\right) + nat_n\left(\left[\!\left[t_2\right]\!\right]\right) \mod 2^n\right)$
bvsub:	$\left[\!\left[{t_1}^{[n]} - {t_2}^{[n]}\right]\!\right] := \left[\!\left[t_1 + (-t_2)\right]\!\right]$
bvmul:	$\left[\!\left[{t_1}^{[n]} \cdot {t_2}^{[n]}\right]\!\right] := bv_n\left(nat_n\left(\left[\!\left[t_1\right]\!\right]\right) \cdot nat_n\left(\left[\!\left[t_2\right]\!\right]\right) \mod 2^n\right)$
bvudiv:	$\left[\!\left[{t_1}^{[n]} /_{\mathbf{u}} {t_2}^{[n]}\right]\!\right] := bv_n\left(\left\lfloor nat_n\left(\left[\!\left[t_1\right]\!\right]\right) / nat_n\left(\left[\!\left[t_2\right]\!\right]\right)\right\rfloor\right)$
bvurem:	$\left[\!\left[\mathit{bvurem}\left({t_1}^{[n]}, {t_2}^{[n]}\right)\right]\!\right] := \left[\!\left[t_1 - (t_1 /_{\mathbf{u}} t_2) \cdot t_2\right]\!\right]$
bvsdiv:	$\left[\!\left[\mathit{bvsdiv}\left({t_1}^{[n]}, {t_2}^{[n]}\right)\right]\!\right]$ $:= \begin{cases} \left[\!\left[\mathit{abs}(t_1) /_{\mathbf{u}} \mathit{abs}(t_2)\right]\!\right] & \text{if } \mathit{msb}\,(t_1) = \mathit{msb}\,(t_2) \\ \left[\!\left[-(\mathit{abs}(t_1) /_{\mathbf{u}} \mathit{abs}(t_2))\right]\!\right] & \text{otherwise} \end{cases}$
bvsrem:	$\left[\!\left[\mathit{bvsrem}\left({t_1}^{[n]}, {t_2}^{[n]}\right)\right]\!\right]$ $:= \begin{cases} \left[\!\left[-\mathit{bvurem}\,(\mathit{abs}(t_1), \mathit{abs}(t_2))\right]\!\right] & \text{if } \mathit{msb}\,(t_1) \\ \left[\!\left[\mathit{bvurem}\,(\mathit{abs}(t_1), \mathit{abs}(t_2))\right]\!\right] & \text{otherwise} \end{cases}$
bvsmod:	$\left[\!\left[\mathit{bvsmod}\left({t_1}^{[n]}, {t_2}^{[n]}\right)\right]\!\right]$ $:= \begin{cases} \left[\!\left[\mathit{bvsrem}\,(t_1, t_2)\right]\!\right] & \text{if } \left[\!\left[\mathit{bvsrem}\,(t_1, t_2)\right]\!\right] = 0 \\ & \vee \mathit{msb}\,(t_1) = \mathit{msb}\,(t_2) \\ \left[\!\left[\mathit{bvsrem}\,(t_1, t_2) + t_2\right]\!\right] & \text{otherwise} \end{cases}$

Table 4.2: Semantics (interpretation) for common bit-vector operators

In Appendix 4.11, we use the notation $t^{[n]} \overline{\overline{=}}\ d$, where $d \in D_n$, as an alternative for $\left[\!\left[t^{[n]}\right]\!\right] = d$, assuming an appropriate model for t, implied by the context.

A formula Φ (over Σ_{Op}) is *satisfiable* over an interpretation $\widehat{Op}$ (of Σ_{Op}) if and only if there exists a model M for Φ such that $\left[\!\left[\Phi\right]\!\right]_M^{\widehat{Op}} = 1$. M is called a *satisfying model* for Φ over $\widehat{Op}$.

Definition 4.7 (Bit-blasting)**.** *Bit-blasting* (or flattening [158]) a bit-vector formula Φ means to construct an equisatisfiable Boolean formula ϕ. Φ and ϕ are *equisatisfiable* over an interpretation $\widehat{Op}$ if and only if the following condition holds: there exists a satisfying model for Φ over $\widehat{Op}$ if and only if there exists a satisfying assignment for ϕ.

Bit-blasting techniques represent bit-vector variables as strings of Boolean variables and encode bit-vector operations as corresponding Boolean circuits. It is a well-known fact that for all common operations, interpreted by $\widehat{\mathbf{Op}}$, a corresponding *polynomial-size* (in the bit-widths of operands) Boolean circuit can be constructed. This fact plays an important role in several of our proofs.

Logics and Encodings

For the rest of this chapter, we fix the operator set. We use $\mathbf{Op}$, with the signature $\mathbf{\Sigma_{Op}}$ (Table 4.1) and the interpretation $\widehat{\mathbf{Op}}$ (Table 4.2), and we refer to this framework as the *Common Operator Framework*.

By considering bitwise operators in the Boolean case (i.e., for bit-width 1) as logical connectives, the same separation of a Boolean level and a bit-vector level can be made in any bit-vector formula as in most approaches in the literature [19, 49, 51, 74, 96]. Note, however, that relational operations can occur not only at the Boolean level, but even below that, due to Definition 4.2, which allows any operations to be nested. In order to be compatible with the above-mentioned two-level approaches, we introduce a normal form for bit-vector formulas as follows:

Definition 4.8 (Flat Form)**.** A bit-vector formula Φ is in *flat form* if and only if it does not contain any nested relational operations.

It is easy to see that any bit-vector formula Φ can be translated into *flat form* with only linear growth in formula size. For each nested relational operation in Φ, iteratively replace the innermost one $o({t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]}, i_1, \ldots, i_l)$ by introducing a new (Tseitin) variable $ts^{[1]}$ existentially quantified at the innermost prefix position and adding the constraint $ts^{[1]} \Leftrightarrow o({t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]}, i_1, \ldots, i_l)$ to the formula (i.e., conjuncting it with the matrix).

In this chapter, we investigate the following four common bit-vector logics, as well as fragments and extensions thereof:

QF_BV: quantifier-free bit-vector formulas without uninterpreted functions;

QF_UFBV: quantifier-free formulas allowing uninterpreted functions;

BV: formulas allowing quantification, but no uninterpreted functions;

UFBV: formulas allowing quantification and uninterpreted functions.

We distinguish between logics that use a *unary* or a *binary* encoding on *scalars* appearing in formulas. Recall that binary encoding can be replaced with any other

logarithmic encoding. Note that a scalar can appear either as a bit-width or a scalar operand. The value c of a bit-vector constant $c^{[n]}$ is always encoded in binary format, since it represents a bit-vector.

Definition 4.9 (Logic with Unary and Binary Encoding)**.** Given a bit-vector logic $\mathcal{L}$, let $\mathcal{L}1$ and $\mathcal{L}2$ denote the logic $\mathcal{L}$ using unary and binary encoding on all the scalars in formulas, respectively.

In the rest of this chapter, we investigate the complexity of the satisfiability problem for QF_BV1, QF_UFBV1, BV1, UFBV1, QF_BV2, QF_UFBV2, BV2, and UFBV2. For this, we define the size of a formula.

Definition 4.10 (Formula Size)**.** Suppose we are given a bit-vector logic $\mathcal{L}$ and a formula $\Phi \in \mathcal{L}$, with $\Phi := Q_0 x_0{}^{[n_0]} Q_1 x_1{}^{[n_1]} \dots Q_k x_k{}^{[n_k]}.t^{[1]}$. The *size* of Φ is defined as $|\Phi| := \left|x_0{}^{[n_0]}\right| + \dots + \left|x_k{}^{[n_k]}\right| + \left|t^{[1]}\right|$.

The expression $|t^{[n]}|$ denotes the size of a term $t^{[n]}$ and is defined as follows:

	expression	size
constant:	$\left\|c^{[n]}\right\|$	$1 + \mathsf{L}(c+1) + enc_{\mathcal{L}}(n)$
variable:	$\left\|v^{[n]}\right\|$	$1 + enc_{\mathcal{L}}(n)$
operation:	$\left\|o\left(t_1{}^{[n_1]}, \dots, t_k{}^{[n_k]}, i_1, \dots, i_l\right)\right\|$	$1 + \left\|t_1{}^{[n_1]}\right\| + \dots + \left\|t_k{}^{[n_k]}\right\| + enc_{\mathcal{L}}(i_1) + \dots + enc_{\mathcal{L}}(i_l)$
uninterpreted function:	$\left\|f^{[n]}\left(t_1{}^{[n_1]}, \dots, t_k{}^{[n_k]}\right)\right\|$	$1 + enc_{\mathcal{L}}(n) + \left\|t_1{}^{[n_1]}\right\| + \dots + \left\|t_k{}^{[n_k]}\right\|$
scalar:	$enc_{\mathcal{L}}(n)$	$1+n$, if $\mathcal{L}$ uses unary encoding $1 + \mathsf{L}(n+1)$, if $\mathcal{L}$ uses binary encoding

4.4 Logics With Unary Encoding

First, we consider bit-vector logics with *unary encoding*. The results of this section can also be found in our previous work [151].

Without uninterpreted functions nor quantification, i.e., for QF_BV1, the following complexity result can be shown (for partial results and related work see also [19] and [51]):

Proposition 4.11. Q*F_BV*1 *is* NP*-complete.*[2]

Proof. Recall that QF_BV1 uses the Common Operator Framework. Therefore, by *bit-blasting*, QF_BV1 can be (polynomially) reduced to *Boolean formulas*, for

[2]This kind of result is often called unary NP-completeness [105].

which the satisfiability problem (SAT) is NP-complete. i.e., the class of Boolean formulas is a subset of QF_BV1. □

Adding uninterpreted functions to QF_BV1 does not increase complexity:

Proposition 4.12. Q*F_UFBV*1 *is* NP*-complete.*

Proof. In a quantifier-free formula, uninterpreted functions can be eliminated by replacing each occurrence with a new bit-vector variable and then adding (at most quadratic many) Ackermann constraints (see, e.g., [158, Chapter 3.3.1]). Therefore, QF_UFBV1 can be polynomially translated into QF_BV1. The other direction follows from the fact that QF_BV1 $\subset$ QF_UFBV1. □

Adding quantifiers to QF_BV1 yields the following complexity (see also [71]):

Proposition 4.13. B*V*1 *is* PSPACE*-complete.*

Proof. By applying bit-blasting, BV1 can be reduced to *Quantified Boolean Formulas* (QBF), which is PSPACE-complete. Hardness follows from the fact that QBF $\subset$ BV1 (following the same argument as in Proposition 4.11). □

Adding quantifiers to QF_UFBV1 increases complexity exponentially:

Proposition 4.14. U*FBV*1 *is* NEXPTIME*-complete (see [233]).*

Proof. The *Effectively Propositional Logic* (EPR) is NEXPTIME-complete [161], and can be reduced to UFBV1 [233, Theorem 7]. For completing the other direction, apply the reduction in [233, Theorem 7] combined with bit-blasting of the bit-vector operations. □

4.5 Scalar-Bounded Problems

For some of our remaining complexity results, we apply the concept of re-encoding scalars from binary to unary format. Due to the nature of these encodings, this process can lead to an exponential growth in formula size for the general case. However, this exponential growth can be avoided sometimes.

In [151], we introduced the concept of bit-width bounded bit-vector problems. In this section, we generalize this concept by introducing the concept of *scalar-boundedness*, a sufficient condition for bit-vector problems to remain in the "lower" complexity class, when re-encoding scalars from binary to unary format. This condition tries to capture the bounded nature of scalars in certain problems.

Note that, in any bit-vector formula, there has to be at least one scalar, due to the fact that there has to be at least one term with explicit specification of its bit-width (as a scalar).[3] Given a formula Φ, let $max_{\mathsf{scl}}(\Phi)$ denote the *maximal scalar* in Φ and, furthermore, let $cnt_{\mathsf{scl}}(\Phi)$ denote *the number of scalars* in Φ.

[3] Recall that only a variable, a constant, or an uninterpreted function can have *explicit bit-width*.

Definition 4.15 (Scalar-Bounded Formula Set)**.** An infinite set S of bit-vector formulas is *(polynomially) scalar-bounded*, if and only if there exists a polynomial function $p : \mathbb{N} \mapsto \mathbb{N}$ such that $\forall \Phi \in S.\ max_{\mathsf{scl}}(\Phi) \leqslant p(cnt_{\mathsf{scl}}(\Phi))$.

Proposition 4.16. *Given a scalar-bounded set S of formulas with binary encoded scalars, any $\Phi \in S$ grows polynomially when re-encoding the scalars to unary format.*

Proof. Let Φ' denote the formula obtained through re-encoding scalars in Φ to *unary* format. For the size of Φ', the following upper bound holds: $|\Phi'| \leqslant cnt_{\mathsf{scl}}(\Phi) \cdot max_{\mathsf{scl}}(\Phi) + |\Phi|$. Note that $cnt_{\mathsf{scl}}(\Phi) \cdot max_{\mathsf{scl}}(\Phi)$ is an upper bound on the sum over the sizes of all the scalars in Φ'. The second term, $|\Phi|$, represents an upper bound for the part of Φ that does not contain any scalars. Since S is *scalar-bounded*, it holds that

$$\begin{aligned} |\Phi'| &\leqslant cnt_{\mathsf{scl}}(\Phi) \cdot max_{\mathsf{scl}}(\Phi) + |\Phi| \\ &\leqslant cnt_{\mathsf{scl}}(\Phi) \cdot p(cnt_{\mathsf{scl}}(\Phi)) + |\Phi| \ \leqslant\ |\Phi| \cdot p(|\Phi|) + |\Phi| \end{aligned}$$

where p is a polynomial function. Therefore, the size of Φ' is *polynomial* in the size of Φ. □

By applying this proposition to the logics of Section 4.3.3, together with the results from Section 4.4, we get:

Corollary 4.17. *Suppose we are given a scalar-bounded set S of bit-vector formulas. If $S \subseteq$ Q*F_BV*2 (and even if $S \subseteq$ Q*F_UFBV*2), then $S \in$ NP. If $S \subseteq$ B*V*2, then $S \in$* PSPACE*. If $S \subseteq$ U*FBV*2, then $S \in$* NEXPTIME*.*

4.6 Quantifier-Free Logics with Binary Encoding

Our main contribution in [101, 151] was to give complexity results for bit-vector logics with the more common *binary encoding*, in the general case (i.e., for sets of formulas that are *not scalar-bounded*). In this section, we present modified versions of our proofs for the quantifier-free logics and restructured our results in order to give a better overall picture.

First, we introduce our main complexity results as theorems, starting with the full logic of QF_BV2 in Theorem 4.18, and continuing with three fragments of QF_BV2 in Theorems 4.19, 4.20, 4.21. All these theorems reference separate lemmas, which we introduce afterwards.

Theorem 4.18. Q*F_BV*2 *is* NEXPTIME*-complete [151].*

Proof. It is easy to see that QF_BV2 $\in$ NEXPTIME, since a QF_BV2 formula can be translated exponentially to QF_BV1 $\in$ NP (Proposition 4.11), by applying a simple unary re-encoding to all the scalars in the formula. NEXPTIME-hardness of

QF_BV2 is a direct consequence of Lemma 4.23, in which a fragment of QF_BV2 is proved to be NEXPTIME-hard. □

Note that UFBV1 and QF_BV2 have the same complexity. This shows that, informally speaking, binary encoding on scalars has the same expressive power as quantification and uninterpreted functions altogether.

In [101], we investigated the complexity of the satisfiability problem for the following three fragments of QF_BV2, which only allow a restricted set of bit-vector operations in formulas:

QF_BV2$_{\ll c}$: only bitwise operations, equality, and *left shift by constant*, i.e., expressions $t^{[n]} \ll c^{[n]}$ where c is a constant, are allowed.

QF_BV2$_{\ll 1}$: only bitwise operations, equality, and *left shift by 1*, i.e., expressions $t^{[n]} \ll 1^{[n]}$, are allowed.

QF_BV2$_{\mathsf{bw}}$: only bitwise operations and equality are allowed.

Theorem 4.19. Q*F_BV2*$_{\ll c}$ *is* NEXPTIME*-complete [101].*

Proof. In Lemma 4.23, we give a reduction from DQBF (which is NEXPTIME-complete) to QF_BV2$_{\ll c}$. This shows the NEXPTIME-hardness of QF_BV2$_{\ll c}$. The fact that QF_BV2$_{\ll c} \in$ NEXPTIME directly follows from Theorem 4.18. □

Theorem 4.20. Q*F_BV2*$_{\ll 1}$ *is* PSPACE*-complete [101].*

Proof. In Lemma 4.24, we give a reduction from QBF, being PSPACE-complete, to QF_BV2$_{\ll 1}$. This shows the PSPACE-hardness of QF_BV2$_{\ll 1}$. In Lemma 4.25, we then prove PSPACE-inclusion by giving a reduction from deciding satisfiability of QF_BV2$_{\ll 1}$ formulas to the *model checking* problem for sequential circuits. Symbolic model checking for sequential circuits is PSPACE-complete as well [194, 198, 205]. □

Also note that this theorem has an important practical aspect. It allows us to use symbolic model checkers (see the hardware model checking competition) for solving these restricted bit-vector problems instead of using SAT solvers after an exponential explosion through bit-blasting. This is further discussed in Section 4.9.

Theorem 4.21. Q*F_BV2*$_{\mathsf{bw}}$ *is* NP*-complete [101].*

Proof. Since *Boolean formulas* are a subset of QF_BV2$_{\mathsf{bw}}$, NP-hardness follows directly. To show that QF_BV2$_{\mathsf{bw}} \in$ NP, we give a reduction from QF_BV2$_{\mathsf{bw}}$ to a *scalar-bounded* set of formulas $S \subset$ QF_BV2 in Lemma 4.26. The claim then follows from Corollary 4.17. □

As already indicated in Proposition 4.12, adding uninterpreted functions to all quantifier-free logics we discussed so far does not affect complexity. We formalize this in the following proposition:

Proposition 4.22. Q*F_UFBV*2 *and* Q*F_UFBV*$2_{\ll c}$ *are* NEXPTIME*-complete,* Q*F_UFBV*$2_{\ll 1}$ *is* PSPACE*-complete,* Q*F_UFBV*2_{bw} *is* NP*-complete [101, 151].*

Proof. Apply the same arguments as were used in Proposition 4.12. □

As we outlined above, now we propose our main lemmas, referenced in the previous theorems.

Lemma 4.23. D*QBF can be reduced to* Q*F_BV*$2_{\ll c}$ *[101, 151].*

Proof. The basic idea is to use bit-vector expressions to encode function tables in an exponentially more succinct way, which then allows us to characterize independence of an existential variable from a particular universal variable in a polynomial way.

In the proof, we apply bit masks of the form

$$\mathit{binmagic}\,(2^m,\ 2^n) := \overbrace{\underbrace{0\ldots0}_{2^m}\ \underbrace{1\ldots1}_{2^m}\ldots\underbrace{0\ldots0}_{2^m}\ \underbrace{1\ldots1}_{2^m}}^{2^n}$$

Note that these bit masks correspond to the so-called *binary magic numbers* (or magic masks in [145, page 141]), and can arithmetically be calculated in the following way (actually as the result of a geometric sum):

$$\mathit{binmagic}\,(2^m,\ 2^n) \quad := \quad \frac{2^{(2^n)}-1}{2^{(2^m)}+1}$$

In order to reformulate this definition in terms of bit-vectors, (i) the numerator can be written as $\sim 0^{[2^n]}$, (ii) $2^{(2^m)}$ as $1 \ll 2^m$, and (iii) the resulting binary magic number as a bit-vector variable $b^{[2^n]}$:

$$\begin{aligned} b^{[2^n]} &= \sim 0^{[2^n]}\ /_{\mathbf{u}}\ \big((1 \ll 2^m)+1\big) \\ b\cdot\big((1 \ll 2^m)+1\big) &= \sim 0^{[2^n]} \\ (b \ll 2^m)+b &= \sim 0^{[2^n]} \end{aligned}$$

Addition can be eliminated easily as follows, by using two's complement representation for -1 and $-b$:

$$\begin{aligned} (b \ll 2^m)+b &= -1 \\ b \ll 2^m &= -1-b \\ b \ll 2^m &= -1+\sim b+1 \\ b \ll 2^m &= \sim b \end{aligned}$$

We now use the binary magic numbers to create a certain set of fully-specified exponential-size bit-vectors by using a polynomial expression, due to binary encoding on scalars. Afterwards, we then formally point out the well-known fact that

those bit-vectors correspond exactly to the set of *all assignments*. By adding constraints on those bit-vectors, we can then use a polynomial-size bit-vector formula for *cofactoring Skolem-functions* in order to express independency constraints.

First, we describe the reduction, then we show that the reduction is polynomial, and, finally, that it is correct. An example can be found in Appendix 4.11.1.

The Reduction. Let $\psi := Q.\phi$ denote a DQBF with quantifier prefix Q and matrix ϕ. Furthermore, let $u_0, \ldots, u_{n-1}$ and $e_0, \ldots, e_{n'-1}$ denote all the universal and existential variables that occur in Q, respectively. Translate ψ to a QF_BV2$_{\ll c}$ formula Φ by eliminating the quantifier prefix and translating the matrix ϕ as follows:

Step 1. Replace all Boolean constants 0 and 1 with $0^{[2^n]}$ and $\sim 0^{[2^n]}$, all Boolean universal variables u_m and existential variables $e_{m'}$ with bit-vector variables ${U_m}^{[2^n]}$ and ${E_{m'}}^{[2^n]}$, and all logical connectives with corresponding bitwise bit-vector operators (e.g., $\wedge$ with $\&$). Let $t^{[2^n]}$ denote the bit-vector term generated so far. Extend it to the formula $t = \sim 0^{[2^n]}$. We refer to this as Φ_0.

Step 2. We now construct Φ_1 by adding new constraints to Φ_0. For each $u_m \in \{u_0, \ldots, u_{n-1}\}$, in order to assign a binary magic number to U_m, add the following equality (i.e., conjunct it with the current formula):

$$U_m \ll 2^m \quad = \quad \sim U_m$$

Step 3. Next, we construct Φ_2 by adding another set of constraints to Φ_1. For each existential variable $e_{m'} \in \{e_0, \ldots, e_{n'-1}\}$, depending on the universal variables $Deps(e_{m'}) \subseteq \{u_0, \ldots, u_{n-1}\}$, and for each $u_m \notin Deps(e_{m'})$, add the following equality:

$$E_{m'} \& \sim U_m \quad = \quad (E_{m'} \ll 2^m) \& \sim U_m \tag{4.1}$$

Finally, we define $\Phi := \Phi_2$.

Polynomiality. Note that all the scalars and constants in Φ are encoded in *binary* form. Therefore, exponential bit-widths and constants (2^n and 2^m) are encoded into linear many (n and m) binary digits. We now show that each reduction step results only in polynomial growth of the formula size.

Step 1 may introduce additional bit-vector constants to the formula and adds variables $U_m^{[2^n]}, E_{m'}^{[2^n]}$. The total number of elements is bounded by the size of the input. All bit-widths are 2^n and, therefore, the resulting formula is bounded quadratically in the input size. *Step 2* adds n equalities as constraints. Again, all bit-widths are 2^n. Thus, the size of the added constraints is bounded quadratically in the input size. *Step 3* adds at most n constraints for each existential variable. All bit-widths are 2^n. Therefore, the size is bounded cubically in the input size.

Correctness. In order to show that the original DQBF ψ and the resulting bit-vector formula Φ are equisatisfiable, we consider the individual steps separately.

In *Step 1*, we used the matrix ϕ of ψ to create a bit-vector formula with the same underlying structure which is true if and only if each row evaluates to 1. Since all the bits of bit-vectors in Φ_0 are independent of each other and there are no additional constraints on the bit-vector variables, Φ_0 is satisfiable if and only if the Boolean formula ϕ is satisfiable.

Now consider the bit-vector variables U_m after constructing Φ_1 by adding the constraints of *Step 2*. In the following, we formalize the well-known fact that the combination of all the U_ms corresponds exactly to *all possible assignments to the universal variables of* ψ. By construction, all bits of U_m are fixed to some constant value. Additionally, for every bit-index $b_i \in [0, 2^n - 1]$, there exists a bit-index $b_j \in [0, 2^n - 1]$ such that

$$[\![U_m]\!][b_i] \neq [\![U_m]\!][b_j] \quad \text{and} \tag{4.2a}$$

$$[\![U_k]\!][b_i] = [\![U_k]\!][b_j]\,, \ \forall k \neq m. \tag{4.2b}$$

Actually, we can define b_j in the following way (considering the 0th bit to be the least significant):

$$b_j \quad := \quad \begin{cases} b_i - 2^m & \text{if } [\![U_m]\!][b_i] = 0 \\ b_i + 2^m & \text{if } [\![U_m]\!][b_i] = 1 \end{cases}$$

By defining b_j this way, Equation (4.2a) and (4.2b) both hold, which can be seen as follows. Let $R(c, l)$ be the bit-vector of length l with each bit set to the Boolean constant c. Equation (4.2a) holds, since, due to construction, U_m consists of 2^{n-1-m} concatenated bit-vector fragments $0 \ldots 01 \ldots 1 = R(0, 2^m)R(1, 2^m)$ (with both 2^m zeros and 2^m ones). Therefore, it is easy to see that

$$[\![U_m]\!][b_i] \neq [\![U_m]\!][b_i - 2^m] \text{ and } [\![U_m]\!][b_i] \neq [\![U_m]\!][b_i + 2^m] \text{ holds if}$$
$$[\![U_m]\!][b_i] = 0 \text{ and } [\![U_m]\!][b_i] = 1, \text{ respectively.}$$

With a similar argument, we can show that Equation (4.2b) holds:

$$[\![U_k]\!][b_i] = [\![U_k]\!][b_i - 2^m] \text{ and } [\![U_k]\!][b_i] = [\![U_k]\!][b_i + 2^m] \text{ holds if}$$
$$[\![U_k]\!][b_i] = 0 \text{ and } [\![U_k]\!][b_i] = 1, \text{ respectively,}$$

since $b_i - 2^m$ and $b_i + 2^m$ are located either still in the same half or already in a concatenated copy of a $R(0, 2^k)R(1, 2^k)$ fragment, if $k \neq m$.

Now, consider all possible assignments to the universal variables of our original DQBF ψ. For a given assignment $\alpha \in \{0, 1\}^n$, the existence of such a previously defined b_j for every U_m and b_i allows us to iteratively find a b_α such that $\left([\![U_0]\!][b_\alpha], \ldots, [\![U_{n-1}]\!][b_\alpha] \right) = \alpha$. Thus, we have a *bijective mapping* from the *universal assignments* α for ψ to the *bit-indices* b_α for Φ_1. Up to this point, each bit-vector $E_{m'}$ can basically still take $2^{(2^n)}$ different values in Φ_1. The value of each individual bit $[\![E_{m'}]\!][b_\alpha]$ corresponds to the value that $e_{m'}$ takes under a

given universal assignment $\alpha \in \{0,1\}^n$. Note that, without any further restriction, there is no connection between the different bits of $E_{m'}$ and, therefore, the bit-vector represents an arbitrary Skolem-function for $e_{m'}$. It may have different values for all universal assignments and thus would allow $e_{m'}$ to depend on all universal variables. Consequently, Φ_1 is satisfiable if and only if the QBF $\forall u_1, \ldots, u_{n-1} \exists e_1, \ldots, e_{n'-1}.\phi$ is satisfiable.

In *Step 3*, we rule out all those assignments to the $E_{m'}$s that correspond to Skolem-functions which do not respect the dependency scheme of ψ. Whenever $e_{m'}$ does not depend on a universal variable u_m, we add the constraint of Equation (4.1). In DQBF, independence can be formalized in the following way: $e_{m'}$ does not depend on u_m if $e_{m'}$ has to take the same value in the case of all pairs of universal assignments $\alpha, \beta \in \{0,1\}^n$ where $\alpha[k] = \beta[k]$ for all $k \neq m$. Exactly this is enforced by our constraint. Looking at the corresponding bit-indices b_α and b_β for α and β, respectively, our constraint for independence ensures that $[\![E]\!][b_\alpha] = [\![E]\!][b_\beta]$. More precisely, Equation (4.1) ensures that the positive and negative cofactors of the Skolem-function for $e_{m'}$ with respect to an independent variable u_m have the same value. Having added those constraints, Φ_2 is now respecting the dependency scheme and therefore Φ is satisfiable if and only if the original DQBF ψ is satisfiable. □

Lemma 4.24. *QBF can be reduced to QF_BV2$_{\ll 1}$ [101].*

Proof. To show the PSPACE-hardness of QF_BV2$_{\ll 1}$, we give a reduction from QBF, similar to the one from DQBF to QF_BV2$_{\ll c}$ that we used in Lemma 4.23.

For our reduction, we again use the *binary magic numbers*. Note that, in Lemma 4.23, we used *left shift by constant* to construct the binary magic numbers. This is not permitted in QF_BV2$_{\ll 1}$. We therefore give an alternative construction of the binary magic numbers using only *bitwise operations*, *equality*, and *left shift by 1*.

Let ${b_0}^{[2^n]}, \ldots, {b_{n-1}}^{[2^n]}$ be n initially unconstrained bit-vector variables. By adding certain constraints, we ensure that the only possible value the variables can take are those of the binary magic numbers. For the following argument, consider the bit-vector variables ${b_0}^{[2^n]}, \ldots, {b_{n-1}}^{[2^n]}$ as column vectors in a matrix $B^{[2^n \times n]}$. Written next to each other in this way, the matrix formed by the binary magic numbers would be uniquely determined by the following property: If each row of B is interpreted as a number $0 \leqslant c < 2^n$ in binary representation, the next row is equal to $c+1$. The rows of B therefore represent a counter from 0 to 2^{n-1}. We can capture this fact by adding the following n constraints, with $m \in \{0, \ldots, n-1\}$:

$$\Big(\bigwedge_{0 \leqslant i < m} b_i \Big) \oplus b_m \quad = \quad b_m \ll 1$$

The left side of each constraint considers one specific column of B (i.e., one index of the counter) and the value of each position will change if and only if all columns to the right are equal to 1 (i.e., the lower indices of the counter generate

an overflow). In this sense, the left sides of all constraints increment the counter value corresponding to a row of B. The right sides of all constraints ensure that the incremented counter value is placed in the next row of B.

As already mentioned, we now give the reduction which is similar to the one in Lemma 4.23. An example can be found in Appendix 4.11.2.

The Reduction. Let $\psi := Q.\phi$ denote a QBF with quantifier prefix Q and matrix ϕ. Since ψ is a QBF (in contrast to DQBF in Lemma 4.23), we know that Q defines a total order on the universal variables. We assume the universal variables $u_0, \ldots, u_{n-1}$ of ϕ are ordered according to their appearance in Q, with u_0 and u_{n-1} being the innermost and outermost variable, respectively. Translate ψ to a QF_BV2$_{\ll 1}$ formula Φ by eliminating the quantifier prefix and translating the matrix as follows:

Step 1. Replace all Boolean constants 0 and 1 with $0^{[2^n]}$ and $\sim 0^{[2^n]}$, all Boolean universal variables u_m and existential variables $e_{m'}$ with bit-vector variables ${U_m}^{[2^n]}$ and ${E_{m'}}^{[2^n]}$, and all logical connectives with corresponding bitwise bit-vector operators (e.g., $\wedge$ with $\&$). Let $t^{[2^n]}$ denote the bit-vector term generated so far. Extend it to the formula $t = \sim 0^{[2^n]}$. We refer to this as Φ_0.

Step 2. We now construct Φ_1 by adding new constraints to Φ_0. For each universal variable $u_m \in \{u_0, \ldots, u_{n-1}\}$, in order to assign a binary magic number to ${U_m}^{[2^n]}$, add the following equality (i.e., conjunct it with the current formula):

$$\Big(\bigwedge_{0 \leqslant i < m} U_i \Big) \oplus U_m \quad = \quad U_m \ll 1$$

Step 3. Next, we construct Φ_2 by adding another set of constraints to Φ_1. For each existential variable $e_{m'} \in \{e_0, \ldots, e_{n'-1}\}$ depending on the universal variables $Deps(e_{m'}) = \{u_m, \ldots, u_{n-1}\}$, with u_m being the innermost universal variable that $e_{m'}$ depends on, check the following conditions:

if $Deps(e_{m'}) = \varnothing$, add the equality:

$$E_{m'} \& \sim 1 \quad = \quad E_{m'} \ll 1 \tag{4.3}$$

otherwise, if $m \neq 0$, add the two equalities:

$$U'_m \quad = \quad \sim \big((U_m \ll 1) \oplus U_m \big) \tag{4.4}$$

$$E_{m'} \& U'_m \quad = \quad (E_{m'} \ll 1) \& U'_m \tag{4.5}$$

Finally, we define $\Phi := \Phi_2$.

Step 1 and *Step 2* are equal to those of Lemma 4.23 apart from the fact that a different construction for the *binary magic numbers* is used.

Again, each bit-index of Φ corresponds to the evaluation of ψ under a specific assignment to the universal variables $u_0, \ldots, u_{n-1}$, and, by construction of ${U_0}^{[2^n]}, \ldots, {U_{n-1}}^{[2^n]}$, all possible assignments are considered. Equation (4.4) creates a bit-vector ${U'_m}^{[2^n]}$ for which each bit equals to 1 if and only if the corresponding universal variable changes its value from one universal assignment to the next. In contrast to Lemma 4.23, this can now only be done for neighbouring bit-indices since we are only allowed to use *left shift by* 1 instead of arbitrary constants in *Step 3*. For QBF, this is sufficient because Q defines a total order on the universal variables.

Of course, Equation (4.4) does not have to be added multiple times, if several existential variables depend on the same universal variable. Equation (4.5) and Equation (4.3) ensure that the corresponding bits of ${E_{m'}}^{[2^n]}$ satisfy the dependency scheme of ψ by only allowing the value of $e_{m'}$ to change if an outer universal variable takes a different value. If $Deps(e_{m'}) = \{u_0, \ldots, u_{n-1}\}$, i.e., if $e_{m'}$ depends on all universal variables, Equation (4.4) evaluates to $U'_0 = 0^{[2^n]}$, and, as a consequence, Equation (4.5) simplifies to $true$. Because of this, no constraints need to be added for $m = 0$.

A similar approach used for translating QBF to Symbolic Model Verification (SMV) can be found in [84]. See also [194] for a translation from QBF to sequential circuits. □

Lemma 4.25. Q*F_BV*$2_{\ll 1}$ *can be reduced to sequential circuits [101].*

Proof. In [208, 209], the authors give a polynomial translation from quantifier-free Presburger arithmetic with bitwise operations (QFPAbit [200]) to sequential circuits. While they deal with *non-fixed-size* bit-vectors, we focus on *fixed-size* bit-vectors but share the goal of avoiding the exponential explosion due to explicit state representation as for example used in MONA [144]. We can adopt their approach in order to construct a translation for QF_BV$2_{\ll 1}$. Related work, introducing an automata-based representation for Presburger Arithmetic (without bitwise operations), can be found in [235].

For the most part, the basic structure as well as the arguments used throughout the reduction are the same as in [208, 209]. To keep the proof compact, we therefore focus on pointing out the changes compared to their earlier work and regularly refer to [208, 209] for the technical details.

As mentioned, the main difference between QFPAbit and QF_BV$2_{\ll 1}$ is the fact that bit-vectors of arbitrary, non-fixed, size are allowed in QFPAbit while all bit-vectors contained in QF_BV$2_{\ll 1}$ have a fixed bit-width. We now give the reduction.

Given $\Phi \in$ QF_BV$2_{\ll 1}$ in *flat form*, let $x^{[n]}, y^{[n]}$ denote bit-vector variables, $c^{[n]}$ a bit-vector constant, and ${t_1}^{[n]}, {t_2}^{[n]}$ bit-vector terms only containing bit-vector variables and bitwise operations. Following [208, 209], we also assume, w.l.o.g., that Φ only consists of logical combinations of three types of *atomic expressions*: ${t_1}^{[n]} = {t_2}^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \ll 1^{[n]}$. Similar to generating a formula

in *flat form* (Definition 4.8), it is easy to see that any QF_BV2$_{\ll 1}$ formula can be written like this with only linear growth in size by introducing Tseitin variables.

We then encode each equality in Φ into an individual *sequential circuit* separately. In the following, those are referred to as *atomic sequential circuits*. Compared to [208, 209], two modifications for the construction of an atomic sequential circuits are needed. First, we need to give a translation of $x = y \ll 1$ to sequential circuits. This can be done, for example, by using the sequential circuit for $x = 2 \cdot y$ in QFPAbit. The second modification relates to dealing with *fixed-size* bit-vectors. Let n be the bit-width of all bit-vectors in a given atomic expression. We extend each atomic sequential circuit to include a counter (circuit). The counter initially is set to 0 and is incremented by 1 in each clock cycle up to a value of n. When the counter reaches a value of n, the counter as well as the original atomic sequential circuit keep their value during all remaining cycles. In this way, their output also remains the same during all following cycles.

Using D-type flip-flops, as being part of the definition of sequential circuits in Section 4.3.2, this can be easily realized by adding a combinatorial part: Assume that the counter consists of k bits, represented by flip-flops $c_0, \ldots, c_{k-1}$ with outputs $o_0, \ldots, o_{k-1}$, respectively. Checking whether the counter has reached a value of n can be realized by a Boolean function $f(o_0, \ldots, o_{k-1})$, represented as a combinatorial circuit. Furthermore, let c denote the flip-flop of the original atomic sequential circuit and let o and i (which again can be an arbitrary function) denote its output and its input, respectively. We now replace the input i by a combinatorial circuit realizing the function

$$(f(o_0, \ldots, o_{k-1}) \wedge o) \vee (\neg f(o_0, \ldots, o_{k-1}) \wedge i)$$

This forces c to use its own output as its input if the counter has reached a value of n, and use its regular input otherwise. The counter flip-flops $c_1, \ldots, c_k$ will be forced to stabilize after n has been reached in the same way. Note that a counter like this can be realized with Ln gates, i.e., polynomially in the size of Φ. For a practical implementation, it is of course not necessary to introduce separate counters for each atomic sequential circuit. Instead, one counter can be used to address all atomic sequential circuits. However, concerning our complexity result, this obviously makes no difference.

In contrast to the implementation described in [208], we assume that the input streams for all variables start with the least significant bit. As already pointed out by the authors in [208], their choice was arbitrary and it is no more complicated to construct the circuits the other way around.

Finally, after constructing all atomic sequential circuits, their outputs are combined by logical gates following the Boolean structure of Φ, in the same way as for non-fixed bit-width in [208, 209]. Due to the counters being part of the atomic sequential circuits, we ensure that for every input stream x_i, that represents a bit-vector variable of bit-width n_i, only the first n_i bits of x_i influence the result of the whole circuit. □

Lemma 4.26. Q*F_BV*$2_{\mathsf{bw}} \in$ NP *[101].*

Proof. To show that QF_BV$2_{\mathsf{bw}} \in$ NP, we give a reduction from QF_BV2_{bw} to a *scalar-bounded* set of formulas S. With $S \subset$ QF_BV2, the claim then follows from Corollary 4.17. An example, that combines further results from Section 4.7.2, can be found in Appendix 4.11.3.

Suppose we are given a formula $\Phi \in$ QF_BV2_{bw} in *flat form* (Definition 4.8). We assume that any *disequality* ${t_1}^{[n]} \neq {t_2}^{[n]}$ in Φ is expressed by $\sim\left({t_1}^{[n]} = {t_2}^{[n]}\right)$. If Φ contains any constants $c^{[m]}$ where $c \neq 0$, we remove those constants in a (polynomial) preprocessing step. Let ${c_{\mathsf{max}}}^{[m]} := b_{k-1} \ldots b_1 b_0$ be the largest constant in Φ denoted in binary representation with $b_{k-1} = 1$ and arbitrary bits $b_{k-2}, \ldots, b_0$. We now replace each equality ${t_1}^{[n]} = {t_2}^{[n]}$, in Φ with

$$ {t_{1,0}}^{[1]} = {t_{2,0}}^{[1]} \ \wedge \ \ldots \ \wedge \ {t_{1,n-1}}^{[1]} = {t_{2,n-1}}^{[1]}, $$

if $n \leqslant k$. Otherwise, if $n > k$, we instead replace ${t_1}^{[n]} = {t_2}^{[n]}$ with

$$ {t_{1,0}}^{[1]} = {t_{2,0}}^{[1]} \ \wedge \ \ldots \ \wedge \ {t_{1,k-1}}^{[1]} = {t_{2,k-1}}^{[1]} \ \wedge \ {t_{\mathrm{HI}_1}}^{[n-k]} = {t_{\mathrm{HI}_2}}^{[n-k]}. $$

For $0 \leqslant i < \min\{n, k\}$, we use ${t_{1,i}}^{[1]} = {t_{2,i}}^{[1]}$ to express the *i*th row of the original equality. For constructing the terms ${t_{1,i}}^{[1]}$ and ${t_{2,i}}^{[1]}$, (i) replace each occurrence of a variable $x^{[n]}$ with the variable $x_i{}^{[1]}$, and (ii) replace each constant $c^{[n]}$ with $0^{[1]}$ if the *i*th bit of c is 0, and with $\sim 0^{[1]}$ otherwise.

In a similar way, if $n > k$, ${t_{\mathrm{HI}_1}}^{[n-k]} = {t_{\mathrm{HI}_2}}^{[n-k]}$ represents the remaining $n - k$ rows of the original equality corresponding to the most significant bits. For constructing ${t_{\mathrm{HI}_1}}^{[n-k]}$ and ${t_{\mathrm{HI}_2}}^{[n-k]}$, (i) replace each occurrence of a variable $x^{[n]}$ with the variable $x_{\mathrm{HI}}{}^{[n-k]}$, and (ii) replace each constant $c^{[n]}$ with $0^{[n-k]}$.

Since this preprocessing step is logarithmic in the value of c_{max}, it is polynomial in $|\Phi|$. W.l.o.g., we now assume that Φ does not contain any bit-vector constants different from $0^{[n]}$.

We now construct a formula Φ' by reducing the bit-widths of all bit-vector terms in Φ. We use $\mathit{cnt}_{\mathsf{eq}}(\Phi)$ to denote the number of equalities in Φ. Each term $t^{[n]}$ in Φ is then replaced with a term $t^{[n']}$, with $n' := \min\{n, \mathit{cnt}_{\mathsf{eq}}(\Phi)\} \leqslant |\Phi|$. Apart from this, Φ' is exactly the same as Φ. As a consequence, $\mathit{max}_{\mathsf{scl}}(\Phi') \leqslant |\Phi|$. The set of formulas constructed in this way is scalar-bounded according to Definition 4.15. To complete our proof, we now have to show that the proposed reduction is sound, i.e., out of every satisfying assignment to the bit-vector variables ${x_1}^{[n_1]}, \ldots, {x_k}^{[n_k]}$ for Φ we can also construct a satisfying assignment to ${x_1}^{[n'_1]}, \ldots, {x_k}^{[n'_k]}$ for Φ' and vice versa.

It is easy to see that whenever we have a satisfying assignment α' for Φ', we can construct a satisfying assignment α for Φ. This can be done by simply setting all additional bits of all bit-vector variables to the same value as the most significant bit of the corresponding original vector, i.e., by performing a signed extension. Since all equalities still evaluate to the same value under the extended assignment, $\alpha(F) = \alpha'(F')$ for all equalities F and F' of Φ and Φ', respectively. As a direct

consequence, $\alpha(\Phi) = \alpha'(\Phi') = 1$. The other direction needs slightly more reasoning. Given α, with $\alpha(\Phi) = 1$, we need to construct α', with $\alpha'(\Phi') = 1$. Again, we want to ensure that $\alpha'(F') = \alpha(F)$ for all equalities F and F' in Φ and Φ', respectively.

In each variable ${x_i}^{[n_i]}$, $i \in \{1, \ldots, k\}$, we select some of the bits. For each equality F with $\alpha(F) = 0$, we select a bit-index as a witness for its evaluation. If $\alpha(F) = 1$, we select an arbitrary bit-index. We then mark the selected bit-index in all bit-vector variables contained in F, as well as in all other bit-vector variables of the same bit-width. Having done this for all equalities, we end up with sets M_i of selected bit-indices, for all $i \in \{1, \ldots, k\}$, where

$$\begin{aligned} |M_i| &\leqslant \min\{n_i, \mathit{cnt}_{\mathsf{eq}}(\Phi)\} \\ M_i &= M_j \qquad \forall j \in \{1, \ldots, k\} \text{ with } n_i = n_j \end{aligned}$$

The selected indices contain a witness for the evaluation of each equality. We now add arbitrary further bit-indices, again selecting the same indices in bit-vector variables of the same bit-width, until $|M_i| = \min\{n_i, \mathit{cnt}_{\mathsf{eq}}(\Phi)\}\ \forall i \in \{1, \ldots, k\}$.

Finally, we can directly construct α' using the selected indices and, by doing so, we know that $\alpha'(\Phi') = \alpha(\Phi) = 1$, because of the fact that we included a witness for every equality in our index-selection process. Note that we only had to choose a specific witness for the case that $\alpha(F) = 0$. For $\alpha(F) = 1$, we were able to choose an arbitrary bit-index because every satisfied equality is obviously still satisfied when only a subset of all bit-indices is considered. □

Remark 4.27. A similar proof can be found in [140, 141]. While the focus of [140, 141] lies on improving the practical efficiency of SMT solvers by reducing the bit-width of a given formula before bit-blasting, the author does not investigate its influence on the complexity of a given problem class. In fact, the author claims that bit-vector theories with common operations are NP-complete. As we have already shown, this only holds if unary encoding on scalars is used. However, unary encoding leads to the fact that the given class of formulas remains NP-complete, independent of whether a reduction of the bit-width is possible. While the arguments on bit-width reduction given in [140, 141] still hold for binary encoded bit-vector formulas when only bitwise operations are used, our proof considers the effect on the complexity of the problem class.

4.7 Extensions and Alternative Characterizations

In this section, we investigate possible extensions to the fragments we have been dealing with so far and give alternative characterizations of specific logics. We use the term *base operations* to refer to the operations that we previously selected to define a certain class of bit-vector problems. Considering the complexity results

from the previous section, we know that the specific sets of base operations are sufficient to guarantee certain completeness results. This leads towards two potential directions of analysis.

On the one hand, it is interesting to see which common operations could be added to a fragment without increasing the complexity of the satisfiability problem. With QF_BV2$_{\ll c}$ being NEXPTIME-complete, any common operation can extend this fragment without increasing complexity; the full extension is exactly the definition of QF_BV2. It is more interesting to investigate which operations can be added to QF_BV2$_{\mathsf{bw}}$ and QF_BV2$_{\ll 1}$ while still remaining in NP and PSPACE, respectively. In order to check this, we present several reductions of additional operations to base operations.

On the other hand, it is also interesting to explore possible reductions of base operations to additional ones. We showed that satisfiability for QF_BV2$_{\mathsf{bw}}$, i.e., when *bitwise operations* and *equality* are used as base operations, is NP-complete. Using *left shift by 1* or *left shift by constant* as an additional base operation directly causes the satisfiability problem to become PSPACE-hard (Lemma 4.24) or NEXPTIME-hard (Lemma 4.23), respectively. If it is possible to show that any of these two base operations can be reduced to another operation o (together with bitwise operations and equality), then o can be considered as an alternative base operation, ensuring the satisfiability problem to remain hard for the specific complexity class.

4.7.1 Notation

Note that, since binary encoding is used on scalars, all the translations of operations must be *logarithmic* in the bit-widths of operands, in order to ensure that a reduction is polynomial in the formula size.

For describing our reductions, we often use the following form:

$term_1$		
replace with:	$term_2$	,
add assertion(s):	$formula_1$	
	$\vdots$	
	$formula_k$	

By this description, we want to express that we replace a term $term_1$ in a formula Φ with $term_2$, and simultaneously add all the quantifier-free formulas $formula_1, \ldots, formula_k$ to Φ (i.e., conjunct each of them with the matrix of Φ). We call $formula_1, \ldots, formula_k$ the *assertions* in the definition. All the variables that do not occur in $term_1$, but do occur in any of the expressions $term_2$, $formula_1, \ldots, formula_k$ are considered as Tseitin variables, i.e., they are assumed to be added to Φ as *new existential variables* at the innermost prefix position.

Let us note that, in our fragments, it is sufficient to use a *minimal functionally complete set* of bitwise operations, e.g., *bvnand* alone.

By bitwise operations and equality, functional if-then-else (*ite*) can be expressed easily, as follows. Note that, in order to avoid exponential blowup, a Tseitin variable x is introduced for the Boolean condition:

$ite\left({t_1}^{[1]}\ ,\ {t_2}^{[n]}\ ,\ {t_3}^{[n]}\right)$		
replace with:	$y^{[n]}$	,
add assertions:	$x^{[1]} = t_1$	
	$x \Rightarrow y = t_2$	
	$\neg x \Rightarrow y = t_3$	

4.7.2 QF_BV2_{bw}

Let us introduce the operation *indexing* $t^{[n]}[i]$, which is defined as $t[i:i]$, i.e., a special case of *extraction*. Although, in Section 4.7.4, we show that adding *extraction* makes the fragment NEXPTIME-hard, QF_BV2_{bw} can be extended with *indexing* without growth in complexity.

Theorem 4.28. *QF_BV*2_{bw} *extended by indexing is in* NP.

Proof. To show this, we extend the proof of Lemma 4.26 by an additional preprocessing step even before removing the non-zero constants. Suppose we are given a formula $\Phi \in$ QF_BV2_{bw}, also containing expressions $t^{[n]}[i]$. Let

$$I := \{i \mid t^{[n]}[i] \text{ appears in } \Phi\}$$

be the set of all indices that appear explicitly in the formula. We can assume that $I = \{i_1, \ldots, i_m\}$, with $i_l < i_{l+1}$, $\forall l \in \{1, \ldots, m-1\}$. After extracting those bit-indices from Φ, we explicitly encode the corresponding bits into Boolean variables, by translating Φ in a similar way as in Lemma 4.26. Consider three different kinds of terms in the following order:

1. Terms $t^{[n]}[i]$ are replaced by ${t_i}^{[1]}$.
2. Terms $t^{[1]}$ remain in the formula as they are.
3. Any other term has a bit-width $n > 1$. Therefore, we know that it can only occur as part of an equality ${t_1}^{[n]} = {t_2}^{[n]}$. We define

   $$l' := |\{l \in \{1, \ldots, m\} \mid i_l < n\}|$$

   as the number of explicitly specified indices smaller than n. Now, similar to Lemma 4.26, replace each equality ${t_1}^{[n]} = {t_2}^{[n]}$ with

   $$({t_{1,0}}^{[1]} = {t_{2,0}}^{[1]}) \ \wedge \ \ldots \ \wedge \ ({t_{1,n-1}}^{[1]} = {t_{2,n-1}}^{[1]}),$$

if $n = l'$. Otherwise, if $n > l'$, replace ${t_1}^{[n]} = {t_2}^{[n]}$ with

$$\left(\bigwedge_{l \in \{1,\ldots,l'\}} ({t_{1,i_l}}^{[1]} = {t_{2,i_l}}^{[1]}) \right) \wedge \ {t_{\mathrm{REM}_1}}^{[n-l']} = {t_{\mathrm{REM}_2}}^{[n-l']}.$$

As in Lemma 4.26, we use ${t_{1,i}}^{[1]} = {t_{2,i}}^{[1]}$ to express the ith row of the original equality. In the same way, ${t_i}^{[1]}$, being introduced for an *indexing*, represents the ith bit of t. The new terms $t_{1,i}$, $t_{2,i}$, and t_i are constructed in the same way as in Lemma 4.26.

Similarly, if $n > l'$, the expression ${t_{\mathrm{REM}_1}}^{[n-l']} = {t_{\mathrm{REM}_2}}^{[n-l']}$ represents the remaining $n - l'$ rows of the original equality corresponding to the indices that have not been extracted explicitly. Those terms are again constructed in the same way as in Lemma 4.26, except for the construction of new constants: each constant ${c}^{[n]}$ is replaced with a new constant ${c_{\mathrm{REM}}}^{[n-l']}$ by setting the jth bit of c_{REM} to the value of the kth bit of c, for $k := \min\{k' \mid |\{1,\ldots,k'\} \setminus I| = j\}$.

After this translation, the resulting formula Φ' does not contain indexing operations anymore and is equisatisfiable to the original one. Also, $|\Phi'| \leqslant p(|\Phi|)$ for some polynomial p, since the growth in size is bounded by the number of occurrences of the indexing operation in Φ. Note that this reduction is only possible because there is no interaction between different bit-indices, i.e., because Φ only contains bitwise operations and equality, apart from indexing. □

Similarly, extending QF_BV2$_{\mathsf{bw}}$ with additional relational operations from Table 4.1 does not increase complexity, either.

Theorem 4.29. Q*F_BV*2$_{\mathsf{bw}}$ *extended by relational operations from Table 4.1 is in* NP.

Proof. We give a reduction for the relational operation *unsigned less than* (bvult). The remaining relational operations in Table 4.1 can be reduced in a similar way. Given $\Phi \in$ QF_BV2$_{\mathsf{bw}}$ (without indexing), additionally containing expressions ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$, we adopt the proof of Lemma 4.26 in three ways.

First, the elimination of constants has to be modified. Again, let us define $c_{\mathsf{max}} := b_{k-1} \ldots b_1 b_0$ to be the largest constant in Φ denoted in binary representation with $b_{k-1} = 1$ and arbitrary bits $b_{k-2}, \ldots, b_0$. W.l.o.g., assume $n > k$. We now replace each relation ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ in Φ with

$$\begin{aligned}
& ({t_{\mathrm{HI}_1}}^{[n-k]} <_{\mathbf{u}} {t_{\mathrm{HI}_2}}^{[n-k]}) \\
\vee \quad & ({t_{\mathrm{HI}_1}}^{[n-k]} = {t_{\mathrm{HI}_2}}^{[n-k]}) \wedge (\neg {t_{1,k-1}}^{[1]} \wedge {t_{2,k-1}}^{[1]}) \\
\vee \quad & \ldots \\
\vee \quad & ({t_{\mathrm{HI}_1}}^{[n-k]} = {t_{\mathrm{HI}_2}}^{[n-k]}) \wedge ({t_{1,k-1}}^{[1]} \Leftrightarrow {t_{2,k-1}}^{[1]}) \wedge \ldots \wedge (\neg {t_{1,0}}^{[1]} \wedge {t_{2,0}}^{[1]})
\end{aligned}$$

All expressions ${t_{1,i}}^{[1]}$, ${t_{2,i}}^{[1]}$, ${t_{\mathrm{HI}_1}}^{[n-k]}$, and ${t_{\mathrm{HI}_2}}^{[n-k]}$ are defined in the same way as it was done in Lemma 4.26. Second, we need to use the number of all the relational operations $cnt_{\mathsf{rel}}(\Phi)$, when reducing the bit-widths in Φ. The third modification is needed for constructing a satisfying assignment α' for the bit-width reduced

formula Φ' out of the satisfying assignment α for Φ. When selecting the bit-index which is used as a witness for the evaluation of a given expression ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$, we choose the index of the most significant bit which is assigned to a different value in the two terms. As in Lemma 4.26, an arbitrary bit-index can be chosen if both terms are assigned to the same value.

Again, the reduction is only possible because there is no interaction between different bit-indices. While we only considered ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ in our proof, it is easy to see that it holds for all relational operations from Table 4.1. All unsigned operations can be replaced by ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ as in the definition of Table 4.1. For signed operations, an additional *if-then-else* constraint on the most significant bit is needed. □

So far, we only discussed extensions by *indexing* and *relational operations* separately. However, using the same principles, it is indeed possible to show that we can add both kind of operations at the same time without growth in complexity. We only sketch the argument: As in the original proof for indexing, we first remove all occurrences of the indexing operation from the formula. This time, it is not sufficient to extract those bit-indices from the bit-vectors. Instead, we have to split all bit-vectors at the corresponding bit-index. Let i with $0 < i < n$ be an index that explicitly occurs at some point in the formula. Replace ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ with

$$
\begin{aligned}
& ({t\text{HI}_1}^{[n-i-1]} <_{\mathbf{u}} {t\text{HI}_2}^{[n-i-1]}) \\
\vee\ & ({t\text{HI}_1}^{[n-i-1]} = {t\text{HI}_2}^{[n-i-1]}) \wedge (\neg {t_{1,i}}^{[1]} \wedge {t_{2,i}}^{[1]}) \\
\vee\ & ({t\text{HI}_1}^{[n-i-1]} = {t\text{HI}_2}^{[n-i-1]}) \wedge ({t_{1,i}}^{[1]} \Leftrightarrow {t_{2,i}}^{[1]}) \wedge ({t\text{LO}_1}^{[i]} <_{\mathbf{u}} {t\text{LO}_2}^{[i]})
\end{aligned}
$$

For the more general case, with indices $I = \{i_1, \ldots, i_m\}$, the bit-vectors need to be split analogously at all bit-indices i_l. Apart from this, the reduction works as already described. This leads to the following corollary:

Corollary 4.30. Q*F_BV*2_{bw} *extended by indexing together with relational operations from Table 4.1 is in* NP.

See Appendix 4.11.3 for an example.

4.7.3 QF_BV$2_{\ll 1}$

Figure 4.1 depicts our forthcoming results on extending QF_BV$2_{\ll 1}$ with operations. An edge (o_1, o_2) means that o_1 can be reduced to o_2, together with bitwise operations and equality. The vertex *bvshl_1* represents *left shift by 1*, and plays a central role as being a base operation in QF_BV$2_{\ll 1}$. The vertex *bvmul_c* represents *multiplication by constant*, and the four vertices to the right correspond to different kinds of unsigned and signed relational operations. All the other vertices are self-explanatory. Note that each operation which is mutually reachable with *bvshl_1*, namely *bvlshr_1*, *bvadd*, *bvsub*, and *bvmul_c*, can be used as an alternative base operation instead of *bvshl_1*.

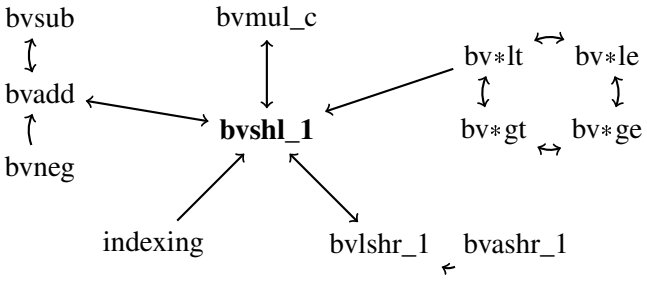


Figure 4.1: Extending QF_BV2$_{\ll 1}$ with operations

First, we show that QF_BV2$_{\ll 1}$ can be extended with *indexing*. Although a similar result was proposed for QF_BV2$_{\mathsf{bw}}$, the reduction we used there is not appropriate for QF_BV2$_{\ll 1}$, because of the presence of shifts in the formulas.

Theorem 4.31. Q*F_BV*2$_{\ll 1}$ *extended by indexing is in* PSPACE.

Proof. The counter we introduced in our translation from QF_BV2$_{\ll 1}$ to sequential circuits (Lemma 4.25) can be used to return the value at a specific bit-index of a bit-vector. □

Instead of *left shift by 1*, we could also have used *logical right shift by 1* to define QF_BV2$_{\ll 1}$.

Theorem 4.32. Left shift by 1 *and* logical right shift by 1 *can be reduced to each other.*

Proof. We give a direct translation:

$t^{[n]} \ll 1^{[n]}$	
replace with:	$x^{[n]}$
add assertions:	$x \gg_{\mathbf{u}} 1 \;=\; t \& \left(\sim 0^{[n]} \gg_{\mathbf{u}} 1\right)$
	$x \& 1^{[n]} \;=\; 0^{[n]}$

$t^{[n]} \gg_{\mathbf{u}} 1^{[n]}$	
replace with:	$x^{[n]}$
add assertions:	$x \ll 1 \;=\; t \& \left(\sim 0^{[n]} \ll 1\right)$
	$x \& v^{[n]} \;=\; 0^{[n]}$
	$v \ll 1 \;=\; 0^{[n]}$
	$v \;\neq\; 0^{[n]}$

,

The claim follows. □

Furthermore, it is well-known that any *arithmetic right shift* ${t_1}^{[n]} \gg_{\mathbf{s}} {t_2}^{[n]}$ can be reduced to *logical right shift*, using *ite* $(t_1[n-1]\ ,\ \sim(\sim t_1 \gg_{\mathbf{u}} t_2)\ ,\ t_1 \gg_{\mathbf{u}} t_2)$.

We now look at arithmetic operations:

Theorem 4.33. QF_BV2$_{\ll 1}$ *extended with* linear modular arithmetic *is in* PSPACE.

Proof. Addition can be expressed as follows:

$t_1{}^{[n]} + t_2{}^{[n]}$		
replace with:	$ts_1 \oplus ts_2 \oplus cin$	,
add assertions:	$ts_1{}^{[n]} = t_1$	
	$ts_2{}^{[n]} = t_2$	
	$cin^{[n]} = cout \ll 1$	
	$cout^{[n]} = (ts_1 \& ts_2) \mid (ts_1 \& cin) \mid (ts_2 \& cin)$	

Multiplication by constant can be splitted into several multiplications by 2, i.e., *left shift by 1*, and *addition*, similar to [208, 209]. Given such a multiplication $t^{[n]} \cdot c^{[n]}$, we introduce two sets of variables, s_i and x_i, $0 \leqslant i \leqslant \mathsf{L}c$. Each s_i represents $t \ll i$, and calculated by shifting s_{i-1} by 1. Note that only logarithmic many steps need to be performed. Each x_i represents the subresult in the ith step. By considering the individual bits of c, s_i either is or is not added to the previous subresult x_{i-1}. Finally, $x_{\mathsf{L}c}$ provides the required product.

$t^{[n]} \cdot c^{[n]}$		
replace with:	$x_{\mathsf{L}c}{}^{[n]}$	
add assertions:	$s_0{}^{[n]} = t$	
	$s_i{}^{[n]} = s_{i-1} \ll 1$	$, 0 < i \leqslant \mathsf{L}c$
	$x_0{}^{[n]} = \begin{cases} s_0 & \text{if } [\![c]\!][0] = 1 \\ 0 & \text{otherwise} \end{cases}$	
	$x_i{}^{[n]} = \begin{cases} x_{i-1} + s_i & \text{if } [\![c]\!][i] = 1 \\ x_{i-1} & \text{otherwise} \end{cases}$	$, 0 < i \leqslant \mathsf{L}c$

Considering the opposite direction, $t \ll 1$ can easily be expressed as $t \cdot 2$. Consequently, it can also be expressed as $ts + ts$ where $ts^{[n]}$ is a Tseitin variable for t. This shows we could also have used *addition* instead of *left shift by 1* to define QF_BV2$_{\ll 1}$.

Unary minus (bvneg) and *subtraction* (bvsub) can obviously be defined in QF_BV2$_{\ll 1}$ by using two's complement representation. Furthermore, it is easy to see that *addition* and *subtraction* can be reduced to each other. Extending QF_BV2$_{\ll 1}$ with common relational operations, such as *unsigned less than* (bvult), does not increase complexity either. A term $t_1{}^{[n]} <_{\mathbf{u}} t_2{}^{[n]}$ is the same as checking whether $t_1 - t_2 <_{\mathbf{u}} 0$ holds, which can be replaced by constructing an adder for $t_1 + (\sim t_2) + 1$, analogously to the one above, and then check whether overflow occurs, i.e., $ts_2 \neq 0 \;\&\; \neg c_{out}[n-1]$. Obviously, the common unsigned or signed relational operations *less than*, *greater than*, *less than or equal*, and *greater than or equal* are equally powerful. □

4.7.4 QF_BV$2_{\ll c}$

Figure 4.2 depicts our forthcoming results on extending QF_BV$2_{\ll c}$ with operations. The vertex *bvshl_c* represents *left shift by constant*, which is a base operation. Since *bvshl_1* is a special case of *bvshl_c*, all the operations that can extend QF_BV$2_{\ll 1}$ (cf. the previous section), represented by the dashed segment in the upper left corner, can obviously be reduced to *bvshl_c*. Actually, as we have already mentioned before, any common operation can extend this fragment, with QF_BV$2_{\ll c}$ being NEXPTIME-complete. This explains why *bvshl_c* is reachable from all the vertices. We only give the most interesting explicit reductions in this direction. The other direction, i.e., presenting operations being reachable from *bvshl_c*, is more important from the theoretical point of view, since those ones can be used as alternative base operations instead of *bvshl_c*. These operations are *extract*, *concat*, *bvmul*, *bvshl*, *bvlshr_c*, and *bvlshr*.

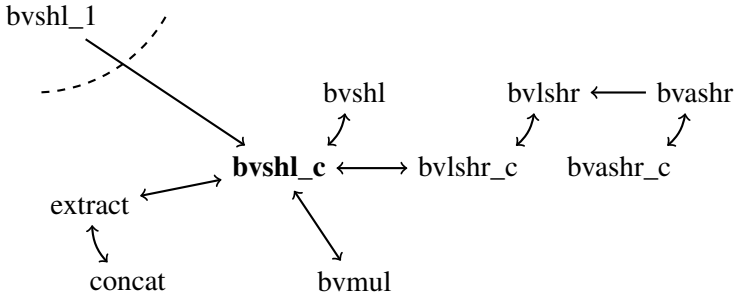


Figure 4.2: Extending QF_BV$2_{\ll c}$ with operations

Theorem 4.34. *The operators* bvshl_c *and* bvlshr_c *can be reduced to each other.*

Proof. Given a term $t^{[n]} \ll c^{[n]}$ or $t^{[n]} \gg_{\mathbf{u}} c^{[n]}$, there are two boundary cases: if $c = 0$ then rewrite the term to t; if $c \geqslant n$ then rewrite to $0^{[n]}$. Otherwise, i.e., for $0 < c < n$, the following reductions can be applied:

$t^{[n]} \ll c^{[n]}$	
replace with:	$x^{[n]}$
add assertions:	$x \gg_{\mathbf{u}} c \;=\; t \,\&\, \left(\sim 0^{[n]} \gg_{\mathbf{u}} c\right)$
	$x \,\&\, \left(\sim 0^{[n]} \gg_{\mathbf{u}} (n-c)^{[n]}\right) \;=\; 0^{[n]}$

,

$t^{[n]} \gg_{\mathbf{u}} c^{[n]}$	
replace with:	$x^{[n]}$
add assertions:	$x \ll c \;=\; t \,\&\, \left(\sim 0^{[n]} \ll c\right)$
	$x \,\&\, \left(\sim 0^{[n]} \ll (n-c)^{[n]}\right) \;=\; 0^{[n]}$

The claim follows. □

Each kind of *shift by constant* is a special case of the respective general *shift*.[4] As mentioned in the previous section, *arithmetic shift* can be expressed by *logical shift*.

Theorem 4.35. Extraction, concatenation, *and* bvshl_c *can be reduced to each other.*

Proof. First, consider extraction and concatenation:

$t_1{}^{[m]} \circ t_2{}^{[n]}$		
replace with:	$x^{[m+n]}$	,
add assertions:	$t_1 = x\,[m+n-1:n]$	
	$t_2 = x\,[n-1:0]$	

$t^{[n]}\,[i:j]$			
replace with:	$x^{[i-j+1]}$		
add assertion:	$t = x$	if $i = n-1$	if $j = 0$
	$t = y_1{}^{[n-i-1]} \circ x$	otherwise	
	$t = x \circ y_2{}^{[j]}$	if $i = n-1$	otherwise
	$t = y_1{}^{[n-i-1]} \circ x \circ y_2{}^{[j]}$	otherwise	

The base operation *bvshl_c* can then easily be expressed by *extraction* and *concatenation* (and also by any of them alone, since they can be reduced to each other). The boundary cases for *bvshl_c* can be handled in the same way as above, therefore we now assume that $0 < c < n$, and rewrite the term $t^{[n]} \ll c^{[n]}$ to $t\,[n-c-1:0] \circ 0^{[c]}$.

The reduction in the other way around, i.e., *extraction* (or *concatenation*) to *bvshl_c* and *bvlshr_c*, takes a special role. Given a term $t^{[n]}\,[i:j]$, *extraction* produces a new term of bit-width $i-j+1$. This change in bit-width (which also occurs for *concatenation*) cannot be captured by only applying rewriting rules using *shifts*. However, we can find a reduction from bit-vector formulas using only *extraction*, bitwise operations, and equality to ones using only *shift by constant*, bitwise operations, and equality, as follows.

Given a formula Φ with bit-vector variables $x_1{}^{[n_1]}, \ldots, x_l{}^{[n_l]}$, let us calculate the maximal bit-width $n_{\mathsf{max}} := \max_k\{n_k\}$. First, replace each *extraction* $t^{[m]}\,[i:j]$ in Φ with

$$\big(t \ll (n_{\mathsf{max}} - 1 - i)\big) \gg_{\mathbf{u}} (n_{\mathsf{max}} - 1 - i + j)$$

In a second step, replace each bit-vector variable $x_k{}^{[n_k]}$ with a new bit-vector variable $x'_k{}^{[n_{\mathsf{max}}]}$. Finally, for each x'_k, add the following assertion to the formula:

$$x'_k{}^{[n_{\mathsf{max}}]} \;=\; \big(x'_k \ll (n_{\mathsf{max}} - n_k)\big) \gg_{\mathbf{u}} (n_{\mathsf{max}} - n_k)$$

[4] Although we do not intend the present a reduction of a general *shift* to the respective *shift by constant*, it is worth to mention that a common approach for such a reduction is the *barrel shifter*.

In the resulting formula, all bit-vectors have the same bit-width, and each bit-vector and each result of an *extraction* can take exactly those values it could take in the original formula, apart from leading zeros. □

We now take a closer look at multiplication:

Theorem 4.36. Multiplication *and* bvshl_c *can be reduced to each other.*

Proof. First, we show how *bvshl_c* can be expressed by *bvmul*. Again, assume that $0 < c < n$. In this case, $t^{[n]} \ll c^{[n]}$ can be expressed as $t \cdot 2^c$. We can construct 2^c, being an exponential number, as a bit-vector in $\mathsf{L}c$ steps using *exponentiation by squaring*. We introduce two sets of variables, p_i and x_i, $0 \leqslant i \leqslant \mathsf{L}c$. Each p_i represents the number $2^{(2^i)}$, and each x_i the subresult in the ith step. By considering the individual bits of c, the previous subresult x_{i-1} either is or is not multiplied by p_i. Finally, $x_{\mathsf{L}c}$ provides the value 2^c.

$t^{[n]} \ll c^{[n]}$		
replace with:	${x_{\mathsf{L}c}}^{[n]}$	
add assertions:	${p_0}^{[n]} = 2$	
	${p_i}^{[n]} = p_{i-1} \cdot p_{i-1}$	$, 0 < i \leqslant \mathsf{L}c$
	${x_0}^{[n]} = \begin{cases} 2 & \text{if } [\![c]\!][0] = 1 \\ 1 & \text{otherwise} \end{cases}$	
	${x_i}^{[n]} = \begin{cases} x_{i-1} \cdot p_i & \text{if } [\![c]\!][i] = 1 \\ x_{i-1} & \text{otherwise} \end{cases}$	$, 0 < i \leqslant \mathsf{L}c$

Although we know, based on the complexity results, that even general *multiplication* can be expressed in this fragment, it is still a non-trivial task to give an explicit reduction. While everal polynomial multiplication algorithms in the bit-width of operands exist, we cannot directly apply them since we now need a *logarithmic* translation in the bit-width. Before showing how to simulate the common "shift and add" algorithm, we first introduce four bit-vector helper operations to make the presentation as transparent as possible: *binmagic*, *selfconcat*, *halfshuffle*, and *expand*. Furthermore, let us introduce the notation $\mathsf{P}n$ for the *nearest power of 2*, and define it as follows: $\mathsf{P}n := 2^{\mathsf{L}n}$.

For the helper operation *binmagic*, which is in fact about constructing a *binary magic number*, we use the same notation and approach as in Lemma 4.23, where $m < n$:

binmagic $(2^m,\ 2^n)$		
replace with:	$x^{[2^n]}$	,
add assertion:	$x \ll 2^m \ = \sim x$	

Selfconcat receives a bit-vector term $t^{[2^m]}$ and concatenates it with itself until constructing a bit-vector of bit-width 2^n, as follows, where $m \leqslant n$:

selfconcat $\left(t^{[2^m]},\ 2^n\right)$		
replace with:	${x_n}^{[2^n]}$	
add assertions:	${x_m}^{[2^m]} = t$	
	${x_i}^{[2^i]} = x_{i-1} \circ x_{i-1}$	$, m < i \leqslant n$

Halfshuffle applies a logarithmic translation, which is based on the generalization of a bit-vector operation called *half-shuffle* [231, Chapter 7]. This generalized variant receives a bit-vector $t^{[2^m]}$ and produces the following bit-vector of bit-width 2^n:

$$\underbrace{0\ldots0}_{2^{n-m}-1}\ t[2^m-1]\ \underbrace{0\ldots0}_{2^{n-m}-1}\ t[2^m-2]\ \ldots\ \underbrace{0\ldots0}_{2^{n-m}-1}\ t[0]$$

In the initialization step, we apply *zero extension* to t. Then, in m steps, we shuffle smaller and smaller bit groups in our bit-vector. In the 1st step, the two halves (i.e., 2^{m-1}-bit groups) are shuffled. In the 2nd step, the halves of all the previously shuffled halves (i.e., 2^{m-2}-bit groups), and so on. In the last step, we shuffle single bits, and this is how to put each bit at its destination. Assume again that $m \leqslant n$.

halfshuffle $\left(t^{[2^m]},\ 2^n\right)$		
replace with:	${x_m}^{[2^n]}$	
add assertions:	${x_0}^{[2^n]} = \mathit{ext}_{\mathbf{u}}\left(t, 2^n - 2^m\right)$	
	${x_i}^{[2^n]} = \begin{pmatrix} x_{i-1} \mid \left(x_{i-1} \ll \left(2^{n-i} - 2^{m-i}\right)\right) \\ \& \\ \mathit{binmagic}\left(2^{m-i},\ 2^n\right) \end{pmatrix}$	$, 0 < i \leqslant m$

As it can be seen above, in the ith step we (i) shift our current bit-vector left by the constant $2^{n-i} - 2^{m-i}$, (ii) merge it with the original bit-vector, by using *bitwise or*, (iii) and we mask some unnecessary bit groups out, by using a binary magic number. For an example, see Appendix 4.11.4.

Expand "multiplies" each bit of $t^{[2^m]}$ into a bit group of size 2^{n-m}. The resulting bit-vector can be visualized as follows:

$$\underbrace{t[2^m-1]\ldots t[2^m-1]}_{2^{n-m}}\ \underbrace{t[2^m-2]\ldots t[2^m-2]}_{2^{n-m}}\ \ldots\ \underbrace{t[0]\ldots t[0]}_{2^{n-m}}$$

In the initial step, we use *halfshuffle*. In the next $n - m$ steps, we copy larger and larger non-zero bit groups, by using *left shift* and *bitwise or*. Assume again that $m \leqslant n$.

expand $\left(t^{[2^m]},\ 2^n\right)$		
replace with:	${x_{n-m}}^{[2^n]}$	
add assertions:	${x_0}^{[2^n]} = \mathit{halfshuffle}\left(t,\ 2^n\right)$	
	${x_i}^{[2^n]} = x_{i-1} \mid \left(x_{i-1} \ll 2^{i-1}\right)$	$, 0 < i \leqslant n - m$

Now we are ready to propose how to express *multiplication* by simulating the common "shift and add" algorithm for integers. In a first step, one of the operands is multiplied independently by each digit of the other operand. Using base 2, this multiplication by a single digit can be expressed by a logical and-operation. Afterwards, the results of the single-digit multiplications are shifted by the offset of the corresponding digit and finally added to give the result of the full multiplication. While this approach is straightforward in a naive implementation, we have to ensure only logarithmic many operations in the bit-width are used in our encoding. To achieve this, we generate bit-vectors of quadratic bit-width $(\mathsf{P}n)^2$ out of our original operands of bit-width n, by applying *selfconcat* to the first one and *expand* to the second one. Using *bitwise and* on the two new bit-vectors, we directly get the results of all single-digit multiplications in one step. More precisely, the resulting bit-vector consists of $\mathsf{P}n$ groups of $\mathsf{P}n$ bits, each group representing the result of one single-digit multiplication. To add all $\mathsf{P}n$ partial results, a binary addition algorithm is used. Iteratively pairs of neighbouring groups are shifted relative to each others' offsets and then added to form one new group. The number of groups therefore is halved in each step, resulting in the final sum after $\log_2(\mathsf{P}n) = \mathsf{L}n$ steps. For a detailed example, see also Appendix 4.11.5.

$$t_1{}^{[n]} \cdot t_2{}^{[n]}$$

replace with: $x_{\mathsf{L}n}\,[n-1:0]$

add assertions:

$$x_0{}^{[(\mathsf{P}n)^2]} = \textit{selfconcat}\left(\textit{ext}_{\mathbf{u}}\left(t_1, \mathsf{P}n-n\right),\ (\mathsf{P}n)^2\right) \;\&\textit{expand}\left(\textit{ext}_{\mathbf{u}}\left(t_2, \mathsf{P}n-n\right),\ (\mathsf{P}n)^2\right)$$

$$b_i{}^{[(\mathsf{P}n)^2]} = \textit{binmagic}\left(2^i \cdot \mathsf{P}n,\ (\mathsf{P}n)^2\right) \quad , \begin{array}{l} 0 \leqslant i \\ i < \mathsf{L}n \end{array}$$

$$x_i{}^{[(\mathsf{P}n)^2]} = \begin{array}{l} (x_{i-1} \& b_{i-1}) + \\ (x_{i-1} \& \sim b_{i-1}) \gg_{\mathbf{u}} \left(2^{i-1} \cdot (\mathsf{P}n-1)\right) \end{array} \quad , \begin{array}{l} 0 < i \\ i \leqslant \mathsf{L}n \end{array}$$

The claim follows. □

4.8 Logics with Quantifiers and Binary Encoding

In this section, we look into the complexity of quantified bit-vector logics with binary encoding. While we already gave some results for BV2 and UFBV2 in [151], we now extend our previous work by discussing some fragments of those logics. Additionally, we also take a look at non-recursive macros (as allowed, e.g., in the SMT-LIB format) for quantifier-free logics, which have a very similar effect to restricting the bit-width of universal variables in quantified logics. We give new complexity results for all fragments and extensions.

4.8.1 General Quantification

By allowing quantification and uninterpreted functions, and using binary encoding, we obtain UFBV2, the most expressive bit-vector logic considered in this chapter.

Theorem 4.37. U*FBV*2 *is* 2*-NExpTime-complete [151].*

Proof. It is straightforward to see that UFBV2 $\in$ 2-NExpTime, considering that every UFBV2 formula can be translated exponentially to a corresponding formula in UFBV1 $\in$ NEXPTIME (Proposition 4.14), by applying a simple unary re-encoding to all the scalars in the formula. 2-NExpTime-hardness directly follows from Lemma 4.40. □

To prove that UFBV2 is 2-NExpTime-hard, we pick a 2-NExpTime-hard problem and then reduce it to UFBV2. We can find a suitable problem among the so-called *domino tiling* problems [65]. First, we give a definition of a domino system and then specify a 2-NExpTime-hard problem on this kind of systems.

Definition 4.38 (Domino System)**.** A domino system is a tuple $\langle T, H, V, n\rangle$, where

- T is a finite set of *tile types*, in our case, $T = [0, k-1]$, where $k \geqslant 1$;
- $H, V \subseteq T \times T$ are the horizontal and vertical matching conditions, respectively;
- $n \geqslant 1$, encoded in *unary* format.

Note that the above definition differs (but not substantially) from the classical one in [65]. W.l.o.g., we use sub-sequential natural numbers for identifying tiles. Similarly to [171, 183], the size factor n, encoded in *unary* form, is part of the input. However, while a start tile α and a terminal tile ω is usually used, in our case, w.l.o.g., the starting tile is denoted by 0 and the terminal tile by $k-1$.

There are different domino tiling problems examined in the literature. In [65], a classical tiling problem is introduced, namely the *square tiling* problem, which can be defined as follows:

Definition 4.39 (Square Tiling)**.** Given a domino system $\langle T, H, V, n\rangle$, an $f(n)$-square tiling is a mapping $\lambda : [0, f(n)-1] \times [0, f(n)-1] \mapsto T$ such that

- the first row starts with the start tile: $\lambda(0,0) = 0$
- the last row ends with the terminal tile: $\lambda(f(n)-1, f(n)-1) = k-1$
- all horizontal matching conditions hold:
$$\big(\lambda(i,j), \lambda(i,j+1)\big) \in H \quad \forall i,j \,.\, 0 \leqslant i < f(n),\ 0 \leqslant j < f(n)-1$$
- all vertical matching conditions hold:
$$\big(\lambda(i,j), \lambda(i+1,j)\big) \in V \quad \forall i,j \,.\, 0 \leqslant i < f(n)-1,\ 0 \leqslant j < f(n)$$

In [65], a general theorem on the complexity of domino tiling problems is proved. More precisely, the $f(n)$-square tiling problem can be shown to be NTIME $(f(n))$-complete. From this, it directly follows that the $2^{(2^n)}$-square tiling problem is 2-NExpTime-complete.

Lemma 4.40. *The $2^{(2^n)}$-square tiling problem can be reduced to* U*FBV*2.

Proof. Given a domino system $\langle T = [0, k-1], H, V, n\rangle$, let us introduce the following notations which we intend to use in the resulting UFBV2 formula.

- Represent each tile in T with the corresponding bit-vector constant of bit-width $\mathsf{L}k$.
- Represent the horizontal and vertical matching conditions with uninterpreted functions (actually predicates) $h^{[1]}({t_1}^{[\mathsf{L}k]}, {t_2}^{[\mathsf{L}k]})$ and $v^{[1]}({t_1}^{[\mathsf{L}k]}, {t_2}^{[\mathsf{L}k]})$, respectively.
- Represent the tiling with an uninterpreted function $\lambda^{[\mathsf{L}k]}(i^{[2^n]}, j^{[2^n]})$. λ returns the tile in the cell at the row index i and column index j. Note that the bit-width of i and j is exponential in the size of the domino system, but due to *binary encoding* it can represented polynomially.

The resulting UFBV2 formula is as follows:

$$
\begin{aligned}
&\forall i^{[2^n]}, j^{[2^n]}.\\
&\qquad \lambda(0,0) = 0 \quad \wedge \quad \lambda\Big(2^{(2^n)}-1, 2^{(2^n)}-1\Big) = k-1\\
&\qquad \wedge \bigwedge_{(t_1,t_2)\in H} h(t_1,t_2) \quad \wedge \bigwedge_{(t_1,t_2)\in V} v(t_1,t_2)\\
&\qquad \wedge \quad \Big(j \neq 2^{(2^n)}-1 \;\Rightarrow\; h\big(\lambda(i,j), \lambda(i,j+1)\big)\Big)\\
&\qquad \wedge \quad \Big(i \neq 2^{(2^n)}-1 \;\Rightarrow\; v\big(\lambda(i,j), \lambda(i+1,j)\big)\Big)
\end{aligned}
$$

This formula contains four kinds of constants. Three can be encoded directly ($0^{[2^n]}$, $0^{[\mathsf{L}k]}$, and $(k-1)^{[\mathsf{L}k]}$). The constant $2^{(2^n)}-1$ has to be encoded as $\sim 0^{[2^n]}$ in order to avoid an exponential representation. The size of the resulting formula, due to *binary encoding* on bit-widths, is polynomial in the size of the domino system. □

Similar to Section 4.6 and to our work in [101], we can now restrict the set of operations in UFBV2 to allow only bitwise operations, equality and *left shift by constant* (or *left shift by 1*). We refer to this logic as UFBV2$_{\ll c}$ (or UFBV2$_{\ll 1}$, in the case of *left shift by 1*). From a different point of view, it is also possible to consider this as an extension of QF_BV2$_{\ll c}$ and QF_BV2$_{\ll 1}$ by quantifiers and uninterpreted functions.

Since we can use bitwise operations, equality and *left shift by constant* to express all common operations, UFBV2$_{\ll c}$ remains 2-NExpTime-complete. However, in contrast to quantifier-free logics, we do not lose any expressiveness in UFBV2$_{\ll 1}$, either. We can see this already from the fact that we only used bitwise operations, equality and *addition* in Lemma 4.40. Since, as we pointed out in Section 4.7.3, *addition* can be reduced to bitwise operations, equality and *left shift by 1*, the following result follows immediately:

Corollary 4.41. U*FBV*$2_{\ll 1}$ *is* 2*-NExpTime-complete.*

Nevertheless, we want to formalize this in a proposition and give a constructive proof by showing how UFBV$2_{\ll c}$ can be reduced to UFBV$2_{\ll 1}$.

Proposition 4.42. U*FBV*$2_{\ll c}$ *can be reduced to* U*FBV*$2_{\ll 1}$.

Proof. Let Φ denote a bit-vector formula, ${x^{[n]}}$, ${y^{[n]}}$ fresh bit-vector variables, and ${f_n}^{[n]}(\cdot,\cdot)$ a fresh uninterpreted function of arity 2, taking arguments of bit-width n. Replace each expression ${t^{[n]}} \ll {c^{[n]}}$ in Φ with ${f_n}^{[n]}\left({t^{[n]}}, {c^{[n]}}\right)$, extend the quantifier prefix of Φ with $\forall {x^{[n]}}, {y^{[n]}}$, and add the following two constraints to the matrix of Φ:

$$\begin{aligned} {f_n}^{[n]}(x,0) &= x \\ {f_n}^{[n]}(x,y+1) &= {f_n}^{[n]}(x,y) \ll 1 \end{aligned}$$

While the second constraint still contains *addition* to improve readability, this can be replaced by using *left shift by 1*, as described in Section 4.7.3. □

Remark 4.43. This result is not very surprising if we consider the alternative characterizations of QF_BV$2_{\ll 1}$ and QF_BV$2_{\ll c}$ as given in Section 4.7. We showed that *addition* is equally expressive as *left shift by 1* and *multiplication* is equally expressive as *left shift by constant*. In *Peano arithmetic*, *multiplication* is defined by using *addition*, uninterpreted functions, and quantification. In the context of bit-vectors, this definition of multiplication can be expressed by introducing $\forall {x^{[n]}}, {y^{[n]}}$ to the quantifier prefix and adding the following constraints:

$$\begin{aligned} {f_n}^{[n]}(x,0) &= 0 \\ {f_n}^{[n]}(x,y+1) &= {f_n}^{[n]}(x,y) + x \end{aligned}$$

With these two axioms, the *multiplication* ${t_1}^{[n]} \cdot {t_2}^{[n]}$ of two elements in Peano arithmetic is uniquely defined by the instance ${f_n}^{[n]}\left({t_1}^{[n]}, {t_2}^{[n]}\right)$ of the uninterpreted function f_n.

While we were also able to give some complexity results for BV2 in [151], it remains unclear whether BV2 is complete for any complexity class.

Proposition 4.44. B*V*2 ∈ EXPSPACE *and* B*V*2 *is* NEXPTIME*-hard [151].*

Proof. Given a BV2 formula, a simple unary re-encoding can be used to give an exponential translation to BV1 ∈ PSPACE (Proposition 4.13). As a consequence, BV2 ∈ EXPSPACE. Because of QF_BV2 ⊂ BV2, NEXPTIME-hardness follows trivially. □

4.8.2 Restricting the Bit-Width of Universal Variables

We now show that a completeness result can be obtained when a certain restriction to the bit-width of the universal variables is applied. For a given formula $\Phi \in \mathrm{BV2}$, let $max_{\mathsf{bw}(\exists)}(\Phi)$ and $max_{\mathsf{bw}(\forall)}(\Phi)$ denote the *maximal bit-width of all the existentially and universally quantified variables*, respectively. (We define $max_{\mathsf{bw}(\exists)}(\Phi) := 0$ and $max_{\mathsf{bw}(\forall)}(\Phi) := 0$ if Φ does not contain any existential and universal variables, respectively.) Now we give a definition, similar to the one of scalar-boundedness in Definition 4.15:

Definition 4.45 (Universally Bit-Width Bounded Formula Set)**.** An infinite set S of quantified bit-vector formulas is *universally bit-width bounded*, if and only if there exists a polynomial function $p : \mathbb{N} \mapsto \mathbb{N}$, such that, for each formula $\Phi \in S$, it holds that $max_{\mathsf{bw}(\forall)}(\Phi) \leqslant p\left(\mathsf{L}max_{\mathsf{bw}(\exists)}(\Phi)\right)$.

Theorem 4.46. *If $S \subset$ UFBV2 (or $S \subset$ BV2) is universally bit-width bounded, then $S \in$ NExpTime.*

Proof. Let $S \subset$ UFBV2 be universally bit-width bounded and let p_0 be the polynomial function that exists according to Definition 4.45. For any $\Phi_0 \in S$, let $n := |\Phi_0|$. We can assume that Φ_0 contains at most $k \leqslant n$ universal variables. Also, let $max_{\mathsf{scl}}(\Phi_0)$ and $cnt_{\mathsf{scl}}(\Phi_0)$ be defined in the same way as it was done in Section 4.5. This implies $max_{\mathsf{bw}(\exists)}(\Phi_0) \leqslant max_{\mathsf{scl}}(\Phi_0) \leqslant 2^n$ and $cnt_{\mathsf{scl}}(\Phi_0) \leqslant n$.

To prove that $S \in$ NExpTime, we now give a reduction to QF_BV1 $\in$ NP which is only single-exponential in $n = |\Phi_0|$ for any $\Phi_0 \in S$. First, all universal variables are eliminated by universal expansion. This produces a quantifier-free formula $\Phi_1 \in$ QF_UFBV2 with

$$\begin{aligned} max_{\mathsf{scl}}(\Phi_1) &= max_{\mathsf{scl}}(\Phi_0) \leqslant 2^n \\ cnt_{\mathsf{scl}}(\Phi_1) &\leqslant cnt_{\mathsf{scl}}(\Phi_0) \cdot 2^{k \cdot max_{\mathsf{bw}(\forall)}(\Phi_0)} \\ &\leqslant cnt_{\mathsf{scl}}(\Phi_0) \cdot 2^{n \cdot p_0\left(\mathsf{L}max_{\mathsf{bw}(\exists)}(\Phi_0)\right)} \\ &\leqslant cnt_{\mathsf{scl}}(\Phi_0) \cdot 2^{p_1(n)} \end{aligned}$$

for some polynomial function p_1. Since Φ_1 does not contain any (universal) quantifiers, it can be polynomially translated to some $\Phi_2 \in$ QF_BV2, by replacing all uninterpreted functions of Φ_1 with bit-vector variables and adding at most quadratic many Ackermann constraints (as in Proposition 4.12). Therefore,

$$\begin{aligned} max_{\mathsf{scl}}(\Phi_2) &= max_{\mathsf{scl}}(\Phi_1) \leqslant 2^n \\ cnt_{\mathsf{scl}}(\Phi_2) &\leqslant p_2\left(cnt_{\mathsf{scl}}(\Phi_1)\right) \leqslant p_2\left(cnt_{\mathsf{scl}}(\Phi_0) \cdot 2^{p_1(n)}\right) \end{aligned}$$

for some polynomial function p_2. In a last step, a unary re-encoding is applied to Φ_2 (similar to Proposition 4.18), resulting in $\Phi_3 \in$ QF_BV1. The size of Φ_3 is bounded by

$$\begin{aligned} |\Phi_3| &\leqslant max_{\mathsf{scl}}(\Phi_2) \cdot cnt_{\mathsf{scl}}(\Phi_2) + c \\ &\leqslant 2^n \cdot p_2\left(cnt_{\mathsf{scl}}(\Phi_0) \cdot 2^{p_1(n)}\right) + c \leqslant 2^{p_3(n)} + c \end{aligned}$$

for some polynomial function p_3. Therefore, $\Phi_3 \in$ QF_BV1 is still only single-exponential in the size of Φ_0. Together with QF_BV1 $\in$ NP (Proposition 4.11), this shows that $S \in$ NEXPTIME. □

We now define $\mathrm{BV}_{\mathsf{log}} \subset$ BV2 and $\mathrm{UFBV}_{\mathsf{log}} \subset$ UFBV2 as the set of all $\Phi \in$ BV2 and $\Phi \in$ UFBV2 with $\mathit{max}_{\mathsf{bw}(\forall)}(\Phi) \leqslant \mathsf{L}\mathit{max}_{\mathsf{bw}(\exists)}(\Phi) + 1$, respectively. These fragments are of special practical interest, because they can be used to express quantification over array indices if arrays are represented as bit-vectors. Arrays play an important role in automated software model checking as, for example, done in the SAGE project by Microsoft [115]. Quantification over array indices is also discussed in [42], where the so-called *bounded array property fragment* is addressed.

Theorem 4.47. B$\mathit{V}_{\mathsf{log}}$ *and* U$\mathit{FBV}_{\mathsf{log}}$ *are* NEXPTIME*-complete.*

Proof. It is easy to see that $\mathrm{BV}_{\mathsf{log}}$ and $\mathrm{UFBV}_{\mathsf{log}}$ are NEXPTIME-hard since both logics are an extension of QF_BV2, which is already NEXPTIME-hard (Proposition 4.22). The other direction is a consequence of Theorem 4.46, since $\mathrm{BV}_{\mathsf{log}}$ and $\mathrm{UFBV}_{\mathsf{log}}$ are universally bit-width bounded by definition. □

Note that this kind of proof only holds for bit-vector logics with binary encoding. When a unary encoding is used, restricting the bit-width of universal variables does not have any effect on the complexity of the given problem class.

4.8.3 Non-Recursive Macros

A very similar effect occurs when non-recursive macros are added to our logics. For example, SMT-LIB allows the usage of non-recursive macros via the keywords `define-fun` and `let`. In the general case, allowing macros can increase the complexity of a given class. For instance, Boolean formulas extended by non-recursive macros equal to the class of *Boolean Programs* or *Nested Boolean Functions* (NBF), which is known to be PSPACE-complete [54, 72]. The same obviously holds for QF_BV1.

However, as shown in Theorem 4.50, extending QF_UFBV2 (or QF_BV2) with non-recursive macros does not give additional expressiveness in terms of complexity. Let the subscript $\mathcal{M}$ denote the fact that, additionally, non-recursive macros can be used in our logic.

Definition 4.48 (Logic with Non-Recursive Macros)**.** Given a bit-vector logic $\mathcal{L}$, let $\mathcal{L}_{\mathcal{M}}$ denote the set of all bit-vector formulas in the following form:

$$
\begin{aligned}
Q\ \forall {u_0}^{[n_0]}, \ldots, {u_k}^{[n_k]}\ . \quad & {t}^{[1]} \\
\wedge \quad & {f_0}^{[w_0]}(u_0, \ldots, u_{k_0}) = {d_0}^{[w_0]} \\
\wedge \quad & \ldots \\
\wedge \quad & {f_m}^{[w_m]}(u_0, \ldots, u_{k_m}) = {d_m}^{[w_m]}
\end{aligned}
$$

where (i) $Q.t^{[1]} \in \mathcal{L}$, (ii) the universal variables ${u_i}^{[n_i]}$ do not appear in $Q.t^{[1]}$, (iii) the uninterpreted functions f_i are called *macros*, (iv) the terms ${d_i}^{[w_i]}$ are called *macro definitions*, and (v) d_i contains no occurrence of f_j if $i \leqslant j$.

Note that t might contain occurrences of any f_i. *Expanding* a macro f_i means to replace all occurrences $f_i(s_0, \ldots, s_{k_i})$ in t with $d_i\sigma$, where $s_0, \ldots, s_{k_i}$ are terms and $\sigma := \{u_0 \mapsto s_0, \ldots, u_{k_i} \mapsto s_{k_i}\}$ is a term substitution.

We now introduce a normal form, similar to the flat form in Definition 4.49, in order to obtain an upper bound for the growth in formula size when applying macro expansion.

Definition 4.49 (Functional Flat Form)**.** A bit-vector formula Φ is in *functional flat form* if and only if every uninterpreted function in Φ has only variables as arguments.

It is easy to see that any Φ can be translated into *functional flat form* with only linear growth in formula size. Given a term $f({t_1}^{[n_1]}, \ldots, {t_k}^{[n_k]})$ in Φ, where f is an uninterpreted function, check if t_i is a variable: if it is, then $x_i := t_i$; otherwise let ${x_i}^{[n_i]}$ be a new Tseitin variable existentially quantified at the innermost prefix position, and add the constraint $x_i = t_i$ to the formula. Then, replace the original term with $f(x_1, \ldots, x_k)$.

Theorem 4.50. Q*F_UFBV*$2_{\mathcal{M}}$ *is* NEXPTIME*-complete.*

Proof. NEXPTIME-hardness is obvious, since QF_UFBV2 $\subset$ QF_UFBV$2_{\mathcal{M}}$. Inclusion can be shown in a similar way as it is done in Theorem 4.46.

Let $\Phi_0 := \forall {u_0}^{[n_0]}, \ldots, {u_k}^{[n_k]} \,.\, t \wedge t_M$ be a QF_UFBV$2_{\mathcal{M}}$ formula of size $n := |\Phi_0|$, where $t \in$ QF_UFBV2 and t_M consists of all the macro definitions. Assume that t is in *functional flat form*. We now inductively expand all macros in t, in the order of $f_m, f_{m-1}, \ldots, f_0$, and also, after each expansion step, we translate the resulting formula into functional flat form again.

First, each macro occurrence $f_m(x_0, \ldots, x_{k_m})$ in t is replaced by an instance $d_m\sigma$ of the macro definition. Since each x_i is a variable, we know that $|d_m\sigma| = |d_m| \leqslant n$. Because f_m has at most n occurrences in t, expanding f_m results in a formula of size bounded by n^2. Recall that we also translate the resulting formula into functional flat form, resulting in formula size bounded linearly in n^2.

Then, we expand f_{m-1}, which now has at most n^2 occurrences. The resulting formula is of size bounded linearly in n^3. By continuing the expansion process with $f_{m-2}, \ldots, f_0$, we finally obtain from t a formula $\Phi_1 \in$ QF_UFBV2 that contains no more macros. It holds that

$$\begin{array}{rcl} \mathit{max}_{\mathsf{scl}}(\Phi_1) & = & \mathit{max}_{\mathsf{scl}}(\Phi_0) \leqslant 2^n \\ \mathit{cnt}_{\mathsf{scl}}(\Phi_1) & \leqslant & l\big(n^{m+1}\big) \leqslant l\big(n^n\big) \leqslant l\big(2^{n \cdot \mathsf{L}n}\big) \end{array}$$

for some linear function l. We now apply a unary re-encoding to Φ_1, yielding $\Phi_2 \in$ QF_UFBV1. The size of Φ_2 is bounded by

$$|\Phi_2| \leqslant \mathit{max}_{\mathsf{scl}}(\Phi_1) \cdot \mathit{cnt}_{\mathsf{scl}}(\Phi_1) + c \leqslant 2^n \cdot l\big(2^{n \cdot \mathsf{L}n}\big) + c,$$

which is only single-exponential in the size of Φ_0. As a consequence, this implies QF_UFBV2$_{\mathcal{M}} \in$ NEXPTIME. □

4.9 Practical Considerations

As mentioned in Section 4.2, our original motivation for considering the complexity of bit-vector logics comes from the fact that state-of-the-art SMT solvers usually rely on bit-blasting when dealing with bit-vector formulas. Our introductory example shows the effect that the exponential explosion caused by bit-blasting can have on a bit-vector formula and, therefore, current SMT solvers often are not able to deal efficiently with bit-vector formulas that are not scalar-bounded.

While our complexity results in Section 4.6 explain why this is the case from a complexity-theoretic point of view, it is of high practical interest if and how state-of-the-art SMT solvers can profit from this knowledge.

4.9.1 Alternative Approaches

Instead of using bit-blasting, we can try to find alternative approaches for solving bit-vector formulas. One possible approach is to polynomially translate bit-vector formulas to some other logic in the same complexity class. For example, target logics for QF_BV2$_{\ll c}$ (or general QF_BV2) are DQBF or EPR, which are both NEXPTIME-complete [161, 188, 187]. For QF_BV2$_{\ll 1}$, a translation to model checking for sequential circuits as given in Lemma 4.25 can be used instead. In both cases, we can profit from the performance of existing techniques for other problem classes. While DQBF solvers have not been considered at all until our recent work in [99], their performance does not nearly reach the one of current EPR solvers as, e.g., iProver [147]. On the other hand, many efficient model checkers for sequential circuits in SMV or AIGER format exist.

In [150], we therefore gave a polynomial translation from QF_BV2 to EPR (this is in contrast to existing translations in [90, 143], which are not guaranteed to be polynomial in the general case), and then did an experimental evaluation using iProver to solve the resulting EPR formulas. The overall results were rather mixed. While we were able to solve some formulas faster, SMT solvers performed better by orders of magnitude on most other problems considering runtime. Looking at the space requirements, iProver performed better in general. However, the gain was less significant than expected. An explanation for this can be found in the way iProver deals with EPR formulas. By solving propositional overapproximations and iteratively applying instantiations of predicates (the underlying concept is known as the Inst-Gen calculus), the formula can also grow exponentially. Of course this is no flaw in iProver but a direct consequence of the NEXPTIME-completeness of EPR and QF_BV2.

A different situation occurs when we look at QF_BV2$_{\ll 1}$. As seen in our introductory example, bit-blasting can still be exponential for formulas of this class. However, we know that it is possible to solve this kind of formulas in polynomial

space, since QF_BV2$_{\ll 1} \in$ PSPACE. In [100], we therefore presented a polynomial translation from QF_BV2$_{\ll 1}$ to SMV. Since current model checkers usually expect input in AIGER format, we then also translated our outputs to AIGER format using `smv2aig`, which is part of the AIGER library. Our experiments showed that, with growing bit-width, BDD-based model checkers (e.g., NuSMV [66] and IImc[5], using techniques described in [40, 43], with BDD-engine enabled) outperformed state-of-the-art SMT solvers on almost all of our benchmarks by orders of magnitude in runtime. Considering space requirements, the gain was even more significant. On the other hand, model checkers based on unrolling performed worse, and comparable to SMT solvers, on most benchmarks. This is not surprising, since unrolling to the full bit-width turns out to be the same as bit-blasting.

Altogether, our experiments show that the theoretical results given in [101, 151] and Section 4.6 can practically lead to improvements in state-of-the-art SMT solving. It is an interesting open problem to look at these results more closely and to integrate those concepts into SMT solvers in order to to increase their overall performance.

4.9.2 Benchmark Problems

Another practical outcome of our theoretical work was the creation of several different benchmark sets.

In [150], we proposed two new sets of QF_BV2 benchmarks for our experiments on evaluating the performance of EPR solvers for quantifier-free bit-vector formulas. In connection with our experiments on using model checkers for efficiently solving restricted bit-vector formulas, we generated six more benchmark sets for QF_BV2$_{\ll 1}$ in [100]. Another family of benchmarks was directly derived from the discussion on the expressiveness of bit-vector operations in this chapter. As we know from Section 4.6, all common bit-vector operations can be logarithmically expressed (in bit-width) by bitwise operations and equality in combination with *shift by constant*, *multiplication*, *concatenation*, or *slicing*. While we did not give direct translations for all common bit-vector operations in this work, we encoded most of them into SMT-LIB instances and used Boolector to verify their correctness for various bit-widths.

These benchmarks, together with those from [100, 150], can be found on our webpage[6] and will be submitted to the SMT-LIB. All of our benchmark sets are challenging for state-of-the-art SMT solvers (as well as for EPR solvers and model checkers) due to the fact that they are not scalar-bounded. For better solvers and future challenges, the difficulty of a problem can be adjusted by simply increasing the bit-widths in the original SMT-LIB instance. Bit-blasted versions of our benchmarks also turned out to be challenging for state-of-the-art SAT solvers in the SAT Competition 2013[7] [152].

[5] `http://ecee.colorado.edu/wpmu/iimc/`

[6] `http://fmv.jku.at/`

[7] `http://www.satcompetition.org/`

4.10 Conclusion

We discussed the complexity of deciding various quantified and quantifier-free fixed-size bit-vector logics. In contrast to existing literature, where usually it is not distinguished between *unary* and *binary encoding* on scalars in formulas, we argued that it is important to make this distinction. Most of our results apply to the actually much more natural binary encoding as it is also used in standard formats, e.g., in the SMT-LIB format. For this kind of logics, already the quantifier-free fragment without uninterpreted functions (QF_BV2) turned out to be NEXPTIME-complete [151].

In this chapter, we extended our previous work from [101, 151]. We first gave a detailed formal framework for fixed-size bit-vector logics including definitions for syntax and semantics. Our self-contained formalization is the first to consider different encodings and to provide a concrete measure for the size of bit-vector formulas as well as to provide the possibility to include arbitrary bit-vector operations.

Regarding the *Common Operator Framework*, as used, e.g., in the SMT-LIB format, we then revisited our previous complexity results from [101, 151] and extended those results in several ways. For quantifier-free logics, we combined our earlier work and restructured it to present several of our proofs in a clearer, easier-to-read way, with some small modifications in the proofs.

We then looked at several bit-vector operations and discussed their expressiveness, and checked if these operations can be logarithmically translated to each other (in bit-width). This kind of analysis helps to understand the complexity that is inherent in certain classes of bit-vector formulas and its relation to the kind of encoding used for bit-widths. While this allows us to check what kind of properties can be expressed in a given fragment, it also enables us to identify easier subclasses of formulas, which then can be solved more efficiently in practice by applying specialized algorithms.

Considering quantified logics, it is still an open question whether BV2 is complete for any complexity class. However, we could give some new results for quantified logics with a restriction on the bit-width of universal variables. We introduced the notion of *universally bit-width bounded* problems and showed that this kind of problems are in NEXPTIME. This then allowed us to prove that $\mathrm{BV2}_{\mathsf{log}}$ is NEXPTIME-complete. Since bit-vector logics with arrays represented by bit-vectors are in this set if quantification is only allowed on array indices, this class is of particular practical interest. For a last complexity theoretical result, we looked into $\mathrm{QF\_BV2}_{\mathcal{M}}$, the class of quantifier-free bit-vector logics extended with *non-recursive macros*, as allowed, e.g., in the SMT-LIB format. Again, we showed that this logic remains NEXPTIME-complete. Altogether, we provide the currently most complete overview on the complexity of common bit-vector logics.

To point out that our theoretical insights are also interesting from a practical point of view, we briefly discussed two approaches of solving bit-vector formulas not by bit-blasting but by using translations based on our complexity results. While

bit-blasting is exponential in general, we proposed polynomial translations into EPR and SMV in recent practical work [100, 150], to show that bit-vector solvers can indeed profit from our techniques. Several QF_BV2 benchmark families that we created throughout our work turned out to be challenging for state-of-the-art SMT and SAT solvers

For future work, it is still an interesting topic to consider our results in the context of parametrized complexity [85]. In particular, our definitions of (polynomially) *scalar-bounded* and *universally bit-width bounded* problem sets might be of relevance in this context. So far, mainly problems in NP are considered in parametrized complexity. This is another reason why extending our work in this direction is of special interest. Also, as already mentioned, the complexity of BV2 is still another open problem. Finally, from the practical side, it would be interesting to investigate how state-of-the-art SMT solvers can profit most from our insights and techniques.

4.11 Appendix

4.11.1 Example: Reduction from DQBF to QF_BV$2_{\ll c}$

Consider the following DQBF:

$$\begin{aligned}\forall u_0, u_1, u_2 \exists x(u_0), y(u_1, u_2)\,.\; & (x \vee y \vee \neg u_0 \vee \neg u_1) \wedge \\ & (x \vee \neg y \vee u_0 \vee \neg u_1 \vee \neg u_2) \wedge \\ & (x \vee \neg y \vee \neg u_0 \vee \neg u_1 \vee u_2) \wedge \\ & (\neg x \vee y \vee \neg u_0 \vee \neg u_2) \wedge \\ & (\neg x \vee \neg y \vee u_0 \vee u_1 \vee \neg u_2)\end{aligned}$$

This DQBF is *unsatisfiable*.

Using the reduction given in Lemma 4.23, this formula is translated to the following QF_BV$2_{\ll c}$ formula:

$$\begin{aligned}&\Big((X \mid Y \mid\!\sim U_0 \mid\!\sim U_1)\&(X \mid\!\sim Y \mid U_0 \mid\!\sim U_1 \mid\!\sim U_2)\&(X \mid\!\sim Y \mid\!\sim U_0 \mid\!\sim U_1 \mid U_2)\\ &\&(\sim X \mid Y \mid\!\sim U_0 \mid\!\sim U_2)\&(\sim X \mid\!\sim Y \mid U_0 \mid U_1 \mid\!\sim U_2)\Big) \;=\; \sim 0^{[8]} \;\wedge\\ &\bigwedge_{m \in \{0,1,2\}} U_m \ll 2^m \;=\; \sim U_m \;\wedge\\ &X \,\&\sim U_1 \;=\; (X \ll 2^1)\,\&\sim U_1 \;\wedge\\ &X \,\&\sim U_2 \;=\; (X \ll 2^2)\,\&\sim U_2 \;\wedge\\ &Y \,\&\sim U_0 \;=\; (Y \ll 2^0)\,\&\sim U_0\end{aligned} \tag{4.6}$$

In the following, let us show that this formula is also unsatisfiable.

Recall that the notation $t^{[n]} \overline{=} d$ is an alternative for $[\![t^{[n]}]\!] = d$, assuming an appropriate model for t. By construction, $U_0 \overline{=} 01010101$, $U_1 \overline{=} 00110011$, and

$U_2 \overline{\overline{=}} 00001111$.

First, we show how the bits of X get restricted by the constraints introduced above. Let us denote the originally unrestricted bits of X with $x_7, x_6, \ldots, x_0$. Since the bit-vectors

$$\begin{aligned} X\& \sim U_1 &\overline{\overline{=}} (x_7, x_6, 0, 0, x_3, x_2, 0, 0) \\ (X \ll 2^1)\& \sim U_1 &\overline{\overline{=}} (x_5, x_4, 0, 0, x_1, x_0, 0, 0) \end{aligned}$$

are forced to be equal, some bits of X should coincide, as follows:

$$X \overline{\overline{=}} (x_5, x_4, x_5, x_4, x_1, x_0, x_1, x_0)$$

Furthermore, considering also the equality

$$\begin{aligned} X\& \sim U_2 &\overline{\overline{=}} (x_7, x_6, x_5, x_4, 0, 0, 0, 0) \\ (X \ll 2^2)\& \sim U_2 &\overline{\overline{=}} (x_3, x_2, x_1, x_0, 0, 0, 0, 0) \end{aligned}$$

results in

$$X \overline{\overline{=}} (x_1, x_0, x_1, x_0, x_1, x_0, x_1, x_0)$$

In a similar fashion, the bits of Y are constrained as follows:

$$Y \overline{\overline{=}} (y_6, y_6, y_4, y_4, y_2, y_2, y_0, y_0)$$

In order to show that the formula (4.6) is unsatisfiable, let us evaluate the "clauses" in the formula:

$$\begin{aligned} X \mid Y \mid\sim U_0 \mid\sim U_1 &\overline{\overline{=}} (1, 1, 1, x_0 \vee y_4, 1, 1, 1, x_0 \vee y_0) \\ X \mid\sim Y \mid U_0 \mid\sim U_1 \mid\sim U_2 &\overline{\overline{=}} (1, 1, 1, 1, 1, 1, x_1 \vee \neg y_0, 1) \\ X \mid\sim Y \mid\sim U_0 \mid\sim U_1 \mid U_2 &\overline{\overline{=}} (1, 1, 1, x_0 \vee \neg y_4, 1, 1, 1, 1) \\ \sim X \mid Y \mid\sim U_0 \mid\sim U_2 &\overline{\overline{=}} (1, 1, 1, 1, 1, \neg x_0 \vee y_2, 1, \neg x_0 \vee y_0) \\ \sim X \mid\sim Y \mid U_0 \mid U_1 \mid\sim U_2 &\overline{\overline{=}} (1, 1, 1, 1, \neg x_1 \vee \neg y_2, 1, 1, 1) \end{aligned}$$

By applying *bitwise and* to them, we get the bit-vector constrained by the formula (4.6):

$$t \overline{\overline{=}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ (x_0 \vee \neg y_4) \wedge (x_0 \vee y_4) \\ \neg x_1 \vee \neg y_2 \\ \neg x_0 \vee y_2 \\ x_1 \vee \neg y_0 \\ (x_0 \vee y_0) \wedge (\neg x_0 \vee y_0) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ x_0 \\ \neg x_1 \vee \neg y_2 \\ \neg x_0 \vee y_2 \\ x_1 \vee \neg y_0 \\ y_0 \end{pmatrix}$$

In order to check if $t =\sim 0^{[8]}$ is satisfiable, it is sufficient to try to satisfy the set of the above Boolean clauses. It is easy to see that this clause set is unsatisfiable, since, by unit propagation, x_1 and y_2 must be assigned to 1, which contradicts with the clause $\neg x_1 \vee \neg y_2$.

4.11.2 Example: Reduction from QBF to QF_BV$2_{\ll 1}$

Consider the following QBF:

$$\begin{aligned}\exists x \forall u_2 \exists y \forall u_1 u_0 \exists z \,.\, & (u_2 \vee u_1 \vee \neg z) \wedge \\ & (u_2 \vee \neg x \vee y) \wedge \\ & (u_0 \vee \neg x \vee \neg z) \wedge \\ & (u_1 \vee \neg y \vee z) \wedge \\ & (u_0 \vee \neg u_1 \vee z)\end{aligned}$$

This QBF is *satisfiable*.

Using the reduction given in Lemma 4.24, this formula is translated to the following QF_BV$2_{\ll 1}$ formula:

$$\begin{aligned}&\Big((U_2 \mid U_1 \mid\!\sim Z) \& (U_2 \mid\!\sim X \mid Y) \& (U_0 \mid\!\sim X \mid\!\sim Z) \& \\ &(U_1 \mid\!\sim Y \mid Z) \& (U_0 \mid\!\sim U_1 \mid Z)\Big) \;=\; \sim 0^{[8]} \;\wedge \\ &\bigwedge_{m \in \{0,1,2\}} \Big(\bigwedge_{0 \leqslant i < m} U_i \Big) \oplus U_m \;=\; U_m \ll 1 \;\wedge \\ &X \& \sim 1 \;=\; X \ll 1 \;\wedge \\ &\Big(U_2' \;=\; \sim\big((U_2 \ll 1) \oplus U_2\big)\Big) \;\wedge\; \big(Y \& U_2' \;=\; (Y \ll 1) \& U_2'\big)\end{aligned} \tag{4.7}$$

In the following, let us show that this formula is also satisfiable. As in the previous example, we have $U_0 \,\overline{=}\, 01010101$, $U_1 \,\overline{=}\, 00110011$, and $U_2 \,\overline{=}\, 00001111$. However, this time the binary magic numbers were created in a different way to ensure that only addition and bitwise operations are used.

First, we show how the bits of X get restricted by the constraints introduced above. Let us denote the originally unrestricted bits of X with $x_7, x_6, \ldots, x_0$. Since the bit-vectors

$$\begin{aligned} X \& \sim 1 \quad &\overline{=} \quad (x_7, x_6, x_5, x_4, x_3, x_2, x_1, 0) \\ X \ll 1 \quad &\overline{=} \quad (x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0)\end{aligned}$$

must be equal, all bits of X are forced to be equal:

$$X \,\overline{=}\, (x_0, x_0, x_0, x_0, x_0, x_0, x_0, x_0)$$

Similarly, we get some constraints on Y. By using the mask

$$U_2' \;=\; \sim\big((U_2 \ll 1) \oplus U_2\big) \,\overline{=}\, 11101110$$

the following bit-vectors

$$\begin{aligned} Y \& U_2' \quad &\overline{=} \quad (y_7, y_6, y_5, 0, y_3, y_2, y_1, 0) \\ (Y \ll 1) \& U_2' \quad &\overline{=} \quad (y_6, y_5, y_4, 0, y_2, y_1, y_0, 0)\end{aligned}$$

are forced to be equal, putting restrictions on the individual bits of Y:

$$Y \overline{\overline{=}} (y_4, y_4, y_4, y_4, y_0, y_0, y_0, y_0)$$

Finally, Z is not restricted in any way since u_0 is the innermost universal variable that z depends on, i.e., z depends on all universal variables.

$$Z \overline{\overline{=}} (z_7, z_6, z_5, z_4, z_3, z_2, z_1, z_0)$$

In order to show that the formula (4.7) is satisfiable, we evaluate the "clauses" in the formula:

$$\begin{aligned}
U_2 \mid U_1 \mid\sim Z &\overline{\overline{=}} (\neg z_7, \neg z_6, 1, 1, 1, 1, 1, 1) \\
U_2 \mid\sim X \mid Y &\overline{\overline{=}} (\neg x_0 \vee y_4, \neg x_0 \vee y_4, \neg x_0 \vee y_4, \neg x_0 \vee y_4, 1, 1, 1, 1) \\
U_0 \mid\sim X \mid\sim Z &\overline{\overline{=}} (\neg x_0 \vee \neg z_7, 1, \neg x_0 \vee \neg z_5, 1, \neg x_0 \vee \neg z_3, 1, \neg x_0 \vee \neg z_1, 1) \\
U_1 \mid\sim Y \mid Z &\overline{\overline{=}} (\neg y_4 \vee z_7, \neg y_4 \vee z_6, 1, 1, \neg y_0 \vee z_4, \neg y_0 \vee z_3, 1, 1) \\
U_0 \mid\sim U_1 \mid Z &\overline{\overline{=}} (1, 1, z_5, 1, 1, 1, z_1, 1)
\end{aligned}$$

By applying *bitwise and* to them, we get the bit-vector constrained by the formula (4.7):

$$t \overline{\overline{=}} \begin{pmatrix} \neg z_7 \wedge (\neg x_0 \vee y_4) \wedge (\neg x_0 \vee \neg z_7) \wedge (\neg y_4 \vee z_7) \\ \neg z_6 \wedge (\neg x_0 \vee y_4) \wedge (\neg y_4 \vee z_6) \\ (\neg x_0 \vee y_4) \wedge (\neg x_0 \vee \neg z_5) \wedge z_5 \\ \neg x_0 \vee y_4 \\ (\neg x_0 \vee \neg z_3) \wedge (\neg y_0 \vee z_4) \\ \neg y_0 \vee z_3 \\ (\neg x_0 \vee \neg z_1) \wedge z_1 \\ 1 \end{pmatrix} = \begin{pmatrix} \neg z_7 \wedge \neg y_4 \\ \neg z_6 \\ z_5 \\ \neg x_0 \\ \neg y_0 \vee z_4 \\ \neg y_0 \vee z_3 \\ z_1 \\ 1 \end{pmatrix}$$

$t =\sim 0^{[8]}$ can easily be satisfied, e.g., by setting

$$\begin{aligned}
z_7 = z_6 = y_4 = y_0 = x_0 = 0 \\
z_5 = z_1 = 1
\end{aligned}$$

Therefore,

$$\begin{array}{cccccccc}
U_0 & \overline{\overline{=}} & 01010101, & U_1 & \overline{\overline{=}} & 00110011, & U_2 & \overline{\overline{=}} & 00001111, \\
X & \overline{\overline{=}} & 00000000, & Y & \overline{\overline{=}} & 00000000, & Z & \overline{\overline{=}} & 00111111
\end{array}$$

is a possible satisfying assignment for the bit-vector formula.

4.11.3 Example: Bit-Width Reduction in QF_BV2_{bw}

Let

$$\Phi_0 := \left(x^{[100]} <_{\mathbf{u}} y^{[100]}\right) \wedge \left(z^{[50]} = w^{[50]}\right) \wedge \left(w^{[100]}[38] = y^{[100]}[72]\right)$$

be a bit-vector formula with maximal bit-width 100. Note that we now use decimal encoding on the scalars. The set of explicit indices in the formula is given by $I := \{38, 72\}$. We now generate Φ_1 by splitting all bit-vectors at the corresponding bit-indices. First, $x^{[100]} <_{\mathbf{u}} y^{[100]}$ is therefore replaced by

$$
\begin{aligned}
& \big({x'_{99:73}}^{[27]} <_{\mathbf{u}} {y'_{99:73}}^{[27]}\big) \\
\vee\ & \big({x'_{99:73}}^{[27]} = {y'_{99:73}}^{[27]}\big) \wedge \big(\neg {x'_{72}}^{[1]} \wedge {y'_{72}}^{[1]}\big) \\
\vee\ & \big({x'_{99:73}}^{[27]} = {y'_{99:73}}^{[27]}\big) \wedge \big({x'_{72}}^{[1]} \Leftrightarrow {y'_{72}}^{[1]}\big) \wedge \big({x'_{71:39}}^{[33]} <_{\mathbf{u}} {y'_{71:39}}^{[33]}\big) \\
\vee\ & \big({x'_{99:73}}^{[27]} = {y'_{99:73}}^{[27]}\big) \wedge \big({x'_{72}}^{[1]} \Leftrightarrow {y'_{72}}^{[1]}\big) \\
& \wedge \big({x'_{71:39}}^{[33]} = {y'_{71:39}}^{[33]}\big) \wedge \big(\neg {x'_{38}}^{[1]} \wedge {y'_{38}}^{[1]}\big) \\
\vee\ & \big({x'_{99:73}}^{[27]} = {y'_{99:73}}^{[27]}\big) \wedge \big({x'_{72}}^{[1]} \Leftrightarrow {y'_{72}}^{[1]}\big) \\
& \wedge \big({x'_{71:39}}^{[33]} = {y'_{71:39}}^{[33]}\big) \wedge \big({x'_{38}}^{[1]} \Leftrightarrow {y'_{38}}^{[1]}\big) \wedge \big({x'_{37:0}}^{[38]} <_{\mathbf{u}} {y'_{37:0}}^{[38]}\big)
\end{aligned}
$$

Next, $z^{[50]} = w^{[50]}$ is replaced by

$$\big({z'_{49:39}}^{[11]} = {w'_{49:39}}^{[11]}\big) \wedge \big({z'_{38}}^{[1]} \Leftrightarrow {w'_{38}}^{[1]}\big) \wedge \big({z'_{37:0}}^{[38]} = {w'_{37:0}}^{[38]}\big)$$

Finally, $w^{[100]}[38] = y^{[100]}[72]$ is replaced by

$${w'_{38}}^{[1]} \Leftrightarrow {y'_{72}}^{[1]}$$

Since we only have 11 relational operations in Φ_1, we can generate a bit-width reduced formula Φ_2 by replacing all bit-widths n in Φ_1 with $\min\{11, n\}$. We therefore replace the variables

$$
\begin{aligned}
& {x'_{99:73}}^{[27]}, {y'_{99:73}}^{[27]}, {x'_{71:39}}^{[33]}, {y'_{71:39}}^{[33]}, \\
& {x'_{37:0}}^{[38]}, {y'_{37:0}}^{[38]}, {z'_{37:0}}^{[38]}, {w'_{37:0}}^{[38]},
\end{aligned}
$$

by

$$
\begin{aligned}
& {x''_{99:73}}^{[11]}, {y''_{99:73}}^{[11]}, {x''_{71:39}}^{[11]}, {y''_{71:39}}^{[11]}, \\
& {x''_{37:0}}^{[11]}, {y''_{37:0}}^{[11]}, {z''_{37:0}}^{[11]}, {w''_{37:0}}^{[11]}
\end{aligned}
$$

respectively.

4.11.4 Example: Half-Shuffle and Expand

halfshuffle$\big(\overbrace{1101}^{t^{[4]}},\ 16\big)$ can be replaced with ${x_2}^{[16]}$, by adding the following assertions. First, *zero extension* is applied to the original vector:

$${x_0}^{[16]} \quad = \quad \mathit{ext}_{\mathbf{u}}\left(t^{[4]}, 12\right) \overline{=} \ 0000\ 0000\ 0000\ 1101$$

Now, in two iterations, the bits of $t^{[4]}$ are separated and moved to the distinct partitions of the extended vector:

$$\begin{aligned}
{x_1}^{[16]} &= \left({x_0}^{[16]} \mid \left({x_0}^{[16]} \ll 6\right)\right) \,\&\, \textit{binmagic}\,(2,\ 16) \\
&\overline{=} \ (0000\ 0000\ 0000\ 1101 \mid 0000\ 0011\ 0100\ 0000) \\
&\quad \,\&\, 0011\ 0011\ 0011\ 0011 \\
&= \ 0000\ 0011\ 0000\ 0001 \\
{x_2}^{[16]} &= \left({x_1}^{[16]} \mid \left({x_1}^{[16]} \ll 3\right)\right) \,\&\, \textit{binmagic}\,(1,\ 16) \\
&\overline{=} \ (0000\ 0011\ 0000\ 0001 \mid 0001\ 1000\ 0000\ 1000) \\
&\quad \,\&\, 0101\ 0101\ 0101\ 0101 \\
&= \ 0001\ 0001\ 0000\ 0001
\end{aligned}$$

The result now can be used for example in *expand*: $\textit{expand}\big(\overbrace{1101}^{t^{[4]}},\ 16\big)$ can be expressed as ${x'_2}^{[16]}$, by adding the following assertions:

$$\begin{aligned}
{x'_0}^{[16]} &= \textit{halfshuffle}\left(t^{[4]},\ 16\right) \overline{=} \ 0001\ 0001\ 0000\ 0001 \\
{x'_1}^{[16]} &= {x'_0}^{[16]} \mid \left({x'_0}^{[16]} \ll 1\right) \\
&\overline{=} \ 0001\ 0001\ 0000\ 0001 \mid (0010\ 0010\ 0000\ 0010) \\
&= \ 0011\ 0011\ 0000\ 0011 \\
{x'_2}^{[16]} &= {x'_1}^{[16]} \mid \left({x'_1}^{[16]} \ll 2\right) \\
&\overline{=} \ 0011\ 0011\ 0000\ 0011 \mid (1100\ 1100\ 0000\ 1100) \\
&= \ 1111\ 1111\ 0000\ 1111
\end{aligned}$$

4.11.5 Example: Multiplication

The multiplication $\overbrace{0011}^{{t_1}^{[4]}} \cdot \overbrace{0101}^{{t_2}^{[4]}}$ can be expressed as ${x_2}^{[16]}\,[3:0]$, by adding the following assertions. First, both bit-vectors are transformed by *selfconcat* and *expand* to quadratic size in order to generate all single-digit multiplications in one step by using *bitwise and*:

$$\begin{aligned}
x^{[16]} &= \textit{selfconcat}\,(t_1,\ 16)\ \&\textit{expand}\,(t_2,\ 16) \\
&\overline{=} \ 0011\ 0011\ 0011\ 0011 \& 0000\ 1111\ 0000\ 1111 \\
&= \ \underbrace{0000}_{{g_3}^{[4]}}\ \underbrace{0011}_{{g_2}^{[4]}}\ \underbrace{0000}_{{g_1}^{[4]}}\ \underbrace{0011}_{{g_0}^{[4]}}
\end{aligned}$$

${g_3}^{[4]}, {g_2}^{[4]}, {g_1}^{[4]}$, and ${g_0}^{[4]}$ are the bit groups representing the bit-vector which is the result of single-digit multiplication of ${t_1}^{[4]} = 0011$ with the single bits of

${t_2}^{[4]} = 0101$. Now, the neighbouring groups have to be shifted to their relative offsets and are added:

$$\begin{aligned}
{b_0}^{[16]} &= \textit{binmagic}\,(4,\ 16) \overline{=} 0000\ 1111\ 0000\ 1111 \\
{x_1}^{[16]} &= (x_0 \& b_0) + \big((x_0 \& \sim b_0) \gg_{\mathbf{u}} 3\big) \\
&\overline{=} (0000\ 0011\ 0000\ 0011) + (0000\ 0000\ 0000\ 0000 \gg_{\mathbf{u}} 3) \\
&= \underbrace{0000\ 0011}_{{g_{32}}^{[8]}}\ \underbrace{0000\ 0011}_{{g_{10}}^{[8]}}
\end{aligned}$$

${g_{32}}^{[8]}$ and ${g_{10}}^{[8]}$ are the bit groups representing the bit-vectors which would be obtained by adding the bit-vectors represented by ${g_3}^{[4]}$, ${g_2}^{[4]}$ and ${g_1}^{[4]}$, ${g_0}^{[4]}$, respectively. This involves respecting their relative offsets, i.e., $g_{32} = (g_3 \ll 1) + g_2$ and $g_{10} = (g_1 \ll 1) + g_0$.

Since we still have several partial results, we have to continue adding neighbouring groups:

$$\begin{aligned}
{b_1}^{[16]} &= \textit{binmagic}\,(8,\ 16) \overline{=} 0000\ 0000\ 1111\ 1111 \\
{x_2}^{[16]} &= (x_1 \& b_1) + \big((x_1 \& \sim b_1) \gg_{\mathbf{u}} 6\big) \\
&\overline{=} (0000\ 0000\ 0000\ 0011) + (0000\ 0011\ 0000\ 0000 \gg_{\mathbf{u}} 6) \\
&= \underbrace{0000\ 0000\ 0000\ 1111}_{{g_{3210}}^{[16]}}
\end{aligned}$$

After the last step, there is only one bit group left and the least significant bits of the bit-vector ${x_2}^{[16]} \overline{=} 0000\ 0000\ 0000\ 1111$ correspond to the solution of the multiplication, i.e., $0011 \cdot 0101 = {x_2}^{[16]}\,[3:0] \overline{=} 1111$.

Further examples for multiplication or for other operations can easily be generated by feeding our benchmark family of bit-vector operations encoded in the SMT-LIB format into an SMT solver.

Chapter 5

A DPLL Algorithm for Solving DQBF

Published. In Proceedings International Workshop on Pragmatics of SAT (POS 2012), Affiliated to SAT 2012, Trento, Italy, 2012, Informal Proceedings [99].

Authors. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere.

Abstract. Dependency Quantified Boolean Formulas (DQBF) comprise the set of propositional formulas which can be formulated by adding Henkin quantifiers to Boolean logic. We are not aware of any published attempt in solving this class of formulas in practice. However, with DQBF being NEXPTIME-complete, efficient ways of solving it would have many practical applications. In this chapter, we describe a DPLL-style approach (DQDPLL) for solving DQBF. We show how methods successfully applied in similar algorithms for SAT/QBF can be lifted to this richer logic. This enables to reuse efficient SAT and QBF solving techniques.

5.1 Introduction

Dependency Quantified Boolean Formulas (DQBF), as first defined in [187], are obtained by adding Henkin quantifiers [122] to Boolean formulas. In contrast to QBF, the dependencies of a variable in DQBF are not implicitly defined by the order of the quantifier prefix, but are explicitly specified. The dependencies, therefore, are not forced to represent a total order, but only a partial one.

While QBF is PSPACE-complete [185], DQBF was shown to be NEXPTIME-complete [187, 188]. Because of this, DQBF offers more succinct descriptions than QBF, provided that the two classes do not collapse. Apart from DQBF, many practical problems are known to be NEXPTIME-complete, e.g., partial information non-cooperative games [188] or certain bit-vector logics [151, 234] used in the context of Satisfiability Modulo Theories (SMT).

There have been theoretical results on succinct formalizations using DQBF

and certain subclasses, e.g., DQBF-Horn has been shown to be solvable in polynomial time [55]. However, we are not aware of any description on solving DQBF problems in practice, nor any actual implementation of a decision procedure for DQBF. More recently, formula expansion and transformations specific to QBF have been discussed [9], which stayed only on the theoretical side but might yield an expansion-based DQBF solver, similar to those existing for QBF [24].

Effectively Propositional Logic (EPR) is another class of problems, for which the decision problem is NEXPTIME-complete. Thus, there exist polynomial reductions from DQBF to EPR, and vice versa. Consequently, it is also possible to use EPR solvers, e.g., iProver [147] being the currently most successful one, to solve DQBF, given some translation from DQBF to EPR. However, since EPR solvers, in general, have to reason with predicates and larger domains, solvers directly working on the propositional level should have advantages when DQBF formalizations of a problem are more natural.

Implementations of the DPLL algorithm [78], and improved variants, commonly known as CDCL solvers [170], are successfully used in many industrial applications. Inspired by the success of modern SAT solvers, similar algorithms have been developed for QBF, extending the algorithm by quantifier-reasoning and new concepts like cube learning. Although modern QBF solvers do not reach the performance of their SAT-counterparts yet, their capability also increased considerably in recent years.

This success of DPLL-style algorithms in the context of SAT and QBF gives reason to investigate how a similar algorithm could be adapted to DQBF. In the following, we propose a DPLL-style [78] algorithm (DQDPLL) for solving DQBF.

5.2 Definitions

Let V be a set of propositional variables. A *literal* l is a variable $x \in V$ or its negation $\neg x$, and let $x = var(l)$ denote its variable. A $clause\ C$ is a disjunction of literals. A propositional formula ϕ is in *conjunctive normal form (CNF)*, if it is a conjunction of clauses. A DQBF ψ can always be expressed as

$$\psi = Q.\phi = \forall u_1, \ldots, u_m \exists e_1(u_{1,1}, \ldots, u_{1,m_1}), \ldots, e_n(u_{n,1}, \ldots, u_{n,m_n}).\phi\ ,$$

with Q being the quantifier prefix and ϕ being a propositional formula (matrix) in CNF over the variables $V := U \cup E$ and $U = \{u_1, \ldots, u_m\}$, $E = \{e_1, \ldots, e_n\}$, $u_{i,j} \in U\ \forall\ i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\}$. In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given and not implicitly defined by the order of the prefix (in contrast to QBF).

Given an existential variable e_i, we use $dep(e_i) := \{u_{i,1}, \ldots, u_{i,m_i}\}$ to denote its dependencies. For universal variables u, we define $dep(u) := \varnothing$. We also extend the notion of dependency to literals, defining $dep(l) := dep(var(l))$ for

any literal l. An *assignment* is a mapping $\alpha : V \rightarrow \{true, false\}$ from the variables of a formula to truth values. Similarly, a *partial assignment* is a mapping $\beta : V \rightarrow \{true, false, undef\}$. To simplify the notation, we extend the definition of assignments and partial assignments to literals, clauses and formulas in the natural way. In the rest of this chapter, $\alpha(l)$, $\alpha(C)$, and $\alpha(F)$ will denote the truth value a literal l, a clause C, and a formula F, take under the assignment α, respectively. We extend the notation for partial assignments β in the same way defining $undef \vee true := true$, $undef \wedge true := undef$, $undef \vee false := undef$, and $undef \wedge false := false$.

A propositional formula ϕ in CNF is satisfiable, if and only if all clauses in ϕ are satisfied by at least one assignment α. We then call α a model of ϕ. In QBF and DQBF, a model cannot be expressed by a single assignment. We use *assignment trees* [197] instead, more precisely the variant of [166]. Given a DQBF ψ, an assignment tree T is a tree with the following attributes: Every node N in T, except the root, represents a truth assignment to a variable. A node has a sibling (exactly one representing the opposite truth value) if and only if it assigns a truth value to a universal variable.

Every path from the root to a leaf of T corresponds to an assignment α for the variables in ψ. In the same way, a path from the root to an internal node corresponds to a partial assignment β. Compared to QBF, there are two differences on the restrictions for possible trees:

Property 1: For every existential variable e and every universal variable u, such that $u \in dep(e)$, the node N_u for u must be an ancestor of the node N_e for e. This ensures that for every possible path and every node N_e for an existential variable, the variable is allowed to take different values for different assignments to its dependencies, since the assignment tree splits in the corresponding node N_u.

Property 2: For each pair of paths, with corresponding assignments α_1, α_2, it has to hold that $\alpha_1(e) = \alpha_2(e)$, if $\alpha_1(u) = \alpha_2(u)\ \forall\ u \in dep(e)$. This guarantees that an existential variable takes the same value in two distinct paths whenever its dependencies were assigned the same values in both paths.

A model for a DQBF $\psi = Q.\phi$, therefore, is an assignment tree that fulfills both property 1 & 2, and, at the same time, for each path from the root to a leaf, the corresponding assignment is a model for ϕ.

Actually, property 1 is not needed to make sure that ψ has a solution: There is a model respecting property 1 & 2 if and only if there is a model respecting only property 2. This follows from the fact that removing property 1 allows existential variables to move up in the assignment tree and, therefore, to be assigned even before all their dependencies are assigned, i.e., to remove some dependencies. However, removing dependencies makes a formula more difficult to satisfy, and, therefore, it is enough to consider satisfiability given property 1 & 2. This already rules out many assignment trees and yields a smaller search space.

```
1  procedure QDPLL(F)
2    while(true)
3      state = checkState(β);
4      if (state = STATE_UNSAT)
5        level = analyzeUNSAT();
6        if (level = 0) return UNSAT;
7        backtrack(level);
8      else if (state = STATE_SAT)
9        level = analyzeSAT();
10       if (level = 0) return SAT;
11       backtrack(level);
12     else
13       literal = selectLiteral();
14       β = updateAssignment(literal);
15       addDecision(literal);
```

Figure 5.1: Main loop of QDPLL as pseudo-code

5.3 DQDPLL Architecture

In the following, we assume that the reader is familiar with the design of a DPLL solver for SAT/QBF. Figure 5.1 shows the typical pseudo-code for a QBF solver based on the DPLL algorithm. In Figure 5.2, the pseudo-code of our adapted version for DQBF is presented. We will now discuss the DQDPLL algorithm in detail and point out the changes in specific methods compared to the original QBF-version.

The main underlying aspect when dealing with DQBF is the concept of dependency. As described in the previous section, a model for a DQBF exists if and only if there is an assignment tree where all paths satisfy the propositional matrix and, at the same time, the tree respects the restrictions defined by the underlying variable dependencies given in the prefix.

Instead of constructing arbitrary assignment trees, and at the end checking whether they fulfill the dependency restrictions (property 1 & 2), our algorithm will only construct the subset of assignment trees that does respect those restrictions.

Given a partial assignment tree, *selectLiteral* decides on the next node to branch on. An arbitrary selection heuristic can be used for doing so as long as it preserves property 1 of our assignment tree. This means a universal variable can be chosen at any time and an existential variable e can be chosen whenever all $u \in dep(e)$ are already assigned in the current path of our tree. Compared to QDPLL, this gives more possible decisions in each step, even given a QBF as an input, since decisions on existential variables may always be "delayed".

```
1 procedure DQDPLL(F)
2   while(true)
3     state = checkState(β);
4     if (state = STATE_UNSAT)
5       level = analyzeUNSAT();
6       if (level = 0) return UNSAT;
7       backtrack(level);
8     else if (state = STATE_SAT)
9       level = analyzeSAT();
10      if (level = 0) return SAT;
11      restoreAssignment(level);
12    else
13      literal = selectLiteral();
14      skolemClause = generateSkolemClause(β, literal);
15      β = updateAssignment(literal);
16      addDecision(beta, skolemClause);
17
18 procedure backtrack(level)
19   while (stack.Size > level) popStack();
20   (β, _) = stack.Element(level);
21
22 procedure restoreAssignment(level)
23   (β, _) = stack.Element(level);
24
25 procedure addDecision(β, skolemClause)
26   pushStack(β, skolemClause);
```

Figure 5.2: Main methods of DQDPLL as pseudo-code

Now we have to ensure that the constructed assignment tree also fulfills property 2 from the previous section. In our DQDPLL-approach, it is possible that an existential variable is set after a universal variable on which it does not depend. This cannot be avoided since we enforce a total order on the variables by our assignment tree whereas the dependency scheme of a DQBF is only partially ordered. To make sure that our assignment tree, nevertheless, fulfills property 2 we, therefore, have to "remember" the choice for an existential variable under a certain assignment of its dependencies. It will then be forced to take the same value in all other branches of the tree which imply the same assignment to those universal variables.

In our algorithm, this happens in the *addDecision* method. While the QDPLL algorithm only has to save the literal that was assigned during a deci-

sion, the DQDPLL algorithm additionally saves a *Skolem clause* C_{sk} linked with the branch on the literal of an existential variable on the decision stack. For a decision on a universal variable, no Skolem clause is added (i.e., $C_{sk} = \mathit{true}$ in the context of our pseudo-code). The Skolem clause added for an existential decision corresponds to the restriction implied for future branches due to property 2.

Note that, in our pseudo-code for DQDPLL, we do not push the branching literal on the decision stack, but instead push the current assignment β. Of course we could at any point reconstruct the branching literal from two consecutive assignments or the other way round, reconstruct an assignment from the sequence of branching literals. We have chosen to use the notation of storing assignments in our pseudo-code because this will simplify *backtrack* and *restoreAssignment*. In a real implementation, however, a version saving only the branching literals probably is a better choice since it reduces the memory requirement by a factor corresponding to the number of variables.

The Skolem clause C_{sk} linked with the decision can be constructed as follows: Let β be the partial assignment corresponding to the path from the root to the current branching node and let l_{e_i} be the branching literal with $var(l_{e_i}) = e_i$, $dep(e_i) = \{u_{i,1}, \ldots, u_{i,m_i}\}$, then:

$$C_{sk} := (l_{i,1}, \ldots, l_{i,m_i}, l_{e_i}),\ l_{i,j} = \begin{cases} u_{i,j}, & \text{if } \beta(u_{i,j}) = \mathit{false} \\ \neg u_{i,j}, & \text{if } \beta(u_{i,j}) = \mathit{true} \end{cases}$$

Since we only are allowed to branch on e_i if all $u_{i,j} \in dep(e_i)$ have already been assigned, we know that $\beta(u_{i,j}) \neq \mathit{undef}$, i.e., C_{sk} is well-defined. Adding this C_{sk} to the formula ensures that e_i will take the same value in all other paths of the tree where all $u_{i,j} \in dep(e_i)$ are assigned the same way as in the current path, i.e., property 2 is preserved. We decided to name this a Skolem clause because it corresponds to a partial definition of the Skolem function associated with an existential variable.

It is important to note that, in each step, the current set of clauses consists of the original matrix conjuncted with all Skolem clauses added so far. Depending on whether *checkState* returns the current set of clauses to be satisfied, unsatisfied or undecided under the partial assignment corresponding to the current path, the algorithm continues by conflict handling, solution handling or just assigning further literals.

Whenever the current set of clauses is discovered to be *UNSAT*, a call to *analyzeConflict* returns an existential decision which can be flipped. In a naive implementation, this could be simply the last existential variable that was picked by a call to *selectLiteral*. During the following call to *backtrack*, all decisions up to that point are undone and the corresponding Skolem clauses are removed. The decision variable is set to the opposite value and a new Skolem clause representing the necessary constraint is introduced.

If, on the other hand, the current set of clauses is *SAT* at some point, *analyzeSolution* returns a previous decision on a universal variable that still has to consider

the second branch. In a naive implementation, this could be just the latest universal variable that was picked by a call to *selectLiteral*, for which the second branch has not been checked yet. This condition should actually be considered as part of β in the pseudo-code. This time, however, in contrast to QDPLL, no backtracking takes place. Instead, *restoreAssignment* is called. This method restores the assignment to the one at the point of the decision but does not undo any decisions or remove any Skolem clauses. This is important because it means we keep the Skolem clauses over different universal branches and preserve property 2 of our assignment tree.

Note that, after calling *backtrack* as well as after calling *restoreAssignment*, the second branch at the corresponding level has to be checked. This is not explicitly specified in our pseudo-code, but, for simplicity, just is assumed to be part of *selectLiteral*.

Soundness and completeness of the algorithm can be checked easily:

Soundness: Altogether, the given specifications of the methods guarantee that every constructed assignment tree fulfills property 1 and property 2. Furthermore, the algorithm only returns *SAT* when all possible universal branches have been visited. This shows soundness of the DQDPLL-approach.

Completeness: Backtracking occurs as long as an existential variable can take a different value. The algorithm only returns *UNSAT* if no more backtracking is possible. Thus, in the worst case, all possible Skolem functions for all existential variables are enumerated, which implies completeness.

Aside from this, it is also easy to check runtime and space requirements of the proposed algorithm. Due to the fact that all possible Skolem functions are enumerated in the worst case, the runtime is double-exponential. This is no surprise considering that DQBF is NEXPTIME-complete. The space required is bounded exponentially. This corresponds to the size of the current assignment tree being checked for whether it is a solution to the formula.

There are several optimizations one can consider when implementing the proposed algorithm. For example, as already mentioned, it is not necessary to save the whole assignment on the stack for each decision, but, instead, one can only use the decision literal and later reconstruct previous assignments during *backtrack* and *restoreAssignment*. This is a bit more complicated as it is in QBF, since sometimes several universal branches have to be considered and, therefore, variables might first get unassigned and then reassigned again to exactly reconstruct the assignment in a certain state. Still, this is quite straightforward to implement but was neglected here in order to keep the pseudo-code easier to read.

Further optimizations are possible which do not backtrack in a linear way, but take advantage of the underlying tree-structure, instead of iterating through the whole stack. This again is neglected here to improve readability. As a low level optimization, it is not the focus of this chapter. In the next section, we will look at different concepts used in DPLL algorithms for SAT/QBF and describe how they can be adapted to be used in the DQDPLL-framework.

5.4 Conversion of Concepts from SAT/QBF

Having described the general design of DQDPLL, we now want to investigate if and how several techniques used in DPLL algorithms for SAT/QBF can be converted to the DQBF context. During the last decades, many concepts have been introduced to speed up DPLL algorithms for SAT, and many of those concepts have later been adapted to QBF. Some of these are *unit propagation*, *pure literal reduction*, and *clause learning*. Additionally, there were also concepts especially defined for QBF, e.g., *universal reduction* and *cube learning*. Apart from these, *selection heuristics* and *watched literal schemes* also play an important role in the performance of various solvers in those domains. In this section, we will describe how those concepts can be used for DQBF.

Unit Propagation. As one of the most important techniques used in DPLL-style algorithms for SAT and QBF, unit propagation is usually implemented as part of *checkState*, which is then often referred to as *Boolean Constraint Propagation (BCP)*.

Consider a clause $C = (l_1, \ldots, l_k)$ and a partial assignment β, so that $\exists j \in \{1, \ldots, k\} : \beta(l_j) = \textit{undef}, \beta(l_i) = \textit{false}\ \forall\, i \in \{1, \ldots, k\} \backslash \{j\}$. For SAT, $\beta(l_j)$ can then be set to *true*. For QBF, $\beta(l_j)$ can be set to *true*, if $var(l_j)$ is an existential variable, and *checkState* returns *STATE_UNSAT* otherwise. The latter one also trivially holds for DQBF, following the arguments used in the QBF version.

However, in the case of an existential variable being assigned because of unit propagation, there are the following aspects we have to consider: In contrast to selecting an existential variable e due to a decision, it is possible that not all $u \in dep(e)$ have been assigned yet when it gets propagated. Assigning e before all $u \in dep(e)$ are assigned, actually, violates property 1 defined in Section 5.2. Nevertheless, it is still sound to do so and will help pruning the search tree.

We already argued in Section 5.2 that property 1 only was added to prevent the algorithm from constructing irrelevant assignment trees, since an assignment tree not respecting property 1 corresponds only to an under-approximation of the original formula and does not preserve satisfiability. In the case of unit propagation, the last observation is not true any more. If unit propagation on e is possible under a certain partial assignment β, then the same unit propagation step is possible under all possible partial assignments β' which can be constructed from β by assigning all remaining variables $u \in dep(e), \beta(u) = \textit{undef}$. This means that assigning a unit e earlier, i.e., before all the universal variables on which it depends are assigned, does not violate any dependency restrictions of e. Actually, the same effect occurs in QBF during propagation on an existential unit variable, if not all universal variables in the outer scopes are assigned yet.

In order to ensure property 2 of the assignment tree, a Skolem clause needs to be added for all possible remaining assignments of $\{u \in dep(e) \mid \beta(u) = \textit{undef}\}$. Using resolution and subsumption, this can be expressed by adding only

one clause:

$$C_{sk} := (l_{i,1}, \ldots, l_{i,m_i}, l_{e_i}),\ l_{i,j} = \begin{cases} u_{i,j}, & \text{if } \beta(u_{i,j}) = \textit{false} \\ \neg u_{i,j}, & \text{if } \beta(u_{i,j}) = \textit{true} \\ \textit{false}, & \text{if } \beta(u_{i,j}) = \textit{undef}, \end{cases}$$

assuming $var(l_{e_i}) = e_i,\ dep(e_i) = \{u_{i,1}, \ldots, u_{i,m_i}\}$.

Pure Literal Reduction. For universal variables, pure literal reduction can be implemented exactly as it is done for QBF. Whenever a pure universal literal l_u is found, it can be set to *false*. To see that this procedure is sound, one can move the concerned universal variable outwards and expand it [9]. It is enough to consider the part where the literal is set to *false* since it subsumes the other part.

For existential variables, this becomes more complicated and there is no dual version, as there exists for QBF. The reason for this is the following: setting a pure existential literal to *true* does not guarantee to preserve satisfiability since it adds a new Skolem clause to the formula (i.e., restricts the solutions), which might force the literal to the same value in some later branch of the assignment tree, although the literal is not pure there anymore. In QBF, this was possible because all branches of the decision tree were independent of each other.

To guarantee that pure literal reduction on existential variables remains sound for DQBF, it can only be applied under certain conditions: An existential literal l_{e_i} can be set to *true* if every clause containing $\neg l_{e_i}$ is already satisfied by at least one $l_u,\ var(l_u) \in dep(l_{e_i})$, or by an existential literal $l_{e_j},\ dep(l_{e_j}) \subseteq dep(l_{e_i})$. In this case, we know that all clauses containing $\neg l_{e_i}$ are already satisfied whenever the newly added Skolem clause propagates l_{e_i}. This means l_{e_i} is still pure whenever the Skolem clause propagates, and, therefore, the Skolem clause does not put an additional restriction on the original formula.

Clause Learning. Adding clause learning to DPLL-based SAT algorithms is responsible for a huge performance improvement of SAT solvers during the last two decades, particularly in the combination with *conflict driven clause learning* (CDCL) solvers [170]. Clause learning was then also applied to QBF [114, 160, 238]. In SAT as well as in QBF, it often allows to prune large parts of the search tree.

It turns out that conflict clauses in DQDPLL can be generated in the same way as it was done for QBF, and originally for SAT. The simple reason is that clause learning is based on (propositional) resolution and, therefore, can be applied on the matrix level, totally ignoring variable dependencies. Any resolvent of two clauses can be added to a formula without affecting satisfiability. In SAT/QBF it is common to perform resolution with clauses on the decision stack while backtracking. It can be shown that, like this, the conflict can be resolved and the new clause is asserting under the current assignment after backtracking.

However, if clause learning is applied in the same way in DQDPLL, it is possible that Skolem clauses are used for resolution. The resulting resolvent, therefore,

is only valid as long as all Skolem clauses used to create it are still part of the formula. Because of this, we need to differentiate between *temporary learned clauses* and *permanent learned clauses*.

Any learned clause created by resolution with at least one Skolem clause or with a temporary learned clause is only valid as long as all clauses participating in the resolution steps are still part of the formula, and will be a temporary learned clause itself. Therefore, it will be linked with the latest such clause and is removed whenever the linked clause is. A permanent learned clause is created when no Skolem clause and no temporary learned clause was part of the resolution process applied during backtracking. A clause like this can be kept or removed at any point, in the same way as it is done in SAT/QBF.

Apart from this, it is also possible to create a permanent clause during backtracking if there are Skolem clauses or temporary learned clauses participating in the conflict. The algorithm can just skip the resolution steps with those clauses and, of course, ends up with a permanent clause. However, in this case, it is not guaranteed that the resulting clause is asserting under the current assignment after backtracking, and the permanent learned clause is less restrictive than the corresponding temporary learned clause.

It is, therefore, reasonable to generate both types of clauses in order to profit from the individual advantages. The temporary clause will prune larger parts of the current search tree, while the permanent clause can still affect other parts of the search tree whenever the temporary clause gets removed during further backtracking. If the permanent learned clause is too weak and does not contribute, it can be automatically deleted if removal schemes, like those proposed in [116], are used.

Universal Reduction. This can be adopted for DQBF in a straightforward way. Consider a universal variable u and a clause $C = (l_u, l_1, \ldots, l_k)$, and let $var(l_u) = u$, $\beta(l_i) \neq \mathit{true}\ \forall\, l_i \in C$. If $u \notin \bigcup_{\beta(l_i)=\mathit{undef}} \{dep(l_i)\}$, l_u can be set to *false*.

This can be seen when considering the universal expansion of C considering u. Let $C_{l_u=v}$ be the clause obtained from C by setting l_u to $v \in \{\mathit{true}, \mathit{false}\}$. A solution for F has to satisfy $C_{l_u=\mathit{true}}$ and $C_{l_u=\mathit{false}}$. Since all variables that are contained in C and are still unassigned at the current node in the assignment tree do not depend on u, they have to take the same value in both $C_{l_u=\mathit{true}}$ and $C_{l_u=\mathit{false}}$. Since $C_{l_u=\mathit{true}}$ is already satisfied by l_u, only $C_{l_u=\mathit{false}}$ needs to be considered instead of C, i.e., l_u can be removed from C.

Cube Learning. Introduced for QBF in [114, 160, 239], cube (goods / solution) learning is used to prune satisfied branches of the assignment tree. It can be considered as the dual concept to clause (no goods) learning, creating so-called *cubes*, i.e., a subset of literals already satisfying the formula. A cube therefore is a conjunction of literals and is added to the formula by disjunction. *Initial cubes* are created from a satisfying assignment by extracting a minimal subset of literals necessary to satisfy it. Later, further cubes can be generated by using resolution on existing cubes, similar to the way new clauses are created when a conflict occurs.

The same principle can still be applied to DQBF since all reasoning for creating cubes is done on the matrix level. However, similar to the reasoning necessary for adapting clause learning, a cube in DQBF is not permanent in a certain sense. When a Skolem clause is added during a decision, the set of satisfying assignments for the formula matrix shrinks. Because of this, it is possible that a cube which was added to the DQBF in a previous step does not represent a satisfying assignment for the formula matrix anymore after adding additional Skolem clauses. Whenever a Skolem clause is added to the formula, the algorithm, therefore, has to check whether it is satisfied by the existing cubes. Cubes not satisfying the new clause are linked with the Skolem clause and get flagged "inactive". They are not removed from the formula because they can be flagged "active" again if the Skolem clause later gets removed during backtracking.

An important point to note is that reasoning with cubes changes compared to QBF. While unit propagation on universal variables in cubes is still sound, a cube only consisting of existential variables cannot considered to be satisfied in DQBF. The reasoning behind this is the same as for pure literal reduction. Setting the remaining existential variables in a cube to *true* implies restricting the formula by Skolem clauses, i.e., it might rule out solutions and therefore does not preserve satisfiability.

Selection Heuristics. An important aspect determining the performance of a SAT solver is given by its selection heuristic. A selection heuristic determines the order of the variables getting assigned and the value they first get assigned to. In SAT, there is a huge choice of different heuristics. Recently, the most common heuristics are VSIDS [176] and phase saving [192]. QBF solvers suffer from the fact that variable selection is much more restricted due to the total order defined by the quantifier prefix. Only variables from the current quantifier scope can be chosen. Sometimes, this constraint can be reduced by explicitly checking for dependencies between the different variables on the matrix level, as done for example by DepQBF [165]. Note that this is a different concept of dependency. While independence on the matrix level means that the result of the formula will be the same no matter which ordering for the variables is chosen, independence in the context of DQBF is a constraint forcing a variable to take consistent values on different branches of the assignment tree.

Since variable dependencies in DQBF are less strict and the design of DQDPLL allows to "delay" decisions on existential variables, this offers more freedom on the selection of variables compared to QBF. We, therefore, suggest that selection heuristics have more influence in the DQBF case. For our implementation, we used VSIDS [176] and phase saving [192] in the same way as it is done in SAT, but restricted to the set of possible candidates defined by the properties of our assignment trees. It might, however, also be interesting to extend heuristics for DQBF by incorporating information specified on the quantifier-level, e.g., preferring existential variables over universal variables, or picking those existential variables with dependencies most "similar" to the current universal assignment.

Watched Literal Schemes. The watched literal scheme, as a lazy data structure for unit literal detection, has proved itself to be efficient in SAT solving [237, 176]. The basic idea is that the clauses are kept untouched (i.e., no literals are ever removed), and, furthermore, the data structure does not require any update during backtracking. The watched literal scheme has been adapted also to QBF [108, 165]. In the *two literal watching scheme*, in each clause two literals l_1 and l_2 are watched, fulfilling the following invariant: l_1 is existential, and if l_2 is universal then $var(l_2) \in dep(l_1)$. Note that, in QBF, this latter condition about dependency only requires to check whether $var(l_2)$ is quantified before $var(l_1)$ in the prefix. This can be adapted to DQBF in a straightforward way, by checking the explicit dependencies of $var(l_1)$. It is important to initialize watchers on the fly for each fresh clause (i.e., conflict clause or Skolem clause). The detection of falsified, satisfied, and unit clauses can be done in the same way just like in QBF.

However, a special situation, right after backtracking, has to be considered: l_1 is assigned and $l_2 =$ *undef* is universal. In QBF solvers, or even in DQBF solvers respecting property 1, this situation cannot occur. However, when neglecting property 1, *backtracking to a previous path* might result in such a situation. Nevertheless, it is easy to improve the solver to avoid this situation: update the watchers of all the literals which are assigned by β, provided by the *backtrack* method. We would like to point out that this update could be highly optimized by the implementation optimization mentioned in Section 5.3, namely that only the branching literals should be saved on the decision stack instead of assignments. Given the current node n and the node n' to jump back to, let $lca(n, n')$ denote the lowest common ancestor of n and n'. During traversing the path from $lca(n, n')$ to n', update the watchers of the literals assigned by the touched nodes.

5.5 Preliminary Results

We implemented a prototype of our DQDPLL algorithm, as introduced in Section 5.3, and added all the concepts described in Section 5.4. Testing was rather difficult since there is no DQBF library yet nor any other DQBF solver to compare results with.

Since EPR is also NEXPTIME-complete, we used EPR formulas from the TPTP and converted the formulas to DQBF. Unfortunately, the conversion caused a large blow-up in the formula size. Bit-blasting of the domain, introduction of Ackermann constraints when removing predicates [158, Chapter 3.3.1], inverse destructive equality reasoning [234] to remove dependencies on other existential variables (which are not defined in DQBF), and final transformation to CNF led to an explosion in formula size. This blow-up, though being polynomial, produced formulas which were too large for our algorithm to solve.

Using QBF benchmarks as an input, we then compared our solver with DepQBF [165]. As expected, DepQBF was faster by several orders of magnitude, since it is much more specialized while our solver has additional exponential over-

head dealing with the stack of Skolem clauses which are not necessary for QBF. Nevertheless, we could check that the returned satisfiability status of all instances solved by our algorithm was equal to the one returned by DepQBF, and, therefore, QBF seems to be solved correctly.

To check whether DQBF instances can be solved at all, we wrote a tool for generating random DQBF with different parameters, including number of clauses, number of existential variables, number of universal variables and expected number of dependencies per existential. We then used medium sized instances (10-50 variables, 100-1000 clauses), generated by our tool, to check that our algorithm can deal with those problems and that it always produces consistent results during several hundred randomized runs. We also gernerated very small sized instances (2-6 variables) to check correctness on this subset manually.

A further way to check correctness could be obtained by translating our randomly generated DQBF to EPR, and then comparing our results with the results of an EPR solver on the converted benchmark, as done for QBF in [202].

5.6 Future Work

At the moment, our algorithm is not able to solve translated EPR instances and therefore cannot compete with EPR solvers. One reason is that there is a huge blow-up during conversion. A second explanation could be the fact that those instances often were especially created using the properties of EPR. It might be interesting to look for problems which have a natural representation as DQBF instead. Maybe, in domains that fit well to Boolean reasoning and do not directly suggest the usage of predicates, the use of a low level DPLL-style approach is better suited, and allows to profit from the well-established techniques already successful in SAT/QBF.

Apart from this, our solver is still a prototype and there are many possible optimizations concerning data structures, and other details related to implementation of our techniques, which we should consider in the future. We also do not use restarts yet. Regarding the proposed concepts, it will be interesting to analyze in detail, if and how each of them improves the performance of a DQBF solver based on our DQDPLL architecture.

It might also be of interest to create an expansion-based solver for DQBF and see how it would compare to a DPLL-style solver such as the one we proposed. Additionally, expansion also could be used to construct a QBF out of a DQBF by expanding universal variables until the quantifiers can be totally ordered. A QBF generated this way can be given to any DPLL-based QBF solver to see if our approach of applying the concepts directly on the more succinct DQBF level gives any benefits over dealing with the less succinct QBF representation.

Finally, considering the increased complexity compared to solvers for QBF and SAT, it becomes even more important to verify results. While the Skolem clauses on the decision stack after termination of our algorithm exactly define a Skolem

function representing a solution, it might be interesting to check if certificates for conflicts can be generated similar to how it is done for QBF [182].

5.7 Conclusion

In this chapter, we described DQDPLL, a DPLL-style algorithm for DQBF. We have formally defined necessary conditions for assignment trees representing solutions for DQBF. Based on this, we have also shown what adaptations of the DPLL-architecture to DQBF are necessary and how they can be implemented by introducing a stack of Skolem clauses, representing partial definitions of the Skolem functions defining the existential variables.

With the main reason for the success of DPLL algorithms in SAT and QBF being found in various techniques such as *unit propagation*, *pure literal reduction* and *clause learning*, *universal reduction*, *cube learning*, *selection heuristics* and *watched literal schemes*, we also discussed how these can be translated to DQBF.

Our implementation shows that it is indeed possible to solve DQBF with this approach, at the same time, however, it does not perform very well. We have given reasons for why this is the case for EPR formulas, and suggested to find problems which can be formalized in DQBF more naturally.

Since the introduction of DQBF in [187], our work contains the first detailed description of an algorithm to solve this class of problems. While still a lot of progress has to be made in this field, we hope that our contribution helps getting a better insight into the topic of DQBF, as well as into possibilities and pitfalls on the way of practically solving it.

Chapter 6

Bv2epr: A Tool for Polynomially Translating Quantifier-free Bit-Vector Formulas into EPR

Published. In Proceedings 24th International Conference on Automated Deduction (CADE-24), Lecture Notes in Computer Science (LNCS) volume 7898, pages 443–449, Springer 2013 [150].

Authors. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere.

Abstract. Bit-precise reasoning is essential in many applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. Most of these approaches rely on bit-blasting. In [151], we argued that bit-blasting is not polynomial in general, and then showed that solving quantifier-free bit-vector formulas (QF_BV) is NExpTime-complete. In this chapter, we present a tool based on a new polynomial translation from QF_BV into Effectively Propositional Logic (EPR). This allows us to solve QF_BV problems using EPR solvers and avoids the exponential growth that comes with bit-blasting. Additionally, our tool allows us to easily generate new challenging benchmarks for EPR solvers.

6.1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for fixed-sized bit-vector logics are Boolector, MathSAT, STP, Z3, and Yices. All these solvers rely on *bit-blasting* in order to translate bit-vector formulas into propositional logic (SAT). The result is then checked by a SAT solver.

In practice, e.g., in the SMT-LIB [18], the BTOR [48], and the Z3 format, the

bit-widths in bit-vector formulas are encoded as binary, decimal, or hexadecimal numbers, i.e., a *logarithmic encoding* is used. In [151], we proved that the encoding of bit-widths affects the complexity of the decision problem of bit-vector logics. In particular, logarithmic encoding makes the quantifier-free fragment QF_BV2 NEXPTIME-complete.[1] Thus, bit-blasting is *not polynomial* in general. For a polynomial reduction, the target logic has to be NEXPTIME-hard.

In this chapter, we introduce our new tool BV2epr. BV2epr translates QF_BV formulas into Effectively Propositional Logic (EPR), which is NEXPTIME-complete [161], by using a new (polynomial) reduction. This is in contrast to existing translations in [143, 90], which produce exponential EPR formulas in general, as we will point out in Section 6.2.1. We give some experimental results in Section 6.4 with the EPR solver iProver.

6.2 Preliminaries

We assume the usual syntax for QF_BV. A bit-vector term t of bit-width n ($n \in \mathbb{N}$, $n \geqslant 1$) is denoted by $t^{[n]}$. An atomic term can be either (a) a bit-vector *constant* $c^{[n]}$, where $c \in \mathbb{N}$, $0 \leqslant c < 2^n$; or (b) a bit-vector *variable* $v^{[n]}$. Compound terms and formulas can contain the usual bit-vector *operators* (cf. SMT-LIB [18]), e.g., bitwise operators, shifts, arithmetic operators, relational operators, etc. The decision problem for QF_BV is NEXPTIME-complete [151].

EPR, known as the Bernays-Schönfinkel class, is a NEXPTIME-complete fragment of first-order logic [161]. It corresponds to the set of first-order formulas that, written in prenex form, contain (a) no function symbol of arity greater than 0; and (b) no existential quantifier within the scope of a universal quantifier. After Skolemization, existential variables turn into constants (i.e., function symbols of arity 0), and quantifiers can be omitted. Consequently, an EPR *atom* can be defined as an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and each t_i is either a (universal) variable or a constant.

6.2.1 Existing Translations

In [143], encodings of hardware verification problems with bit-vectors into first-order logic are proposed. In particular, an encoding into EPR is given and called the *relational encoding* [90], since bit-vectors are modeled as unary predicates. These predicates are over bit-indices, represented by dedicated constants. For instance, the ith bit of a bit-vector $x^{[n]}$, $0 \leqslant i < n$, is represented by the atom $p_x(bitInd_i)$, where $bitInd_i$ is a constant. Note that for QF_BV2, such a translation might introduce exponentially many constants, since bit-widths like n are encoded logarithmically. The so-called *range-aware* relational encoding in [90], furthermore, introduces exponentially many assertions into the EPR formula in

[1] In [151], we introduced the notation QF_BV1 and QF_BV2 for QF_BV using a *unary* and a *logarithmic* (w.l.o.g., *binary*) encoding, respectively.

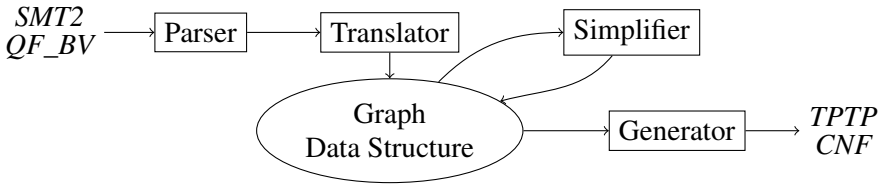


Figure 6.1: The architecture of Bv2epr

general, e.g., atoms $less_k(bitInd_i)$ for all $0 \leqslant i < k$. Finally, not all the QF_BV bit-vector operators are addressed by the relational encoding, but only *equality*.[2] All the *arithmetic operators* are assumed to be synthesized/bit-blasted in the verification front-end [90], potentially leading to an exponential blowup already before the actual encoding. In [143], an abstraction of *shifts* is proposed, which is, however, basically the same as bit-blasting. Consequently, the relational encoding is exponential in general, in contrast to our translation in Section 6.3.1.

6.3 The Tool

Bv2epr takes a QF_BV formula in SMT2 format as input, and outputs an EPR clause set in TPTP format. The tool is implemented in C and available at [59]. The architecture of Bv2epr can be seen in Figure 6.1, consisting of the following modules:

Parser. The Parser is Boolector's SMT2 parser.

Translator. The Translator provides an interface accessed by the Parser, in order to deal with the SMT2 QF_BV operators. This module builds a graph data structure, in which each bit-vector operation is modeled by an EPR predicate. Predicates are represented by shared nodes in the graph data structure. A node for a predicate p stores, besides other data, the functional definition of p as an EPR clause set. With each of these clauses, an argument list $i_{n-1}, \ldots, i_0$ for p is stored, indicating that this clause is part of the functional definition of the EPR atom $p(i_{n-1}, \ldots, i_0)$. Such a clause is realized as a list of EPR literals, each of which contains a reference to a predicate p' and an argument list for p'.

Simplifier. The graph constructed by the Translator is a good basis for various simplifications. Note that only polynomial simplification steps are acceptable. Among others, we implemented two kinds of simplification, both proposed in [125]: (a) *unused definition elimination* and (b) *non-growing definition inlining*.

Generator. Out of the (simplified) graph, this module generates a TPTP clause set. Since the graph might contain cycles, the Generator detects and avoids them. Due to the construction of the graph data structure, clauses can be extracted directly, i.e., no additional approach for clause generation is needed.

[2] Bitwise operators could be handled in a similar way.

6.3.1 The Translator

We briefly sketch the (polynomial) reduction of QF_BV to EPR used by the Translator, without striving for completeness. As it will turn out, the target logic of this reduction is actually not general EPR, but rather its fragment which uses only two constants, 0 and 1. We call this fragment EPR2.[3] To each bit-vector term of bit-width n, a dedicated $\lceil \log_2 n \rceil$-ary EPR2 predicate is introduced and assigned. For example, a term $x^{[32]}$ is represented by a 5-ary predicate p_x. Since p_x is an EPR2 predicate, each of its arguments can be either 0, 1, or a universal variable. For instance, the atom $p_x(1,1,0,0,1)$ represents the 25th bit of x, since $25_{10} = 11001_2$. Using universal variables as arguments makes it possible to represent several bits by a single EPR2 formula; for instance, the atom $p_x(i_4, i_3, i_2, i_1, 0)$ represents all even bits of x.

Bitwise Operators. Translating bitwise operators is quite natural. We demonstrate the translation for *bitwise or* (denoted by $|$): Given a term $x^{[2^n]} \mid y^{[2^n]}$, where x and y are bit-vector terms, to which the predicates p_x and p_y have already been assigned, respectively. We need to specify each bit of the resulting bit-vector as the disjunction of the corresponding bits of x and y. We introduce a new predicate p_{or}, and give the following functional definition:

$$p_{or}(i_{n-1}, \ldots, i_0) \quad \Leftrightarrow \quad p_x(i_{n-1}, \ldots, i_0) \vee p_y(i_{n-1}, \ldots, i_0)$$

Addition. Given a term $x^{[2^n]} + y^{[2^n]}$, let us first rewrite it to the following bit-vector equations, where $\oplus$ denotes *bitwise xor*, $\&$ *bitwise and*, and $\ll$ *left shift*.

$$add^{[2^n]} = x^{[2^n]} \oplus y^{[2^n]} \oplus cin^{[2^n]} \tag{6.1}$$

$$cin^{[2^n]} = cout^{[2^n]} \ll 1 \tag{6.2}$$

$$cout^{[2^n]} = (x^{[2^n]} \,\&\, y^{[2^n]}) \mid (x^{[2^n]} \,\&\, cin^{[2^n]}) \mid (y^{[2^n]} \,\&\, cin^{[2^n]}) \tag{6.3}$$

Note that Equation (6.1) and (6.3) only contain bitwise operators (and equality). Therefore, both can be translated into EPR2 as introduced previously. Only Equation (6.2), which contains *shift by 1*, has to be handled differently.

We introduce a helper predicate $succ$ which will represent the fact that a bit-index j is the successor of a bit-index i, i.e., $j = i + 1$. Since i is represented by an EPR2 argument list $i_{n-1}, \ldots, i_0$ and, similarly, j by $j_{n-1}, \ldots, j_0$, the $2n$-ary predicate $succ(i_{n-1}, \ldots, i_0,\ j_{n-1}, \ldots, j_0)$ can be defined by n facts:

$$\begin{array}{c} succ(i_{n-1}, \ldots, i_3, i_2, i_1, 0,\ i_{n-1}, \ldots, i_3, i_2, i_1,\ 1) \\ succ(i_{n-1}, \ldots, i_3, i_2,\ 0, 1,\ i_{n-1}, \ldots, i_3, i_2,\ 1,\ 0) \\ succ(i_{n-1}, \ldots, i_3,\ 0,\ 1, 1,\ i_{n-1}, \ldots, i_3,\ 1,\ 0,\ 0) \\ \vdots \\ succ(0, 1, \ldots, 1,\ 1, 0, \ldots, 0) \end{array}$$

Using this helper predicate, Equation (6.2) can be translated into EPR2 as follows:

$$\begin{array}{l} \neg p_{cin}(0, \ldots, 0) \\ succ(i_{n-1}, \ldots, i_0,\ j_{n-1}, \ldots, j_0) \ \Rightarrow \\ \qquad (p_{cin}(j_{n-1}, \ldots, j_0) \ \Leftrightarrow \ p_{cout}(i_{n-1}, \ldots, i_0)) \end{array} ,$$

[3] The Herbrand universe of EPR2 can be considered as the Boolean domain.

This kind of adder can be adapted to represent other arithmetic operators like *unary minus* and *subtraction*. In BV2EPR, all the relational operators, like *equality* and *unsigned less than*, are also represented by such an adapted adder.

Shifts. Shifts are translated into EPR2 by applying *barrel shift*. For instance, given a term $x^{[2^n]} \ll y^{[2^n]}$, for all bit-indices i, $0 \leqslant i < n$, the ith bit of y is checked: if it is 1, then *left shift by* 2^i has to be done.

$$\begin{array}{l}
\neg p_y(0,\ldots,0) \;\Rightarrow \\
\qquad \big(p^0_{shl}(i_{n-1},\ldots,i_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0)\big) \qquad , \\
\\
\left(\begin{array}{c} p_y(0,\ldots,0) \wedge \\ succ(i_{n-1},\ldots,i_0,\; j_{n-1},\ldots,j_0) \end{array}\right) \Rightarrow \\
\qquad \big(p^0_{shl}(j_{n-1},\ldots,j_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0)\big) \qquad , \\
\\
\neg p_y(0,\ldots,0,1) \;\Rightarrow \\
\qquad \big(p^1_{shl}(i_{n-1},\ldots,i_0) \Leftrightarrow p^0_{shl}(i_{n-1},\ldots,i_0)\big) \qquad , \\
\\
\left(\begin{array}{c} p_y(0,\ldots,0,1) \wedge \\ succ(0,i_{n-1},\ldots,i_1,\; 0,j_{n-1},\ldots,j_1) \end{array}\right) \Rightarrow \\
\qquad \big(p^1_{shl}(j_{n-1},\ldots,j_1,i_0) \Leftrightarrow p^0_{shl}(i_{n-1},\ldots,i_0)\big) \quad , \\
\qquad\qquad \vdots
\end{array}$$

Multiplication. The Translator applies a *shift-and-add* approach for translating a term $x^{[2^n]} \cdot y^{[2^n]}$. We generate 2^n subproducts of bit-width 2^n, and represent all of them by a single $2n$-ary predicate p_{mul}: the ith bit of the jth subproduct is represented by the atom $p_{mul}(j_{n-1},\ldots,j_0,\; i_{n-1},\ldots,i_0)$.

First, the (2^n-1)th subproduct is computed, by checking the most significant bit of y: if it is 0, this subproduct is set to 0; otherwise, it is set equal to x.

$$\begin{array}{rcl}
\neg p_y(1,\ldots,1) & \Rightarrow & \neg p_{mul}(1,\ldots,1,\; i_{n-1},\ldots,i_0) \\
p_y(1,\ldots,1) & \Rightarrow & (p_{mul}(1,\ldots,1,\; i_{n-1},\ldots,i_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0))
\end{array}$$

The jth subproduct, $0 \leqslant j < 2^n - 1$, is computed by checking the jth bit of y: if it is 0, then the $(j+1)$th subproduct has to be *shifted left by 1* (represented by the predicate p_{shl}); otherwise, the shifted subproduct and x have to be *added* (represented by p_{add}).

$$\begin{array}{l}
\left(\begin{array}{c} \neg p_y(j_{n-1},\ldots,j_0) \wedge \\ succ(j_{n-1},\ldots,j_0,\; j'_{n-1},\ldots,j'_0) \end{array}\right) \Rightarrow \\
\qquad \left(\begin{array}{c} p_{mul}(j_{n-1},\ldots,j_0,\; i_{n-1},\ldots,i_0) \Leftrightarrow \\ p_{shl}(j'_{n-1},\ldots,j'_0,\; i_{n-1},\ldots,i_0) \end{array}\right) \quad , \\
\left(\begin{array}{c} p_y(j_{n-1},\ldots,j_0) \wedge \\ succ(j_{n-1},\ldots,j_0,\; j'_{n-1},\ldots,j'_0) \end{array}\right) \Rightarrow \\
\qquad \left(\begin{array}{c} p_{mul}(j_{n-1},\ldots,j_0,\; i_{n-1},\ldots,i_0) \Leftrightarrow \\ p_{add}(j'_{n-1},\ldots,j'_0,\; i_{n-1},\ldots,i_0) \end{array}\right)
\end{array}$$

The final product is given by $p_{mul}(0,\ldots,0,\; i_{n-1},\ldots,i_0)$.

bmark	bw	smt2	btor	Boolector	aig	cnf	Lingeling	epr	iProver
mulhs	$8^{\ddagger}$	947	1K	10.3s	3K	44K	9.0s	45K	1m 44s
	$16^{\ddagger}$	959	1K	TO	12K	205K	TO	55K	TO
	$64^{\ddagger}$	982	2K	TO	221K	4M	TO	78K	TO
lfsr_2_*bw*_16	$63^{\ddagger}$	6K	9K	0.2s	64K	258K	0.7s	56K	18.0s
	$127^{\ddagger}$	7K	9K	1.2s	139K	545K	1.3s	61K	1m 14s
	1023	7K	11K	5.1s	1M	5M	4.7s	74K	TO
	8191	7K	18K	2m 37s	11M	43M	3m 10s	89K	TO
add2n	2^5	452	455	0.0s	3K	25K	0.1s	12K	1m 21s
	2^6	456	671	0.1s	7K	53K	0.7s	13K	TO
	2^{12}	484	8K	3m 5s	549K	4M	1m 28s	21K	TO
addmul	2^7	149	99	0.2s	174K	3M	2.4s	8K	0.1s
	2^9	149	99	2.7s	3M	58M	3m 22s	11K	0.1s
	2^{11}	151	103	TO	48M	1G	TO	13K	0.1s

Table 6.1: Evaluation for the original SMT2 file

Polynomiality and Correctness. All above translation steps are polynomial in the input size since they are polynomial in the number of atoms and logarithmic in their bit-width. Formally showing correctness exceeds the scope of this chapter and is part of future work. We also investigated correctness empirically by exhaustively testing consistency of the solving results by Boolector and BV2epr+iProver, for each bit-vector operation, up to a certain bit-width.

6.4 Benchmarks and Experiments

Solving QF_BV formulas in general is NEXPTIME-complete [151]. However, certain families of QF_BV formulas are in NP, under certain restrictions on the bit-widths. We called this kind of families *bit-width bounded* [151]. Since solving EPR formulas is NEXPTIME-complete, our translation fits well to families which are *not* bit-width bounded. In [151], two examples of this kind were given: (a) `QF_BV/brummayerbiere3/mulhs`bw represents instances of computing the *high-order half of product* problem, parameterized by the bit-width of multiplicands (bw); (b) `QF_BV/bruttomesso/lfsr/lfsr`t_bw_n formalizes the behaviour of a *linear feedback shift register* [51]. Furthermore, we propose two new benchmark families that are *not bit-width bounded*: (a) `add2n`bw describes how bit-vectors of bit-width $2bw$ can be added by using two adders for bit-vectors of bit-width bw. (b) `addmul`bw checks, whether the sum of two bit-vectors of bit-width bw can differ from their product.

In order to demonstrate the exponential blow-up of bit-blasting, in contrast to our translation into EPR, we used the bit-blaster Synthebtor, part of the Boolector distribution, to generate AIGER files and DIMACS (CNF) files out of BTOR files. Table 6.1 summarizes these results, when *word-level rewriting* in Boolector is switched off. We give the file sizes (in bytes) in all formats and additionally provide the runtimes of Boolector (for SMT2), Lingeling (for CNF), and iProver

bmark	bw	smt2	btor	Boolector	aig	cnf	Lingeling	epr	iProver
mulhs	8‡	2K	804	9.8s	3K	42K	8.1s	63K	1m 48s
	16‡	2K	956	TO	11K	197K	TO	77K	TO
	64‡	2K	1K	TO	215K	4M	TO	110K	TO
lfsr_2_*bw*_16	63‡	126K	59K	0.5s	81K	254K	0.9s	156K	3.0s
	127‡	126K	59K	0.6s	174K	540K	1.4s	158K	9.5s
	1023	126K	60K	7.0s	1M	5M	5.1s	165K	9m 21s
	8191	126K	67K	46.1s	13M	43M	TO	173K	TO
add2n	2^5	1K	575	0.0s	4K	25K	0.1s	17K	23.6s
	2^6	1K	671	0.1s	9K	53K	0.7s	18K	5m 0s
	2^{12}	2K	9K	2m 42s	711K	4M	1m 16s	32K	TO
addmul	2^7	239	75	0.2s	174K	3M	2.5s	8K	0.1s
	2^9	239	75	2.8s	3M	58M	1m 40s	11K	0.1s
	2^{11}	241	79	TO	48M	1G	TO	13K	0.1s

Table 6.2: Evaluation for the simplified SMT2 file

(for EPR), using a timeout of 10 minutes. In order to test the effect of word-level rewriting, we added a module to Boolector which reads an SMT2 file, performs rewriting, and outputs the simplified SMT2 file. In Table 6.2, we give the results for the simplified SMT2 files.

6.5 Conclusion

We presented Bv2epr, a tool for polynomially translating QF_BV into EPR. The motivation for our tool lies in previous work [151], where we have shown QF_BV to be NExpTime-complete. Thus, bit-blasting QF_BV to SAT, as it is usually done in current SMT solvers, results in exponentially larger formulas in general. Previous translations from QF_BV into EPR also apply bit-blasting on certain operators and introduce exponentially many constants and constraints in the general case [143, 90]. In contrast to this, the Translator used in Bv2epr always produces EPR formulas of polynomial size. After discussing Bv2epr, we evaluated the size of the formulas produced by our tool and compared it to other commonly used formats. Our results show that the overhead in size is rather small when translating QF_BV into EPR, while all other formats often suffer from exponential blow-up as soon as the bit-widths in the input formula grow larger. However, our results also show that the runtime of iProver on the generated EPR formulas is usually worse compared to the runtime of Boolector on the original QF_BV formula or the one of Lingeling after bit-blasting has been applied. Nevertheless, the evaluation also shows that there exist benchmarks where iProver is faster. While it is probably still possible to improve EPR solvers on this kind of instances, formulas generated by Bv2epr can also help providing challenging benchmarks for current state-of-the-art solvers. The tool Bv2epr is available at [59].

‡Official SMT-LIB benchmarks.

Chapter 7

iDQ: Instantiation-Based DQBF Solving

Published. In Proceedings International Workshop on Pragmatics of SAT (POS 2014), Affiliated to SAT 2014, EPiC Series, volulme 27, pages 103–116, EasyChair 2014 [102].

Authors. Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith.

Abstract. Dependency Quantified Boolean Formulas (DQBF) are obtained by adding Henkin quantifiers to Boolean formulas and have seen growing interest in the last years. Since deciding DQBF is NEXPTIME-complete, efficient ways of solving it would have many practical applications. Still, there is only few work on solving this kind of formulas in practice. In this chapter, we present an instantiation-based technique to solve DQBF efficiently. Apart from providing a theoretical foundation, we also propose a concrete implementation of our algorithm. Finally, we give a detailed experimental analysis evaluating our prototype iDQ on several DQBF as well as QBF benchmarks.

7.1 Introduction

With steadily increasing success of decision procedures for propositional formulas (SAT) and Quantified Boolean Formulas (QBF), also interest in Dependency Quantified Boolean Formulas (DQBF) has grown during the last years.

DQBF has first been described in [187] and comprises the set of propositional formulas which are obtained by adding Henkin quantifiers [122] to Boolean logic. In contrast to QBF, the dependencies of a variable in DQBF are explicitly specified instead of being implicitly defined by the order of the quantifier prefix. This enables us to also use partial variable orders as part of a formula instead of only allowing total ones.

As a result, problem descriptions in DQBF can possibly be exponentially more

succinct. Whereas QBF is PSPACE-complete [185], DQBF was shown to be NEXPTIME-complete [188, 187]. Aside from DQBF, many practical problems are known to be NEXPTIME-complete. This includes, e.g., partial information non-cooperative games [188] or certain bit-vector logics [151, 234] used in the context of Satisfiability Modulo Theories (SMT). More recently, also applications in the area of equivalence for partial implementations [109, 110] and synthesis for fragments of linear temporal logic [64] have been discussed and translations to DQBF have been proposed.

There has been theoretical work on succinct formalizations using DQBF and certain subclasses, e.g., DQBF-Horn has been shown to be solvable in polynomial time [55]. However, apart from our previous work on adapting DPLL for DQBF [99] and a recent incomplete approach (only allowing refutation of unsatisfiable formulas) [94], there have not been many attempts to solve DQBF problems in practice nor actual implementations of decision procedures for DQBF. As already pointed out in [99], our previous approach did not end up being very efficient. Apart from this, formula expansion and transformations specific to QBF have been discussed in [9, 8], which stayed only on the theoretical side but can yield an expansion-based DQBF solver similar to those existing for QBF [24]. In [94], an expansion-based solver is also briefly mentioned. A (not publicly available) expansion-based solver was used in [110]. Furthermore, in [9, 8], it has been conjectured that QBF solvers based on Skolemization [22] could easily be adapted for DQBF. However, the current implementation of the described QBF solver SKIZZO [22] does not solely use Skolemization but also relies on an additional top-level DPLL approach for larger formulas. Adapting this kind of approach is not straightforward but requires special techniques as described in our previous work [99] and might have a similar negative impact on the performance of the resulting solver.

Effectively Propositional Logic (EPR) is another logic which is NEXPTIME-complete [161]. This implies that there exist polynomial reductions from DQBF to EPR and vice versa. Thus, it is possible to use EPR solvers, e.g., IPROVER [147] being the currently most successful one, to solve DQBF given some translation from DQBF to EPR. In [202], a translation from QBF to EPR is described which can be extended to DQBF easily. However, since EPR solvers in general have to reason with predicates and larger domains, solvers directly working on the propositional level should have an advantage if a DQBF formalization of a problem is more natural.

In the following, we present an instantiation-based approach to solving DQBF. Our approach is closely related to the so-called Inst-Gen calculus [148, 149], which can be considered as the state-of-the-art decision procedure for EPR [147]. While DQBF can be translated to EPR, we focus on applying the decision procedure directly on the given input logic. This results in a simpler framework and an algorithm which is easy to implement and adapt. At the same time, our approach can also be applied to QBF without further modifications. After defining some preliminaries in Section 7.2 and giving related work in Section 7.3, we provide the theoret-

ical foundation in Section 7.4 and point out parallel features used in EPR solving. We also propose a concrete implementation of our algorithm in Section 7.5, and provide detailed experiments, comparing our prototype iDQ with state-of-the-art solvers on several DQBF as well as QBF benchmarks in Section 7.6. It turns out that our implementation results in an efficient DQBF solver that works on practical benchmarks and is even able to compete with QBF solvers on some problems. We conclude and propose directions for future work in Section 7.7.

7.2 Preliminaries

Let V be a set of propositional variables. A *literal* l is a variable $x \in V$ or its negation $\overline{x}$. For a given literal l, we write $var(l)$ to reference the corresponding variable. A *clause* C is a disjunction of literals. A propositional formula ϕ is in *conjunctive normal form (CNF)*, if it is a conjunction of clauses. Any DQBF can always be expressed as

$$\psi \equiv Q.\phi \equiv \forall u_1, \ldots, u_m \exists e_1(u_{1,1}, \ldots, u_{1,k_1}), \ldots, e_n(u_{n,1}, \ldots, u_{n,k_n}).\phi$$

with Q being the quantifier prefix and ϕ being a propositional formula (matrix) in CNF over the variables $V := U \cup E$ and $U = \{u_1, \ldots, u_m\}$, $E = \{e_1, \ldots, e_n\}$, $u_{i,j} \in U$, $\forall\, i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}$. We refer to the elements of U and E as the *universal variables* and *existential variables* of ψ, respectively. In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given and not implicitly defined by the order of the prefix (in contrast to QBF).

Given an existential variable e_i, we use $dep(e_i) := \{u_{i,1}, \ldots, u_{i,k_i}\}$ to denote its dependencies. For universal variables u, we define $dep(u) := \varnothing$. We extend the notion of dependency to literals, defining $dep(l) := dep(var(l))$ for any literal l. Obviously, any QBF ψ_{qbf} can be translated to some ψ_{dqbf} in the specified form by moving all universal variables to the beginning and then setting $dep(e) = \{u \in U \mid u \textit{ is before } e \textit{ in the quantifier prefix of } \psi_{qbf}\}$ for all existential variables.

An *assignment* is a (partial) mapping $\alpha : V \to \{1, 0\}$ from the variables of a formula to truth values. To simplify the notation, we extend the definition of assignments to literals, clauses and formulas in the natural way. In the rest of this chapter, $\alpha(l)$, $\alpha(C)$, or $\alpha(F)$ will denote the truth value (under the assignment α) of a literal l, a clause C, or a formula F, respectively. An assignment α to a formula F is satisfying, if and only if $\alpha(F) = 1$.

A propositional formula ϕ in CNF is satisfiable, if and only if all clauses in ϕ are satisfied by at least one assignment α. We then call α a *model* of ϕ. In DQBF (as well as in QBF), a model cannot be expressed by a single assignment. Instead, we use Skolem functions to represent solutions of a formula. A Skolem function $f_e : \{1, 0\}^{|dep(e)|} \to \{1, 0\}$ describes the evaluation of an existential variable e under a given assignment to its dependencies. Let ϕ_{sk} denote the formula obtained from ϕ by replacing all existential variables e by their Skolem function

f_e. A DQBF $\psi = Q.\phi$ is satisfiable if and only if there exist Skolem functions $f_{e_1}, \ldots, f_{e_n}$, so that ϕ_{sk} is satisfied for all possible assignments to the universal variables of ψ.

Universal expansion is defined as the process of removing a universal variable u from a formula ψ considering both its values separately. This can be done by removing all existential variables e with $u \in dep(e)$ and introducing two new existential variables $e_{u=1}, e_{u=0}$ with $dep(e_{u=1}) = dep(e_{u=0}) = dep(e)\backslash\{u\}$. Additionally, the matrix ϕ is replaced by $\phi_{u=1} \wedge \phi_{u=0}$. With $\phi_{u=v}$, we describe the formula obtained from ϕ by replacing u by a constant $v \in \{1, 0\}$ and all occurrences of e with $u \in dep(e)$ by $e_{u=v}$. We can use universal expansion to reduce any DQBF ψ to an equisatisfiable propositional formula. If the resulting propositional formula is satisfiable, the Skolem functions of the original formula can be directly constructed from the assignments to the propositional variables by setting $f_e(v_1, \ldots, v_k) = e_{u_1=v_1,\ldots,u_k=v_k}$. In the following, we sometimes use the shorter notation ϕ_u and $\phi_{\overline{u}}$ instead of $\phi_{u=1}$ and $\phi_{u=0}$, respectively. We also extend this notation to clauses in the same way as we introduced it for formulas and refer to this as a *clause instance*, in the sense the Inst-Gen calculus [148, 149] uses *instantiation*, applied to the natural encoding of (D)QBF into first-order logic [202]. Furthermore, for a given clause instance $C_{l_1,\ldots,l_k}$, we define $ctx(C_{l_1,\ldots,l_k}) := \{l_i \mid i = 1, \ldots, k\}$. We call this the *context* of an instantiation.

The unique identifiers for the new existential variables introduced in this way make sure that the same existential variable is referred even if the individual clauses are considered separately. Also, the identifiers and the dependencies of all existential variables introduced during universal expansion are implicitly defined by the original quantifier prefix description. For example, for the DQBF

$$\forall u_1, u_2 \exists e_1(u_1), e_2(u_1, u_2), e_3(u_1, u_2) \,.\, (u_1 \vee e_1) \wedge (\overline{u}_2 \vee e_2) \wedge (\overline{u}_1 \vee u_2 \vee \overline{e}_3) \quad (7.1)$$

we can now write equations of clause instances such as:

$$\begin{aligned} &= (e_1)_{\overline{u}_1} \wedge (\overline{u}_2 \vee e_2) \wedge (\overline{u}_1 \vee u_2 \vee \overline{e}_3) &&= (e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (\overline{u}_1 \vee u_2 \vee \overline{e}_3) \\ &= (e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (u_2 \vee \overline{e}_3)_{u_1} &&= (e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (\overline{e}_3)_{u_1\overline{u}_2} \end{aligned}$$

The last line is a succinct representation of the full universal expansion of the original formula and minimal in the sense of our algorithm. We refer to each individual step as a *local universal expansion*. Note that we immediately dropped all trivially satisfied clauses (due to $u_i = 1$) in each step. Also, all intermediate steps can be performed in arbitrary order, e.g., although we started with expanding the first clause regarding u_1, it is not necessary to expand all other clauses on u_1 before expanding some clauses on u_2. Obviously, we could continue applying local universal expansion and obtain equivalent formulas of growing size:

$$(e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (\overline{e}_3)_{u_1\overline{u}_2} \;= (e_1)_{\overline{u}_1} \wedge (e_2)_{\overline{u}_1 u_2} \wedge (e_2)_{u_1 u_2} \wedge (\overline{e}_3)_{u_1\overline{u}_2}$$

The last expression is maximal and of the same size as the full universal expansion of ψ. There is no point in further expanding the first clause instance, because

$u_2 \notin dep(e_1)$ implies that $(e_1)_{\overline{u}_1} = (e_1)_{\overline{u}_1\overline{u}_2} = (e_1)_{\overline{u}_1 u_2}$. Obviously, if a clause instance $C_{l_1,\ldots,l_k}$ is part of a formula, we can always add a more specific instance $C_{l_1,\ldots,l_k,l_{k+1},\ldots,l_{k'}}$ without affecting satisfiability. The more specific instance is actually subsumed by the original one, i.e. the full local universal expansion of the new instance is a subset of the full local universal expansion of the less specific one. This fact is crucial for the algorithm presented in Section 7.4.

EPR, known as the Bernays-Schönfinkel class, is a NEXPTIME-complete fragment of first-order logic [161]. It consists of the set of first-order formulas that, written in prenex form, contain (1) no function symbol of arity greater than 0, and (2) no existential quantifier within the scope of a universal quantifier. After Skolemization, existential variables turn into constants (i.e., function symbols of arity 0). Consequently, an EPR atom can be defined as an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and each t_i is either a (universal) variable or a constant.

In [202], a translation from QBF to EPR is proposed. The approach consists of three steps and can be easily adapted to DQBF: (1) replace each existential variable e with its Skolem function f_e (which is in fact a predicate due to the Boolean domain), (2) replace each universal variable u with $p(u)$ where p is a fixed predicate, and (3) add the constraints $p(1)$ and $\neg p(0)$ to the formula. For example, for the DQBF in Equation (7.1) the resulting EPR formula is

$$\forall u_1, u_2 \,.\, \big(p(u_1) \vee f_{e_1}(u_1)\big) \wedge \big(\neg p(u_2) \vee f_{e_2}(u_1, u_2)\big) \wedge \big(\neg p(u_1) \vee p(u_2) \vee \neg f_{e_3}(u_1, u_2)\big) \wedge p(1) \wedge \neg p(0)$$

7.3 Related Work

The concepts of instantiation and expansion that we defined in Section 7.2 are similar to the notation used in [22], describing the solver SKIZZO, which in particular shares similarities in the use of clause instances (cf. *symbolic representation* in [22]). But apart from slightly different notation, there are three fundamental differences in the underlying algorithms: First, our method aims at solving DQBF while SKIZZO, as described, targets QBF solving. Second, SKIZZO uses a top-level QDPLL step, which cannot be applied to DQBF without introducing additional concepts as presented in our previous work [99]. Finally, the most important difference is that SKIZZO performs a full Skolemization after preprocessing, while our solver uses local extension to iteratively generate a (potentially exponentially) more succinct formula which is sufficient to prove (un)satisfiability of the original input, as described in Section 7.4.

Another similar notation and related work is proposed in [134, 135, 136]. Their solver RAREQS [135] creates propositional abstractions and uses a CEGAR approach [68] for refinement. As we will discuss in Section 7.4, this is also what our solver does. However, the way abstractions are generated and refined is different. One main difference can be found in the expansion of universal variables.

```
1 procedure CEGAR(F)
2   F' = initInstantiation(F);
3   while(true)
4     F'' = propositionalAbstraction(F');
5     (state, assignment) = checkState(F'');
6     if (state = UNSAT) return UNSAT;
7     if (isValid(assignment, F, F')) return SAT;
8     F' = refineInstantiation(assignment, F, F');
```

Figure 7.1: Pseudo-code of a CEGAR loop as used in the Inst-Gen procedure [147, 148, 149].

In contrast to SKIZZO, both, RAREQS as well as our solver, allow partial expansion in the sense that only $\phi_{u=1}$ or $\phi_{u=0}$ might be considered for some formula ϕ containing u. Nevertheless, even the restricted expansion of universal variables in [134, 135, 136] always applies to *all* clauses of a formula, whereas our approach uses the previously described concept of *local universal expansion*, which allows to expand clauses individually. Furthermore, RAREQS is a QBF solver and cannot tackle DQBF. Due to the usage of recursive calls depending on the order of the quantifier prefix, an extension to DQBF does not seem to be straightforward.

Another solver that relies on abstraction refinement, is given in [234]. While they target quantified bit-vector formulas with uninterpreted functions, QBF and DQBF of course can be seen as a special case. To generate abstractions, they apply Skolemization and use templates for functions. The effectiveness of their approach heavily relies on the right choice of templates, which can be difficult for QBF and DQBF. Finally, another algorithm that has a similar structure can be found in [190]. Again, their solver actually targets more general SMT formulas, but could theoretically also be used for QBF. Since their approach expects an ordered quantifier prefix, it cannot be directly applied to DQBF.

7.4 IDQ architecture

In this section, we present the IDQ architecture. It is based on the more general Inst-Gen calculus [148, 149] for EPR as used in IPROVER [147], but reduced to the more specific case of DQBF. Instead of dealing with predicates, we use the notion of clause instances as introduced in Section 7.2. The Inst-Gen architecture is based on the CEGAR paradigm [68] and the pseudo-code is given in Figure 7.1.

For EPR, usually no specific initial instantiation is used, i.e., the formula is completely uninstantiated. A propositional abstraction is then created by grounding the current formula and can be solved by a SAT solver. If the SAT solver returns *unsat*, the original formula is *unsat* too, since the ground formula is an overapproximation. On the other hand, if the SAT solver returns *sat*, the result-

ing assignment has to be checked for consistency with the EPR formula. In each propositional clause, we select a satisfying literal, determined by a fixed *selection function*. If there is no pair of oppositely signed, selected literals, such that the corresponding EPR literals can be unified, the solution is also valid for the original EPR formula. If there are such pairs of literals, then we try to apply the following *inference step* to each corresponding EPR clause: apply the most general unifier (MGU) to the clause and add the result as a new clause. By checking if the new clause is already part of the formula with respect to some *redundancy* concept, it is also possible that no new clause is added. The formula is then called *saturated* and the current assignment is also valid for the input formula. Otherwise, the calculus starts the next iteration.

Using the approach described in [202], any DQBF can be translated to EPR. All universal variables u are embedded into EPR by introducing a predicate p and replacing each occurrence of u by $p(u)$. Additionally, the constraints $p(1)$ and $\neg p(0)$ are added to the formula. Obviously, this implies that $p(u)$ and $\neg p(u)$ can never end up being the only satisfying literal of a clause. If this was the case, unification with $p(1)$ and $\neg p(0)$ would be possible, respectively. As a result, the corresponding instance would be added to the formula and, from that point on, in every loop iteration the SAT solver would immediately set the instantiated literal to 0 by unit propagation.

Knowing that we deal with DQBF, this will always be the case. Therefore, we can directly simplify the formula in the beginning by starting with a more specific initial instantiation. For each clause, we only care about those assignments to the universal variables which do not trivially satisfy the clause. In our notation, this initial instantiation is equal to the minimal instantiation created by local universal expansion as described in Section 7.2. Consider the following example:

$$\psi \;=\; \forall u_1, u_2 \exists e_1(u_1, u_2), e_2(u_2) \,.\, (u_1 \vee e_1) \wedge (u_1 \vee \overline{e}_1) \wedge (\overline{u}_1 \vee u_2 \vee e_1 \vee e_2)$$

We now create the initial set of clause instances, using the unique minimal instantiation that removes all universal variables from the clauses:

$$(e_1)_{\overline{u}_1} \wedge (\overline{e}_1)_{\overline{u}_1} \wedge (e_1 \vee e_2)_{u_1 \overline{u}_2}$$

We then create a propositional abstraction of the current clause instance set, by assuming that all existential variables that do not occur in the same instantiation context can be different. This means, for $\mathfrak{P}$ denoting the power-set, we use a function $m : E \times \mathfrak{P}(\{l \mid var(l) \in U\}) \to V'$ for some new set of propositional variables V', and map each literal e in a clause instance C to a propositional variable $m(e, ctx(C))$. We restrict m as follows:

$$\begin{aligned} m(e_1, ctx(C_1)) &= m(e_2, ctx(C_2)) \\ &\qquad\qquad \text{if and only if} \\ e_1 &= e_2 \\ \big\{l_1 \in ctx(C_1) \mid var(l_1) \in dep(e_1)\big\} &= \big\{l_2 \in ctx(C_2) \mid var(l_2) \in dep(e_2)\big\} \end{aligned}$$

Obviously, the propositional formula generated by this mapping is an overapproximation of the current set of clause instances. It will often be the case that there is some kind of dependency between different variables.

In our example, we get the following propositional formula:

$$(x_1) \wedge (\overline{x}_1) \wedge (x_2 \vee x_3)$$

Satisfiability can easily be checked by using any off-the-shelf SAT solver. In this specific example, the propositional overapproximation is unsatisfiable. This implies that the original formula is also unsatisfiable.

If, on the other hand, the propositional formula was satisfiable, we would need additional reasoning. For this, consider a second example:

$$\psi \;=\; \forall u_1, u_2 \exists e_1(u_1, u_2), e_2(u_2) \,.\, (u_1 \vee e_1) \wedge (\overline{u}_2 \vee \overline{e}_1 \vee e_2)$$

Again, we create the initial set of clause instances using the unique minimal instantiation that removes all universal variables from the clauses:

$$(e_1)_{\overline{u}_1} \wedge (\overline{e}_1 \vee e_2)_{u_2}$$

The propositional overapproximation now looks as follows:

$$(x_1) \wedge (\overline{x}_2 \vee x_3)$$

Note that the same existential variable e_1 is mapped to two different variables x_1, x_2 because it appears in different contexts. The SAT solver would now tell us that this abstraction is satisfiable and return a satisfying assignment α, e.g., $\alpha = \{x_1 \to 1, x_2 \to 0, x_3 \to 0\}$.

We now check, whether α is a valid satisfying assignment for the current set of clause instances. This is the case if and only if no pair of oppositely signed, selected (satisfying) literals corresponds to the same existential variable in overlapping contexts. For EPR, this is exactly what happens in the Inst-Gen calculus when there is a check on whether the corresponding literals can be unified [147, 148, 149]. In the case that a satisfying assignment is valid for the current set of clause instances, we know that the original DQBF is satisfiable. If, however, the assignment is not valid, we refine the instantiation on the clauses that contain the conflicting literals by adding new instances. Those instances are actually subsumed by the original ones but lead to a different propositional abstraction by the definition of m. In the next step, the propositional abstraction will automatically rule out this conflicting assignment.

In our latest example, α is indeed not a valid assignment for the current set of clause instances: x_1 and x_2 correspond to e_1, appear in overlapping contexts and, therefore, the propositional variables cannot be assumed to be independent of each other. We therefore apply the inference step of *merging* the two contexts and adding new clause instances. Now, the resulting formula looks as follows:

$$(e_1)_{\overline{u}_1} \wedge (e_1)_{\overline{u}_1 u_2} \wedge (\overline{e}_1 \vee e_2)_{u_2} \wedge (\overline{e}_1 \vee e_2)_{\overline{u}_1 u_2}$$

The propositional abstraction is given by:

$$(x_1) \wedge (x_2) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee x_4)$$

Note that e_2 is mapped to the same variable x_4 in both clause instances although it appears in a different instantiation context. This is due to $u_1 \notin dep(e_2)$, which implies that $(e_2)_{u_2} = (e_2)_{\overline{u}_1 u_2}$. Again, this propositional formula is satisfiable and the SAT solver could return a satisfying assignment $\alpha = \{x_1 \to 1, x_2 \to 1, x_3 \to 0, x_4 \to 1\}$. However, this time we can pick a literal in each clause so that no implicit dependencies are violated. Therefore, the algorithm terminates and the original formula is known to be satisfiable.

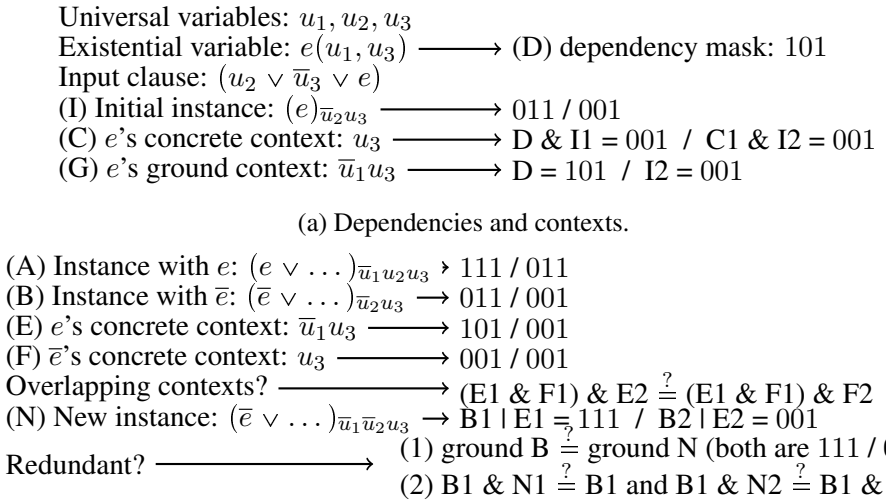
Furthermore, also note that in our particular case, we could have directly applied local universal expansion to our instances instead of adding a single more specific one, e.g., yielding $(e_1)_{\overline{u}_1 \overline{u}_2} \wedge (e_1)_{\overline{u}_1 u_2}$ instead of $(e_1)_{\overline{u}_1} \wedge (e_1)_{\overline{u}_1 u_2}$. However, this can only be done without growth in formula size, if there is exactly one additional literal in the new context of the instance, which we would have added otherwise. Nevertheless, this is a possible DQBF-specific extension, which is part of future work, and sometimes might reduce the number of loop iterations.

7.5 Implementation

In this section, we describe how we actually implemented the proposed algorithm and point out where we can profit from DQBF-specific restrictions. For our solver, we use input files in a format that is an extension of the `QDIMACS` format and which we call `DQDIMACS`. The only difference to `QDIMACS` is the fact that we additionally allow partially ordered dependencies by using expressions of the form `d <int32> [<int32> ... <int32>] 0` in the quantifier prefix description. This defines a new existential variable given by the first ID as integer which (optionally) depends on a list of previously defined universal variables. All other quantifier definitions using `a` and `e` are still interpreted in the same way as it is done in the `QDIMACS` format and existential variables defined by using `e` are assumed to depend on all previously defined universal variables as usual. In this way, `DQDIMACS` is easy to parse and a real extension of `QDIMACS`. `DQDIMACS` is also the input format which we use in all our experiments in Section 7.6.

After parsing the input, the data structures we use are similar to those of common SAT solvers. The matrix of the original formula is saved as a list of clauses and a clause is saved as a list of literals represented by integers. Additionally, the quantifier prefix is saved as a list of variables and each variable has an ID, a quantifier type and, if it is an existential variable, a bit-vector, called the *dependency mask*, representing the universal variables that it depends on.

We store a *list of instances* with each clause. An instance is defined by two bit-vectors, called the *context mask* and the *value mask*, representing the universal variables that are assigned by the context and the values they are assigned to, respectively. For example, see instance (I) in Figure 7.2, where the first mask is



(a) Dependencies and contexts.

(b) Inference step and redundancy check.

Figure 7.2: Examples of using bit-vector representation for calculations in IDQ.

the context mask and the second one is the value mask. For the propositional abstraction, a *propositional clause* is also stored with each instance. All propositional clauses are incrementally added to the underlying SAT solver, PICOSAT [26].

Initial Instantiation. Creating the initial instantiation is straightforward. When parsing the clauses of the formula, universal literals l are not added to the literal list of the current clause, but instead the corresponding bits in the context mask and the value mask are set accordingly to represent that $\overline{l}$ is part of the context of the current instance; see (I) in Figure 7.2.

Propositional Abstraction. Each occurrence of a existential variable is mapped to a corresponding propositional variable. This can be done efficiently by using the bit-vectors that are saved with each existential variable and each clause instance. Given an existential variable e that occurs in an instance c, we calculate e's *concrete context* that is to show which part of c's context is relevant for e. The concrete context can be calculated by applying *bitwise and* to e's dependency mask and c's context mask, and another *bitwise and* with c's value mask. This is illustrated in Figure 7.2 as the context mask (C1) being calculated from (D) and (I1), and the value mask (C2) from (C1) and (I2).

We map the variable ID and the concrete context to a unique propositional variable. Accordingly, if two variable occurrences have the same ID (i.e., they represent the same existential variable) and their concrete contexts are equal, they are mapped to the same propositional variable. In order to check whether we already introduced the corresponding propositional variable in a previous step, we keep a hash table with all previously introduced propositional variables.

Grounding. As the Inst-Gen calculus [148, 149] suggests, before mapping an existential variable e and its concrete context to a propositional variable, iDQ generates the grounding of this context. Grounding is basically about assigning a concrete truth value, w.l.o.g., 0, to all the universal variables which e depends on and which are not already assigned by the context. This can easily be done by setting the context mask to e's dependency mask and leaving the value mask as it is, assuming that all bits in our bit-vectors are initialized to 0. Figure 7.2 shows an example, as setting (G1) to (D) and (G2) to (I2).

Active and Passive Instances. Similar to iProver's architecture, clause instances are separated into two sets, called *active* and *passive*. Active instances are the ones among which all possible inference steps have been performed, modulo literal selection. Passive instances are the ones which are waiting to participate in inferences. In iDQ, passive instances are stored in a *priority queue* ordered by a given heuristic. In each solving iteration, iDQ dequeues a given number of passive instances with the highest priority, and sets them active one by one, which involves trying to apply an inference step with each active instance.

In the current implementation of iDQ, an active instance does not move back to the passive instance set whenever its literal selection changes, as opposed to iProver. We rather apply inference steps to it with each active instance, on the newly selected literal.

An inference step on two selected literals can easily be implemented, as illustrated in Figure 7.2. First, to check whether the concrete contexts of the literals are overlapping, we apply *bitwise and*. Second, to calculate the context and value masks for a new instance, we apply *bitwise or* to the masks representing the original instance and the ones representing the literal from the other instance.

Heuristics. Two choices depend on some heuristics: (1) how to order the priority queue of passive instances, and (2) how to select a satisfying literal in an active instance. We have been experimenting with two types of heuristics, using different criteria for both choices.

One of the heuristics is inspired by iProver's default heuristic, based on the lexicographical combination of orders defined on given numerical/Boolean parameters. Similar to iProver's notation [149], we use the following combinations: (1) `[-num_dep;+age;-num_symb]` for the priority queue of instances, and (2) `[+sign;+ground;-num_dep;-num_symb]` for literal selection. This means priority is given to instances with fewer unassigned dependencies, then to instances generated at earlier iterations, and finally to instances with fewer symbols (0 or 1) assigned to dependencies. The heuristic for literal selection can be interpreted in a similar way, where positive and then ground literals are prioritized the most.

The other heuristic is inspired by SAT solving. It is based on the VSIDS scores [176] of propositional variables used in the propositional abstraction. iDQ

counts the occurrences of those variables in the propositional clauses generated so far, and then, after each 50 iterations, all the scores are divided by 2. Based on the VSIDS scores, (1) priority is given to the passive instance with the highest average score of its literals, and (2) the literal with the highest score is selected.

Redundancy Check. Redundancy elimination is crucial for the applicability of any calculus, in order to avoid infinite runs and to obtain a smaller knowledge base. Due to the finite domain property, it is easy to obtain a sufficient, but not practical, redundancy check for both EPR and DQBF, by simply checking the equality of clause instances, i.e., of context/value masks in iDQ. However, a practical redundancy check might be more complicated, e.g., iProver employs dismatching constraints [149]. With iDQ, a practical check can be obtained more easily. iDQ decides if a new instance c would not give any new information to the active instance set, meaning that the propositional abstraction would stay the same and all inference steps with c would also result in redundant instances. We consider c redundant if there exists an active instance d of the *same* clause such that (1) the propositional abstractions of c and d are the same, and (2) d subsumes c. Both checks can be done by bit-vector operations, as illustrated in Figure 7.2. Importantly, (2) requires to check if c's context is a superset of d's contexts.

7.6 Experimental Results

In this section, we report experiments[1] with our solver. The source code, benchmarks, and log files are available at `http://fmv.jku.at/idq`. We tested iDQ with two types of heuristics as proposed in Section 7.5. iDQ and $\text{iDQ}_{\mathsf{vsids}}$ refer to the versions that employ the default heuristic and the VSIDS-based heuristic, respectively. Lacking in publicly available, general-purpose DQBF solvers (the solver DQBF2QBF in [94] can reason only with *unsat* formulas), we decided to also compare iDQ against iProver (v0.8.1).

We also tested iDQ on QBF benchmarks, by exploiting the fact that QBF is a real fragment of DQBF. By doing so, we could compare iDQ not only against iProver, but also against genuine QBF solvers, like the QDPLL-based DepQBF [165] (v3.0), the CEGAR-based RAReQS [135] (v1.1), the expansion-based Nenofex [164] (v1.0), and the Skolemization-based sKizzo [22] (v0.8.2). For the sake of fair comparison, we did not run any preprocessor.

DQBF Benchmarks. We used the only publicly available DQBF benchmarks by Finkbeiner and Tentrup [94]. All of them encode partial equivalence checking (PEC) problems, i.e., circuits containing some "black boxes" compared against full circuits. This benchmark set includes the benchmarks of the 3-bit arithmetic circuits *adder* and the 16-bit arbiter implementations *bitcell* and *lookahead* from [75],

[1] Setup: Vienna Scientific Cluster (VSC-2), AMD Opteron Magny Cours 6132HE CPUs, 2.2 GHz cores, 900 seconds time limit, 3800 MB memory limit.

and also the circuit family *pec_xor* from [110] about comparing the XOR of input bits against a random Boolean function. We converted those benchmarks to DQDIMACS format, and then ran IDQ on them. For IPROVER, we further converted the DQDIMACS instances to EPR (TPTP CNF format) by using the translation from [202], which can be easily adapted to DQBF.

Table 7.1 shows the results: the number of solved instances (#), the number of timeouts (TO), and the average runtime. The number at the end of benchmark names shows the number of black boxes in circuits. In most of the cases, IDQ outperforms IPROVER. $\text{IDQ}_{\text{vsids}}$ performs even better than IDQ on the *bitcell* benchmarks but worse on the *lookahead* and *adder* benchmarks. The gap between the performance of IDQ and IPROVER is significant. On *unsat* instances, DQBF2QBF generally is the fastest solver. However, the performance of IDQ sometimes comes quite close, whereas DQBF2QBF cannot solve *sat* instances at all. Also note that the benchmarks are biased in the way that most sets contain mainly *unsat* instances. Finally, we think that one reason for the better performance of DQBF2QBF on *unsat* instances is the better encoding of the original benchmarks and the overhead introduced by CNF translation. Preliminary results on simple preprocessing techniques show that this can lift the performance of IDQ to come even closer to the one of DQBF2QBF.

	#(sat/uns)	*TO*	*time*	*#(sat/uns)*	*TO*	*time*	*#(sat/uns)*	*TO*	*time*
	bitcell_16_2			bitcell_16_4			bitcell_16_6		
DQBF2QBF	98 (0/98)	2	18.6	98 (0/98)	2	18.8	97 (0/97)	3	27.8
IDQ	88 (2/86)	12	128.1	52 (0/52)	48	488.9	22 (0/22)	78	735.9
$\text{IDQ}_{\text{vsids}}$	97 (2/95)	3	39.2	75 (0/75)	25	255.9	36 (0/36)	64	592.0
IPROVER	82 (0/82)	18	248.6	34 (0/34)	66	684.5	7 (0/7)	93	851.7
	lookahead_16_2			lookahead_16_4			lookahead_16_6		
DQBF2QBF	97 (0/97)	3	27.7	97 (0/97)	3	27.7	96 (0/96)	4	36.6
IDQ	98 (3/95)	2	30.4	88 (0/88)	12	118.9	69 (0/69)	31	342.4
$\text{IDQ}_{\text{vsids}}$	93 (2/91)	7	68.1	62 (0/62)	38	383.0	20 (0/20)	80	729.9
IPROVER	67 (0/67)	33	351.8	32 (0/32)	68	656.3	6 (0/6)	94	862.9
	adder_3_2			adder_3_4			adder_3_6		
DQBF2QBF	94 (0/94)	6	54.8	89 (0/89)	11	99.8	74 (0/74)	26	234.6
IDQ	82 (1/81)	18	246.8	58 (0/58)	42	440.2	11 (0/11)	89	841.4
$\text{IDQ}_{\text{vsids}}$	43 (0/43)	57	546.3	21 (0/21)	79	734.0	6 (0/6)	94	863.9
IPROVER	86 (1/85)	14	221.6	54 (0/54)	46	538.2	5 (0/5)	95	876.9
	pec_xor2			pec_xor3			pec_xor4		
DQBF2QBF	49 (0/49)	51	459.4	77 (0/77)	23	207.5	99 (0/99)	1	10.6
IDQ	100 (51/49)		.5	100 (23/77)		.7	100 (1/99)		3.3
$\text{IDQ}_{\text{vsids}}$	100 (51/49)		.5	100 (23/77)		.6	100 (1/99)		2.2
IPROVER	100 (51/49)		.5	100 (23/77)		.9	100 (1/99)		2.8

Table 7.1: Results for *DQBF PEC* benchmarks

QBF Benchmarks. We used *QBF Gallery 2013* benchmarks, from which we selected instances with a size that does not exceed 2 megabytes. In some cases, we randomly selected instances from the resulting sets. Table 7.2 shows the results, including the number of memory outs (MO) and the number of crashes (CR). Between parentheses after each benchmark name, the number of instances is shown. As expected, genuine QBF solvers outperform IDQ and IPROVER on most benchmarks, although SKIZZO and NENOFEX terminate with memory out quite frequently. On some instances, IPROVER and NENOFEX crash. IDQ performs particularly well on the benchmarks *conformant-planning* and *planning-CTE*, and reasonably well on *sauer-reimer*. In general, the VSIDS-heuristic seems to be the slightly better choice.

	#(sat/uns)	*TO/MO*	*time*	*CR*	*#(sat/uns)*	*TO/MO*	*time*	*CR*
	conformant-planning (100)				planning-CTE (57)			
DEPQBF	89 (19/70)	11/0	130.7		42 (26/16)	15/0	297.0	
RAREQS	94 (17/77)	4/2	49.1		57 (35/22)		1.4	
NENOFEX	95 (19/76)		19.7	5	57 (35/22)		3.8	
SKIZZO	51 (11/40)	34/15	380.9		57 (35/22)		1.8	
IDQ	95 (14/81)	5/0	81.9		57 (35/22)		6.2	
$\text{IDQ}_{\text{vsids}}$	95 (14/81)	5/0	80.2		57 (35/22)		6.5	
IPROVER	91 (14/77)	9/0	90.9		57 (35/22)		4.6	
	qbf-hardness (162)				reduction-finding (100)			
DEPQBF	59 (12/47)	103/0	586.1		65 (34/31)	35/0	348.4	
RAREQS	63 (12/51)	99/0	572.0		81 (41/40)	19/0	201.2	
NENOFEX	26 (12/14)	0/136	487.9		35 (19/16)	0/65	425.0	
SKIZZO	48 (12/36)	79/35	526.8		34 (19/15)	46/20	468.2	
IDQ	44 (12/32)	118/0	665.0		30 (16/14)	70/0	635.4	
$\text{IDQ}_{\text{vsids}}$	42 (12/30)	120/0	666.8		29 (15/14)	64/7	598.2	
IPROVER	26 (12/14)	135/0	762.8	1	31 (18/13)	48/6	554.9	15
	sauer-reimer (100)				eval2012r2 (264)			
DEPQBF	50 (35/15)	50/0	457.3		90 (33/57)	174/0	610.8	
RAREQS	33 (20/13)	0/67	248.2		67 (23/44)	162/35	626.7	
NENOFEX	18 (9/9)	0/82	564.7		54 (28/26)	7/200	519.0	3
SKIZZO	18 (9/9)	43/39	614.8		89 (39/50)	128/47	521.7	
IDQ	20 (8/12)	80/0	724.7		45 (15/30)	217/2	757.8	
$\text{IDQ}_{\text{vsids}}$	27 (17/10)	73/0	658.7		51 (18/33)	178/35	682.2	
IPROVER	19 (10/9)	76/5	725.8		54 (18/36)	178/30	672.7	2

Table 7.2: Results for *QBF Gallery 2013* benchmarks

7.7 Conclusion

In this chapter, we presented an instantiation-based algorithm for solving DQBF, resulting in a complete and at the same time practical DQBF solver.

On the theoretic side, we showed how successful techniques in EPR solving

can be lifted to the more specific DQBF case. We brought together related work on Skolemization with the Inst-Gen calculus. On the other hand, we extended work on iProver by giving a simpler framework. While our implementation is still a prototype, our experiments confirmed that the simpler structure of DQBF compared to the more general EPR, as well as the smaller formula size compared to the full expansion, can have a positive impact on solver performance.

So far, our optimization compared to iProver was mainly on the implementation side using more efficient data structures and operations tailored to the Boolean domain. Apart from the possibility of applying local universal expansion as a special case of instantiation, looking into more potential DQBF-specific benefits, especially on the heuristic level, is part of future work. Specialized preprocessing techniques, e.g., related to those applied in sKizzo [22] or for general QBF solvers [32], as well as removing dependencies of existential variables by analyzing the propositional matrix [165], might also be a further interesting step into the direction of even more efficient DQBF solving.

Another potential benefit of our solver could be related to providing certificates. Certificate construction in QBF has seen increasing interest in recent research [56, 89, 123, 134, 136, 182, 206, 207]. While providing certificates is not implemented in our prototype yet, our architecture can easily be extended by this feature. Obviously, Skolem functions for satisfying formulas can directly be constructed out of a solution as discussed in Section 7.2. However, the more interesting contribution might be for unsatisfiable formulas. As unsatisfiability of a formula is proven by a SAT solver in combination with universal expansion, we can directly use the generated resolution proof for refuting the initial DQBF input, similar to the approach described in [136]. Due to the iterative refinement in the solving process, certificates (for unsatisfiability as well as satisfiability) might be rather small. Further shrinking could be possible by looking for unsatisfiable cores.

Finally, we were able to outperform even more specific QBF solvers on some benchmarks. As an additional side-effect, we therefore hope to get new insights into QBF solving and maybe even QBF solvers might profit from our techniques.

Chapter 8

Efficiently Solving Bit-Vector Problems Using Model Checkers

Published. In Proceedings 11th International Workshop on Satisfiability Modulo Theories (SMT 2013), pages 6–15, Affiliated to SAT 2013, Helsinki, Finland, 2013, Informal Proceedings [100].

Authors. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere.

Abstract. Bit-precise reasoning is essential in many applications of Satisfiability Modulo Theories (SMT). Most approaches for solving quantifier-free fixed-size bit-vector logics (QF_BV) rely on bit-blasting. In previous work, we have shown that bit-blasting is not polynomial in general [151], and later proposed QF_BV$_{\ll 1}$, a class of bit-vector problems that is PSPACE-complete [101]. In this chapter, we give examples of how to create (polynomial) SMV specifications out of QF_BV$_{\ll 1}$ formulas. We then use various model checkers to solve those problems and give detailed experimental results. Our results show that BDD-based model checkers outperform current SMT solvers by several orders of magnitude on our benchmarks. Unrolling and using SAT-based model checking turns out to be the same as bit-blasting and gives worse results. In addition to this, our approach allows us to easily generate new challenging benchmarks for SMT solvers as well as for model checkers.

8.1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for fixed-sized bit-vector logics are Boolector [47], MathSAT [50], STP [104], Z3 [81], and Yices [87]. All these solvers rely on *bit-blasting* in order to translate bit-vector formulas into propositional logic (SAT). The result is then checked by a SAT

solver.

In practice, e.g., in the SMT-LIB [18], the BTOR [48], and the Z3 format, the bit-widths in bit-vector formulas are encoded as binary, decimal, or hexadecimal numbers, i.e., a *logarithmic encoding* is used. In [151], we proved that the encoding of bit-widths affects the complexity of the decision problem of bit-vector logics. In particular, logarithmic encoding makes the quantifier-free fragment QF_BV NEXPTIME-complete.[1] Thus, bit-blasting is *not polynomial* in general. Consider the following example (in SMT2 syntax):

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(declare-fun z () (_ BitVec 1000000))
(assert (= z (bvadd x y)))
(assert (= z (bvshl x (_ bv1 1000000))))
(assert (distinct x y))
```

This formula verifies that, for an arbitrary bit-vector x of bit-width one million, there exists no bit-vector $y \neq x$ with $x + y = x \ll 1$. Written to a file, this formula can be encoded with 225 bytes. Using the SMT solver Boolector (even with all rewritings switched on), bit-blasting produces a circuit of size 129 MB encoded in the actually rather compact AIGER format. Tseitin transformation results in a CNF in DIMACS format of size 843 MB.

In related work [150], we tried to avoid this growth in size by giving a translation from QF_BV to EPR and then using iProver to solve the problem. In most cases, this approach turned out to perform worse than Boolector on the original instance. Since QF_BV is NEXPTIME-complete, it is not clear if it is possible to solve the general case more efficiently. However, the given example only uses *addition*, *shift by one* and *equality*. In [101], we showed that this kind of formulas can be expressed by QF_BV$_{\ll 1}$, a subset of QF_BV which turned out to be PSPACE-complete. In order to prove this, we gave a polynomial translation from QF_BV$_{\ll 1}$ to sequential circuits, similar to the one for linear arithmetic on *non-fixed-size* bit-vectors proposed in [208, 209].

In this chapter, we show how model checkers can be used to solve *fixed-size* bit-vector problems of this class. In contrast to [101], which provided the theoretical background, we now focus on experimental evaluation and analyze the potential benefits for efficiently solving bit-vector formulas. First, in Section 8.2, we provide a short overview of our translation as described in [101] and give some examples to show how we used this concept to convert SMT2 files to SMV. In Section 8.3, we then describe some benchmarks that we generated to evaluate the performance of various model checkers compared to state-of-the-art SMT solvers with support for fixed-sized bit-vector logics. On most of our benchmarks, BDD-based model

[1] In [151], we introduced the notation QF_BV1 and QF_BV2 for QF_BV using a *unary* and a *logarithmic* (w.l.o.g., *binary*) encoding, respectively. In this chapter, QF_BV will always refer to the logarithmic/binary case.

checkers turn out to be faster by several orders of magnitude. We provide experimental data and discuss the results in detail. Finally, in Section 8.4, we conclude the chapter and discuss further topics for future work.

8.2 QF_BV$_{\ll 1}$ to SMV

In [208, 209], the authors gave a polynomial translation for linear arithmetic on *non-fixed-size* bit-vectors (QFPAbit) into sequential circuits. In contrast to [208, 209], we focus on *fixed-size* bit-vectors but share the goal of avoiding the exponential explosion due to explicit state representation as for example used in MONA [144]. We adapted this translation in [101] to deal with fixed-size bit-vectors and extended it by various other operators like *shift by one* and *indexing*.

Given a bit-vector formula $\Phi \in$ QF_BV$_{\ll 1}$ without nested equalities. Let n be a bit-width, $x^{[n]}, y^{[n]}$ denote bit-vector variables, $c^{[n]}$ a bit-vector constant, and ${t_1}^{[n]}, {t_2}^{[n]}$ bit-vector terms only containing bit-vector variables and bitwise operations. Following [208, 209], we assume, w.l.o.g., that Φ only consists of the following types of *atoms*: ${t_1}^{[n]} = {t_2}^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \ll 1^{[n]}$. It is easy to check that any QF_BV$_{\ll 1}$ formula can be written like this with only a linear growth in the number of original variables.

We encode each atom in Φ separately into an atomic sequential circuit. The encoding itself is straightforward in most cases. A concrete example translating QF_BV to SMV is given after the theoretic part of this section. Compared to [208, 209], we have to consider the fact that all bit-vectors have a fixed bit-width.

Let n_{max} be the maximal bit-width of all bit-vectors in the formula. We construct an additional sequential circuit representing a counter. The counter initially is set to 0 and is incremented by 1 in each clock cycle. A counter like this can be realized with $\lceil \log_2(n_{max}) \rceil$ latches, i.e., polynomially in the size of Φ.

Now, for each atomic sequential circuit, we add a check whether the value of the counter reached the bit-width n of the bit-vector variables corresponding to the input streams of the circuit. Once this is the case, the individual circuit does not change its output value anymore. Since $n_{max} \geqslant n$, this will always hold at some point.[2]

Finally, after constructing all atomic circuits, their outputs are combined by logical gates following the Boolean structure of Φ. Other operators, such as *addition* or *indexing*, can either be replaced by *shift by one* in a preprocessing step or directly encoded into a sequential circuit [101].

We now show the translation for the motivational example given in Section 8.1 to the concrete SMV-format. First of all, a counter for the bit-width of the variables has to be introduced. This can be done using logarithmic many variables:

```
init(counter_bit0) := FALSE;
```

[2] In contrast to [208], we assume that the input streams for all variables start with the least significant bit.

```
next(counter_bit0) := counter_bit0 xor (TRUE);
init(counter_bit1) := FALSE;
next(counter_bit1) := counter_bit1 xor (counter_bit0);
...
init(counter_bit19) := FALSE;
next(counter_bit19) := counter_bit19 xor
  (counter_bit0 & ... & counter_bit18);
```

We then keep track of whether the counter already reached the value of a certain bit-width.[3] This variable later serves as a guard for all atoms containing variables of the given bit-width:

```
init(counter_gte_1000000) := FALSE;
next(counter_gte_1000000) := counter_gte_1000000 |
  (counter_bit0 & counter_bit1 & ... &
  !counter_bit6 & ... & counter_bit19);
```

After introducing those helper variables, the actual formula can now be translated. The *distinct* operator is first replaced by negation of an *equality*. The translation to SMV then is straightforward:

```
init(atom_equal) := TRUE;
next(atom_equal) := case
  counter_gte_1000000 : atom_equal;
  TRUE : atom_equal & (x <-> y);
esac;
```

For translating *addition*, two atoms have to be introduced since the carry bit has to be remembered in the next step:

```
init(atom_add) := TRUE;
next(atom_add) := case
  counter_gte_1000000 : atom_add;
  TRUE : atom_add & (z <-> (x xor y xor atom_cin));
esac;

init(atom_cin) := FALSE;
next(atom_cin) := case
  counter_gte_1000000 : atom_cin;
  TRUE : atom_add &
    ((x & y) | (x & atom_cin) | (y & atom_cin));
esac;
```

The *shift* operator can be translated in a very similar way but will not be given here explicitly to keep the example short. Another possibility would be to replace $(x \ll 1)$ by $(x + x)$ in the preprocessing step.

Finally, the specification is defined by the logical combination of the individual atoms and additionally respecting the bit-width:

[3] The counter bits in the *next*-statement correspond to the binary representation of $n - 1$ (i.e., $999999_{10} = 11110100001000111111_2$, in our example).

```
AG(!counter_gte_1000000 |
  !atom_add | !atom_shift | atom_equal)
```

We also implemented our translation, including various different operators, in a tool called Bv2smv. Binaries and source code are available for download at [60].

8.3 Experiments

We first describe our benchmark sets. We generated six different sets of QF_BV formulas in SMT2 format. All sets of benchmarks consist of 32 instances each and have two attributes: First, all benchmark sets are *not bit-width bounded* [101]. Because of this, bit-blasting is known to be exponential in general. Second, all benchmarks only contain *bitwise operators*, *addition*, *subtraction*, *shift by one*, *indexing* and *relational operators*. This ensures that a polynomial translation to SMV exists. The different instances in a particular set of benchmarks only differ in the bit-width of their variables and constants. The bit-widths n of the individual instances are of the form $n = 2^i$ and $n = 1.5 \cdot 2^i$ with $i \in \{5, \ldots, 20\}$ for all six sets. All benchmarks will be submitted to the QF_BV category of SMT-LIB.

QF_BV/froehlichkovasznaibiere/ndist.a.n:
We verify that, for two bit-vector variables $x^{[n]}, y^{[n]}$, it holds that $x^{[n]} < y^{[n]}$ implies $(x^{[n]} + 1^{[n]}) \leqslant y^{[n]}$. The instances are unsatisfiable and use *addition* and *unsigned less/greater than operators*.

QF_BV/froehlichkovasznaibiere/ndist.b.n:
We give a counter-example (due to overflow) to the claim that, for two bit-vector variables $x^{[n]}, y^{[n]}$, it holds that $(x^{[n]} + 1^{[n]}) \leqslant y^{[n]}$ implies $x^{[n]} < y^{[n]}$. The instances are satisfiable and use *addition* and *unsigned less/greater than or equal operators*.

QF_BV/froehlichkovasznaibiere/power2bit.n:
We verify that, for a bit-vector variable $x^{[n]} = 2^j$, it is not possible for two different bits to be both set to 1. The instances are unsatisfiable and use *indexing*, *subtraction*, *bitwise operators*, and *(in)equality*.

QF_BV/froehlichkovasznaibiere/power2eq.n:
We verify that, for two bit-vector variables $x^{[n]} = 2^j, y^{[n]} = 2^k$, with a certain identical bit set to 1, the bit-vectors cannot be distinct. The instances are unsatisfiable and use *indexing*, *subtraction*, *bitwise operators*, and *(in)equality*.

QF_BV/froehlichkovasznaibiere/power2sum.n:
We verify that, for two bit-vector variables $x^{[n]} = 2^j, y^{[n]} = 2^k$, with $j \neq k$, $x^{[n]} + y^{[n]}$ cannot be a power of 2. The instances are unsatisfiable and use *addition*, *subtraction*, *bitwise operators*, and *(in)equality*.

QF_BV/froehlichkovasznaibiere/shift1add.n:
We verify that for an arbitrary bit-vector $x^{[n]}$, there exists no bit-vector $y^{[n]} \neq x^{[n]}$ with $(x^{[n]} + y^{[n]}) = (x^{[n]} \ll 1)$. The instances are unsatisfiable and use *addition*,

shift by one, and *(in)equality*. The example used throughout the chapter is part of this benchmark family.

Out of the benchmark instances in SMT2 format, we generated SMV instances by using Bv2smv and the flattening tool smvflatten.[4] We used the state-of-the-art SMT solvers Boolector, MathSAT, Z3, and STP on the SMT2 instances, and NuSMV [66] on the corresponding SMV instances. In order to involve state-of-the-art model checkers like Tip [88] and IImc[5] (that uses techniques described in [40, 43]), we also converted all the SMV instances to AIGER format by using the translation tool smvtoaig that is part of the AIGER distribution.

All our experiments were run on the same cluster and with the same setup as the latest Hardware Model Checking Competition (HWMCC'12).[6] More precisely, we used a 32-node cluster with Intel Quad Core 2.6 GHz processors and 8 GB RAM. The wall clock time limit was set to 900 seconds and the memory limit to 7 GB. Each solver had full access to one node (4 cores). In total, we used 19 different solvers (or solver configurations) on 6 different benchmark sets each consisting of 32 instances, yielding a total of 3648 runs. All our results are available on our web page at [60] together with generation scripts for all benchmarks in SMT2 format and our tool Bv2smv.

Table 8.1 provides an overview of the total number of solved instances and the average runtime (in seconds) and space requirement (in megabytes) on the solved instances. For BMC solvers, we used the knowledge that the counters in the generated specifications only allow the atomic circuits to change their value in the first number of steps equal to the bit-width n of the original SMT2 formula. We therefore set the bound for unrolling to be equal to $n + 1$ and, whenever a BMC solver reached the bound without timeout or out-of-memory, counted the instance to be shown unsatisfiable. The solvers were executed with default settings if not stated otherwise explicitly. However, in some exceptional cases, we intentionally used some promising or interesting strategies. For instance, in Table 8.1, `Tip-BMC` references Tip using BMC-based strategy. Since we expected and later experienced that BDD-based techniques perform particularly well on our benchmarks, we intended to test model checkers with BDD-based strategies, those which offer such an option. Note that NuSMV uses BDD-based forward reachability analysis by default. We also tested NuSMV with backward reachability analysis, referenced by `NuSMV-bw`. IImc also offers BDD-based solving strategy, with both forward and backward reachability analysis; we reference IImc with default settings, with BDD-based forward, and with backward reachability analysis, as `IImc`, `IImc-BDD-fw`, and `IImc-BDD-bw`, respectively.

Apart from the solvers reported in Table 8.1, we tested other models checkers as well, all submitted to HWMCC'12. We excluded some of them due to uncer-

[4] `http://fmv.jku.at/smvflatten/`

[5] `http://ecee.colorado.edu/wpmu/iimc/`

[6] `http://fmv.jku.at/hwmcc12/`

[‡] Versions submitted to HWMCC'12.

	STP	Boolector	MathSAT5	Z3	IImc-BDD-bw	NuSMV-bw	IImc-BDD-fw	IImc	NuSMV	Blimc‡	Tip-BMC‡	Aigbmc‡	Tip
solved	147	146	127	123	**192**	189	185	172	170	147	130	99	93
sat	23	**32**	13	23	**32**	29	**32**	**32**	27	9	31	21	17
unsat	124	114	114	100	**160**	**160**	153	140	143	138	99	78	76
time	206	190	310	171	12	30	79	132	148	233	266	295	496
space	1063	805	587	2180	8	24	9	74	38	95	1142	2073	6

Table 8.1: Overall results for all solvers

tain results: (a) Super_prove2 and Simple_sat, which employ ABC with improved strategies, produced discrepancies on some satisfiable instances; (b) PdTrav, on some instances, threw exception about syntactical error in input.

In total, IImc-BDD-bw clearly performs best as it can solve all instances. Backward reachability analysis seems to produce better results than forward reachability for BDD-based model checkers in general. While this applies especially to unsatisfiable instances, NuSMV-bw only performs slightly better than NuSMV on the satisfiable ones. Interestingly, Boolector also gives very good results for the satisfiable instances. As expected, in particular the average space requirement of all SMT solvers is very large. Figures 8.1, 8.2, and 8.3 provide a detailed overview of the runtimes and space requirements of various solvers on the individual benchmark sets. We chose Boolector and STP representing the SMT solver class and NuSMV, NuSMV-bw, IImc, IImc-BDD-bw, and Tip-BMC as model checkers. Please consider that sampling memory is imprecise in case of low runtime, causing noise on the plots that show memory consumption.

Figure 8.1 shows the results of the solvers on the ndist.a and ndist.b benchmark sets. On the ndist.a instances, all BDD-based model checkers clearly outperform both SMT solvers considering time and space. Tip-BMC performs very similar to the SMT solvers. This is not surprising since unrolling up to a bound equal to the bit-width will in the end produce the same propositional formula as bit-blasting. With ndist.b being satisfiable, SMT solvers show better runtimes while still requiring similar amounts of space. This can be explained by the fact that it is enough to guess the correct assignment which might be found as a consequence of good heuristics and at the same time could cause the variation in the runtimes of STP. While backward reachability analysis seems to give a clear advantage on the unsatisfiable benchmark, it only slightly increases performance on the satisfiable one.

One interesting aspect in Figure 8.2 is the fact that STP performs really well on both benchmarks. We suppose that this is connected to the fact that power2bit

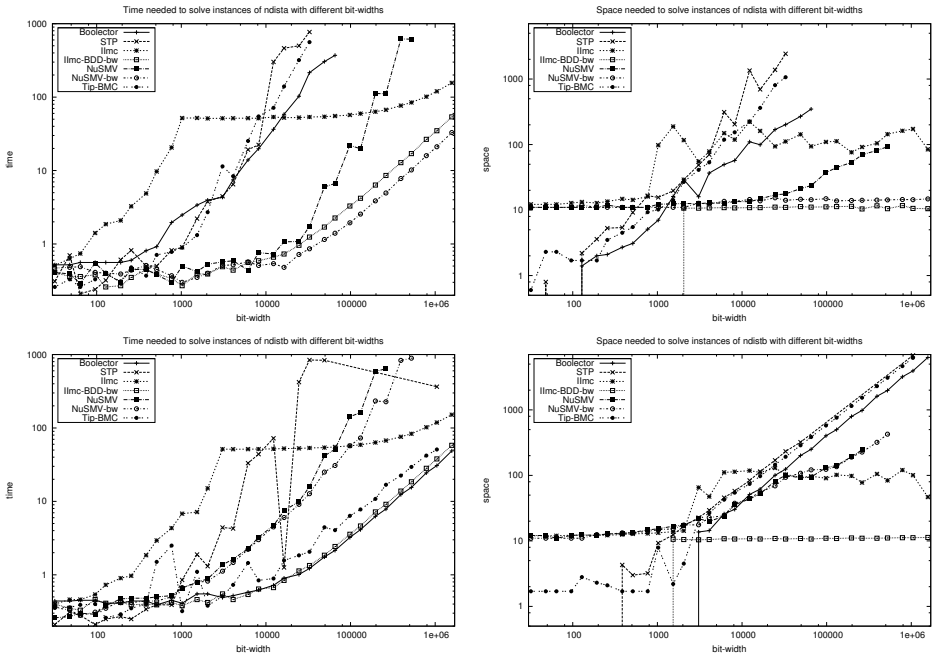


Figure 8.1: Detailed results of the `ndist.a` and `ndist.b` benchmark sets.

and `power2eq` both use indexing with relatively small indices. Interestingly, `Boolector` performs much worse on both instances. The good performance on this kind of formulas, therefore, does not seem to be a result of bit-blasting and applying SAT solvers but rather due to some special technique used in `STP`.

One might notice the typical shape of the runtime curves related to `IImc`: they start steep, but above a certain bit-width they show rather moderate ascent. The curves representing space consumption seem to grow slowly up to a certain point where, after a big jump, space usage almost seems to be fixed to a constant or, in some cases, even starts to decrease. We think that this strange behavior is due to the fact that `IImc` uses several scheduled approaches, such as IC3 [40], BMC, BDDs, etc. Probably due to the same fact, the `IImc` curves are even more hectic on the `power2bit` benchmark in Figure 8.2. During our experiments we also tested IImc with IC3 strategy alone, resulting in timeouts on most instances. Therefore, we assume that above a certain bit-width IImc with default scheduling switches to BDDs, resulting in moderate ascent in memory consumption and runtime.

Probably Figure 8.3 depicts most properly the distinction between BDD-based approaches and those which use SAT-based ones. Although SMT solvers and `Tip-BMC` time out quite soon on both problem sets, and, on the `power2sum` benchmark, the performance of `IImc` now is rather similar, BDD-based model checkers are able to deal even with very large bit-widths.

In general, looking at the runtimes, we can see that SMT solvers can compete well on instances with smaller bit-width, while BBD-based model checkers start to

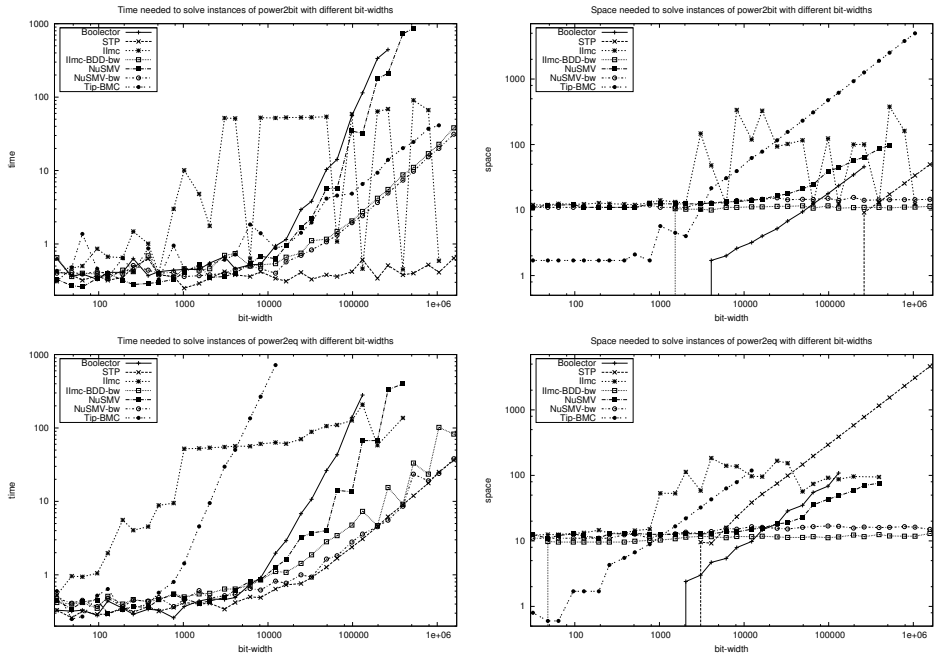


Figure 8.2: Detailed results of the `power2bit` and `power2eq` benchmark sets.

outperform their counter-parts with growing bit-width.

This effect becomes even stronger when we look at the space used during solving the formulas. Judging from the graphs, it might even be possible that the space requirement of BDD-based model checkers is logarithmic compared to that of SMT solvers. This could be the case due to the fact that SMT solvers apply bit-blasting, which is exponential for benchmarks that are not bit-width bounded, while our translation does not cause the problems to leave PSPACE. However, this alone is not sufficient. BDD-based model checkers like NuSMV might create exponential sized BDDs nevertheless. More rigorous arguments or larger empirical analysis are needed.

8.4 Conclusion

In this chapter, we efficiently solved quantifier-free bit-vector formulas by using model checkers. While state-of-the-art SMT solvers usually apply bit-blasting to solve this kind of formulas, we already showed in previous work [151] that this can cause an exponential blowup in general. An approach for polynomially translating QF_BV to EPR exists [150] (as well as exponential ones [90, 143]), but solving the resulting formulas also suffers from the NEXPTIME-completeness of EPR [150, 161]. Building on previous complexity results [101], however, we know that restricting QF_BV to only allowing *bitwise operators*, *shift by one*, *addition*, *subtraction*, *multiplication by constant*, *relational operators* and *indexing* leads to

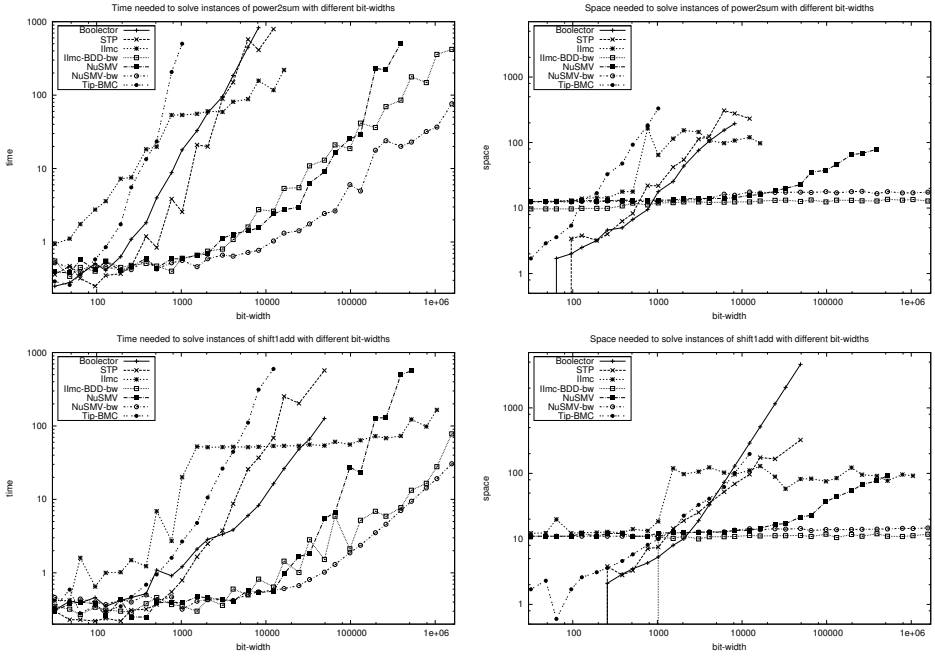


Figure 8.3: Detailed results of the `power2sum` and `shift1add` benchmark sets.

PSPACE-completeness of the resulting logic. This allows us to polynomially translate bit-vector formulas to sequential circuits and use model checkers for reachability analysis.

In order to show the potential benefit of our approach, we created a set of benchmarks and used it to compare the performance of various model checkers on the translated instances to the one of current SMT solvers on the original files. We showed that on most of our problems, state-of-the-art model checkers like IImc and even older ones, such as NuSMV, performed better by several orders of magnitude considering runtime as well as space.

Our results also showed that BDD-based model checking techniques perform much better than SAT-based model checkers. This probably is the case because of the similarity between BMC and bit-blasting, and gives reason to investigate especially BDD-based solving techniques further. Some of the best results were achieved by NuSMV. Considering the fact that NuSMV has seen relatively little development during the last years compared to current SMT solvers, this could lead to even better results if it is possible to improve the underlying techniques. One of the main reasons we assume to be responsible for the good performance of model checkers on our benchmarks, is their better fit to the PSPACE-nature of this problem class. Still, the resulting BDDs can of course be exponential in general.

While we did not pay special attention to the variable ordering during our translation, we ran NuSMV using `-dynamic` command, letting it figure out a good variable order during runtime. We also used the `-reorder` command to output

the optimal variable order found by NuSMV and to look for patterns in it. When using this variable order in a second run instead of choosing the order dynamically, the runtimes usually decreased further.[7] Maybe our translation can be adapted using additional information to directly create variable orders that result in smaller BDDs. In order to do this, it might be interesting to look at the structure of the instances produced by our translation more closely. Especially the usage of counter definitions and constraints is similar throughout all formulas.

Sequential optimization techniques, such as those implemented in state-of-the-art model checkers like ABC [44], are useful even for bounded model checkers which otherwise only rely on unrolling. It is an interesting question whether it is possible to lift these techniques from model checking to bit-vector reasoning in combination or as a preprocessing step before bit-blasting. Finally, only one model checker could solve all of our instances for the largest bit-widths. Constructing this kind of formulas, therefore, offers an easy way to provide challenging benchmarks for state-of-the-art SMT solvers and model checkers at the same time. For better solvers and future challenges, the difficulty of a problem can be adjusted by simply increasing the bit-width of the original SMT formula.

As a related classification problem, it will be interesting to investigate the complexity of Presburger arithmetic on fixed-size bit-vectors.[8] While the corresponding decision problem is known to be NP-complete for non-fixed-size bit-vectors, it is not clear whether we still remain in NP when considering fixed-size bit-vectors and whether translations as proposed in [45] are polynomial if a logarithmic encoding is used for the bit-widths.

[7] This is not included in our results since we did not analyze it in detail yet.

[8] The benchmark sets `ndist.a` and `ndist.b` are in this class.

Chapter 9

Quantifier-Free Bit-Vector Formulas with Binary Encoding: Benchmark Description

Published. In Proceedings SAT Competition 2013, A. Balint, A. Belov, M. Heule, M. Järvisalo (editors), volulme B-2013-1 of Department of Computer Science Series of Publications B, pages 107–108, University of Helsinki, 2013 [152].

Authors. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere.

Abstract This document describes several sets of benchmarks corresponding to quantifier-free bit-vector formulas. A generation script first creates all benchmarks in SMT2 format and then uses Boolector to generate CNF instances in DIMACS format by bit-blasting.

9.1 Introduction

Bit-precise reasoning over fixed-size bit-vector logics (QF_BV) is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. In [151], we argued that a *logarithmic* (w.l.o.g., *binary*) encoding, as used, e.g., in the SMT-LIB format [18], leads to NEXPTIME-completeness of the underlying decision problem. Bit-blasting, as used in most current SMT solvers, therefore, produces exponentially larger CNF formulas on certain QF_BV formulas. We provide generation scripts for several sets of QF_BV benchmarks in SMT-LIB format where this is the case and use bit-blasting to generate SAT benchmarks out of the original SMT2 specifications. All scripts and generated benchmarks are available at `http://fmv.jku.at/smtbench`.

9.2 Benchmarks

Our benchmark sets can be divided into two main categories: Expressing common bit-vector operations by other operations and general properties that can be expressed by a fragment of QF_BV with a restricted set of operations.

9.2.1 Translating Bit-Vector Operations

The first category contains 13 different benchmark sets and was used for verifying correctness of various translations between bit-vector operators. Having proved that *bitwise operations*, *equality*, and *slicing* suffice to derive NEXPTIME-hardness theoretically, we also wanted to give concrete examples of how to replace common bit-vector operations by those *base operations*. To check correctness, we encoded all translations into SMT2 and verified that no counter-example exists. We did this for 13 different operations. All benchmarks are unsatisfiable:

addition (`bvadd`), subtraction (`bvsub`), multiplication (`bvmul`), unsigned division (`bvudiv`), signed division (`bvsdiv`), unsigned remainder (`bvurem`), signed remainder (`bvsrem`), signed modulo (`bvsmod`), shift left (`bvshl`), logical shift right (`bvlshr`), arithmetic shift right (`bvashr`), unsigned less than (`bvult`), and signed less than (`bvslt`). To give one specific example, addition can be expressed by base operations as follows:

$t_1{}^{[n]} + t_2{}^{[n]}$ is replaced by $ts_1{}^{[n]} \oplus ts_2{}^{[n]} \oplus c_{in}{}^{[n]}$ and additional constraints

1. $ts_1{}^{[n]} = t_1{}^{[n]}$
2. $ts_2{}^{[n]} = t_2{}^{[n]}$
3. $c_{out}{}^{[n]} = (ts_1{}^{[n]} \& ts_2{}^{[n]}) \mid (ts_1{}^{[n]} \& c_{in}{}^{[n]}) \mid (ts_2{}^{[n]} \& c_{in}{}^{[n]})$
4. $c_{in}{}^{[n]} = c_{out}{}^{[n]} \ll 1^{[n]}$

are added. Now again, $c_{out}{}^{[n]} \ll 1^{[n]}$ can be replaced by $ts_3{}^{[n]}$ and additional constraints

1. $ts_3{}^{[n]}\,[n:1] = c_{out}{}^{[n]}\,[n-1:0]$
2. $ts_3{}^{[n]}\,[0:0] = 0^{[1]}$

are added.

While this is well-known for the example of addition, expressing multiplication or other operations by using only those base operations is much more complicated and cannot be detailed in the scope of this description. On the other hand, this already explains the benefit of verifying correctness by using our benchmarks.

9.2.2 Bit-Vector Properties in PSPACE

The second category consists of QF_BV benchmark sets with a reduced set of operations. In [101], we showed that QF_BV becomes PSPACE-complete under certain restrictions on the set of allowed operations. While bit-blasting still produces

exponentially larger formulas, the original benchmarks could be solved more efficiently, e.g., by using model checkers. It will be interesting to see whether any of the SAT solvers can also profit from this fact.

The 4 benchmark sets contained in this category are the following ones:

`ndist.a`: We verify that, for two bit-vector variables $x^{[n]}$, $y^{[n]}$, it holds that $x^{[n]} < y^{[n]}$ implies $(x^{[n]} + 1^{[n]}) \leqslant y^{[n]}$. The instances are unsatisfiable.

`ndist.b`: We give a counter-example (due to overflow) to the claim that, for two bit-vector variables $x^{[n]}$, $y^{[n]}$, it holds that $(x^{[n]} + 1^{[n]}) \leqslant y^{[n]}$ implies $x^{[n]} < y^{[n]}$. The instances are satisfiable.

`power2sum`: We verify that, for two bit-vector variables $x^{[n]} = 2^j$, $y^{[n]} = 2^k$, with $j \neq k$, $x^{[n]} + y^{[n]}$ cannot be a power of 2. The instances are unsatisfiable.

`shift1add`: We verify that for an arbitrary bit-vector $x^{[n]}$, there exists no bit-vector $y^{[n]} \neq x^{[n]}$ with $(x^{[n]} + y^{[n]}) = (x^{[n]} \ll 1^{[n]})$. The instances are unsatisfiable.

9.3 SMT2 and CNF generation

For each of the 17 benchmark sets, an individual generation script is provided. The scripts generate several instances of the given problem set, starting from a minimal bit-width up to a maximal bit-width, incrementing the bit-width by a given step size. Given those parameters as input, they output several SMT2 formulas with bit-vector variables of corresponding bit-widths. Additionally, a `generate.sh` script is included. This script automatically calls all individual generation scripts with appropriate parameters (i.e., bit-widths that create challenging but not too-hard instances) and afterwards calls *Boolector* [47] with argument `-de` to bit-blast the SMT2 instances and create CNF formulas in DIMACS format, therefore, directly providing the input benchmarks for the SAT solvers. Additional CNF instances corresponding to different bit-widths can be created manually by using the individual scripts with custom parameters and then translating the output with Boolector.

9.4 Practical Considerations

All our benchmarks were originally created to evaluate the performance of SMT solvers. While most benchmarks were challenging for all SMT solvers, some solvers turned out to perform particularly well on specific instances. So far, it is not clear whether this difference in performance is due to SMT rewriting rules, differences in bit-blasting, or because of the underlying SAT solvers. Therefore, it will be interesting to see how various SAT solvers perform on the bit-blasted version of our benchmarks.

Chapter 10

Stochastic Local Search for Satisfiability Modulo Theories

Published. In Proceedings 29th AAAI Conference on Artificial Intelligence, pages 1136–1143, AAAI Press 2015 [98].

Authors. Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi.

Abstract. Satisfiability Modulo Theories (SMT) is essential for many practical applications, e.g., in hard- and software verification, and increasingly also in other scientific areas like computational biology. A large number of applications in these areas benefit from bit-precise reasoning over finite-domain variables. Current approaches in this area translate a formula over bit-vectors to an equisatisfiable propositional formula, which is then given to a SAT solver. In this chapter, we present a novel stochastic local search (SLS) algorithm to solve SMT problems, especially those in the theory of bit-vectors, directly on the theory level. We explain how several successful techniques used in modern SLS solvers for SAT can be lifted to the SMT level. Experimental results show that our approach can compete with state-of-the-art bit-vector solvers on many practical instances and, sometimes, outperform existing solvers. This offers interesting possibilities in combining our approach with existing techniques, and, moreover, new insights into the importance of exploiting problem structure in SLS solvers for SAT. Our approach is modular and, therefore, extensible to support other theories, potentially allowing SLS to become part of the more general SMT framework.

10.1 Introduction

Satisfiability Modulo Theories (SMT) represents the decision problem for logical formulas with respect to certain background theories. It combines the problem of Boolean satisfiability (SAT) with other areas, e.g., the theories of integers, real

numbers, lists, arrays, and bit-vectors, and has many different applications, predominantly in hard- and software verification [82, 17, 236, 115, 179]. While most of the methods presented in this chapter are generally applicable, we focus on the theory of bit-vectors (quantifier-free and fixed-size), which enjoys decidability, but pays the high price of being NEXPTIME-complete, as [151] have shown. Examples of state-of-the-art SMT solvers with support for bit-precise reasoning are Boolector [47], MathSAT [50], and Z3 [81].

Most approaches for solving this kind of formulas rely on translating the input formula into SAT (dubbed 'bit-blasting') and then handing it to a SAT solver, most often of the conflict driven clause learning (CDCL) kind. In this chapter, we present a novel stochastic local search (SLS) algorithm to solve bit-vector formulas directly on the theory level. SLS is a heuristic method which has always played an important role in AI and is successfully applied to many different problems in various areas, e.g., see [128]. Today, SLS is not a usual ingredient in SMT solvers. We intend to close this gap by providing an SLS algorithm (specialized for the theory of bit-vectors), which is, for the most part, easy to adapt for other theories. Besides avoiding the blowup in size that often comes with bit-blasting, applying SLS techniques on the bit-vector level has several advantages. For example, structural information, i.e., word-level information, is used to guide the search directly. In contrast, a CDCL SAT solver operating on the propositional representation is not aware of this information. Nevertheless, it is possible to profit from techniques used in SAT solving also in SLS on the SMT representation. We show how several techniques that are common in SLS SAT solvers are successfully lifted to the SMT level. In many cases of practical applications [236, 115, 179], input formulas are actually expected to be satisfiable, making them well-suited for SLS algorithms. The idea of integrating SLS solvers with other solvers has been explored before, either by employing an SLS solver on the Boolean skeleton of a formula [119], or by explicit incorporation of high-level constraints, either learned automatically, or provided by the user [178]. Apart from this, model-driven techniques (though, not local search based) exist for arithmetic theories [83].

Our experimental results show that the SLS approach we present is competitive with state-of-the-art bit-vector solvers on many practical instances and that it frequently outperforms existing SMT solvers based on bit-blasting. While we found that bit-blasting solvers are still faster overall, this offers an interesting line of research combining SLS with existing techniques, with the goal of improving the state-of-the-art for SMT solvers. Although having undergone years of developement, existing SLS solvers for SAT turn out to perform worse than our approach, sometimes by orders of magnitude. From a theoretical point of view, the importance of exploiting problem structure in SAT solvers has often been conjectured and discussed [218, 21, 86, 139, 189]. Nevertheless, previous attempts have not yielded in efficient techniques that play a role in state-of-the-art solvers so far. However, the performance of our algorithm clearly demonstrates that SLS solvers can indeed benefit from structural information during search.

The remaining part of the chapter is structured as follows: In Section 10.2, we

define the logic and the input format that we use. A brief overview about stochastic local search and the architecture of our algorithm is presented in Section 10.3. In Section 10.4, we give details and concrete implementations of all components used in our algorithm. Furthermore, we describe how several techniques used in SLS solvers for SAT are adopted for SMT. We present an experimental evaluation in Section 10.5 and discuss insights we gained, as well as possible future work in Section 10.6. Finally, we compare our approach to related work in Section 10.7 and conclude in Section 10.8.

10.2 Preliminaries

The theory of fixed-size bit-vector logics (i.e., logics where each bit-vector has a given, fixed bit-width) is discussed in many different settings (e.g., [19, 35, 51, 74, 96]). Several different formats for bit-vector logics exist, perhaps currently the most common being the SMT-LIB format [18]. In this chapter, we use a restricted definition of a bit-vector logic, which is the input that our SLS algorithm accepts. This form of (simplified) formulas is easily obtained through the means of any SMT solver that has facilities for converting to Negation Normal Form (NNF; we use Z3 as our SMT solver). A bit-vector formula F in NNF is defined by the following grammar:

$$\begin{aligned} F &= < oexpr > \wedge \cdots \wedge < oexpr > \\ < oexpr > &= < aexpr > \vee \cdots \vee < aexpr > \\ < oexpr > &= Atom \mid \neg Atom \\ < aexpr > &= < oexpr > \wedge \cdots \wedge < oexpr > \\ < aexpr > &= Atom \mid \neg Atom \end{aligned}$$

Atoms are either Boolean variables or relations ($=, \leqslant$) between two bit-vector expressions. We refer to the top-level expressions of F as *assertions*. It is easy to check that every bit-vector formula can be translated to an equivalent one in this grammar with only polynomial growth. To see this, consider that $\leqslant$ can be replaced by a combination of $<$ and $=$, $\geqslant$ by negation, and $<$, and if-then-else constructs by using implications on the Boolean level. In some cases, to achieve conversion in polynomial time (and space), it is helpful to introduce Tseitin variables.

The concrete definition of a bit-vector term is left open on purpose; the exact syntax and semantics of the terms are not relevant in the context of our approach. All common operators, e.g., those from SMT-LIB [18], can be used to build arbitrary syntactically valid expressions. The only condition we require is that there is a function to evaluate expressions if fixed input values are assigned to all variables they contain.

Example 10.1. As a running example, consider the assertion

$$x + 3 = \sim x \ ,$$

where x is a bit-vector of size n (sometimes a large number), $\sim$ denotes bitwise negation, and the $+$ operation is as usual, i.e., with overflow semantics. To simplify our example, assume $n = 6$. If we initialize the search at $x = 0$, or in vector notation, at

$$x = [0, 0, 0, 0, 0, 0] \ ,$$

then the assertion evaluates to

$$[0, 0, 0, 0, 1, 1] = [1, 1, 1, 1, 1, 1] \ .$$

Furthermore, assume that the cost function s for $=$ is the relative number of bits that are assigned equal. Initially, $s = \frac{2}{6}$.

10.3 Architecture

Given an optimization problem, a generic local search algorithm starts from an initial state and then iteratively moves to a neighbouring state. For the problem of propositional satisfiability, a state corresponds to a truth assignment to all Boolean variables of a given formula. The neighbourhood of a given assignment α is usually defined to be the set of all assignments that have a Hamming distance of 1 from α. Therefore, a neighbouring assignment is obtained by flipping the value of a single Boolean variable and a search consists of repeatedly flipping the values of Boolean variables until a satisfying assignment is found. Most SAT solvers consider input formulas in conjunctive normal form (CNF), i.e., formulas which are sets of clauses. In that case, a scoring function for evaluating the quality of an assignment and optimizing is naturally given by the number of unsatisfied clauses. Actual implementations mainly differ in the heuristics used during the search [128].

We use a similar architecture to obtain an SLS solver for SMT problems by generalizing the notion of states to assignments to theory variables; our focus being on fixed-size bit-vector variables. A natural neighbourhood relation is then given by the set of assignments that are reached by flipping a single bit of a bit-vector variable, or the value of a Boolean variable. In the following, whenever we use the term 'variable' without giving an explicit specification of its type, the variable is either a bit-vector variable or Boolean. When we have a set that contains both types of variables and we only give a certain definition for the bit-vector variables, we implicitly treat Boolean variables as bit-vector variables of bit-width 1. Extensions to this neighbourhood relation are discussed in Section 10.4. Figure 10.1 describes the high-level concept of our SLS algorithm for SMT.

To drive the search and to evaluate the quality of an assignment, we require a scoring function. We define the score s of a nested expression with respect to an assignment α recursively as a floating value:

$$s(e_1 \vee \cdots \vee e_n, \alpha) = max\{s(e_1, \alpha), \ldots, s(e_n, \alpha)\}$$

$$s(e_1 \wedge \cdots \wedge e_n, \alpha) = \tfrac{1}{n} \cdot (s(e_1, \alpha) + \cdots + s(e_n, \alpha))$$

```
1 procedure SLS4SMT(F)
2   for i = 1 to ∞
3     α = initialize(F);
4     for j = 1 to maxSteps(i)
5       V = selectCandidates(F,α);
6       move = findBestMove(F,α,V);
7       if (move ≠ none) α = update(α,move);
8       else α = randomize(α,V);
```

Figure 10.1: Pseudo-Code of our SLS architecture for SMT.

Furthermore, the score of an atom is defined by

$$s(x^{[1]}, \alpha) = x|_\alpha \,,$$

if the atom is a Boolean variable and, for $0 \leqslant c_1 \leqslant 1$, by

$$s(t_1^{[n]} = t_2^{[n]}, \alpha) = \begin{cases} 1 & \text{if } t_1|_\alpha = t_2|_\alpha \\ c_1 \cdot (1 - \frac{h(t_1|_\alpha, t_2|_\alpha)}{n}) & \text{otherwise} \end{cases},$$

$$s(t_1^{[n]} \leqslant t_2^{[n]}, \alpha) = \begin{cases} 1 & \text{if } t_1|_\alpha \leqslant t_2|_\alpha \\ c_1 \cdot (1 - \frac{t_1|_\alpha - t_2|_\alpha}{2^n}) & \text{otherwise} \end{cases},$$

with h being the Hamming distance, if the atom is a bit-vector expression. Negated atoms are evaluated analogously. The constant c_1 allows to focus on satisfying expressions by scaling all unsatisfied atoms. It is easy to check that, given a formula F and an assignment α, $s(F, \alpha)$ evaluates to 1 if and only if α is a satisfying assignment for F.

10.4 Implementation

Note that the algorithm in Figure 10.1 contains two loops. The inner loop describes a single round of search, while the outer loop is used to implement restarts after a certain number of search steps. Facilities for restarts are not strictly required, but they increase performance in practice.

Initialization. An initial assignment is generated by setting all variables to some specific value. While SLS solvers for SAT usually use random values to initialize Boolean variables, setting all bit-vectors to 0 can sometimes be beneficial in the context of verification domains. Note that setting all bit-vectors to 0 does not correspond to setting all Boolean variables to 0 in the CNF representation. Without explicit tracking, this information is usually lost during bit-blasting.

Candidate Selection. The time spent in each search step is directly proportional

to the number of possible moves that are considered. Checking the full neighbourhood of an assignment is often expensive. To avoid this, we look at the restricted neighbourhood with respect to certain *candidate variables*. Since we are looking for a satisfying assignment and our input formula is a conjunction of top-level assertions, it is reasonable to consider those variables as candidates that occur in at least one unsatisfied assertion. Changing any other variable cannot increase the score by definition. This is a well-known concept in SLS for SAT; similar to the one applied in selection heuristics of the so-called class of *GSAT* algorithms. The set of candidate variables is then further shrunk by considering only variables from *one* unsatisfied assertion, which is selected according to some heuristic beforehand. This is inspired by the so-called class of *WalkSAT* algorithms for SAT. While this comes at the cost of potentially missing the best move with respect to the score function, the overall performance of the algorithm often improves because it performs more moves per second and, at the same time, it is guaranteed that each clause has at least one variable with a wrong assignment. Furthermore, not picking the best move with respect to the overall score is even beneficial sometimes, because it offers some diversification and makes the algorithm more robust with respect to local minima of the search space. For those reasons, WalkSAT architectures are often preferred for SAT. More details and a discussion of the GSAT and WalkSAT architectures are found in [128].

For SAT, clause selection in most WalkSAT algorithms is usually done randomly. However, recent work shows that clause selection has a strong impact on the performance of WalkSAT algorithms and random selection is sometimes suboptimal. For example, breadth-first selection heuristics sometimes achieve better results [13]. Still, clause selection has to be rather simple because SLS solvers for SAT often perform several million moves per second. In contrast, fast assertion selection is less important for our architecture, due to the fact that a single move is much more complicated compared to SAT. This allows us to use more sophisticated heuristics for assertion selection without the risk of it becoming the bottleneck of our algorithm.

Running some preliminary experiments showed that it is frequently better to select assertions that already have a high score (i.e., are almost satisfied). Given the results from [13], it is likely that some diversification is beneficial as well. Therefore, we use a heuristic inspired by the field of bandit theory used in the UCB (Upper Confidence Bounds) algorithm [1]. Let a_i be the assertions of a given formula and c_2 be some constant. We select the unsatisfied assertion that maximizes the term

$$s(a_i, \alpha) + c_2 \cdot \sqrt{\frac{\log selected(a_i)}{moves}},$$

where $selected(a_i)$ is the number of times, the specific assertion a_i has already been selected and $moves$ is the total number of search steps that have been performed so far.

Move Selection. Given a candidate set, we inspect the neighbourhood of the current assignment with respect to all candidate variables. In particular, this implies

evaluating the score of each possible assignment obtained by flipping any bit of any candidate variable. Given a set of candidate variables $\{x_1^{[n_1]}, \ldots, x_k^{[n_k]}\}$, the neighbourhood is of size $N = \Sigma_i n_i$. Compared to SAT, the neighbourhood used in our architecture is way larger. Furthermore, evaluating the score function implies updating the whole formula. This is again in contrast to SAT, where the score change is either cached or easy to compute on the fly [13]. Evaluating possible moves, therefore, is the bottleneck of our current implementation. As pointed out in Section 10.3, we try to maximize the score function. One important feature to use during score computation is early pruning. In our solver, a formula is saved in a directed acyclic graph (DAG) structure. When evaluating a new assignment, we have to proceed bottom-up. We start evaluating the atoms and then iteratively continue evaluating parent expressions. We do this in a breadth first way and save all current expressions in a queue structure. Due to the definition of the score function and the fact that our formula does not contain any negations other than those at the atom level, we can immediately stop evaluating a specific assignment if, at one point, the queue only contains expressions with lower scores than those for the same expressions with respect to the original assignment.

In the end, the move with the largest improvement in score is selected. If no improvement in score is possible, no move is returned. This is similar to the behaviour of many SLS algorithms for SAT. For example, the state-of-the-art solver *Sparrow* [14] also applies a similar deterministic highest-reward strategy in one of its components. In order to prevent getting stuck in local minima, we optionally allow *random walks* [203]. With a certain walk probability wp, a random move is selected (even if it is non-improving). From a theoretical point of view, a random walk additionally causes our algorithm to be *probabilistically approximately complete (PAC)* [128].

Example 10.2. Consider the assertion in Example 10.1 and the state of the search being such that x is assigned 0 (for $n = 6$). The assertion evaluates to

$$[0, 0, 0, 0, 1, 1] = [1, 1, 1, 1, 1, 1] \ .$$

Flipping any single-bit either increases the score by $\frac{1}{6}$ ($\frac{1}{n}$, in the general case), or does not increase the score at all. In contrast, negating x directly improves the score by $\frac{3}{6}$ ($\frac{n-2}{n}$, in the general case). Thus, the algorithm would negate x and the assertion would now evaluate to

$$[0, 0, 0, 0, 1, 0] = [0, 0, 0, 0, 0, 0] \ ,$$

with $x = [1, 1, 1, 1, 1, 1]$ and $s = \frac{5}{6}$ being the new values of x and the score function, respectively.

Update. After an improving move was found, the assignment is updated and propagated through the DAG structure of the formula. As mentioned before, scores are

also stored for each subexpression when updating, in order to allow early pruning in the next search steps.

Randomization. Whenever no improving move was found, we simply set one of the candidate variables to a random value within its range and update all nodes in the formula DAG. This strong kind of randomization enables the algorithm to efficiently escape many local minima and allows to traverse new parts of the search space. In contrast to random walks during move selection, this part of the algorithm is essential for solving practical instances. Nevertheless, using only this kind of randomization does not guarantee the PAC-property from the theoretical point of view.

Assertion Weights. In SLS solvers for SAT, clause weighting schemes were an important novelty and are part of many efficient algorithms [128]. PAWS was the first solver to apply an additive weighting scheme and the same approach can still be found in many modern solvers [219]. We adopted this approach to SMT and used it to dynamically assign weights to the top-level assertions of an input formula during search. Each assertion a_i of F gets assigned a weight w_i. Initially, all weights are set to 1. Updates occur whenever no increasing move is possible, i.e., when we randomize, in the following way: With probability $(1 - sp)$, increase the weight w_i of all unsatisfied assertions by c_3. With probability sp, decrease the weight w_i of all satisfied assertions by c_3 to a minimum of 1. Whenever the score of the formula F with respect to an assignment α is evaluated in order to select the best move, we do so according to

$$s(F, \alpha) = w_1 \cdot s(a_1, \alpha) + \cdots + w_n \cdot s(a_n, \alpha) \ .$$

Although this new score function is not normalized anymore, this does not affect the correctness of the algorithm.

Restarts. While restarts are one of the most important features of CDCL solvers, they are usually not beneficial in SLS solvers for SAT. For our bit-vector approach, restarts turn out to be beneficial. We implemented an exponential restart scheme, similar to those that are used in CDCL solvers, specifically, the Luby scheme [168]. We define the maximum number of steps in the i-th round as

$$\textit{maxSteps(i)} := \begin{cases} c_4, & \text{if } i \text{ is odd} \\ c_4 \cdot 2^{\frac{i}{2}}, & \text{if } i \text{ is even} \end{cases}$$

This is different to existing schemes in the sense that it has more very short runs but at the same time it grows faster.

Note that restarts in our implementation only refer to a reset of the current assignment. In contrast, they do not imply a reset of information gathered during search, e.g., how often an assertion has already been selected or the weight of an assertion. Preliminary experiments showed that it is beneficial to keep those values. Intuitively, this allows learning from previous runs to make better decisions in later ones.

Extended Neighbourhoods. As pointed out in Section 10.3, a simple neighbourhood relation is given by flipping single bits of bit-vector variables, which is very similar to the neighbourhood considered in SLS solvers for SAT. It is easy to see that this neighbourhood relation already allows traversing the full search space. Nevertheless, extended neighbourhoods tailored towards bit-vectors often have advantages. We therefore included three additional moves for bit-vector variables in our algorithm: Incrementing by 1, decrementing by 1, and bitwise negation. Given a set of candidate variables $\{x_1^{[n_1]}, \ldots, x_k^{[n_k]}\}$, the neighbourhood then is of size $N' = \Sigma_i(n_i + 3)$. Considering the fact that n_i often is 16 or 32 in bit-vector applications, the overhead is relatively small and usually outweighed by the benefit of permitting those natural bit-vector moves. We also tried implementing other moves, such as shifts by 1, multiplication by 3, or unary minus, but could not further improve performance by doing so, in general. For future work, it will be interesting to combine our approach with techniques used in the context of *Variable Neighbourhood Search (VNS)* [174, 121]. Preliminary experiments showed promising results.

Example 10.3. We left Example 10.2 at $x = [1, 1, 1, 1, 1, 1]$, with the assertion evaluating to

$$[0, 0, 0, 0, 1, 0] = [0, 0, 0, 0, 0, 0] \ ,$$

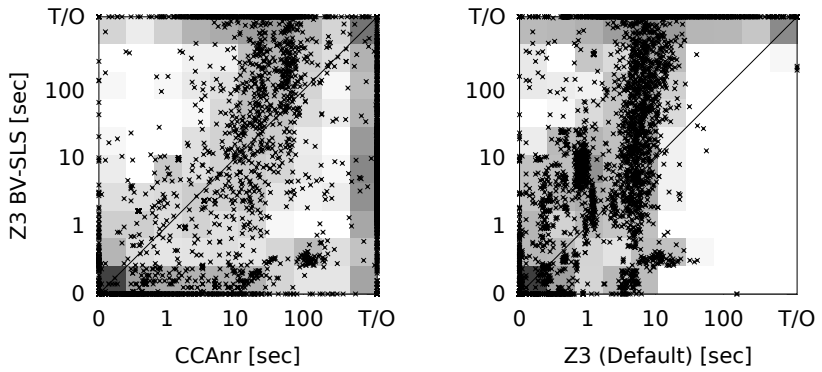
which is already very close to a solution. Flipping the least significant bit is the only move that will further increase the score. We get $x = [1, 1, 1, 1, 1, 0]$, and the assertion will now evaluate to

$$[0, 0, 0, 0, 0, 1] = [0, 0, 0, 0, 0, 1] \ .$$

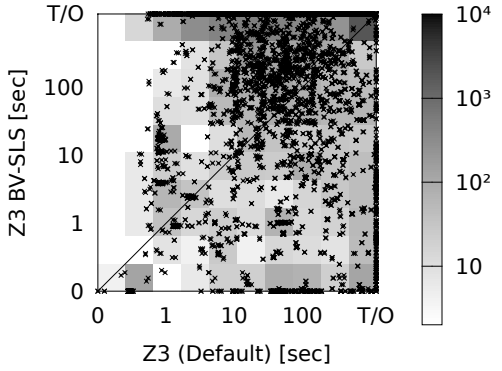
Thus, we arrive at a solution for this example in only two moves. Note that the number of moves does not depend on the size of the vectors; $n - 1$ separate bit-flip moves, each of which slightly improving the cost, are replaced by two moves.

10.5 Experimental Results

To evaluate the performance of our algorithm, we ran experiments on two different sets of benchmarks. The first benchmark family is the QF_BV benchmark set, which can be found in the SMT-LIB and is also part of the SMT Competition. The QF_BV benchmark set is a huge and broad collection of benchmarks, consisting of many smaller families and is the standard reference for measuring the performance of bit-vector solvers. We ran Z3 [81] on the full benchmark set of 33068 instances and removed all those which Z3 proved to be unsatisfiable within 1200 seconds. From the remaining 11715 instances, we further filtered out those 4543 formulas, that were shown to be satisfiable only by using preprocessing techniques. This left us with a total of 7498 instances in the QF_BV set for the following experiments. A second benchmark family is given by the SAGE2 benchmark set. Those



(a) QF_BV (b) QF_BV



(c) SAGE2

Figure 10.2: Scatter plots and heat maps comparing BV-SLS to CCAnr on QF_BV and to Z3 on both benchmark sets.

problems were generated as part of the SAGE project at Microsoft [115], describing some testcases for automated whitebox fuzz testing. Older benchmarks from the SAGE project can also be found as part of the QF_BV benchmark set. The SAGE2 set consists of 8017 instances (filtered out of original 17920 instances, 9903 were shown to be unsatisfiables within 1200 seconds, none were solved by preprocessing only), which are known to be hard for state-of-the-art SMT solvers. All experiments were run on a Windows HPC cluster of dual Quad-Xeon (E54xx) machines, 16 GB RAM, and used a time limit of 1200 seconds.

We compared our new solver BV-SLS to the most recent version of the state-of-the-art SMT solver Z3, which is based on bit-blasting and then running a CDCL SAT solver on the propositional encoding. For all benchmarks, we used the default configuration of BV-SLS: All variables are initialized to 0, candidate selection occurs using the UCB scheme, constants are set to $c_1 = 0.5$, $c_2 = 20$, $c_3 = 0.025$, $c_4 = 100$, $wp \approx 0.1$, $sp \approx 0.05$, and it uses the extended neighbourhood relation that additionally allows increment by 1, decrement by 1, and bitwise negation.

To evaluate the direct benefit of using bit-vector information for SLS, we ran

	QF_BV	SAGE2
CCAnr	**5409**	64
CCASat	4461	8
probSAT	3816	10
Sparrow	3806	12
VW2	2954	4
PAWS	3331	**143**
YalSAT	3756	142
Z3 (Default)	7173	5821
BV-SLS	**6172**	**3719**

Table 10.1: Number of solved instances.

several state-of-the-art SLS solvers for SAT on propositional encodings of our benchmarks as CNF. To obtain those encodings, we used the bit-blasting component of Z3. This conversion was done together with preprocessing in advance to the experiments (also using a time limit of 1200 seconds). We did not add this to the actual runtime of the solvers, assuming that the input is directly given as a preprocessed CNF. In theory, this gives an advantage to the SAT solvers. Furthermore, CNF conversion did not succeed for all instances. This was either because Z3 ran out of memory (M/O) or because it did simply not terminate in the given time limit (T/O). In total, CNF conversion produced 21 M/O and 75 T/O results for QF_BV, and 29 T/O results for SAGE2. We considered those instances as not being solved by the SAT solvers, since it was not feasible to obtain a CNF representation in the first place. Z3, using bit-blasting, could not solve any of those either. BV-SLS was also not able to solve any of the corresponding QF_BV instances, but found a solution in 13 cases for the SAGE2 formulas. As SLS SAT solvers, we used several versions of CCA [62], probSAT [15], Sparrow [14], YalSAT [27], and the implementations of PAWS [219] and VW2 [195] in UBCSAT [220]. CCASat, probSAT, and Sparrow have consistently achieved good results over the last SAT Competitions. CCAnr, PAWS and VW2 are known for performing particularly well on some application benchmarks. YalSAT was among the few 'pure' SLS solvers (i.e., not using a CDCL component) that produced good results in the application track of the latest SAT Competition. We also tried to include Sattime2014r [163], but encountered difficulties porting it to Windows. However, Sattime2014r had very similar performance to YalSAT in the application track of the SAT Competition 2014.

The number of solved instances is given in Table 10.1. The scatter plots and heat maps in Figures 10.2a, 10.2b, and 10.2c provide details about the runtime behavior of our implementation. All solvers were run once (i.e., with one seed) per instance.

10.6 Discussion

The results from Section 10.5 provide several insights. First of all, comparing our SLS algorithm on the bit-vector representation with SLS solvers for SAT showed that we can actually profit from using additional word-level information, especially on the SAGE2 benchmark set (Table 10.1). This is particularly interesting in the context of SAT solvers. While SLS solvers for SAT are known to perform well on randomly generated formulas and, sometimes, on hard combinatorial benchmarks, CDCL solvers usually perform much better on so-called *structured* formulas, often coming from practical problems in industry which have been re-encoded into propositional logic. This is often attributed to the fact that CDCL solvers are able to learn during search and make inferences, extracting this kind of original structure from a propositional formula. In contrast, most SLS solvers only use very local information. Exploiting structure for SLS SAT solvers has been looked at before, but did not yield in efficient solvers so far. Our results clearly show that SLS solvers can actually profit from using structural information during search.

While there is still a gap between the average performance of our solver compared to state-of-the-art SMT solvers based on bit-blasting and CDCL (Table 10.1), our approach can actually outperform Z3 on several instances, especially among those contained in the SAGE2 benchmark set (Figure 10.2c). This is interesting from two different points of view. First, our algorithm is a completely new approach, which has not yet had several years of research, development and tuning that CDCL algorithms have seen. It is very likely that data structures, implementation and heuristics can still be improved easily for our approach. For example, improvements might be found by adopting further techniques which have already been applied successfully in SAT solving or by using more complex heuristics that are not possible to realize in the propositional case. Since SLS solvers are known to be very sensitive with regard to their parameters, automated configuration, as applied for SAT [130], could be beneficial for our approach too. Second, combinations of our algorithm with existing SMT solvers seem promising, because both kind of solvers often perform well on distinct kind of problems (Figure 10.2c). Simply running an SLS component for a very short time before the actual SMT solver could already help finding solutions for many additional problems, potentially improving the state-of-the-art in SMT by SLS. In a further step, solvers could also start to exchange information between each other, as it has already been tried for SLS and CDCL solvers for SAT [155], e.g., by initializing the VSIDS values of the CDCL solver according to information gained by a previous SLS run.

Another possibility for future work is the extension of our algorithm to allow other theories apart from bit-vectors. As described in Section 10.3, the underlying architecture of our algorithm is very general. The only time we actually use bit-vector information is in the definition of the score function for bit-vector expressions and the neighbourhood relation. All other components, as described in Section 10.4, as well as their improvements by techniques known from SAT solving, only take into account the Boolean part of a given formula. To allow dealing

with arbitrary other theories, it is sufficient to provide a score function for the theory expressions as well as a neighbourhood relation on the theory variables.

10.7 Related Work

[119] defines the *WalkSMT* algorithm, which uses an SLS solver for the Boolean abstraction of a given problem, but they do not exploit SLS on the theory level. An additional theory solver is used to check satisfying Boolean assignments for theory consistency and, if they are inconsistent, to refine the abstraction in a lazy way.

[178] presents a stochastic CSP solver that applies a bit-string encoding and then uses a stochastic search algorithm. However, their description is rather high-level and no concrete implementation is given. The most significant difference to our work can be found in the fact that we explicitly look at the problem from an SMT perspective. By doing so, we are able to successfully lift many sophisticated techniques from SAT solving to our approach. The experimental evaluation in [178] is only performed on a limited set of crafted benchmarks. By using the full QF_BV benchmark for our evaluation, we provide a more detailed picture of the overall performance of our solver.

10.8 Conclusion

In this chapter, we proposed an approach towards bridging the gap between SMT and SLS. We presented a novel SLS algorithm to solve bit-vector formulas directly on the theory level. Furthermore, we explained how several techniques used in SLS solvers for SAT can be lifted to the SMT level and gave experimental results, confirming the benefit of applying local search directly on the bit-vector representation instead of using a propositional encoding. This gave new insights into the importance of exploiting problem structure also in SLS solvers for SAT. While there is still a gap in performance compared to state-of-the-art bit-vector solvers, our approach outperforms Z3 on many instances of practical relevance. This offers interesting possibilities in combining our solver with existing approaches, potentially improving the performance of both.

Chapter 11

Contributions

As noted at several occasions, all of the work presented in Chapters 2–10 was the result of joint work with other authors. Much of this would probably not have been possible individually, and all papers were influenced by each of the authors. In this section, my individual contribution to each of the chapters is pointed out. Note that, even when some specific idea is attributed to a concrete author, the idea usually was a product of many iterations, always being the result of steadily ongoing discussions between all of the authors.

Chapter 2 [151]. The idea of satisfiability of general bit-vector formulas, i.e., not unary encoded ones, probably not being NP-complete, resulted from a joint discussion of the authors. While NP-completeness does hold for the unary case, we quickly realized that the principle of bit-blasting cannot be used to argue for NP-inclusion when a logarithmic encoding is used. It was Gergely who first came up with the translations that we used to show NEXPTIME and 2-NEXPTIME-hardness of our distinct classes. Gergely also was the first to formalize the concept of bit-width boundedness. My main contribution was to formally show correctness and polynomiality of the main reduction. Aside from this, I specified the inclusion results and was involved in the writing process.

Chapter 3 [101]. The central idea from this work, being the fact that restricted sets of operators probably lead to less expressive bit-vector logics, again originated from joint discussions of all authors. I then was the first to prove that the presented fragments are actually PSPACE-complete and NP-complete. Furthermore, I showed how addition, multiplication and slicing are related to the presented logics. On the other side, Gergely proved the possible extension for relational operators. While Gergely also provided a formal definition of the bit formulas used in our paper, I was responsible for writing the remaining part of the paper.

Chapter 4 [153]. Being a journal submission, a large part of the content in this paper originated from the previous work presented in Chapter 2 and Chapter 3. Considering the modifications and extensions of our previous work, Gergely formally introduced the detailed formalization including syntax and semantic of our

bit-vector formulas, as well as the Common Operator Framework and the extended concept of scalar boundedness. Gergely also reworked the NEXPTIME-completeness proof for our main bit-vector logic and added several extensions for some of the fragments. In particular, Gergely introduced if-then-else constraints, certain relational constraints and right shifts for the PSPACE-fragment, and general shifts (using barrel shifts), equivalence of extraction and concatenation, and shuffle versions, for the NEXPTIME-fragment. I reworked the PSPACE- and NP-completeness proofs and, moreover, introduced several new extensions for the NP-fragment, allowing indexing and relational operators. Furthermore, I formalized the alternative characterizations for the PSPACE class, including addition and multiplication by constant. For the NEXPTIME class, I showed the equivalent characterization by extraction or concatenation, and the particularly sophisticated one by multiplication. Additionally, I pointed out the relation of our 2-NEXPTIME class to Peano arithmetic and introduced the concept of universal bit-width boundedness as well as quantification over array indices. I also showed the complexity results for the universal bit-width-bounded class and for quantifier-free logics with non-recursive macros. Writing was done in equal parts.

Chapter 5 [99]. The idea of developing a dedicated DQBF solver was the result of joint discussion. I then introduced the DQDPLL approach by proposing a way to extend the well-known QDPLL approach to allow dealing with partially ordered quantification in the form of adding Skolem clauses to the solver. Prototypical implementation as well as addition of most techniques in the paper was done by me. Gergely then added the concept of watched literals and optimized some of the data structures. The main part of the writing was done by me.

Chapter 6 [150]. The main idea to solve bit-vector formulas by giving a translation to EPR and then using IPROVER originated in a joint discussion. The actual translation then was formulated by Gergely, who also did the main part of writing in this paper. My main contribution was the construction of the add2n benchmark set that was used for the experiments.

Chapter 7 [102]. Combining the goal of developing an efficient DQBF solver with the observed effectiveness of IPROVER on many problems, I came up with the idea of creating the instantiation based DQBF solving approach presented in this paper. The prototypical implementation as well as the main part of the writing was done by me. Gergely contributed by extending the original implementation, fixing bugs as well as increasing performance of data structures, causing the prototypical algorithm to become an efficient solver in its final version.

Chapter 8 [100]. The idea for this part was a natural consequence of our PSPACE-inclusion proof. I formalized the initial translation to SMV, including bitwise operators, left shifts, and relational operators. Furthermore, I realized their first implementation in our translation tool. Gergerly extended this first version by addition, indexing, and right shifts. He also fixed some bugs and memory leaks in the orig-

inal implementation, and provided the parser, necessary to translate concrete formulas. The benchmarks were constructed by me, and the experiments were done by Gergely. The plots and main part of the writing were done by me.

Chapter 9 [152]. All benchmarks were the result of our previous theoretical work. While the PSPACE benchmarks were constructed by me and also used as parts of the experiments in Chapter 8, the instances representing correctness proofs for operator equivalences from Chapter 4 were formalized by Gergely.

Chapter 10 [98]. The original idea of applying an SLS solver for bit-vectors, directly on the theory level, was proposed by Christoph, who also did a first implementation already some years before our actual paper. I was then responsible for improving performance of the algorithm, e.g., by adopting techniques from SAT solvers, but also by improving data structures, which resulted in the practically efficient final version. All authors were involved in the discussion of specific ideas. The main part of the writing was done by me.

Further Related Contributions. Aside from the presented work, corresponding to Chapters 2–10, the author was involved in several other papers [29, 30, 181, 16, 13, 154]. While all of those ended up being published at a peer-reviewed venue, they are not part of this thesis and should not be considered the author's main scientific contribution. This is either because they are only partially related to the topic of this thesis (as for [29, 30, 16, 13, 154]), or because they will be part of the corresponding main author's thesis at some point (as for [181]). Note that, with bit-blasting still being the most common approach in solving bit-vector formulas, improving SAT solvers (as in [29, 30, 16, 13]) usually directly affects the performance of SMT solvers as well.

In [16], we revisited *blocked sets* for SAT. Blocked sets consist of clauses that, due to certain attributes, can be added or removed from formulas without affecting satisfiability. We formally proved several new theoretical properties and presented improvements to various practical algorithms that are related to blocked clauses. One specific improvement applies a re-encoding to existing formulas by making use of the circuit related structure that can be generated by the use of blocked clause decomposition. The resulting re-encoding in combination with the CDCL solver Lingeling [27] further improved state-of-the-art in SAT solving. In particular, we were able to solve 7 instances, that were not solved by any sequential solver in the previous SAT competition [12]. The topic of blocked sets recently also turned out to be very important in the context of DQBF [232]. This will be further discussed in Chapter 12.

In [29] and [30], we revisited CDCL *variable scoring schemes* and CDCL *restart schemes*, respectively. Both are widely considered to be essential ingredients for the strong performance of modern CDCL solvers. Nevertheless, it is not fully clear what exactly are the aspects which make certain schemes work so well. In both papers, our goal was to increase the overall understanding of the individual concepts by running and analyzing large sets of empirical evaluations, as well as

providing several theoretical models. We extracted key features of the individual heuristics, looked at the importance of careful implementation, and proposed new variants for variable scoring schemes as well as restart schemes. In both papers, we ended up improving the state-of-the-art in SAT solving by implementing the resulting techniques in Lingeling [27].

In [13], we showed how the performance of state-of-the-art SLS solvers for SAT can be improved in several different ways. In contrast to CDCL solvers, which have been heavily optimized regarding implementation, SLS solvers for SAT have seen less attention in this respect. We show that using implementation concepts derived from CDCL solvers also results in faster SLS solvers. Given the fact that the performance of SLS solvers heavily relies on the heuristic variable scoring, we also proposed a new scoring scheme particularly optimized for random k-SAT formulas with large k. Finally, we were the first to give a detailed analysis on the importance of heuristics also during clause selection in modern SLS solvers.

In [181], we revisited the SLS approach on the bit-vector level from Chapter 10 and reimplemented the original algorithm from Z3 [81] in Boolector [47, 180]. As a first part, we confirmed that the general concept of the algorithm is independent from the underlying solver framework, not only in theory but also in practice. We then extended the algorithm by *path propagation*, an approach consisting of propagation from top level constraints down to the individual atoms through the structural representation of the formula, showing that this can significantly increase the overall performance. As a second extension, we added the SLS approach as a possible prefilter to the Boolector engine, leveraging the strengths of both individual approaches.

In [154], we analyzed the complexity of a very broad range of problems, succinctly encoded as bit-vector formulas. In particular, we gave an *upgrading theorem*, proving that any problem which is complete for a complexity class C under quantifier-free reductions in its explicit representation, becomes complete for a multi-exponentially harder complexity class $\exp_\nu(C)$ under log-space reductions, if it is succinctly encoded by a bit-vector formula that uses multi-logarithmic encoded scalars. This is similar to related work for succinct encodings by circuits [167, 186, 225], Boolean formulas [226], and OBDDs [227]. To prove our result, we employed techniques from the field of *descriptive complexity theory* [131, 132], making it only partially related to the remaining complexity results of this thesis. Note that the motivation for the work in [154] originated from the practical problem of word-level model checking, but earlier results on the topic then were superseded by this more general theorem. Word-level model checking will be revisited in Chapter 12. Furthermore, note that this kind of *upgrading theorem* cannot be applied in the case of satisfiability, since no completeness result for an explicit representation is available. Nevertheless, it is possible to extend our previous results for satisfiability to multi-logarithmic bit-vector logics. This will be shown as a new contribution in Chapter 12.

Chapter 12

Beyond Previous Work

In this chapter, we will revisit some of the topics discussed in the previous parts of this thesis. All papers from Chapters 2–10 were peer-reviewed and published over the last four years, at the time of this writing. Further progress in research, that has been made between the publication of each paper and the time of this writing, allows to discuss certain earlier results under the light of new findings or related work by other authors. This opens up the possibility to provide a better understanding about certain results as well as a clearer picture of potential future work that can be done in several directions. Furthermore, we will present some extensions to the previous work, either by giving alternative proofs or by proposing some novel results that have not been published before.

12.1 Complexity of Quantified Bit-Vector Formulas

Regarding our complexity results for bit-vector logics, the question of whether BV is complete for any class, is still open. So far, we were only able to proof that BV is NEXPTIME-hard and that it can be solved in EXPSPACE. As a first intuition, it might seem unlikely that BV could be decidable in NEXPTIME, since this would imply that pure quantification (without uninterpreted functions) does not provide any additional expressiveness for bit-vectors with logarithmic encoded scalars. NEXPTIME-inclusion would be in sharp contrast to the unary case, where quantification actually does lift complexity from being NP-complete to PSPACE-complete. A more natural assumption might be to expect EXPSPACE-hardness of BV. This would be in analogy to the remaining logics, the quantifier-free case (with and without uninterpreted functions) and the quantified one which additionally includes uninterpreted functions. For all those cases, we can witness an exponential growth in complexity when switching from unary encoding to a logarithmic one, as shown in Chapter 2. In particular, complexity of QF_BV (and also of QF_UFBV) increases from NP to NEXPTIME, and complexity of QF_UFBV increases from NEXPTIME to 2-NEXPTIME. The same exponential growth in complexity takes place, e.g., for word-level model checking. While the unary case (i.e.,

bit-level model checking) is PSPACE-complete, the logarithmic one is EXPSPACE-complete, as proved in [154] (or later in this chapter).

Nevertheless, we want to give an intuition why this assumption should not necessarily be expected to hold. To prove PSPACE-hardness of QBF, one can use Savitch's Theorem [198]. Given the fact that a program only uses polynomial space, the total number of different states that can exist is bounded single-exponentially. The original idea in Savitch's Theorem, is to iteratively split the search in two parts of half length each. In particular, a final state can be reached in $2^{p(n)}$ steps from an initial state, if there exists a mid-point m that can be reached in $2^{p(n)}/2$ steps from the initial state and the final state can be reached from m in $2^{p(n)}/2$ steps as well. The mid-point is existentially quantified and the new bound of $2^{p(n)-1}$ is required to hold for both parts of the search, i.e., is universally quantified. After a polynomial number of iterations, the bound for the number of steps has been decreased to 1 and satisfiability can be checked by directly using the transition function. The resulting QBF, therefore, is of polynomial size as well. Note, however, that in each step a quantifier alternation is introduced and, therefore, also the number of quantifier alternations is polynomial.

If we now try to lift this proof in a naive way, we will see the following effect. Trying to prove EXPSPACE-hardness, we know that the total number of states is bounded double-exponentially, i.e., by $2^{2^{p(n)}}$ or, w.l.o.g., by 2^{2^n}. Iteratively splitting in two parts would, therefore, require at least 2^n steps. Even if we allowed splitting in more than two parts, since a universally quantified bit-vector of bit-width n can take 2^n different values, it would still take a super-polynomial many steps to decrease the bound to 1, because $(2^n)^{q(n)} \in o(2^{2^n})$ for any polynomial function q. Since we have to introduce new variables in the context of each quantifier alternation in each step, the resulting BV formula will also be of super-polynomial size.

Of course this only shows that the naive approach does not work, and does not prove that no polynomial reduction is actually possible. However, this construction points to a specific effect that seems to be inherent in the definition of quantified bit-vector formulas with logarithmic encoded scalars. While the large bit-widths due to the logarithmic encoding allow to put succinctly encoded information (compared to unary encodings) into a single bit-vector, there is no corresponding succinct version of quantifier alternations, because each alternation at least requires the introduction of two variables, independent from the bit-width. In this sense, the quantifier representation is less succinct than the actual encoding of bit-vector information. Consider, for example, an arbitrary BV formula with the restriction that it is not bit-width bounded. If we translate this formula into QBF by bit-blasting, the resulting QBF will have exponentially more variables than quantifier alternations, because each index of a bit-vector will become a variable, but the number of alternations will be the same as in the original formula. In other words, this implies that the number of quantifier alternations in the QBF will be logarithmic in the number of variables.

In QBF literature, there are several results for formulas with restricted num-

bers of quantifier alternations. In particular, it is known that k-QBF, with k being a constant number of quantifier alternations, is complete for the kth class of the polynomial hierarchy [215]. Note that all classes resulting from different k are assumed to be distinct from each other, and to be a true superset of NP and a true subset of PSPACE. Those results show that the number of quantifier alternations indeed directly affects the complexity of a formula class. Note that, compared to k-QBF, a logarithmic bound on the number of quantifier alternations is still more expressive than a constant bound for any k. Therefore, it is not clear whether this class of formulas is still a true subset of PSPACE, or whether it could be equally powerful. We are not aware of any results in literature considering this specific restriction to QBF, but think that this could be an interesting direction for future research. Regarding bit-vectors, it might be possible that the similar kind of restriction, inherent in BV, also reduces expressiveness to be less than EXPSPACE-hard.

12.2 Bit-Vector Problems in Practice

Similar to the statement for the complexity of SAT in Chapter 1, the related question for bit-vector formulas seems apparent: With QF_BV being NEXPTIME-complete, how is it actually possible that we solve this kind of formulas in practice? Considering the fact that state-of-the-art SMT solvers for bit-vector formulas usually rely on bit-blasting, even exponential space is required as, e.g., illustrated in Chapter 6 and Chapter 8. There are two main reasons why many practical bit-vector formulas, nevertheless, can be solved. First, SAT solvers are very efficient. As we showed in the previous chapters, even for our specifically crafted benchmarks, SMT solvers usually only failed for very large bit-widths. This is due to the fact that, although the bit-blasted formulas become very large, they are highly structured, since they correspond to circuit representations. CDCL solvers are known to perform very well on the task of exploiting many different kinds of structure within formulas. The second factor is given by the even more specific nature of application benchmarks that are used in practice. In contrast to our crafted benchmark sets, many application benchmarks are bounded in a certain nature. While we already analyzed some benchmark families with respect to bit-width boundedness in Chapter 2, there is a further aspect to this: Our concept of bit-width boundedness (as well as the extension to scalar boundedness in Chapter 4) is of theoretical nature, in the sense that it considers an infinite set of instances, analyzing whether the bit-width of all variables in this set can be bounded by a polynomial function of the formula size. In practice, however, even for a specific benchmark family with potentially infinitely many instances, we often do not consider the whole set of formulas, but just a finite subset. The finite subset can be defined, e.g., by assuming a constant bound k for the bit-width. This is often realistic because the bit-widths, in many cases, correspond to the size of some integer or word-level representation in software or hardware, e.g., 16, 32, or 64. There has been recent work on this topic, analyzing the complexity of two applications at Intel, in the context of bit-width

boundedness and beyond [177]. As proposed for future work in Chapter 2, the authors of [177] extend our previous work on the topic of bit-width boundedness by relating it to the field of parametrized complexity [85]. In particular, they show that the important application of *Clock Routing* is NP-complete and the application of *Microcode Validation* is para-NP-complete, from a parametrized complexity point of view. This provides a further link between theory and practice, as well as additional explanation on the strong performance of actual SMT solvers based on bit-blasting [177].

Nevertheless, we want to point out that scalar boundedness is not the only criterion for deciding whether bit-blasting techniques or some of our alternative approaches are more efficient. Consider, e.g., the conversion to SMV for the PSPACE fragment, based on the theory from Chapter 3, and practically applied in Chapter 8. Aside from our published work, certain instances from an industrial collaboration with Intel showed that, even for application instances with small bit-widths (< 16), the use of Model Checkers after translation to SMV was sometimes able to outperform state-of-the-art SMT solvers by orders of magnitude. In particular, we witnessed that Model Checkers were able to solve certain *Word-Level Symbolic Trajectory Evaluation* instances from [63], which could not be solved by common SMT solvers. Our approach was able to solve formulas for the target bit-width 32, while bit-blasting solvers already failed (within much larger time limits) for instances of bit-width 10, and with runtimes growing exponentially. While those instances are currently not publicly available, the reader should rather consider this as a statement of existence, keeping in mind that considering alternative solving approaches might actually be beneficial, not only on the crafted benchmark sets from Chapter 6 and Chapter 8, but also on application instances which have not been tested in this context yet. Finding more of those instances, and better understanding which is the criterion that makes Model Checkers perform badly or well, might be another interesting topic for future work. With Boolector [47, 180] being used for many industrial applications, e.g., by Intel [63, 91], implementing a dedicated version of our Model Checking approach as a component into Boolector is of interest as well.

12.3 Progress and Issues in DQBF Solving

Over the last years, the research topic of DQBF has seen growing interest. With SAT solvers having become increasingly efficient in solving the underlying NP-complete problem, QBF and DQBF, being prototypical PSPACE- and NEXPTIME-complete problems, seem more likely to be also feasible now. While practical QBF solvers have already emerged for over a decade, their performance is still way behind the one of SAT solvers. Nevertheless, the even more complex topic of DQBF solving seems tempting, since DQBF allows very natural representations for many problems. Since our first DQDPLL approach to DQBF solving, which was presented in Chapter 5, and which was not even able to solve very small instances,

a lot of progress has been made. However, evaluation of solver performances has faced some issues so far, mainly due to the lack of available benchmark sets. We want to give an overview about the developments in DQBF solving, specifically the most recent ones, which occurred after our latest related work from Chapter 7. At the same time, we also want to point out the issues with DQBF benchmarks.

The theoretical concept of an expansion-based solver was first proposed in [9, 8]. An actual implementation of a very simple expansion-based solver was then proposed in [110], but was not publicly accessible. The authors used this solver to deal with DQBF formulations of certain *partial equivalence checking* (PEC) benchmarks [199]. In [94], DQBF refutation is specifically addressed by proposing an overapproximation-based solver, only being able to handle unsatisfiable instances. In particular, the proposed solver iteratively produces QBF approximations of the original DQBF. This is done by only considering a certain number of paths for the universal variables in the assignment tree. If a formula can be refuted only by considering k different universal assignments, then the the original DQBF is obviously unsatisfiable as well. The value for k is increased iteratively. They evaluated their solver on PEC benchmarks constructed from circuit families proposed in [75, 110], outperforming a reimplementation of the simple expansion-based solver from [110]. Lacking alternatives, the same set of benchmarks from [94] was then used as part of our evaluation for IDQ, as presented in Chapter 7. As mentioned in Chapter 7, the benchmark set is biased towards unsatisfiable instances. Since QBF is a subset of DQBF, we suggested to additionally evaluate performance on QBF instances to increase the number of benchmarks, but also to being able to compare efficiency of general DQBF solvers to the one of dedicated QBF solvers, providing a base line.

The probably fastest DQBF solver at the present moment is HQS [111], which is also based on a reduction to QBF, but in a different way and not approximation-based, compared to [94]. Extending work from [110], a small number (often minimal) of variables is eliminated in such a way, that the resulting quantifier prefix is totally ordered [111]. All simplifications are applied on an AIG representation of the problem [111]. Once a QBF is obtained, the AIG-based QBF solver AIGsolve [191] is called. Aside from being more efficient, the process of not using approximations enables HQS to solve satisfiable instances as well. For their experiments, the benchmarks from [94] were extended by further PEC benchmarks created from the specifications in [199]. Unfortunately, benchmark selection is again restricted to PEC benchmarks. HQS outperforms IDQ by orders of magnitude on their benchmark set [111]. While this difference is very significant, part of this effect is most likely caused by the specific benchmark set. Dealing with black boxes in circuits, AIG representations have a significant advantage on PEC problems. Note that, in our previous experiments from Chapter 7, IDQ was frequently able to outperform state-of-the-art QBF solvers on several instances from a broader range of QBF benchmarks. While we do not want to claim that IDQ is more efficient in general, this argument helps to illustrate the importance of providing a broader range of benchmark sets, e.g., as part of a common library, such

as SAT-LIB [127], QBF-LIB [112], or SMT-LIB [18]. In the light of the fact that only few DQBF benchmarks exist at all, we would also like to encourage people to make benchmarks publicly available.

Recently, a practical way to obtain DQBF benchmarks was also proposed in [10], by offering a simple translation from SAT to DQBF, resulting in more succinct formulas, i.e., with only logarithmic many variables compared to the original SAT formula. Furthermore, a translation for reachability problems to DQBF was described. The authors argue that this kind of encoding has some preferable properties compared to existing ones [10]. With lots of SAT and reachability problems being available, both approaches could help to generate an interesting collection of DQBF benchmarks. However, one has to be aware of the fact that, e.g., in the case of re-encoding from SAT, the resulting representation is most likely not the most natural one that would be obtained by directly encoding the original problem into DQBF. Formulas generated in this way will automatically have a certain structural property that is inherently encoded due to the translation. This makes it an interesting addition for larger benchmark sets, but, in general, a more diverse set of families should be the overall goal.

Aside from full decision procedures, also preprocessing techniques for DQBF have started to see increased interest. We already showed how many simple techniques from SAT or QBF can be lifted to DQBF, in particular in the context of DPLL based solvers, in Chapter 5. A more recent investigation of preprocessing for DQBF is done in [232]. The authors address blocked clause elimination [137, 32], equivalence reasoning [113], structure extraction [191], and variable elimination by resolution [24]. Furthermore, it is proposed to use more complex, yet incomplete, algorithms, e.g., the approximation-based decision procedure from [94] with small k, as a "filter" that is run in advance to the full decision procedure [232]. As a benchmark set, the authors again use the PEC benchmarks from previous work on HQS [111], but also add benchmarks from controller synthesis [36]. Combining preprocessing and filtering, the authors show that this can increase the number of solved instances for HQS and iDQ, each, by more than a factor of 2. The gain is also significant considering the overall runtime, showing that the additional overhead of preprocessing and filtering more than pays off [232]. Note that the particular case of blocked clause elimination takes a slightly special role in their experiments. Being very important for iDQ, it can actually be harmful for HQS. This is due to the fact that HQS works on the circuit representation of a formula, which can be destroyed by BCE [232]. In contrast, iDQ can particularly profit from BCE since it is able to simulate some of the circuit simplification techniques possible on AIGs. In general, the AIG techniques still have an advantage on the specific benchmark sets and more general benchmarks are desired.

Considering the success of preprocessing for SAT [137] and QBF [32], the importance of preprocessing for DQBF is not surprising and the effect is even stronger with growing complexity of the underlying problem, e.g., with SAT, QBF, and DQBF being NP, PSPACE, and NEXPTIME-complete, respectively. As noted in [232], a complexity-wise more expensive problem allows to use also more ex-

pensive sub-routines in the solving process. For example, SAT solvers are very efficient and, therefore, each additional simplification step needs to be very fast as well—otherwise, the overall performance will decrease because the additional costs do not outweigh the gain (e.g., see [13]). In contrast, with a more complex base-problem, as it is the case for DQBF, sub-routines can be more expensive as well. For example, IDQ uses several full SAT solver calls, each time solving a problem that is theoretically NP-complete (see Chapter 7). Similarly, finding backbones of a formula is also NP-complete and could potentially be used as a simplification technique in DQBF [232]. As a result, using slightly more expensive preprocessing techniques will hardly contribute to the overall cost, but each simplification of the later solving process will possibly increase the gain even more.

In general, the current state-of-the-art suggests that a combination of strong simplification techniques, together with a reduction to a simpler problem, such as SAT (as done by IDQ [102]) or QBF (as done by HQS [111]), seems to be the most promising route for DQBF solving in the near future. This allows to profit most from the succinct structure contained in a DQBF encoding, but also from the efficiency of existing solvers, and the progress which has been made in related problems over the last decades. Note that this is similar to the case of bit-vector formulas, where state-of-the-art solvers usually rely on rewriting rules for preprocessing, in combination with bit-blasting and the use of SAT algorithms for actually solving the formula [47, 50, 104, 81, 87]. However, as discussed for bit-vectors in the previous chapters, we still think that alternative solving approaches should always be considered as well [100, 150, 98].

12.4 Improvements and Applications for SLS in SMT

In Chapter 10, we presented the first stochastic local search algorithm for SMT, which is directly operating on the theory representation of a formula. We lifted many techniques from SAT to the SMT level and showed that similarities between SAT and SMT can be exploited. Dealing with bit-vectors, we provided a score function and a neighbourhood relation that both respected certain properties of the underlying theory. Nevertheless, our approach still did not consider other structural properties of the formula, e.g., satisfaction of specific parts in the DAG representation. Due to the introduction of a weighting factor, penalizing unsatisfied constraints and the resulting increase in performance, we showed that putting focus on satisfying individual constraints can indeed be beneficial. This work was extended in [181], by introducing the concept of *path propagation*.

The authors proposed to add so-called propagation steps to our original SLS framework. In a propagation step, no regular SLS move according to the score function is performed but, instead, individual constraints in the SMT formula are "fixed", in the sense that a satisfying assignment for the constraint is constructed. For example, in the case of an equality, this can be done by assuming one side of the equality as constant and inverting the functions occurring on the other side.

Note that, in general, this is neither guaranteed to be possible, nor to be uniquely determined, but only under certain conditions. If propagation is not unique, non-deterministic choices can occur. If it is not possible to fix constraints in a single step, a two-step-approach can occur as well. If no propagation fix is possible in general, e.g., if the considered sub-problem is not just a simple constraint but NP-complete on its own, the algorithm from [181] falls back to regular SLS moves. In general, it is also able to interleave regular SLS moves with propagation moves, but the resulting algorithm usually performs worse than pure versions [181].

In Chapter 10, we also pointed out that state-of-the-art bit-vector solvers, based on bit-blasting, and our new SLS approach both perform particularly well on distinct instances. Therefore, we assumed that combinations of current SMT solvers with our SLS algorithm might be of particular interest. This has since been analyzed in [181], by running an SLS implementation (with and without path propagation) for a short time before calling a regular SMT solver. Indeed, the resulting combination turned out to solve significantly more instances compared to using the SMT solver on its own. The overall runtime decreased as well, even though the preceding SLS call produces an overhead whenever it does not find a solution [181].

This directly offers a possible direction for future work, e.g., by later using information gathered from the SLS solver, in the regular SMT solver. The experimental results from Chapter 10 and from the reported work in [181] also point towards the usefulness of our SLS approach in the context of parallelization. If a combination of our SLS approach with bit-blasting solvers works in a sequential setting, without reusing information, it will most likely be even more useful in a parallel one, since the SLS search is not restricted by a cutoff and the minimal runtime is not affected by an additional overhead. A similar approach is already used in SAT solving by the parallel solvers Plingeling and Treengeling, which optionally use the SLS solver YalSAT [27].

A further pointer towards parallelization can be found by the distinct performances of SLS implementations with propagation moves compared to those with regular moves on different sets of benchmark classes, as reported in [181]. This could be leveraged by parallel SLS solvers. Similarly, our original implementation in Z3 produces different results than the Boolector implementation on some benchmark families in [181]. It is not fully clear, whether this is due to different preprocessing steps in the corresponding SMT frameworks, or if this can be contributed to the different score function that is used. In the latter case, parallel SLS solvers could profit from this by purposely using distinct score functions in individual threads. In the context of SAT solving, this kind of approach has already been taken one step further, by combining the advantages of different SLS algorithms and additionally exchanging information leading towards better assignments [2]. In our opinion, this seems promising in the context of SMT as well.

12.5 Word-level Model Checking

In hardware and software verification, bit-vector logics are a natural framework for word-level system descriptions. For example, registers in digital circuits and variables in software can be represented by bit-vectors, and word-level operators, such as bitwise ones and arithmetic ones, can be applied to them. We now take a closer look at the problem of *word-level model checking*, a particular example of a bit-vector encoded problem that is of importance in practice. Word-level model checking was also used as the motivational example in our related work about succinct bit-vector encodings [154].

A strictly formal description of word-level model checking usually employs the concept of Kripke structures (see, e.g., [154]). In this section, we use the term word-level model checking, slightly informally, to denote the problem of checking whether there is a solution to a transition system (cf. Chapter 4, sequential circuits), with states consisting of a set of bit-vectors as used in actual computer system representations, and a transition relation consisting of all common bit-vector operators (cf. Chapter 4, common operators). This representation allows us to naturally encode design information captured at a higher level than that of individual wires and primitive gates and, therefore, is very practical for hardware and software verification [154]. Naturally, all scalars are encoded in a binary way.

In the past, there has been lots of research on bit-level model checking [70], as well as on bit-vector formula decision procedures [53, 169]. Comparatively few work has yet been published on word-level model checking. However, with increasing performance of state-of-the-art model checkers [41] and SMT solvers [47, 81], also the interest in word-level model checking is growing recently [33, 34].

While there are some practical approaches to solve word-level model checking [33, 34], we are not aware of any work that is dealing with the complexity of the underlying decision problem, apart from our work in [154]. The alternative approach in this section does not provide any new complexity results compared to the work from [154], but explains how we originally addressed our motivational problem, and emphasizes the natural and simplistic encoding of arbitrary Turing machines as word-level model checking problems. Furthermore, we point towards differences compared to the complexity gap for the satisfiability problem on the QF_BV$_{\ll 1}$ fragment from Chapter 3, and we present some constructive proofs by giving concrete encodings.

In the following, we consider two different classes of word-level model checking problems, differing in the operators that are allowed in the transition relation. Both proofs were also part of our original approach for [154], but then superseded by the more general results that we obtained.

12.5.1 Word-level Model Checking with all Common Operators

If we allow all common SMT-LIB operators, the following complexity result for word-level model checking (*WLM*) holds.

Theorem 12.1. *WLM is* EXPSPACE*-complete.*

Proof. We obtained this result as part of a general *upgrading theorem* in [154], using techniques from *descriptive complexity theory* [131, 132]. The theorem also follows directly from Lemma 12.2. □

With inclusion EXPSPACE-inclusion being trivial, we now want to give a constructive hardness-proof by showing how every language $L \in$ EXPSPACE can be polynomially translated into a word-level model checking problem.

Lemma 12.2. $\forall L \in$ EXPSPACE, $L \leqslant_p$ *WLM*

Proof. Since $L \in$ EXPSPACE, we know that there exists an exponential space deterministic Turing machine T that can decide L. The Turing machine has a fixed-size description and a tape which can contain $O(exp(n))$ symbols, with n denoting the size of the input. W.l.o.g., we assume that the tape is bounded to the right by the starting position of the Turing machine's head. W.l.o.g., we also assume that the alphabet of T only contains the symbols 1 and 0 (representing the blank symbol). We now convert this Turing machine into a word-level model checking problem as follows: Let $tape^{[N]}$ be a bit-vector representing the tape of T, with $N := \exp(n)$, using a binary representation to write down the bit-width N. Next, we introduce another bit-vector $head^{[m]}$ ($m := \lceil \log N \rceil$) containing the current position of the Turing machine's head. We can now formalize all possible operations of T by a sequential system over $tape^{[N]}$ and $head^{[m]}$.

- $head'^{[m]} := head^{[m]} + 1^{[m]}$ and $head'^{[m]} := head^{[m]} - 1^{[m]}$ define the movement of the head to the left and to the right, respectively.
- $tape'^{[N]} := (tape^{[N]} \& (\sim (1^{[N]} \ll head^{[m]}))) \mid (v^{[N]} \ll head^{[m]})$ writes the symbol $v \in \{1, 0\}$ to the current position of the head.
- The evaluation of the term $((tape^{[N]} \& (1^{[N]} \ll head^{[m]}) \neq 0^{[N]})$ returns the symbol $v \in \{1, 0\}$ at the current position of the head.

The fixed-size description of the Turing machine, including possible internal states and the transition relation, are defined separately in a fixed-size (combinatorial) circuit. Using word-level model checking, we can now decide whether an accepting state of T is reached for a given input with a maximum of N symbols on the tape in any step. To conclude the proof, we have to look at the size of the bit-vectors in our translation. We know that the length of the tape of T (and, therefore, the number of bits of $tape^{[N]}$) is bounded by $O(exp(n))$. By using a binary representation for the bit-width of all bit-vectors, we can write down $tape^{[N]}$ (and all other bit-vectors) with only $p(n)$ bits for some polynomial function p. This proves that the given reduction is polynomial. □

12.5.2 Word-level Model Checking for QF_BV$_{\ll 1}$

In hardware verification, we often encounter restricted classes of bit-vector problems. One specific example is given by the class QF_BV$_{\ll 1}$. This is the class of

bit-vector formulas which only contain *bitwise operators*, *equality*, and *shift by one*. In Chapter 3 (and Chapter 4), we argued that the decision problem for formulas in QF_BV$_{\ll 1}$ is PSPACE-complete. If, instead, shifts by arbitrary constants are allowed, we get the class QF_BV$_{\ll c}$, which is already NEXPTIME-complete, i.e. as expressive as general QF_BV with all common bit-vector operators. In Chapter 8, we then gave a practical translation from QF_BV$_{\ll 1}$ to SMV and used model checkers to solve this kind of formulas more efficiently compared to state-of-the-art SMT solvers.

This gives rise to the question of whether a similar result can be obtained for word-level model checking, i.e., whether the complexity of word-level model checking for QF_BV$_{\ll 1}$ (*WLM*$_{\ll 1}$) is lower than for general QF_BV. This kind of result might lead to more efficient algorithms, similar to our work on satisfiability of QF_BV$_{\ll 1}$, in Chapter 8. However, this is not the case. Instead, the following theorem holds.

Theorem 12.3. *WLM*$_{\ll 1}$ *is* EXPSPACE*-complete.*

Proof. As for *WLM*, this is also obtained as part of the *upgrading theorem* in [154]. The theorem also follows directly from Lemma 12.4. □

We give a constructive proof to show how *WLM* can be reduced to *WLM*$_{\ll 1}$. (Note that the same kind of reduction could also be used to further eliminate $\ll 1$ and replace it by $+_1$, in order to obtain the same set of operators that was used in [154].)

Lemma 12.4. *WLM can be reduced to WLM*$_{\ll 1}$.

Proof. Since we know that all operators in QF_BV can be polynomially expressed in QF_BV$_{\ll c}$ [101], it is sufficient to show that *shift by constant*, in a sequential system, can be expressed by *shift by one*.

W.l.o.g., we assume that each shift expression in the given sequential system is of the form $y^{[2^n]} = x^{[2^n]} \ll c^{[n]}$, with input variables $y^{[2^n]}$ and $x^{[2^n]}$, as well as a constant $c^{[n]}$. Any sequential system can be translated to this kind of format with only polynomial growth in size. This can be done as follows. Each shift $term^{[2^n]} \ll c^{[n]}$, potentially even being part of a nested expression, can be replaced by a new Tseitin variable variable ${ts_1}^{[2^n]}$, defined non-deterministically by a new input. In a second step, $term^{[2^n]}$ is also replaced by a new input variable ${ts_2}^{[2^n]}$. Additionally, constraints for ${ts_2}^{[2^n]} = term^{[2^n]}$ and ${ts_1}^{[2^n]} = {ts_2}^{[2^n]} \ll c^{[n]}$ are added to the specification of the formula.

We now remove all occurrences of *shift by constant*. Assume we encounter the expression ${y_i}^{[2^{n_i}]} = {x_i}^{[2^{n_i}]} \ll {c_i}^{[n_i]}$ (with i enumerating all shift expressions in the formula). First, we introduce two additional registers. We use ${temp_i}^{[2^{n_i}]}$ to save a temporary value and ${count_i}^{[n_i]}$ to represent a counter. We now initialize ${temp_i}^{[2^{n_i}]}$ with ${x_i}^{[2^{n_i}]}$, and ${count_i}^{[n_i]}$ with ${c_i}^{[n_i]}$, respectively. In each cycle of the system, we then check the value of the counter. While ${count_i}^{[n_i]} \neq 0^{[n_i]}$, we set ${count'_i}^{[n_i]} := {count_i}^{[n_i]} - 1^{[n_i]}$ and ${temp'_i}^{[2^{n_i}]} := {temp_i}^{[2^{n_i}]} \ll 1^{[n_i]}$. Once the counter reaches zero, the values of the registers are not changed anymore. Aside

from this, we add a constraint to the formula, specifying that ${y_i}^{[2^{n_i}]} = {temp_i}^{[2^{n_i}]}$. We therefore know that ${y_i}^{[2^{n_i}]}$ will have the value of ${x_i}^{[2^{n_i}]} \ll {c_i}^{[n_i]}$ after c_i cycles. This is done for all shift expressions separately.

Finally, we need to *delay* all other registers in the sequential circuit (i.e., all registers apart from the newly introduced ${temp_i}^{[2^{n_i}]}, {count_i}^{[2^{n_i}]}$), expressing that the results of all shifts are only available after $\max_i\{c_i\}$ cycles. This means that all other registers should only be updated whenever *all* counters (i.e., the counter corresponding to the largest c_i) have reached zero. Otherwise, the value from the previous cycle is simply preserved. At the same time, all ${temp_i}^{[2^{n_i}]}$ and ${count_i}^{[2^{n_i}]}$ registers are reinitialized in that step. □

12.6 Upgrading Satisfiability

In [154], we gave a generic *upgrading theorem*, lifting complexity results for explicit problem representations, to complexity results for succinct problem encodings as bit-vector logics. Similar work has been done for succinct encodings by circuits [167, 186, 225], Boolean formulas [226], and OBDDs [227]. In [154], it was shown that, under certain restrictions, complete problems for a standard complexity class will become complete for a (multi-)exponentially harder complexity class when succinctly encoded by a bit-vector formula. The order of growth in complexity, thereby, depends on the degree of succinctness used for the encoding of scalars in the underlying bit-vector logic. Since the full details of the used approach are out of the scope of this section, the reader is referred to [154] for further details. In particular, this kind of succinctness of scalar representation was addressed by the concept of *multi-logarithmic* bit-vector encodings.

In this section, we will prove a related result for satisfiability of bit-vector formulas with multi-logarithmic encoded scalars. In contrast to [154], we will refer to a ν-logarithmic encoding if we consider a bit-vector formula with scalars (in particular bit-widths) $2^{2^{\cdots 2^c}}$ being encoded as c in binary form, where the degree of exponentiation is $\nu - 1$, for $\nu > 0$. This denotes an extension of our previous defintions, considering QF_BV with binary encoded scalars ($\nu = 1$) to be 1-logarithmic (single-logarithmic). For $\nu > 1$, we use the term multi-logarithmic. This notation differs from the one in [154], in the sense that this was considered to be a $(\nu{+}1)$-logarithmic encoding, e.g., calling QF_BV with binary encoded scalars 2-logarithmic. Basically, the two notations only differ in being "off-by-one". This is attributed to the fact that, in [154], succinct representations of certain explicit problems were considered. In that context, even the propositional case, which is equally powerful to bit-vectors with unary encoded scalars, can already be used for succinct representations of problems, and results in a (single-)logarithmic size compared to the explicit representation (thus, being called 1-logarithmic in [154]). However, this would not be consistent with our use of "logarithmic" throughout this thesis. Furthermore, we will use the terms "ν-logarithmic encoded formula/problem" and "ν-logarithmic encoded scalars" interchangeably.

As already pointed out in [154], a ν-logarithmic encoding can be found in practice, e.g., in the SMT-LIB. To declare arrays in SMT-LIB format, the expression `(Array idx elem)` is used, where idx is the sort for array indexes, and $elem$ is the sort for array elements. If idx is a bit-vector sort `(_ BitVec n)`, where n is encoded in binary form, the size of the array is double-exponential in the length of the binary encoding of n [154].

12.6.1 Satisfiability of $\mathcal{BV}_\nu^{\ll}$

Symbolic encodings of decision problems by Boolean formalisms are well-known to increase the problem complexity [11, 38, 77, 93, 103, 117, 167, 186, 225, 226, 228, 230]. Examples of such problems, including those from [154], can be found in [210, 211, 212, 213, 214]. However, the generic *upgrading theorem* from [154] only applies to problems that are complete for a complexity class when using an explicit encoding. While we know that the satisfiability problem is NP-complete for bit-vector formulas with unary encoded scalars (we slightly abuse notation by writing $\nu = 0$) and NEXPTIME-complete with binary encoded scalars ($\nu = 1$), as discussed in Chapters 2-4, no general result for multi-logarithmic encodings of the scalars ($\nu > 1$) was implied so far.

Nevertheless, a similar result can indeed be obtained for ν-logarithmic QF_BV, with $\nu > 1$, and an operator set consisting of bitwise operators, equality, and shifts. We would like to acknowledge Helmut Veith for initially conjecturing that this kind of theorem could probably be shown for satisfiability of multi-logarithmic encoded bit-vector logics. In the following, we will use a slightly different notation to the previous chapters of this thesis. Similar to the notation in [154], we use $\mathcal{BV}_\nu^\Omega$ to denote a (quantifier-free) bit-vector logic with ν-logarithmic encoding (unary encoding, for $\nu = 0$) and an operator set Ω. Furthermore, similar to the notation in Chapter 3 and Chapter 4, we define $bw := \{\&, |, \sim, =\}$, and, as a general extension for a specific operator o, we define the logics $\mathcal{BV}_\nu^o := \mathcal{BV}_\nu^{bw \cup \{o\}}$. As we require $\Omega \supseteq \{\&, |, \sim, =, \ll\}$, we first consider $\mathcal{BV}_\nu^{\ll}$.

In the following, we slightly abuse notation in regard to the symbols that we use for bitwise operators. In contrast to most previous chapters, the operators $\wedge$, $\vee$, and $\neg$ will not represent the corresponding Boolean operators, but denote *bitwise and* ($\&$), *bitwise or* ($|$), and *bitwise negation* ($\sim$), respectively. We simply do this since we have realized that the slightly abusive notation significantly improves the intuitive readability of our equations. Similarly, $\leftrightarrow$ and $\rightarrow$ will denote *bitwise equivalence* and *bitwise implication*, respectively. Although $\leftrightarrow$ and $\rightarrow$ are not included in Ω, they could simply be replaced by combinations of $\wedge$, $\vee$, and $\neg$. Furthermore, $\ll$ is assumed to take precedence over the bitwise operators whenever parenthesis are omitted. As in the previous chapters, e.g., in Chapter 4, bit-vectors (of bit-width n) are assumed to start with the largest bit-index $n-1$ on the left-hand side, and end with the smallest bit-index 0 on the right-hand side.

Furthermore, we use the notation $\exp_\nu$ in two different ways: For a number n, the expression $\exp_\nu(n)$ donates the repeated exponentiation function, with

$\exp_0(n) := n$, and $\exp_{i+1}(n) := 2^{\exp_i(n)}$, for $i \in \mathbb{N}_0$. The second usage is in the context of complexity classes. For a complexity class C, the expression $\exp_\nu(C), \nu > 0$ donates the corresponding ν-exponentially harder complexity class, e.g., $\exp_1(\text{NP}) = \text{NEXPTIME}$, $\exp_2(\text{NP}) = \text{2-NEXPTIME}$, etc. This follows the notation from [154]. For a formal definition of this concept, it is possible to use the concept of leaf languages [225, 38]. We define $\exp_0(C) := C$.

Similar to the results from [154], the following theorem holds:

Theorem 12.5. *Satisfiability of $\mathcal{BV}_\nu^{\ll}$, $\nu \in \mathbb{N}$, is $\exp_\nu(\text{NP})$-complete.*

Proof. With $\nu = 0$ being the propositional case, we only have to consider $\nu > 0$, and show that satisfiability is ν-NEXPTIME-complete for those classes.[1] Membership follows directly from applying bit-blasting. Hardness is a consequence of Lemma 12.6 or, alternatively, of Lemma 12.8. □

12.6.2 Encoding of Turing Machines

A direct way of proving hardness of the given set of bit-vector logics $\mathcal{BV}_\nu^{\ll}$ is obtained by showing how corresponding Turing machines can be encoded. This approach was also the one taken by Cook, when showing NP-hardness of SAT [73].

Lemma 12.6. *Any ν-NEXPTIME Turing machine can be polynomially reduced to a $\mathcal{BV}_\nu^{\ll}$ formula.*

Proof. For a given non-deterministic Turing machine M which solves a problem in ν-NEXPTIME, we construct a $\mathcal{BV}_\nu^{\ll}$ formula that is satisfiable if and only if M accepts a given input I. The construction is similar to the one for proving the Cook-Levin theorem [73] as used in [106].

Let $M = (Q, \Sigma, s, F, \delta)$ be a Turing machine, where Q is the set of states, Σ is the alphabet of tape symbols, $s \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta \subseteq ((Q \backslash F) \times \Sigma) \times (Q \times \Sigma \times \{-1, +1\})$ is the transition relation. Furthermore, suppose that M accepts or rejects an instance of the problem in time $\exp_\nu(n)$, where n is the size of the input I. With the number of steps being bounded by $steps := \exp_\nu(n)$, we also know that it is sufficient to consider a tape length of $size := 2 \cdot \exp_\nu(n) + 1$, if we assume the initial head position 0 to be in the middle of the tape, at offset $mid := \exp_\nu(n)$.

The $\mathcal{BV}_\nu^{\ll}$ formula uses the variables $H^{[N]}$, ${T_\sigma}^{[N]}$, and ${Q_q}^{[N]}$, with bit-width $N := size \cdot steps = (2 \cdot \exp_\nu(n) + 1) \cdot \exp_\nu(n)$, $\sigma \in \Sigma$ and $q \in Q$. First, we need to check that we can indeed write down N polynomially in the input size n. According to the definition of ν-logarithmic encoding, we have scalars $\exp_{\nu-1}(c)$, being encoded as c in binary form. With c in binary form, c itself can be exponential in the size of the input, i.e., $c = 2^n$, and n bits are sufficient to specify c. Thus, we can indeed encode $N = p(\exp_\nu(n)) = p(\exp_{\nu-1}(2^n)) \leqslant \exp_{\nu-1}(2^{p(n)})$ by using $p(n)$ bits, i.e., polynomial in the size of the input I.

[1] The case of $\nu = 1$ corresponds to our previous results from Chapters 2-4. Nevertheless, it is also included as part of this more general theorem.

For easier readability, we store the original bound for the tape size and the offset for the midpoint of the tape (position 0) in bit-vectors $size^{[N]} = 2 \cdot \exp_\nu(n) + 1$ and $mid^{[N]} = \exp_\nu(n)$. Note that evaluation of $\exp_\nu(n)$ is polynomial when implemented by using $(1^{[N]} \ll (\dots(1^{[N]} \ll n^{[N]})\dots)$, with the number of shifts being equal to ν.

Intuitively, $H^{[N]}$, ${T_\sigma}^{[N]}$, and ${Q_q}^{[N]}$ represent the head position, the tape content, and the state of M in each single step during its computation, respectively. In particular, the variables are uniquely defined by the following criteria:

1. The $(j \cdot size + mid + i)$th bit of $H^{[N]}$ is 1 if and only if the head of M is at position i at step j of the computation.
2. The $(j \cdot size + mid + i)$th bit of ${T_\sigma}^{[N]}$ is 1 if and only if tape cell i of M contains symbol σ at step j of the computation.
3. The $(j \cdot size + mid + i)$th bit of ${Q_q}^{[N]}$ is 1 if and only if the head of M is at position i at step j of the computation and M is in state q at step j of the computation.

Here, $-\exp_\nu(n) \leqslant i \leqslant \exp_\nu(n)$ and $0 \leqslant j < \exp_\nu(n)$.

Furthermore, we define some particular bit-vectors constants that we will use for masking specific bits of our variables.

1. $hi^{[N]} = \neg 0^{[N]} \ll size^{[N]}$
2. $lo^{[N]} = \neg(\neg 0^{[N]} \ll size^{[N]})$

The latter mask allows us to consider only the initial state of the computation, the first one considers all remaining steps. We now construct the formula as a conjunction of the following constraints:

1. The tape initially contains the input $I = \sigma_{n-1} \dots \sigma_0$. We add n constraints, one for each symbol σ_k, $k \in \{n-1, \dots, 0\}$ in I:

   $$ {T_{\sigma_k}}^{[N]} \wedge (1^{[N]} \ll mid^{[N]} \ll k^{[N]}) \neq 0^{[N]} $$

   All other cells should contain the blank symbol ▫ in the first step of the computation. We ensure this by adding one more constraint:

   $$ {T_\square}^{[N]} \vee hi^{[N]} \vee (2^n - 1)^{[N]} \ll mid^{[N]} = \neg 0^{[N]} $$

   Note that we can write down the constant $2^n - 1$ by using n bits, evaluating the bit-vector term $((1^{[N]} \ll n^{[N]}) - 1^{[N]})$.

2. The head initially is at position 0:

   $$ H^{[N]} \wedge lo^{[N]} = 1^{[N]} \ll mid^{[N]} $$

3. M initially is in state s:

$$Q_s{}^{[N]} \wedge lo^{[N]} = 1^{[N]} \ll mid^{[N]}$$

4. In each computation step, there is at most one symbol per tape cell, i.e., $\forall \sigma, \sigma' \in \Sigma$, with $\sigma \neq \sigma'$, we add:

$$\neg T_\sigma{}^{[N]} \vee \neg T_{\sigma'}{}^{[N]} = \neg 0^{[N]}$$

5. In each computation step, there is at least one symbol per tape cell:

$$\bigvee_{\sigma \in \Sigma} T_\sigma{}^{[N]} = \neg 0^{[N]}$$

6. In each computation step, there is at most one state at a time, i.e., $\forall q, q' \in Q$, with $q \neq q'$, we add:

$$\neg Q_q{}^{[N]} \vee \neg Q_{q'}{}^{[N]} = \neg 0^{[N]}$$

7. The bits of the state variables can only be set at the head positions:

$$\bigvee_{q \in Q} Q_q{}^{[N]} \wedge \neg H^{[N]} = 0^{[N]}$$

8. The tape does not change at positions different from those of the head, i.e., $\forall \sigma \in \Sigma$, we add:

$$(T_\sigma{}^{[N]} \ll size^{[N]} \leftrightarrow T_\sigma{}^{[N]}) \vee (H^{[N]} \ll size^{[N]}) \vee lo^{[N]} = \neg 0^{[N]}$$

9. The transition relation, i.e., $\forall q \in Q, \sigma \in \Sigma$, we add:

$$(H^{[N]} \wedge Q_q{}^{[N]} \wedge T_\sigma{}^{[N]}) \ll size^{[N]} \rightarrow$$

$$\bigvee_{(q,\sigma,q',\sigma',d) \in \delta} (H^{[N]} \circ_d 1^{[N]} \wedge Q_{q'}{}^{[N]} \wedge T_{\sigma'}{}^{[N]}) = \neg 0^{[N]}$$

As in the definition of δ, $d \in \{-1, +1\}$. We define $\circ_{-1} = \ll$ and $\circ_{+1} = \gg_{\mathbf{u}}$. The $\gg_{\mathbf{u}}$ operator can then be replaced by $\ll$, as shown in Chapter 4, Theorem 4.32.

10. M must reach a final state at one point:

$$\bigvee_{q \in F} Q_q{}^{[N]} \wedge H^{[N]} \neq 0^{[N]}$$

All constraints so far were very similar to those used for the translation in [106].

Adding the special condition for states being only defined at the head positions is slightly technical and not strictly necessary, but allows to have a unique state vector as stated in the definition of $Q^{[N]}$. A small redundancy is given by the fifth constraint, requiring a symbol to be on each position of the tape in each computation step. Actually, this implicitly follows from the initial constraints, in combination with the encoding of the transition relation (assuming it is well-defined for M) and the definition of head movement in the next paragraph.

The main difference between our reduction and the propositional version [106] is found in the definition of the head movement. To specify the fact that the head can only be at one position in each computation step, the proof in [106] uses constraints similar to those for the tape and the states. However, this is only possible if the number of different head positions and, therefore, also the number of needed constraints, is polynomial. This is true for the propositional case in their approach, but since we allow $2 \cdot \exp_\nu(n) + 1$ different head positions, we need a different encoding. In our approach, unique head positions and head movement can be realized by adding the following three constraints:

1. If the head is in a certain position in a computation step, it cannot be at any position other than left or right of the current one in the next step:

$$lo^{[N]} \vee \neg H^{[N]} \vee H^{[N]} \ll (size + 1)^{[N]} \vee H^{[N]} \ll (size - 1)^{[N]} = \neg 0^{[N]}$$

2. If the head is in a certain position in a computation step, it has to be at position left or right (non-exclusive) of the current one in the next step:

$$\neg(H^{[N]} \ll size^{[N]}) \vee H^{[N]} \ll 1^{[N]} \vee H^{[N]} \gg_{\mathbf{u}} 1^{[N]} = \neg 0^{[N]}$$

3. In any computation step, the head will never be at two distinct positions exactly two indices apart from each other:

$$H^{[N]} \ll 1^{[N]} \wedge H^{[N]} \gg_{\mathbf{u}} 1^{[N]} = 0^{[N]}$$

Assume that the head position in a certain computation step is given by a vector $h = (0, \ldots, 0, 0, 1, 0, 0, \ldots, 0)$ of bit-width $size$, and let i be the unique index with $h[i] = 1$. The new head position is the next consecutive computation step is given by the a vector $h' = (h'[size - 1], \ldots, h'[0])$. We need to formalize that the head will move left or right (exclusively). Condition 1 states that $h'[j] = 0, \forall j \notin \{i + 1, i-1\}$, i.e., $h' = (0, \ldots, 0, h'[i+1], 0, h'[i-1], 0, \ldots, 0)$. Condition 2 guarantees that $h'[i + 1] + h'[i - 1] \geqslant 1$. Condition 3 ensures that $h'[i + 1] + h'[i - 1] \leqslant 1$.

Note that the given encoding is again slightly redundant. Due to the fact that there is always exactly one state in each computation step and the state vectors Q can only be different from 0 at the bits corresponding to the head positions, we actually would not need a separate variable H for the head of M, but could just replace all occurrences of $H^{[N]}$ by $\bigvee_{q \in Q} {Q_q}^{[N]}$. Nevertheless, we chose to keep a

separate variable for the head in order to stick closer to the version for propositional logic [106] and to improve the readability of the proof.

□

Remark 12.7. An easier way to guarantee the unique position for the head can be obtained by using certain bit operations, similar to those in some benchmarks from Chapter 8 and Chapter 9, but might be less intuitive for readers that are not familiar with the topic of low level "bit hacks" as, e.g., found in [231]. For some of the benchmarks in our earlier chapters, we used "is a power of two"-constraints. Those constraints were generated using the following kind of condition:

$$h^{[n]} \& (h^{[n]} - 1^{[n]}) = 0^{[n]}$$

It is straightforward to check that this kind of assertion is satisfied if and only if h is a power of two, i.e., if exactly one bit of h is set to 1. This can be extended to generate bit-vectors with exactly one bit being set to 1 in a certain range of indices and, thus, allows us to formalize the fact that the head is at exactly one position in every computation step.

Consider the bitmask $br^{[N]} = \overbrace{\underbrace{00\ldots01}_{size}\underbrace{00\ldots01}_{size}\ldots\underbrace{00\ldots01}_{size}\underbrace{00\ldots01}_{size}}^{N}$.

We can uniquely define this $br^{[N]}$ by adding the following two constraints:

1. $br^{[N]} \wedge lo^{[N]} = 1^{[N]}$
2. $br^{[N]} \ll size^{[N]} = br^{[N]} \wedge hi^{[N]}$

Bit-vectors $lo^{[N]}$, $hi^{[N]}$, and $size^{[N]}$ are defined according to the previous lemma. By using the bitmask $br^{[N]}$, the assertion

$$H^{[N]} \& (H^{[N]} - br^{[N]}) = 0^{[N]}$$

will result in a natural extension of the previous constraint, forcing exactly one bit of H to be set to 1 in each interval of bit-indices $H[(j+1) \cdot size - 1 : j \cdot size]$, with $0 \leqslant j < steps$.

12.6.3 Encoding of Domino Tiling Problems

While a direct reduction from Turing machines to bit-vector logics is an interesting construction by itself, an easier way to prove the given complexity result can be obtained by using a reduction from a set of domino tiling problems [65, 171, 183], as used in Chapter 2 and Chapter 4. We would like to acknowledge Moshe Vardi for initially suggesting that a simpler proof probably might be derived by considering domino tiling problems—it turned out he was right.

Lemma 12.8. *The* $\exp_\nu(n)$*-square tiling problem can be polynomially reduced to a* $\mathcal{BV}_\nu^{\ll}$ *formula.*

Proof. Let $D = (T = [0, \ldots, k-1], H, V, n)$ be a $\exp_\nu(n)$ square domino system as defined in Chapter 2 and Chapter 4. We use i and j, to denote the horizontal and vertical indices, respectively. For each $t \in T$, we introduce a bit-vector ${T_t}^{[N]}$, with $N := \exp_\nu(n)^2$, defining all positions that contain tile t. The $(j \cdot \exp_\nu(n) + i)$th bit of T_t is 1 if and only if the ith column in row j of D contains tile t. As already argued in Lemma 12.6, writing down N is polynomial in n when using a ν-logarithmic encoding. Note that, for $N = \exp_\nu(n)^2$, it is easy to see that even much tighter bounds can be given, with $N = \exp_\nu(n)^2 = \exp_{\nu-1}(2^n)^2 \leqslant \exp_{\nu-1}((2^n)^2) = \exp_{\nu-1}(2^{2n})$, only requiring $2n$ bits. This effect further increases for larger ν. For $\nu > 1$, $N = \exp_\nu(n)^2 \leqslant \exp_{\nu-1}(2^{n+1})$ can already be specified by $n + 1$ bits.[2] Nevertheless, tighter bounds do not affect our complexity results.

We now construct a $\mathcal{BV}_\nu^{\ll}$ formula that is satisfiable if and only if there exists a tiling for D, starting with tile 0 and ending with tile $k - 1$, which respects the horizontal and vertical matching conditions H and V.

The formula is a conjunction of the following constraints:

1. D starts with tile 0:
$$ {T_0}^{[N]} \wedge 1^{[N]} \neq 0^{[N]} $$

2. D ends with tile $k - 1$:
$$ {T_{k-1}}^{[N]} \wedge \neg((\neg 0^{[N]}) \gg_{\mathbf{u}} 1^{[N]}) \neq 0^{[N]} $$

3. Each position in D contains exactly one tile, i.e., $\forall t, t' \in T$:
$$ \neg {T_t}^{[N]} \vee \neg {T_{t'}}^{[N]} = \neg 0^{[N]} $$

4. The vertical matching conditions hold, i.e., $\forall t \in T$:
$$ ({T_t}^{[N]} \ll \exp_\nu(n)^{[N]} \rightarrow \bigvee_{(t,t') \in V} {T_{t'}}^{[N]}) = \neg 0^{[N]} $$

5. The horizontal matching conditions hold, i.e., $\forall t \in T$:
$$ ({T_t}^{[N]} \ll 1^{[N]} \rightarrow \bigvee_{(t,t') \in H} {T_{t'}}^{[N]} \vee br^{[N]}) = \neg 0^{[N]} $$

The mask $br^{[N]}$ is used to exclude neighbouring indices from the horizontal matching conditions that actually represent column $\exp_\nu(n) - 1$ of row j, and column 0 of row $j + 1$, i.e., are not horizontally neighbouring tiles in the domino system. It can be generated by adding the following constraints:

1. $hi^{[N]} = \neg 0^{[N]} \ll \exp_\nu(n)^{[N]}$

[2] Similar bounds hold for Lemma 12.6, but have to be constructed more carefully.

2. $lo^{[N]} = \neg(\neg 0^{[N]} \ll \exp_\nu(n)^{[N]})$
3. $br^{[N]} \wedge lo^{[N]} = 1^{[N]}$
4. $br^{[N]} \ll \exp_\nu(n)^{[N]} = br^{[N]} \wedge hi^{[N]}$

□

12.6.4 Remarks on the Expressiveness of $\ll_c$ and $\ll$

Note that we defined $\Omega \supseteq \{\&, |, \sim, =, \ll\}$ at the beginning of this section. In particular, we required $\ll \in \Omega$, whereas we only used $\ll_c$, i.e., *shift by constant*, for the complexity results in Chapters 2–4. With hindsight, using $\ll$, i.e., general shift, might have been the better choice. As argued in Section 4.2, general shift can be realized by a so-called barrel shifter, using shift by constant. One application thereof is also used for our translation to EPR, in Section 6.3.1. In general, this can be done by replacing expressions $x = y \ll z$ with the following set of equations:

$$\begin{aligned} x_0 &= y, \\ x_i &= \mathit{ite}(z \,\&\, 2^{i-1} = 0, x_{i-1}, x_{i-1} \ll 2^{i-1}), \quad i \in \{1, \ldots, k\}, \\ x &= \mathit{ite}(z \,\&\, (\sim 0 \ll 2^k) = 0, x_k, 0), \end{aligned}$$

for bit-vector variables and constants of bit-width N, and with $k := \lceil \log_2 N \rceil$. Since all constants 2^i are encoded as bit-vectors, each can be written down using $i \leqslant k$ bits. Thus, the total size of a barrel shifter is polynomial in k, i.e., logarithmic in N. As a consequence, using a single-logarithmic encoding for the bit-width N (e.g., a binary one, as in Chapters 2–4) implies that the total size of a barrel shifter is polynomial in the size of the input formula. Therefore, shift by constant and general shift are equally powerful for single-logarithmic encoded scalars.

The situation is different for the multi-logarithmic case. With N being super-exponential in the size of the input formula, k will be super-polynomial, and, thus, the size of a barrel shifter will also be super-polynomial. Note that general shift was used for proving Lemma 12.6 and Lemma 12.8, when evaluating $\exp_\nu(n)$, by using $(1^{[N]} \ll (\ldots (1^{[N]} \ll n^{[N]}) \ldots)$, a chain of ν shifts.

While all of the previous work remains correct, we would like to propose two alternative characterizations, which allow a more uniform treatment of the single-logarithmic and the multi-logarithmic case.

Multi-Logarithmic Constants. One unification approach could be obtained by redefining the characterization of constants in bit-vector formulas. So far, we considered constants to be represented as bit-vectors, as it is the case in the SMT-LIB format [18]. However, constants could alternatively be considered as scalars. According to the definition of scalars in Chapter 4, this would allow multi-logarithmic encodings of constants as well. For example, one could use an extension of floating

point representations, such as $1\mathrm{d}n := 1 \cdot 2^n, 1\mathrm{dd}n := 1 \cdot 2^{2^n}, \ldots$, i.e., $1\mathrm{d}^\nu n := exp_\nu(n)$, for the constants in Lemma 12.6 and Lemma 12.8.[3] Unfortunately, a different representation of constants could affect constant-related arguments in some other proofs, e.g., in Lemma 4.26, Lemma 4.29, and Theorem 12.9. Aside from this, it is not consistent with common constant representations, e.g., in the SMT-LIB format. We therefore recommend the second approach.

General Shifts. The second unification approach could be obtained by redefining the bit-vector classes QF_BV$_{\ll c}$ and QF_BV$_{\ll 1}$, introduced in Chapter 3 and Chapter 4. For example, one could use $\mathcal{BV}_1^{\ll}$ instead of QF_BV$_{\ll c}$ and $\mathcal{BV}_1^{\ll_1}$ instead of QF_BV$_{\ll 1}$, with operator sets $\Omega_{\ll} = \{\&, |, \sim, =, \ll\}$, and $\Omega_{\ll_1} = \{\&, |, \sim, =, \ll_1\}$, respectively. To further distinguish between $\ll$ and $\ll_1$, we would propose to define $\ll_1$ as a separate unary operator, similar to the definition of $+_1$ in [154]. As it will have no effect on any existing proof, and since it can be expressed in the SMT-LIB format, we recommend this approach.

12.6.5 Satisfiability of $\mathcal{BV}_1^{+_1}$

As pointed out in the previous two sections, there is a gap between the operators required for hardness results of satisfiability, and the ones required for general problems that are complete under explicit representations, e.g., model checking. As shown in [154], word level model checking is already EXPSPACE-hard for an operator set that contains increment ($+_1$) instead of general shift ($\ll$). In Lemma 12.4 of this thesis, we then showed *how* the $\ll_c$-operator can be encoded by $\ll_1$, in the context of word-level model checking, and also argued that a further reduction to $+_1$ can be done in the same way.

In Chapter 3 and Chapter 4, however, we proved that this is not true in the case of satisfiability. We showed that satisfiability of $\mathcal{BV}_1^{\ll}$ is NEXPTIME-hard for $\Omega \supseteq \{\&, |, \sim, =, \ll\}$, but also that hardness does *not* hold anymore for $\Omega_{\ll_1} = \{\&, |, \sim, =, \ll_1\}$. Instead, satisfiability of the resulting logic, $\mathcal{BV}_1^{\ll_1}$, turns out to be PSPACE-complete. By further reducing the operator set to $\Omega_{bw} = \{\&, |, \sim, =\}$, satisfiability for the resulting logic, $\mathcal{BV}_1^{bw}$, even becomes NP-complete.

As a consequence of those results, investigating the complexity of satisfiability for $\Omega_{+_1} = \{\&, |, \sim, =, +_1\}$ arises to be of natural interest. We will now prove that satisfiability for $\mathcal{BV}_1^{+_1}$ is NP-complete. Thus, $+_1$ turns out not to provide any additional expressiveness over bitwise operators when considering satisfiability of a formula. This is captured by the following theorem.

Theorem 12.9. *Satisfiability for $\mathcal{BV}_1^{+_1}$ is* NP*-complete.*

Proof. NP-hardness immediately follows from $\mathcal{BV}_1^{bw} \subseteq \mathcal{BV}_1^{+_1}$. The proof for inclusion follows the concept of the proofs from Chapter 4, when showing $\mathcal{BV}_1^{bw} \in$ NP, and its possible extension by relational operators or indexing: We show that

[3] d^ν being ν times the concatenation of d.

a given formula $\Phi \in \mathcal{BV}_1^{+1}$ is satisfiable if and only if Φ', a certain *small domain encoding* of Φ, i.e., a bit-width reduced, scalar bounded version of Φ, is satisfiable.

Since many proof steps are identical to those from Chapter 4 and also somewhat technical, we will assume the reader is familiar with the general proof concept and only point out necessary adjustments, in a less formal way than we did in Chapter 4. In particular, consider Lemma 4.26. and proofs for related results in Section 4.7.2.

W.l.o.g., we consider formulas Φ in *flat form* and assume that all occurrences of terms $t^{[n]}{+}1$ in the formula have been replaced by Tseitin variables $y^{[n]}$. As part of this process, the constraints $y^{[n]} = x^{[n]}{+}1$ and $x^{[n]} = t^{[n]}$ are conjuncted with the formula, with $x^{[n]}$ being further Tseitin variables.

Let $cnt_{eq}(\Phi)$ and $cnt_{+_1}(\Phi)$ be the number of equalities and the number of increment expressions in Φ, respectively. Furthermore, let ${y_j}^{[n]} = {x_j}^{[n]}{+}1$, for $j \in \{0, \ldots, m{-}1\}$ and $m := cnt_{+_1}$, donate all increment expressions in Φ. We can then write, $\Phi = \Phi_{bw} \wedge \Phi_{+_1}$, with $\Phi_{bw} \in \mathcal{BV}_1^{bw}$ and $\Phi_{+_1} = \bigwedge_{j \in \{0,\ldots,m-1\}} {y_j}^{[n_j]} = {x_j}^{[n_j]}{+}1$. In particular, this implies that all occurrences of $+1$ are within positive equalities between variables. Note that we write $+1$ as a unary operator in affix notation, similar to the usage of increment operators in many common programming languages. This is done to improve readability by avoiding confusion with operators such as unary minus. Finally, we assume that all constants in Φ have been removed. This is similar to the approach in Lemma 4.26 and will be addressed as a remark after this proof.

In our original proof for Lemma 4.26, we argued why it is sufficient to consider a formula Φ' with smaller bit-widths, and we also argued why $cnt_{eq}(\Phi)$, the number of equalities in Φ is a sufficient bound for the bit-width n'. In particular, we defined $n' := \min\{n, cnt_{eq}(\Phi)\}$ for a specific bit-vector in Φ'. We now slightly increase this bound to $n' := \min\{n, cnt_{eq}(\Phi) + 1\}$. Note that this is just to cover the corner case of $cnt_{eq}(\Phi) = cnt_{+_1}(\Phi)$. Obviously, n' is polynomial in $|\Phi|$, and if we can show how solutions for Φ and Φ' can be reduced to each other, we have given a reduction to a scalar bounded formula set (see Definition 4.15).

Remember that the central argument in Lemma 4.26 was based on the fact that different bit-indices are not related and, thus, it is sufficient to guarantee that there can be a distinct bit-index which acts as a *witness* for each disequality. Now, with $+_1$ being part of the operator set, an interaction between different bit-indices actually can happen. However, due to the nature of $+_1$, this is only in a very restricted way, whenever a carry bit is propagated: Propagation of a carry bit in a bit-vector $x^{[n]} = x_{n-1} \ldots x_{i+1} 01 \ldots 1$ is always starting from bit-index 0, until bit-index i, with x_i being the lowest bit of x that is equal to 0. This will result in $y^{[n]} = x_{n-1} \ldots x_{i+1} 10 \ldots 0$. For a set of vectors, $\{x_0, x_1, \ldots, x_{m-1}\}$, we therefore have m such indices i_l, with $l \in \{0, \ldots, m-1\}$, up to which point a carry bit is propagated.

Given a satisfying assignment α' for Φ' we now show how to construct a satisfying assignment α for Φ. As for Lemma 4.26, we simply set all additional bits of bit-vector variables with $n' < n$, to a certain value that we copy from one

of the existing bit-indices. With all bit-indices being unrelated in Lemma 4.26, it was sufficient to choose an arbitrary bit-index, e.g., the most significant one. For the current proof, this still holds for the bitwise part of our formula, but we need to be more specific in order to guarantee that all equalities over increments are still satisfied. Since ${y_j}^{[n'_j]} = {x_j}^{[n'_j]}{+_1} = x_{j,n'-1} \dots x_{j,i_j+1} 01 \dots 1 {+_1} = x_{j,n'-1} \dots x_{j,i_j+1} 10 \dots 0$, it is easy to see that the equality will still hold if and only if we copy any bit-index other than i_j. In particular, considering the full formula, we can choose any bit-index $i \notin \{i_0, \dots, i_{m-1}\}$. Since $n' > cnt_{eq}(\Phi) \geqslant cnt_{+_1}(\Phi)$, we know that this i always exists.

We now prove the other direction. Given α with $\alpha(\Phi) = 1$, we show how to construct α' with $\alpha'(\Phi')$. Similar to Lemma 4.26, we need to select sets M_k with $|M_k| = \min\{n_k, cnt_{eq}(\Phi)\}$ bit-indices, which are sufficient to satisfy the formula. Note that those sets are selected individually for each distinct bit-width n_k in Φ (cf. Lemma 4.26). In order to guarantee satisfiability, we again have to use those bit-indices that can act as a *witness* for each equality $F \in \Phi_{bw}$ with $\alpha(F) = 0$. Arbitrary bits can be selected for equalities $F \in \Phi_{bw}$ with $\alpha(F) = 1$. For equalities $F \in \Phi_{+_1}$, the situation is different. Since all equalities in Φ_{+_1} are positive, $\alpha(F) = 1$ has to hold for all satisfying assignment of Φ, and we do not need to consider $\alpha(F) = 0$. On the other hand, equalities $F \in \Phi_{+_1}$ with $\alpha(F) = 1$ can actually become unsatisfied when removing the wrong bit. As before, ${y_j}^{[n_j]} = {x_j}^{[n_j]}{+_1}$ will evaluate to $x_{j,n-1} \dots x_{j,i+1} 10 \dots 0 = x_{j,n-1} \dots x_{j,i+1} 01 \dots 1 {+_1}$. Thus, a previously satisfied equality will remain satisfied if and only if bit-index i is not removed. We therefore always select the specific bit-indices $i \in \{i_0, \dots, i_{m-1}\}$ for equalities in Φ_{+_1}.

Since $cnt_{eq}(\Phi_{+_1}) = cnt_{+_1}(\Phi_{+_1})$, the total number $|M_k|$ of bit-indices selected for each bit-width n_k in Φ is bounded by $|M_k| \leqslant cnt_{eq}(\Phi)$. If $|M_k| < n'_k = \min\{n_k, cnt_{eq}(\Phi) + 1\}$, we add $n'_k - |M_k|$ arbitrary additional indices to M_k. Finally, all bit-indices apart from those in M_k are removed from all bit-vectors with bit-width n_k. After applying this process for all different bit-widths n_k, the result corresponds to a satisfying assignment α' for Φ'. □

Remark 12.10. Note that we initially used the assumption that constants have been removed from the formula in a preprocessing step. Having discussed the particular effect of the increment operator on a sequence of bits, starting from the lowest index, it is not difficult to see how this can be achieved. This is similar to the removal of constants in Lemma 4.26, and, even more, to the treatment of constants for inequality constraints in Theorem 4.29, by considering the least significant bits seperately. For example, the least significant bit of an increment constraint

$$y^{[n]} = x^{[n]}{+_1}$$

can be removed by splitting into

$$({x_0}^{[1]} = {\sim} 0^{[1]}) \rightarrow \Big(\Big({y_0}^{[1]} = 0^{[1]}\Big) \wedge \Big({y_{\mathrm{HI}}}^{[n-1]} = {x_{\mathrm{HI}}}^{[n-1]}{+_1}\Big)\Big)$$

$$\wedge$$

$$(x_0{}^{[1]} = 0^{[1]}) \rightarrow \left(\left(y_0{}^{[1]} = \sim 0^{[1]}\right) \wedge \left(y\text{HI}^{[n-1]} = x\text{HI}^{[n-1]}\right)\right),$$

using the notion of $x\text{HI}$ and $y\text{HI}$ from Lemma 4.26. Iterative application will remove further bits and, thus, all non-zero bits of any constants in the formula can be removed (cf. Theorem 4.29).

12.6.6 Satisfiability of $\mathcal{BV}_\nu^{bw}$, $\mathcal{BV}_\nu^{+_1}$ and $\mathcal{BV}_\nu^{\ll_1}$

Bringing together the previous results, we already covered a broad range of operators and encodings. We showed how moving from a unary to a single-logarithmic encoding affects complexity of bit-vector logics with certain sets of operators. We want to provide a further unification by discussing the effect of multi-logarithmic encodings on two restricted sets of operators that turned out to be easier for the single-logarithmic case.

In particular, we know that $\mathcal{BV}_0^{bw}$, $\mathcal{BV}_1^{bw}$, $\mathcal{BV}_0^{+_1}$ and $\mathcal{BV}_1^{+_1}$ are NP-complete, i.e., the complexity of the respective operator sets does not differ between unary encodings and single-logarithmic encodings for scalars. It is not surprising that this is still true for multi-logarithmic encodings. This directly follows from the corresponding proofs, e.g., those for Lemma 4.26 and for Theorem 12.9. The reader can easily check that the succinctness of the scalar encodings will have no influence on the reductions used in both proofs.

In contrast to that, we know that the complexity increases from being NP-complete to being PSPACE-complete, when moving from $\mathcal{BV}_0^{\ll_1}$ to $\mathcal{BV}_1^{\ll_1}$—which makes the situation less obvious for the specific operator set in combination with multi-logarithmic encodings. Nevertheless, we conjecture that $\mathcal{BV}_\nu^{\ll_1}$ will remain PSPACE-complete for $\nu > 1$. While leaving a full formal proof to future work, we will sketch our argument: Remember that we proved PSPACE-inclusion for $\nu = 1$ by providing a reduction to the model checking problem on sequential circuits, similar to a reduction for QFPAbit in [208, 209]. Note that the related proof in [208, 209] already considered bit-vectors of infinite length, and we explicitly extended our version by introducing a counter to check for fixed bit-widths up to 2^n. While we cannot realize a counter for $\exp_\nu(n), \nu > 1$ as a polynomial size sequential circuit, we actually conjecture that this is not necessary. Since there are only $\exp_1(|\Phi|) = O(2^n)$ possible combinations of input values and state values for a sequential circuit representation of Φ (leaving aside the counter variables), we can argue that the underlying structure has to be *lasso shaped* and it must be possible to reach a loop after at most 2^n steps. This is the same argument used for the *recurrence diameter* in *bounded model checking* [28, 157]. Therefore, we should be able to reduce any problem $\Phi \in \mathcal{BV}_\nu^{\ll_1}$ to a problem with bit-widths in $O(2^{|\Phi|})$.

Interestingly, this consideration in the context of bounded model checking also offers a further point of view on formulas in $\mathcal{BV}_\nu^{+_1}$. Since we know that $\mathcal{BV}_\nu^{+_1}$ is NP-complete, and that it is sufficient to consider corresponding formulas with $O(p(|\Phi|))$ bits, this should also hold when the problem is re-encoded as a model

Scalar Encoding → ↓ *Operators*	unary $\nu = 0$	binary $\nu = 1$	ν-logarithmic $\nu > 1$
$\mathcal{BV}_\nu^{bw}, \mathcal{BV}_\nu^{+_1}$	NP	NP	NP
$\mathcal{BV}_\nu^{+}, \mathcal{BV}_\nu^{\ll_1}$	NP	PSPACE	PSPACE [4]
$\mathcal{BV}_\nu^{\ll_c}$	NP	NEXPTIME	?
$\mathcal{BV}_\nu^{\ll}$	NP	NEXPTIME	ν-NEXPTIME

Table 12.1: Complexity classes for various operator sets in combination with different degrees of succinctness. All operator sets include bitwise operators and equality. Additionally, the operator in the superscript is included (none, for bw).

checking problem in our common way. If we now used a bounced model checker, it should be sufficient to consider polynomial bounds. This is related to the topic of *completeness thresholds*, as discussed in [69, 157]. In general, completeness thresholds are very difficult to obtain. Our results might help doing so for restricted problem classes. This could improve the performance of bounded model checkers on those problem classes. However, a more thorough problem discussion and a more formal analysis are needed first—both are out of the scope of this thesis. Therefore, this should only be considered as preliminary arguments and a proposal for future work.

Table 12.1 depicts the conjectures from this section. Note that this is also related to the discussion about expressiveness of $\ll_c$ and $\ll$, depending on the degree of the logarithm used of the scalar encoding (see Section 12.6.4). It seems that, the more succinct the bit-width encoding is, the more "powerful" a shift (or addition) operation has to be in order to profit from this succinctness. Whereas $+_1$ is too weak to increase complexity for any logarithmic encoding, $+$ or $\ll_1$ are sufficient to lift complexity to being complete for PSPACE, but not for NEXPTIME, as it is possible for $\ll_c$. However, our proof of Theorem 12.5, for a general ν-logarithmic encoding, actually required $\ll$, as discussed in Section 12.6.4. We would expect that a general NEXPTIME-inclusion result could probably be obtained for the restricted version with $\ll_c$, also for all $\nu > 1$. This might be another topic for future work.

12.6.7 Quantified $\mathcal{BV}_\nu^{\Omega}$ with Uninterpreted Functions

Those preliminary results regarding operators lead us to a final consideration regarding quantified logics. In Chapter 2 and Chapter 4, we discussed several results for quantified bit-vector logics with uninterpreted functions. The unary case was already proved to be NEXPTIME-complete in [234, 233], and our results showed 2-NEXPTIME-completeness for binary encodings (see Theorem 4.37). Revisit-

[4] Only preliminary arguments were given so far. A formal proof is still required.

ing these results in the context of multi-logarithmic encodings, it is easy to see that all arguments in our reduction (see Lemma 4.40) still hold. This implies $(\nu+1)$-NEXPTIME-completeness of ν-logarithmic encoded bit-vector logics with quantifiers and uninterpreted functions, thus, for all $\nu > 1$, remaining exponentially harder than their quantifier-free counterpart (for $\Omega \supseteq \{\&, |, \sim, =, \ll\}$).

Care has to be taken when considering the required operator sets. As shown in Proposition 4.42, the hardness result for quantified bit-vector logics with uninterpreted functions already holds for $\Omega \supseteq \{\&, |, \sim, =, \ll_1\}$. We showed this by giving a defintion similar to the one in Peano arithmetic. It is not difficult to see that this can be further reduced to $\Omega \supseteq \{\&, |, \sim, =, +_1\}$, in the same way.

Throughout our earlier work, we sometimes used phrases, such as "This shows that, informally speaking, binary encoding on scalars has the same expressive power as quantification and uninterpreted functions altogether."—while this is true for bit-vector logics with $\Omega \supseteq \{\&, |, \sim, =, \ll\}$, it does not fully capture the overall picture. Instead, one should also consider the fact that quantified bit-vector logics with uninterpreted functions require a much weaker operator set, compared to their quantifier-free counterparts with more succinct scalar encodings. In this sense, quantification and uninterpreted functions can actually be considered to remain more expressive.

Finally, note that the complexity results for quantified bit-vector logics with uninterpreted functions, thus, are very close to those for succinct bit-vector encodings in [154]—both only require an increment operator $+_1$, in addition to logical/bitwise operators and equality. Furthermore, for bit-vector logics with quantifiers and uninterpreted functions, as also for the problem classes in [154], even $+_1$ can be removed when a unary encoding for the scalars is used.

Chapter 13

Conclusion

As already mentioned at several occasions, bit-vector logics are of central importance in many areas of computer science. With growing computational power and with the development of more sophisticated algorithms, the general interest in bit-vectors has steadily grown over the last decades. Considering that more and more applications are getting into the reach of current solvers, this process will probably keep going on in the near future. In particular, the question of satisfiability is of huge relevance for many applications. SAT solvers, over the last two decades, have become incredibly efficient, solving industrial formulas with up to millions of variables—an effect, that has not nearly been expected when research on SAT was still at its beginning or when, in 1971, Cook presented his NP-completeness proof. This huge success of SAT solvers was mainly due to the CDCL paradigm [170] and related heuristics, such as variable scoring schemes [29], or restart schemes [30]. The increase in performance of SAT solvers paved the way to deal with even more complex problems, such as quantified formulas or bit-vector logics. Research in all those areas is still an ongoing process and solvers steadily improve further. In this thesis, we discussed bit-vectors and related topics from several perspectives. We contributed to scientific research areas related to theory as well as practice.

Regarding theory, we proved a large number of new complexity results for several classes of bit-vector problems. In particular, we were the first to show the effect of particular encodings used for all kind of scalars in bit-vector logics [151, 101, 153, 154]. For example, quantifier-free bit-vector formulas turned out to be NEXPTIME-complete, when using a binary encoding [151]. This exponential growth, compared to the logic with unary encoding, also holds if quantification and uninterpreted functions are used, resulting in a 2-NEXPTIME-complete logic [151]. In the quantifier-free case, certain simple restrictions on operators can also produce prototypical NP, PSPACE, and NEXPTIME-complete logics [101]. Equivalences between certain operators lead to several alternative characterizations and, also for the quantified case, certain restrictions can be shown to decrease or not decrease expressiveness [153].

As a side result, our theoretical work also lead to the construction of new benchmarks for SMT and SAT solvers [152]. On the practical side, we presented

new algorithms for several different problems, including general bit-vector formulas [98, 150], restricted classes thereof [100], and DQBF [99, 102]. In the general case, quantifier-free bit-vector logics can either be solved by directly applying a stochastic local search approach on the theory representation [98], or can be translated to EPR, then being solved by an EPR-solver [150]. For the more restricted PSPACE class, a translation to SMV can allow model checkers to efficiently solve certain bit-vector formulas [100]. To solve DQBF, we proposed as DPLL based approach [99], similar to the one for SAT [78] and QBF [61], and an instantiation based one [102], as also used for EPR [147, 148].

Aside from those results, which were presented in Chapters 2–10, we then discussed the author's individual contribution in Chapter 11 and extended previous work in Chapter 12, by putting it into relation to other recent developments, and pointing to possible directions of future work. In regard to complexity, we took a closer look at possible limits of quantification in bit-vector logics, related to quantifier alternations. We also pointed out the importance of alternative solving approaches for application problems, even in the light of recently shown para-NP-completeness of many practical problems [177]. We then discussed new developments in actual DQBF solving [111], together with the importance of recently developed preprocessing techniques for DQBF [232], and emphasized the need for benchmark libraries. Concerning SLS for SMT, we briefly described the recent concept of path propagation [181] and suggested possible directions for parallelization of our SLS framework. On the complexity side, we also presented a way of encoding of Turing machines into word-level model checking problems.

Moreover, Chapter 12 also contains several new theoretical results. In related work [154], a generic upgrading theorem has been obtained, showing that succinct representations by bit-vector formulas with multi-logarithmic scalar encodings increase complexity of certain problems exponentially for each degree of succinctness. We now showed a similar result for satisfiability of bit-vector formulas, proving that its complexity also increases exponentially for each degree of succinctness of scalar encodings. From the theoretical perspective, those results are particularly nice in the overall context of our work, since it further confirms the very general nature of bit-vector representations.

Highlighting the difference in the operators required for the upgrading theorem in [154] and our new results for satisfiability, we further analyzed the expressiveness of the involved operators. As a result, we were able to provide a new NP-completeness results for a certain class of bit-vector logics, containing an increment operator. This extended, e.g., previous fragments from [153], and allowed us to emphasize the fundamental way, in which a restriction of addition and shifts influences the underlying complexity of bit-vector formulas.

Aside from this, we think that research on bit-vectors and related topics still offers many more possible directions for future work. We hope that our results will also help other researchers in their work to continuously improve our understanding of those topics as well as our ability of solving practical problems in applications.

Bibliography

[1] Rajeev Agrawal. Sample mean based index policies with o(log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.

[2] Alejandro Arbelaez and Youssef Hamadi. Improving parallel local search for SAT. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, volume 6683 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Boutilier [39], pages 399–404.

[4] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In Michela Milano, editor, *Principles and Practice of Constraint Programming - CP 2012 - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 118–126. Springer, 2012.

[5] Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 Competition. In Balint et al. [12], pages 42–43.

[6] Abdelwaheb Ayari, David A. Basin, and Felix Klaedtke. Decision procedures for inductive boolean functions based on alternating automata. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification – 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 170–185. Springer, 2000.

[7] Salman Azhar, Gary Peterson, and John Reif. Lower bounds for multiplayer non-cooperative games of incomplete information. *Computers & Mathematics with Applications*, 41:957–992, 2001.

[8] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*, 523:86–100, 2014.

[9] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong Roland Jiang. Henkin quantifiers and boolean formulae. In Cimatti and Sebastiani [67], pages 129–142.

[10] Valeriy Balabanov and Jie-Hong Roland Jiang. Reducing satisfiability and reachability to DQBF, 2015. Presentation at *3rd International Workshop on Quantified Boolean Formulas, QBF 2015, Affiliated to SAT 2015, Austin, TX, USA, September 23, 2015.*

[11] José L. Balcázar, Antoni Lozano, and Jacobo Torán. The complexity of algorithmic problems on succinct instances. *Computer Science*, pages 351–377, 1992.

[12] Adrian Balint, Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors. *Proceedings SAT Competition 2013, Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2013.

[13] Adrian Balint, Armin Biere, Andreas Fröhlich, and Uwe Schöning. Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses. In Sinz and Egly [204], pages 302–316.

[14] Adrian Balint and Andreas Fröhlich. Improving stochastic local search for SAT with a new probability distribution. In Strichman and Szeider [217], pages 10–15.

[15] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In Cimatti and Sebastiani [67], pages 16–29.

[16] Tomáš Balyo, Andreas Fröhlich, Marijn Heule, and Armin Biere. Everything you always wanted to know about blocked sets (but were afraid to ask). In Sinz and Egly [204], pages 317–332.

[17] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. [31], February 2009.

[18] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In *Proceedings 8th International Workshop on Satisfiability Modulo Theories, SMT 2010, Affiliated to CAV 2010 and SAT 2010, Edinburgh, UK, July 14–15, 2010*. Informal Proceedings, 2010.

[19] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In M. J. Irwin, editor, *Proceedings 35th Design Automation Conference, DAC 1998, San Francisco, CA, USA, June 15-19, 1998*, pages 522–527. ACM, 1998.

[20] Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors. *Proceedings SAT Competition 2014, Solver and Benchmark Descriptions*, volume B-2014-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2014.

[21] Anton Belov, Matti Järvisalo, and Zbigniew Stachniak. Depth-driven circuit-level stochastic local search for SAT. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 504–509. IJCAI/AAAI, 2011.

[22] Marco Benedetti. Evaluating QBFs via symbolic skolemization. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, volume 3452 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2005.

[23] Paul Bernays and Moses Schönfinkel. Zum entscheidungsproblem der mathematischen logik. *Mathematische Annalen*, 99(1):342–372, 1928.

[24] Armin Biere. Resolve and expand. In Hoos and Mitchell [126], pages 59–70.

[25] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Büning and Zhao [57], pages 28–33.

[26] Armin Biere. PicoSAT essentials. *Journal on Satisfiability*, 4(2-4):75–97, 2008.

[27] Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In Belov et al. [20], pages 39–40.

[28] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[29] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In Heule and Weaver [124], pages 405–422.

[30] Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In *Proceedings 6th International Workshop on Pragmatics of SAT, POS 2015, Affiliated to SAT 2015, Austin, TX, USA, September 23, 2015*, EPiC Series. EasyChair, 2016. to appear.

[31] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[32] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2011.

[33] Per Bjesse. A practical approach to word level model checking of industrial netlists. In Gupta and Malik [120], pages 446–458.

[34] Nikolaj Bjorner, Kenneth McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In Fontaine and Goel [95], pages 3–11.

[35] Nikolaj Bjørner and Mark C. Pichora. Deciding fixed and non-fixed size bitvectors. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems - 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1384 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 1998.

[36] Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-based synthesis methods for safety specs. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.

[37] Roderick Bloem and Natasha Sharygina, editors. *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. IEEE, 2010.

[38] Bernd Borchert and Antoni Lozano. Succinct circuit representations and leaf language classes are basically the same concept. *Information Processing Letters*, 59(4):211–215, 1996.

[39] Craig Boutilier, editor. *IJCAI 2009, Proceedings 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. AAAI Press, 2009.

[40] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin,*

TX, USA, January 23-25, 2011. Proceedings, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[41] Aaron R. Bradley. Understanding IC3. In Cimatti and Sebastiani [67], pages 1–14.

[42] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation - 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[43] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 144–153. FMCAD Inc., 2011.

[44] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.

[45] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the ASPDAC 2002 / VLSI Design 2002, CD-ROM, 7-11 January 2002, Bangalore, India*, pages 741–746. IEEE Computer Society, 2002.

[46] Ed Brinksma and Kim Guldstrand Larsen, editors. *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[47] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[48] Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: bit-precise modelling of word-level problems for model checking. In Clark Barrett, editor, *Proceedings of the Joint Workshops of the 6th International Workshop*

on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, SMT/BPR 2008, Affiliated to CAV 2008, July 14, 2008, Princeton, NJ, USA, pages 33–38. ACM, 2008.

[49] Roberto Bruttomesso. *RTL Verification: From SAT to SMT(BV)*. PhD thesis, University of Trento, 2008.

[50] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In Gupta and Malik [120], pages 299–303.

[51] Roberto Bruttomesso and Natasha Sharygina. A scalable decision procedure for fixed-width bit-vectors. In Jaijeet S. Roychowdhury, editor, *2009 International Conference on Computer-Aided Design, ICCAD 2009, San Jose, CA, USA, November 2-5, 2009*, pages 13–20. ACM, 2009.

[52] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[53] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In Brinksma and Larsen [46], pages 78–92.

[54] Uwe Bubeck and Hans Kleine Büning. Encoding nested boolean functions as quantified boolean formulas. *Journal on Satisfiability*, 8(1/2):101–116, 2012.

[55] Uwe Bubeck and Hans Kleine Büning. Dependency quantified horn formulas: Models and complexity. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 198–211. Springer, 2006.

[56] Uwe Bubeck and Hans Kleine Büning. Nested boolean functions as models for quantified boolean formulas. In Järvisalo and Gelder [138], pages 267–275.

[57] Hans Kleine Büning and Xishun Zhao, editors. *Theory and Applications of Satisfiability Testing - SAT 2008 – 11th International Conference, SAT 2008,*

Guangzhou, China, May 12-15, 2008. Proceedings, volume 4996 of *Lecture Notes in Computer Science*. Springer, 2008.

[58] Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve, and Steven P. Miller. A bitvectors library for PVS. Technical report, NASA Langley Research Center, Hampton, Virginia, USA, August 1996.

[59] Bv2epr project page. Website. `http://fmv.jku.at/bv2epr/`.

[60] Bv2smv project page. Website. `http://fmv.jku.at/bv2smv/`.

[61] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. In Jack Mostow and Chuck Rich, editors, *Proceedings Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 262–267. AAAI Press / The MIT Press, 1998.

[62] Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for SAT. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012.

[63] Supratik Chakraborty, Zurab Khasidashvili, Carl-Johan H. Seger, Rajkumar Gajavelly, Tanmay Haldankar, Dinesh Chhatani, and Rakesh Mistry. Word-level symbolic trajectory evaluation. In Kroening and Pasareanu [156], pages 128–143.

[64] Krishnendu Chatterjee, Thomas A. Henzinger, Jan Otop, and Andreas Pavlogiannis. Distributed synthesis for LTL fragments. In Barbara Jobstmann and Sandip Ray, editors, *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 18–25. IEEE, 2013.

[65] Bogdan S. Chlebus. From domino tilings to a new model of computation. In Andrzej Skowron, editor, *Computation Theory - Fifth Symposium, Zaborów, Poland, December 3-8, 1984, Proceedings*, volume 208 of *Lecture Notes in Computer Science*, pages 24–33. Springer, 1984.

[66] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Brinksma and Larsen [46], pages 359–364.

[67] Alessandro Cimatti and Roberto Sebastiani, editors. *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*. Springer, 2012.

[68] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003.

[69] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[70] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[71] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In Esparza and Majumdar [92], pages 236–250.

[72] Stephen Cook and Michael Soltys. Boolean programs and quantified propositional proof systems. *Bulletin of the Section of Logic*, 28(3):119–129, 1999.

[73] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[74] David Cyrluk, M. Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1997.

[75] William J. Dally and R. Curtis Harting. *Digital Design, A Systems Approach*. Cambridge University Press, 2012.

[76] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[77] Bireswar Das, Patrick Scharpfenecker, and Jacobo Torán. Succinct encodings of graph isomorphism. In Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 285–296. Springer, 2014.

[78] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[79] Martin Davis, Yuri Matijasevich, and Julia Robinson. Hilbert's tenth problem: Diophantine equations: positive aspects of a negative solution. In *Proceedings of Symposia in Pure Mathematics: Vol.: 28. : Mathematical Developments Arising from Hilbert : Problems*, volume 28 of *Proceedings of Symposia in Pure Mathematics*, pages 323–378. American Mathematical Society, 1976.

[80] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[81] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[82] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2009.

[83] Leonardo Mendonça de Moura and Dejan Jovanovic. Model-driven decision procedures for arithmetic. In Nikolaj Bjørner, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie, editors, *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, page 11. IEEE Computer Society, 2013.

[84] Francesco M. Donini, Paolo Liberatore, Fabio Massacci, and Marco Schaerf. Solving QBF by SMV. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*, pages 578–592. Morgan Kaufmann, 2002.

[85] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.

[86] Rolf Drechsler, Tommi Junttila, and Ilkka Niemelä. *Non-Clausal SAT and*

ATPG, chapter 21, pages 655–693. Volume 185 of Biere et al. [31], February 2009.

[87] Bruno Dutertre and Leonardo Mendonça de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[88] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[89] Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In McMillan et al. [173], pages 291–308.

[90] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In Bloem and Sharygina [37], pages 137–144.

[91] Amit Erez and Alexander Nadel. Finding bounded path in graph using SMT for automatic clock routing. In Kroening and Pasareanu [156], pages 20–36.

[92] Javier Esparza and Rupak Majumdar, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*. Springer, 2010.

[93] Joan Feigenbaum, Sampath Kannan, Moshe Y. Vardi, and Mahesh Viswanathan. Complexity of problems on graphs represented as OBDDs. *Chicago Journal of Theoretical Computer Science*, 5(5), 1999.

[94] Bernd Finkbeiner and Leander Tentrup. Fast DQBF refutation. In Sinz and Egly [204], pages 243–251.

[95] Pascal Fontaine and Amit Goel, editors. *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Affiliated to IJCAR 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series*. EasyChair, 2013.

[96] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.

[97] Edwin E. Freed. Binary magic numbers – some applications and algorithms. *Dr. Dobb's Journal of Software Tools*, 8(4):24–37, 1983.

[98] Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef

Hamadi. Stochastic local search for satisfiability modulo theories. In Blai Bonet and Sven Koenig, editors, *Proceedings Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1136–1143. AAAI Press, 2015.

[99] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. A DPLL algorithm for solving DQBF. In *Proceedings 3rd International Workshop on Pragmatics of SAT, POS 2012, Affiliated to SAT 2012, Trento, Italy, June 16, 2012*. Informal Proceedings, 2012.

[100] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. Efficiently solving bit-vector problems using model checkers. In *Proceedings 11th International Workshop on Satisfiability Modulo Theories, SMT 2013, Affiliated to SAT 2013, Helsinki, Finland, July 8-9, 2013*, pages 6–15. Informal Proceedings, 2013.

[101] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In Andrei A. Bulatov and Arseny M. Shur, editors, *Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings*, volume 7913 of *Lecture Notes in Computer Science*, pages 378–390. Springer, 2013.

[102] Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. iDQ: Instantiation-based DQBF solving. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop, Affiliated to SAT 2014, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 of *EPiC Series*, pages 103–116. EasyChair, 2014.

[103] Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56(3):183–198, 1983.

[104] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Damm and Hermanns [76], pages 519–531.

[105] Michael R. Garey and David S. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *Journal of the ACM*, 25(3):499–508, July 1978.

[106] Michael R. Garey and David .S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[107] William I. Gasarch. Guest column: the second P =? NP poll. *SIGACT News*, 43(2):53–77, 2012.

[108] Ian P. Gent, Enrico Giunchiglia, Massimo Narizzano, Andrew G. D. Row-

ley, and Armando Tacchella. Watched data structures for QBF solvers. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2003.

[109] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking for partial implementations revisited. In Christian Haubelt and Dirk Timmermann, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Warnemünde, Germany, March 12-14, 2013.*, pages 61–70. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2013.

[110] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking of partial designs using dependency quantified boolean formulae. In *2013 IEEE 31st International Conference on Computer Design, ICCD 2013, Asheville, NC, USA, October 6-9, 2013*, pages 396–403. IEEE Computer Society, 2013.

[111] Karina Gitina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker. Solving DQBF through quantifier elimination. In Wolfgang Nebel and David Atienza, editors, *Proceedings 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 1617–1622. ACM, 2015.

[112] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2005. `www.qbflib.org`.

[113] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. sQueezeBF: An effective preprocessor for QBFs based on equivalence reasoning. In Strichman and Szeider [217], pages 85–98.

[114] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for quantified boolean logic satisfiability. In Rina Dechter and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 649–654. AAAI Press / The MIT Press, 2002.

[115] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[116] Evguenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust satsolver. In *2002 Design, Automation and Test in Europe Conference and*

Exposition (DATE 2002), 4-8 March 2002, Paris, France, pages 142–149. IEEE Computer Society, 2002.

[117] Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a source of complexity in logical formalisms. *Annals of Pure and Applied Logic*, 97(1):231–260, 1999.

[118] Alexandra Goultiaeva, Martina Seidl, and Armin Biere. Bridging the gap between dual propagation and CNF-based QBF solving. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 811–814. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[119] Alberto Griggio, Quoc-Sang Phan, Roberto Sebastiani, and Silvia Tomasi. Stochastic local search for SMT: combining theory solvers with walksat. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2011.

[120] Aarti Gupta and Sharad Malik, editors. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[121] Pierre Hansen, Nenad Mladenović, and José A. Moreno Pérez. Variable neighbourhood search: methods and applications. *4OR*, 6(4):319–360, 2008.

[122] L. Henkin. Some remarks on infinitely long formulas. In *Infinistic Methods*, pages 167–183. Pergamon Press, 1961.

[123] Marijn Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2014.

[124] Marijn Heule and Sean Weaver, editors. *Theory and Applications of Satisfiability Testing - SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*. Springer, 2015.

[125] Krystof Hoder, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Preprocessing techniques for first-order clausification. In Gianpiero Cabodi and Satnam Singh, editors, *Formal Methods in Computer-*

Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012, pages 44–51. IEEE, 2012.

[126] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2005.

[127] Holger H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In H. van Maaren I. P. Gent and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, pages 283–292. IOS Press, 2000.

[128] Holger H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann, 2005.

[129] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Veloso [229], pages 2318–2323.

[130] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[131] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[132] Neil Immerman. *Descriptive complexity*. Springer, 1999.

[133] Russell Impagliazzo and Ramamohan Paturi. Complexity of k-SAT. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity, Atlanta, Georgia, USA, May 4-6, 1999*, pages 237–240. IEEE Computer Society, 1999.

[134] Mikoláš Janota, Radu Grigore, and João Marques-Silva. On QBF proofs and preprocessing. In McMillan et al. [173], pages 473–489.

[135] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. In Cimatti and Sebastiani [67], pages 114–128.

[136] Mikoláš Janota and Joao Marques-Silva. On propositional QBF expansions and Q-resolution. In Järvisalo and Gelder [138], pages 67–82.

[137] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Esparza and Majumdar [92], pages 129–144.

[138] Matti Järvisalo and Allen Van Gelder, editors. *Theory and Applications of*

Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings, volume 7962 of *Lecture Notes in Computer Science*. Springer, 2013.

[139] Matti Järvisalo, Tommi A. Junttila, and Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for boolean circuits. *Ann. Math. Artif. Intell.*, 44(4):373–399, 2005.

[140] Peer Johannsen. Reducing bitvector satisfiability problems to scale down design sizes for RTL property checking. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop 2001, Monterey, California, USA, November 7-9, 2001*, pages 123–128. IEEE Computer Society, 2001.

[141] Peer Johannsen. *Speeding Up Hardware Verification by Automated Data Path Scaling*. PhD thesis, CAU Kiel, Germany, 2002.

[142] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[143] Zurab Khasidashvili, Mahmoud Kinanah, and Andrei Voronkov. Verifying equivalence of memories using a first order logic theorem prover. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 128–135. IEEE, 2009.

[144] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In Sheng Yu and Andrei Paun, editors, *Implementation and Application of Automata, 5th International Conference, CIAA 2000, London, Ontario, Canada, July 24-25, 2000, Revised Papers*, volume 2088 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2000.

[145] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms*. Addison-Wesley, 2011.

[146] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.

[147] Konstantin Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[148] Konstantin Korovin. Instantiation-based automated reasoning: From theory to practice. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 163–166. Springer, 2009.

[149] Konstantin Korovin. Inst-gen - A modular approach to instantiation-based automated reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 239–270. Springer, 2013.

[150] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. BV2EPR: A tool for polynomially translating quantifier-free bit-vector formulas into epr. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 443–449. Springer, 2013.

[151] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In Fontaine and Goel [95], pages 44–55.

[152] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Quantifier-free bit-vector formulas with binary encoding: Benchmark description. In Balint et al. [12], pages 107–108.

[153] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, pages 1–54, 2015.

[154] Gergely Kovásznai, Helmut Veith, Andreas Fröhlich, and Armin Biere. On the complexity of symbolic verification and decision problems in bit-vector logic. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, volume 8635 of *Lecture Notes in Computer Science*, pages 481–492. Springer, 2014.

[155] Lukas Kroc, Ashish Sabharwal, Carla P. Gomes, and Bart Selman. Integrating systematic and local search paradigms: A new strategy for maxsat. In Boutilier [39], pages 544–551.

[156] Daniel Kroening and Corina S. Pasareanu, editors. *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*. Springer, 2015.

[157] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[158] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[159] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based boolean reasoning. In *Proceedings 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 232–237. ACM, 2001.

[160] Reinhold Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In Uwe Egly and Christian G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Copenhagen, Denmark, July 30 - August 1, 2002, Proceedings*, volume 2381 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2002.

[161] Harry R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.

[162] Chengqian Li and Yi Fan. Cca2013. In Balint et al. [12].

[163] Chu Min Li and Yu Li. Satisfying versus falsifying in local search for satisfiability. In Cimatti and Sebastiani [67], pages 477–478.

[164] Florian Lonsing and Armin Biere. Nenofex: Expanding NNF for QBF solving. In Büning and Zhao [57], pages 196–210.

[165] Florian Lonsing and Armin Biere. Integrating dependency schemes in search-based QBF solvers. In Strichman and Szeider [217], pages 158–171.

[166] Florian Lonsing and Armin Biere. Failed literal detection for QBF. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2011.

[167] Antoni Lozano and José L. Balcázar. The complexity of graph problems for succinctly represented graphs. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG '89, Castle Rolduc, The Netherlands, June 14-16, 1989, Proceedings*, volume 411 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 1989.

[168] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[169] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. BAT: the bit-level analysis tool. In Damm and Hermanns [76], pages 303–306.

[170] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, 48:506–521, 1999.

[171] Maarten Marx. Complexity of modal logic. In *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 139–179. Elsevier, 2007.

[172] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[173] Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*. Springer, 2013.

[174] N. Mladenović. A variable neighborhood algorithm – a new metaheuristics for combinatorial optimization. In *Abstracts of Papers Presented at Optimization Days. Montréal*, page 112, 1995.

[175] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings DAC'01*, pages 530–535, 2001.

[176] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[177] Alexander Nadel and Nachum Dershowitz. Is bit-vector reasoning as hard as nexptime in practice? In *Proceedings 13th International Workshop on Satisfiability Modulo Theories, SMT 2015, Affiliated to CAV 2015, San Francisco, CA, USA, July 18-19, 2015*. Informal Proceedings, 2015.

[178] Yehuda Naveh. Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography. In *Proceedings 1st International Workshop on Local Search Techniques, LSCS 2004, Affiliated to CP 2004, September 27, 2004, Toronto, Canada*. Informal Proceedings, 2004.

[179] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Magazine*, 28(3):13–30, 2007.

[180] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability*, 9:53–58, 2015.

[181] Aina Niemetz, Mathias Preiner, Andreas Fröhlich, and Armin Biere. Improving local search for bit-vector logics in SMT with path propagation. In *Proceedings 4th International Workshop on Design and Implementation of Formal Tools and Systems, DIFTS 2015, Affiliated to FMCAD 2015, Austin, TX, USA, September 26-27, 2015*. Informal Proceedings, 2015.

[182] Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF - (tool presentation). In Cimatti and Sebastiani [67], pages 430–435.

[183] Matthias Niewerth and Thomas Schwentick. Two-variable logic and key constraints on data words. In Tova Milo, editor, *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 138–149. ACM, 2011.

[184] Chanseok Oh. MiniSat HACK 999ED, MiniSat HACK 1430ED and SWDiA5BY. In Belov et al. [20], pages 46–47.

[185] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[186] Christos H Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71(3):181–185, 1986.

[187] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 348–363. IEEE Computer Society, 1979.

[188] Gary L. Peterson, John H. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7-8):957–992, April 2001.

[189] Duc Nghia Pham, John Thornton, and Abdul Sattar. Building structure into local search for SAT. In Veloso [229], pages 2359–2364.

[190] Anh-Dung Phan, Nikolaj Bjørner, and David Monniaux. Anatomy of alternating quantifier satisfiability (work in progress). In Fontaine and Goel [95], pages 120–130.

[191] Florian Pigorsch and Christoph Scholl. An aig-based qbf-solver using SAT for preprocessing. In Sachin S. Sapatnekar, editor, *Proceedings 47th Design*

Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010, pages 170–175. ACM, 2010.

[192] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[193] David A. Plaisted. A decision procedure for combinations of propositional temporal logic and other specialized theories. *Journal of Automated Reasoning*, 2(2):171–190, 1986.

[194] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[195] Steven David Prestwich. Random walk with continuously smoothed variable weights. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 203–215. Springer, 2005.

[196] Vadim Ryvchin and Ofer Strichman. Local restarts. In Büning and Zhao [57], pages 271–276.

[197] Horst Samulowitz, Jessica Davies, and Fahiem Bacchus. Preprocessing QBF. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 514–529. Springer, 2006.

[198] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.

[199] Christoph Scholl and Bernd Becker. Checking equivalence for partial implementations. In *Proceedings 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 238–243. ACM, 2001.

[200] Tobias Schüle and Klaus Schneider. Verification of data paths using unbounded integers: Automata strike back. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers*, volume 4383 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2006.

[201] Michael J. Schulte, Mustafa Gok, Pablo I. Balzola, and Robert W. Brocato. Combined unsigned and two's complement saturating multipliers. In Franklin T. Luk, editor, *Advanced Signal Processing Algorithms, Architectures, and Implementations X, July 30, 2000, San Diego, CA, USA*, volume 4116 of *Proceedings of SPIE*, pages 185–196. SPIE, July 2000.

[202] Martina Seidl, Florian Lonsing, and Armin Biere. qbf2epr: A tool for generating EPR formulas from QBF. In Pascal Fontaine, Renate A. Schmidt, and Stephan Schulz, editors, *PAAR-2012. Third Workshop on Practical Aspects of Automated Reasoning, Affiliated to CADE 2012, Manchester, UK, June 30 - July 1, 2012*, volume 21 of *EPiC Series*, pages 139–148. EasyChair, 2012.

[203] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, pages 337–343. AAAI Press / The MIT Press, 1994.

[204] Carsten Sinz and Uwe Egly, editors. *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*. Springer, 2014.

[205] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[206] Friedrich Slivovsky and Stefan Szeider. Variable dependencies and Q-resolution. In *1st International Workshop on Quantified Boolean Formulas, QBF 2013, Affiliated to SAT 2013, Helsinki, Finnland, June 9, 2013.*, page 22. Informal Workshop Report, 2013.

[207] Friedrich Slivovsky and Stefan Szeider. Variable dependencies and Q-resolution. In Sinz and Egly [204], pages 269–284.

[208] Andrej Spielmann and Viktor Kuncak. On synthesis for unbounded bit-vector arithmetic. Technical report, EPFL, Lausanne, Switzerland, February 2012.

[209] Andrej Spielmann and Viktor Kuncak. Synthesis for unbounded bit-vector arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 2012.

[210] Iain A. Stewart. Complete problems involving boolean labelled structures

and projection transactions. *Journal of Logic and Computation*, 1(6):861–882, 1991.

[211] Iain A. Stewart. On completeness for NP via projection translations. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, and Michael M. Richter, editors, *Computer Science Logic, 5th Workshop, CSL '91, Berne, Switzerland, October 7-11, 1991, Proceedings*, volume 626 of *Lecture Notes in Computer Science*, pages 353–366. Springer, 1991.

[212] Iain A. Stewart. Using the Hamiltonian path operator to capture NP. *Journal of Computer and System Sciences*, 45(1):127–151, 1992.

[213] Iain A. Stewart. On completeness for NP via projection translations. *Mathematical Systems Theory*, 27(2):125–157, 1994.

[214] Iain A. Stewart. Complete problems for monotone NP. *Theoretical Computer Science*, 145(1&2):147–157, 1995.

[215] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theorertical Computer Science*, 3(1):1–22, 1976.

[216] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 1–9. ACM, 1973.

[217] Ofer Strichman and Stefan Szeider, editors. *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*. Springer, 2010.

[218] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with DPLL search. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.

[219] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 191–196. AAAI Press / The MIT Press, 2004.

[220] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: an implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In Hoos and Mitchell [126], pages 306–320.

[221] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2(115-125):10–13, 1968.

[222] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings London Mathematical Society*, 2(42):230–265, 1936.

[223] Peter van der Tak, Antonio Ramos, and Marijn Heule. Reusing the assignment trail in CDCL solvers. *Journal on Satisfiability*, 7(4):133–138, 2011.

[224] Moshe Y. Vardi. Boolean satisfiability: Theory and engineering. *Communications of the ACM, editor's letter*, 57(3):5, March 2014.

[225] Helmut Veith. Succinct representation, leaf languages, and projection reductions. In Steven Homer and Jin-Yi Cai, editors, *Proceedings of the Eleveth Annual IEEE Conference on Computational Complexity, Philadelphia, Pennsylvania, USA, May 24-27, 1996*, pages 118–126. IEEE Computer Society, 1996.

[226] Helmut Veith. Languages represented by boolean formulas. *Inf. Process. Lett.*, 63(5):251–256, 1997.

[227] Helmut Veith. How to encode a logical structure by an OBDD. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity, Buffalo, New York, USA, June 15-18, 1998*, pages 122–131. IEEE Computer Society, 1998.

[228] Helmut Veith. Succinct representation, leaf languages, and projection reductions. *Information and Computation*, 142(2):207–236, 1998.

[229] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.

[230] Klaus W. Wagner. The complexity of combinatorial problems with succinct input representation. *Acta Informatica*, 23(3):325–356, 1986.

[231] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman, 2002.

[232] Ralf Wimmer, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker. Preprocessing for DQBF. In Heule and Weaver [124], pages 173–190.

[233] Christoph M. Wintersteiger. *Termination Analysis for Bit-Vector Programs*. PhD thesis, ETH Zurich, Switzerland, 2011.

[234] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. In Bloem and Sharygina [37], pages 239–246.

[235] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In Alan Mycroft, editor, *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 1995.

[236] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-based verification*. Springer, 2006.

[237] Hantao Zhang. SATO: an efficient propositional prover. In William McCune, editor, *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 1997.

[238] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2002, San Jose, California, USA, November 10-14, 2002*, pages 442–449. ACM / IEEE Computer Society, 2002.

[239] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2002.

Appendix

Brief Biography

Personal Details

Name:	Andreas Fröhlich
Private Address:	Heckenweg 25, 86739 Ederheim, Germany
Date and Place of Birth:	16th June 1983 in Nördlingen, Germany

Research

Since Sep. 2011	Research and Teaching Assistant at the Institute for Formal Models and Verification (FMV), Johannes Kepler Universität Linz, Austria. Research Interests: Bit-Vectors, SAT, SMT, QBF, DQBF, Formal Verification, Local Search. Website: `http://fmv.jku.at/froehlich/`
Feb. 2014 – Apr. 2014	Internship at Microsoft Research, Cambridge, UK.

Education

Since Sep. 2011	Doctorate Computer Science, Johannes Kepler Universität Linz, Austria
Sep. 2002 – Jul. 2010	Diplom Computer Science, Universität Ulm, Germany
Sep. 1993 – Jun. 2002	Theodor Heuss Gymnasium Nördlingen, Germany
Sep. 1989 – Jul. 1993	Primary School in Ederheim, Germany