Computational Complexity and Solving Approaches for Bit-Vector Reasoning

Habilitation dissertation

Gergely Kovásznai

Faculty of Informatics
University of Debrecen
Debrecen, Hungary
2018
ACKNOWLEDGEMENTS

I would like to thank to my parents for supporting me in several ways and, in particular, in becoming a computer scientist. Thanks for buying me my first computer, a Videoton TV-Computer, in my childhood. This was an excellent choice for getting the first juicy taste of programming.

I would also like to thank to Melinda for being my partner in everything, including moving together to Linz and then to Vienna. Further thanks to our cats, Cecil and Momo, for cheering me up every day.

I am very thankful to Armin for showing me a real example of full dedication to research and, at the same time, being a highly supportive and cool person. I am very grateful for the possibility of working with Helmut and I feel so sad that he passed away so suddenly. Further thanks to Andreas for sharing great ideas and an office, and for being a good learner in keeping houseplants alive.
9.7.1 Table: Comparison of Completeness Results for Fixed-Size and Non-Fixed-Size Bit-Vector Logics 77
9.7.2 Example: A reduction of QBF to $\text{QF}_{\text{BV2}} \lessgtr 1$ 77

10 QUANTIFIER-FREE BIT-VECTOR FORMULAS: BENCHMARK DESCRIPTION 81
10.1 Introduction 82
10.2 Benchmarks 82
10.2.1 Translating Bit-Vector Operations 82
10.2.2 Bit-Vector Properties in PSPACE 83
10.3 SMT2 and CNF generation 83
10.4 Practical Considerations 83

11 COMPLEXITY OF FIXED-SIZE BIT-VECTOR LOGICS 85
11.1 Introduction 86
11.2 Motivation 87
11.3 Preliminaries 88
11.3.1 SAT, QBF, and DQBF 88
11.3.2 Circuits 89
11.3.3 Fixed-Size Bit-Vector Logics 90
11.4 Logics With Unary Encoding 98
11.5 Scalar-Bounded Problems 99
11.6 Quantifier-Free Logics with Binary Encoding 99
11.7 Fragment Extensions and Alternative Characterizations 107
11.7.1 Notation 108
11.7.2 $\text{QF}_{\text{BV2}}_{\text{bw}}$ 108
11.7.3 $\text{QF}_{\text{BV2}} \lessgtr 1$ 110
11.7.4 $\text{QF}_{\text{BV2}} \lessgtr c$ 112
11.8 Logics with Quantifiers and Binary Encoding 116
11.8.1 General Quantification 116
11.8.2 Restricting the Bit-Width of Universal Variables 118
11.8.3 Non-Recursive Macros 120
11.9 Practical Considerations 121
11.9.1 Alternative Approaches 121
11.9.2 Benchmark Problems 122
11.10 Conclusion 122
11.11 Appendix 124
11.11.1 Example: A Reduction of a DQBF to $\text{QF}_{\text{BV2}} \lessgtr c$ 124
11.11.2 Example: A Reduction of a QBF to $\text{QF}_{\text{BV2}} \lessgtr 1$ 125
11.11.3 Example: Bit-Width Reduction of a $\text{QF}_{\text{BV2bw}}$ Formula with Indexing and Relational Operations 127
11.11.4 Example: Half-Shuffle and Expand Applied to a Bit-Vector 127
11.11.5 Example: Multiplication of Two Bit-Vectors 128

12 COMPLEXITY OF VERIFICATION AND DECISION PROBLEMS IN BIT-VECTOR LOGIC 131
12.1 Introduction 132
12.2 Preliminaries 134
12.3 Bit-Vector Logic 135
12.4 Motivating Example: Word-Level Model Checking 136
12.5 Bit-Vector Representation of Problems 137
12.6 Lifting Hardness 138
12.7 Conclusion 139
12.8 Appendix 140
12.8.1 Common Bit-Vector Operators 140
### Contents

12.8.2 Additional Proofs 143  
12.8.3 Definability of Bit-Vector Fragments 146  

13 **BV2EPR: TRANSLATING QUANTIFIER-FREE BIT-VECTOR FORMULAS INTO EPR** 147  
13.1 Introduction 148  
13.2 Preliminaries 148  
13.2.1 Existing Translations 149  
13.3 The Tool 149  
13.3.1 The Translator 150  
13.4 Benchmarks and Experiments 151  
13.5 Conclusion 152  

14 **EFFICIENTLY SOLVING BIT-VECTOR PROBLEMS USING MODEL CHECKERS** 155  
14.1 Introduction 156  
14.2 QF_BV_{≤1} to SMV 157  
14.3 Experiments 159  
14.4 Conclusion 162  

15 **A DPLL ALGORITHM FOR SOLVING DQBF** 165  
15.1 Introduction 166  
15.2 Definitions 167  
15.3 DQDPLL Architecture 168  
15.4 Conversion of Concepts from SAT/QBF 171  
15.5 Preliminary Results 174  
15.6 Future Work 175  
15.7 Conclusion 175  

16 **IDQ: INSTANTIATION-BASED DQBF SOLVING.** 177  
16.1 Introduction 178  
16.2 Preliminaries 179  
16.3 Related Work 181  
16.4 IDQ architecture 181  
16.5 Implementation 184  
16.6 Experimental Results 186  
16.7 Conclusion 188
Part I

THESSES
INTRODUCTION
We develop programs that read other programs in order to find logical mistakes in them. If you like, our programs do a psychoanalysis on programs. But this leads to logical contradictions that are already known from the time of Aristotle: Who guarantees that it is not the psychiatrist who is crazy?

Helmut Veith (1971-2016)  
a quote from an interview on Vienna Summer of Logic 2014

The static verification of hardware and software is an essential tool for avoiding errors and threats in digital circuits, source code, IT systems, etc., and to ensure that every expected requirements are fulfilled. Satisfiability Modulo Theories (SMT) and, in particular, bit-precise reasoning over bit-vector logics are the cornerstones of such verification tasks. There exist a lot of state-of-the-art SMT solvers with support for bit-vector logics, and they are widely used in industry as well. There are a lot of open issues though which require further research, the invention of solving approaches and the development of actual solvers.

Although the computational complexity of a certain logic is a theoretical question, in the case of bit-vector logics it is crucial in practice to know the answer. Those logics are indeed applied in practice and there exist a couple of solving approaches. Knowing the computational complexity can also help to find new, promising approaches.

Interestingly, the computation complexity of bit-vector logics had not been a deeply-researched area of computer science before. Nevertheless, there exist a couple of related scientific works, but some of them make statements that do not hold in general. All the afore-mentioned reasons motivated me to try to investigate the exact computation complexity of common bit-vector logics, that is, to find out if they are complete for any complexity classes or they are not. In this dissertation, I give a collection of own papers that propose corresponding new results.

Later, by knowing the complexity of certain bit-vector logics, I started to investigate new solving approaches for bit-vector logics of common interest. The first task is to choose a “target” logic that is complete for the same complexity class and provides efficient solving approaches, and, preferably, actual existing solvers. The second task is to invent a polynomial reduction to that “target” logic from the bit-vector logic. Finally, use the existing solver of the “target” logic to solve bit-vector problems. In a few papers of mine, such reductions to common logics, such as EPR, are proposed and experiments with solvers are reported. A “target” logic that interests me in particular is the Dependency Quantified Boolean Formulas (DQBF), for which I and my co-authors were pioneers in inventing solving approaches.

In Chapter 2, I give the necessary introduction and preliminaries into SAT solving, QBF, DQBF, EPR, SMT and bit-vector logics. Chapter 3 discusses the computational complexity of common bit-vector logics and some fragments of practical interest. In Chapter 4, I give details on the reductions I invented to certain “target” logics. Chapter 5 introduces the DQBF solving approaches we invented. In Chapter 6, I am going to give a summary on the citations that our papers have received over the years. Finally, in Chapter 7, I am going to give a list of my most important scientific achievements.
PRELIMINARIES
2.1 SATISFIABILITY CHECKING AND SAT

In computer science, satisfiability can be considered to be one of the most fundamental questions to ask: given a formal description of a statement, also called a formula, does there exist a model (or interpretation) for the syntactical elements in the formula such that the formula is true. The formula is considered to be satisfiable (SAT) if such a model exists, otherwise it is considered to be unsatisfiable (UNSAT).

In real life, for instance in industrial use cases, satisfiability checking and model finding is an extremely important tool for verifying systems. Given a system described as a formula $S$ and a (safety) condition $C$ to check on the system, one might want to check if $C$ always holds for $S$, under any circumstances, i.e., under any models. This check can be done by checking the satisfiability of $S \land \neg C$, where $\neg$ denotes logical not or negation, and $\land$ logical and or conjunction. Similar Boolean operators are $\lor$ as logical or or disjunction, $\Rightarrow$ as implication, $\Leftrightarrow$ as equivalence, etc. If $S \land \neg C$ is satisfiable, then there exists a model, which gives us the exact circumstances in the system $S$ under which the condition $C$ is violated. This makes model finding an excellent tool for debugging.

Another example could be equivalence checking in hardware industry. Given an original circuit design described as a formula $D_1$, let us suppose that engineers do some optimization and get a new design $D_2$. It is important to check if the new design provides the same functionality as the old one. For this, the satisfiability of the formula $D_1 \Leftrightarrow D_2$ can be checked.

It is a matter of the logic we choose, what syntactical elements build up a formula and what semantical rules to follow for evaluating a formula. The most simple logic is called the Boolean logic, also known as propositional logic, where syntactical elements are the (Boolean) variables and the model is an assignment of values to those variables. A value can be either false or true, or alternatively, 0 or 1.

Definitions can be given as follows. 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) $\neg \phi_0$, $(\phi_0 \land \phi_1)$, $(\phi_0 \lor \phi_1)$, $(\phi_0 \Rightarrow \phi_1)$, and $(\phi_0 \Leftrightarrow \phi_1)$ are Boolean formulas where $\phi_0, \phi_1$ are Boolean formulas. A Boolean formula $\phi$ is satisfiable iff there exists an assignment $\alpha : V \rightarrow \{0, 1\}$ to the variables, such that $\phi$ evaluates to 1 under $\alpha$.

The standard normal form for Boolean formulas is the Conjunctive Normal Form (CNF). A formula is said to be in CNF if it is conjunction of clauses. A clause is a disjunction of literals, where a literal is defined as a variable or the negation of a variable. The SAT problem is usually meant the satisfiability checking of Boolean formulas in CNF.

Although Boolean logic seems extremely simple, the computation complexity of SAT is very high. In fact, SAT was the first computational problem that was shown to be NP-complete by encoding any polynomial time-bounded non-deterministic Turing machine as a SAT instance [Coo71]. Assuming $P \neq NP$, SAT cannot be solved by a polynomial time (deterministic) algorithm in general.

Due to combinatorial explosion, naive SAT solving approaches might already fail for small formulas with a few hundreds of variables. Therefore, for a long time, it seemed that SAT solving was computationally intractable in practice. However, with the advent of heuristic SAT solvers and, especially, of the DPLL-based solvers that apply conflict-driven clause learning (CDCL), state-of-the-art SAT solvers are able to solver huge formulas with several million of variables. Formulas of such extent are sufficient for encoding industrial problems and, therefore, modern SAT solvers are widely used in industry.
2.2 QBF and DQBF

SAT can naturally be extended by using quantifiers \( \forall \) and \( \exists \). By applying quantification, the semantics dramatically changes. Consider the quantifier-free formula \((x_1 \lor x_2) \land (\neg x_1 \lor x_2)\), which is satisfiable since there exists values for \( x_1 \) and \( x_2 \) such that the formula evaluates to true. What happens if we add quantifiers to the formula and get \( \exists x_1 \forall x_2 \cdot (x_1 \lor x_2) \land (\neg x_1 \lor x_2) \)? This formula is unsatisfiable since no value for \( x_1 \) exists which makes the formula true for all values for \( x_2 \). \( \exists \) and \( \forall \) are called the existential and universal quantifiers, respectively. Note that SAT can be considered to be a special case when all the variables are existentially quantified.

The class of Quantified Boolean Formulas (QBF) is obtained by adding quantifiers to Boolean formulas and is defined as

\[
Q_1 x_1 \ldots Q_n x_n \cdot \phi
\]

where \( Q_i \in \{\forall, \exists\} \) are quantifiers, \( x_j \in V \) are distinct variables, and \( \phi \) is a (quantifier-free) Boolean formula in CNF over the variables \( x_1, \ldots, x_n \). We call \( Q_1 x_1 \ldots Q_n x_n \) the quantifier prefix, and \( \phi \) the matrix.

A variable \( x_i \) depends on a variable \( x_j \) iff \( i > j \). This defines a total order on the variables of a QBF. A QBF is satisfiable iff there exist Skolem functions for its existential variables such that the matrix \( \phi \) is satisfied by all possible assignments to the universal variables. The computational complexity of the satisfiability problem for QBF is higher than that for SAT. QBF can be proved to be \( \text{PSPACE}\)-complete by applying Savitch’s theorem for encoding the graph reachability problem as a QBF [Pap94]; [SM73]. There exist several practical QBF solvers, based on different approaches. One of these approaches is the extension of DPLL and is called DQPLL [CGS98].

Instead of using totally ordered quantifiers, it is also possible to extend Boolean formulas with Henkin quantifiers [Hen61]. Henkin quantifiers specify variable dependencies explicitly instead of using implicit dependencies defined by the quantifier order. Adding Henkin quantifiers to Boolean formulas results in the class of Dependency Quantified Boolean Formulas (DQBF) [PR79], which can be defined as

\[
\forall u_1 \ldots \forall u_m \exists e_1(u_{1,1}, \ldots, u_{1,k_1}) \ldots \exists e_n(u_{n,1}, \ldots, u_{n,k_n}) \cdot \phi
\]

where \( \phi \) is a Boolean formula in CNF over the variables \( u_1, \ldots, u_m, e_1, \ldots, e_n \). The formalism \( e_i(u_{i,1}, \ldots, u_{i,k}) \) means that the existential variable \( e_i \) depends only on the universal variables \( u_{i,1}, \ldots, u_{i,k} \). We use \( \text{dep}_{e_i} \) to denote \( e_i \)'s dependency set.

Note that in DQBF the dependencies of existential variables are always explicitly given, in contrast to QBF where an existential variable depends on all the universal variables to the left in the quantifier prefix. Thus, QBF can be considered as a special case of DQBF, where for all \( Q_i = \exists \) it holds that \( \text{dep}_{x_i} = \{x_j \mid 1 \leq j < i, \ Q_j = \forall\} \). While in QBF the dependencies of the existential variables induce linear ordering, in DQBF this is not always the case.

The more general quantifier order makes DQBF more powerful than QBF and allows more succinct encodings. The satisfiability problem for DQBF is \( \text{NEXPTime-complete} \) [PRA01]; [PR79]. Our approach called DQDPLL [FKB12] was the very first implementation of a dedicated DQBF solver. There exists other solving approaches [FT14]; [Git+15]; [Rab17], including our instantiation-based approach iDQ [Frö+14], which is currently the only publicly available DQBF solver.
2.3 PREDICATE LOGIC AND EPR

Predicate logic, also known as first-order logic, takes abstraction to a new level, compared to Boolean logic and its quantified variants QBF and DQBF. Predicate logic uses quantified variables over objects from any domain and, furthermore, allows to introduce function symbols over them. Functions that return Boolean values are called predicates. Common logical operators, such as negation, conjunction or disjunction, are applied to atoms in the form \( p(t_1, \ldots, t_n) \) where \( p \) is a predicate symbol and each \( t_i \) is either a variable or a function symbol with arguments. As it can be expected, predicate logic is much more expressive than Boolean logic.

Similar to DQBF, the common normal form for predicate logic formulas is prenex CNF where each existentially quantified variable is eliminated by Skolemization, thus, the quantifier prefix consists only of universal quantifiers and the matrix is in CNF.

A formula in predicate logic is satisfiable iff there exist functions for all its function symbols such that the matrix is satisfied by all possible assignments to the universal variables over any domain. Alonzo Church and Alan Turing proved the satisfiability problem for predicate logic to be undecidable.

The Effectively Propositional Logic (EPR), also known as the Bernays-Schönfinkel class, is a decidable and NExpTime-complete fragment of predicate logic [Lew80]. EPR formulas have a \( \exists^*\forall^* \) quantifier prefix and contain function symbols only with arity 0, also known as constants. By Skolemization, similar to DQBF, existential variables can be eliminated by introducing new constants. This basically means that functions do not call functions, which makes the semantical evaluation of EPR formulas relatively simple.

Although any theorem prover for predicate logic can solve EPR formulas, the dedicated EPR solver \texttt{iProver} [Kor08] usually wins the EPR track of the CASC competition\(^1\). \texttt{iProver} applies an instantiation-based approach called the \textit{Inst-Gen calculus} [Kor09]; [Kor13].

\(^{1}\text{http://www.cs.miami.edu/~tptp/CASC/J8/WWWFiles/Results.html#EPRProblems} \)
2.4 SMT AND BIT-VECTOR LOGICS

It is a fairly natural idea to extend SAT solving with background theories such as integer or real arithmetic, or arrays. Needless to say that such an extension would have clear practical value since it would let our logical formulas contain atoms which, for instance, might evaluate some arithmetic expression over numbers or might check the value of array elements. Satisfiability Modulo Theories (SMT) is the decision problem of satisfiability checking of Boolean formulas with respect to some background theory and logic. The most common examples of theories are the integer numbers, the real numbers, the fixed-size bit-vectors, and the arrays. The logics that one could use might differ from each other in the linearity or non-linearity of arithmetic, the presence or absence of quantifiers, or in the presence or absence of uninterpreted functions. The SMT-LIB format [BFT15], as the common input format for SMT solvers, defines the syntax for several such logics 2, such as QF_UFLIA as the quantifier-free logic of linear integer arithmetic with uninterpreted functions, or LRA as the logic of linear real arithmetic allowing quantification, or AUFLIA as the logic of linear integer arithmetic with quantifiers, uninterpreted functions and arrays.

In most of our papers in this thesis, we are focusing on the background theory of fixed-size bit-vectors, also known as words or sequences of bits, i.e., Boolean values. The fundamental building blocks of bit-vector formulas are the bit-vector variables \( x^{[n]} \) and constants \( c^{[n]} \) of certain bit-widths \( n \). To those variables and constants, different kinds of bit-vector operators can be applied, such as bitwise operators, arithmetic operators, relational operators, shifts, rotations, extensions, etc. As a contribution, we defined the syntax and the semantics of those bit-vector logics in a precise and unified way in our TOCS paper [KFB16]. Table 2.1 from [KFB16] shows the syntax of the most common operators provided by the SMT-LIB format [BST10] and the literature [BDL98]; [BP98]; [BS09]; [CMR97]; [Fra10], 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. For a detailed introduction into the semantics of those bit-vector logics, we recommend the relevant parts of our TOCS paper [KFB16].

<table>
<thead>
<tr>
<th>operation</th>
<th>condition</th>
<th>bit-width</th>
<th>alternative syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation:</td>
<td>( \text{bnot} \ t^{[n]} )</td>
<td>( n )</td>
<td>( \sim t^{[n]} )</td>
</tr>
<tr>
<td>and:</td>
<td>( \text{band} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>( n )</td>
<td>( t_1^{[n]} \land t_2^{[n]} )</td>
</tr>
<tr>
<td>or:</td>
<td>( \text{bor} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>( n )</td>
<td>( t_1^{[n]} \lor t_2^{[n]} )</td>
</tr>
<tr>
<td>xor:</td>
<td>( \text{bxor} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>( n )</td>
<td>( t_1^{[n]} \oplus t_2^{[n]} )</td>
</tr>
<tr>
<td>nand:</td>
<td>( \text{bnot} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>( n )</td>
<td></td>
</tr>
<tr>
<td>nor:</td>
<td>( \text{bnot} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>( n )</td>
<td></td>
</tr>
<tr>
<td>xnor:</td>
<td>( \text{bxor} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>( n )</td>
<td></td>
</tr>
<tr>
<td>if-then-else:</td>
<td>( \text{ite} \ t_1^{[n]}, t_2^{[n]}, t_3^{[n]} )</td>
<td>( n )</td>
<td></td>
</tr>
<tr>
<td>equality:</td>
<td>( \text{bvcomp} \ t_1^{[n]}, t_2^{[n]} )</td>
<td>1</td>
<td>( t_1^{[n]} = t_2^{[n]} )</td>
</tr>
</tbody>
</table>

continued on next page

2http://smtlib.cs.uiowa.edu/logics.shtml
<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned (u.) less than</td>
<td><code>bult (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>u. less than or equal</td>
<td><code>bule (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>u. greater than</td>
<td><code>buge (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>u. greater than or equal</td>
<td><code>buge (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>signed (s.) less than</td>
<td><code>bslt (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>s. less than or equal</td>
<td><code>bsle (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>s. greater than</td>
<td><code>bsgt (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>s. greater than or equal</td>
<td><code>bsge (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td>shift left</td>
<td><code>bslt (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>logical shift right</td>
<td><code>bshr (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>arithmetic shift right</td>
<td><code>bshr (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>extraction</td>
<td><code>extract (t^{[n]}, i, j)</code></td>
<td><code>n &gt; i ≥ j</code></td>
<td><code>i - j + 1</code></td>
</tr>
<tr>
<td>concatenation</td>
<td><code>concat (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>m + n</code></td>
</tr>
<tr>
<td>zero extend</td>
<td><code>zeroextend (t^{[n]}, i)</code></td>
<td></td>
<td><code>n + i</code></td>
</tr>
<tr>
<td>sign extend</td>
<td><code>signextend (t^{[n]}, i)</code></td>
<td></td>
<td><code>n + i</code></td>
</tr>
<tr>
<td>rotate left</td>
<td><code>rotate_left (t^{[n]}, i)</code></td>
<td><code>n &gt; i ≥ 0</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td>rotate right</td>
<td><code>rotate_right (t^{[n]}, i)</code></td>
<td><code>n &gt; i ≥ 0</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td>repeat</td>
<td><code>repeat (t^{[n]}, i)</code></td>
<td><code>i &gt; 0</code></td>
<td><code>n - i</code></td>
</tr>
<tr>
<td>unary minus</td>
<td><code>buneg (t^{[n]})</code></td>
<td></td>
<td><code>-t^{[n]}</code></td>
</tr>
<tr>
<td>addition</td>
<td><code>bueadd (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>t_1^{[n]} + t_2^{[n]}</code></td>
</tr>
<tr>
<td>subtraction</td>
<td><code>buesub (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>t_1^{[n]} - t_2^{[n]}</code></td>
</tr>
<tr>
<td>multiplication</td>
<td><code>bumeul (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>t_1^{[n]} \cdot t_2^{[n]}</code></td>
</tr>
<tr>
<td>unsigned division</td>
<td><code>budeiv (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>t_1^{[n]} / u t_2^{[n]}</code></td>
</tr>
<tr>
<td>u. remainder</td>
<td><code>burem (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>signed division</td>
<td><code>busdiv (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>s. remainder</td>
<td><code>burem (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>s. remainder with rounding to +</td>
<td><code>burems (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
<tr>
<td>s. remainder with rounding to -</td>
<td><code>buserm (t_1^{[n]}, t_2^{[n]})</code></td>
<td></td>
<td><code>n</code></td>
</tr>
</tbody>
</table>

Table 2.1: Syntax (signature) for common bit-vector operators [KFB16]

QF_BV denotes the quantifier-free logic of bit-vectors. By adding uninterpreted functions to this logic, i.e., allowing to introduce custom signatures of function symbols on demand, we get the
logic of QF_UFBV. When quantification is introduced, we get the logics BV and UFBV, depending on whether uninterpreted functions are allowed to use.

Bit-vector logics play an important role in many practical applications of computer science, most prominently in hardware and software verification, due to the fact that every piece of data in hardware or software has a given bit-width. In hardware verification, the quantifier-free bit-vector logics QF_BV and QF_UFBV are commonly used in practice, while quantified bit-vector logics BV and UFBV are preferably applied in software verification.

Compared to other theories, bit-vector logics can be considered to be the closest to Boolean logic. A bit-vector formula can always be directly translated into a Boolean formula by using the circuit representation of bit-vector operations, as realized in hardware. This approach is called bit-blasting and used by most state-of-the-art bit-vector solvers, which then feed the resulting Boolean formula into a SAT solver.

The computational complexity of bit-blasting for the common bit-vector logics had not been clear for long time. This is what we intended to investigate in most of our papers, for the sake of proving the membership of bit-vector logics in certain complexity classes. It was even more difficult to prove their hardness to those complexity classes, for the sake of investigating the precise characterization of the computation complexity of bit-vector logics.
COMPLEXITY OF BIT-VECTOR LOGICS
Although the computational complexity of a certain logic is a theoretical question, in the case of bit-vector logics it is crucial in practice to know the answer. Those logics are indeed applied in practice and there exist a lot of solving approaches. Knowing the computational complexity can also help to find new, promising approaches.

The vast majority of bit-vector solvers rely on bit-blasting. This is a technique to translate a bit-vector formula to a Boolean formula whose Boolean variables represent the individual bits of the bit-vectors. Bit-blasting is known to be a polynomial reduction in the bit-width of the bit-vectors (regarding the commonly used bit-vector operators). Therefore it seems logical to say that bit-blasting is polynomial, and thus the satisfiability problem of a bit-vector logic is in the same complexity class as the underlying Boolean satisfiability problem. For instance, QF_BV should be in NP since SAT is in NP.

I remember the exact moment when Prof. Armin Biere was telling an exciting story about a quite difficult discussion he witnessed in the program committee of FMCAD 2010. One of the PC members had serious objections against one of the papers that had received positive reviews and, therefore, was about to be accepted at the conference. That particular PC member tried to convince the others that the proof of one of the theorems in the paper was not correct. The proof used the commonly accepted belief of bit-blasting being polynomial (in the size of the input) and showed that BV was NExT-Time-complete. His argument was based on the fact that the bit-widths were encoded as decimal numbers in the input formula, i.e., they employed exponentially succinct encoding, and, therefore, bit-blasting could be exponential. He was not able to convince the PC and the paper was accepted [WHM 10].

This story that Armin told us did not let my brain stop. I started to analyze the problem and was pretty soon convinced that bit-blasting was not always polynomial. With my colleague, Andreas Fröhlich, we started to try to prove that certain bit-vector logics are “harder” than assumed before by the scientific community. First, we focused on the quantifier-free bit-vector logic QF_BV. We spent quite some time to prove that QF_BV is NExT-Time-hard [KFB12]. That proof is one of my most important scientific contributions and is cited by numerous publications. Note that our results highlight that the claim in [Bry+07] about QF_BV being NP-complete does not hold in general, but only if the bit-widths of bit-vectors are encoded in unary format.

Pretty soon we could also prove that the quantified bit-vector logic UFBV is 2-NExT-Time-hard [KFB12]. This result shows that the claim in [WHM10]; [Win11] about UFBV being NExT-Time-complete, similarly to the result in [Bry+07], only holds if we assume unary encoded bit-widths.

In the subsequent years, we published numerous further complexity results on bit-vector logics. Those results came from two different directions:

1. Searching for minimal fragments of those logics that are complete for certain complexity classes [FKB13b]; [KFB16]: Such investigations are valuable because they show the exact causes of why the complexity of a certain logic increases or decreases and, more importantly, they suggest solving approaches for those fragments [KFB13a]; [FKB13a].

2. Generalizing our complexity results for any decision problem and any encoding of scalars in bit-vector formulas [Kov+14]: By using those generic theorems of ours, the completeness of any decision problem, such as the reachability problem (in model checking) or the circuit value problem, can be easily determined no matter how succinct encoding of scalars we use.
3.1 Complexity of Common Bit-Vector Logics

We showed for the first time that the commonly used bit-vector logics have higher complexity in general than the verification community had thought before [KFB12]. The higher complexity is due to the exponentially succinct, logarithmic encoding used in practice to represent the bit-widths of bit-vectors in the input formulas. In the paper, we focused only on the theory of fixed-sized bit-vector formulas.

The introduction of the paper cites the previously mentioned paper [WHM10] and claims that its proof of UFBV being $\text{NExpTime}$-complete only holds if bit-widths are encoded in unary form, which is, of course, not the encoding used in practice. The main goal of our paper is to investigate how complexity varies if we consider a logarithmic w.l.o.g. binary encoding. The paper shows that the binary encoding of bit-widths has at least as much expressive power as quantification.

Table 3.1 summarizes our new complexity results in this paper [KFB12], complemented by a result provided later by [JS16a].

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Quantifiers</th>
<th>Uninterpreted Functions</th>
<th>Uninterpreted Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary</td>
<td>no</td>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>binary</td>
<td>yes</td>
<td>PSPACE</td>
<td>$\text{NExpTime}$</td>
</tr>
</tbody>
</table>

Table 3.1: Completeness results for common bit-vector logics [KFB12]

From our complexity results it follows that BV is $\text{NExpTime}$-hard and is in $\text{ExpSpace}$, but we have never been able to prove if BV is complete for any of the complexity classes. Finally in 2016, Jonáš and Strejček proved that BV is complete for $\text{AExp(poly)}$ in [JS16a], as shown in Table 3.1.

The main contribution of the paper [KFB12] is to prove that $\text{QF}_\text{BV}$ with binary encoding is $\text{NExpTime}$-hard. For this, we picked an $\text{NExpTime}$-hard decision problem, the satisfiability problem of DQBF, and gave a polynomial reduction from it to $\text{QF}_\text{BV}$. The proof cannot be done in a trivial way since the exponential many bits of bit-vectors should be somehow handled in a polynomial way. For this, we need to split the bit-vectors of bit-width $2^n$ into polynomial many chunks of exponential size. I realized that this could be done by applying the following special bitmasks, also known as the binary magic numbers:

$$
\begin{align*}
\begin{array}{cccc}
0 & \ldots & 0 & \ldots & 1 \\
2^i & & & & 1 \\
& & & & \\
& & & & \\
& & 2^i & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
\end{array}
\end{align*}
$$

I discovered that those bit-mask could be “calculated” by the following bit-vector expression of polynomial size:

$$
M_i^n := 0^{[2^n]} / u \left( (1 \ll (1 \ll i)) + 1 \right)
$$

The crucial part of the proof is to represent each universally resp. existentially quantified DQBF variable $u$ resp. $e$ as a bit-vector variable $U^{[2^n]}$ resp. $E^{[2^n]}$, and, more importantly, to specify certain constraints on $U$ and $E$ to make them respect the dependencies between $u$ and $e$.

The $i$th universal variable $u_i$ is “defined” by the corresponding binary magic number:

$$
U_i = M_i^n
$$
The independence of \( e \) on \( u_i \) is represented by the following constraint:

\[
(E \& U_i) = ((E \gg_u (1 \ll i)) \& U_i)
\]

The latter constraint can be considered as the core of the proof and might not be easy to understand. Basically bit-segments of size \( 2^i \) are made equal to each other if they correspond to \( u_i \), i.e., the values of those bits do not depend on the value of \( u_i \).

Another important contribution of the paper is to prove that UFBV with binary encoding is 2-NExpTime-hard. The 2-NExpTime-hard decision problem we reduced to UFBV was the \( 2^{(2^n)} \)-square tiling problem. The \( f(n) \)-square tiling problem is about to place dominoes on an \( f(n) \times f(n) \) board, respecting certain horizontal and vertical matching conditions \( H \) and \( V \), respectively. Any instance of the \( 2^{(2^n)} \)-square tiling problem can be expressed as a UFBV formula:

\[
\lambda(0,0) = 0 \land \lambda\left(2^{(2^n)} - 1, 2^{(2^n)} - 1\right) = k - 1 \\
\land \bigwedge_{(t_1,t_2) \in H} h(t_1,t_2) \land \bigwedge_{(t_1,t_2) \in V} v(t_1,t_2) \\
\land \forall i, j \in [2^n] \left( i < 2^{(2^n)} - 1 \Rightarrow h(\lambda(i,j), \lambda(i,j+1)) \right) \land \\
\land \forall i, j \in [2^n] \left( j < 2^{(2^n)} - 1 \Rightarrow v(\lambda(i,j), \lambda(i+1,j)) \right)
\]

The formula contains two universally quantified bit-vector variables, \( i \) and \( j \). The uninterpreted function \( \lambda(i,j) \) represents the type of the tile in the cell at row index \( i \) and column index \( j \).

It is crucial to see that although the formula contains exponential bit-widths \( 2^n \), they are encoded as binary numbers, i.e., by using \( n \) digits. Furthermore, the double-exponential scalars \( 2^{(2^n)} - 1 \) can be represented as \( \sim 0^{2^n} \). Thus, we gave a polynomial reduction of the \( 2^{(2^n)} \)-square tiling problem to UFBV.

Last but not least, the paper defines a practically reasonable condition, called the \textit{bit-width boundedness}, which if holds, then the encoding of the bit-widths has not effect on the computational complexity. For bit-width bounded formula sets, the complexity is the one that we proved for the unary case, e.g., NP for QF_BV and QF_UFBV, PSPACE for BV, and NExpTime for UFBV.
3.2 COMPLEXITY OF FRAGMENTS OF BIT-VECTOR LOGICS

Two follow-up papers [FKB13b], [KFB16] propose new computational complexity results on certain fragments of the common bit-vector logics, on QF_BV in particular. After we had proved in [KFB12] that QF_BV was NE\textsuperscript{XP}T\textsuperscript{ime}-complete, the question arose: Are there any practically reasonable fragments of QF_BV which have lower complexity? Let us remember that [KFB12] had already proposed such a fragment of bit-width bounded formula sets, which is in NP, similar to a more general fragment called scalar-bounded formula sets [KFB16].

We investigated how the set of bit-vector operators used in formulas affected computational complexity. We defined three fragments:

- \texttt{QF\textsubscript{BV}{$\ll$}c}: only bitwise operators, equality, and shift by any constant \(c\) are allowed;
- \texttt{QF\textsubscript{BV}{$\ll$}1}: only bitwise operators, equality, and shift by \(c = 1\) are allowed;
- \texttt{QF\textsubscript{BV}bw}: only bitwise operators and equality are allowed.

We proved all those fragments to be complete for certain complexity classes, as shown in Table 3.3. Note that we also address non-fixed-sized bit-vector logics.

<table>
<thead>
<tr>
<th></th>
<th>fixed-size</th>
<th>non-fixed-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF\textsubscript{BV}</td>
<td>NExpTime [KFB12]</td>
<td>undecidable [DMR76]</td>
</tr>
<tr>
<td>QF\textsubscript{BV}{$\ll$}c</td>
<td>[]</td>
<td>?</td>
</tr>
<tr>
<td>QF\textsubscript{BV}{$\ll$}1</td>
<td>PSpace [*]</td>
<td>PSpace [SK12b]; [SK12a]</td>
</tr>
<tr>
<td>QF\textsubscript{BV}bw</td>
<td>NP [*]</td>
<td>NP [*]</td>
</tr>
</tbody>
</table>

(* marks our new results)

Table 3.3: Completeness results for fragments of bit-vector logics [FKB13b]; [KFB16]

The NExpTime-completeness of QF\textsubscript{BV}{$\ll$}c directly follows from the proof in [KFB12], since the reduction we gave in that proof used only bitwise operations, equality and shift by constant. Note that in [FKB13b] we eliminated the division in (3.1) by rewriting (3.2) to

\[
\left( (U_i \ll (1 \ll i)) + U_i \right) = \sim 0[2^k]
\]

It is interesting that restricting shifts to be used only with \(c = 1\) causes the complexity to drop to PSpace-completeness, as being proved for QF\textsubscript{BV}{$\ll$}1 in [FKB13b]. Finally, if no shifts are allowed to use, the resulting fragment QF\textsubscript{BV}bw becomes NP-complete [FKB13b].

Our paper [KFB16] investigates possible extensions of the aforementioned fragments and their alternative characterizations. Speaking of QF\textsubscript{BV}bw, it turns out that the set of bitwise operations and equality can be extended by indexing and relational operations without pulling out the fragment from NP. In a similar manner, QF\textsubscript{BV}{$\ll$}1 stays in PSpace even if we extend the set of bitwise operations, equality, and left shift by 1 with any of the operations in Figure 3.1. It is even more interesting, as the figure shows, that the operations right shift by 1, addition, subtraction, and multiplication by constant can be used as alternative base operations instead of left shift by 1.

QF\textsubscript{BV}{$\ll$}c stays in NExpTime even if we extend bitwise operations, equality, and left shift by constant with any of the operations in Figure 3.2. Any of right shift by constant, extraction, concatenation, and multiplication can serve as an alternative base operation instead of left shift by constant. The most difficult proof in this section is about reducing multiplication to left shift by...
constant and vice versa. This proof uses several tools such as exponentiation by squaring, the binary magic numbers, the half-shuffle operation, and the shift-and-add algorithm.

[KFB16] proposes new complexity results for fragments of quantified bit-vector logics as well. We already proved in [KFB12] that UFBV is 2-NExpTime-complete, therefore the fragment UFBV_{≤c}, and all its alternative characterizations, have the same complexity. Interestingly, if we restrict shifts to be applied only by 1, the complexity does not change, as opposed to the quantifier-free case. That is, both UFBV_{≤c} and UFBV_{≤1} are 2-NExpTime-complete.

We also address two fragments that are important in practice and have something to do with quantification:

**bvlog**: In this fragment, the bit-width of the universally quantified variables must not exceed the logarithm of the bit-width of the existentially quantified variables. This fragment is of special practical interest since it relates to the theory of arrays. In practice, if an array is expressed as a bit-vector, array indices are of logarithmic bit-width and are often quantified universally. We proved that BV_{log} and UFBV_{log} are NExpTime-complete.

**qf UF BV**: In the SMT-LIB, non-recursive macros are basic tools. Such a macro provides an uninterpreted function and assigns a functional definition to it. We can formalize a QF_UFBV formula $\Phi$ with non-recursive macros as follows:

$$\forall u_0^{[n_0]}, \ldots, u_k^{[n_k]} . \quad \Phi$$

$$\land \ f_0^{[w_0]}(u_0, \ldots, u_{k_0}) = d_0^{[w_0]}$$

$$\land \ \ldots$$

$$\land \ f_m^{[w_m]}(u_0, \ldots, u_{k_m}) = d_m^{[w_m]}$$

Here, $f_0, \ldots, f_m$ are the macros as uninterpreted functions and $d_0, \ldots, d_m$ are their functional definitions as bit-vector terms. Note that the macros’ parameters are universally
3.2 Complexity of Fragments of Bit-Vector Logics

quantified variables and, therefore, the fragment $\text{QF}_{\text{UFBV}}_M$ is basically a quantified bit-vector logic. We proved however that using non-recursive macros does not increase the complexity of $\text{QF}_{\text{UFBV}}$, i.e., $\text{QF}_{\text{UFBV}}_M$ is NE$\text{xp}$-complete.

Remark 3.1. Although Figure 3.2 shows that left shift (by any term) can be reduced to left shift by constant, there is no specific proof on that in [KFB16]. Probably, when writing the paper, we did not feel necessary to give such a proof since this reduction could be done by applying the well-known technique of barrel shifting. Nevertheless, I would now like to fill this gap and to give an explicit reduction as follows.

Given the two operands $t_1$ and $t_2$ of bit-width $n$, the shift can be done in $L_n$ steps by using barrel shifting, where $L_n := \lceil \log_2 n \rceil$. In the $i$th step, we check the $i$th bit of $t_2$, and if it is 1 then we shift $t_1$ by $2^i$. This algorithm is precisely formalized as follows:

| $egin{aligned} t_1^{[n]} &\ll t_2^{[n]} \\ \text{replace with: } & \text{ite} \left( y_{0} < n , x_{L_n} , 0^{[n]} \right) \\ \text{add assertions: } & x_0^{[n]} = t_1 \\ & y_0^{[n]} = t_2 \\ & x_i^{[n]} = \text{ite} \left( y_{i-1} \& 1^{[n]} = 0^{[n]} , x_{i-1} , x_{i-1} \ll 2^{i-1} \right) , 0 \leq i \leq L_n \\ & y_i^{[n]} = y_{i-1} \gg 1 , 0 \leq i < L_n \end{aligned}$ |

Note that we do not apply indexing for checking the $i$th bit of $t_2$, but rather in each step we do logical right shift by 1 and apply the bit mask $1^{[n]}$ for accessing the least significant bit. Notice that, based on Figure 3.1, logical right shift by 1 can then be reduced to left shift by 1.

Of course, barrel shifting only needs to be applied if the value of $t_2$ is less than $n$. Otherwise the result must be a constant 0 bit-vector. In the above reduction, this is checked by using the operation unsigned less than, which can then be reduced to left shift by 1 as shown by Figure 3.1.

Remark 3.2. Figure 3.2 shows that logical resp. arithmetic right shift (by any term) can be reduced to logical resp. arithmetic right shift by constant. In the paper we proved that all those operations can be reduced to the corresponding left shift operation and vice versa, therefore the above reduction already covers all those cases.
3.3 Complexity of Decision Problems in Bit-Vector Logic

All our previous results on the complexity of bit-vector logics were focusing on the decision problem of satisfiability and showed that complexity grows exponentially as a logarithmic, w.l.o.g. binary, encoding is used on all the scalars in bit-vector formulas. In [Kov+14], we propose much more general complexity results. Most importantly, this paper is not focusing on a single decision problem, but its theorems are applicable for any decision problem which is complete for some standard complexity class.

The main motivation for this paper was to investigate the computational complexity of word-level model checking, which is a very important tool in hardware and software verification. Given a transition system $(U, I, T, P)$ where $U$ is the set of states, $I$ the initial relation, $T$ the transition relation, and $P$ the condition to check, model checking is about to check if from an $I$-state a $P$-state can be reached according to $T$. This decision problem is called reachability in (explicit) model checking and is NL-complete. If the states are represented as bit-vector variables and the relations $I$, $T$, $P$ as bit-vector (QF_BV) formulas, then we are talking about word-level model checking. What is the complexity of that? Well, this depends on the scalar encoding we use in those bit-vector formulas. Table 3.4 shows that wold-level model checking is PSPACE-complete in the case of unary encoding and ExpSpace-complete in the case of binary encoding.

<table>
<thead>
<tr>
<th>Encoding $\rightarrow$</th>
<th>Problem</th>
<th>explicit</th>
<th>Boolean</th>
<th>unary</th>
<th>binary</th>
<th>$\nu$-logarithmic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>circ./formula,</td>
<td>QF_BV</td>
<td>QF_BV</td>
<td>QF_BV</td>
</tr>
<tr>
<td>Word-Level MC, Reachability</td>
<td>NL</td>
<td>PSPACE</td>
<td>PSPACE</td>
<td>ExpSpace</td>
<td>$(\nu-1)$-ExpSpace</td>
<td></td>
</tr>
<tr>
<td>Circuit Value, Alternating Reachability</td>
<td>P</td>
<td>ExpTime</td>
<td>ExpTime</td>
<td>2-ExpTime</td>
<td>$\nu$-ExpTime</td>
<td></td>
</tr>
<tr>
<td>Clique, 3-SAT, SAT, Knapsack</td>
<td>NP</td>
<td>NExpTime</td>
<td>NExpTime</td>
<td>2-NExpTime</td>
<td>$\nu$-NExpTime</td>
<td></td>
</tr>
<tr>
<td>$k$-QBF</td>
<td>$\Sigma_k^P$</td>
<td>NE$^{\Sigma_k^P}$</td>
<td>NE$^{\Sigma_k^P}$</td>
<td>2-NE$^{\Sigma_k^P}$</td>
<td>$\nu$-NE$^{\Sigma_k^P}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Examples of complexity increase by symbolic encoding. New results are indicated in boldface. All membership results hold for logics whose operators allow log-space computable bit-blasting. Hardness requires the operators $\land$, $\lor$, $\neg$, $=$, $+1$. The column with $\nu$ holds for all $\nu > 1$. [Kov+14]

Our theorems can deal with multi-logarithmic encoding as well, where the degree of scalar exponentiation is given by a parameter $\nu > 1$, as shown in the rightmost column of Table 3.4. Such a 3-logarithmic encoding is applied, for instance, in the SMT-LIB when declaring arrays, therefore world-level model checking with arrays is 2-NExpTime-complete.

As the caption of Table 3.4 shows, these results hold for any QF_BV fragment with operators that allow log-space computable bit-blasting. Note that all the operators in SMT-LIB are of this kind. Let us note that in this paper of ours we filled a gap by precisely defining what bit-blasting means and does. Last but not least, hardness holds for the minimal set of operators $\land$, $\lor$, $\neg$, $=$, and the increment operator $+1$.

The main contribution of our paper is to show how hardness for a standard complexity class can be automatically lifted due to the so-called Upgrading Theorem, for which the key was to prove the so-called Conversion Lemma. Although in our original paper [Kov+14] the proofs for the lemmas were only provided for the reviewers and were not included in the camera-ready paper, I am providing all the necessary additional material in my dissertation.
Our proofs employ the framework of descriptive complexity theory [Imm87]. The framework builds on the concepts of relational signatures, finite structures, quantifier-free and log-space reductions. The paper precisely defines what to mean by the bit-vector definition of relations and how to acquire a structure from it. Based on that, we can define a bit-vector representation $bv^Ω_ν(A)$ of any decision problem $A$ with respect to a scalar encoding $ν$ and a bit-vector operator set $Ω$. As a consequence of the theorems in the paper, the ultimate result of our paper is as follows:

**Corollary 3.3.** Given a standard complexity class $C$, a problem $A$, and a set $Ω$ of bit-vector operators that allow log-space computable bit-blasting, if $A$ is $C$-complete under quantifier-free reductions, then $bv^Ω_ν(A)$ is $Exp_ν(C)$-complete under log-space reductions.

To see how to apply this generic upgrading result, see again examples in Table 3.4.
SOLVING APPROACHES FOR BIT-VECTOR LOGICS
As we discussed in Section 3, knowing the computational complexity can help to find new, promising solving approaches for certain logics. The most straightforward way is to find a “target” logic in the same complexity class for which there exist efficient solving approaches, and then to invent a reduction from our logic to that “target” logic.

Bit-blasting is the most well-known such reduction, where Boolean logic is the “target”, for which the satisfiability checking problem is NP-complete. Unfortunately, our previous complexity results for bit-vector logics show that bit-blasting is an exponential reduction for even the most basic bit-vector logic QF$_{BV}$, which is NExpTime-complete. In general, bit-blasting is polynomial only for two classes of bit-vector problems:

- Bit-width bounded [KFB12] or scalar-bounded [KFB16] formula sets, which we proved to be in NP.
- The fragment QF$_{BV_{bw}}$ which is NP-complete [FKB13b]; [KFB16].

In order to come up with a polynomial reduction for the full logic of QF$_{BV}$, the “target” logic must be in NExpTime or, preferably, NExpTime-complete. In our paper [KFB13a], such a reduction from QF$_{BV}$ to the logic EPR is proposed. EPR, known as the Bernays-Schönfinkel class in first-order logic, is not only an NExpTime-complete logic [Lew80], but it also has efficient solving approaches such as the Inst-Gen approach [Kor09]; [Kor13] on which the solver iProver [Kor08] is based.

Another direction is to propose a polynomial reduction for a fragment of QF$_{BV}$. In our paper [FKB13a], we give such a reduction from (the satisfiability checking of) QF$_{BV_{\preceq 1}}$ to reachability in symbolic model checking. As we know, both decision problems are PSpace-complete. Needless to say that state-of-the-art model checkers are considered to be quite efficient, therefore one can hope to solve QF$_{BV_{\preceq 1}}$ formulas by using such a model checker.
4.1 REDUCTION OF QF$_{BV}$ INTO EPR

EPR is also known as the Bernays-Schönfinkel class, which is an NExpTime-complete fragment of first-order logic [Lew80]. EPR formulas, written in Skolemized form, contain only universal quantifiers and atoms in form $p(t_1, \ldots, t_n)$ where $t_i$ is either a (universal) variable or a constant.

In our paper [KFB13a], we choose EPR as a “target” logic for QF$_{BV}$. As it turns out in the paper, the “target” logic is actually not general EPR, but rather its fragment EPR2 which uses only two constants, 0 and 1. The paper, without striving for completeness, briefly shows how to translate any QF$_{BV}$ expression into EPR in a polynomial way. Note that a polynomial reduction in the formula size must be logarithmic in the bit-width of bit-vectors, since bit-widths are inherently logarithmically encoded in QF$_{BV}$.

There exist previous approaches to encode hardware verification problems into first-order logic [KKV09] or, in particular, into EPR [Emm+10]. The latter one is called the relational encoding [Emm+10], since bit-vectors are modeled as unary predicates. These predicates are over bit-indices, represented by dedicated constants. For instance, the $i$th bit of a bit-vector $x[n]$, $0 \leq i < n$, is represented by the atom $p_i(bitInd)$, where $bitInd$ is a constant. Note that for QF$_{BV}$, such a translation might introduce exponentially many constants, since bit-widths like $n$ are encoded logarithmically. Furthermore, in [Emm+10], not all the common bit-vector operators are addressed. All the arithmetic operators are assumed to be synthesized/bit-blasted in the verification front-end [Emm+10], potentially leading to an exponential blowup already before the actual encoding.

In contrast with the relational encoding, our translation [KFB13a] of QF$_{BV}$ into EPR is polynomial, meaning that all the common bit-vector operators can be translated to EPR formulas of polynomial size. To each bit-vector term of bit-width $n$, a dedicated $[\log_2 n]$-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 $x$.

Regarding the translation of bit-vector operators into EPR2, let us show the translation of addition as an example [KFB13a]. Given a term $x[2^n] + y[2^n]$, let us first rewrite it to the following bit-vector equations:

\begin{align*}
\text{add}[2^n] &= x[2^n] \oplus y[2^n] \oplus \text{cin}[2^n] & (4.1) \\
\text{cin}[2^n] &= \text{cout}[2^n] \ll 1 & (4.2) \\
\text{cout}[2^n] &= (x[2^n] \& y[2^n]) \mid (x[2^n] \& \text{cin}[2^n]) \mid (y[2^n] \& \text{cin}[2^n]) & (4.3)
\end{align*}

Note that Eqn. (4.1) and (4.3) only contain bitwise operators (and equality). Therefore, both can be translated into EPR2 in a direct way, by exploiting the succinctness of universal quantification, as follows:

\begin{align*}
p_{\text{add}}(i_{n-1}, \ldots, i_0) &\iff p_x(i_{n-1}, \ldots, i_0) \oplus p_y(i_{n-1}, \ldots, i_0) \oplus p_{\text{cin}}(i_{n-1}, \ldots, i_0) \\
p_{\text{cout}}(i_{n-1}, \ldots, i_0) &\iff (p_x(i_{n-1}, \ldots, i_0) \land p_y(i_{n-1}, \ldots, i_0)) \lor (p_x(i_{n-1}, \ldots, i_0) \land p_{\text{cin}}(i_{n-1}, \ldots, i_0)) \lor (p_y(i_{n-1}, \ldots, i_0) \land p_{\text{cin}}(i_{n-1}, \ldots, i_0))
\end{align*}

However, Eqn. (4.2), which contains shift by 1, has to be handled differently. We introduce a helper predicate $\text{succ}$ 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 \( \text{succ}(i_{n-1}, \ldots, i_0, j_{n-1}, \ldots, j_0) \) can be defined by \( n \) facts:

\[
\begin{align*}
\text{succ}(i_{n-1}, \ldots, i_3, i_2, 0, i_{n-1}, \ldots, i_3, i_2, 1) \\
\text{succ}(i_{n-1}, \ldots, i_3, i_2, 0, 1, i_{n-1}, \ldots, i_3, i_2, 1, 0) \\
\text{succ}(i_{n-1}, \ldots, i_3, 0, 1, i_{n-1}, \ldots, i_3, 1, 0, 0) \\
\vdots \\
\text{succ}(0, 1, \ldots, 1, 1, 0, \ldots, 0)
\end{align*}
\]

Using this helper predicate, Eqn. (4.2) can be translated into EPR2 as follows:

\[
\neg p_{cin}(0, \ldots, 0) \\
\text{succ}(i_{n-1}, \ldots, i_0, j_{n-1}, \ldots, j_0) \Rightarrow (p_{cin}(j_{n-1}, \ldots, j_0) \iff p_{cout}(i_{n-1}, \ldots, i_0))
\]

Our tool \texttt{bv2epr} builds a graph data structure, in which each bit-vector operation is represented by an EPR predicate, whose vertex stores its own functional definition as an EPR clause set. Figure 4.1 shows an example for the relational operator \( <_u \). Such graphs can be used as building blocks in the graphs of other operations, as the example in Figure 4.2 shows for \( <_s \).

![Graph representation of \( x^{[S]} <_{u} y^{[S]} \) by \texttt{bv2epr}](image_url)

In the paper, we report on experiments and provide new QF\_BV benchmarks. The results show that the overhead in formula size is rather small when translating QF\_BV into EPR, while all other formats often suffer from exponential blow-up. However, our results also show that the runtime of \texttt{iProver} is usually worse compared to the runtime of the bit-vector solver Boolector, which applies bit-blasting. Nevertheless, the evaluation also shows that there exist benchmarks where \texttt{iProver} is faster.

We made \texttt{bv2epr} and all the benchmarks publicly available at [BV2EPR].
Figure 4.2: Graph representation of $x^{[5]} <_s y^{[5]}$ by bv2epr
4.2 REDUCTION OF QF_BV_{≤1} INTO SYMBOLIC MODEL CHECKING

In our paper [FKB13b], and later also in [KFB16], we proved that the fragment QF_BV_{≤1} was \textit{FSPACE}-complete. The proof we presented there gave a (polynomial) reduction from QF_BV_{≤1} to sequential circuits.

As we show in [KFB16], QF_BV_{≤1} only allows \textit{linear arithmetic}. In [SK12b]; [SK12a], the authors give a translation from quantifier-free Presburger arithmetic with bitwise operations (QFPA_{bit}) to sequential circuits. We adopted their approach in order to construct a translation for QF_BV2_{≤1}. Given \( \Phi \in QF_BV_{≤1} \), we assume w.l.o.g that \( \Phi \) only consists of three types of expressions:

\[
\begin{align*}
  t_1[n] &= t_2[n] & (4.4) \\
  x[n] &= c[n] & (4.5) \\
  x[n] &= y[n] \ll 1[n] & (4.6)
\end{align*}
\]

where \( x[n] \), \( y[n] \) 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. We encode each equality in \( \Phi \) separately into an \textit{atomic sequential circuit}. Each circuit receives input bit streams for all variables start with the least significant bit. The circuits for Eqn. (4.4) and (4.5) apply equality checking on their input bits. The circuit for Eqn. (4.6) passes the input bit of \( y \) as the output bit of \( x \). Furthermore, we extend each atomic sequential circuit to include a \textit{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 the 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. Finally, after constructing atomic circuits, their outputs are combined by logical gates following the Boolean structure of \( \Phi \).

In [FKB13a], we present our tool \texttt{bv2smv} for translating QF_BV_{≤1} SMT-LIB instances into SMV instances, which then can be fed into symbolic model checkers. The translation is based on the previously described translation and is extended with the support for all bit-vector operators in QF_BV_{≤1}. In the paper, we provide new QF_BV_{≤1} benchmarks and report on experiments with state-of-the-art model checkers. We found that \textit{BDD-based model checkers} performed faster by several orders of magnitude on most of our benchmarks, compared to state-of-the-art SMT solvers.

Figure 4.3 shows a detailed overview of runtimes and memory consumption of various solvers on one of the benchmarks. Note that the plots use a logarithmic scale. The SMT solvers \texttt{STP} and \textbf{Boolector} have exponentially increasing runtime and memory consumption. The same holds...
for the BMC-based model checker Tip BMC, due to bit-blasting being an exponential reduction in general. The BDD-based model checkers NuSMV bw and IImc BDD bw perform best. Notice that the memory consumption of those model checkers is basically constant on our benchmarks. To summarize, our results showed that BDD-based model checking techniques performed much better than SAT-based approaches.

We made bv2smv and all the benchmarks publicly available at [BV2SMV].
SOLVING APPROACHES FOR DQBF
As we discussed in Section 4, bit-blasting is exponential for QF_BV, due to QF_BV being NExpTime-complete. The best scenario would be to choose an NExpTime-complete “target” logic to which we can come up with a polynomial reduction from QF_BV. This is exactly what we did in [KFB13a], where EPR served as a “target” logic. EPR seems a good choice since there exists efficient solvers for EPR. Nevertheless, our experiments in [KFB13a] provided moderate success in solving QF_BV formulas by the use of the EPR solver iProver.

Is there any other NExpTime-complete logic that we could pick as a “target” logic for QF_BV solving? A promising candidate is Dependency Quantified Boolean Formulas (DQBF) [PR79]; [PR79]; [PRA01]. DQBF can be considered to be in the middle between Quantified Boolean Formulas (QBF) and EPR. On the one hand, DQBF is an extension of QBF with Henkin quantifiers and, on the other hand, DQBF is a special case of EPR where the argument lists of a predicate are always the same. Interestingly, DQBF has the same computational complexity as EPR does, while QBF has lower complexity (PSPACE-complete).

The success of the DPLL-style algorithm QDPLL in solving QBF gives reason to investigate how QDPLL could be adapted to DQBF. In our paper [FKB12], this is exactly what we do and how we introduce DQDPLL, the very first solving algorithm dedicated to DQBF. In the paper, we show how to adapt certain components of the QDPLL approach, one by one, and how to extend the approach itself. Most importantly, a set of a new kind of clauses, the so-called Skolem clauses, must be maintained during the solving process. We show how to adapt important QDPLL components such as unit propagation, backtracking, clause learning, universal reduction, watched literal schemes, and selection heuristics such as VSIDS. Although our prototypical implementation of DQDPLL is not publicly available, the paper [FKB12] reports on preliminary experiments with mixed and not very convincing results. Either way, our algorithm DQDPLL can be considered to be the first dedicated DQBF solving approach.

On the other hand, the success of iProver [Kor08] in EPR solving motivated us to adapt its solving approach, the so-called Inst-Gen calculus [Kor09]; [Kor13] to DQBF. We propose such an adaptation and report on experiments in our paper [Frö+14]. Since DQBF can be considered to be a fragment of EPR and, consequently, the Inst-Gen calculus is inherently able to solve DQBF, we investigate special techniques to speed up the solving process by exploiting the Boolean domain of DQBF. For each input clause, our new solver iDQ keeps track of a list of clause instances, which are represented as bit-vectors. We show how to simulate certain Inst-Gen operations on those bit-vectors, such as unification, instantiation and redundancy check. iDQ can be considered the first publicly available, complete DQBF solver, and our experiments show that iDQ is quite efficient in solving DQBF.
5.1 A DPLL Algorithm for Solving DQBF

Our approach DQDPLL [FKB12] is a DQBF-adaptation of QDPLL, which is inherently for solving QBF. The fundamental difference between DQDPLL and QDPLL is how decisions are saved in the decision stack. While QDPLL only has to save the decision \( l_e \) for an existential variable \( e \), DQDPLL additionally has to save a so-called Skolem clause linked with the current branch of the search tree. The Skolem clause \( C_{sk} \) represents this decision according to the assignment \( \beta \) to the universal variables on which \( e \) depends. A Skolem clause is defined as

\[
C_{sk} := (l_{u_1}, \ldots, l_{u_m}, l_e), \quad \text{where} \quad l_{u_i} = \begin{cases} u_{i}, & \text{if } \beta(u_i) = \text{false} \\ \neg u_{i}, & \text{if } \beta(u_i) = \text{true} \\ \text{false}, & \text{if } \beta(u_i) = \text{undef}, \end{cases}
\]

assuming \( \text{var}(l_e) = e, \text{dep}(e) = \{u_1, \ldots, u_m\} \).

Skolem clauses are to be stored in the decision stack and are used for restoring an assignment to a previous level or for backtracking to a previous branch. The pseudo-code of DQDPLL can be seen in Figure 5.1.

DQDPLL(F) {
    while (true) {
        state = checkState(beta);
        if (state == STATE_UNSAT) {
            level = analyseUNSAT();
            if (level == 0) return UNSAT;
            backtrack(level);
        } else if (state == STATE_SAT) {
            level = analyseSAT();
            if (level == 0) return SAT;
            restoreAssignment(level);
        } else {
            literal = selectLiteral();
            skolemClause = generateSkolemClause(beta, literal);
            beta = updateAssignment(literal);
            addDecision(beta, skolemClause);
        }
    }
}

backtrack(level) {
    while (stack.Size > level) popStack();
    (beta, _) = stack.Element(level);
}

restoreAssignment(level) { (beta, _) = stack.Element(level); }

addDecision(beta, skolemClause) { pushStack(beta, skolemClause); }

Figure 5.1: Main methods of DQDPLL as pseudo-code [FKB12]

Consequently, whenever the current branch turns out to be UNSAT and backtracking takes places, it might happen that the algorithm jumps back to a previous branch far away in the search tree. Figure 5.2 illustrates this particular phenomenon, where the original DQBF has an existential variable \( e \) with \( \text{dep}_e = \{u_2, u_3\} \) and DQDPLL has constructed the search tree shown
in the figure so far. Let us suppose that the rightmost branch is UNSAT and, therefore, the algorithm analyses the conflict and discovers that the current assignment to $e(1,0)$ is one of the causes of the conflict and thus it is needed to be changed, e.g., from 0 to 1. Since $e$ does not depend on $u_1$, it might be the case that $e(1,0)$ was assigned not on the rightmost branch, but on a previous branch which assigned the same truth values to $u_2$ and $u_3$ as the rightmost branch does. If this is the case, we need to backtrack to that particular branch.

![Diagram](image-url)

**Figure 5.2:** Example of DQDPLL backtracking to a previous branch of the search tree

Consequently, DQDPLL sometimes needs to jump several branches back and to start to traverse those branches again. Note that for QBF this cannot happen since $e$ would depend on all of $u_1$, $u_2$, and $u_3$. We think that this phenomenon is one of the most fundamental reasons why DQDPLL “does not perform very well” [FKB12] in practice, while QDPLL is considered to be quite efficient.

Nevertheless, DQDPLL was a pioneer approach in DQBF solving. It addressed a lot of topics which later were taken up by others. For example, pure literal reduction for DQBF was later extended and generalized by [Git+15]; [Wim+15]. Universal reduction was also used by [Wim+15]. VSIDS-based selection heuristics were applied by [Frö+14] as well. Furthermore, DQDPLL investigated the topic of clause and cube learning which has not been addressed by any other DQBF solving approaches yet.
Our paper [Frö+14] proposes an instantiation-based DQBF solving approach and the first publicly available complete DQBF solver \( iDQ \).

The Inst-Gen calculus [Kor08]; [Kor13] is a promising solving approach for being adapted to DQBF. The Inst-Gen architecture is based on the counterexample-guided abstraction refinement (CEGAR) paradigm as the pseudo-code shows in Figure 5.3. For each clause in the input formula \( F \), one or more instances are generated and stored. In each iteration, a propositional abstraction is created by grounding all the clause instances and can be solved by any off-the-shelf SAT solver. If the SAT solver returns \( UNSAT \), the original formula is \( UNSAT \), too. On the other hand, if the SAT solver returns \( SAT \), the resulting assignment has to be checked for consistency with the original formula. If the assignment is not valid in this sense, we generate a few new clause instances and execute the loop again.

\[
\begin{align*}
F' &:= \text{initInstantiation}(F) \\
\text{while } &true \text{ do} \\
\text{\quad } F'' &:= \text{propositionalAbstraction}(F') \\
\text{\quad } (\text{state, assignment}) &= \text{checkSat}(F'') \\
\text{\quad if } &\text{(state == unsat) then return unsat} \\
\text{\quad if } &\text{isValid(assignment, } F, F') \text{ then return sat} \\
\text{\quad } F' &= \text{refineInstantiation}(assignment, F, F')
\end{align*}
\]

Figure 5.3: Pseudo-code of a CEGAR loop as used in the Inst-Gen calculus [Kor08]; [Kor09]; [Kor13]

The validity check is based on the unification of clauses in the original formula. If the current assignment passes the validity check, then new clause instances are generated by applying resolution steps. Before adding any clause instance to the knowledge base, the instance is checked for redundancy with respect to the already existing clauses instances.

Since each of the operations unification, resolution and redundancy check are needed to be applied lots of times during the solving process, it is necessary to optimize them as much as possible. In \( iDQ \), all of them are implemented by only using bit-vectors and applying bitwise operations, for the sake of taking advantage of the Boolean domain.

Beside the use of bit-vectors, other implementation issues are addressed as well, such as the maintenance of clauses instances in 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.

We have been experimenting with two types of heuristics. One of the heuristics is inspired by \( iProver \)'s default heuristic, based on the lexicographical combination of orders defined on given numerical/Boolean parameters. For instance, similar to \( iProver \)'s notation [Kor13], we use \([-\text{num\_dep};+\text{age};-\text{num\_symb}]\) for the priority queue of instances, i.e., 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 other heuristic is inspired by SAT solving and is based on the VSIDS scores [Mos+01] 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, priority is given to the passive instance with the highest average score of its literals.
Our experiments show that tDQ is quite efficient in solving DQBF, although, lacking in general-purpose DQBF solvers, we could only compare tDQ against the EPR solver tProver and the DQBF “refuter” DQBF2QBF [FT14], which can solve only UNSAT instances. We used the only publicly available DQBF benchmarks by Finkbeiner and Tentrup [FT14]. All of them encode partial equivalence checking (PEC) problems, i.e., circuits containing some “black boxes” compared against full circuits. For encoding benchmark instances, we proposed and used the file format DQDIMACS, which is an extension of QDIMACS, the standard format for QBF. Table 5.1 shows the results: the number of solved instances (#i), 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.

<table>
<thead>
<tr>
<th></th>
<th>#sat/uns</th>
<th>TO</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitcell_16_2</td>
<td>DQBF2QBF</td>
<td>98</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>88</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>97</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>82</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lookahead_16_2</td>
<td>DQBF2QBF</td>
<td>97</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>98</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>93</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adder_3_2</td>
<td>DQBF2QBF</td>
<td>94</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>82</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>43</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>86</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pec_xor2</td>
<td>DQBF2QBF</td>
<td>49</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitcell_16_4</td>
<td>DQBF2QBF</td>
<td>98</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>52</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lookahead_16_4</td>
<td>DQBF2QBF</td>
<td>97</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>88</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>62</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>32</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adder_3_4</td>
<td>DQBF2QBF</td>
<td>89</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>tDQ</td>
<td>58</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>tDQ_vids</td>
<td>21</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>tProver</td>
<td>54</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adder_3_6</td>
<td></td>
<td>74</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pec_xor3</td>
<td></td>
<td>77</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pec_xor4</td>
<td></td>
<td>99</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Results for DQBF PEC benchmarks [Frö+14]

In most of the cases, tDQ outperforms tProver. tDQ_vids performs even better than tDQ on the bitcell benchmarks but worse on the lookahead and adder benchmarks. The gap between the performance of tDQ and tProver is significant. On UNSAT instances, DQBF2QBF generally is the fastest solver. However, the performance of tDQ 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.

We made tDQ and all the benchmarks publicly available at [tDQ].
In my dissertation, I select and present 9 papers of mine, all of them being published between 2012 and 2016. Those papers received several citations from literature since then. In this section, I would like to give a summary on those citations.

6.1 COMPLEXITY OF BIT-VECTOR LOGICS

Bit-vector logics can be considered to be the cornerstones of SMT solving and, in particular, bit-precise reasoning. Although those logics are being used in verification tasks in industry on a daily basis, their precise computational complexity had never been identified by anyone. In our paper [KFB12] in 2012, we showed that several such logics were complete for certain classical complexity classes. For instance, we proved QF_BV, that is massively used in hardware verification, to be NExpTime-complete, and UFBV, that is applied in software verification, to be 2-NExpTime-complete. In our subsequent papers [FKB13b]; [KFB16]; [Kov+14], we extended and generalized the aforementioned results of ours.

I am giving the following summary on the citations we received to those papers.

- In 2016, an important book entitled the Handbook of Model Checking [CHV16] was published by Edmund Clarke, the father of model checking, and their co-authors. In the book, Clark Barrett (New York University) and Cesare Tinelli (University of Iowa) cite our paper [FKB13b] in their book chapter on SMT.
- Roberto Sebastiani (University of Trento), who is one of the most prominent persons in SMT, proposes a general criterion for T-solving to be NP-hard in [Seb16].
- In our paper [KFB12], we drew the attention to an inaccurate result published by researchers at Microsoft Research, Oxford University and ETH Zurich. In a paper of theirs [WHM10], UFBV is claimed to be NExpTime-complete. Those authors cite our paper in their follow-up paper [Coo+13] in the journal of Formal Methods in System Design in 2013.
- Researchers at the Uppsala University and Microsoft Research cite our papers [KFB12]; [Kov+14] in their paper [ZWR16]; [Zel16] in which they propose a novel DPLL-style bit-vector solver.
- In the same paper, and also in [Kaf+17] from the University of Melbourne some of our bit-vector benchmarks are used for experiments.
- We could never prove if BV is complete for a complexity class. [JS16a] fills this gap by proving BV to be AEXP(poly)-complete and cites our papers [FKB13b]; [KFB16]. The same authors propose a new approach for solving BV formulas in [JS16b].
- Researchers at the University of Melbourne describe a bit-vector solving approach with word-level propagation and learning [WSS16], and cite our paper [KFB12].
- Our paper [KFB16] is cited by [MJB17], which proposes an approach for boosting the performance of semi-symbolic model checkers. [BL17] cites the same paper of ours in constructing resolution proofs for verification problems that use nonlinear integer arithmetic.
Among the SMT solvers available on the market, the developers of CVC$_4$ and Boolector cite our results. A module of CVC$_4$ is responsible for generating and verifying bit-vector proof certificates [Had15]. PBoolector is a parallel version of Boolector [Rei14].

Alexander Nadel at Intel cites our papers several times. [Nad16] introduces a solver for routing problems. In [VNM17a] and [VNM17b], the authors apply model checking in solving problems that use bit-vectors and linear integer arithmetic, respectively, as being inspired by our work on applying model checkers for solving bit-vector problems [FKB13a]. In [DN15], the authors formulate criticism of some of our results by introducing problem classes for which QF$_{BV}$ is only NP-complete. However, this result is not really a novelty, since in 2012 we have already defined the concept of so-called bit-width bounded formula sets in [KFB12], which fall into NP and into which the problems proposed by the above-mentioned researchers fall.

### 6.2 Solving Approaches for DQBF

In our paper [FKB12] in 2012, we have started something new. Before this paper, there existed no published algorithm specifically for DQBF. Although this algorithm of ours had no publicly available implementation that could be used in practice, our paper acted as a catalyst and more people started to work on the possibility of applying DQBF solving in practice. In 2014, we published another DQBF solving approach [Frö+14] and made our solver $iDQ$ publicly available [iDQ], which is cited and used by several papers for comparative experiments.

I am giving the following summary on the citations we received to those results.

- A research group at the university of Freiburg cites our papers [FKB12]; [Frö+14] regularly and uses $iDQ$ in experiments. Their work is a big part of that DQBF solving is booming and finds practical applications. The group publishes new approaches for DQBF solving (quantifier elimination [Git+15], QBF approximation [Wim+17a]), preprocessing [Wim+15]; [Wim+17b], Skolem functions [Wim+16a], dependency schemes [Wim+16b], and defines new problem classes such as partial equivalence checking [Git+13b]; [Git+13a]; [Wim+18], testing and verification with unknown values [Bec+15].

- A research group at Saarland University proposes a fast approach for solving unsatisfiable DQBF formulas in [FT14]. They build a framework for bounded synthesis [FFT17]; [Fay+17] that translates problems into several logics such as SAT, QBF, DQBF, EPR, etc., and uses our solver $iDQ$ as the underlying DQBF solver.

- Researchers at the University of Leeds and the University of Manchester examine existing resolution systems for QBF and investigate the possibility of lifting them to DQBF [Bey+16]. Markus N. Rabe at UC Berkeley proposes a sound and complete resolution-style proof system for DQBF in [Rab17].

- Others also consider the possibility of applying DQBF solvers in synthesis problems [BKS14]; [Kön15]; [Blo+16].

- The synthesis problem for distributed systems that fulfill a given specification is also an interesting topic. By a reduction to DQBF, the researchers of IST Austria proved the synthesis problem to be $\text{NEXP}^\text{TIME}$-complete, in case of LTL specification not containing the “next” operator [Cha+13].
6.3 Reducing Bit-Vector Problems into Other Logics

Although, due to the its computational complexity, the reduction of \( \text{QF}_\text{BV} \) to \( \text{DQBF} \) seems an obvious choice, we also propose reductions to other logics which are rather widely used. Our paper \([\text{KFB}13a]\) proposes a reduction to \( \text{EPR} \) and \([\text{FKB}13a]\) to \( \text{CTL} \) model checking. We implemented those reductions and made those tools publicly available \([\text{BV}_{2\text{EPR}}]\); \([\text{BV}_{2\text{SMV}}]\). Although our experiments showed that solving bit-vector problems by applying those reductions was not effective enough in general, on certain problem classes (e.g., some provided by Intel) they boosted the solving process.

I am giving the following summary on the citations we received to those papers.

- Nikolaj Bjørner is a principal researcher at Microsoft and a leading developer of Microsoft’s SMT solver \( \text{Z}3 \). Konstantin Korovin, at the University of Manchester, is the developer of the EPR solver \( t\text{Prover} \). Both cite our EPR reduction in \([\text{Bjø+}13]\).

- Christoph M. Wintersteiger at Microsoft, together with co-authors from the Uppsala University, proposes a novel bit-vector solving approach \([\text{ZWR}16]; \text{[Zel}16\text{]}\) that can deal with benchmarks introduced in our paper \([\text{FKB}13a]\). The same sets of benchmarks are used in \([\text{Kaf+}17]\) for experiments.

- Alexander Nadel at Intel, as already mentioned, cites our approach of reducing \( \text{QF}_\text{BV} \) to model checking in a couple of his papers. \([\text{VNM}17a]\) upgrades our approach and gives support for quantifier-free linear arithmetic over integers modulo \( 2^n \). The follow-up paper \([\text{VNM}17b]\) extends the approach to the same arithmetic for integers in general.

- \([\text{Kri}17]\), which cites both of our afore-mentioned papers, translates bit-vector formulas into explicit state model checking instead of symbolic model checking.
SUMMARY OF NEW SCIENTIFIC RESULTS (THESES)
The following items summarize my new scientific results:

1. The encoding of scalars in bit-vector formulas dominates the complexity of bit-vector logics. If unary encoding is used, QF_BV and QF_UFBV are NP-complete, BV is PSPACE-complete, and UFBV is NEXPTime-complete.

   If a logarithmic w.l.o.g. binary encoding is used, QF_BV and QF_UFBV are NEXPTime-complete, BV is NEXPTime-hard and in ExpSpace, and UFBV is 2-NEXPTime-complete.

   In a scalar-bounded formula set, the value of the maximal scalar is polynomial in the number of scalars for each formula in the set. Given a scalar-bounded formula set $S$, $S \subseteq QF_BV$ or $S \subseteq QF_UFBV$ implies $S \in NP$, $S \subseteq BV$ implies $S \in PSPACE$, and $S \subseteq UFBV$ implies $S \notin NEXPTime$, even if scalars are encoded in binary format.

   This thesis is based on my papers [KFB12]; [KFB16].

2. It is worth to investigate the computational complexity of the following fragments of QF_BV, which can be defined by restricting the set of bit-vector operations in formulas. QF_BV_{bw} allows only bitwise operations and equality, and is NP-complete. QF_BV_{\leq 1} additionally allows left shift by 1 and is PSpace-complete. QF_BV_{\ll} additionally allows left shift by any constant and is NEXPTime-complete.

   Those fragments can be extended with certain bit-vector operations without increasing complexity. QF_BV_{bw} can be extended with indexing and relational operators. QF_BV_{\leq 1} can additionally be extended with additive operators and multiplication by constant. QF_BV_{\ll} can additionally be extended with general multiplication, concatenation and extraction.

   Addition, subtraction, multiplication by constant and logical right shift by 1 can be considered to be alternative base operations for QF_BV_{\leq 1}. Multiplication, concatenation, extraction, logical right shift by constant, general left shift and logical right shift can be considered to be alternative base operations for QF_BV_{\ll}.

   This thesis is based on my papers [FKB13b]; [KFB16].

3. It is worth to investigate the computational complexity of the following fragments of quantified bit-vector logics. Both UFBV_{\ll} and UFBV_{\leq 1} are 2-NEXPTime-complete.

   The fragment BV_{log} restricts the bit-width of universally quantified variables not to exceed the logarithm of the bit-width of existentially quantified variables. Both BV_{log} and UFBV_{log} are NEXPTime-complete.

   QF_{UFBV_M} extends QF_UFBV with non-recursive macros, and is basically a quantified bit-vector logic. QF_{UFBV_M} is NEXPTime-complete.

   This thesis is based on my paper [KFB16].

4. Given a decision problem $A$, let $bv^\Omega_\nu(A)$ denote the bit-vector representation of $A$, using a $\nu$-logarithmic scalar encoding and a set $\Omega$ of log-space bit-blastable bit-vector operators.

   The membership of $A$ for a standard complexity class $C$ can be automatically lifted as follows: if $A \in C$, then $bv^\Omega_\nu(A) \in \text{Exp_\nu}(C)$.

   Similarly, the hardness of $A$ for a standard complexity class $C$ can be automatically lifted as follows: if $A$ is $C$-hard under quantifier-free reductions, then $bv^\Omega_\nu(A)$ is $\text{Exp_\nu}(C)$-hard under log-space reductions, if $\nu > 1$ and $\Omega \supseteq \{\land, \lor, \sim, =, +1\}$.

   As a direct consequence, the computation complexity of reachability in word-level model checking can be proved to be $(\nu-1)$-ExpSpace-complete, if $\nu > 1$ and $\Omega \supseteq \{\land, \lor, \sim, =, +1\}$, under log-space reductions. In particular, in the case of $\nu = 2$, i.e., when scalars are encoded as w.l.o.g. binary numbers, word-level model checking is ExpSpace-complete.

   This thesis is based on my paper [Kov+14].
5. QF\_BV formulas can be polynomially translated into EPR, which requires a polynomial reduction in the formula size to be logarithmic in bit-width.

Experimental results show that the overhead in formula size is rather small, while all other formats often suffer from exponential blow-up. The runtime of the EPR solver iP\(\text{rover}\) is usually worse compared to the runtime of bit-blasters. The evaluation also shows that there exist benchmarks where iP\(\text{rover}\) is faster.

This thesis is based on my paper [KFB\textsubscript{13}a].

6. QF\_BV\(_{\leq 1}\) formulas can be polynomially translated into sequential circuits and solved by symbolic model checkers.

Experimental results show that BDD-based model checkers perform faster by several orders of magnitude on most of our benchmarks, compared to state-of-the-art SMT solvers.

This thesis is based on my paper [FKB\textsubscript{13}a].

7. The algorithm of QDPLL, which is inherently for solving QBF, can be adapted to DQBF. In the decision stack, Skolem clauses are needed to be maintained.

Several components of state-of-the-art QDPLL solvers can be adapted to DQBF as well, such as unit propagation, pure literal reduction, clause learning, universal reduction, selection heuristics and watched literal schemes.

This thesis is based on my paper [FKB\textsubscript{12}].

8. The Inst-Gen calculus, which is inherently for solving EPR, can be adapted to DQBF. The operations unification, resolution and redundancy check can be implemented by only using bit-vectors and applying bitwise operations, for the sake of taking advantage of the Boolean domain.

Experimental results show that our solver iDQ outperforms iP\(\text{rover}\) on most of our benchmarks. VSIDS heuristics can boost the solving in several cases.

This thesis is based on my paper [Frö+14].
Part II

SELECTED PAPERS
ON THE COMPLEXITY OF FIXED-SIZE BIT-VECTOR LOGICS WITH BINARY ENCODED BIT-WIDTH
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 paper 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 \textsc{NExpTime}-complete. We also show that under certain restrictions the increase of complexity when using binary encoding can be avoided.

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. Syntax and semantics of fixed-size bit-vector logics do not differ much in the literature [CMR97]; [BDL98]; [BP98]; [Fra10]; [BS09]. Concrete formats for specifying bit-vector problems also exist, like the SMT-LIB format or the BTOR format [BBL08]. Working with non-fixed-size bit-vectors has been considered for instance in [BP98]; [ABK00] and more recently in spielman:2012 but will not be further discussed in this paper. 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 [BDL98] the authors consider quantifier-free bit-vector logic, and give an argument for \textsc{NP}-hardness of its satisfiability problem. In [BS09], a sublogic of the previous one is claimed to be \textsc{NP}-complete. In [WHM10]; [Win11], the quantified case is addressed, and the satisfiability of this logic with uninterpreted functions is proven to be \textsc{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 [Coo+10, 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 (actually without loss of generality) 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 resp. 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 in general bit-blasting is not polynomial. 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:

{\begin{verbatim}
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(assert (distinct (bvadd x y) (bvadd y x)))
\end{verbatim}
Using Boolector [BBL08] 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 bit-blasted anymore with our tool-flow (due to integer overflow). As this example shows, checking bit-vector logics through bit-blasting can not 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 NEExpTime-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.

8.2 preliminaries

We assume the common syntax for (fixed-size) bit-vector formulas, c.f. SMT-LIB and [CMR97; BDL98; BP98; Fra10; BS09; BBL08]. Every bit-vector possesses a bit-width \( n \), either explicit or implicit, where \( n \) is a natural number, \( n \geq 1 \). We denote a bit-vector constant with \( c^n \), where \( c \) is a natural number, \( 0 \leq c < 2^n \). 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 \( (t[i]^n) \), \( 0 \leq i < n \), bitwise negation \( (~t^n) \), bitwise and \( (t_1^n \land t_2^n) \), bitwise or \( (t_1^n \lor t_2^n) \), shift left \( (t_1^n \ll t_2^n) \), logical shift right \( (t_1^n \gg t_2^n) \), addition \( (t_1^n + t_2^n) \), multiplication \( (t_1^n \cdot t_2^n) \), unsigned division \( (t_1^n / u t_2^n) \), 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 \( t_1^n, \ldots, t_m^n \) are terms.

Let QF_BV1 resp. QF_BV2 denote the logics of quantifier-free bit-vectors with unary resp. binary encoded bit-width (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 resp. QF_UFBV2. Uninterpreted functions are powerful tools for abstraction, e.g., they can formalize reads on arrays. When quantification is introduced, we get the logics BV1 resp. BV2 when uninterpreted functions are prohibited. When they are allowed, we get UFBV1 resp. UFBV2. These latter logics are expressive enough, for instance, to formalize reads and writes on arrays with quantified indices.\(^1\)

8.3 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 8.6.1.

\(^1\)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.
First, consider unary encoding of bit-widths. Without uninterpreted functions nor quantification, i.e., for $\text{QF}_\text{BV1}$, the following complexity result can be proposed (for partial results and related work see also [BDL98] and [BS09]):

**Proposition 8.1.** $\text{QF}_\text{BV1}$ is NP-complete\(^2\)

*Proof.* By bit-blasting, $\text{QF}_\text{BV1}$ 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 $\text{QF}_\text{BV1}$ formulas whose all terms are of bit-width 1. \(\square\)

Adding uninterpreted functions to $\text{QF}_\text{BV1}$ does not increase complexity:

**Proposition 8.2.** $\text{QF}_\text{UFBV1}$ 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. [KS08, Chapter 3.3.1]. Therefore, $\text{QF}_\text{UFBV1}$ can be polynomially translated to $\text{QF}_\text{BV1}$. The other direction directly follows from the fact that $\text{QF}_\text{BV1} \subseteq \text{QF}_\text{UFBV1}$. \(\square\)

Adding quantifiers to $\text{QF}_\text{BV1}$ yields the following complexity (see also [Coo+10]):

**Proposition 8.3.** $\text{BV1}$ is PSpace-complete.

*Proof.* By bit-blasting, $\text{BV1}$ can be polynomially reduced to Quantified Boolean Formulas (QBF), which is PSpace-complete. The other direction directly follows from the fact that QBF $\subseteq \text{BV1}$ (following the same argument as in Prop. 11.11). \(\square\)

Our main contribution is to give complexity results for the more common logarithmic (actually without loss of generality) binary encoding. Even without uninterpreted functions nor quantification, i.e., for $\text{QF}_\text{BV2}$, we obtain the same complexity as for $\text{UFBV1}$.

**Proposition 8.4** (see [Win11]). $\text{UFBV1}$ is $\text{NExpTime}$-complete.

*Proof.* Effectively Propositional Logic (EPR), being $\text{NExpTime}$-complete, can be polynomially reduced to UFBV1 [Win11, Theorem 7]. For completing the other direction, apply the reduction in [Win11, Theorem 7] combined with the bit-blasting of the bit-vector operations. \(\square\)

Our main contribution is to give complexity results for the more common logarithmic (actually without loss of generality) binary encoding. Even without uninterpreted functions nor quantification, i.e., for $\text{QF}_\text{BV2}$, we obtain the same complexity as for $\text{UFBV1}$.

**Proposition 8.5.** $\text{QF}_\text{BV2}$ is $\text{NExpTime}$-complete.

*Proof.* It is obvious that $\text{QF}_\text{BV2} \in \text{NExpTime}$, since a $\text{QF}_\text{BV2}$ formula can be translated exponentially to $\text{QF}_\text{BV1} \in \text{NP}$ (Prop. 11.11), by a simple unary re-encoding of all bit-widths. The proof that $\text{QF}_\text{BV2}$ is $\text{NExpTime}$-hard is more complex and given in Sect. 8.3.1. \(\square\)

Adding uninterpreted functions to $\text{QF}_\text{BV2}$ does not increase complexity, again using Ackermann constraints, as in the proof for Prop. 11.12:

**Proposition 8.6.** $\text{QF}_\text{UFBV2}$ is $\text{NExpTime}$-complete.

However, adding quantifiers to $\text{QF}_\text{UFBV2}$ increases complexity exponentially:

**Proposition 8.7.** $\text{UFBV2}$ is $2\text{-NExpTime}$-complete.

*Proof.* Similarly to the proof of Prop. 8.5, a $\text{UFBV2}$ formula can be exponentially translated to $\text{UFBV1} \in \text{NExpTime}$ (Prop. 8.4), simply by re-encoding all the bit-widths to unary. It is more difficult to prove that $\text{UFBV2}$ is $2\text{-NExpTime}$-hard, which we show in Sect. 8.3.2. \(\square\)

Notice that deciding $\text{QF}_\text{BV2}$ has the same complexity as $\text{UFBV1}$. Thus, starting with $\text{QF}_\text{BV1}$, re-encoding bit-widths to binary gives the same expressive power, in a precise complexity theoretical sense, as introducing uninterpreted functions and quantification all together. 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 [WHM10]; [Win11].

---

\(^2\)This kind of result is often called unary NP-completeness [GJ78].
8.3 complexity

8.3.1 QF_BV2 is \text{NExpTime-hard}

In order to prove that QF_BV2 is \text{NExpTime-hard}, we pick a \text{NExpTime-hard} problem and, then, we reduce it to QF_BV2. Let us choose the satisfiability problem of \text{Dependency Quantified Boolean Formulas} (DQBF), which has been shown to be \text{NExpTime-complete} \cite{APR01}.

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 \( m \geq 0 \). Given an existential variable \( e \), we will use \( \text{Deps}(e) \) to denote the set of universal variables that \( e \) depends on. A more formal definition can be found in \cite{APR01}. Without loss of generality, we can assume that a DQBF formula is in clause normal form.

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

\[
\begin{array}{cccccccc}
0 & \ldots & 0 & 1 & \ldots & 1 & 0 & \ldots & 0 \\
2^i & 2^i & 2^i & 2^i & 2^i & 2^i & 2^i & 2^i & 2^i \\
\end{array}
\]

Given \( n \geq 1 \) and \( i \), with \( 0 \leq i < n \), we denote such a bitmask with \( M_n^i \). Notice that these bitmasks correspond to the \text{binary magic numbers} \cite{Fre83} (see also Chpt. 7 of \cite{War02}), and, can thus arithmetically be calculated in the following way (actually as sum of a geometric series):

\[
M_n^i := \frac{2^{(2^i)} - 1}{2^{(2^i)} + 1}
\]  

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

\[
M_n^i := \sim 0^{[2^i]} / u \left( (1 \ll (1 \ll i)) + 1 \right)
\]  \hspace{1cm} (8.1)

\textbf{Theorem 8.8.} DQBF can be (polynomially) reduced to QF_BV2.

\textit{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 Eqn. (8.1), 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 (c.f. an example in Appendix 9.7.2), then show that the reduction is polynomial, and, finally, that it is correct.

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

\textbf{Step 1.} \ Replace Boolean constants 0 and 1 with \( 0^{[2^i]} \) resp. \( \sim 0^{[2^i]} \) and logical connectives with corresponding bitwise bit-vector operators (\( \lor, \land, \neg \) with |, &, \( \sim \), resp.).

Let \( \Phi' \) denote the formula generated so far. Extend it to the formula \( \left( \Phi' = \sim 0^{[2^i]} \right) \).

\textbf{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 Eqn. (8.1) to $U_i$, add the following equation (i.e., conjunct it with the current formula):

$$U_i = M_i^k$$

(8.2)

For an optimization see Remark 8.9 further down.

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

1. translate (all the occurrences of) $e$ to a new bit-vector variable $E^{[2^k]}$;

2. for each $u_i \notin \text{Deps}(e)$, add the following equation:

$$(E \& U_i) = \left( (E \gg_u (1 \ll i)) \& U_i \right)$$

(8.3)

As it is going to be detailed in the rest of the proof, the above equations enforce the corresponding bits of $E^{[2^k]}$ to satisfy the dependency scheme of $\phi$. More precisely, Eqn. (11.1) 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 $\Phi$, and thus exponential bit-widths ($2^k$) are encoded into linear many ($k$) bits. We show now 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 the original $\phi$ and the result $\Phi$ of the translation to be equisatisfiable. Consider one bit-vector variable $U_i$ introduced in Step 2. In the following, we formalize the well-known fact that all the $U$s 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

$$U_i[b_m] = U_j[b_n] \quad \text{and} \quad U_i[b_n] = U_j[b_m], \quad \forall j \neq i.$$ 

(8.4a)

(8.4b)

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

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

By defining $b_n$ this way, Eqns. (11.2a) and (11.2b) 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 $c$. Eqn. (11.2a) holds, since, due to construction, $U_i$ consists of several $(2^{k-1-i})$ concatenated bit-vector fragments $0 \ldots 01 \ldots 1 = R(0, 2^l)R(1, 2^l)$ (with both $2^l$ zeros and $2^l$ ones). Therefore it is easy to see that $U_i[b_m] \neq U_i[b_m - 2^i] \ (\text{resp. } U_i[b_m] \neq U_i[b_m + 2^i])$ holds if $U_i[b_m] = 0 \ (\text{resp. } U_i[b_m] = 1)$. With a similar argument, we can show that Eqn. (11.2b) holds: $U_i[b_m] = U_j[b_m - 2^i]$ (resp. $U_i[b_m] = U_j[b_m + 2^i]$) if $U_i[b_m] = 0 \ (\text{resp. } U_i[b_m] = 1)$, since $b_m - 2^i \ (\text{resp. } b_m + 2^i)$ is
located either still in the same half or already in a concatenated copy of a \( R(0,2^i)R(1,2^i) \) fragment, if \( j \neq i \).

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

In Step 3 we first replace each existential variable \( e \) with a new bit-vector variable \( E \), which can take \( 2^{(2^i)} \) different values. The value of each individual bit \( E[b_\alpha] \) corresponds to the value \( e \) takes under a given assignment \( a \in \{0,1\}^k \) to the universal variables. Note that without any further restriction, 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 universals.

If, however, \( e \) does not depend on a universal variable \( u_i \), we add the constraint of Eqn. (11.1). 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 \( a, \beta \in \{0,1\}^k \) where \( a[j] = \beta[j] \) for all \( j \neq i \). Exactly this is enforced by our constraint. We have already shown that for \( a \) we have a corresponding bit-vector index \( b_\beta \), and we have defined how we can construct a bit-vector index \( b_\beta \) for \( \beta \). 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 \( \phi \) evaluates to 1 for each universal assignment.

\( \square \)

**Remark 8.9.** Using Eqn. (8.1) in Eqn. (8.2) seems to require the use of division, which, however, can easily be eliminated by rewriting Eqn. (8.2) to

\[
\left( U_i ( (1 \ll (1 \ll i)) + 1 ) \right) = 0^{[2^i]}
\]

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

\[
\left( (U_i \ll (1 \ll i)) + U_i \right) = 0^{[2^i]}
\]

### 8.3.2 UFBV2 is \( 2\text{-NExpTime-hard} \)

In order to prove that UFBV2 is \( 2\text{-NExpTime-hard} \), we pick a \( 2\text{-NExpTime-hard} \) problem and then, we reduce it to UFBV2. We can find such a problem among the so-called domino tiling problems [Chl84]. Let us first define what a domino system is, and then we specify a \( 2\text{-NExpTime-hard} \) problem on such systems.

**Definition 8.10** (Domino System). A domino system is a tuple \( (T,H,V,n) \), where

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

Let us note that the above definition differs (but not substantially) from the classical one in [Chl84], in the sense that we use sub-sequential natural numbers for identifying tiles, as it is common in recent papers. Similarly to [Mar07] and [NS11], the size factor \( n \), encoded *unary*, is part of the input. However while a start tile \( a \) and a terminal tile \( \omega \) is used usually, in our case the starting tile is denoted by 0 and the terminal tile by \( k-1 \), without loss of generality.

There are different domino tiling problems examined in the literature. In [Chl84] a classical tiling problems is introduced, namely the *square tiling problem*, which can be defined as follows.
Theorem 8.12 (from [Chl84]). The $f(n)$-square tiling problem is $\text{NTime}(f(n))$-complete.

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

Corollary 8.13. The $2^{(2^n)}$-square tiling problem is 2-$\text{NExpTime}$-complete.

Theorem 8.14. The $2^{(2^n)}$-square tiling problem can be (polynomially) reduced to UFBV2.

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 $l := \lceil \log k \rceil$.
- 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 $\lambda^{[1]}(i^{[2^n]}, j^{[2^n]})$. As it is obvious, $\lambda$ represents the type of the tile in the cell at the row index $i$ and column index $j$. Notice 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:

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

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, namely 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. 

8.4 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 formula \( \phi \), let us denote the maximal explicit bit-width in \( \phi \) with \( \max_{\text{bw}} (\phi) \). Furthermore, let \( \text{size}_{\text{bw}} (\phi) \) denote the number of terms with explicit bit-width in \( \phi \).

**Definition 8.15** (Bit-Width Bounded Formula Set). An infinite set \( S \) of bit-vector formulas is (polynomially) bit-width bounded, if there exists a polynomial function \( p : \mathbb{N} \rightarrow \mathbb{N} \) such that \( \forall \phi \in S. \max_{\text{bw}} (\phi) \leq p(\text{size}_{\text{bw}} (\phi)) \).

**Proposition 8.16.** 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 \( \phi' \) denote the formula obtained through re-encoding bit-widths in \( \phi \) to unary. For the size of \( \phi' \) the following upper bound can be shown: \( |\phi'| \leq \text{size}_{\text{bw}} (\phi) \cdot \max_{\text{bw}} (\phi) + c \). Notice that \( \text{size}_{\text{bw}} (\phi) \cdot \max_{\text{bw}} (\phi) \) is an upper bound on the sum over the sizes of all the terms with explicit bit-width in \( \phi' \). The constant \( c \) represents the size of the rest of the formula. Since \( S \) is bit-width bounded, it holds that

\[
|\phi'| \leq \text{size}_{\text{bw}} (\phi) \cdot \max_{\text{bw}} (\phi) + c \leq \text{size}_{\text{bw}} (\phi) \cdot p(\text{size}_{\text{bw}} (\phi)) + c \leq |\phi| \cdot p(|\phi|) + c
\]

where \( p \) is a polynomial function. Therefore, the size of \( \phi' \) is polynomial in the size of \( \phi \). \( \Box \)

By applying this proposition to the logics of Sect. 16.2 we get:

**Corollary 8.17.** Let us assume a bit-width bounded set \( S \) of bit-vector formulas. If \( S \subseteq \text{QF}_{\text{UFBV2}} \) (and even if \( S \subseteq \text{QF}_{\text{BV2}} \)), then \( S \in \text{NP} \). If \( S \subseteq \text{BV2} \), then \( S \in \text{PSPACE} \). If \( S \subseteq \text{UFBV2} \), then \( S \in \text{NEXP\textsc{Time}} \).

### 8.4.1 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 \( \text{QF}_\text{BV} \), which can be considered as \( \text{QF}_\text{BV2} \) formulas.

First consider the benchmark family \( \text{QF}_\text{BV}/brummayerbiere2/umulov2bwb \), which represent instances of an unsigned multiplication overflow detection equivalence checking problem, and is 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 [Sch+oo], 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 [Sch+oo]. 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 family \( \text{QF}_\text{BV}/brummayerbiere3/mulhsb \) 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 [War02, 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 \( \text{QF}_\text{BV}/brutnomesso/lfsrc/lfsrc_b.n \) formalizes the behaviour of a linear feedback shift register [BS09]. 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.
8.5 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 actual 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 ⊂ BV2, it is not clear whether QF_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 [DF99].

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. [FKB12]; [Kor08]).

8.6 Appendix

8.6.1 Table: Completeness results for bit-vector logics

<table>
<thead>
<tr>
<th>encoding</th>
<th>quantifiers</th>
<th>no</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>uninterpreted functions</td>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td></td>
<td>uninterpreted functions</td>
<td>PSpace</td>
<td>NExpTime</td>
</tr>
<tr>
<td>unary</td>
<td>NExpTime</td>
<td>NExpTime</td>
<td>?</td>
</tr>
<tr>
<td>binary</td>
<td>NExpTime</td>
<td>NExpTime</td>
<td>2-NExpTime</td>
</tr>
</tbody>
</table>

Table 8.1: Completeness results for various bit-vector logics considering different encodings

8.6.2 Example: A reduction of DQBF to QF_BV2

Consider the following DQBF formula:

$$\forall u_0, u_1, u_2 \exists x(u_0), y(u_1, u_2) : (x \lor y \lor \neg u_0 \lor \neg u_1) \land$$
$$x \lor y \lor u_0 \lor \neg u_1 \lor \neg u_2) \land$$
$$x \lor y \lor \neg u_0 \lor \neg u_1 \lor u_2) \land$$
$$\neg x \lor \neg y \lor \neg u_0 \lor \neg u_2) \land$$
$$\neg x \lor \neg y \lor u_0 \lor u_1 \lor \neg u_2)$$
This DQBF formula is unsatisfiable. Let us note that by adding one more dependency for \(y\), or even by making \(x\) and \(y\) dependent on all \(u\)'s, the resulting QBF formula becomes satisfiable.

Using the reduction in Sect. 8.3.1, this formula is translated to the following QF_BV2 formula:

\[
\bigl( (X \mid Y \sim U_0 \mid \sim U_1) \& (X \mid \sim Y \mid U_0 \mid \sim U_1 \mid \sim U_2) \& (X \mid \sim Y \mid \sim U_0 \mid \sim U_1 \mid \sim U_2) \& (X \mid \sim Y \mid U_0 \mid \sim U_1 \mid \sim U_2) \& (X \mid \sim Y \mid U_0 \mid \sim U_1 \mid \sim U_2) \bigr) = \sim 0^{[8]} \land \\
\wedge_{i \in \{0,1,2\}} \left( (U_i \ll (1 \ll i)) + U_i \right) = \sim 0^{[8]} \right) \land (8.5) \\
(X \& U_1) = ((X \gg_u (1 \ll 1)) \& U_1) \land \\
(X \& U_2) = ((X \gg_u (1 \ll 2)) \& U_2) \land \\
(Y \& U_0) = ((Y \gg_u (1 \ll 0)) \& U_0)
\]

In the following, let us show that this formula is also unsatisfiable. Note that \(M^1_3 = 55_{16}^{[8]} = 010101011_2^{[8]}, M^2_3 = 33_{16}^{[8]} = 001110011_2^{[8]}\), and \(M^3_3 = 0F_{16}^{[8]} = 000011111_2^{[8]}\), where “\( \sim 16 \)” resp. “\( \sim 2 \)” denotes hexadecimal resp. binary encoding of the binary magic numbers.

In the following, let us show that the formula (11.6) 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

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

and

\[
((X \gg_u (1 \ll 1)) \& U_1) = (0, 0, X[7], X[6], 0, 0, X[3], X[2])
\]

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

\[
X := (x_3, x_4, x_5, x_4, x_1, x_0, x_1, x_0)
\]

Furthermore, considering also the equation of

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

and

\[
((X \gg_u (1 \ll 2)) \& U_2) = (0, 0, 0, 0, X[7], X[6], X[5], X[4])
\]

results in

\[
X := (x_1, x_0, x_0, x_1, x_1, x_0, x_1, x_0)
\]

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

\[
Y := (y_6, y_6, y_4, y_4, y_2, y_2, y_0, y_0)
\]

In order to show that the formula (11.6) is unsatisfiable, let us evaluate the “clauses” in the formula:

\[
(X \mid Y \sim U_0 \sim U_1) = (1, 1, 1, x_0 \lor y_4, 1, 1, 1, x_0 \lor y_0)
\]

\[
(X \mid \sim Y \mid U_0 \sim U_1 \sim U_2) = (1, 1, 1, 1, 1, 1, x_1 \lor \bar{y_0}, 1)
\]

\[
(X \mid \sim Y \sim U_0 \sim U_1 \sim U_2) = (1, 1, 1, x_0 \lor \bar{y_4}, 1, 1, 1, 1)
\]

\[
(\sim X \mid Y \sim U_0 \sim U_2) = (1, 1, 1, 1, \bar{x_0} \lor y_2, 1, \bar{x_0} \lor y_0)
\]

\[
(\sim X \mid \sim Y \mid U_0 \mid U_1 \sim U_2) = (1, 1, 1, 1, \bar{x_1} \lor \bar{y_2}, 1, 1, 1)
\]
By applying bitwise and to them, we get the bit-vector represented by the formula (11.6):

\[
\begin{pmatrix}
1 \\
1 \\
1 \\
(x_0 \lor \neg y_4) \land (x_0 \lor y_4) \\
\neg x_1 \lor \neg y_2 \\
\neg x_0 \lor y_2 \\
x_1 \lor \neg y_0 \\
(x_0 \lor y_0) \land (\neg x_0 \lor y_0)
\end{pmatrix}
= \begin{pmatrix}
1 \\
1 \\
1 \\
x_0 \\
\neg x_1 \lor \neg y_2 \\
\neg x_0 \lor y_2 \\
x_1 \lor \neg y_0 \\
y_0
\end{pmatrix}
\]

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 \(\neg x_1 \lor \neg y_2\).
MORE ON THE COMPLEXITY OF QUANTIFIER-FREE FIXED-SIZE BIT-VECTOR LOGICS WITH BINARY ENCODING
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 [KFB12], 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 NE\(\text{xp}\)\(T\)ime-complete.

In this paper, 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 NE\(\text{xp}\)\(T\)ime-completeness, we show that the logic becomes PS\(\text{pace}\)-complete if, instead of shift by constant, only shift by 1 is permitted, and even NP-complete if no shifts are allowed at all.

9.1 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 [CMR97]; [BDL98]; [BP98]; [BS09]; [Fra10]. Concrete formats for specifying bit-vector problems also exist, e.g. the SMT-LIB format [BST10] or the BTOR format [BBL08].

Working with non-fixed-size bit-vectors has been considered for instance in [BP98]; [ABK00], and more recently in [SK12b], but is not the focus of this paper. 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 [BDL98] the authors consider quantifier-free bit-vector logic and give an argument for the NP-hardness of its satisfiability problem. In [BS09], a sublogic of the previous one is claimed to be NP-complete. Interestingly, in [Bry+07] 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 [WHM10]; [Win11], the quantified case is addressed, and the satisfiability problem of this logic with uninterpreted functions (UFBV) is proved to be NE\(\text{xp}\)\(T\)ime-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 [KFB12], we already investigated how complexity varies if we consider either a unary or a logarithmic, actually without loss of generality, 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 [Coo+10, Page 239, Footnote 3]). Tab. 9.1 summarizes the completeness results we obtained in [KFB12].

In this paper, 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
Table 9.1: Completeness results of [KFB12] for various bit-vector logics and encodings.

<table>
<thead>
<tr>
<th>encoding</th>
<th>quantifiers</th>
<th>uninterpreted functions</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>unary</td>
<td>no</td>
<td>yes</td>
<td>NExpTime</td>
</tr>
<tr>
<td>binary</td>
<td>yes</td>
<td>no</td>
<td>PSpace</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>yes</td>
<td>NExpTime</td>
</tr>
</tbody>
</table>

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 resp. word-level description into a bit-level circuit, as in hardware synthesis. The result can then be checked by a (propositional) SAT solver. In [KFB12], 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:

```smt2
(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 [KFB12], that deciding bit-vector logics, even without quantifiers, is much harder. It turned out to be NExpTime-complete in the general case.

However, in [KFB12] 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 paper, 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 [KFB12] with work on linear arithmetic on non-fixed-size bit-vectors [SK12b]; [SK12a] and work on the reduction of bit-widths [Joh01]; [Joh02]. The formula in the given example only contains bitwise operations, equality, and addition. Solving this kind of formulas turns out to be PSpace-complete.

9.3 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 9.1** (Term). A bit-vector term $t$ of bit-width $n$ ($n \in \mathbb{N}$, $n \geq 1$) is denoted by $t[n]$. A term is defined inductively as follows:

<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We also define how to measure the size of bit-vector expressions:

**Definition 9.2 (Size).** The size of a bit-vector term \( t[n] \) is denoted by \( |t[n]| \) and is defined inductively as follows:

<table>
<thead>
<tr>
<th>term</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>natural number:</td>
<td>( \text{enc}(n) )</td>
</tr>
<tr>
<td>bit-vector constant:</td>
<td>(</td>
</tr>
<tr>
<td>bit-vector variable:</td>
<td>(</td>
</tr>
<tr>
<td>bitwise negation:</td>
<td>(</td>
</tr>
<tr>
<td>bitwise and/or/xor:</td>
<td>(</td>
</tr>
<tr>
<td>equality:</td>
<td>(</td>
</tr>
<tr>
<td>shift by constant:</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>( \log_2 (n + 1) + 1 )</td>
</tr>
<tr>
<td></td>
<td>( enc(c) + enc(n) )</td>
</tr>
<tr>
<td></td>
<td>( 1 + enc(n) )</td>
</tr>
<tr>
<td></td>
<td>( 1 +</td>
</tr>
<tr>
<td></td>
<td>( 1 +</td>
</tr>
</tbody>
</table>

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 paper, we may omit parentheses in a formula for the sake of readability.

Let \( \Phi \) be a bit-vector formula and \( \alpha \) an assignment to the variables in \( \Phi \). We use the notation \( \alpha(\Phi) \) to denote the evaluation of \( \Phi \) under \( \alpha \), with \( \alpha(\Phi) \in \{0, 1\} \). \( \alpha \) satisfies \( \Phi \) 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, \( \text{QF_BV2}_{bw} \subseteq \text{QF_BV2}_{\ll 1} \subseteq \text{QF_BV2}_{\ll c} \). In Sec. 9.4, we investigate the complexity of the satisfiability problem for these logics:

- **QF_BV2_{\ll c}** is \( \text{NExpTime}\)-complete.
- **QF_BV2_{\ll 1}** is \( \text{PSpace}\)-complete.
- **QF_BV2_{bw}** is \( \text{NP}\)-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. Without loss of generality, we therefore do not explicitly deal with uninterpreted functions.

9.4 Complexity Results

Theorem 9.3. $\text{QF}_\text{BV2} \subseteq c$ is NE$\text{xp}$-time-complete.

Proof. The claim directly follows from our previous work in [KFB12]. We informally defined $\text{QF}_\text{BV2}$ 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 $\text{QF}_\text{BV2}$ is NE$\text{xp}$-time-complete.

Obviously, $\text{QF}_\text{BV2} \subseteq c \subseteq \text{QF}_\text{BV2}$ and therefore, $\text{QF}_\text{BV2} \subseteq c \in \text{NE}\text{xp}$-time. To show the NE$\text{xp}$-time-hardness of $\text{QF}_\text{BV2}$, we gave a (polynomial) reduction from $\text{DQBF}$ (which is PS$\text{ime}$) to $\text{QF}_\text{BV2}$. Since we only used bitwise operations, equality, and shift by constant in our reduction, we also immediately get the NE$\text{xp}$-time-hardness of $\text{QF}_\text{BV2} \subseteq c$. □

Theorem 9.4. $\text{QF}_\text{BV2} \subseteq 1 \in \text{PSPACE}$-complete.

Proof. In Lemma 9.5, we give a (polynomial) reduction from $\text{QBF}$ (which is PSPACE-complete) to $\text{QF}_\text{BV2} \subseteq 1$. This shows the PSPACE-hardness of $\text{QF}_\text{BV2} \subseteq 1$. In Lemma 9.6, we then prove that $\text{QF}_\text{BV2} \subseteq 1 \in \text{PSPACE}$ by giving a translation from $\text{QF}_\text{BV2} \subseteq 1$ to (polynomial sized) Sequential Circuits. As pointed out for example in [PBG05], symbolic reachability problem is PSPACE-complete as well. □

Lemma 9.5. $\text{QBF}$ can be (polynomially) reduced to $\text{QF}_\text{BV2} \subseteq 1$.

Proof. To show the PSPACE-hardness of $\text{QF}_\text{BV2} \subseteq 1$, we give a polynomial reduction from $\text{QBF}$ similar to the one from DQBF to $\text{QF}_\text{BV2}$ that we proposed in [KFB12]. For our reduction, we again use the so-called binary magic numbers (or magic masks in [Knu11, p. 141]). Appendix 9.7.2 demonstrates how the reduction works.

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

$$\text{binnmagic} (2^m, 2^n) = 0 \ldots 0 1 \ldots 1 \ldots 0 \ldots 0 1 \ldots 1$$

Note that in [KFB12], we used shift by constant to construct the binary magic numbers, as done in the literature [Knu11]. This is not permitted in $\text{QF}_\text{BV2} \subseteq 1$. We therefore give an alternative construction using only bitwise operations, equality, and shift by 1:

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

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

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

\footnote{Note, logical right shifts were used in the proof in [KFB12]. However, by applying negated bit masks throughout the proof, all right shifts can be rewritten as left shifts.}
row of $B$ is interpreted as a number $0 \leq c < 2^n$ in binary representation, the corresponding row of $B'$ is equal to $c + 1$.

Now, again for all $m$, $0 \leq m < n$, 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 $0 \leq c < 2^n$. 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: $\text{binmagic}(2^m, 2^n) := b_m^{[2^n]}$.

Of course, all $b_m'$, for $0 \leq m < n$, can be eliminated and the two sets of constraints can be replaced by a single set of constraints:

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

Now let $\phi := Q.M$ denote a QBF formula with quantifier prefix $Q$ and matrix $M$. Since $\phi$ is a QBF formula (in contrast to DQBF in [KFB12]), 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 $\phi$ are ordered according to their appearance in $Q$, with $u_0$ (resp. $u_{n-1}$) being the innermost (resp. outermost) variable.

Translate $\phi$ to a QF_BV2 formula $\Phi$ by eliminating the quantifier prefix and translating the matrix as follows:

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

**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):
   $$U_m^{[2^n]} = \text{binmagic}(2^m, 2^n)$$

**STEP 3.** For an existential variable $e$ depending on $\text{Deps}(e) = \{u_m, \ldots, u_{n-1}\}$, 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 $\text{Deps}(e) = \emptyset$ add the following equation:
   $$(E \land \sim 1) = (E \ll 1) \quad (9.1)$$
   otherwise, if $m \neq 0$ add the two equations:
   $$U_m' = \sim ((U_m \ll 1) \oplus U_m) \quad (9.2)$$
   $$(E \land U_m') = ((E \ll 1) \land U_m') \quad (9.3)$$

Note that we omitted the bit-widths in the last equations to improve readability. Each bit position of $\Phi$ corresponds to the evaluation of $\phi$ 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. Eqn. (11.4) creates a bit-vector $U_{m}^{[2^e]}$ 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, Eqn. (11.4) does not have to be added multiple times, if several existential variables depend on the same universal variable. Eqn. (11.5) (resp. Eqn. (11.3)) ensures that the corresponding bits of $E^{[2^e]}$ satisfy the dependency scheme of $\phi$ 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, Eqn. (11.4) evaluates to $U_{0}^{[2^e]} = 0$, and as a consequence Eqn. (11.5) 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 [Don+02]. See also [PBG05] for a translation from QBF to Sequential Circuits.

\[\text{Lemma 9.6. QF}_\text{BV2}_1 \text{ can be (polynomially) reduced to Sequential Circuits.} \]

\[\text{Proof. In [SK12b]; [SK12a], the authors give a translation from quantifier-free Presburger arithmetic with bitwise operations (QFPA\text{bit}) to Sequential Circuits. We can adopt their approach in order to construct a translation for QF}_\text{BV2}_1 \text{. The main difference between QFPA\text{bit} and QF}_\text{BV2}_1 \text{ is the fact that bit-vectors of arbitrary, non-fixed, size are allowed in QFPA\text{bit} while all bit-vectors contained in QF}_\text{BV2}_1 \text{ have a fixed bit-width.} \]

\[\text{Given } \Phi \in \text{QF}_\text{BV2}_1 \text{ in flat form. Let } x^n, y^n \text{ denote bit-vector variables, } c^n \text{ a bit-vector constant, and } t^n_1, t^n_2 \text{ bit-vector terms only containing bit-vector variables and bitwise operations. Following [SK12b]; [SK12a] we further assume w.l.o.g that } \Phi \text{ only consists of three types of expressions: } t^n_1 = t^n_2, x^n = c^n, \text{ and } x^n = y^n \ll 1^n, \text{ since any QF}_\text{BV2}_1 \text{ formula can be written like this with only a linear growth in the number of original variables.} \]

\[\text{We encode each equality in } \Phi \text{ separately into an atomic Sequential Circuit. Compared to [SK12b]; [SK12a], two modifications are needed. First, we need to give a translation for } x = y \ll 1 \text{ to Sequential Circuits. This can be done for example by using the Sequential Circuit for } x = 2 \cdot y \text{ in QFPA\text{bit}. However, a direct translation can also easily be constructed.} \]

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

\[\text{When the counter reaches a value of } n, \text{ it does not change anymore and the output of the atomic Sequential Circuit is set to the same value as the output in the previous cycle. A counter like this can be realized with } \lceil \log_2(n) \rceil \text{ gates, i.e. polynomially in the size of } \Phi. \text{ In contrast to the implementation described in [SK12a], we assume that the input streams for all variables start with the least significant bit. However, as already pointed out by the authors in [SK12a], their choice was arbitrary and it is no more complicated to construct the circuits the other way round.} \]

\[\text{Finally, after constructing atomic circuits, their outputs are combined by logical gates following the Boolean structure of } \Phi, \text{ in the same way as for unbounded bit-width in [SK12b]; [SK12a]. Due to adding counters, we ensure that for every input stream } x_i, \text{ only the first } n_i \text{ bits of } x_i \text{ influence the result of the whole circuit.} \]

\[\text{For the proof of Thm. 9.9, we need the following definition and lemma from [KFB12]:} \]

\[\text{Definition 9.7 (Bit-Width Bounded Formula Set [KFB12]). Given a formula } \Phi, \text{ we denote the maximal bit-width in } \Phi \text{ with } \max_{\text{bw}}(\Phi). \text{ An infinite set } S \text{ of bit-vector formulas is (polynomially) bit-width bounded, if there exists a polynomial function } p : N \rightarrow N \text{ such that } \forall \Phi \in S. \max_{\text{bw}}(\Phi) \leq p(|\Phi|). \]

\[\text{Lemma 9.8 ([KFB12]). } S \in \text{NP for any bit-width bounded formula set } S \subseteq \text{QF}_\text{BV2}. \]

\[\text{Theorem 9.9. QF}_\text{BV2}_{\text{bw}} \text{ is NP-complete.} \]
Proof. Since Boolean Formulas are a subset of QF_BV_{bw}, NP-hardness follows directly. To show that QF_BV_{bw} ∈ NP, we give a reduction from QF_BV_{bw} to a bit-width bounded set of formulas. The claim then follows from Lemma 9.8.

Given a formula Φ ∈ QF_BV_{bw} in flat form. If Φ contains any constants c[i] ≠ 0[i], we remove those constants in a (polynomial) pre-processing step. Let c_{max}[i] = b_{k-1}...b_{i}b_0 be the largest constant in Φ denoted in binary representation by b_{k-1} = 1 and arbitrary bits b_{k-2},...,b_0. We now replace each equality t_1[i] = t_2[i] in Φ with

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

where k = min(m,k), and, if m > k, we additionally add

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

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

In a similar way, if m > k, (t_{1,h_i}[m-k] = t_{2,h_i}[m-k]) represents the remaining (m – k) rows of the original equality corresponding to the most significant bits. All occurrences of a variable x[i] are replaced with a new variable x_{h_i} and all occurrences of a constant c[i] are replaced with 0^[m-k]. Since this pre-processing step is logarithmic in the value of c_{max}, it is polynomial in |Φ|. Without loss of generality, we now assume that Φ does not contain any bit-vector constants different from 0[i].

We now construct a formula Φ’ by reducing the bit-widths of all bit-vector terms in Φ. Each term t[i] in Φ with bit-width n is replaced with a term t’[i], with n’ := min(n, |Φ|}. Apart from this, Φ’ is exactly the same as Φ. As a consequence, maxbw(Φ’) ≤ |Φ|. The set of formulas constructed in this way is bit-width bounded according to Def. 11.15.

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[i],...,x_k[i] for Φ we can also construct a satisfying assignment to x_1[i],...,x_k[i] 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, α(Φ) = α’(Φ’) for all equalities F (resp. F’) of Φ (resp. Φ’). As a direct consequence, α(Φ) = α’(Φ) = 1.

The other direction needs slightly more reasoning. Given α, with α(Φ) = 1, we need to construct α’, with α’(Φ’) = 1. Again, we want to ensure that α’(F’) = α(F) for all equalities F (resp. F’) in Φ (resp. Φ’).

In each variable x_i[i], i ∈ {1,...,k}, we are going to select some of the bits. For each equality F with α(F) = 0, we select a bit-index as a witness for its evaluation. If α(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 ∈ {1,...,k}, where

\[|M_i| ≤ min\{n_j,|Φ|\}\]

\[M_i = M_j \quad ∀ j \in \{1,...,k\} \text{ with } n_i = n_j\]

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_j,|Φ|\} ∀ i ∈ {1,...,k}.
Finally, we can directly construct $\alpha'$ using the selected indices and get $\alpha'(\Phi') = a(\Phi) = 1$ 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 $a(F) = 0$. For $a(F) = 1$, 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.

\begin{remark}
A similar proof can be found in [Joh01]; [Joh02]. While the focus of [Joh01]; [Joh02] 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 [KFB12], 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 [Joh01]; [Joh02] still hold for binary encoded bit-vector formulas when only bitwise operators are used, our proof considers the complexity of the problem class.
\end{remark}

9.5 Discussion

The complexity results given in Sec. 9.4 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-BV$_{2\leq 1}$. This can be seen by replacing expressions $x[n] = y[n] + z[n]$ by

\[
\left( x[n] = y[n] \oplus z[n] \oplus c_{in}[n] \right) \land \left( c_{in}[n] = c_{out}[n] \ll 1[n] \right) \land \\
\left( c_{out}[n] = \left( x[n] \land y[n] \right) \lor \left( c_{in}[n] \land y[n] \right) \lor \left( c_{in}[n] \land c_{in}[n] \right) \right)
\]

with new variables $c_{in}[n], c_{out}[n]$, and by splitting multiplication by constant into several multiplications by 2 (resp. shift by 1), similar to [SK12b]; [SK12a]. However, this is not surprising since QFPAnrr is already known to be PSPACE-complete [SK12a].

More interestingly, we can also extend QF-BV$_{2\leq 1}$ (resp. QFPAnrr) by indexing (denoted by $x[n][i]$) without growth in complexity. The counter we introduced in our translation from QF-BV$_{2\leq 1}$ to Sequential Circuits can be used to return the value at a specific bit-index of a bit-vector. Extending QF-BV$_{2\leq 1}$ with additional relational operators like e.g. unsigned less than (denoted by $x[n] <_{u} y[n]$) 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.,

\[
\left( y[n] \neq 0[n] \right) \land \left( c_{out}[n] \ll 1[n] = 0[1] \right).
\]

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

\[
\left( x[n][n - 1 : c] = y[n][n - c - 1 : 0] \right) \land \left( x[n][c - 1 : 0] = 0[c] \right)
\]
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 [DMR76]. We are not aware of any complexity results concerning non-fixed-size bit-vector logics with slicing or shift by constant.

9.6 Conclusion

In this paper, we discussed the complexity of fixed-size bit-vector logics with binary encoding on numbers. In contrast to existing literature, except for [KFB12], 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 [KFB12], 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_BV2bw). While deciding QF_BV2\ll c remains NExpTime-complete, we proved that QF_BV2\ll 1 is PSpace-complete, and QF_BV2bw 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 leads to 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 [KFB12], possible candidates for the general case are techniques used in EPR and/or DQBF solvers (see e.g. [FKB12]; [Kor08]).

On the other hand, we described various logics that remain in lower complexity classes. For QF_BV2bw this shows the importance of bit-width reduction as proposed in [Joh01]; [Joh02] 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 [LB10b] or symbolic model checking on Sequential Circuits [PBG05] might be of interest.
## 9.7 Appendix

### 9.7.1 Table: Comparison of Completeness Results for Fixed-Size and Non-Fixed-Size Bit-Vector Logics

<table>
<thead>
<tr>
<th>Logic</th>
<th>Fixed-Size</th>
<th>Non-Fixed-Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_BV2:</td>
<td>NExpTime [KFB12]</td>
<td>undecidable [DMR76]</td>
</tr>
<tr>
<td>QF_BV2_{&lt;c}:</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>QF_BV2_{&lt;1}:</td>
<td>PSpace [*]</td>
<td>PSpace [SK12b]; [SK12a]</td>
</tr>
<tr>
<td>QF_BV2_{bw}:</td>
<td>NP [*]</td>
<td>NP [*]</td>
</tr>
<tr>
<td>Presburger arithmetic:</td>
<td>?</td>
<td>NP</td>
</tr>
</tbody>
</table>

(*) cites our current paper

| Table 9.5: Completeness results for various fixed-size and non-fixed-size bit-vector logics. |

### 9.7.2 Example: A reduction of QBF to QF_BV2_{<1}

Consider the following QBF formula:

$$\exists x \forall u_2 \exists y \forall u_1 u_0 \exists z . (u_2 \lor u_1 \lor \neg z) \land (u_2 \lor \neg x \lor y) \land (u_0 \lor \neg x \lor \neg z) \land (u_1 \lor \neg y \lor z) \land (u_0 \lor \neg u_1 \lor z)$$

This QBF formula is satisfiable, and is translated to the following QF_BV2_{<1} formula:

$$(u_2 \mid u_1 \mid \neg Z) \land (u_2 \mid \neg X \mid Y) \land (u_0 \mid \neg X \mid \neg Z) \land$$

$$\bigwedge_{m \in \{0, 1, 2\}} \left( \bigwedge_{0 \leq i < m} U_i \oplus U_m = U_m \ll 1 \right) \land$$

$$(X \land \neg 1) = (X \ll 1) \land$$

$$(U_2' = \neg ((U_2 \ll 1) \oplus U_2)) \land ((Y \land U_2') = ((Y \ll 1) \land U_2'))$$

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.

---

*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 Thm. 9.9 still holds for the corresponding non-fixed-size one if we set all bit-widths in $\Phi'$ to $n' := |\Phi|$.**
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 $U_2 = 00011111_2[8]$. The following table gives some insight into the process of generating these binary magic numbers:

<table>
<thead>
<tr>
<th>$1 \odot U_1$</th>
<th>$U_0 \leq 1$</th>
<th>$U_1$</th>
<th>$U_1 \odot U_2$</th>
<th>$U_0 \leq 1$</th>
<th>$U_2$</th>
<th>$U_2 \odot U_1$</th>
<th>$U_0 \leq 1$</th>
<th>$U_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg 0,7$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg 1,7$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg 2,7$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
</tr>
<tr>
<td>$\neg 0,3$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,3}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,3}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\neg 0,1$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,1}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,1}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\neg 0,0$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,0}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,0}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\neg 0,7$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,7}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,7}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\neg 0,3$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,3}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,3}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\neg 0,1$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,1}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,1}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\neg 0,0$</td>
<td>$U_{0,4}$</td>
<td>$0$</td>
<td>$\neg u_{1,0}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
<td>$\neg u_{2,0}$</td>
<td>$U_{0,4}$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

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

$$
(X \& \neg 1) = (x_7, x_6, x_5, x_4, x_3, x_2, x_1, 0)
$$

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 $Y$:

$$
U'_2 := \neg ((U_2 \ll 1) \oplus U_2) = 11101110
$$

and therefore

$$
(Y \& U'_2) = (y_7, y_6, y_5, 0, y_3, y_2, y_1, 0)
$$

$$
(Y \ll 1) \& U'_2) = (y_6, y_5, y_4, 0, y_2, y_1, y_0, 0)
$$

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 (11.7) is satisfiable, let us evaluate the “clauses” in the formula:

$$
(U_2 \mid U_1 \mid \neg Z) = (\neg z_7, \neg z_6, 1, 1, 1, 1, 1)
$$

$$
(U_2 \mid \neg X \mid Y) = (\neg x_0 \lor y_4, \neg x_0 \lor y_4, \neg x_0 \lor y_4, \neg x_0 \lor y_4, 1, 1, 1)
$$

$$
(U_0 \mid \neg X \mid Z) = (\neg x_0 \lor \neg z_7, 1, \neg x_0 \lor \neg z_5, 1, \neg x_0 \lor \neg z_5, 1, \neg x_0 \lor \neg z_7, 1)
$$

$$
(U_1 \mid \neg Y \mid Z) = (\neg y_4 \lor z_7, \neg y_4 \lor z_4, 1, 1, \neg y_0 \lor z_4, \neg y_0 \lor z_3, 1, 1)
$$

$$
(U_0 \mid \neg U_1 \mid Z) = (1, 1, z_5, 1, 1, 1, z_1, 1)
$$

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

$$
\Phi' = \begin{pmatrix}
\neg z_7 \land (\neg x_0 \lor y_4) \land (\neg x_0 \lor \neg z_7) \land (\neg y_4 \lor z_7) \\
\neg z_6 \land (\neg x_0 \lor y_4) \land (\neg y_4 \lor z_6) \\
(\neg x_0 \lor y_4) \land (\neg x_0 \lor \neg z_5) \land z_5 \\
\neg x_0 \lor y_4 \\
(\neg x_0 \lor \neg z_3) \land (\neg y_0 \lor z_4) \\
\neg y_0 \lor z_3 \\
(\neg x_0 \lor \neg z_1) \land z_1 \\
1
\end{pmatrix}
$$
In order to check if $\Phi' = \sim 0^{[8]}$ is satisfiable, one can check the satisfiability of the following simplified clause set:

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

This can be satisfied e.g. by setting

$$z_7 = x_0 = y_4 = z_6 = y_0 = 0$$

$$z_5 = z_1 = 1$$

Therefore,

$$U_0 = 01010112^{[8]}, \quad U_1 = 001100112^{[8]}, \quad U_2 = 00011112^{[8]},$$

$$X = 000000002^{[8]}, \quad Y = 000000002^{[8]}, \quad Z = 001111112^{[8]}$$

is a possible solution of the bit-vector formula (11.7).
QUANTIFIER-FREE BIT-VECTOR FORMULAS WITH BINARY ENCODING: BENCHMARK DESCRIPTION
Quantifier-free Bit-Vector Formulas: Benchmark Description

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 [KFB12], we argued that a logarithmic (w.l.o.g. binary) encoding, as used e.g. in the SMT-LIB format [BST10], 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.

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.

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 (bvadiv), unsigned remainder (bvremp), signed remainder (bvsrem), signed modulo (bvsmod), logical shift right (bvlsr), arithmetic shift right (bvashr), shift left (bvshl), 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.

### 10.2.2 Bit-Vector Properties in PSpace

The second category consists of QF_BV benchmark sets with a reduced set of operations. In [FKB13b], 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 \((x[n] + 1[n]) \leq y[n]\). 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 \((x[n] + 1[n]) \leq y[n]\) 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.

### 10.3 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 [BB09] 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.

### 10.4 Practical Considerations

All 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. It therefore will be interesting to see how various SAT solvers perform on the bit-blasted version of our benchmarks.
COMPLEXITY OF FIXED-SIZE BIT-VECTOR LOGICS
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 [BB09], MathSAT [Bru+08], STP [GD07], Z3 [DMBo8], and Yices [DM06].

The theory of fixed-size bit-vector logics is investigated in several scientific works [BDL98]; [BP98]; [BS09]; [CMR97]; [Fra10], and even concrete formats for specifying such bit-vector problems exist, e.g., the SMT-LIB format [BST10] or the BTOR format [BBL08]. Working with non-fixed-size bit-vectors has been considered for instance in [ABK00]; [BP98], and more recently in [SK12a]; [SK12b], but is not further discussed in this paper. Most industrial applications (and examples in the SMT-LIB\footnote{http://www.smtlib.org/}) have fixed bit-width.

We investigate the complexity of solving fixed-size bit-vector formulas. Some papers propose such complexity results, e.g., in [BDL98], the authors consider the common quantifier-free bit-vector logic and give an argument for NP-hardness of its satisfiability problem. In [BS09], a sublogic of the previous one is claimed to be NP-complete. Interestingly, in [Bry+07], 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 [Win11]; [WHM10], the quantified case is addressed, and the satisfiability problem for this logic with uninterpreted functions is proved to be \textsf{NExpTime}-complete. However, the proof, similarly to the decision procedure in [Bry+07], only holds if we assume unary encoded bit-widths.

Parts of our paper already appeared as previous work [FKB13b]; [KFB12]. 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 [Coo+10, Page 239, Footnote 3]). In practice, the more natural and exponentially more succinct logarithmic encoding is used, such as in the SMT-LIB [BST10] or the BTOR [BBL08] format. We investigate how complexity varies if we consider either a unary
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 [BST10]:

```
(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 [FKB12], 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 [FKB13b]; [KFB12], 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 [FKB13a]; [KFB13a] 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.

### 11.3 Preliminaries

$\mathbb{N}$ denotes the set of natural numbers $\{0, 1, 2, \ldots\}$, while $\mathbb{N}^+$ denotes $\mathbb{N} \setminus \{0\}$. $\mathbb{B} := \{0, 1\}$ is the Boolean domain, thus truth values $false$ and $true$ are represented by 0 and 1, respectively.

Given $n \in \mathbb{N}^+$, let $\ln$ denote the ceiling of the logarithm of $n$ base 2: $\ln := \lceil \log_2 n \rceil$.

#### 11.3.1 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) $\neg \varphi_0$, $(\varphi_0 \land \varphi_1)$, $(\varphi_0 \lor \varphi_1)$, $(\varphi_0 \rightarrow \varphi_1)$, and $(\varphi_0 \leftrightarrow \varphi_1)$ are Boolean formulas where $\varphi_0, \varphi_1$ are Boolean formulas. A Boolean formula $\varphi$ is satisfiable iff there exists an assignment $\alpha : V \rightarrow \mathbb{B}$ to the variables, such that $\varphi$ evaluates to 1 under $\alpha$. 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 $\psi$ can be written in prenex normal form, i.e., as a closed formula $Q. \varphi$ where $Q$ is a quantifier prefix $\exists V_0 \forall V_1 \exists V_2 \forall V_3 \ldots \forall V_{m-1} \exists V_m$, the $V_i$s are pairwise disjoint sets of variables, and $\varphi$ is a Boolean formula, which is called the matrix of $\psi$. A variable $v \in V_i$ depends on a variable $v' \in V_j$ iff $i > j$. This defines a total order on the variables of $\psi$. A QBF is satisfiable iff there exist Skolem functions for its existential variables to make the formula evaluate to 1. The satisfiability problem for QBF is PSpace-complete [Pap94]; [SM73].

Instead of using totally ordered quantifiers, it is also possible to extend Boolean formulas with *Henkin quantifiers* [Hen61]. Henkin quantifiers specify variable dependencies 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 [PR79]. 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

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

where each $u_{i,j}$ is a universally quantified variable, $m_i \in \mathbb{N}$, and the matrix $\phi$ 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 iff 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 $\text{NExpTime}$-complete [PRA01]; [PR79]. Although we did not formally specify the dependencies of universal variables, this can be done by the use of Herbrand functions [BCJ12].

Throughout our paper, 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 $\text{NExpTime}$-complete logic used in reductions in the context of unary encoded bit-vector logics [Win11] is Effectively Propositional Logic (EPR) [Lew80]. However, due to its simplicity, we consider DQBF to be a better choice for our purposes.

### 11.3.2 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 [SK12a]:

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 $w \in (\{0,1\}^n)^+$ is said to be accepted by a sequential circuit $SC$ with one output $o$, 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 [PBG05; Sav70; SC85].

11.3.3 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 [ABKoo; BP98; SK12a; SK12b], working with fixed-size bit-vectors is the focus of this paper.

Let \( D_n \) denote the set of all bit-vectors of bit-width \( n \). Given \( d \in D_n \), the \( i \)th bit of \( d \) is denoted by \( d[i] \), where \( i \in \mathbb{N} \) and \( i < n \). Using vector notation, \( d \) is written as \( \langle d[n-1], \ldots, d[i], \ldots, d[0] \rangle \), 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 [BDL98; BP98; BS09; CMR97; Fra10]. Concrete formats for specifying bit-vector problems also exist, e.g., the SMT-LIB format [BST10] or the BTOR format [BBL08]. 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.

11.3.3.1 Syntax

The main objective of this section is to define bit-vector formulas. As it turns out in Definition 11.2 and 11.3, 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 11.1 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 \textit{addition} has 2 bit-vector and 0 scalar operands, while \textit{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 \textit{addition} must be of the same bit-width; the scalar operands \( i, j \) of \textit{extraction} must be less than the bit-width of the bit-vector operand and \( i \geq j \). (3) A bit-width of the resulting bit-vector is assigned to each legal combination of bit-widths and scalar values of operands.

Definition 11.1 (Signature). A signature for an operator set \( \text{Op} \) is defined as a set \( \Sigma_{\text{Op}} := \{ \langle \text{arity}_o, \text{cond}_o, \text{wid}_o \rangle \mid o \in \text{Op} \} \), where

- \( \text{arity}_o \in \mathbb{N} \times \mathbb{N} \);
- \( \text{cond}_o : (\mathbb{N}^+)^k \times \mathbb{N}^l \mapsto \mathbb{B} \) where \( \langle k, l \rangle := \text{arity}_o \);
- \( \text{wid}_o : \text{Par}_o \mapsto \mathbb{N}^+ \) where
  \( \text{Par}_o := \{ p \in (\mathbb{N}^+)^k \times \mathbb{N}^l \mid \langle k, l \rangle := \text{arity}_o, \text{cond}_o(p) \} \).

Table 11.1 shows the set of the most common operators provided by the SMT-LIB format [BST10] and the literature [BDL98; BP98; BS09; CMR97; Fra10], 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 11.1. $\mathbf{Op}$ 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 $\Sigma_{\mathbf{Op}}$ denote the common signature for $\mathbf{Op}$. Note that Table 11.1 specifies some of the syntactic properties provided by $\Sigma_{\mathbf{Op}}$ in an implicit way: the arity is completely, the condition is partly implicit.

<table>
<thead>
<tr>
<th>operation</th>
<th>condition</th>
<th>bit-width</th>
<th>alternative syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation:</td>
<td>$\text{bvnor } t_1^{[n]}$</td>
<td>$n$</td>
<td>$\sim t_1^{[n]}$</td>
</tr>
<tr>
<td>and:</td>
<td>$\text{bvand } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td>$(t_1^{[n]} \land t_2^{[n]})$</td>
</tr>
<tr>
<td>or:</td>
<td>$\text{bvor } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td>$(t_1^{[n]} \lor t_2^{[n]})$</td>
</tr>
<tr>
<td>xor:</td>
<td>$\text{bvxor } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td>$(t_1^{[n]} \oplus t_2^{[n]})$</td>
</tr>
<tr>
<td>nand:</td>
<td>$\text{bvand } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>nor:</td>
<td>$\text{bvnor } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>xnor:</td>
<td>$\text{bxnor } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>if-then-else:</td>
<td>$\text{ite } (t_1^{[1]}, t_2^{[n]}, t_3^{[n]})$</td>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>equality:</td>
<td>$\text{bvcomp } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td>$(t_1^{[n]} = t_2^{[n]})$</td>
</tr>
<tr>
<td>unsigned (u.) less than:</td>
<td>$\text{bult } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td>$(t_1^{[n]} &lt;_{u} t_2^{[n]})$</td>
</tr>
<tr>
<td>u. less than or equal:</td>
<td>$\text{bule } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>u. greater than:</td>
<td>$\text{buge } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>signed (s.) less than:</td>
<td>$\text{bult } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>s. less than or equal:</td>
<td>$\text{bule } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>s. greater than:</td>
<td>$\text{buge } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>s. greater than or equal:</td>
<td>$\text{buge } t_1^{[n]}, t_2^{[n]}$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>shift left:</td>
<td>$\text{bushl } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td>$(t_1^{[n]} \ll t_2^{[n]})$</td>
</tr>
<tr>
<td>logical shift right:</td>
<td>$\text{bvlshr } t_1^{[n]}, t_2^{[n]}$</td>
<td>$n$</td>
<td>$(t_1^{[n]} \gg_{u} t_2^{[n]})$</td>
</tr>
</tbody>
</table>

continued on next page
The simplest bit-vector expressions, or terms, are the variables and constants, as Definition 11.2 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 11.2** (Term). A bit-vector term \( t \) of bit-width \( n \in \mathbb{N}^+ \) is denoted by \( t[n] \). A term over a signature \( \Sigma_{Op} \) is defined inductively as follows:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic shift right:</td>
<td>( bvashr(t_1[n], t_2[n]) )</td>
<td>( n \quad (t_1[n] \gg s \ t_2[n]) )</td>
</tr>
<tr>
<td>extraction:</td>
<td>( extract(t_1[n], i, j) )</td>
<td>( n &gt; i \geq j \quad i - j + 1 \quad t[n][i : j] )</td>
</tr>
<tr>
<td>concatenation:</td>
<td>( concat(t_1[n], t_2[n]) )</td>
<td>( m + n \quad (t_1[n] \circ t_2[n]) )</td>
</tr>
<tr>
<td>zero extend:</td>
<td>( zeroextend(t[n], i) )</td>
<td>( n + i \quad ext_u(t[n], i) )</td>
</tr>
<tr>
<td>sign extend:</td>
<td>( signextend(t[n], i) )</td>
<td>( n + i )</td>
</tr>
<tr>
<td>rotate left:</td>
<td>( rotateleft(t[n], i) )</td>
<td>( n &gt; i \geq 0 \quad n )</td>
</tr>
<tr>
<td>rotate right:</td>
<td>( rotateright(t[n], i) )</td>
<td>( n &gt; i \geq 0 \quad n )</td>
</tr>
<tr>
<td>repeat:</td>
<td>( repeat(t[n], i) )</td>
<td>( i &gt; 0 \quad n \cdot i )</td>
</tr>
<tr>
<td>unary minus:</td>
<td>( bvneg(t[n]) )</td>
<td>( n \quad -t[n] )</td>
</tr>
<tr>
<td>addition:</td>
<td>( bvadd(t_1[n], t_2[n]) )</td>
<td>( n \quad (t_1[n] + t_2[n]) )</td>
</tr>
<tr>
<td>subtraction:</td>
<td>( bvsub(t_1[n], t_2[n]) )</td>
<td>( n \quad (t_1[n] - t_2[n]) )</td>
</tr>
<tr>
<td>multiplication:</td>
<td>( bvmul(t_1[n], t_2[n]) )</td>
<td>( n \quad (t_1[n] \cdot t_2[n]) )</td>
</tr>
<tr>
<td>unsigned division:</td>
<td>( bvurem(t_1[n], t_2[n]) )</td>
<td>( n \quad (t_1[n] / u \ t_2[n]) )</td>
</tr>
<tr>
<td>signed division:</td>
<td>( bvurem(t_1[n], t_2[n]) )</td>
<td>( n )</td>
</tr>
<tr>
<td>s. remainder with rounding to 0:</td>
<td>( bvsrem(t_1[n], t_2[n]) )</td>
<td>( n )</td>
</tr>
<tr>
<td>s. remainder with rounding to ( -\infty ):</td>
<td>( bvsmod(t_1[n], t_2[n]) )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Table 11.1: Syntax (signature) for common bit-vector operators
<table>
<thead>
<tr>
<th>term</th>
<th>condition</th>
<th>bit-width</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant: $c[n]$</td>
<td>$c \in \mathbb{N}, 0 \leq c &lt; 2^n$</td>
<td>$n$</td>
</tr>
<tr>
<td>variable: $x[n]$</td>
<td>$x$ is an identifier</td>
<td>$n$</td>
</tr>
<tr>
<td>operation: $o \left(t_1^{[n]}, \ldots, t_k^{[n]}, i_1, \ldots, i_l\right)$</td>
<td>$o \in Op$, $(k,l) := \text{arity}_o$, $t_1^{[n]}, \ldots, t_k^{[n]}$ are terms, $i_1, \ldots, i_l \in \mathbb{N}$, $\text{cond}_o(n_1, \ldots, n_k, i_1, \ldots, i_l)$</td>
<td>$\text{wid}_o(n_1, \ldots, i_l)$</td>
</tr>
<tr>
<td>uninterpreted function: $f[n] \left(t_1^{[n]}, \ldots, t_k^{[n]}\right)$</td>
<td>$f$ is an identifier, $k \in \mathbb{N}$, $t_1^{[n]}, \ldots, t_k^{[n]}$ are terms</td>
<td>$n$</td>
</tr>
</tbody>
</table>

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 11.3 (Formula).** A bit-vector formula is an expression $Q.t[l]$, where $t[l]$ is a bit-vector term, $Q$ is a quantifier prefix $Q_0x_0^{[n_0]}Q_1x_1^{[n_1]} \ldots Q_kx_k^{[n_k]}$, 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.

Without loss of generality, 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 [BDL98]; [Bru08]; [BS09]; [CMR97]; [Fra10]. 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 11.8, the same separation of a Boolean level and a bit-vector level can be made in any bit-vector formula over $\Sigma_{Op}$, assuming the common interpretation of $\Sigma_{Op}$, defined in Section 11.3.3.2.

### 11.3.3.2 Semantics

Given a signature $\Sigma_{Op}$ and an operator $o \in Op$ where $(k,l) := \text{arity}_o$, each $p := (n_1, \ldots, n_k, i_1, \ldots, i_l) \in \text{Par}_o$ 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:

$$\text{Dom}_o(p) := D_{n_1} \times \cdots \times D_{n_k} \times \{i_1\} \times \cdots \times \{i_l\}$$

$$\text{Range}_o(p) := D_{\text{wid}_o(p)}$$

In order to evaluate a term or formula, it is first necessary to interpret all the operators we use (Definition 11.4), and then to assign domain elements to free variables and to interpret uninterpreted functions (Definition 11.5).

**Definition 11.4 (Interpretation).** An interpretation of a signature $\Sigma_{Op}$ is defined as a set $\widehat{\delta}$ of functions, consisting of an $\delta$ for each $o \in Op$, such that

$$\widehat{\delta} : \bigcup_{p \in \text{Par}_o} \text{Dom}_o(p) \mapsto \bigcup_{p \in \text{Par}_o} \text{Range}_o(p)$$
where
\[ \forall p \in \text{Par}_o, d \in \text{Dom}_o(p) \cdot \hat{o}(d) \in \text{Range}_o(p) \]

Let \( \hat{\text{Op}} \) denote the common interpretation of \( \Sigma_{\text{Op}} \), detailed in Table 11.2, based on [BS09]; [But+96]; [Fra10] and the SMT-LIB. Note that Table 11.2 uses a notation that is introduced by the following definitions.

**Definition 11.5 (Model).** \( M := \langle a, \hat{F} \rangle \) is a model for a formula \( \Phi \) where

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

To facilitate the presentation, similar to [BS09]; [Fra10], we define an auxiliary bijective meta-function \( \text{nat}_n : D_n \to [0, 2^n - 1] \). Given a bit-vector \( d \in D_n \), \( \text{nat}_n(d) := \sum_{i=0}^{n-1} 2^i d[i] \). We also introduce the inverse meta-function \( bvn := \text{nat}_n^{-1} \).

**Definition 11.6 (Evaluation).** Given a signature \( \Sigma_{\text{Op}, \circ} \), a formula \( \Phi \) over \( \Sigma_{\text{Op}, \circ} \), an interpretation \( \hat{\text{Op}} \) of \( \Sigma_{\text{Op}, \circ} \), and a model \( M := \langle a, \hat{F} \rangle \) for \( \Phi \), \( \Phi \) can be evaluated to either 0 or 1, by using the inductive definition of the evaluation function \( \llbracket \cdot \rrbracket_M^{\hat{\text{Op}}} \), as follows:

| Constant | \[ c^{[n]} \]^{\hat{\text{Op}}}_M := bvn(c) |
| Variable | \[ x^{[n]} \]^{\hat{\text{Op}}}_M := a(x) |
| Operation | \[ o(t_1^{[n_1]}, \ldots, t_k^{[n_k]}, t_1, \ldots, t_l) \]^{\hat{\text{Op}}}_M := \hat{o} \left( \left[ t_1^{[n_1]} \right]^{\hat{\text{Op}}}_M, \ldots, \left[ t_k^{[n_k]} \right]^{\hat{\text{Op}}}_M, t_1, \ldots, t_l \right) |
| Uninterpreted Function | \[ f^{[n]}(t_1^{[n_1]}, \ldots, t_k^{[n_k]}) \]^{\hat{\text{Op}}}_M := f \left( \left[ t_1^{[n_1]} \right]^{\hat{\text{Op}}}_M, \ldots, \left[ t_k^{[n_k]} \right]^{\hat{\text{Op}}}_M \right) |
| Quantifiers | \[ \forall x^{[n]} \Phi \]^{\hat{\text{Op}}}_M := \bigwedge_{d \in D_n} \left[ \Phi \right]^{\hat{\text{Op}}}_{\langle a, (x^{[n]} \leftarrow d), \hat{F} \rangle} |
| | \[ \exists x^{[n]} \Phi \]^{\hat{\text{Op}}}_M := \bigvee_{d \in D_n} \left[ \Phi \right]^{\hat{\text{Op}}}_{\langle a, (x^{[n]} \leftarrow d), \hat{F} \rangle} |

As mentioned before, the common interpretation \( \hat{\text{Op}} \) is given in Table 11.2. In the table, we omit the interpretation and the model for evaluation. Furthermore, we use two abbreviations:

\[ \text{msb}(t^{[n]}) := \lfloor t \rfloor[n - 1] \]
\[ \text{abs}(t^{[n]}) := \begin{cases} -t & \text{if } \text{msb}(t) \\ t & \text{otherwise} \end{cases} \]

| BV Notation | \[ \neg t^{[n]} \] := bvn \left( \sum_{i=0}^{n-1} 2^i \left( \neg \left[ t \right][i] \right) \right) |
| BV And | \[ t_1^{[n]} \& t_2^{[n]} \] := bvn \left( \sum_{i=0}^{n-1} 2^i \left( \left[ t_1 \right][i] \& \left[ t_2 \right][i] \right) \right) |
| BV Or | \[ t_1^{[n]} \mid t_2^{[n]} \] := bvn \left( \sum_{i=0}^{n-1} 2^i \left( \left[ t_1 \right][i] \lor \left[ t_2 \right][i] \right) \right) |

Continued on next page.
\textbf{bvxor:}\n\[ t_1[n] \oplus t_2[n] := \text{bv}_n \left( \sum_{i=0}^{n-1} 2^i \left( -\left[ t_1 \right] i \Rightarrow \left[ t_2 \right] i \right) \right) \]

\textbf{bvnand:}\n\[ \text{bvnand} \left( t_1[n], t_2[n] \right) := \left[ \neg \left( t_1[n] \& t_2[n] \right) \right] \]

\textbf{bvnor:}\n\[ \text{bvnor} \left( t_1[n], t_2[n] \right) := \left[ \neg \left( t_1[n] \lor t_2[n] \right) \right] \]

\textbf{bvxnor:}\n\[ \text{bvxnor} \left( t_1[n], t_2[n] \right) := \left[ \neg \left( t_1[n] \oplus t_2[n] \right) \right] \]

\textbf{ite:}\n\[ \text{ite} \left( t_1[1], t_2[n], t_3[n] \right) := \begin{cases} t_2 & \text{if } t_1 \\ t_3 & \text{otherwise} \end{cases} \]

\textbf{bvcomp:}\n\[ t_1[n] = t_2[n] := \text{bv}_1 \left( \text{nats}_n \left( \left[ t_1 \right] \right) = \text{nats}_n \left( \left[ t_2 \right] \right) \right) \]

\textbf{bvult:}\n\[ t_1[n] < \text{u} t_2[n] := \text{bv}_1 \left( \text{nats}_n \left( \left[ t_1 \right] \right) < \text{nats}_n \left( \left[ t_2 \right] \right) \right) \]

\textbf{bvule:}\n\[ \text{bvule} \left( t_1[n], t_2[n] \right) := \left[ \neg \left( t_2 < \text{u} t_1 \right) \right] \]

\textbf{bvugt:}\n\[ \text{bvugt} \left( t_1[n], t_2[n] \right) := \left[ t_2 < \text{u} t_1 \right] \]

\textbf{bvuge:}\n\[ \text{bvuge} \left( t_1[n], t_2[n] \right) := \left[ \neg \text{bvule} \left( t_2, t_1 \right) \right] \]

\textbf{bvslt:}\n\[ \text{bvslt} \left( t_1[n], t_2[n] \right) := \text{bv}_1 \left( \left( \text{msb} \left( t_1 \right) \land \neg \text{msb} \left( t_2 \right) \right) \lor \left( \left( \text{msb} \left( t_1 \right) \leftrightarrow \text{msb} \left( t_2 \right) \right) \land t_1 < \text{u} t_2 \right) \right) \]

\textbf{bvle:}\n\[ \text{bvle} \left( t_1[n], t_2[n] \right) := \left[ \neg \text{bvslt} \left( t_2, t_1 \right) \right] \]

\textbf{bvsge:}\n\[ \text{bvsge} \left( t_1[n], t_2[n] \right) := \left[ \neg \text{bvule} \left( t_2, t_1 \right) \right] \]

\textbf{bvshl:}\n\[ t_1[n] \ll t_2[n] := \text{bv}_n \left( \text{nats}_n \left( \left[ t_1 \right] \right) \cdot 2^k \mod 2^n \right) \text{ where } k := \text{nats}_n \left( \left[ t_2 \right] \right) \]

\textbf{bvshr:}\n\[ t_1[n] \gg \text{u} t_2[n] := \text{bv}_n \left( \left[ \text{nats}_n \left( \left[ t_1 \right] \right) / 2^k \right] \right) \text{ where } k := \text{nats}_n \left( \left[ t_2 \right] \right) \]

\textbf{bvshr:}\n\[ t_1[n] \gg \text{s} t_2[n] := \begin{cases} \left[ \neg \left( t_1 \gg \text{u} t_2 \right) \right] & \text{if } \text{msb} \left( t_1 \right) \\ \left[ t_1 \gg \text{s} t_2 \right] & \text{otherwise} \end{cases} \]

\textbf{extract:}\n\[ \left[ i[n] \mid j : i \right] := \text{bv}_{i-j+1} \left( \left( \text{nats}_n \left( \left[ \cdot \right] \right) / 2^j \right) \mod 2^i \right) \]

\textbf{concat:}\n\[ t_1[n] \circ t_2[n] := \text{bv}_{m+n} \left( 2^n \text{nats}_m \left( \left[ t_1 \right] \right) + \text{nats}_n \left( \left[ t_2 \right] \right) \right) \]

\textbf{zero extend:}\n\[ \text{ext}_0 \left( i[n], i \right) := \text{bv}_{n+i} \left( \text{nats}_n \left( \left[ \cdot \right] \right) \right) \]

\textbf{sign extend:}\n\[ \text{sign} \text{extend} \left( i[n], i \right) := \begin{cases} \text{bv}_{n+i} \left( 2^{n+i} - 2^n + \text{nats}_n \left( \left[ \cdot \right] \right) \right) & \text{if } \text{msb} \left( t \right) \\ \text{ext}_0 \left( i[n], i \right) & \text{otherwise} \end{cases} \]

\textbf{rotate left:}\n\[ \text{rotate left} \left( i[n], i \right) := \begin{cases} \left[ t \left[ n-i-1 : 0 \right] \circ t \left[ n-1 : n-i \right] \right] \text{ if } n = 1 \lor i = 0 \\ \left[ t \left[ i-1 : 0 \right] \circ t \left[ n+1 : i \right] \right] \text{ if } n = 1 \lor i = 0 \end{cases} \]

\textbf{rotate right:}\n\[ \text{rotate right} \left( i[n], i \right) := \begin{cases} \left[ t \left[ n-i-1 : 0 \right] \circ t \left[ n-1 : n-i \right] \right] \text{ if } n = 1 \lor i = 0 \\ \left[ t \left[ i-1 : 0 \right] \circ t \left[ n+1 : i \right] \right] \text{ if } n = 1 \lor i = 0 \end{cases} \]
The complexity of fixed-size bit-vector logics.

iff the following condition holds: there exists a satisfying model for $\Phi$.

For the rest of this paper, we fix the operator set we use to $\mathcal{O}_p$ with the signature $\Sigma_{\mathcal{O}_p}$ (Table 11.1) and the interpretation $\hat{\mathcal{O}}_p$ (Table 11.2), 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 [BDL98]; [Bru08]; [BS97]; [CMR97]; [Fra10]. Notice, however, that relational operations can occur not only at the Boolean level, but even below that, due to Definition 11.2, 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 11.8 (Flat Form).** A bit-vector formula $\Phi$ is in flat form iff it does not contain any nested relational operations.

It is easy to see that any bit-vector formula $\Phi$ can be translated into flat form with only linear growth in formula size. For each nested relational operation in $\Phi$, 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 paper, 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 11.9 (Logic with Unary and Binary Encoding).** Given a bit-vector logic $L$, let $L_1$ and $L_2$ denote the logic $L$ using unary and binary encoding on all the scalars in formulas, respectively.

In the rest of this paper, 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 11.10 (Formula Size).** Suppose we are given a bit-vector logic $L$ and a formula $\Phi \in L$, with $\Phi := \Phi_0 x_0[n_0] \Phi_1 x_1[n_1] \ldots \Phi_k x_k[n_k] t[1]$. The size of $\Phi$ is defined as $|\Phi| := |x_0[n_0]| + \ldots + |x_k[n_k]| + |t[1]|$.

The expression $|t[n]|$ denotes the size of a term $t[n]$ and is defined as follows:
11.4 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 [KFB12].

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 [BDL98] and [BS09]):

**Proposition 11.11.** QF\(_{BV1}\) is NP-complete.

*Proof.* Recall that QF\(_{BV1}\) uses the Common Operator Framework. Therefore, 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. i.e., the class of Boolean formulas is a subset of QF\(_{BV1}\).

Adding uninterpreted functions to QF\(_{BV1}\) does not increase complexity:

**Proposition 11.12.** QF\(_{UFBV1}\) 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 adding (at most quadratic many) Ackermann constraints (see, e.g., [KS08, Chapter 3.3.1]). Therefore, QF\(_{UFBV1}\) can be polynomially translated into QF\(_{BV1}\). The other direction follows from the fact that QF\(_{BV1}\) ⊂ QF\(_{UFBV1}\).

Adding quantifiers to QF\(_{BV1}\) yields the following complexity (see also [Coo+10]):

**Proposition 11.13.** BV1 is PSpace-complete.

*Proof.* By bit-blasting, BV1 can be reduced to Quantified Boolean Formulas (QBF), which is PSpace-complete. Hardness follows from the fact that QBF ⊂ BV1 (following the same argument as in Proposition 11.11).

Adding quantifiers to QF\(_{UFBV1}\) increases complexity exponentially:

**Proposition 11.14.** UFBV1 is NExpTime-complete (see [Win11]).

*Proof.* The Effectively Propositional Logic (EPR), is a common NExpTime-complete [Lew80] logic, and can be reduced to UFBV1 [Win11, Theorem 7]. For completing the other direction, apply the reduction in [Win11, Theorem 7] combined with bit-blasting of the bit-vector operations.

---

This kind of result is often called unary NP-completeness [GJ78].
11.5 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 [KFB12], 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).\footnote{Recall that only a variable, a constant, or an uninterpreted function can have explicit bit-width.} Given a formula \( \Phi \), let \( \max_{bw} (\Phi) \) denote the maximal scalar in \( \Phi \) and, furthermore, let \( \size_{bw} (\Phi) \) denote the number of scalars in \( \Phi \).

**Definition 11.15 (Scalar-Bounded Formula Set).** An infinite set \( S \) of bit-vector formulas is (polynomially) scalar-bounded, iff there exists a polynomial function \( p : \mathbb{N} \rightarrow \mathbb{N} \) such that \( \forall \Phi \in S. \max_{bw} (\Phi) \leq p(\size_{bw} (\Phi)) \).

**Proposition 11.16.** 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 \( \Phi' \) denote the formula obtained through re-encoding scalars in \( \Phi \) to unary format. For the size of \( \Phi' \), the following upper bound holds:\( \size_{bw} (\Phi) \cdot \max_{bw} (\Phi) + |\Phi| \). Note that \( \size_{bw} (\Phi) \cdot \max_{bw} (\Phi) \) is an upper bound on the sum over the sizes of all the scalars in \( \Phi' \). The second term, \( |\Phi| \), represents an upper bound for the part of \( \Phi \) that does not contain any scalars. Since \( S \) is scalar-bounded, it holds that

\[
|\Phi'| \leq \size_{bw} (\Phi) \cdot \max_{bw} (\Phi) + |\Phi| \\
\leq \size_{bw} (\Phi) \cdot p(\size_{bw} (\Phi)) + |\Phi| \leq |\Phi| \cdot p(\size_{bw} (\Phi)) + |\Phi|
\]

where \( p \) is a polynomial function. Therefore, the size of \( \Phi' \) is polynomial in the size of \( \Phi \). \( \square \)

By applying this proposition to the logics of Section 11.3.3 together with the results from Section 11.4, we get:

**Corollary 11.17.** Suppose we are given a scalar-bounded set \( S \) of bit-vector formulas. If \( S \subseteq QF_{BV2} \) (and even if \( S \subseteq QF_{UFBV2} \)), then \( S \in \text{NP} \). If \( S \subseteq BV2 \), then \( S \in \text{PSPACE} \). If \( S \subseteq UFBV2 \), then \( S \in \text{NEXP} \).

11.6 Quantifier-Free Logics with Binary Encoding

Our main contribution in [FKB13b]; [KFB12] 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 11.18, and continuing with three fragments of QF_BV2 in Theorem 11.19, 11.20, 11.21. All these theorems reference separate lemmas, which we introduce afterwards.

**Theorem 11.18.** QF_BV2 is \( \text{NEXP} \)-complete [KFB12].
Proof. It is easy to see that $\text{QF}_{\text{BV}2} \in \text{NE}^\text{XP}$, since a $\text{QF}_{\text{BV}2}$ formula can be translated exponentially to $\text{QF}_{\text{BV}1} \in \text{NP}$ (Proposition 11.11), by applying a simple unary re-encoding to all the scalars in the formula. $\text{NE}^\text{XP}$-hardness of $\text{QF}_{\text{BV}2}$ is a direct consequence of Lemma 11.23, in which a fragment of $\text{QF}_{\text{BV}2}$ is proved to be $\text{NE}^\text{XP}$-hard. □

Note that UFBV1 and $\text{QF}_{\text{BV}2}$ 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 [FKB13b], we investigated the complexity of the satisfiability problem for the following three fragments of $\text{QF}_{\text{BV}2}$, which only allow a restricted set of bit-vector operations in formulas:

- $\text{QF}_{\text{BV}2,c}$: only bitwise operations, equality, and left shift by constant, i.e., $t^n \ll c^n$ where $c$ is a constant, are allowed.

- $\text{QF}_{\text{BV}2,1}$: only bitwise operations, equality, and left shift by 1, i.e., $t^n \ll t^n$, are allowed.

- $\text{QF}_{\text{BV}2,bw}$: only bitwise operations and equality are allowed.

**Theorem 11.19.** $\text{QF}_{\text{BV}2,c}$ is $\text{NE}^\text{XP}$-complete [FKB13b].

**Proof.** In Lemma 11.23, we give a reduction from DQBF (which is $\text{NE}^\text{XP}$-complete) to $\text{QF}_{\text{BV}2,c}$. This shows the $\text{NE}^\text{XP}$-hardness of $\text{QF}_{\text{BV}2,c}$. The fact that $\text{QF}_{\text{BV}2,c} \in \text{NE}^\text{XP}$ directly follows from Theorem 11.18. □

**Theorem 11.20.** $\text{QF}_{\text{BV}2,1}$ is $\text{PSPACE}$-complete [FKB13b].

**Proof.** In Lemma 11.24, we give a reduction from QBF (which is $\text{PSPACE}$-complete) to $\text{QF}_{\text{BV}2,1}$. This shows the $\text{PSPACE}$-hardness of $\text{QF}_{\text{BV}2,1}$. In Lemma 11.25, we then prove $\text{PSPACE}$-inclusion by giving a reduction from satisfiability for $\text{QF}_{\text{BV}2,1}$ to the model checking problem for sequential circuits. Symbolic model checking for sequential circuits is $\text{PSPACE}$-complete as well [PBG05]; [Sav70]; [SC85].

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 11.9.

**Theorem 11.21.** $\text{QF}_{\text{BV}2,bw}$ is $\text{NP}$-complete [FKB13b].

**Proof.** Since Boolean formulas are a subset of $\text{QF}_{\text{BV}2,bw}$, NP-hardness follows directly. To show that $\text{QF}_{\text{BV}2,bw} \in \text{NP}$, we give a reduction from $\text{QF}_{\text{BV}2,bw}$ to a scalar-bounded set of formulas $S \subset \text{QF}_{\text{BV}2}$ in Lemma 11.26. The claim then follows from Corollary 11.17. □

As already hinted in Proposition 11.12, adding uninterpreted functions to all quantifier-free logics we discussed so far does not affect complexity. We formalize this in the following proposition:

**Proposition 11.22.** $\text{QF}_{\text{UFBV}2}$ and $\text{QF}_{\text{UFBV}2,c}$ are $\text{NE}^\text{XP}$-complete, $\text{QF}_{\text{UFBV}2,1}$ is $\text{PSPACE}$-complete, and $\text{QF}_{\text{UFBV}2,bw}$ is $\text{NP}$-complete [FKB13b]; [KFB12].

**Proof.** Apply the same arguments as were used in Proposition 11.12. □

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

**Lemma 11.23.** DQBF can be reduced to $\text{QF}_{\text{BV}2,c}$ [FKB13b]; [KFB12].

**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

\[ \text{binmagic}(2^m, 2^n) := \begin{array}{cccccccc}
0 & \cdots & 0 & 1 & \cdots & 1 & \cdots & 0 & \cdots & 1 \\
2^m & \cdots & 2^m & \cdots & 2^m & \cdots & 2^m & \cdots & 2^m
\end{array} \]

Note that these bit masks correspond to the so-called binary magic numbers (or magic masks in \cite[p. 141]{knuth}), and can arithmetically be calculated in the following way (actually as the result of a geometric sum):

\[ \text{binmagic}(2^m, 2^n) := \frac{2^{(2^n)} - 1}{2^{(2^m)} + 1} \]

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

\[
\begin{align*}
    b^{[2^n]} &= \sim 0^{[2^n]} / u ((1 \ll 2^n) + 1) \\
    b \cdot ((1 \ll 2^n) + 1) &= \sim 0^{[2^n]} \\
    (b \ll 2^n) + b &= \sim 0^{[2^n]}
\end{align*}
\]

Addition can be eliminated easily as follows, by using two’s complement representation for \(-1\) and \(-b\):

\[
\begin{align*}
    (b \ll 2^n) + b &= -1 \\
    b \ll 2^n &= -1 - b \\
    b \ll 2^n &= -1 + \sim b + 1 \\
    b \ll 2^n &= \sim b
\end{align*}
\]

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 11.11.1.

**THE REDUCTION.** Let \( \psi := Q.\phi \) denote a DQBF with quantifier prefix \( Q \) and matrix \( \phi \). Further, 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 \( \psi \) to a QFBV2 \( \ll \) formula \( \Phi \) by eliminating the quantifier prefix and translating the matrix \( \phi \) 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 \( \Phi_0 \).

**STEP 2.** We now construct \( \Phi_1 \) by adding new constraints to \( \Phi_0 \). 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):

\[ U_m \ll 2^m = \sim U_m \]
STEP 3. Next, we construct $\Phi_2$ by adding another set of constraints to $\Phi_1$. For each existential variable $e_{m'} \in \{e_0, \ldots, e_{m'-1}\}$, 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:

$$E_{m'} \& \sim U_m = (E_{m'} \ll 2^m) \& \sim U_m$$ (11.1)

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

POLYNOMIALITY. Note that all the scalars and constants in $\Phi$ 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 $U_{m'}^n, E_{m'}^n$. 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 $\psi$ and the resulting bit-vector formula $\Phi$ are equisatisfiable we consider the individual steps separately.

In Step 1, we used the matrix $\phi$ of $\psi$ to create a bit-vector formula with the same underlying structure which is true iff each row evaluates to 1. Since all the bits of bit-vectors in $\Phi_0$ are independent of each other and there are no additional constraints on the bit-vector variables, $\Phi_0$ is satisfiable iff the Boolean formula $\phi$ is satisfiable.

Now consider the bit-vector variables $U_m$ after constructing $\Phi_1$ by adding the constraints of Step 2. In the following, we formalize the well-known fact that the combination of all the $U_m$'s corresponds exactly to all possible assignments to the universal variables of $\psi$. 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

$$[U_m] [b_i] \neq [U_m] [b_j] \text{ and } [U_k] [b_i] = [U_k] [b_j], \forall k \neq m.$$ (11.2a)

$$[U_k] [b_i] = 0 \text{ if } [U_m] [b_i] = 0 \text{ and } [U_k] [b_i] = 1 \text{ if } [U_m] [b_i] = 1.$$ (11.2b)

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

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

By defining $b_j$ this way, Eqn. (11.2a) and (11.2b) 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 $c$. Eqn. (11.2a) 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

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

With a similar argument, we can show that Eqn. (11.2b) holds:

$$[U_k] [b_i] = [U_k] [b_i - 2^m] \text{ and } [U_k] [b_i] = [U_k] [b_i + 2^m] \text{ holds if } [U_k] [b_i] = 0 \text{ and } [U_k] [b_i] = 1,$$
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 \( \psi \).

For a given assignment \( a \in \{0,1\}^n \), the existence of such a previously defined \( b_i \) for every \( U_m \) and \( b_i \) allows us to iteratively find a \( b_k \) such that \( \left( [U_0] [b_k], \ldots, [U_{n-1}] [b_k] \right) = a \). Thus, we have a bijective mapping from the universal assignments \( a \) for \( \psi \) to the bit-indices \( b_i \) for \( \Phi_1 \). Up to this point, each bit-vector \( E_m' \) can basically still take \( 2(2^r) \) different values in \( \Phi_1 \). The value of each individual bit \( [E_m'] [b_i] \) corresponds to the value that \( e_m' \) takes under a given universal assignment \( a \in \{0,1\}^n \). 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, \( \Phi_1 \) is satisfiable iff 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 \( E_m' \) that correspond to Skolem-functions which do not respect the dependency scheme of \( \psi \). Whenever \( e_m' \) does not depend on a universal variable \( u_m \), we add the constraint of Eqn. (11.1). 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 \( a, \beta \in \{0,1\}^n \) where \( a[k] = \beta[k] \) for all \( k \neq m \). Exactly this is enforced by our constraint. Looking at the corresponding bit-indices \( b_i \) and \( b_j \) for \( a \) and \( \beta \), respectively, our constraint for independence ensures that \( [E] [b_i] = [E] [b_j] \). More precisely, Eqn. (11.1) 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, \( \Phi_2 \) is now respecting the dependency scheme and therefore \( \Phi \) is satisfiable iff the original DQBF \( \psi \) is satisfiable.

**Lemma 11.24.** QBF can be reduced to QF$_{\text{BV2} \leq 1}$ [FKB13b].

**Proof.** To show the PSpace-hardness of QF$_{\text{BV2} \leq 1}$, we give a reduction from QBF, similar to the one from DQBF to QF$_{\text{BV2} \leq c}$ that we used in Lemma 11.23.

For our reduction, we again use the binary magic numbers. Note that, in Lemma 11.23, we used left shift by constant to construct the binary magic numbers. This is not permitted in QF$_{\text{BV2} \leq 1}$. 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 want to 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 \( 0 \leq c < 2^n \) 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 \( m \in \{0, \ldots, n-1\} \):

\[
\left( \bigwedge_{0 \leq i < m} b_i \right) \oplus b_m = b_m \ll 1
\]

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 iff 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 11.23. An example can be found in Appendix 11.11.2.
**The Reduction.** Let \( \psi := Q \phi \) denote a QBF with quantifier prefix \( Q \) and matrix \( \phi \). Since \( \psi \) is a QBF (in contrast to DQBF in Lemma 11.23), we know that \( Q \) defines a total order on the universal variables. We assume the universal variables \( u_0, \ldots, u_{n-1} \) of \( \phi \) are ordered according to their appearance in \( Q \), with \( u_0 \) and \( u_{n-1} \) being the innermost and outermost variable, respectively. Translate \( \psi \) to a QF,BV2\( \leq 1 \) formula \( \Phi \) by eliminating the quantifier prefix and translating the matrix as follows:

**Step 1.** Replace all Boolean constants 0 and 1 with \( 0^{[2^m]} \) and \( \sim 0^{[2^m]} \), all Boolean universal variables \( u_m \) and existential variables \( e_m \) with bit-vector variables \( U_m^{[2^m]} \) and \( E_m^{[2^m]} \), and all logical connectives with corresponding bitwise bit-vector operators (e.g., \( \land \) with \( \& \) ). Let \( t^{[2^m]} \) denote the bit-vector term generated so far. Extend it to the formula \( t = \sim 0^{[2^m]} \). We refer to this as \( \Phi_0 \).

**Step 2.** We now construct \( \Phi_1 \) by adding new constraints to \( \Phi_0 \). For each universal variable \( u_m \in \{ u_0, \ldots, u_{n-1} \} \), in order to assign a binary magic number to \( U_m^{[2^m]} \), add the following equality (i.e., conjunct it with the current formula):

\[
\left( \bigwedge_{0 \leq i < m} U_i \right) \oplus U_m = U_m \ll 1
\]

**Step 3.** Next, we construct \( \Phi_2 \) by adding another set of constraints to \( \Phi_1 \). For each existential variable \( e_m' \in \{ e_0, \ldots, e_{n-1} \} \) depending on the universal variables \( \text{Deps} (e_m') = \{ u_m, \ldots, u_{n-1} \} \), with \( u_m \) being the innermost universal variable that \( e_m' \) depends on, check the following conditions:

- If \( \text{Deps} (e_m') = \emptyset \), add the equality:
  \[
  E_m' \land \sim 1 = E_m' \ll 1
  \]  
  \[
  (11.3)
  \]

- Otherwise, if \( m \neq 0 \), add the two equalities:
  \[
  U_m' = (U_m \ll 1) \oplus U_m
  \]
  \[
  E_m' \land U_m' = (E_m' \ll 1) \land U_m'
  \]  
  \[
  (11.4)
  (11.5)
  \]

Finally, we define \( \Phi := \Phi_2 \).

Step 1 and Step 2 are equal to those of Lemma 11.23 apart from the fact that a different construction for the binary magic numbers is used.

Again, each bit-index of \( \Phi \) corresponds to the evaluation of \( \psi \) under a specific assignment to the universal variables \( u_0, \ldots, u_{n-1} \), and, by construction of \( U_0^{[2^m]}, \ldots, U_{n-1}^{[2^m]} \), all possible assignments are considered. Eqn. (11.4) creates a bit-vector \( U_m^{[2^m]} \) for which each bit equals to 1 iff the corresponding universal variable changes its value from one universal assignment to the next. In contrast to Lemma 11.23, 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, Eqn. (11.4) does not have to be added multiple times, if several existential variables depend on the same universal variable. Eqn. (11.5) and Eqn. (11.3) ensure that the corresponding bits of \( E_m^{[2^m]} \) satisfy the dependency scheme of \( \psi \) by only allowing the value of \( e_m' \) to change if an outer universal variable takes a different value. If \( \text{Deps} (e_m') = \{ u_0, \ldots, u_{n-1} \} \), i.e., if \( e_m' \) depends on all universal variables, Eqn. (11.4) evaluates to \( U_0 = 0^{[2^m]} \), and, as a consequence, Eqn. (11.5) simplifies to \( \text{true} \). 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 [Don+02]. See also [PBG05] for a translation from QBF to sequential circuits. \( \square \)
Lemma 11.25. QF_BV_{\leq 1} can be reduced to sequential circuits [FKB13b].

Proof. In [SK12a]; [SK12b], the authors give a polynomial translation from quantifier-free Presburger arithmetic with bitwise operations (QFPA\texttt{arr} [SS07]) 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 [KMS00]. We can adopt their approach in order to construct a translation for QF_BV_{\leq 1}. Related work, introducing an automata-based representation for Presburger Arithmetic (without bitwise operations), can be found in [WB95].

For the most part, the basic structure as well as the arguments used throughout the reduction are the same as in [SK12a]; [SK12b]. To keep the proof compact, we therefore focus on pointing out the changes compared to their earlier work and regularly refer to [SK12a]; [SK12b] for the technical details.

As mentioned, the main difference between QFPA\texttt{arr} and QF_BV_{\leq 1} is the fact that bit-vectors of arbitrary, non-fixed, size are allowed in QFPA\texttt{arr} while all bit-vectors contained in QF_BV_{\leq 1} have a fixed bit-width. We now give the reduction.

Given $\Phi \in \text{QF_BV}_{\leq 1}$ 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 [SK12a]; [SK12b], we further assume w.l.o.g. that $\Phi$ only consists of logical combinations of three types of atomic expressions: $t_1^{[n]} = t_2^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \leq 1^{[n]}$.

Similar to generating a formula in flat form (Definition 11.8), it is easy to see that any QF_BV_{\leq 1} formula can be written like this with only linear growth in size by introducing Tseitin variables.

We then encode each equality in $\Phi$ into an individual sequential circuit separately. In the following, those are referred to as atomic sequential circuits. Compared to [SK12a]; [SK12b], two modifications for the construction of an atomic sequential circuits are needed. First, we need to give a translation of $x = y \leq 1$ to sequential circuits. This can be done, for example, by using the sequential circuit for $x = 2 \cdot y$ in QFPA\texttt{arr}. 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 in the definition of sequential circuits in Section 11.3.2, this can be easily realized by adding a combinational 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 combinational circuit. Further, 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 combinational circuit realizing the function

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

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 L\texttt{t} gates, i.e., polynomially in the size of $\Phi$. 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 [SK12a], we further assume that the input streams for all variables start with the least significant bit. As already pointed out by the authors...
in [SK12a], 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 $\Phi$, in the same way as for non-fixed bit-width in [SK12a]; [SK12b]. Due to the counters being part of the atomic sequential circuits, we ensure that for every input stream $x_j$, that represents a bit-vector variable of bit-width $n_j$, only the first $n_j$ bits of $x_i$ influence the result of the whole circuit.

\[ \square \]

**Lemma 11.26.** $\text{QF}_{\text{BV2}_{bw}} \in \text{NP}$ [FKB13b].

**Proof.** To show that $\text{QF}_{\text{BV2}_{bw}} \in \text{NP}$, we give a reduction from $\text{QF}_{\text{BV2}_{bw}}$ to a scalar-bounded set of formulas $S$. With $S \subseteq \text{QF}_{\text{BV2}}$, the claim then follows from Corollary 11.17. An example, that combines further results from Section 11.7.2, can be found in Appendix 11.11.3.

Suppose we are given a formula $\Phi \in \text{QF}_{\text{BV2}_{bw}}$ in flat form (Definition 11.8). We assume that any inequality $t_1^{[n]} \neq t_2^{[n]}$ in $\Phi$ is expressed by $\sim \left( t_1^{[n]} = t_2^{[n]} \right)$. If $\Phi$ contains any constants $c^{[n]}$ where $c \neq 0$, we remove those constants in a (polynomial) pre-processing step. Let $c_{\max}^{[m]} := b_{k-1} \ldots b_1 b_0$ be the largest constant in $\Phi$ 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 $\Phi$ with

\[ t_1^{[1]} = t_2^{[1]} \wedge \ldots \wedge t_{1,n-1}^{[1]} = t_{2,n-1}^{[1]}, \]

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

\[ t_1^{[1]} = t_2^{[1]} \wedge \ldots \wedge t_{1,k-1}^{[1]} = t_{2,k-1}^{[1]} \wedge t_{1,n-k}^{[n-k]} = t_{2,n-k}^{[n-k]}. \]

For $0 \leq i < \min \{n,k\}$, we use $t_{i,1}^{[1]} = t_{i,2}^{[1]}$ to express the $i$th row of the original equality. For constructing the terms $t_{1,j}^{[1]}$ and $t_{2,j}^{[1]}$, (i) replace each occurrence of a variable $x_i^{[n]}$ 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 $n > k$, $t_{1,n-k}^{[n-k]} = t_{2,n-k}^{[n-k]}$ represents the remaining $n-k$ rows of the original equality corresponding to the most significant bits. For constructing $t_{1,n-k}^{[n-k]}$ and $t_{2,n-k}^{[n-k]}$, (i) replace each occurrence of a variable $x_i^{[n]}$ with the variable $x_i^{[n-k]}$, and (ii) replace each constant $c^{[n]}$ with $0^{[n-k]}$.

Since this pre-processing step is logarithmic in the value of $c_{\max}$, it is polynomial in $|\Phi|$. Without loss of generality, we now assume that $\Phi$ does not contain any bit-vector constants different from $0^{[n]}$.

We now construct a formula $\Phi'$ by reducing the bit-widths of all bit-vector terms in $\Phi$. We use $cn_{\text{eq}}(\Phi)$ to denote the number of equalities in $\Phi$. Each term $t^{[n]}$ in $\Phi$ is then replaced with a term $t'^{[n]}$, with $n' := \min \{n, cn_{\text{eq}}(\Phi) \} \leq |\Phi|$. Apart from this, $\Phi'$ is exactly the same as $\Phi$. As a consequence, $\text{max}_{\text{bw}}(\Phi') \leq |\Phi|$. The set of formulas constructed in this way is scalar-bounded according to Definition 11.15.

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 $\Phi$ we can also construct a satisfying assignment to $x_1^{[n_1]}, \ldots, x_k^{[n_k]}$ for $\Phi'$ and vice versa.

It is easy to see that whenever we have a satisfying assignment $\alpha$ for $\Phi'$, we can construct a satisfying assignment $\alpha'$ for $\Phi$. 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 $\Phi$ and $\Phi'$, respectively. As a direct consequence, $\alpha(\Phi) = \alpha'(\Phi') = 1$. 

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

In each variable \( x_i[m_i], i \in \{1, \ldots, k\} \), 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

\[
|M_i| \leq \min\{n_i, \text{cnt}_{eq}(\Phi)\} \\
M_i = M_j \quad \forall j \in \{1, \ldots, k\} \text{ with } n_i = n_j
\]

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 \( |M_i| = \min\{n_i, \text{cnt}_{eq}(\Phi)\} \forall i \in \{1, \ldots, k\} \).

Finally, we can directly construct \( a' \) 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 is obviously still satisfied when only a subset of all bit-indices is considered.

\[\square\]

**Remark 11.27.** A similar proof can be found in [Joh01]; [Joh02]. While the focus of [Joh01]; [Joh02] 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 [Joh01]; [Joh02] 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.

**11.7 Fragment 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 \( \text{QF\_BV}_{2\text{\_cc}} \) being \( \text{NExpTime-complete} \), any common operation can extend this fragment without increasing complexity; the full extension is exactly the definition of \( \text{QF\_BV} \). It is more interesting to investigate which operations can be added to \( \text{QF\_BV}_{2\text{\_bw}} \) and \( \text{QF\_BV}_{2\text{\_c1}} \) while still remaining in \( \text{NP} \) and \( \text{PSPACE} \), 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 the satisfiability problem for \( \text{QF\_BV}_{2\text{\_bw}} \) 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 makes the satisfiability problem PSpace-hard (Lemma 11.24) or NExpTime-hard (Lemma 11.23), 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.

### 11.7.1 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:

```
replace with: \quad \text{term}_2
add assertion(s): \quad \text{formula}_1, \ldots, \text{formula}_k
```

By this description, we want to express that we replace a term \( \text{term}_1 \) in a formula \( \Phi \) with \( \text{term}_2 \), and simultaneously add all the quantifier-free formulas \( \text{formula}_1, \ldots, \text{formula}_k \) to \( \Phi \) (i.e., conjunct each of them with the matrix of \( \Phi \)). We call \( \text{formula}_1, \ldots, \text{formula}_k \) the assertions in the definition. All the variables that do not occur in \( \text{term}_1 \), but do occur in any of the expressions \( \text{term}_2, \text{formula}_1, \ldots, \text{formula}_k \) are considered as Tseitin variables, i.e., they are assumed to be added to \( \Phi \) 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., \( \text{bv\text{and}} \) 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:

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

### 11.7.2 QF_BV2_{bw}

Let us introduce the operation indexing \( t^{[n]}[i] \), which is defined as \( t[i:j] \), i.e., a special case of extraction. Although, in Section 11.7.4, we show that adding extraction makes the fragment NExpTime-hard, QF_BV2_{bw} can be extended with indexing without growth in complexity.

**Theorem 11.28.** QF_BV2_{bw} extended by indexing is in NP.

**Proof.** To show this, we extend the proof of Lemma 11.26 by an additional pre-processing step even before removing the non-zero constants. Suppose we are given a formula \( \Phi \in \text{QF_BV2}_{bw} \) 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. Assume $I = \{i_1, \ldots, i_n\}$ with $i_l < i_{l+1}$, $\forall l \in \{1, \ldots, m-1\}$. After extracting those bit-indices from $\Phi$, we explicitly encode the corresponding bits into Boolean variables, by translating $\Phi$ in a similar way as in Lemma 11.26. Consider three different kinds of terms in the following order:

1. Terms $t^{[n]}[i]$ are replaced by $t^{[n]}_1$.

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 $t^{[n]}_1 = t^{[n]}_2$. We define $I' := \{|I| \in \{1, \ldots, m\} | i_l < n\}$ as the number of explicitly specified indices smaller than $n$. Now, similar to Lemma 11.26, replace each equality $t^{[n]}_1 = t^{[n]}_2$ with

$$ (t^{[1]}_1[i] = t^{[1]}_2[i]) \land \cdots \land (t^{[1]}_{n-1}[i] = t^{[1]}_{n-1}[i]), $$

if $n = I'$. Otherwise, if $n > I'$, replace $t^{[n]}_1 = t^{[n]}_2$ with

$$ \left( \bigwedge_{l \in \{1, \ldots, I'\}} (t^{[1]}_1[i] = t^{[1]}_2[i]) \right) \land (t^{[1]}_{n-I'}[i] = t^{[1]}_{n-I'}[i]). $$

As in Lemma 11.26, we use $t^{[1]}_1 = t^{[1]}_2$ to express the $i$th row of the original equality. In the same way, $t^{[1]}_1$, being introduced for an indexing, represents the $i$th bit of $t$. The new terms $t^{[1]}_1, t^{[1]}_2, \ldots, t^{[1]}_n$ are constructed in the same way as in Lemma 11.26.

Similarly, if $n > I'$, the expression $t^{[1]}_{n-I'}[i] = t^{[1]}_{n-I'}[i]$ represents the remaining $n - I'$ 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 11.26, except for the construction of new constants: each constant $c^{[n]}$ is replaced with a new constant $c^{[n]}_{REM}$ by setting the $j$th bit of $c^R$ to the value of the $k$th bit of $c$, for $k := \min \{k' \mid \{1, \ldots, k\} \setminus I = j\}$.

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

Similarly, extending QF_BV2bw with additional relational operations from Table 11.1 does not increase complexity, either.

**Theorem 11.29.** QF_BV2bw extended by relational operations from Table 11.1 is in NP.

**Proof.** We give a reduction for the relational operation unsigned less than (bvult). The remaining relational operations in Table 11.1 can be reduced in a similar way. Given $\Phi \in \text{QF_BV2bw}$ (without indexing), additionally containing expressions $t^{[n]}_1 <_u t^{[n]}_2$, we adopt the proof of Lemma 11.26 in three ways.

First, the elimination of constants has to be modified. Again, let $c_{\text{max}} := b_{k-1} \ldots b_0$ be the largest constant in $\Phi$ denoted in binary representation with $b_{k-1} = 1$ and arbitrary bits $b_{k-2}, \ldots, b_0$. Without loss of generality, assume $n > k$. We now replace each relation $t^{[n]}_1 <_u t^{[n]}_2$ in $\Phi$ with

$$ (t^{[n]}_1[n-k] <_u t^{[n]}_2[n-k]) $$

$$ \lor (t^{[n]}_1[n-k] = t^{[n]}_2[n-k]) \land (\neg t^{[1]}_{k-1}[1] \land t^{[1]}_{k-1}[1]) $$

$$ \lor \cdots $$

$$ \lor (t^{[n]}_1[n-k] = t^{[n]}_2[n-k]) \land (t^{[1]}_{k-1}[1] \land t^{[1]}_{k-1}[1] \land \cdots \land \neg t^{[1]}_0[1] \land t^{[1]}_0[1]) $$


All expressions $t_1[i]$, $t_2[i]$, $th_1[n-i]$, and $th_2[n-i]$ are defined in the same way as it was done in Lemma 11.26.
Second, we need to use the number of all the relational operations $cnt_{rel}(\Phi)$, when reducing the bit-widths in $\Phi$.

The third modification is needed for constructing a satisfying assignment $\alpha'$ for the bit-width reduced formula $\Phi'$ out of the satisfying assignment $\alpha$ for $\Phi$. When selecting the bit-index which is used as a witness for the evaluation of a given expression $t_1[n] <_u t_2[n]$, we choose the index of the most significant bit which is assigned to a different value in the two terms. As in Lemma 11.26, 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 $t_1[n] <_u t_2[n]$ in our proof, it is easy to see that it holds for all relational operations from Table 11.1. All unsigned operations can be replaced by $t_1[n] <_u t_2[n]$ as in the definition of Table 11.1. 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 $t_1[i] <_u t_2[i]$ with

$$
(t_{hi_1}[n-i-1] < u \ t_{hi_2}[n-i-1]) \\
\vee
(t_{hi_1}[n-i-1] = t_{hi_2}[n-i-1]) \wedge (\neg t_{lo_1}[1] \wedge t_{lo_2}[1]) \\
\vee
(t_{hi_1}[n-i-1] = t_{hi_2}[n-i-1]) \wedge (t_{lo_1}[1] \leftrightarrow t_{lo_2}[1]) \wedge (t_{lo_1}[0] \leq u t_{lo_2}[0])
$$

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 $i_j$. Apart from this, the reduction works as already described. This leads to the following corollary:

**Corollary 11.30.** $QF_{BV2_{bw}}$ extended by indexing together with relational operations from Table 11.1 is in NP.

See Appendix 11.11.3 for an example.

11.7.3 $QF_{BV2_{\leq 1}}$

Figure 11.1 depicts our forthcoming results on extending $QF_{BV2_{\leq 1}}$ 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 $bshl_\tau$ represents left shift by $\tau$, and plays a central role as being a base operation in $QF_{BV2_{\leq 1}}$. The vertex $bmul_c$ 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 $bshl_\tau$, namely $bshr_\tau$, $badd$, $bsub$, and $bmul_c$, can be used as an alternative base operation instead of $bshl_\tau$.

First, we show that $QF_{BV2_{\leq 1}}$ can be extended with indexing. Although a similar result was proposed for $QF_{BV2_{\geq 0}}$, the reduction we used there is not appropriate for $QF_{BV2_{\leq 1}}$, because of the presence of shifts in the formulas.

**Theorem 11.31.** $QF_{BV2_{\leq 1}}$ extended by indexing is in PSpace.
Figure 11.1: Extending QF\_BV2_{\leq 1} with operations

Proof. The counter we introduced in our translation from QF\_BV2_{\leq 1} to sequential circuits (Lemma 11.25) 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_{\leq 1}.

Theorem 11.32. 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:

$$x \gg u 1 = t \& (\neg 0[n] \gg u 1)$$
$$x \& 1[n] = 0[n]$$

$$t[n] \gg u 1[n]$$

replace with: $x[n]$

add assertions:

$$x \ll 1 = t \& (\neg 0[n] \ll 1)$$
$$x \& v[n] = 0[n]$$
$$v \ll 1 = 0[n]$$
$$v \neq 0[n]$$

□ Further, it is well-known that any arithmetic right shift $t_1[n] \gg u t_2[n]$ can be reduced to logical right shift, as follows: $ite (t_1[n-1], \neg (t_1 \gg u t_2), t_1 \gg u t_2)$.

We now look at arithmetic operations:

Theorem 11.33. QF\_BV2_{\leq 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:

$$ts_1[n] = t_1$$
$$ts_2[n] = t_2$$
$$cin[n] = cout \ll 1$$

$$cout[n] = (ts_1 \& ts_2) | (ts_1 \& cin) | (ts_2 \& cin)$$

Multiplication by constant can be split into several multiplications by 2, i.e., left shift by 1, and addition, similar to [Sk12a]; [Sk12b]. Given such a multiplication $t[n] \cdot c[n]$, we introduce two sets of variables, $s_i$ and $x_i, 0 \leq i \leq Lc$. 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 $i$th step. By considering the individual bits of $c$, $s_i$ either is or is not added to the previous
subresult $x_{i-1}$. Finally, $x_{Lc}$ provides the required product.

\[
\begin{align*}
\text{replace with:} & \quad x_{Lc}[n] \\
\text{add assertions:} & \quad s_0[n] = t \\
& \quad s_i[n] = s_{i-1} \ll 1, \quad 0 < i \leq Lc \\
& \quad x_0[n] = \begin{cases} 
    s_0 & \text{if } [c][0] = 1 \\
    0 & \text{otherwise}
\end{cases} \\
& \quad x_i[n] = \begin{cases} 
    x_{i-1} + s_i & \text{if } [c][i] = 1 \\
    x_{i-1} & \text{otherwise}
\end{cases}, \quad 0 < i \leq Lc
\end{align*}
\]

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$_{\text{BV2} \ll 1}$.

Unary minus (bvneg) and subtraction (bvsup) can obviously be added to QF$_{\text{BV2} \ll 1}$ by using two’s complement representation. Furthermore, it is easy to see that addition and subtraction can be reduced to each other. Extending QF$_{\text{BV2} \ll 1}$ with additional relational operations, such as unsigned less than (bvult), does not increase complexity either. A term $t_1[n] <_u t_2[n]$ is the same as checking whether $t_1 - t_2 <_u 0$ holds, which can be replaced by constructing an adder for $t_1 + (\neg t_2) + 1$, 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. □

11.7.4 QF$_{\text{BV2} \ll c}$

Figure 11.2 depicts our forthcoming results on extending QF$_{\text{BV2} \ll c}$ 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$_{\text{BV2} \ll 1}$ (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$_{\text{BV2} \ll c}$ 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, bvslshr, and bvlshr.

\[
\begin{align*}
\text{bvshl}_1 & \quad \leftarrow \quad \text{bvshl} & \quad \text{bvlshr} \leftarrow \quad \text{bvashr} \\
\text{extract} & \quad \uparrow & \quad \uparrow & \quad \uparrow \\
\text{concat} & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
\text{bvmul} & \quad \leftarrow \quad \text{bvshl}_c & \quad \text{bvslshr}_c & \quad \text{bvashr}_c
\end{align*}
\]

Figure 11.2: Extending QF$_{\text{BV2} \ll c}$ with operations

**Theorem 11.34.** bvshl$_c$ and bvslshr$_c$ can be reduced to each other.
Theorem 11.35. 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: \( t_{1} = x_{1}^{[m+n-1 : n]} \)
\( t_{2} = x_{2}^{[n-1 : 0]} \)

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 \( t \ll c \).

The reduction in the other way around, i.e., extraction (or concatenation) to bvshl_c and bvshlr_c, takes a special role. Given a term \( t^{[n]} \ll [i : j] \), 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 shifts by constant, bitwise operations, and equality, as follows.

Given a formula \( \Phi \) with bit-vector variables \( x_{1}^{[n_{1}]} \ldots x_{l}^{[n_{l}]} \), let us calculate the maximal bit-width \( n_{\text{max}} := \max_{k} \{n_{k}\} \). First, replace each extraction \( t^{[n]} \ll [i : j] \) in \( \Phi \) with

\[ (t \ll (n_{\text{max}} - 1 - i)) \gg_{u} (n_{\text{max}} - 1 - i + j) \]

\footnote{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.}
Then, replace each bit-vector variable $x_k[n_i]$ with a new bit-vector variable $x_k'^{[n_{\text{max}}]}$. Finally, for each $x_k'$, add the following assertion to the formula:

$$x_k'^{[n_{\text{max}}]} = (x_k' \ll (n_{\text{max}} - n_k)) \gg u (n_{\text{max}} - n_k)$$

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 11.36.** multiplication and $\text{bvshl}_c$ can be reduced to each other.

**Proof.** First, we show how $\text{bvshl}_c$ can be expressed by $\text{bmul}$. 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 $L_c$ steps using exponentiation by squaring. We introduce two sets of variables, $p_i$ and $x_i$, $0 \leq i \leq L_c$. Each $p_i$ represents the number $2^{(2^i)}$, and each $x_i$ the subresult in the $i$th step. By considering the individual bits of $c$, the previous subresult $x_{i-1}$ either is or is not multiplied by $p_i$. Finally, $x_{L_c}$ provides the value $2^c$.

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

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 several 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 $P_n$ for the nearest power of 2, and define it as follows: $P_n := 2^{L_n}$.

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 11.23, where $m < n$:

- replace with: $x^{[2n]}$
- 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 $m \leq n$:

- replace with: $x_i^{[2^m]}$
- add assertions:
  - $x_m^{[2^m]} = t$
  - $x_i^{[2^m]} = x_{i-1} \circ x_{i-1}$

, $m < i \leq n$
Halfshuffle applies a logarithmic translation, which is based on the generalization of a bit-vector operation called half-shuffle [Waro2, Chpt. 7]. This generalized variant receives a bit-vector $t^{[2^m]}$ and produces the following bit-vector of bit-width $2^n$:

\[
\begin{array}{cccc}
0 & \ldots & 0 & t[2^m - 1] \\
2^{n-m} - 1 \\
0 & \ldots & 0 & t[2^m - 2] \\
2^{n-m} - 1 \\
\vdots & \ddots & \ddots & \vdots \\
0 & \ldots & 0 & t[0] \\
2^{n-m} - 1 \\
\end{array}
\]

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 $m \leq n$.

As it can be seen above, in the $i$th step we (i) shift our current bit-vector left by the constant $2^{m-i} - 2^{m-i}$, (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 11.11.4.

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:

\[
\begin{array}{cccc}
0 & \ldots & 0 & t[2^m - 1] \\
2^n - m \\
0 & \ldots & 0 & t[2^m - 2] \\
2^n - m \\
\vdots & \ddots & \ddots & \vdots \\
0 & \ldots & 0 & t[0] \\
2^n - m \\
\end{array}
\]

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 $m \leq n$.

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 $(Pn)^2$ 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 $Pn$ groups of $Pn$ bits, each group representing the result of one single-digit multiplication. To add all $Pn$ 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 \( \log_2(Pn) = Ln \) steps. For a detailed example, see also Appendix 11.11.5.

\[
\begin{align*}
\text{replace with:} & \quad x_{ln}^{[n-1]} : 0 \\
\text{add assertions:} & \quad x_0^{[(Pn)^2]} = \text{selfconcat} \left( \text{ext}_u (t_1, Pn-n), (Pn)^2 \right) \\
& \quad \& \text{expand} \left( \text{ext}_u (t_2, Pn-n), (Pn)^2 \right) \\
& \quad b_i^{[(Pn)^2]} = \text{binmagic} \left( 2^i \cdot Pn, (Pn)^2 \right) \\
& \quad x_i^{[(Pn)^2]} = \left( x_{i-1} \& b_{i-1} \right) + \left( x_{i-1} \& \sim b_{i-1} \right) \gg u \left( 2^{i-1} \cdot (Pn-1) \right)
\end{align*}
\]

\[\square\]

11.8 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 [KFB12], 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.

11.8.1 General Quantification

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

**Theorem 11.37.** UFBV2 is 2-NExpTime-complete [KFB12].

**Proof.** It is straightforward to see that UFBV2 \( \in \text{2-NExpTime} \), since every UFBV2 formula can be translated exponentially to a formula in UFBV1 \( \in \text{NExpTime} \) (Proposition 11.14), by applying a simple unary re-encoding to all the scalars in the formula. 2-NExpTime-hardness directly follows from Lemma 11.40.

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 [Chl84]. First, we give a definition of a domino system and then specify a 2-NExpTime-hard problem on this kind of systems.

**Definition 11.38 (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 \( k \geq 1 \);
- \( H, V \subseteq T \times T \) are the horizontal and vertical matching conditions, respectively;
- \( n \geq 1 \), encoded in unary format.

Note that the above definition differs (but not substantially) from the classical one in [Chl84]. Without loss of generality, we use sub-sequential natural numbers for identifying tiles. Similarly to [Mar07]; [NS11], the size factor \( n \), encoded in unary form, is part of the input. However, while a start tile \( a \) and a terminal tile \( \omega \) is usually used, in our case the starting tile is denoted by 0 and the terminal tile by \( k - 1 \), without loss of generality.

There are different Domino Tiling problems examined in the literature. In [Chl84], a classical tiling problem is introduced, namely the Square Tiling problem, which can be defined as follows:
Definition 11.39 (Square Tiling). Given a domino system \( \langle T, H, V, n \rangle \), an \( f(n) \)-square tiling is a mapping \( \lambda : \{0, f(n) - 1\} \times \{0, f(n) - 1\} \rightarrow T \) 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:
  \[
  (\lambda(i, j), \lambda(i, j + 1)) \in H \quad \forall_i < f(n), j < f(n) - 1
  \]
- all vertical matching conditions hold:
  \[
  (\lambda(i, j), \lambda(i + 1, j)) \in V \quad \forall_i < f(n) - 1, \; j < f(n)
  \]

In [Chl84], 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 \( \text{NTIME}(f(n)) \)-complete.

In particular, this implies that the \( 2^{(2^n)} \)-square tiling problem is \( 2 - \text{NExpTime} \)-complete.

Lemma 11.40. The \( 2^{(2^n)} \)-square tiling problem can be reduced to UFBV2.

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 \( L_k \).
- Represent the horizontal and vertical matching conditions with the uninterpreted functions (actually predicates) \( h^{[1]}((t_1, t_2), (t_1, t_2)) \) and \( v^{[1]}((t_1, t_2), (t_1, t_2)) \), respectively.
- Represent the tiling with an uninterpreted function \( \lambda^{[L_k]}((i, j)) \). \( \lambda \) returns the tile in the cell at the row index \( i \) and column index \( j \). Notice 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:

\[
\forall (i, j), \; k^{(2^n)}.
\lambda(0, 0) = 0 \quad \land \quad \lambda \left( 2^{(2^n)} - 1, 2^{(2^n)} - 1 \right) = k - 1
\land \quad \bigwedge_{(t_1, t_2) \in H} h(t_1, t_2) \quad \land \quad \bigwedge_{(t_1, t_2) \in V} v(t_1, t_2)
\land \quad \left( j \neq 2^{(2^n)} - 1 \implies h(\lambda(i, j), \lambda(i, j + 1)) \right)
\land \quad \left( i \neq 2^{(2^n)} - 1 \implies v(\lambda(i, j), \lambda(i + 1, j)) \right)
\]

This formula contains four kinds of constants. Three can be encoded directly \( 0^{(2^n)}, 0^{[L_k]}, \) and \( (k - 1)^{[L_k]} \). 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. \( \square \)

Similar to Section 11.6 and to our work in [FKB13b], 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_{\leq c} (or UFBV2_{\leq 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_{\leq c} and QF_BV2_{\leq 1} by quantifiers and uninterpreted functions.

Since we can use bitwise operations, equality and left shift by constant to express all common operations, UFBV2_{\leq c} remains 2 - \text{NExpTime}-complete. However, in contrast to quantifier-free
logics, we do not lose any expressiveness in UFBV2_{≤1}, either. We can see this already from
the fact that we only used bitwise operations, equality and addition in Lemma 11.40. Since, as we
pointed out in Section 11.7.3, addition can be reduced to bitwise operations, equality and left shift
by 1, the following result follows immediately:

**Corollary 11.41.** UFBV2_{≤1} is 2-NeXtTime-complete.

Nevertheless, we want to formalize this in a proposition and give a constructive proof by
showing how UFBV2_{≤c} can be reduced to UFBV2_{≤1}.

**Proposition 11.42.** UFBV2_{≤c} can be reduced to UFBV2_{≤1}.

**Proof.** Let Φ denote a bit-vector formula, x^{[n]}, y^{[n]} fresh bit-vector variables, and fn^{[n]} (·, ·) a fresh
uninterpreted function of arity 2, taking arguments of bit-width n. Replace each expression
i^{[n]} ≪ c^{[n]} in Φ with fn^{[n]} (i^{[n]}, c^{[n]}), extend the quantifier prefix of Φ with ∀x^{[n]}, y^{[n]}, and add
the following two constraints to the matrix of Φ:

\[
\begin{align*}
fn^{[n]} (x, 0) &= x \\
fn^{[n]} (x, y + 1) &= fn^{[n]} (x, y) ≪ 1
\end{align*}
\]

While the second constraint still contains addition to improve readability, this can be replaced by
using left shift by 1, as described in Section 11.7.3. □

**Remark 11.43.** This result is not very surprising if we consider the alternative characterizations of
QF_BV2_{≤1} and QF_BV2_{≤c}, as given in Section 11.7. 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 ∀x^{[n]}, y^{[n]}
to the quantifier prefix and adding the following constraints:

\[
\begin{align*}
fn^{[n]} (x, 0) &= 0 \\
fn^{[n]} (x, y + 1) &= fn^{[n]} (x, y) + x
\end{align*}
\]

With these two axioms, the multiplication t_1^{[n]} · t_2^{[n]} of two elements in Peano arithmetic is
uniquely defined by the instance fn^{[n]} (t_1^{[n]}, t_2^{[n]}) of the uninterpreted function fn.

While we were also able to give some complexity results for BV2 in [KFB12], it remains unclear
whether BV2 is complete for any complexity class.

**Proposition 11.44.** BV2 ∈ ExpSpace and BV2 is NeXtTime-hard [KFB12].

**Proof.** Given a BV2 formula, a simple unary re-encoding can be used to give an exponential
translation to BV1 ∈ PSpace (Proposition 11.13). Therefore, BV2 ∈ ExpSpace. Because of
QF_BV2 ⊆ BV2, NeXtTime-hardness follows trivially. □

### 11.8.2 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 Φ ∈ BV2, let max_{bw(3)} (Φ) and
max_{bw(v)} (Φ) denote the maximal bit-width of all the existentially and universally quantified variables,
respectively. (We define max_{bw(3)} (Φ) := 0 and max_{bw(v)} (Φ) := 0 if Φ does not contain any
existential or universal variables respectively.) Now we give a definition, similar to the one of
scalar-boundedness in Definition 11.15:
Definition 11.45 (Universally Bit-Width Bounded Formula Set). An infinite set $S$ of quantified bit-vector formulas is universally bit-width bounded, iff there exists a polynomial function $p : \mathbb{N} \to \mathbb{N}$ such that $\forall \Phi \in S. \max_{bw}(\Phi) \leq p \left( \max_{bw}(\Phi) \right)$.

Theorem 11.46. If $S \subseteq UFBV_2$ (or $S \subseteq BV_2$) is universally bit-width bounded, then $S \in \text{NE}$.

Proof. Let $S \subseteq UFBV_2$ be universally bit-width bounded and let $p_0$ be the polynomial function that exists according to Definition 11.45. For any $\Phi_0 \in S$, let $n := |\Phi_0|$. We can assume that $\Phi_0$ contains at most $k \leq n$ universal variables. Also, let $\max_{bw}(\Phi_0)$ and $\size_{bw}(\Phi_0)$ be defined in the same way as it was done in Section 11.5. This implies $\max_{bw}(\Phi_0) \leq 2^n$ and $\size_{bw}(\Phi_0) \leq n$.

In order to prove that $S \in \text{NE}$, we now give a translation into QF_BV1 $\in \text{NP}$ 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 $\Phi_1 \in \text{QF}_2$ with

$$\max_{bw}(\Phi_1) = \max_{bw}(\Phi_0) \leq 2^n$$

$$\size_{bw}(\Phi_1) \leq \size_{bw}(\Phi_0) \cdot \max_{bw}(\Phi_0)$$

for some polynomial function $p_1$. Since $\Phi_1$ does not contain any (universal) quantifiers, it can be polynomially translated to some $\Phi_0 \in \text{QF}_2$, by replacing all uninterpreted functions of $\Phi_1$ with bit-vector variables and adding at most quadratic many Ackermann constraints (as in Proposition 11.12). Therefore,

$$\max_{bw}(\Phi_2) = \max_{bw}(\Phi_1) \leq 2^n$$

$$\size_{bw}(\Phi_2) \leq p_2(\size_{bw}(\Phi_1)) \leq p_2 \left( \size_{bw}(\Phi_0) \cdot 2^{p_1(n)} \right)$$

for some polynomial function $p_2$. In a last step, a unary re-encoding is applied to $\Phi_2$ (similar to Proposition 11.18), resulting in $\Phi_3 \in \text{QF}_1$. The size of $\Phi_3$ is bounded by

$$|\Phi_3| \leq \max_{bw}(\Phi_2) \cdot \size_{bw}(\Phi_2) + c$$

$$\leq 2^n \cdot p_2 \left( \size_{bw}(\Phi_0) \cdot 2^{p_1(n)} \right) + c \leq 2^{p_3(n)} + c$$

for some polynomial function $p_3$. Therefore, $\Phi_3 \in \text{QF}_1$ is only single-exponential in the size of $\Phi_0$. Together with QF_BV1 $\in \text{NP}$ (Proposition 11.11), this shows that $S \in \text{NE}$.

We now define $\text{BV}_2 \subseteq \text{BV}_2$ and $\text{UFBV}_2 \subseteq \text{BV}_2$ as the set of all $\Phi \in \text{BV}_2$ and $\Phi \in \text{UFBV}_2$ with $\max_{bw}(\Phi) \leq \max_{bw}(\Phi) + 1$, 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 [GLMo8]. Quantification over array indices is also discussed in [BMS06], where the so-called bounded array property fragment is addressed.

Theorem 11.47. $\text{BV}_2$ and $\text{UFBV}_2$ are NEXPTIME-complete.

Proof. It is easy to see that $\text{BV}_2$ and $\text{UFBV}_2$ are NEXPTIME-hard since both logics are an extension of QF_BV2, which is already NEXPTIME-hard (Proposition 11.22). The other direction is a consequence of Theorem 11.46, since $\text{BV}_2$ and $\text{UFBV}_2$ 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.
11.8.3 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 [BB12; CS99]. The same obviously holds for QF_BV1.

However, as shown in Theorem 11.50, extending QF_UFBV2 (and even QF_BV2) with non-recursive macros does not give additional expressiveness, in terms of complexity. Let the subscript \( M \) denote the fact that, additionally, non-recursive macros can be used in our logic.

**Definition 11.48 (Logic with Non-Recursive Macros).** Given a bit-vector logic \( \mathcal{L} \), let \( \mathcal{L}_M \) denote the set of all bit-vector formulas in the following form:

\[
Q \forall u_0^{[n_0]}, \ldots, u_k^{[n_k]} . \ f_1^{[1]} \land \ f_0^{[w_0]}(u_0, \ldots, u_k) = d_0^{[w_0]} \\
\land \ldots \\
\land \ f_m^{[w_m]}(u_0, \ldots, u_{k_m}) = d_m^{[w_m]}
\]

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 \( i \leq j \).

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

We now introduce a normal form, similar to the flat form in Definition 11.49, in order to obtain an upper bound for the growth in formula size when applying macro expansion.

**Definition 11.49 (Functional Flat Form).** A bit-vector formula \( \Phi \) is in function flat form if every uninterpreted function in \( \Phi \) has only variables as arguments.

It is easy to see that any \( \Phi \) can be translated into function flat form with only linear growth in formula size. Given a term \( f(t_1^{[n_1]}, \ldots, t_k^{[n_k]}) \) in \( \Phi \), 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 11.50.** QF_UFBV2_M is \textsc{NExpTime}-complete.

**Proof.** \textsc{NExpTime}-hardness is obvious, since QF_UFBV2 \( \subseteq \) QF_UFBV2_M. Inclusion can be shown in a similar way as it is done in Theorem 11.46.

Let \( \Phi_0 := \forall u_0^{[n_0]}, \ldots, u_k^{[n_k]} . \ t \land t_M \) be a QF_UFBV2_M formula of size \( n := |\Phi_0| \), where \( t \in \text{QF_UFBV2} \) 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 \( |d_m \sigma| = |d_m| \leq n \). 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 $\Phi_1 \in \text{QF}_\text{UFBV2}$ that contains no more macros. It holds that

\[
\begin{align*}
\max_{\text{bw}} (\Phi_1) &= \max_{\text{bw}} (\Phi_0) \leq 2^n \\
\text{size}_{\text{bw}} (\Phi_1) &\leq l(n^{m+1}) \leq l(n^m) \leq l(2^{n-Ln})
\end{align*}
\]

for some linear function $l$. We now apply a unary re-encoding to $\Phi_1$, yielding $\Phi_2 \in \text{QF}_\text{UFBV1}$. The size of $\Phi_2$ is bounded by

\[|\Phi_2| \leq \max_{\text{bw}} (\Phi_1) \cdot \text{size}_{\text{bw}} (\Phi_1) + c \leq 2^n \cdot l(2^{n-Ln}) + c\]

which is only single exponential in the size of $\Phi_0$. This gives $\text{QF}_\text{UFBV2}_M \in \text{NExpTime}$. \qed

11.9 Practical Considerations

As mentioned in Section 11.2, 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 11.6 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.

11.9.1 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 $\text{QF}_{\text{BV2}} \ll c$ (or general $\text{QF}_{\text{BV2}}$) are DQBF or EPR, which are both NExpTime-complete [Lew80]; [PRA01]; [PR79]. For $\text{QF}_{\text{BV2}} \ll 1$, a translation to model checking for sequential circuits as given in Lemma 11.25 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 [FKB12], their performance does not nearly reach the one of current EPR solvers as, e.g., iProver [Kor08]. On the other hand, many efficient model checkers for sequential circuits in SMV or AIGER format exist.

In [KFB13a], we therefore gave a polynomial translation from $\text{QF}_{\text{BV2}}$ to EPR (this is in contrast to existing translations in [Emm+10]; [KKV09], 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 $\text{QF}_{\text{BV2}}$.

A different situation occurs when we look at $\text{QF}_{\text{BV2}} \ll 1$. 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 $\text{QF}_{\text{BV}2_{\leq 1}} \in \text{PSPACE}$. In [FKB13a], we therefore presented a polynomial translation from $\text{QF}_{\text{BV}2_{\leq 1}}$ 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 [Cim+02] and Ilmc5, using techniques described in [Bra11]; [Bra+11], 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 [FKB13b]; [KFB13a] and Section 11.6 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 increase their overall performance.

11.9.2 Benchmark Problems

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

In [FKB13a], we proposed two new sets of $\text{QF}_{\text{BV}2}$ 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 $\text{QF}_{\text{BV}2_{\leq 1}}$ in [FKB13a].

Another family of benchmarks was directly derived from the discussion on the expressiveness of bit-vector operations in this paper. As we know from Section 11.6, all common bit-vector operations can be logarithmically expressed (in bit-width) by bitwise operations and equality in combination with \emph{shift by constant}, \emph{multiplication}, \emph{concatenation}, or \emph{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 [FKB13a]; [KFB13a], 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\(^7\) [KFB13b].

11.10 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 \emph{unary} and \emph{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 ($\text{QF}_{\text{BV}2}$) turned out to be \text{NEXPTIME}-complete [KFB12].

---

\(^1\)http://ecee.colorado.edu/wpmu/iimc/
\(^2\)http://fmv.jku.at/
\(^3\)http://www.satcompetition.org/
In this paper, we extended our previous work from [FKB13b]; [KFB12]. 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 [FKB13b]; [KFB12] 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 BV2_{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 QF_{BV2M}, 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 [FKB13a]; [KFB13a], 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 [DF99]. 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.
11.11 Appendix

11.11.1 Example: A Reduction of a DQBF to QF\(_{BV2}\)\(_{\leq c}\)

Consider the following DQBF:

\[
\forall u_0, u_1, u_2 \exists x(u_0), y(u_1, u_2) : (x \lor y \lor \neg u_0 \lor \neg u_1) \land \\
(x \lor \neg y \lor u_0 \lor \neg u_1 \lor \neg u_2) \land \\
(x \lor \neg y \lor \neg u_0 \lor \neg u_1 \lor u_2) \land \\
(\neg x \lor y \lor \neg u_0 \lor \neg u_2) \land \\
(\neg x \lor \neg y \lor u_0 \lor u_1 \lor \neg u_2)
\]

This DQBF is unsatisfiable.

Using the reduction given in Lemma 11.23, this formula is translated to the following QF\(_{BV2}\)\(_{\leq c}\) formula:

\[
\left(\left( X \mid Y \right) \left( \neg U_0 \lor \neg U_1 \right) \land \left( X \mid Y \right) \left( U_0 \lor \neg U_1 \right) \land \left( X \mid \neg U_0 \lor U_1 \right) \land \left( X \mid \neg U_0 \lor \neg U_1 \right) \right) = \sim 0^{[8]} \land \\
\bigwedge_{m \in \{0, 1, 2\}} U_m \ll 2^m = \sim U_m \land \\
X \land \sim U_1 = (X \ll 2^1) \land \sim U_1 \land \\
X \land \sim U_2 = (X \ll 2^2) \land \sim U_2 \land \\
Y \land \sim U_0 = (Y \ll 2^0) \land \sim U_0
\]

(11.6)

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

Recall that the notation \( t[n] \equiv d \) is an alternative for \( \left\lfloor t[n] \right\rfloor = d \), assuming an appropriate model for \( t \). By construction, \( U_0 \equiv 01010101, U_1 \equiv 00110011 \), and \( U_2 \equiv 00001111 \).

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

\[
X \land \sim U_1 \equiv (x_7, x_6, 0, 0, x_3, x_2, 0, 0) \\
(X \ll 2^1) \land \sim U_1 \equiv (x_5, x_4, 0, 0, x_1, x_0, 0, 0)
\]

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

\[
X \equiv (x_5, x_4, x_5, x_4, x_1, x_0, x_1, x_0)
\]

Furthermore, considering also the equality

\[
X \land \sim U_2 \equiv (x_7, x_6, x_5, x_4, 0, 0, 0, 0) \\
(X \ll 2^2) \land \sim U_2 \equiv (x_3, x_2, x_1, x_0, 0, 0, 0, 0)
\]

results in

\[
X \equiv (x_1, x_0, x_1, x_0, x_1, x_0, x_1, x_0)
\]

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

\[
Y \equiv (y_6, y_6, y_4, y_4, y_2, y_2, y_0, y_0)
\]
In order to show that the formula (11.6) is unsatisfiable, let us evaluate the “clauses” in the formula:

\[
X | Y | \sim U_0 | \sim U_1 = (1, 1, 1, x_0 \lor y_4, 1, 1, 1, x_0 \lor y_0)
\]

\[
X | \sim Y | U_0 | \sim U_1 | \sim U_2 = (1, 1, 1, 1, 1, 1, x_1 \lor y_0, 1)
\]

\[
\sim X | \sim Y | U_0 | \sim U_1 | U_2 = (1, 1, 1, 1, x_0 \lor \sim y_4, 1, 1, 1, 1)
\]

\[
\sim X | \sim Y | U_0 | \sim U_1 | \sim U_2 = (1, 1, 1, 1, \sim x_0 \lor y_2, 1, 1, 1, 1)
\]

By applying bitwise and to them, we get the bit-vector constrained by the formula (11.6):

\[
t = \begin{pmatrix}
1 \\
1 \\
1 \\
(x_0 \lor \sim y_4) \land (x_0 \lor y_4) \\
\sim x_1 \lor \sim y_2 \\
\sim x_0 \lor y_2 \\
x_1 \lor y_0 \\
(x_0 \lor y_0) \land (\sim x_0 \lor y_0)
\end{pmatrix}
\]

\[
= \begin{pmatrix}
1 \\
1 \\
1 \\
x_0 \\
\sim x_1 \lor \sim y_2 \\
\sim x_0 \lor y_2 \\
x_1 \lor y_0 \\
y_0
\end{pmatrix}
\]

In order to check if \( t \sim 0 \) 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 \( \sim x_1 \lor \sim y_2 \).

11.11.2 Example: A Reduction of a QBF to QF,BV2≤1

Consider the following QBF:

\[
\exists x \forall u_2 \exists y \forall u_1 u_0 \exists z. (u_2 \lor u_1 \lor \sim z) \land
\]

\[
(u_2 \lor \sim x \lor y) \land
\]

\[
(u_0 \lor \sim x \lor \sim z) \land
\]

\[
(u_1 \lor \sim y \lor z) \land
\]

\[
(u_0 \lor \sim u_1 \lor z)
\]

This QBF is satisfiable.

Using the reduction given in Lemma 11.24, this formula is translated to the following QF,BV2≤1 formula:

\[
\left( (U_2 | U_1 | \sim Z) & (U_2 | \sim X | Y) & (U_0 | \sim X | \sim Z) \&
\right.

\[
(U_1 | \sim Y | Z) & (U_0 | \sim U_1 | Z) \right) = \sim 0 \]

\[
\bigwedge_{m \in \{0,1,2\}} \left( \bigwedge_{0 \leq i < m} U_i \right) \oplus U_m = U_m \ll 1 \land
\]

\[
X \land \sim 1 = X \ll 1 \land
\]

\[
(U_1 = \sim (U_2 \ll 1 \oplus U_2)) \land (Y \land U_1 = (Y \ll 1) \land U_1)
\]

In the following, let us show that this formula is also satisfiable. As in the previous example, we have \( U_0 \equiv 01010101, U_1 \equiv 00110011, \) and \( U_2 \equiv 00001111 \). 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

$$X \& \sim 1 \equiv (x_7, x_6, x_5, x_4, x_3, x_2, x_1, 0)$$
$$X \ll 1 \equiv (x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0)$$

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

$$X \equiv (x_0, x_0, x_0, x_0, x_0, x_0, x_0, x_0)$$

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

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

the following bit-vectors

$$Y \& U_2' \equiv (y_7, y_6, y_5, 0, y_3, y_2, y_1, 0)$$
$$(Y \ll 1) \& U_2' \equiv (y_6, y_5, y_4, 0, y_2, y_1, y_0, 0)$$

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

$$Y \equiv (y_4, y_4, y_4, y_4, y_0, y_0, y_0, y_0)$$

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.

$$Z \equiv (z_7, z_6, z_5, z_4, z_3, z_2, z_1, z_0)$$

In order to show that the formula (11.7) is satisfiable, let us evaluate the “clauses” in the formula:

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

By applying bitwise and to them, we get the bit-vector constrained by the formula (11.7):

$$t \equiv \begin{pmatrix}
\neg z_7 \land \neg (x_0 \lor y_4) \land (\neg x_0 \lor \neg z_7) \land (\neg y_4 \lor z_7) \\
\neg z_6 \land (\neg x_0 \lor y_4) \land (\neg y_4 \lor z_6) \\
(\neg x_0 \lor y_4) \land (\neg x_0 \lor \neg z_5) \land z_5 \\
\neg x_0 \lor y_4 \\
(\neg x_0 \lor \neg z_3) \land (\neg y_0 \lor z_4) \\
\neg y_0 \lor z_3 \\
(\neg x_0 \lor \neg z_1) \land z_1 \\
1
\end{pmatrix} = \begin{pmatrix}
\neg z_7 \land \neg y_4 \\
\neg z_6 \\
z_5 \\
\neg x_0 \\
\neg y_0 \lor z_4 \\
\neg y_0 \lor z_3 \\
z_1 \\
1
\end{pmatrix}$$

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

$$z_7 = z_6 = y_4 = y_0 = x_0 = 0$$
$$z_5 = z_1 = 1$$

Therefore,

$$
\begin{align*}
U_0 & \equiv 01010101, \ U_1 \equiv 00110011, \ U_2 \equiv 00001111, \\
X & \equiv 00000000, \ Y \equiv 00000000, \ Z \equiv 00111111
\end{align*}
$$

is a possible satisfying assignment for the bit-vector formula.
11.11-3 Example: Bit-Width Reduction of a QF_BV_{2bw} Formula with Indexing and Relational Operations

Let
\[ \Phi_0 := (x^{[100]} <_u y^{[100]}) \land (z^{[50]} = w^{[50]}) \land (w^{[100]}[38] = y^{[100]}[72]) \]

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 \( \Phi_1 \) by splitting all bit-vectors at the corresponding bit-indices. First, \( x^{[100]} <_u y^{[100]} \) is therefore replaced by
\[ (x'_{99:73}^{[27]} <_u y'_{99:73}^{[27]}) \lor (x'_{99:73}^{[27]} = y'_{99:73}^{[27]}) \land (\neg x'_{72}^{[1]} \land y'_{72}^{[1]}) \]
\[ \lor (x'_{99:73}^{[27]} = y'_{99:73}^{[27]}) \land (x'_{71:39}^{[33]} <_u y'_{71:39}^{[33]}) \land (\neg x'_{38}^{[1]} \land y'_{38}^{[1]}) \]
\[ \lor (x'_{99:73}^{[27]} = y'_{99:73}^{[27]}) \land (x'_{71:39}^{[33]} = y'_{71:39}^{[33]}) \land (\neg x'_{38}^{[1]} \land y'_{38}^{[1]}) \]
\[ \lor (x'_{99:73}^{[27]} = y'_{99:73}^{[27]}) \land (x'_{71:39}^{[33]} = y'_{71:39}^{[33]}) \land (x'_{37:0}^{[38]} <_u y'_{37:0}^{[38]}) \]

Next, \( z^{[50]} = w^{[50]} \) is replaced by
\[ (z'_{49:39}^{[11]} = w'_{49:39}^{[11]}) \land (z'_{38}^{[1]} \leftrightarrow w'_{38}^{[1]}) \land (z'_{37:0}^{[38]} = w'_{37:0}^{[38]}) \]

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 \( \Phi_1 \), we can generate a bit-width reduced formula \( \Phi_2 \) by replacing all bit-widths \( n \) in \( \Phi_1 \) with \( \min\{11, n\} \). We therefore replace the variables
\[ x'_{99:73}^{[27]}, y'_{99:73}^{[27]}, x'_{71:39}^{[33]}, y'_{71:39}^{[33]}, \]
\[ x'_{37:0}^{[38]}, y'_{37:0}^{[38]}, z'_{37:0}^{[38]}, w'_{37:0}^{[38]} \]

by
\[ x''_{99:73}^{[11]}, y''_{99:73}^{[11]}, x''_{71:39}^{[11]}, y''_{71:39}^{[11]}, \]
\[ x''_{37:0}^{[11]}, y''_{37:0}^{[11]}, z''_{37:0}^{[11]}, w''_{37:0}^{[11]} \]

respectively.

11.11.4 Example: Half-Shuffle and Expand Applied to a Bit-Vector

The halfshuffle \((1101, 16)\) can be replaced with \( x_2^{[16]} \), by adding the following assertions. First, zero extension is applied to the original vector:
\[ x_0^{[16]} = \text{ext}_u(t^{[4]}, 12) \equiv 0000 0000 0000 1101 \]
Now, in two iterations, the bits of \( t^{[4]} \) are separated and moved to the distinct partitions of the extended vector:

\[
\begin{align*}
x_1^{[16]} &= (x_0^{[16]} \mid (x_0^{[16]} \ll 6)) \& \text{binmagic} (2, 16) \\
&= (0000 0000 0000 1101 | 0000 0011 0100 0000) \& 0011 0011 0011 0011 \\n&= 0000 0011 0000 0001 \\
x_2^{[16]} &= (x_1^{[16]} \mid (x_1^{[16]} \ll 3)) \& \text{binmagic} (1, 16) \\
&= (0000 0011 0000 0001 | 0001 1000 0000 1000) \& 0101 0101 0101 0101 \\n&= 0001 0001 0000 0001
\end{align*}
\]

The result now can be used for example in \( \text{expand} \): \( \text{expand}(1101, 16) \) can be expressed as \( x_2^{[16]} \), by adding the following assertions:

\[
\begin{align*}
x_0^{[16]} &= \text{halfshuffle} (t^{[4]}, 16) \equiv 0001 0001 0000 0001 \\
x_1^{[16]} &= x_0^{[16]} \mid (x_0^{[16]} \ll 1) \\
&\equiv 0001 0001 0000 0001 | (0010 0010 0000 0010) = 0011 0011 0000 0011 \\
x_2^{[16]} &= x_1^{[16]} \mid (x_1^{[16]} \ll 2) \\
&\equiv 0011 0011 0000 0011 | (1100 1100 0000 1100) = 1111 1111 0000 1111
\end{align*}
\]

11.1.5 Example: Multiplication of Two Bit-Vectors

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

\[
\begin{align*}
x^{[16]} &= \text{selfconcat} (t_1, 16) \& \text{expand} (t_2, 16) \\
&\equiv 0011 0011 0011 0011 \& 0000 1111 0000 1111 = 0000 0011 0000 0011 \\
&= g_3^{[4]} g_2^{[4]} g_1^{[4]} g_0^{[4]}
\end{align*}
\]

\( g_3^{[4]}, g_2^{[4]}, g_1^{[4]}, \) 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:

\[
\begin{align*}
b_0^{[16]} &= \text{binmagic} (4, 16) \equiv 0000 1111 0000 1111 \\
x_1^{[16]} &= (x_0 \& b_0) + ((x_0 \& \sim b_0) \gg u 3) \\
&\equiv (0000 0011 0000 0011) + (0000 0000 0000 0000 \gg u 3) \\
&= 0000 0011 0000 0011
\end{align*}
\]

\( g_3^{[8]} \) and \( g_0^{[8]} \) 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., \( g_3^{[8]} = (g_3 \ll 1) + g_3^{[8]} \) and \( g_0^{[8]} = (g_1 \ll 1) + g_0^{[8]} \).
Since we still have several partial results, we have to continue adding neighbouring groups:

\[b_1^{[16]} = \text{binmagic}(8, 16) \equiv 0000\ 0000\ 1111\ 1111\]

\[x_2^{[16]} = (x_1 \& b_1) + ((x_1 \& \sim b_1) \gg u 6)\]

\[\equiv (0000\ 0000\ 0000\ 0011) + (0000\ 0011\ 0000\ 0000\ \gg u 6)\]

\[= 0000\ 0000\ 0000\ 1111\]

After the last step, there is only one bit group left and the least significant bits of the bit-vector \(x_2^{[16]} \equiv 0000\ 0000\ 0000\ 1111\) correspond to the solution of the multiplication, i.e.,
\[0011 \cdot 0101 = x_2^{[16]}[3 : 0] \equiv 1111.\]

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.
ON THE COMPLEXITY OF SYMBOLIC VERIFICATION ANDDecision
PROBLEMS IN BIT-VECTOR LOGIC

AUTHORS. Gergely Kovácsnai, Helmut Veith, Andreas Fröhlich, and Armin Biere.

ABSTRACT. We study the complexity of decision problems encoded in bit-vector logic. This class of problems includes word-level model checking, i.e., the reachability problem for transition systems encoded by bit-vector formulas. Our main result is a generic theorem which determines the complexity of a bit-vector encoded problem from the complexity of the problem in explicit encoding. In particular, NL-completeness of graph reachability directly implies PSPACE-completeness and EXPSPACE-completeness for word-level model checking with unary and binary arity encoding, respectively. In general, problems complete for a complexity class \( C \) are shown to be complete for an exponentially harder complexity class than \( C \) when represented by bit-vector formulas with unary encoded scalars, and further complete for a double exponentially harder complexity class than \( C \) with binary encoded scalars. We also show that multi-logarithmic succinct encodings of the scalars result in completeness for multi-exponentially harder complexity classes. Technically, our results are based on concepts from descriptive complexity theory and related techniques for OBDDs and Boolean encodings.

12.1 INTRODUCTION

Symbolic encodings of decision problems by Boolean formalisms are well-known to increase the problem complexity [BLT92; BL96; DST14; Fei+99; GW83; GLV99; LB90; PY86; Veig6; Veig97; Veig98b; Wag86]. In particular, the literature has studied graph problems and other relational problems whose adjacency relation is given by a Boolean formula, circuit or BDD. As Table 12.1 shows, the complexity of these problems typically rises by an exponential, e.g., from NL to PSPACE, from NP to NEXPTIME, etc. In this paper, we show that symbolic encodings by quantifier-free bit-vector logic (QF_{BV}) will in general also lead to a complexity increase which ranges from exponential to multi-exponential. Interestingly, the increase depends on a single factor, namely how the bit-width of bit-vectors is encoded. For unary encoding, bit-vector logic shows the same complexity behavior as Boolean logic, and for binary encoding, the complexity increase is double exponential. We can generalize the latter encoding, and call it “\( \nu \)-logarithmic”: encode the bit-width \( 2^\nu \) as \( \nu \) in binary form, where the degree of exponentiation is \( \nu - 2 \). We achieve a \( \nu \)-exponential increase in this case. Importantly, hardness already holds for bit-vector logics with the simple operators \( \land, \lor, \sim, = \), and the increment operator \( +_1 \). Membership holds for all bit-vector operators which allow log-space computable bit-blasting. Note that \( \land, \lor, \sim, =, +_1 \) defines a very weak logic: \( \land, \lor, \sim, = \) are contained in all reasonable logics, and the increment operator \( +_1 \) can be defined from other operators easily [FKB13b]. Therefore, our results determine the complexity of decision problems for a large class of bit-vector logics.

<table>
<thead>
<tr>
<th>Encoding →</th>
<th>explicit</th>
<th>Boolean circ./formula, BDD</th>
<th>unary QF_{BV}</th>
<th>binary QF_{BV}</th>
<th>( \nu )-logarithmic QF_{BV}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word-Level MC, Reachability</td>
<td>NL</td>
<td>PSPACE</td>
<td>PSPACE</td>
<td>ExpSpace</td>
<td>( (\nu - 1) )·ExpSpace</td>
</tr>
<tr>
<td>Circuit Value, Alternating Reachability</td>
<td>ExpTime</td>
<td>ExpTime</td>
<td>2·ExpTime</td>
<td>( \nu )·ExpTime</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.1: Examples of complexity increase by symbolic encoding. New results are indicated in boldface. All membership results hold for logics whose operators allow log-space computable bit-blasting. Hardness requires the operators $\land, \lor, \neg, =, +_1$. The column with $\nu$ holds for all $\nu > 1$.

<table>
<thead>
<tr>
<th>Logic</th>
<th>Complexity Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clique, 3-SAT, SAT, Knapsack</td>
<td>$\text{NP}$</td>
</tr>
<tr>
<td>$k$-QBF</td>
<td>$\Sigma^P_k$</td>
</tr>
</tbody>
</table>

**BIT-VECTOR LOGIC.** The theory of fixed-width bit-vector logics (i.e., logics where each bit-vector has a given fixed bit-width) is investigated in several scientific works [BDL98]; [BP98]; [BS09]; [CMR97]; [Fra10], and even concrete formats for specifying such bit-vector problems exist, e.g., the SMT-LIB format [BST10] or the BTOR format [BBL08]. In this paper, we restrict ourselves to quantifier-free bit-vector (QF_BV [BST10]) logics.

As discussed below, bit-vector logics have attracted significant interest in computer-aided verification and SMT solvers. From a theory perspective, bit-vector logics are very succinct logics to express Boolean functions. In contrast to Boolean logic, BDDs, and QBF, they are based on variables for bit-vectors rather than variables for individual bits. Thus, for instance $x[32] = y[32]$ expresses that two bit-vectors $x$ and $y$ of bit-width 32 are equal. Bit-vector operators are therefore defined for arbitrary bit-width $n$, for instance bitwise and/or/xor, shift operators, etc. This has important consequences: (1) a bit-vector logic is given by a list of operators, (2) there is an infinite number of bit-vector logics, and (3) there is no finite functionally complete set of operators from which all other operators can be defined. Moreover, it is evident that the encoding of scalars such as the number 32 in the above simple example is related to the complexity of bit-vector logic.

In previous work by some of the authors [KFB12]; [FKB13b], we investigated the complexity of satisfiability checking of bit-vector formulas. For instance, we showed in [KFB12] that satisfiability checking of QF_BV is NP-complete resp. NExpTime-complete if unary resp. binary encoding of scalars is used and any standard operator of the SMT-LIB [BST10] is allowed. (All these operators allow log-space computable bit-blasting.) In the binary case, we further analyzed what happened if we restricted the operator set; e.g., if only bitwise operators, equality, and left shift by one are allowed, then the complexity turns out to be PSpace-complete [FKB13b]. In fact, it is easy to see that also the logic of the operators $\land, \lor, \neg, =, +_1$ has a satisfiability problem in PSpace.

**WORD-LEVEL MODEL CHECKING AND DECISION PROBLEMS.** In hardware and software verification, bit-vector logics are a natural framework for word-level system descriptions; e.g., 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. The main practical motivation for our work is word-level model checking, a bit-vector encoded problem that is of importance in practice. With word-level model checking, we refer to the problem of reachability in a transition system where a state is given by a valuation of one or more bit-vectors, and the transition relation over the states is expressed as a bit-vector formula. Such a representation provides a natural encoding for design information captured at a higher level than that of individual wires and primitive gates. In the past, there has been lots of research on bit-level model checking [CGP99] as well as bin vector formula decision procedures [BLSo2]; [MSVo7]. Comparatively few work has yet been published on word-level model checking. However, with increasing performance of state-of-the-art model checkers [Bra12] and SMT solvers [BB09]; [DMBo8], also the interest in word-level model checking is growing [Bje08]; [BBL08]; [BMR12].
While there are some practical approaches to attack word-level model checking [Bjo08; BBL08; BMR12], we are not aware of any work that is dealing with the complexity of the underlying decision problem. Row 1 of Tab. 12.1 shows that we determine the complexity of word-level model checking for a large class of operators and scalar encodings.

Beyond word-level model checking, we also address the complexity of other decision problems. Rows 2-4 of Tab. 12.1 give examples of the complexity results for well-known decision problems in bit-vector encoding.

**Technical Contribution.** Instead of individual complexity results, the paper presents a generic technique to lift known complexity results for explicit encodings to the case of bit-vector encodings. Similar techniques were previously developed for symbolic encodings by circuits [LB90; PY86; Veig6], Boolean formulas [Veig7], and OBDDs [Veig8a]. Lifting membership for a complexity class is the easier part, for which we give a general result in Thm. 12.5. Lifting hardness requires more effort. Similarly as in [Veig7; Veig8a], our method assumes that the problems in explicit encoding are hard under quantifier-free reductions, a notion of reduction introduced in descriptive complexity theory [Imm87]. Note that the problems in Tab. 12.1 fulfill this requirement. The key theorem is Thm. 12.9, from which a general hardness result is implied in Corr. 12.10.

**Discussion.** The results of this paper show that the complexity of bit-vector encoded problems depends crucially on the formalism to represent the bit-width of the bit-vectors. At first sight, these results may seem unexpected, e.g., a small part of the formalism clearly dominates the complexity. From an algorithmic perspective, however, this is not surprising: executing a for-loop from 0 to $\text{INT\_MAX}$ on architectures with bit-width 16, $2^{16}$ or $2^{2^{16}}$ will result in drastically different runtimes!

It may also be surprising that QF\_BV fragments with PSPACE satisfiability and fragments with NExtTime satisfiability have the same complexity, e.g., for word-level model checking. This is however a common phenomenon: Boolean logic has an NP satisfiability problem, while satisfiability of BDDs is constant time. Nevertheless, the model checking problem for both of them is PSPACE-complete [Veig7; Veig8a].

Using unary and binary encodings for scalars draws a connection to previous work [KFB12]. Intuitively, results for the unary case measure complexity in terms of bit-widths, and those for the binary case measure complexity in the classical sense, i.e., in terms of formula size. The $\nu$-logarithmic encoding also manifests itself in practice, such as the one in the SMB-LIB to declare arrays by writing $(\text{Array idx elem})$, where $\text{idx}$ is the sort for array indexes, and $\text{elem}$ is the sort for array elements. If $\text{idx}$ is a bit-vector sort ($\text{BitVec n}$), where $n$ is encoded w.l.o.g. in binary form, the size of the array is double exponential in the length of the binary encoding of $n$.

We finally note that hardness for the unary case can also be concluded from an analysis of the proofs in [Veig7] using the definitions of symbolic encodings in [Veig8a]. The current paper gives a direct proof for the unary case which is independent of the predecessor papers.

### 12.2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers $\{0, 1, 2, \ldots\}$, while $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. $\mathbb{B} = \{0, 1\}$ is the Boolean domain. Given $i \in \mathbb{N}$, let us define the repeated exponentiation function $\exp_i : \mathbb{N} \to \mathbb{N}$ as follows: $\exp_0(n) = n$ and $\exp_{i+1}(n) = 2^{\exp_i(n)}$. Given a logical formula $\phi$ (in either bit-vector, first-order, or Boolean logic), if $x_1, \ldots, x_k$ are all the free variables that occur in $\phi$, we indicate this by writing $\phi(x_1, \ldots, x_k)$. 
**Complexity Classes.** We assume that the reader is familiar with standard complexity classes such as NL, P, ExpTime, etc., as listed in Tab 12.1. For simplicity, we will refer to these complexity classes as “standard complexity classes”. For a standard complexity class, it is natural to define the exponentially harder complexity class: $\text{Exp}_1(L) = \text{Exp}_1(NL) = \text{PSPACE}$, $\text{Exp}_2(NL) = \text{Exp}_1(\text{PSPACE}) = \text{ExpSPACE}$, etc. Similarly, $\text{Exp}_1(P) = \text{ExpTime}$, $\text{Exp}_2(P) = \text{Exp}_1(2\cdot \text{ExpTime})$, etc., and analogously for other standard complexity classes.

For a formal definition of this concept (which is beyond the scope and goal of this paper) one can use the concept of leaf languages [Vei96]; [BL96].

**Computational Problems in Descriptive Complexity Theory.** A *relational signature* is a tuple $\tau = (p_1^{a_1}, \ldots, p_k^{a_k})$ of relation symbols of arity $a_1, \ldots, a_k$, respectively. A finite *structure* over $\tau$ is a tuple $A = (U, \bar{p}_1^{a_1}, \ldots, \bar{p}_k^{a_k})$ where $U$ is a nonempty finite set (called the universe of $A$) and each $\bar{p}_i^{a_i} \subseteq U^{a_i}$ is a relation over $U$. The class of all finite structures over $\tau$ is denoted by $\text{Struct}(\tau)$. A computational problem over $\tau$ is a class $A \subseteq \text{Struct}(\tau)$, such that $A$ is closed under isomorphism. In this paper, we assume *convex* problems, as introduced in [Vei98a], and similarly in [Sch98]. A problem is convex if adding isolated elements to the universe of a structure does not change membership in the problem. In Sec 12.4 we will show that the model checking problem is naturally presented in this framework. For background on descriptive complexity see [Imm99].

### 12.3 Bit-Vector Logic

A *bit-vector*, or word, is a sequence of bits (i.e. 0 or 1). In this paper, we consider bit-vectors of a fixed size $n \in \mathbb{N}^+$, where $n$ is called the *bit-width* of the bit-vector. We assume the usual syntax and semantics for *quantifier-free bit-vector logic* (QF_BV), cf. the SMT-LIB format [BST10] and the literature [BDL98]; [BP98]; [BS09]; [CMR97]; [Fra10]. Basically, a bit-vector formula contains bit-vector variables and bit-vector constants, each of which is of a certain bit-width specified next to the variable resp. constant, and uses certain bit-vector operators whose semantics is a priori defined. For example, $x^{[16]} \neq y^{[16]} \land (u^{[32]} + v^{[32]} = (x^{[16]} \cdot y^{[16]}) \ll 1^{[32]})$ is a bit-vector formula with variables $x$ and $y$ of bit-width 16, $u$ and $v$ of bit-width 32, and operators for addition, shifting, concatenation, and comparison.

Note that, in bit-vector formulas, there exist such components which themselves do not represent bit-vectors, but rather carry additional *numerical* information to the bit-vectors. We call them *scalars*. Bit-width is a scalar, and there might be also other types of scalars in a formula\(^1\). This paper demonstrates the effect of encoding the scalars in different ways. For instance, scalars $x$ priori defined. For example, $\text{bit-vector}$

\[^1\text{For example, the common operators extraction and zero/sign extensions use scalar arguments as well, cf. [BST10]; [BDL98]; [BP98]; [BS09]; [CMR97]; [Fra10].}\]
We now demonstrate that our generic main results can be applied to the important example we want to verify. Thus, a structure $A$ is essentially a Kripke structure.Reachability analysis in $A$ means to check if there exists a reachable $\hat{P}$-state in the defined transition system, i.e., if $\exists s_0,s_1,\ldots,s_k \in U$ such that (1) $s_0 \in \hat{I}$, (2) $\forall i \in [1,k]. (s_{i-1},s_i) \in \hat{T}$, and (3) $s_k \in \hat{P}$. We call $MC = \{ A \in \text{Struct} (\tau) \mid \exists$ a reachable $\hat{P}$-state in $A \}$ the (explicit) model checking problem. Since
MC is a simple variant of graph reachability, we know from [Imm87] that MC is NL-complete under quantifier-free reductions.

The word-level encoding of MC means to encode the states by tuples of bit-vectors, and to define the relations $I, T, \bar{P}$ by bit-vector formulas. The corresponding decision problem is called $bv_\nu^\Omega(MC)$, where $\nu$ specifies the scalar encoding and $\Omega$ is a set of bit-vector operators that are allowed in the formulas. We will formally define this problem in Sec. 12.5.

Our results require the following assumptions on $\Omega$: (1) $\Omega$ contains only such operators for which bit-blasting is log-space computable in bit-width and (2) $\Omega$ contains all the simple operators $\land, \lor, \neg, =, +$. In particular, $\Omega$ may contain all common bit-vector operators [BST10] that are used in practice.

Then we obtain the following results as a direct consequence of Thm. 12.5, Cor. 12.10, and the NL-completeness of MC:

**Corollary 12.2.** Let $\Omega \subseteq \Pi$. The decision problem $bv_\nu^\Omega(MC)$ is

1. $\text{PSPACE}$-complete, if $\nu = 1$ and $\Omega \supseteq \{\land, \lor, \neg\}$,
2. $(\nu - 1)$-$\text{ExpSPACE}$-complete, if $\nu > 1$ and $\Omega \supseteq \{\land, \lor, \neg, =, +\}$,
under log-space reductions.

In practice, the term word-level model checking usually refers to the problem $bv_\nu^\Omega(MC)$, i.e., all scalars in the formulas are encoded as w.l.o.g. binary numbers. Thus, our results show that word-level model checking is $\text{ExpSPACE}$-complete.

### 12.5 Bit-Vector Representation of Problems

Our intention is to represent instances of computational problems as bit-vector formulas. More precisely, given a relational signature $\tau$ $= (P_1^{a_1}, \ldots, P_k^{a_k})$, we define what the bit-vector definition of a corresponding relation $P_i^{a_i}$ looks like and what structure these definitions generate.

In order to simplify the presentation, we introduce the concept of term vectors. A term vector is a sequence $t_1^{[s_1]}, \ldots, t_l^{[s_l]}$ of bit-vector terms. We write term vectors in boldface, i.e., $t = t_1^{[s_1]} \ldots, t_l^{[s_l]}$, and say that $t$ has the bit-width signature $s_1, \ldots, s_l$. We distinguish the special case when terms are variables, by denoting variable vectors as $x, y, z$.

Word-level model checking can again serve as motivation here, since it represents the states of a transition system by the same set of bit-vector variables $x_1^{[s_1]}, \ldots, x_l^{[s_l]}$. I.e., a state is in fact can be represented as the valuation of terms $t_1^{[s_1]}, \ldots, t_l^{[s_l]}$ assigned to those variables. Therefore, it is important that each state must have the same bit-width signature $s_1, \ldots, s_l$.

**Definition 12.3.** Let $x_1, \ldots, x_n$ be variable vectors each of which has the bit-width signature $s_1, \ldots, s_l$. Let $\nu$ be a scalar encoding, and let $n_i = \text{decode}_\nu(s_i)$ denote the actual bit-widths. A bit-vector formula $\psi(x_1, \ldots, x_n)$ defines the $a$-ary relation

$$
\text{gen}_\nu^a(\psi) = \{(d_1, \ldots, d_n) \in (\mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k})^a \mid \psi(d_1, \ldots, d_n) = \text{true}\}.
$$

Let $\tau = (P_1^{a_1}, \ldots, P_k^{a_k})$ be a relational signature. The tuple of definitions

$$
\Psi = \left( P_1(x_1^1, \ldots, x_{s_1}^1) := \psi_1(x_1^1, \ldots, x_{s_1}^1), \ldots, \right.
$$

$$
\left. P_k(x_1^k, \ldots, x_{s_k}^k) := \psi_k(x_1^k, \ldots, x_{s_k}^k) \right)
$$

where each $\psi_i$ is a bit-vector formula and each $x_i^j$ is a variable vector that has the bit-width signature $s_i, \ldots, s_{i_j}$, defines the $\tau$-structure

$$
\text{gen}_\nu^T(\Psi) = (\mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k}, \text{gen}_\nu^{a_1}(\psi_1), \ldots, \text{gen}_\nu^{a_k}(\psi_k)).
$$
The bit-vector representation of a computational problem consists of all the bit-vector representations of all the structures in the problem. Besides the definitions $\Psi$ of relations, it is also necessary to include the scalar encoding $v$ to use, as follows.

**Definition 12.4.** Let $A \subseteq \text{Struct}(\tau)$ be a problem, $v$ a scalar encoding, and $\Omega$ a set of bit-vector operators. Then we define

$$\text{bv}^\Omega_v(A) = \{ (\Psi, v) \mid \text{gen}_v^\psi(\Psi) \in A, \text{ and } \Psi \text{ contains only } \text{bv}^\Omega_v \text{ formulas} \}.$$  

In order to show how membership for a standard complexity class $C$ can be automatically lifted when bit-vector representation is used, we give a necessary, although not very strong, criterion on the operator set. This criterion is based on bit-blasting, and requires to use operators from $\Pi$, i.e., those which allow log-space computable bit-blasting in bit-width.

**Theorem 12.5.** Given a problem $A$, a standard complexity class $C$, and an operator set $\Omega \subseteq \Pi$, if $A \in C$, then $\text{bv}^\Omega_v(A) \in \text{Exp}_v(C)$.

### 12.6 LIFTING HARDNESS

The main contribution of this paper is to show how hardness for a standard complexity class $C$ can also be automatically lifted. Our most important theorem, Thm. 12.9 gives a rather general hardness result, from which we derive Cor. 12.10 to show hardness of $\text{bv}^\Omega_v$ for $\text{Exp}_v(C)$, where $\Omega \supseteq \{ \land, \lor, \neg, =, + \}$.

Our proofs employ the framework of descriptive complexity theory [Imm87]. In particular, we use the standard assumption that all structures are equipped with a binary successor relation. Thus, the universe of a structure can be naturally seen as an initial segment of the natural numbers. Our complexity results for bit-vector encoded problems assume that the problems in explicit encoding are hard under quantifier-free reductions, i.e., quantifier-free interpretations with equality and the successor relation. Examples of such problems including those in Tab. 12.1 can be found in [Ste91]; [Ste92a]; [Ste92b]; [Ste94]; [Ste95]. For natural problems, it is usually not difficult to rephrase an existing reduction as a quantifier-free reduction. Let $A \leq_{\text{qf}} B$ resp. $A \leq_{\text{l}} B$ denote that the problem $A$ has a quantifier-free resp. log-space reduction to the problem $B$.

Note that quantifier-free reductions are weaker than log-space reductions, i.e., $A \leq_{\text{qf}} B$ implies $A \leq_{\text{l}} B$. For exact background material and definitions, see [Imm87].

The key steps for Thm. 12.9 are two lemmas. Lemma 12.6 ("Conversion Lemma") shows that a quantifier-free reduction between $A$ and $B$ can be lifted to a log-space reduction between $\text{bv}_v(A)$ and $\text{bv}_v(B)$. Lemma 12.8 shows that $A$ is log-space reducible to $\text{bv}_v(\text{long}_v(A))$ where $\text{long}_v(\cdot)$ is an operator which decreases the complexity $v$-exponentially. From these two lemmas, Thm. 12.9 follows easily. The methodology of this paper is closest to [Vei84a], which contains a more thorough discussion of related work, descriptive complexity, and complexity theoretic background.

**Lemma 12.6 (Conversion Lemma).** Let $\Omega \supseteq \{ \land, \lor, \neg, =, + \}$. Given two problems $A \subseteq \text{Struct}(\sigma)$ and $B \subseteq \text{Struct}(\tau)$, if $A \leq_{\text{qf}} B$, then $\text{bv}_v^\Omega(A) \leq_{\text{l}} \text{bv}_u^\Omega(B)$, for any $v$.

The role of the following definition is to obtain from a problem $A$ another problem $\text{long}_v(A)$ of $v$-exponentially lower complexity. In order to construct this latter problem, we are going to “blow up” the size of a structure in a potentially $v$-exponential way. To this end, we view a structure $A$ as a bit string, and interpret the bit string as a binary number $\text{char}(A)$. The bit string is obtained from the characteristic sequences of the relations in $A$, i.e., for each tuple in lexicographic order, a single bit indicates whether the tuple is in the relation. Due to the presence of the successor relation, this notion is well defined.
Definition 12.7. Given a structure $A = (U, P_1, \ldots, P_k)$, let $\text{char}(P_i)$ denote the characteristic sequence of the tuples in $P_i$ in lexicographical order. Let $\text{char}(A)$ denote the binary number obtained by concatenating a leading 1 with the concatenation of $\text{char}(P_1), \ldots, \text{char}(P_k)$.

We define $\text{long}_v(A) = \{(V, R) \mid |V| = \exp_{v-1}(\text{char}(A))$ and $|R| = |V|\}$. For a problem $A$, let $\text{long}_v(A) = \bigcup_{A \in C} \text{long}_v(A)$. For a complexity class $C$, let $\text{long}_v(C) = \bigcup_{A \in C} \text{long}_v(A)$.

The next lemma shows that the problem $\text{long}_v(A)$ as bit-vector formulas applying $v$-encoding to scalars gives a $v$-exponentially more succinct representation, to which, consequently, the original problem $A$ can be reduced.

Lemma 12.8. Given a problem $A$, $A \leq_L \text{bv}^\Omega_v(\text{long}_v(A))$ if one of the following conditions holds:

1. $v = 1$ and $\Omega \supseteq \{<_u\}$
2. $v > 1$ and $\Omega \supseteq \{=\}$

Theorem 12.9 (Upgrading Theorem). Let $C_1$ and $C_2$ be complexity classes such that $\text{long}_v(C_1) \subseteq C_2$. If a problem $A$ is $C_2$-hard under quantifier-free reductions, then $\text{bv}^\Omega_v(A)$ is $C_1$-hard under log-space reductions if one of the following conditions holds:

1. $v = 1$ and $\Omega \supseteq \{\land, \lor, \neg, =, +_1, <_u\}$
2. $v > 1$ and $\Omega \supseteq \{\land, \lor, \neg, =, +_1\}$

Proof. For any $B \in C_1$, by assumption $\text{long}_v(B) \in C_2$, and hence $\text{long}_v(B) \leq_{qt} A$. By Lemma 12.6, it follows that $\text{bv}^\Omega_v(\text{long}_v(B)) \leq_{L} \text{bv}^\Omega_v(A)$, regardless of the additional operator $<_u$ in the unary case. Furthermore, by Lemma 12.8, it holds that $B \leq_L \text{bv}^\Omega_v(\text{long}_v(B))$. To put them together, $B \leq_L \text{bv}^\Omega_v(\text{long}_v(B)) \leq_L \text{bv}^\Omega_v(A)$ and, therefore, $\text{bv}^\Omega_v(A)$ is $C_1$-hard.

As we discussed before, the case of $v = 1$ shows the same complexity behavior as Boolean logic. Of course, this is no wonder, since all the operators in $\Omega = \{\land, \lor, \neg, =, +_1, <_u\}$, or more precisely, $\text{BV}_1^\Omega$ allows log-space computable bit-blasting in bit-width, and also in formula size, since bit-widths are now encoded in unary form. Thus, $\text{BV}_1^\Omega$ is log-space reducible to $\text{BV}_1^{\{\land, \lor, \neg\}}$, since $\{\land, \lor, \neg\}$ is a functionally complete set of Boolean operators. As a consequence, one can strengthen the first statement of Thm. 12.9 further as follows: $\text{bv}^\Omega_v(A)$ is $C_1$-hard for any $\Omega' \supseteq \{\land, \lor, \neg\}$. Note that this is consistent with corresponding results in [Vei97]; [Vei98a]. As a direct consequence, we can give the following corollary.

Corollary 12.10. Given a standard complexity class $C$ and a problem $A$, if $A$ is $C$-hard under quantifier-free reductions, then $\text{bv}^\Omega_v(A)$ is $\text{Exp}_v(C)$-hard under log-space reductions if one of the following conditions holds:

1. $v = 1$ and $\Omega \supseteq \{\land, \lor, \neg\}$
2. $v > 1$ and $\Omega \supseteq \{\land, \lor, \neg, =, +_1\}$

12.7 Conclusion

This paper gives a generic method for asserting the complexity of bit-vector logic encoded problems. As corollary we obtain a new complexity result for word-level model checking, an important practical problem. Since all complexity classes with complete problems have problems complete under quantifier-free reductions [Vei98b], we obtain a comprehensive picture of the worst case complexity of problems in bit-vector encoding. Note that our results do not apply to satisfiability of bit-vector logic, because “existence of a solution” is not hard for a complexity class, and thus the assumption of the Conversion Lemma is not satisfied. Nevertheless, we expect that the complexity of satisfiability for multi-logarithmic encodings shows a similar behavior as the problems studied here. We leave an analysis of this question to future work.
12.8 Appendix

12.8.1 Common Bit-Vector Operators

We give the syntax and semantics of common bit-vector operators, i.e., of those in SMTLIB [BST10].

**Syntax.** Tab. 12.2 shows details on the syntax of common operators, including possible condition on applicability and the bit-width of the resulting bit-vector. For the sake of simplicity, here we do not address the encoding of scalars.

<table>
<thead>
<tr>
<th>operation</th>
<th>condition</th>
<th>bit-width</th>
<th>alternative syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise negation:</td>
<td>bnot</td>
<td>n</td>
<td>~t[n]</td>
</tr>
<tr>
<td>bitwise and:</td>
<td>bind</td>
<td>n</td>
<td>t_1[n] &amp; t_2[n]</td>
</tr>
<tr>
<td>bitwise or:</td>
<td>bior</td>
<td>n</td>
<td>t_1[n]</td>
</tr>
<tr>
<td>bitwise xor:</td>
<td>bixor</td>
<td>n</td>
<td>t_1[n] ⊕ t_2[n]</td>
</tr>
<tr>
<td>bitwise nand:</td>
<td>binnand</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>bitwise nor:</td>
<td>binor</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>bitwise xnor:</td>
<td>bixnor</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>if-then-else:</td>
<td>ite</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>equality:</td>
<td>bcomp</td>
<td>1</td>
<td>t_1[n] = t_2[n]</td>
</tr>
<tr>
<td>unsigned less than:</td>
<td>bule</td>
<td>1</td>
<td>t_1[n] &lt;u t_2[n]</td>
</tr>
<tr>
<td>u. less than or equal:</td>
<td>bule</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>u. greater than:</td>
<td>buge</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>u. greater than or equal:</td>
<td>buge</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>signed less than:</td>
<td>bslt</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s. less than or equal:</td>
<td>bslt</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s. greater than:</td>
<td>bsgt</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s. greater than or equal:</td>
<td>bsgt</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>shift left:</td>
<td>bslr</td>
<td>n</td>
<td>t_1[n] &lt;&lt; t_2[n]</td>
</tr>
<tr>
<td>logical shift right:</td>
<td>bshr</td>
<td>n</td>
<td>t_1[n] &gt;&gt; u t_2[n]</td>
</tr>
<tr>
<td>arithmetic shift right:</td>
<td>bshr</td>
<td>n</td>
<td>t_1[n] &gt;&gt; u t_2[n]</td>
</tr>
<tr>
<td>extraction:</td>
<td>extract</td>
<td>n &gt; i ≥ j</td>
<td>i - j + 1</td>
</tr>
<tr>
<td>concatenation:</td>
<td>concat</td>
<td>n</td>
<td>t_1[n] ⊕ t_2[n]</td>
</tr>
<tr>
<td>zero extend:</td>
<td>zeroextend</td>
<td>n</td>
<td>extru (t[n], i)</td>
</tr>
<tr>
<td>sign extend:</td>
<td>signextend</td>
<td>n + i</td>
<td></td>
</tr>
<tr>
<td>rotate left:</td>
<td>rotl</td>
<td>n &gt; i ≥ 0</td>
<td>n</td>
</tr>
<tr>
<td>rotate right:</td>
<td>rotr</td>
<td>n &gt; i ≥ 0</td>
<td>n</td>
</tr>
<tr>
<td>repeat:</td>
<td>repeat</td>
<td>i &gt; 0</td>
<td>n · i</td>
</tr>
</tbody>
</table>

*continued on next page*
### Table 12.2: Syntax for common bit-vector operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary minus</strong></td>
<td>( \text{bv\texttt{-}not}(\textit{t}[n]) )</td>
<td>Denoted by ( i ), where ( i \in \mathbb{N} ) and ( i &lt; n ). Using vector notation, ( d ) is written as ( [d[n-1], \ldots, d[1], d[0]] ), i.e., the most significant bit standing on the left-hand side and the least significant bit on the right-hand side.</td>
</tr>
<tr>
<td><strong>Addition</strong></td>
<td>( \text{bv\texttt{add}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Denoted by ( + ) derivation with rounding to ( ) remainder signed division: ( \text{bvsrem})(\textit{t}_1[n], \textit{t}_2[n]) ) ( n )</td>
</tr>
<tr>
<td><strong>Subtraction</strong></td>
<td>( \text{bv\texttt{sub}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Denoted by ( - ) derivation with rounding to ( ) remainder.</td>
</tr>
<tr>
<td><strong>Multiplication</strong></td>
<td>( \text{bv\texttt{mul}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Derived by ( \times ) derivation with rounding to ( ) remainder.</td>
</tr>
<tr>
<td><strong>Unsigned division</strong></td>
<td>( \text{bvd\texttt{iv}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Denoted by ( ) derivation with rounding to ( ) remainder.</td>
</tr>
<tr>
<td><strong>Signed division</strong></td>
<td>( \text{bvs\texttt{div}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Denoted by ( ) derivation with rounding to ( ) remainder.</td>
</tr>
<tr>
<td><strong>With rounding to ( ) remainder</strong></td>
<td>( \text{bv\texttt{rem}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Denoted by ( ) derivation with rounding to ( ) remainder.</td>
</tr>
<tr>
<td><strong>With rounding to ( ) remainder</strong></td>
<td>( \text{bv\texttt{mod}}(\textit{t}_1[n], \textit{t}_2[n]) )</td>
<td>Denoted by ( ) derivation with rounding to ( ) remainder.</td>
</tr>
</tbody>
</table>

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

To facilitate the presentation, similar to [BS09]; [Fra10], we define an auxiliary bijective meta-function \( \text{nat}_n : D_n \rightarrow [0, 2^n - 1] \). Given a bit-vector \( d \in D_n \), \( \text{nat}_n(d) := \sum_{i=0}^{n-1} 2^i d[i] \). We also introduce the inverse meta-function \( \text{bv}_n := \text{nat}_n^{-1} \).

Tab. 12.3 shows the semantics of common operators. The evaluation function for terms is denoted by \( [ \cdot ] \). Furthermore, we use two abbreviations, one for the most significant bit of a bit-vector and one for absolute value:

\[
\text{msb}(t[n]) := \left[ \text{t}[n-1] \right] \\
\text{abs}(t[n]) := \left\{ \begin{array}{ll}
-t & \text{if msb}(t) \\
t & \text{otherwise}
\end{array} \right.
\]
continued from previous page

\begin{align*}
\text{bvult:} & \quad [t_1[n] <_u t_2[n]] := \text{bv}_1 (\text{nat}_n ([t_1]) < \text{nat}_n ([t_2])) \\
\text{bvule:} & \quad [\text{bvule}(t_1[n], t_2[n])] := [\sim (t_2 <_u t_1)] \\
\text{bvugt:} & \quad [\text{bvugt}(t_1[n], t_2[n])] := [t_2 <_u t_1] \\
\text{bvgue:} & \quad [\text{bvgue}(t_1[n], t_2[n])] := [\text{bvule}(t_2, t_1)] \\
\text{bvsit:} & \quad [\text{bvsit}(t_1[n], t_2[n])] := \text{bv}_1 \left( (\msb(t_1) \land \sim \msb(t_2)) \lor \right.
onumber\left. (\msb(t_1) \leftrightarrow \msb(t_2)) \land [t_1 <_u t_2] \right) \\
\text{bvsle:} & \quad [\text{bvsle}(t_1[n], t_2[n])] := [\sim \text{bvslt}(t_2, t_1)] \\
\text{bvsqt:} & \quad [\text{bvsqt}(t_1[n], t_2[n])] := [\text{bvslt}(t_2, t_1)] \\
\text{bvsge:} & \quad [\text{bvsge}(t_1[n], t_2[n])] := [\text{bvsle}(t_2, t_1)] \\
\text{bvsll:} & \quad [t_1[n] \ll_<_u t_2[n]] := \text{bv}_n \left( \text{nat}_n ([t_1]) \cdot 2^k \mod 2^n \right) \text{ where } k := \text{nat}_n ([t_2]) \\
\text{bvssh:} & \quad [t_1[n] \gg_>_u t_2[n]] := \text{bv}_n \left( \text{nat}_n ([t_1]) / 2^k \right) \text{ where } k := \text{nat}_n ([t_2]) \\
\text{extract:} & \quad [t[n] [i : j]] := \text{bv}_{i+1} \left( \text{nat}_n ([t]) / 2^j \right) \mod 2^n \\
\text{concat:} & \quad [t_1[n] \circ t_2[n]] := \text{bv}_{m+n} \left( 2^n \text{nat}_m ([t_1]) + \text{nat}_n ([t_2]) \right) \\
\text{zero extend:} & \quad [\text{ext}_u(t[n], i)] := \text{bv}_{n+i} \left( \text{nat}_n ([t]) \right) \\
\text{sign extend:} & \quad [\text{sign extend}(t[n], i)] := \left\{ \begin{array}{ll}
\text{bv}_{n+i} \left( 2^{n+i} - 2^n + \text{nat}_n ([t]) \right) & \text{if } \msb(t) \\
[\text{ext}_u(t[n], i)] & \text{otherwise}
\end{array} \right.
\text{otherwise} \\
\text{rotate left:} & \quad [\text{rotate left}(t[n], i)] := \left\{ \begin{array}{ll}
[t] & \text{if } n = 1 \lor i = 0 \\
[t \cdot i - 1] : 0 \circ t [n-1 : n-i] & \text{otherwise}
\end{array} \right.
\text{otherwise} \\
\text{rotate right:} & \quad [\text{rotate right}(t[n], i)] := \left\{ \begin{array}{ll}
[t] & \text{if } n = 1 \lor i = 0 \\
[t \cdot i - 1] : 0 \circ t [n-1 : n-i] & \text{otherwise}
\end{array} \right.
\text{otherwise} \\
\text{repeat:} & \quad [\text{repeat}(t[n], i)] := \left\{ \begin{array}{ll}
[t] & \text{if } i = 1 \\
[t \circ \text{repeat}(t, i - 1)] & \text{otherwise}
\end{array} \right.
\text{otherwise} \\
\text{bvneg:} & \quad [\sim t[n]] := \text{bv}_n \left( 2^n - \text{nat}_n ([t]) \right) \\
\text{bvaadd:} & \quad [t_1[n] + t_2[n]] := \text{bv}_n \left( \text{nat}_n ([t_1]) + \text{nat}_n ([t_2]) \mod 2^n \right) \\
\text{bvsub:} & \quad [t_1[n] - t_2[n]] := [t_1 + (-t_2)] \\
\text{bvmul:} & \quad [t_1[n] \cdot t_2[n]] := \text{bv}_n \left( \text{nat}_n ([t_1]) \cdot \text{nat}_n ([t_2]) \mod 2^n \right) \\
\text{bvudiv:} & \quad [t_1[n] /_u t_2[n]] := \text{bv}_n \left( \left\lfloor \text{nat}_n ([t_1]) / \text{nat}_n ([t_2]) \right\rfloor \right) \\
\text{bvurem:} & \quad [\text{bvurem}(t_1[n], t_2[n])] := [t_1 - (t_1 /_u t_2) \cdot t_2] \\
\text{bvudiv:} & \quad [\text{bvudiv}(t_1[n], t_2[n])] := \left\{ \begin{array}{ll}
\text{abs}(t_1) /_u \text{abs}(t_2) & \text{if } \msb(t_1) = \msb(t_2) \\
(\sim \text{abs}(t_1) /_u \text{abs}(t_2)) & \text{otherwise}
\end{array} \right.
\text{otherwise}
\end{align*}

continued on next page
12.8 APPENDIX

Table 12.3: Semantics for common bit-vector operators

\[
\text{bvsrem: } \text{bvsrem}(t_1, t_2) := \begin{cases} 
- \text{burem}(\text{abs}(t_1), \text{abs}(t_2)) & \text{if msb}(t_1) \\
\text{burem}(\text{abs}(t_1), \text{abs}(t_2)) & \text{otherwise}
\end{cases}
\]

\[
\text{bvsmod: } \text{bvsmod}(t_1, t_2) := \begin{cases} 
\text{bvsrem}(t_1, t_2) & \text{if } \text{bvsrem}(t_1, t_2) \neq 0 \\
\text{bvsrem}(t_1, t_2) + t_2 & \text{otherwise}
\end{cases}
\]

12.8.2 Additional Proofs

In the following proofs, we are going to measure the overall size of structures and their bit-vector representations, as the sum of the sizes of all their components.

The overall size of a structure \( A = (U, P_1^{\tau}, \ldots, P_k^{\tau}) \) is defined as \( |A| = |U| + \sum_{i=1}^{k} |P_i| \).

Analogously, we define the overall size of a bit-vector representation as \( |(\Psi, \nu)| = \sum_{i=1}^{k} |\Psi_i| \) where \( \Psi_1, \ldots, \Psi_k \) are the bit-vector formulas in \( \Psi \).

12.8.2.1 Proofs for Sec. 12.5

**Theorem 12.5.** Given a problem \( A \), a standard complexity class \( C \), and an operator set \( \Omega \subseteq \Pi \), if \( A \in C \), then \( \text{bv}^\Omega(A) \in \text{Exp}_v(C) \).

**Proof.** Let us assume that \( A \subseteq \text{Struct}(\tau) \). Given an instance \((\Psi, \nu) \in \text{bv}^\Omega(A)\), we are going to bit-blast each formula \( \psi_P(x_1^{[s_1]}, \ldots, x_k^{[s_k]}) \) in \( \Psi \), where \( P^{\tau} \) is a relation symbol in \( \tau \). I.e., we construct the Boolean formula \( \phi_P = \text{bblast}^\tau(\psi_P) \). Since, by assumption, \( \Omega \) allows log-space computable bit-blasting in bit-width, we know that there exists a polynomial \( p \) such that \( |\phi_P| \leq p(\Sigma_{i=1}^{k} n_i) \), where \( n_i = \text{decod}_v(s_i) \).

We know that \( n_i \leq \exp_{v-1}(|s_i|) \). (In particular, \( n_i = |s_i| \) if \( v = 1 \).) Hence, \( \sum_{i=1}^{k} n_i \leq \sum_{i=1}^{k} \exp_{v-1}(|s_i|) \leq \exp_{v-1} \left( \sum_{i=1}^{k} |s_i| \right) \leq \exp_{v-1} \left( |\Psi_P| \right) \). Consequently, \( |\phi_P| \leq p(\exp_{v-1}(|\Psi_P|)) \).

The bit-vector formula \( \psi_P \) and the Boolean formula \( \phi_P \) encode the same relation \( \tilde{P}^\tau = \text{gen}^{\tau}_v(\psi) \), whose size can be constrained as follows: since \( |\tilde{P}| \leq 2^{\exp_1(|\Psi_P|)} = \exp_1(|\Psi_P|) \), we know that \( |\tilde{P}| \leq \exp_1(p(|\Psi_P|)) \).

Consequently, the relations in the structure \( \text{gen}^{\tau}_v(\Psi) \) are of size at most \( \exp_1(p(|\Psi|)) \) altogether. Obviously, the universe of \( \text{gen}^{\tau}_v(\Psi) \) cannot exceed this upper limit either. Therefore, \( \|\text{gen}^{\tau}_v(\Psi)\| \leq 2\exp_1(p(|\Psi|)) \).

Since \( \text{gen}^{\tau}_v(\Psi) \in A \) and, by assumption, \( A \in C \), it follows that \( \text{bv}^\Omega(A) \in \text{Exp}_v(C) \). \( \square \)

12.8.2.2 Proofs for Sec. 12.6

Recall the notions of a relational signature, a structure, and a computational problem from Sec. 16.2. First-order logic \( F\Omega\alpha(\tau) \) is the language of all first-order formulas over the signature \( \tau \) with logical predicates for equality = and successor s(ω, ·), and two constants 0 resp. \( \max \) denoting the minimal resp. maximal element w.r.t. the successor relation. Given \( \phi(x_1, \ldots, x_k) \in F\Omega\alpha(\tau) \) and a structure \( A \in \text{Struct}(\tau) \) with universe \( U \), let \( rel_A(\phi) \) denote the relation defined by \( \phi \) over \( A \), i.e., the set of all satisfying assignments for \( \phi \) over \( A \). Formally, \( rel_A(\phi) = \{ (d_1, \ldots, d_k) \in U^k \mid A \models \phi(d_1, \ldots, d_k) \} \).
For signatures $\sigma$ and $\tau = (P_1^{|}, \ldots, P_k^{|})$, and $m \in \mathbb{N}^+$, the $m$-ary interpretation of $\sigma$ into $\tau$ is a tuple of formulas $(\phi_1, \ldots, \phi_k)$ where each $\phi_i(x_1, \ldots, x_{a_i}) \in \mathcal{F}\Omega_4(\sigma)$. Intuitively, an interpretation gives the definition of each $\tau^{(m)}$ relation in terms of $\sigma$ relations, where $\tau^{(m)} = (P_1^{(|m)}, \ldots, P_k^{(|m)})$ is the $m$-ary variant (or vectorization) of $\tau$. Given a structure $A \in \text{Struct}(\sigma)$, to obtain the structure $I(A) \in \text{Struct}(\tau^{(m)})$ specified by $I$, we define $I(A) = (U, \text{rel}_A(\phi_1), \ldots, \text{rel}_A(\phi_k))$.

Given a fragment $F$ of first-order logic and two problems $A \subseteq \text{Struct}(\sigma)$ and $B \subseteq \text{Struct}(\tau)$, we say that $A$ has an $F$-reduction to $B$, denoted by $A \leq_F B$, if there exists an interpretation $I$ of $\sigma$ into $\tau$ such that $I$ consists of $F$-formulas and for all $A \in \text{Struct}(\sigma)$, $A \in A$ iff $I(A) \in B$. Let $A \leq_{qf} B$ resp. $A \leq_{L} B$ denote that $A$ has a quantifier-free resp. log-space reduction to $B$. Note that quantifier-free reductions are weaker than log-space reductions, i.e., $A \leq_{qf} B$ implies $A \leq_{L} B$.

**Lemma 12.6 (Conversion Lemma).** Let $\Omega \supseteq \{\&, \lor, \neg, =, +, 1\}$. Given two problems $A \subseteq \text{Struct}(\sigma)$ and $B \subseteq \text{Struct}(\tau)$, if $A \leq_{qf} B$, then $\text{bv}_\tau^\Omega(A) \leq_{L} \text{bv}_\sigma^\Omega(B)$, for any $\nu$.

**Proof.** First, we construct some bit-vector formulas that express useful relations on $l$-ary term vectors $u$ and $v$ having the bit-width signature $s_1, \ldots, s_l$. Our intention is to use these constructs in the rest of the proof.

1. **Equality.** Let $\text{eq}_\nu(u, v)$ denote the fact that $u$ and $v$ are bitwise equal. Then $\text{eq}_\nu$ can be defined as follows:

$$\text{eq}_\nu(u, v) = \bigwedge_{i=1}^{l} u_i^{[s_i]} = v_i^{[s_i]}$$

2. **Successor.** Let $\text{succ}_\nu(u, v)$ denote that the number encoded by $v$ is the successor of the one encoded by $u$. Let $n_i = \text{decode}_\nu(s_i)$. In order to define $\text{succ}_\nu$, let us first introduce the following notations:

$$\alpha_i = \bigwedge_{j=i+1}^{l} u_j^{[s_i]} = v_j^{[s_i]}, \quad \beta_i = \bigwedge_{j=1}^{i-1} u_j^{[s_i]} = 0^{[s_i]} \land v_j^{[s_i]} = 0^{[s_i]}$$

Note that, instead of encoding the value for $u_j^{[s_i]}$ as the bit-vector constant $2_{n_j}$, we rather use $0^{[s_i]}$, in order to avoid exponential blowup in $\beta_i$.

Then $\text{succ}_\nu$ can be defined as follows:

$$\text{succ}_\nu(u, v) = \bigvee_{i=1}^{l} \alpha_i \land u_i^{[s_i]} + 1 = v_i^{[s_i]} \land \beta_i$$

By assumption, there exists a quantifier-free $m$-ary reduction $I$ from $A$ to $B$, where $I$ is represented as a $\tau$-tuple of quantifier-free $\mathcal{F}\Omega_4(\sigma)$ formulas. For any $R^b$ in $\tau^{(m)}$, let $\phi_R(x_1, \ldots, x_b)$ denote the corresponding formula in $I$.

Given an instance $(\Psi, \nu) \in \text{bv}_\nu^\Omega(A)$, let $A$ denote $\text{gen}_\nu^\Omega(\Psi)$. Let $U$ denote the universe of $A$. In order to propose a reduction $M_I$ from $\text{bv}_\nu^\Omega(A)$ into $\text{bv}_\sigma^\Omega(B)$, we show how to construct the image $M_I(\Psi, \nu) = (\Psi', \nu)$. For each relation symbol $P^a$ in $\sigma$, $\Psi$ gives a definition in the form $P(y_1, \ldots, y_a) := \phi_P(y_1, \ldots, y_a)$, where each $y_i$ has the bit-width signature $s_1, \ldots, s_l$. Let $n_i = \text{decode}_\nu(s_i)$.

---

2Recall that in a bit-vector constant $c^{|i}$, the number $c$ is not a scalar and, therefore, is always encoded in binary form.
We now show how to construct the definition of an $R^b$ in $\tau^{(m)}$, in the form $R(z_1, \ldots, z_b) := \psi^*_R(z_1, \ldots, z_b)$. Each $z_i$ has the bit-width signature $s_1, \ldots, s_i$. From $\phi_R(x_1, \ldots, x_b)$ in $I$, we obtain $\psi^*_R(z_1, \ldots, z_b) = tr(\phi_R)$, where the translation function $tr$ is inductively defined as follows:

\[
\begin{align*}
tr(x_i) &= z_i \\
tr(0) &= 0^{[s_i], \ldots, 0^{[s_i]}} \\
tr(\text{max}) &= \sim 0^{[s_i], \ldots, \sim 0^{[s_i]}} \\
tr(P(u_1, \ldots, u_b)) &= \psi_p(tr(u_1), \ldots, tr(u_b)) \\
tr(u = v) &= eq_v(tr(u), tr(v)) \\
tr(s(u, v)) &= succ_v(tr(u), tr(v))
\end{align*}
\]

The construction of a term vector for $\text{max}$ requires some explanation. The number $|U| - 1$ has to be encoded as a term vector, without exponential blowup. We can exploit the fact that, by Def. 12.3, $|U| = 2^{n_1 + \cdots + n_b}$. Therefore, instead of encoding $|U| - 1$ directly by bit-vector constants that might consist of exponentially many 1s, we can rather encode it by using $\sim 0^{[h_i]}$.

\[\square\]

**Lemma 12.8.** Given a problem $A$, $A \leq_L \text{bv}^\Omega_1(\text{long}^\nu_1(A))$ if one of the following conditions holds:

1. $v = 1$ and $\Omega \supseteq \{ <_u \}$
2. $v > 1$ and $\Omega \supseteq \{ = \}$

**Proof.** Given $A = (U, \vec{R}_1, \ldots, \vec{R}_k) \in A$, let us choose a structure $(V, \vec{R}_1^1) \in \text{long}^\nu_1(A)$. Let us recall that $|\vec{R}_i| = |V|$. Therefore, we need to encode $\vec{R}_1^1$ as a bit-vector formula $\psi_R$ that has exactly $|V| = exp_{v-1}(\text{char}(A))$ satisfying assignments and contains exactly one variable. For the two different cases above, we can choose $\psi_R$ as follows:

1. $\psi_R = (x^{[s_i]} <_u \text{char}(A)^{[s_i]})$, where $s = \text{encode}_1\left(1 + \sum_{i=1}^{k} |U|^{n_i}\right)$, which is by definition equal to $\text{char}(A)$
2. $\psi_R = (x^{[s_i]} = x^{[s_i]})$, where $s = \text{encode}_v(\text{char}(A))$.

By setting $\Psi = (R(x^{[s_i]} := \psi_R(x^{[s_i]}))$, the bit-vector representation $(\Psi, v)$ falls into $\text{bv}^\Omega_1(\text{long}^\nu_1(A))$. \[\square\]

**Corollary 12.10.** Given a standard complexity class $C$ and a problem $A$, if $A$ is $C$-hard under quantifier-free reductions, then $\text{bv}^\Omega_1(A)$ is $\text{Exp}_v(C)$-hard under log-space reductions if one of the following conditions holds:

1. $v = 1$ and $\Omega \supseteq \{ \wedge, \vee, \neg \}$
2. $v > 1$ and $\Omega \supseteq \{ \wedge, \vee, \neg, =, +_1 \}$

**Proof.** In order to apply Thm. 12.9, we need to show that $\text{long}^\nu_1(\text{Exp}_v(C)) \subseteq C$. We show that for any problem $A \in \text{Exp}_v(C)$, it holds that $\text{long}^\nu_1(A) \in C$.

For any structures $A \in C$ and $A' \in \text{long}^\nu_1(A)$, we know that $|A'| = 2^{\exp_{v-1}(\text{char}(A))}$, by Def. 12.7. By denoting the universe of $A$ as $U$, it holds, also by definition, that $\text{char}(A) \geq 2^{|U|}$ and $\text{char}(A) \geq 2^{|A| - |U|}$, and thus $\text{char}(A) \geq 2^{\frac{1}{2}|A|} = \exp_{1}\left(\frac{1}{2}|A|\right)$. Consequently, it follows that $|A'| \geq 2^{\exp_{v}\left(\frac{1}{2}|A|\right)} \geq \exp_{v}\left(|A|\right)$.

Since from $A' \in \text{long}^\nu_1(A)$ one can directly construct a structure that is isomorphic with $A \in A$, where $A \in \text{Exp}_v(C)$, it holds that $\text{long}^\nu_1(A) \in C$. \[\square\]
12.8.3 Definability of Bit-Vector Fragments

**Definition 12.11** (Definability). Given two operator sets $\Omega_1$ and $\Omega_2$, we say that $\BV_{\nu}^{\Omega_1}$ is definable by $\BV_{\nu}^{\Omega_2}$ if for any formula $\psi_1(x_1[s], \ldots, x_k[s]) \in \BV_{\nu}^{\Omega_1}$, there exists a formula

$$\psi_2(x_1[s], \ldots, x_k[s], x_{k+1}[s], \ldots, x_{k+l}[s]) \in \BV_{\nu}^{\Omega_2}$$

where $x_{k+1}[s], \ldots, x_{k+l}[s]$ are new variables to $\psi_1$, such that $\forall d_i \in \mathbb{B}^{n_i}, \ldots, d_k \in \mathbb{B}^{n_k}$

$$\exists ! d_{k+1} \in \mathbb{B}^{n_{k+1}}, \ldots, d_{k+l} \in \mathbb{B}^{n_{k+l}}. \quad \psi_1(d_1, \ldots, d_k) = \text{true} \iff \psi_2(d_1, \ldots, d_k, d_{k+1}, \ldots, d_{k+l}) = \text{true}$$

where $n_i = \text{decode}_v(s_i)$.

Note that the additional bit-vectors $d_{k+1}, \ldots, d_{k+l}$ are uniquely existentially quantified. Therefore, in fact, each new variable $x_{k+i}$ can rather be considered as a bit-vector function $f_i(x_1[s], \ldots, x_k[s]) : \mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k} \mapsto \mathbb{B}^{n_{k+i}}$. Thus, $\psi_1$ and $\psi_2$ encode the same $k$-ary relation.

We say that $\BV_{\nu}^{\Omega_1}$ is log-space definable in bit-width by $\BV_{\nu}^{\Omega_2}$ if $\psi_2$ can be computed in log-space in $\sum_{i=1}^k n_i$.

**Proposition 12.12.** Given two operator sets $\Omega_1$ and $\Omega_2$ such that $\BV_{\nu}^{\Omega_1}$ is log-space definable in bit-width by $\BV_{\nu}^{\Omega_2}$, it holds that $\bv_{\nu}^{\Omega_1}(A) \leq_L \bv_{\nu}^{\Omega_2}(A)$, for any problem $A$.

Our hardness results rely on the operator set $\Omega = \{ \land, \lor, \sim, =, +_1 \}$. As a consequence of Prop. 12.12, the hardness result of Corr. 12.10 can be extended to any operator set that can be defined by $\Omega$. For instance, we can easily show hardness for the following operator sets, in particular for that includes $\ll$, i.e., left shift by one, which serves as a fundamental operator in hardware verification problems [FKB13a].

**Corollary 12.13.** Let $\Omega_+ = \{ \land, \lor, \sim, =, +_1 \}$ and $\Omega_{\ll 1} = \{ \& , |, \sim, \oplus, =, \ll_1 \}$. If $A$ is C-hard under quantifier-free reductions, then $\bv_{\nu}^{\Omega_+}(A) \leq_L \bv_{\nu}^{\Omega_{\ll 1}}(A)$ are Exp$_\nu(C)$-hard under log-space reductions.

**Proof.** Following Corr. 12.10 and Prop. 12.12, we only need to show that $\BV_{\nu}^{\Omega_+}$ is log-space definable in bit-width by $\BV_{\nu}^{\Omega_+}$ and $\BV_{\nu}^{\Omega_{\ll 1}}$, respectively.

It is obvious that a formula $\psi_1 \in \BV_{\nu}^{\Omega_+}$ can translated to an equivalent formula $\psi_2 \in \BV_{\nu}^{\Omega_{\ll 1}}$, since $+_1$ is just a restricted version of $+$. To prove the second statement, each term $t^s + 1$ can be replaced by $t^s \oplus \text{cin}^s$, working as an adder, where $\text{cin}^s$ is a new variable to $\psi_1$, and let $\psi_2 \in \BV_{\nu}^{\Omega_{\ll 1}}$ be the conjunction of the resulting formula and the following expression: $\text{cin}^s = ( (t^s \& \text{cin}^s) \ll 1 ) \ll t^s$. \qed
BV2EPR: A TOOL FOR POLYNOMIALLY TRANSLATING QUANTIFIER-FREE BIT-VECTOR FORMULAS INTO EPR
13.1 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 [BST10], the BTOR [BBL08], 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 [KFB12], 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\_2 NE\_xp\_T\_ime-complete.\(^1\) Thus, bit-blasting is not polynomial in general. For a polynomial reduction, the target logic has to be NE\_xp\_T\_ime-hard.

In this paper, we introduce our new tool \texttt{bv2epr}. \texttt{bv2epr} translates QF\_BV formulas into Effectively Propositional Logic (EPR), which is NE\_xp\_T\_ime-complete [Lew80], by using a new (polynomial) reduction. This is in contrast to existing translations in [KKV09]; [Emm+10], which produce exponential EPR formulas in general, as we will point out in Sect. 13.2.1. We give some experimental results in Sect. 16.6 with the EPR solver iProver.

13.2 Preliminaries

We assume the usual syntax for QF\_BV. A bit-vector term \( t \) of bit-width \( n \) (\( n \in \mathbb{N}, n \geq 1 \)) is denoted by \( t^{[n]} \). An atomic term can be either (a) a bit-vector constant \( c^{[n]} \), where \( c \in \mathbb{N}, 0 \leq c < 2^n \); or (b) a bit-vector variable \( v^{[n]} \). Compound terms and formulas can contain the usual bit-vector operators (c.f. SMT-LIB [BST10]), like e.g. bitwise operators, shifts, arithmetic operators, relational operators, etc. The decision problem for QF\_BV is NE\_xp\_T\_ime-complete [KFB12].

EPR, known as the Bernays-Schönfinkel class, is a NE\_xp\_T\_ime-complete fragment of first-order logic [Lew80]. 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

\(^1\)In [KFB12], we introduced the notation QF\_BV\_1 resp. QF\_BV\_2 for QF\_BV using a \textit{unary} resp. a \textit{logarithmic}, actually without loss of generality, binary encoding.
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.

13.2.1 Existing Translations

In [KKV09], 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 [Emm+10], since bit-vectors are modeled as unary predicates. These predicates are over bit-indices, represented by dedicated constants. For instance, the $i$th bit of a bit-vector $x[n]$, $0 \leq i < n$, is represented by the atom $p_x(\text{bitInd}_i)$, where $\text{bitInd}_i$ is a constant. Note that for QF_BV, such a translation might introduce exponentially many constants, since bit-widths like $n$ are encoded logarithmically. The so-called range-aware relational encoding in [Emm+10], furthermore, introduces exponentially many assertions into the EPR formula in general, e.g., atoms $\text{less}_k(\text{bitInd}_i)$ for all $0 \leq i < k$. Finally, not all the QF_BV bit-vector operators are addressed by the relational encoding, but only equality. All the arithmetic operators are assumed to be synthesized/bit-blasted in the verification front-end [Emm+10], potentially leading to an exponential blowup already before the actual encoding. In [KKV09], 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 with our translation in Sect. 13.3.1.

13.3 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 [BV2EPR]. The architecture of **bv2epr** can be seen in Fig. 13.1, 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 [Hod+12]: (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

---

*Bitwise operators could be handled in a similar way.*
the graph data structure, clauses can be extracted directly, i.e., no additional approach for clause generation is needed.

13.3.1 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. To each bit-vector term of bit-width n, a dedicated \( \lfloor \log_2 n \rfloor \)-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 \( \lor \)): 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:

\[
p_{or}(i_{n-1}, \ldots, i_0) \iff p_x(i_{n-1}, \ldots, i_0) \lor p_y(i_{n-1}, \ldots, i_0)
\]

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.

\[
\begin{align*}
add^{2^n} &= x^{2^n} \oplus y^{2^n} \oplus cin^{2^n} \\
cin^{2^n} &= cout^{2^n} \ll 1 \\
cout^{2^n} &= (x^{2^n} \& y^{2^n}) | (x^{2^n} \& cin^{2^n}) | (y^{2^n} \& cin^{2^n})
\end{align*}
\]

Note that Eqn. (13.1) and (13.3) only contain bitwise operators (and equality). Therefore, both can be translated into EPR2 as introduced previously. Only Eqn. (13.2), which contains shift by 1, has to be handled differently.

We introduce a helper predicate \( \text{succ} \) 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 \( \text{succ}(i_{n-1}, \ldots, i_0, j_{n-1}, \ldots, j_0) \) can be defined by \( n \) facts:

\[
\begin{align*}
\text{succ}(i_{n-1}, \ldots, i_3, i_2, i_1, 0, i_{n-1}, \ldots, i_3, i_2, i_1, 1) \\
\text{succ}(i_{n-1}, \ldots, i_3, i_2, 0, 1, i_{n-1}, \ldots, i_3, i_2, 1, 0) \\
\text{succ}(i_{n-1}, \ldots, i_3, 0, 1, i_{n-1}, \ldots, i_3, 1, 0, 0) \\
\vdots \\
\text{succ}(0,1, \ldots, 1, 1,0, \ldots, 0)
\end{align*}
\]

Using this helper predicate, Eqn. (13.2) can be translated into EPR2 as follows:

\[
\neg p_{cin}(0, \ldots, 0) \implies (p_{cin}(j_{n-1}, \ldots, j_0) \iff p_{cout}(i_{n-1}, \ldots, i_0))
\]

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.

\(^3\)The Herbrand universe of EPR2 can be considered as the Boolean domain.
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 \( i, 0 \leq i < n \), the \( i \)th bit of \( y \) is checked: if it is 1, then left shift by \( 2^i \) has to be done.

\[
\neg p_y(0, \ldots, 0) \Rightarrow \left( p_{\text{shl}}(i_{n-1}, \ldots, i_0) \leftrightarrow p_x(i_{n-1}, \ldots, i_0) \right)
\]

\[
\neg p_y(0, \ldots, 0) \land \text{succ}(i_{n-1}, \ldots, i_0, j_{n-1}, \ldots, j_0) \Rightarrow \left( p_{\text{shl}}(i_{n-1}, \ldots, i_0) \leftrightarrow p_x(i_{n-1}, \ldots, i_0) \right)
\]

\[
\neg p_y(0, \ldots, 0, 1) \Rightarrow \left( p_{\text{shl}}(i_{n-1}, \ldots, i_0) \leftrightarrow p_0(i_{n-1}, \ldots, i_0) \right)
\]

\[
\neg p_y(0, \ldots, 0, 1) \land \text{succ}(0, i_{n-1}, \ldots, i_1, 0, j_{n-1}, \ldots, j_1) \Rightarrow \left( p_{\text{shl}}(i_{n-1}, j_1, \ldots, j_1, i_0) \leftrightarrow p_0(i_{n-1}, \ldots, i_0) \right)
\]

\[
\vdots
\]

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 \( p_{\text{mul}} \): the \( i \)th bit of the \( j \)th subproduct is represented by the atom \( p_{\text{mul}}(j_{n-1}, \ldots, j_0, i_{n-1}, \ldots, i_0) \).

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 \( x \).

\[
\neg p_y(1, \ldots, 1) \Rightarrow \neg p_{\text{mul}}(1, \ldots, 1, i_{n-1}, \ldots, i_0)
\]

\[
p_y(1, \ldots, 1) \Rightarrow (p_{\text{mul}}(1, \ldots, 1, i_{n-1}, \ldots, i_0) \leftrightarrow p_x(i_{n-1}, \ldots, i_0))
\]

The \( j \)th subproduct, \( 0 \leq j < 2^n - 1 \), is computed by checking the \( j \)th bit of \( y \): if it is 0, then the \((j + 1)\)th subproduct has to be shifted left by \( 1 \) (represented by the predicate \( p_{\text{shl}} \)); otherwise, the shifted subproduct and \( x \) have to be added (represented by \( p_{\text{add}} \)).

\[
\left( \neg p_y(j_{n-1}, \ldots, j_0) \land \text{succ}(j_{n-1}, \ldots, j_0, j'_{n-1}, \ldots, j'_0) \right) \Rightarrow \left( p_{\text{mul}}(j_{n-1}, \ldots, j_0, i_{n-1}, \ldots, i_0) \leftrightarrow \right.
\]

\[
\left. \left( p_{\text{shl}}(j'_{n-1}, \ldots, j'_0, i_{n-1}, \ldots, i_0) \leftrightarrow \right) \right)
\]

\[
\left( p_{\text{shl}}(j'_{n-1}, \ldots, j'_0, i_{n-1}, \ldots, i_0) \leftrightarrow \right)
\]

\[
\left. \left( p_{\text{add}}(j'_{n-1}, \ldots, j'_0, i_{n-1}, \ldots, i_0) \leftrightarrow \right) \right)
\]

The final product is given by \( p_{\text{mul}}(0, \ldots, 0, i_{n-1}, \ldots, i_0) \).

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 paper and is part of future work. We also investigated correctness empirically by exhaustively testing consistency of the solving results by Boolector and bv2epr+ilProver, for each bit-vector operation, up to a certain bit-width.

13.4 Benchmarks and Experiments

Solving QF_BV formulas in general is \textsc{NExpTime}-complete [KFB12]. However, certain families of QF_BV formulas are in \textsc{NP}, under certain restrictions on the bit-widths. We called this kind of families \textit{bit-width bounded} [KFB12]. Since solving EPR formulas is \textsc{NExpTime}-complete, our translation fits well to families which are not bit-width bounded. In [KFB12], two examples of this kind were given: (a) QF_BV/brummayerbire3/mulshbw represents instances of computing the high-order half of product problem, parameterized by the bit-width of multiplicands (bw); (b) QF_BV/brumtonessso/1fsrc/1fsrc_bv橱柜 formalizes the behavior of a linear feedback shift register [BS09]. We further propose two new benchmark families that are not bit-width bounded: (a) \textit{add2n} describes how bit-vectors of bit-width 2bw can be added by using two adders for bit-vectors of bit-width bw. (b) \textit{addmulbw} 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. Tab. 16.1 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 (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 Tab. 16.2, we give the results for the simplified SMT2 files.

13.5 Conclusion

We presented \textsc{bv2epr}, a tool for polynomially translating QF\_BV into EPR. The motivation for our tool lies in previous work [KFB12], 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 resp. constraints in the general case [KKV09]; [Emm+10]. In contrast to this, the Translator used in \textsc{bv2epr} always produces EPR formulas of polynomial size.

After discussing \textsc{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...

<table>
<thead>
<tr>
<th>bmark</th>
<th>bw</th>
<th>smt2</th>
<th>btor</th>
<th>Boolector</th>
<th>aig</th>
<th>cnf</th>
<th>Lingeling</th>
<th>epr</th>
<th>iProver</th>
</tr>
</thead>
<tbody>
<tr>
<td>muls</td>
<td>8²</td>
<td>947</td>
<td>1K</td>
<td>10.38</td>
<td>3K</td>
<td>44K</td>
<td>9.08</td>
<td>45K</td>
<td>1m 44s</td>
</tr>
<tr>
<td></td>
<td>16²</td>
<td>959</td>
<td>1K</td>
<td>TO</td>
<td>12K</td>
<td>203K</td>
<td>TO</td>
<td>55K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>64²</td>
<td>982</td>
<td>2K</td>
<td>TO</td>
<td>221K</td>
<td>4M</td>
<td>TO</td>
<td>78K</td>
<td>TO</td>
</tr>
<tr>
<td>lfsr25,6,16</td>
<td>63²</td>
<td>6K</td>
<td>9K</td>
<td>0.28</td>
<td>64K</td>
<td>258K</td>
<td>0.78</td>
<td>56K</td>
<td>18.08</td>
</tr>
<tr>
<td></td>
<td>127²</td>
<td>7K</td>
<td>9K</td>
<td>1.28</td>
<td>139K</td>
<td>545K</td>
<td>1.38</td>
<td>61K</td>
<td>1m 14s</td>
</tr>
<tr>
<td></td>
<td>1023</td>
<td>7K</td>
<td>11K</td>
<td>5.18</td>
<td>1M</td>
<td>5M</td>
<td>4.78</td>
<td>74K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>8191</td>
<td>7K</td>
<td>18K</td>
<td>2m 37s</td>
<td>11M</td>
<td>43M</td>
<td>3m 10s</td>
<td>89K</td>
<td>TO</td>
</tr>
<tr>
<td>add1n</td>
<td>26</td>
<td>452</td>
<td>455</td>
<td>0.08</td>
<td>3K</td>
<td>25K</td>
<td>0.18</td>
<td>12K</td>
<td>1m 21s</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>456</td>
<td>671</td>
<td>0.18</td>
<td>7K</td>
<td>53K</td>
<td>0.78</td>
<td>13K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>484</td>
<td>8K</td>
<td>3m 5s</td>
<td>549K</td>
<td>4M</td>
<td>1m 28s</td>
<td>21K</td>
<td>TO</td>
</tr>
<tr>
<td>addmul</td>
<td>26</td>
<td>149</td>
<td>99</td>
<td>0.28</td>
<td>174K</td>
<td>3M</td>
<td>2.48</td>
<td>8K</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>149</td>
<td>99</td>
<td>2.78</td>
<td>3M</td>
<td>58M</td>
<td>3m 22s</td>
<td>11K</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>151</td>
<td>103</td>
<td>TO</td>
<td>48M</td>
<td>1G</td>
<td>TO</td>
<td>13K</td>
<td>0.18</td>
</tr>
</tbody>
</table>

Table 13.1: Evaluation for the original SMT2 file
Table 13.2: Evaluation for the simplified SMT2 file

on this kind of instances, formulas generated by \texttt{bv2epr} can also help providing challenging benchmarks for current state-of-the-art solvers. The tool \texttt{bv2epr} is available at [BV2EPR].
EFFICIENTLY SOLVING BIT-VECTOR PROBLEMS USING MODEL CHECKERS
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 [BB09], MathSAT [Bru+08], STP [GD07], Z3 [DMB08], and Yices [DMo6]. 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 [BST10], the BTOR [BBL08], 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 [KFB12], 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 EXPTime-complete. Thus, bit-blasting is not polynomial in general. Consider the following example (in SMT2 syntax):

```smt2
(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 [KFB13a], 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 EXPTime-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 [FKB13b], we showed that this
kind of formulas can be expressed by QF$_{BV} \leq_1$, a subset of QF$_{BV}$ which turned out to be PSPACE-complete. In order to prove this, we gave a polynomial translation from QF$_{BV} \leq_1$ to Sequential Circuits, similar to the one for linear arithmetic on non-fixed-size bit-vectors proposed in [SK12a]; [SK12b].

In this paper, we show how model checkers can be used to solve fixed-size bit-vector problems of this class. In contrast to [FKB13b] which provided the theoretical background, we now focus on experimental evaluation and analyze the potential benefits for efficiently solving bit-vector formulas. First, in Sec. 14.2, we provide a short overview of our translation as described in [FKB13b] and give some examples to show how we used this concept to convert SMT2 files to SMV. In Sec. 16.6, 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 checkers turn out to be faster by several orders of magnitude. We provide experimental data and discuss the results in detail. Finally, in Sec. 16.7, we conclude the paper and discuss further topics for future work.

14.2 QF$_{BV} \leq_1$ TO SMV

In [SK12a]; [SK12b], the authors gave a polynomial translation for linear arithmetic on non-fixed-size bit-vectors (QFPAartr) into Sequential Circuits. In contrast to [SK12a]; [SK12b], 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 [KMS00]. We adapted this translation in [FKB13b] to deal with fixed-size bit-vectors and extended it by various other operators like shift by one and indexing.

Given $\Phi \in$ QF$_{BV} \leq_1$ 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 [SK12a]; [SK12b], we assume w.l.o.g that $\Phi$ 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$_{BV} \leq_1$ formula can be written like this with only a linear growth in the number of original variables.

We encode each atom in $\Phi$ 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 [SK12a]; [SK12b], we have to consider the fact that all bit-vectors have a fixed bit-width.

Let $n_{\text{max}}$ 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 $\lceil \log_2(n_{\text{max}}) \rceil$ latches, i.e. polynomially in the size of $\Phi$.

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 $n_{\text{max}} \geq n$, this will always hold at some point.\footnote{In contrast to [SK12a], we assume that the input streams for all variables start with the least significant bit.}

Finally, after constructing all atomic circuits, their outputs are combined by logical gates following the Boolean structure of $\Phi$. 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 [FKB13b].

We now show the translation for the motivational example given in Sec. 14.1 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:
We then keep track of whether the counter already reached the value of a certain bit-width. This variable later serves as a guard for all atoms containing variables of the given bit-width:

\[
\text{init}(\text{counter}_{\text{gte}_1000000}) := \text{FALSE};
\]
\[
\text{next}(\text{counter}_{\text{gte}_1000000}) := \text{counter}_{\text{gte}_1000000} \lor
\quad (\text{counter}_0 \& \text{counter}_1 \& \ldots \& \neg \text{counter}_6 \& \ldots \& \text{counter}_{19});
\]

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:

\[
\text{init}(\text{atom}_{\text{equal}}) := \text{TRUE};
\]
\[
\text{next}(\text{atom}_{\text{equal}}) := \text{case}
\quad \text{counter}_{\text{gte}_1000000} : \text{atom}_{\text{equal}};
\quad \text{TRUE} : \text{atom}_{\text{equal}} \& (x \leftrightarrow y);
\text{esac};
\]

For translating `addition`, two atoms have to be introduced since the carry bit has to be remembered in the next step:

\[
\text{init}(\text{atom}_{\text{add}}) := \text{TRUE};
\]
\[
\text{next}(\text{atom}_{\text{add}}) := \text{case}
\quad \text{counter}_{\text{gte}_1000000} : \text{atom}_{\text{add}};
\quad \text{TRUE} : \text{atom}_{\text{add}} \& (z \leftrightarrow (x \oplus y \oplus \text{atom}_{\text{cin}}));
\text{esac};
\]
\[
\text{init}(\text{atom}_{\text{cin}}) := \text{FALSE};
\]
\[
\text{next}(\text{atom}_{\text{cin}}) := \text{case}
\quad \text{counter}_{\text{gte}_1000000} : \text{atom}_{\text{cin}};
\quad \text{TRUE} : \text{atom}_{\text{add}} \& ((x \& y) \lor (x \& \text{atom}_{\text{cin}}) \lor (y \& \text{atom}_{\text{cin}}));
\text{esac};
\]

The `shift` operator can be translated in a very similar way but will not be given here explicitely to keep the example short. Another way 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:

\[
\text{AG}(\neg \text{counter}_{\text{gte}_1000000} \lor \neg \text{atom}_{\text{add}} \lor \neg \text{atom}_{\text{shift}} \lor \text{atom}_{\text{equal}})
\]

We also implemented our translation including various operators in a tool called BV2SMV. Binaries and source code are available for download at [BV2SMV].

---

3The counter bits in the next-statement correspond to the binary representation of \(n - 1\) (i.e. \(999999_{10} = 11110100001001111112\) in our example).
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* [FKB13b]. 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 \( (x[n] + 1[n]) \leq y[n] \). 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 \( (x[n] + 1[n]) \leq y[n] \) 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 \( x[n], y[n] = 2^k \), 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 paper 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 [Cim+02] on the corresponding SMV instances. In order to involve state-of-the-art model checkers like Tip [ES03] and Ilmc\(^5\) (that uses techniques described in [Bra11]; [Bra+11]), 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 (resp. 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 [BV2SMV] together with generation scripts for all benchmarks in SMT2 format and our tool bv2smv.

Tab. 16.1 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,\(^4\)\(^5\)\(^6\)

\( ^4 \)http://fmv.jku.at/smvflatten/
\( ^5 \)http://eceee.colorado.edu/wpmu/iimc/
\( ^6 \)http://fmv.jku.at/hwmcc12/
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 Tab. 16.1, 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 \( \text{NuSMV-bw} \). IImc also offers BDD-based solving strategy, with both forward resp. backward reachability analysis; we reference IImc with default settings resp. with BDD-based forward resp. backward reachability analysis as IImc resp. IImc-BDD-fw resp. IImc-BDD-bw.

<table>
<thead>
<tr>
<th></th>
<th>STP</th>
<th>Boolector</th>
<th>MathSAT5</th>
<th>NuSMV-bw</th>
<th>IImc-BDD-bw</th>
<th>IImc</th>
<th>NuSMV</th>
<th>BLImc</th>
<th>Tip-BMC</th>
<th>Aigbmc</th>
<th>Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved</td>
<td>147</td>
<td>146</td>
<td>127</td>
<td>123</td>
<td>192</td>
<td>189</td>
<td>185</td>
<td>172</td>
<td>170</td>
<td>147</td>
<td>130</td>
</tr>
<tr>
<td>sat</td>
<td>23</td>
<td>32</td>
<td>13</td>
<td>23</td>
<td>32</td>
<td>29</td>
<td>32</td>
<td>27</td>
<td>31</td>
<td>9</td>
<td>31</td>
</tr>
<tr>
<td>unsat</td>
<td>124</td>
<td>114</td>
<td>114</td>
<td>100</td>
<td>160</td>
<td>160</td>
<td>153</td>
<td>140</td>
<td>143</td>
<td>138</td>
<td>99</td>
</tr>
<tr>
<td>time</td>
<td>206</td>
<td>190</td>
<td>310</td>
<td>171</td>
<td>160</td>
<td>160</td>
<td>160</td>
<td>148</td>
<td>233</td>
<td>266</td>
<td>295</td>
</tr>
<tr>
<td>space</td>
<td>1063</td>
<td>805</td>
<td>587</td>
<td>2180</td>
<td>8</td>
<td>24</td>
<td>9</td>
<td>74</td>
<td>38</td>
<td>95</td>
<td>1142</td>
</tr>
</tbody>
</table>

Table 14.1: Overall results for all solvers

Apart from those in Tab. 16.1, we tested other model checkers as well, all submitted to HWMCC’12. We excluded some of them due to uncertain 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, \( \text{NuSMV-bw} \) only performs slightly better than \( \text{NuSMV} \) 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.

Fig. 14.1, 14.2, and 14.3 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 \( \text{NuSMV}, \text{NuSMV-bw}, \text{IImc}, \text{IImc-BDD-bw}, \) 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.

Fig. 14.1 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.

\(^1\)Versions submitted to HWMCC’12.
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 Fig. 14.2 is the fact that STP performs really well on both benchmarks. We suppose that this is connected to the fact that power2bit 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 [Bra11], BMC, BDDs, etc. Probably due to the same fact, the IImc curves are even more hectic on the power2bit benchmark in Fig. 14.2. 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 Fig. 14.3 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 BDD-based model checkers start to outperform their counterparts 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.

14.4 Conclusion

In this paper, we efficiently solved quantifier-free bit-vector formulas 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 [KFB12] that this can cause an exponential blowup in general. An approach for polynomially translating QF_BV to EPR exists [KFB13a] (as well as exponential ones [Emm+10]; [KKV09]), but solving the resulting formulas also suffers from the NEXPTIME-completeness of EPR [KFB13a]; [Lew80]. Building on previous complexity results [FKB13b], 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 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. 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 [BM10], 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

---

7This is not included in our results since we did not analyze it in detail yet.
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. 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 [BD02] are polynomial if a logarithmic encoding is used for the bit-widths.

---

8The benchmark sets ndist.a and ndist.b are in this class.
A DPLL ALGORITHM FOR SOLVING DQBF
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 $N_{\text{Exp}}T_{\text{ime}}$-complete, efficient ways of solving it would have many practical applications. In this paper 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.

15.1 Introduction

Dependency Quantified Boolean Formulas (DQBF), as first defined in [PR79], are obtained by adding Henkin quantifiers [Hen61] 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 $P_{\text{Space}}$-complete [Pap94], DQBF can be shown to be $N_{\text{Exp}}T_{\text{ime}}$-complete [PR79]; [PRA01]. 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 $N_{\text{Exp}}T_{\text{ime}}$-complete, e.g. partial information non-cooperative games [PRA01] or certain bit-vector logics [KFB12]; [WHM10] 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 [BB06]. 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 [BCJ12], which stayed on only the theoretical side but might yield an expansion-based DQBF solver similar to those existing for QBF [Bie04].

Effectively Propositional Logic (EPR) is another class of problems, for which the decision problem is $N_{\text{Exp}}T_{\text{ime}}$-complete. Thus exist polynomial reductions from DQBF to EPR and vice versa. Consequently, it is also possible to use EPR solvers, e.g. iProver [Kor08] 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 [DLL62], and improved variants, commonly known as CDCL solvers [MSS99], are successfully used in many industrial applications. Inspired by the success of effective techniques used in SAT-solving, 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 [DLL62] algorithm (DQDPLL) for solving DQBF.
15.2 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 \( x = \text{var}(l) \) denote its variable. A clause \( C \) is a disjunction of literals. A propositional formula \( \phi \) is in conjunctive normal form (CNF), if it is a conjunction of clauses. A DQBF \( \psi \) 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 \( \phi \) being a propositional formula (matrix) in CNF over the variables \( V := U \cup E \) and \( U = \{u_1, \ldots, u_m\}, E = \{e_1, \ldots, e_n\}, u_{ij} \in U \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\} \). 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 \( \text{dep}(e_i) := \{u_{1,1}, \ldots, u_{i,m_i}\} \) to denote its dependencies. For universal variables \( u \), we define \( \text{dep}(u) := \emptyset \). We also extend the notion of dependency to literals, defining \( \text{dep}(l) := \text{dep}((\text{var}(l))) \) for any literal \( l \). An assignment is a mapping \( \alpha : V \rightarrow \{\text{true}, \text{false}\} \) from the variables of a formula to truth values. A partial assignment is a mapping \( \beta : V \rightarrow \{\text{true}, \text{false}, \text{undef}\} \). 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 the paper \( \alpha(l) \) (resp. \( \alpha(C) \), resp. \( \alpha(F) \)) will denote the truth value a literal \( l \) (resp. a clause \( C \), resp. a formula \( F \)) takes under the assignment \( \alpha \). We extend the notation for partial assignments \( \beta \) in the same way defining \( \text{undef} \lor \text{true} := \text{true}, \text{undef} \land \text{true} := \text{undef}, \text{undef} \lor \text{false} := \text{false} \), and \( \text{undef} \land \text{false} := \text{false} \).

A propositional formula \( \phi \) in CNF is satisfiable, if all clauses in \( \phi \) are satisfied by at least one assignment \( \alpha \). We then call \( \alpha \) a model of \( \phi \). In QBF and DQBF a model can not be expressed by a single assignment. We use assignment trees [SDB06] instead, more precisely the variant of [LB11].

Given a DQBF \( \psi \), 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 \( \alpha \) for the variables in \( \psi \). In the same way a path from the root to an internal node corresponds to a partial assignment \( \beta \). Compared with 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 \( u \in \text{dep}(e) \), 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 \( \alpha_1, \alpha_2 \), it has to hold that \( \alpha_1(e) = \alpha_2(e) \), if \( \alpha_1(u) = \alpha_2(u) \forall u \in \text{dep}(e) \). 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 \( \phi \).

Actually property 1 is not needed to make sure that \( \psi \) has a solution: There is a model respecting property 1 & 2 iff 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...
QDPLL(F) {
    while(true) {
        state = checkState(beta);
        if (state == STATE_UNSAT) {
            level = analyseUNSAT();
            if (level == 0) return UNSAT;
            backtrack(level);
        } else if (state == STATE_SAT) {
            level = analyseSAT();
            if (level == 0) return SAT;
            backtrack(level);
        } else {
            literal = selectLiteral();
            beta = updateAssignment(literal);
            addDecision(literal);
        }
    }
}

Figure 15.1: Main loop of QDPLL as pseudo-code

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.

15.3 DQDPLL Architecture

In the following we assume that the reader is familiar with the design of a DPLL solver for SAT/QBF. Figure 15.1 shows the typical pseudo-code for a QBF solver based on the DPLL algorithm. In Fig. 15.2 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 formula exists iff 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.

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 ∈ 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-formula as an input since decisions on existential variables may always be “delayed”.

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-formula is only partially ordered. To make sure that our assignment tree nevertheless
DQDPLL(F) {
    while (true) {
        state = checkState(beta);
        if (state == STATE_UNSAT) {
            level = analyseUNSAT();
            if (level == 0) return UNSAT;
            backtrack(level);
        } else if (state == STATE_SAT) {
            level = analyseSAT();
            if (level == 0) return SAT;
            restoreAssignment(level);
        } else {
            literal = selectLiteral();
            skolemClause = generateSkolemClause(beta, literal);
            beta = updateAssignment(literal);
            addDecision(beta, skolemClause);
        }
    }
}

backtrack(level) {
    while (stack.Size > level) popStack();
    (beta, _) = stack.Element(level);
}

restoreAssignment(level) { (beta, _) = stack.Element(level); }

addDecision(beta, skolemClause) { pushStack(beta, skolemClause); }

Figure 15.2: Main methods of DQDPLL as pseudo-code

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 universals.

In our algorithm this happens in the addDecision method. While the QDPLL algorithm only has to save the literal that was assigned during a decision, 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. $C_{sk} = true$ 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 actually do not push the branching literal on the decision stack but instead the current assignment $\beta$. 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 $\beta$ 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 $\text{var}(l_{e_i}) = e_i$, $\text{dep}(e_i) = \{u_{i,1}, \ldots, u_{i,m_i}\}$. Then
$C_{sk} := (l_{i,1}, \ldots, l_{i,m}, l_{e_i})$, $l_{i,j} = \begin{cases} u_{i,j}, & \text{if } \beta(u_{i,j}) = \text{false} \\ \neg u_{i,j}, & \text{if } \beta(u_{i,j}) = \text{true} \end{cases}$

Since we only are allowed to branch on $e_i$ if all $u_{i,j} \in \text{dep}(e_i)$ have already been assigned, we know that $\beta(u_{i,j}) \neq \text{undef}$, 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 $u_{i,j} \in \text{dep}(e_i)$ 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 analyseConflict 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, analyseSolution returns a previous decision on a universal variable that still has to consider the second branch. Again 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 $\beta$ 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 will fulfill 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.

Apart 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. E.g. 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 paper. 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.

15.4 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 the abovementioned 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 $\beta$, so that $\exists j \in \{1, \ldots, k\} : \beta(l_j) = \text{undef}, \beta(l_i) = \text{false} \forall i \in \{1, \ldots, k\} \setminus \{j\}$. 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 $u \in \text{dep}(e)$ have been assigned yet when it gets propagated. Assigning $e$ before all $u \in \text{dep}(e)$ are assigned actually violates property 1 defined in Sect. 15.2. Nevertheless it is still sound to do so and will help pruning the search tree.

We already argued in Sect. 15.2 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 $\beta$, then the same unit propagation step is possible under all possible partial assignments $\beta'$ which can be constructed from $\beta$ by assigning all remaining variables $u \in \text{dep}(e), \beta(u) = \text{undef}$. This means that assigning a unit $e$ earlier, i.e. before all the universals 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, if not all universals in 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 $\{u \in \text{dep}(e) \mid \beta(u) = \text{undef}\}$. Using resolution and subsumption, this can be expressed by adding only one clause:

$$C_{sk} := (l_{i_1}, \ldots, l_{i_{m}}, l_{i_j}), \quad l_{i_j} = \begin{cases} u_{i,j}, & \text{if } \beta(u_{i,j}) = \text{false} \\ \neg u_{i,j}, & \text{if } \beta(u_{i,j}) = \text{true} \\ \text{false}, & \text{if } \beta(u_{i,j}) = \text{undef}, \end{cases}$$
assuming \( \text{var}(l_e) = e_i, \text{dep}(e_i) = \{u_{i,1}, \ldots, u_{i,m_i}\} \).

**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 [BCJ12]. 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 it is the case 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 \( l_e \) can be set to true if every clause containing \( \neg l_e \) is already satisfied by at least one \( l_u, \text{var}(l_u) \in \text{dep}(l_e) \) or by an existential literal \( l_{e_j}, \text{dep}(l_{e_j}) \subseteq \text{dep}(l_e) \). In this case we know that all clauses containing \( \neg l_e \) are already satisfied whenever the newly added Skolem clause propagates \( l_e \). This means \( l_e \) 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 [MSS99]. Clause learning was then also applied to QBF [GNT02]; [Let02]; [Zha02]. 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. It therefore 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 [GN02] 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 \( \text{var}(l_u) = u, \beta(l_i) \neq \text{true} \forall l_i \in C \). If \( u \not\in \bigcup_{\beta(l_i) = \text{undef}} \{ \text{dep}(l_i) \} \), \( l_u \) can be set to \( \text{false} \).

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 \( v \in \{\text{true}, \text{false}\} \). A solution for \( F \) has to satisfy \( C_{l_u=\text{true}} \) and \( C_{l_u=\text{false}} \). 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 \( C_{l_u=\text{true}} \) and \( C_{l_u=\text{false}} \). Since \( C_{l_u=\text{true}} \) is already satisfied by \( l_u \), only \( C_{l_u=\text{false}} \) needs to be considered instead of \( C \), i.e. \( l_u \) can be removed from \( C \).

**Cube Learning.** Introduced for QBF in [GNT02]; [Let02]; [ZM02], 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 is therefore 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 \( \text{true} \) 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 [Mos+01] and phase saving [PD07]. 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 [LB10a]. Note that this is a different concept. 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 [Mos+01] and phase saving [PD07] in the same way 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 universals 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 [Zha97]; [Mos+01]. 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 [Gen+03]; [LB10a]. 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 \( \text{var}(l_2) \in \text{dep}(l_1) \). Notice that in QBF this latter condition about dependency only requires to check whether \( \text{var}(l_2) \) is quantified before \( \text{var}(l_1) \) in the prefix. This can be adapted to DQBF in a straightforward way, by checking the explicit dependencies of \( \text{var}(l_1) \). 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 \( l_2 = \text{undef} \) 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 \( \beta \), provided by the backTrack method. We would like to point out that this update could be highly optimized by the implementation optimization mentioned in Sect. 15.3, 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 \( \text{lca}(n,n') \) denote the lowest common ancestor of \( n \) and \( n' \). During traversing the path from \( \text{lca}(n,n') \) to \( n' \), update the watchers of the literals assigned by the touched nodes.

15.5 Preliminary Results

We implemented a prototype of our DQDPLL algorithm as introduced in Sect. 15.3 and added all the concepts described in Sect. 15.4. Testing was rather difficult since there is no DQBF library yet nor any other DQBF solver to compare results with.

Since EPR is also \text{NE}x\text{P}T\text{ime}-complete, we used EPR formulas from the TPTP and converted those 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 [KS08, Chapter 3.3.1], inverse destructive equality reasoning [WHM10] 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 [LB10a]. As expected DepQBF was faster by several orders of magnitude since it is much more specialized while our solver has additional exponential overhead 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, as well as very small sized instances (2-6 variables) to check correctness on this subset manually.

A further way to check correctness could be translating our randomly generated DQBF to EPR and then compare our results with the results of an EPR solver on the converted benchmark as done for QBF in [SLB12].

15.6 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 regarding data structures and implementational details of our techniques 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 QBF and SAT solvers, 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 [Nie+12].

15.7 Conclusion

In this paper 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 could 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 [PR79], this paper is 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 on the topic of DQBF, and possibilities and pitfalls on the way of practically solving it.
IDQ: INSTANTIATION-BASED DQBF SOLVING.
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 $\text{NExpTime}$-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 paper, 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.

16.1 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 [PR79] and comprises the set of propositional formulas which are obtained by adding Henkin quantifiers [Hen61] 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. While QBF is $\text{PSpace}$-complete [Pap94], DQBF was shown to be $\text{NExpTime}$-complete [PRA01]; [PR79]. Aside from DQBF, many practical problems are known to be $\text{NExpTime}$-complete. This includes, e.g., partial information non-cooperative games [PRA01] or certain bit-vector logics [KFB12]; [WHM10] used in the context of Satisfiability Modulo Theories (SMT). More recently, also applications in the area of equivalence for partial implementations [Git+13a]; [Git+13b] and synthesis for fragments of linear temporal logic [Cha+13] 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 [BB06]. However, apart from our previous work on adapting DPLL for DQBF [FKB12] and a recent incomplete approach (only allowing refutation of unsatisfiable formulas) [FT14], 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 [FKB12], our previous approach did not end up being very efficient. Apart from this, formula expansion and transformations specific to QBF have been discussed in [BCJ12]; [BCJ14], which stayed only on the theoretical side but can yield an expansion-based DQBF solver similar to those existing for QBF [Bie04]. In [FT14], an expansion-based solver is also briefly mentioned. A (not publicly available) expansion-based solver was used in [Git+13b]. Further, in [BCJ12]; [BCJ14], it has been conjectured that QBF solvers based on Skolemization [Ben04] could easily be adapted for DQBF. However, the current implementation of the described QBF solver sKizzo [Ben04] 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 [FKB12] and might have a similar negative impact on the performance of the resulting solver.

Effectively Propositional Logic (EPR) is another logic which is $\text{NExpTime}$-complete [Lew80]. 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 [Koro08] being the currently most successful one, to solve DQBF given some translation from DQBF to EPR. In [SLB12], 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 [Koro]; [Kor13], which can be considered as the state-of-the-art decision procedure for EPR [Koro8]. 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 Sect. 16.2 and giving related work in Sect. 16.3, we provide the theoretical foundation in Sect. 16.4 and point out parallel features used in EPR solving. We also propose a concrete implementation of our algorithm in Sect. 16.5, and provide detailed experiments, comparing our prototype iDQ with state-of-the-art solvers on several DQBF as well as QBF benchmarks in Sect. 16.6. 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 Sect. 16.7.

### 16.2 Preliminaries

Let $V$ be a set of propositional variables. A literal $l$ is a variable $x \in V$ or its negation $\neg x$. For a given literal $l$, we write $\text{var}(l)$ to reference the corresponding variable. A clause $C$ is a disjunction of literals. A propositional formula $\phi$ is in conjunctive normal form (CNF), if it is a conjunction of clauses. Any DQBF can always be expressed as

$$\psi \equiv \bigwedge \phi \equiv \forall u_1, \ldots, u_m \exists \varepsilon_1(u_1, \ldots, u_{1,k_1}), \ldots, \varepsilon_n(u_n, \ldots, u_{n,k_n}) \phi$$

with $\bigwedge$ being the quantifier prefix and $\phi$ being a propositional formula (matrix) in CNF over the variables $V := U \cup E$ and $U = \{u_1, \ldots, u_m\}$, $E = \{\varepsilon_1, \ldots, \varepsilon_n\}$, $u_{ij} \in U$, $\forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}$. We refer to the elements of $U$ and $E$ as the universal variables and existential variables of $\psi$, 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 $\varepsilon_i$, we use $\text{dep}(\varepsilon_i) := \{u_{i,1}, \ldots, u_{i,k_i}\}$ to denote its dependencies. For universal variables $u$, we define $\text{dep}(u) := \emptyset$. We extend the notion of dependency to literals, defining $\text{dep}(l) := \text{dep}(\text{var}(l))$ for any literal $l$. Obviously, any QBF $\psi_{\text{qbf}}$ can be translated to some $\psi_{\text{dqb}}$ in the specified form by moving all universal variables to the beginning and then setting $\text{dep}(e) = \{u \in U \mid u \text{ is before } e \text{ in the quantifier prefix of } \psi_{\text{qbf}}\}$ for all existential variables.

An assignment is a (partial) mapping $\alpha : V \rightarrow \{0, 1\}$ 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 paper, $\alpha(l)$, $\alpha(C)$, or $\alpha(F)$ will denote the truth value (under the assignment $\alpha$) of a literal $l$, a clause $C$, or a formula $F$, respectively. An assignment $\alpha$ to a formula $F$ is satisfying, if and only if $\alpha(F) = 1$.

A propositional formula $\phi$ in CNF is satisfiable, if and only if all clauses in $\phi$ are satisfied by at least one assignment $\alpha$. We then call $\alpha$ a model of $\phi$. In DQBF (as well as in QBF), a model can not be expressed by a single assignment. Instead, we use Skolem functions to represent solutions of a formula. A Skolem function $f_e : \{0, 1\}^{\text{dep}(\varepsilon_i)} \rightarrow \{0, 1\}$ describes the evaluation of an existential variable $\varepsilon_i$ under a given assignment to its dependencies. Let $\phi_{\varepsilon_i}$ denote the formula obtained from $\phi$ by replacing all existential variables $\varepsilon_i$ 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 $\phi_{de}$ is satisfied for all possible assignments to the universal variables of $\psi$.

Universal expansion is defined as the process of removing a universal variable $u$ from a formula $\psi$ considering both its values separately. This can be done by removing all existential variables $e$ with $u \in \text{dep}(e)$ and introducing two new existential variables $e_{u=1}, e_{u=0}$ with $\text{dep}(e_{u=1}) = \text{dep}(e_{u=0}) = \text{dep}(e) \setminus \{u\}$. Additionally, the matrix $\phi$ is replaced by $\phi_{u=1} \land \phi_{u=0}$. With $\phi_{u=v}$, we describe the formula obtained from $\phi$ by replacing $u$ by a constant $v \in \{1, 0\}$ and all occurrences of $e$ with $u \in \text{dep}(e)$ by $e_{u=v}$. We can use universal expansion to reduce any DQBF $\psi$ 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 $\phi_{de}$ instead of $\phi_{u=1} \land \phi_{u=0}$, 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 [Kor09; Kor13] uses instantiation, applied to the natural encoding of (D)QBF into first-order logic [SLBt2]. Furthermore, for a given clause instance $C_{i_1, \ldots, i_k}$ we define $\text{ctx}(C_{i_1, \ldots, i_k}) := \{l_i \mid i = 1, \ldots, k\}$. 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

$$\forall u_1, u_2 \exists e_1(u_1), e_2(u_1, u_2), e_3(u_1, u_2) \cdot (u_1 \lor e_1) \land (u_2 \lor e_2) \land (u_2 \lor \bar{e}_3)$$

we can now write equations of clause instances such as:

$$= (e_1)_{\pi_1} \land (u_2 \lor e_2) \land (u_1 \lor \bar{e}_3) = (e_1)_{\pi_1} \land (u_2 \lor \bar{e}_3)$$

$$= (e_1)_{\pi_1} \land (e_2)_{u_2} \land (u_2 \lor \bar{e}_3)_{u_1} = (e_1)_{\pi_1} \land (e_2)_{u_2} \land (\bar{e}_3)_{u_1 \pi_2}$$

The last line is a succinct representation of the full universal expansion of the original formula and minimal in the sense 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:

$$(e_1)_{\pi_1} \land (e_2)_{u_2} \land (\bar{e}_3)_{u_1 \pi_2} = (e_1)_{\pi_1} \land (e_2)_{u_1 u_2} \land (e_2)_{u_1 u_2} \land (\bar{e}_3)_{u_1 \pi_2}$$

The last expression is maximal and of the same size as the full universal expansion of $\psi$. There is no point in further expanding the first clause instance since $u_2 \notin \text{dep}(e_1)$, i.e. $(e_1)_{\pi_1} = (e_1)_{\pi_1 \pi_2} = (e_1)_{\pi_1 u_2}$. Obviously, if a clause instance $C_{j_1, \ldots, j_k}$ is part of a formula, we can always add a more specific instance $C_{i_1, \ldots, j_k, l_{k+1}, \ldots, l_{k'}}$ 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 Sect. 16.4.

EPR, known as the Bernays-Schönfinkel class, is a NExTTime-complete fragment of first-order logic [Lew80]. 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

$$\text{ctx}(C_{i_1, \ldots, i_k}) = \{l_i \mid i = 1, \ldots, k\}$$
$p(t_1, \ldots, t_n)$ where $p$ is a predicate symbol of arity $n$ and each $t_i$ is either a (universal) variable or a constant.

In [SLB12], 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 Eqn. (16.1) the resulting EPR formula is

$$\forall u_1, u_2 \cdot (p(u_1) \lor f_{e_1}(u_1)) \land (\neg p(u_2) \lor f_{e_2}(u_1, u_2)) \land (\neg p(u_1) \lor p(u_2) \lor \neg f_{e_3}(u_1, u_2)) \land p(1) \land \neg p(0)$$

16.3 RELATED WORK

The concepts of instantiation and expansion that we defined in Sect. 16.2 are similar to the notation used in [Ben04], describing the solver sKizzo, which in particular shares similarities in the use of clause instances (c.f. symbolic representation in [Ben04]). 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 formulas without introducing additional concepts as presented in our previous work [FKB12]. 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 Sect. 16.4.

Another similar notation and related work is proposed in [JGMS13]; [Jan+12]; [JMS13]. Their solver RAReQS [Jan+12] creates propositional abstractions and uses a CEGAR approach [Cla+03] for refinement. As we will discuss in Sect. 16.4, 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. 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 $\phi$ containing $u$. Nevertheless, even the restricted expansion of universal variables in [JGMS13]; [Jan+12]; [JMS13] 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. Further, RAReQS is a QBF solver and cannot tackle DQBF formulas. 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 [WHM10]. 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 [PBM12]. 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.

16.4 IDQ ARCHITECTURE

In this section, we present the IDQ architecture. It is based on the more general Inst-Gen calculus [Kor09]; [Kor13] for EPR as used in tProver [Kor08], but reduced to the more specific case of DQBF. Instead of dealing with predicates, we use the notion of clause instances as
We now create the initial set of clause instances, using the unique minimal instantiation that

\[ F' := \text{initInstantiation}(F) \]

while true do

\[ F'' = \text{propositionalAbstraction}(F') \]

(state, assignment) = checkSat(F''

if (state == unsat) then return unsat

if isValid(assignment, F, F') then return sat

\[ F' = \text{refineInstantiation}(assignment, F, F') \]

Figure 16.1: Pseudo-code of a CEGAR loop as used in the Inst-Gen procedure [Koro8]; [Koro9]; [Kor13].

introduced in Sect. 16.2. The Inst-Gen architecture is based on the CEGAR paradigm [Cla+03]
and the pseudo-code is given in Fig. 16.1.

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 resulting 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 w.r.t. 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 [SLB12], 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 Sect. 16.2. Consider the
following example:

\[ \psi = \forall u_1, u_2 \exists e_1 (u_1, u_2), e_2(u_2) \cdot (u_1 \lor e_1) \land (u_1 \lor \overline{e_1}) \land (\overline{u_1} \lor u_2 \lor e_1 \lor e_2) \]

We now create the initial set of clause instances, using the unique minimal instantiation that
removes all universal variables from the clauses:

\[ (e_1)\overline{u_1} \land (e_1)\overline{u_1} \land (e_1 \lor e_2)u_2 \]

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 \( \mathcal{P} \) denoting the power-set, we use a function \( m : E \times \mathcal{P} \{ l \mid var(l) \in U \} \rightarrow V' \) 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:

\[
m(e_1, ctx(C_1)) = m(e_2, ctx(C_2)) \quad \text{if and only if} \quad e_1 = e_2, \ \{ l_1 \in ctx(C_1) \mid \var(l_1) \in dep(e_1) \} = \{ l_2 \in ctx(C_2) \mid \var(l_2) \in dep(e_2) \}
\]

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:

\[
(x_1) \land (\overline{x}_1) \land (x_2 \lor x_3)
\]

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:

\[
\psi = \forall u_1, u_2 \exists e_1(u_1, u_2), e_2(u_2) \cdot (u_1 \lor e_1) \land (\overline{\pi}_2 \lor \overline{\tau}_1 \lor e_2)
\]

Again, we create the initial set of clause instances using the unique minimal instantiation that removes all universal variables from the clauses:

\[
(e_1)_{\pi_1} \land (\overline{\pi}_1 \lor e_2)_{u_2}
\]

The propositional overapproximation now looks as follows:

\[
(x_1) \land (\overline{x}_2 \lor x_3)
\]

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 \( \alpha \), e.g., \( \alpha = \{ x_1 \rightarrow 1, x_2 \rightarrow 0, x_3 \rightarrow 0 \} \).

We now check, whether \( \alpha \) 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 [Kor08], [Kor09], [Kor13]. 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, \( \alpha \) 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:

\[
(e_1)_{\pi_1} \land (e_1)_{\pi_1 u_2} \land (\overline{\pi}_1 \lor e_2)_{u_2} \land (\overline{\tau}_1 \lor e_2)_{\pi_1 u_2}
\]

The propositional abstraction is given by:

\[
(x_1) \land (x_2) \land (\overline{x}_3 \lor x_4) \land (\overline{x}_2 \lor x_4)
\]
Note that $c_2$ is mapped to the same variable $x_4$ in both clause instances although it appears in a different instantiation context. This is due to $u_1 \notin \text{dep}(c_2)$, which implies that $(e_2)_{u_2} = (e_2)_{u_2}$. Again, this propositional formula is satisfiable and the SAT solver could return a satisfying assignment $\alpha = \{x_1 \rightarrow 1, x_2 \rightarrow 1, x_3 \rightarrow 0, x_4 \rightarrow 1\}$. 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 $(e_1)_{u_1} \land (e_1)_{u_2}$ instead of $(e_1)_{u_1} \land (e_1)_{u_2}$. 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.

### 16.5 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 $d <\text{int32}> \ [<\text{int32}> \ldots <\text{int32}>] 0$ 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 Sect. 16.6.

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. E.g., see instance (I) in Fig. 16.2a, where the first mask is 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 [Bie08].

**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 $l$ is part of the context of the current instance; see (I) in Fig. 16.2a.

**Propositional Abstraction.** Each occurrence of each 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
Universal variables: $u_1, u_2, u_3$
Existential variable: $e(u_1, u_3)$
Input clause: $(u_2 \lor \pi_3 \lor e)$
(I) Initial instance: $(e)_{\pi u_3} \rightarrow 011 / 001$
(C) $e$'s concrete context: $u_3 \rightarrow D \& I_1 = 001 / C_1 \& I_2 = 001$
(G) $e$'s ground context: $\pi_1 u_3 \rightarrow D = 101 / I_2 = 001$

(a) Dependencies and contexts.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$e$: $(e \lor \ldots)_{\pi u_2 u_3} \rightarrow 111 / 011$</td>
</tr>
<tr>
<td>B</td>
<td>$\pi$: $(\pi \lor \ldots)_{\pi u_3} \rightarrow 011 / 001$</td>
</tr>
<tr>
<td>E</td>
<td>$e$'s concrete context: $\pi_1 u_3 \rightarrow 101 / 001$</td>
</tr>
<tr>
<td>F</td>
<td>$\pi$'s concrete context: $u_3 \rightarrow 001 / 001$</td>
</tr>
<tr>
<td>N</td>
<td>New instance: $(\pi \lor \ldots)_{\pi_1 \pi u_3} \rightarrow B_1</td>
</tr>
</tbody>
</table>

Overlapping contexts?

Redundant?

(b) Inference step and redundancy check.

Figure 16.2: Examples of using bit-vector representation for various calculations in iDQ.

illustrated in Fig. 16.2a as the context mask ($C_1$) being calculated from (D) and (I1), and the value mask ($C_2$) from ($C_1$) 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 [Kor09; Kor13] 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. Fig. 16.2a 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 Fig. 16.2b. 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 tProver’s default heuristic, based on the lexicographical combination of orders defined on given numerical/Boolean parameters. Similar to tProver’s notation [Kor13], 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. I.e., 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 [Mos+01] of propositional variables used in the propositional abstraction. tDQ 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 tDQ.

However, a practical redundancy check might be more complicated, e.g., tProver employs mismatching constraints [Kor13]. With tDQ, a practical check can be obtained more easily. tDQ 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 Fig. 16.2b. Importantly, (2) requires to check if c’s context is a superset of d’s contexts.

### 16.6 Experimental Results

In this section, we report experiments with our solver. The source code, benchmarks, and log files are available at http://fmv.jku.at/idq. We tested tDQ with two types of heuristics as proposed in Sect. 16.5. tDQ and tDQvsids 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 [FT14] can reason only with unsat formulas), we decided to also compare tDQ against tProver (v0.8.1).

We also tested tDQ on QBF benchmarks, by exploiting the fact that QBF is a real fragment of DQBF. By doing so, we could compare tDQ not only against tProver, but also against genuine QBF solvers, like the QDPLL-based DerQBF [LB10a] (v3.0), the Skolemization-based skizzo [Ben04] (v0.8.2), the CEGAR-based RARsQS [Jan+12] (v1.1), and the expansion-based Nenofex [LB08] (v1.0). For the sake of fair comparison, we did not run any preprocessor.

---

1Setup: Vienna Scientific Cluster (VSC-2), AMD Opteron Magny Cours 6132HE CPUs, 2.2 GHz cores, 900 seconds time limit, 3800 MB memory limit.
**DQBF benchmarks.** We used the only publicly available DQBF benchmarks by Finkbeiner and Tentrup [FT14]. 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 [DH12], and also the circuit family `pec_xor` from [Git+13b] 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 [SLB12], which can be easily adapted to DQBF.

Table 16.1 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. iDQ_pecs 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.

<table>
<thead>
<tr>
<th>bitcell_16,2</th>
<th>TO</th>
<th>time</th>
<th>bitcell_16,4</th>
<th>TO</th>
<th>time</th>
<th>bitcell_16,6</th>
<th>TO</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQBF2QBF</td>
<td>98 (0/98)</td>
<td>2 18.6</td>
<td>98 (0/98)</td>
<td>2 18.8</td>
<td>97 (0/97)</td>
<td>3 27.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ</td>
<td>88 (2/86)</td>
<td>12 128.1</td>
<td>52 (0/52)</td>
<td>48 488.9</td>
<td>22 (0/22)</td>
<td>78 735.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ_pecs</td>
<td>97 (2/95)</td>
<td>3 39.2</td>
<td>75 (0/75)</td>
<td>25 255.9</td>
<td>36 (0/36)</td>
<td>64 592.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iProver</td>
<td>82 (0/82)</td>
<td>18 248.6</td>
<td>34 (0/34)</td>
<td>66 684.5</td>
<td>7 (0/7)</td>
<td>93 851.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lookahead_16,2</td>
<td>TO</td>
<td>time</td>
<td>lookahead_16,4</td>
<td>TO</td>
<td>time</td>
<td>lookahead_16,6</td>
<td>TO</td>
<td>time</td>
</tr>
<tr>
<td>DQBF2QBF</td>
<td>97 (0/97)</td>
<td>3 27.7</td>
<td>97 (0/97)</td>
<td>3 27.7</td>
<td>96 (0/96)</td>
<td>4 36.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ</td>
<td>98 (3/95)</td>
<td>2 30.4</td>
<td>88 (0/88)</td>
<td>12 118.9</td>
<td>69 (0/69)</td>
<td>31 342.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ_pecs</td>
<td>93 (2/91)</td>
<td>7 68.1</td>
<td>62 (0/62)</td>
<td>38 383.0</td>
<td>20 (0/20)</td>
<td>80 729.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iProver</td>
<td>67 (0/67)</td>
<td>33 351.8</td>
<td>32 (0/32)</td>
<td>68 656.3</td>
<td>6 (0/6)</td>
<td>94 862.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>adder_3,2_4</th>
<th>TO</th>
<th>time</th>
<th>adder_3,4</th>
<th>TO</th>
<th>time</th>
<th>adder_3,6</th>
<th>TO</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQBF2QBF</td>
<td>94 (0/94)</td>
<td>6 54.8</td>
<td>89 (0/89)</td>
<td>11 90.8</td>
<td>74 (0/74)</td>
<td>26 234.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ</td>
<td>82 (1/81)</td>
<td>18 248.6</td>
<td>58 (0/58)</td>
<td>42 440.2</td>
<td>11 (0/11)</td>
<td>89 841.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ_pecs</td>
<td>43 (0/43)</td>
<td>57 546.5</td>
<td>21 (0/21)</td>
<td>79 734.0</td>
<td>6 (0/6)</td>
<td>94 863.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iProver</td>
<td>86 (1/85)</td>
<td>14 221.6</td>
<td>54 (0/54)</td>
<td>46 538.0</td>
<td>5 (0/5)</td>
<td>95 876.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pec_xor2</th>
<th>TO</th>
<th>time</th>
<th>pec_xor3</th>
<th>TO</th>
<th>time</th>
<th>pec_xor4</th>
<th>TO</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQBF2QBF</td>
<td>49 (0/49)</td>
<td>51 459.4</td>
<td>77 (0/77)</td>
<td>23 207.5</td>
<td>99 (0/99)</td>
<td>1 10.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ</td>
<td>100 (51/49)</td>
<td>.5</td>
<td>100 (25/77)</td>
<td>.7</td>
<td>100 (1/99)</td>
<td>3.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iDQ_pecs</td>
<td>100 (51/49)</td>
<td>.5</td>
<td>100 (25/77)</td>
<td>.6</td>
<td>100 (1/99)</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iProver</td>
<td>100 (51/49)</td>
<td>.5</td>
<td>100 (25/77)</td>
<td>.9</td>
<td>100 (1/99)</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 16.1: Results for DQBF PEC benchmarks

**QBF benchmarks.** We used QBF Gallery 2013 benchmarks, from which we selected instances whose size do not exceed 2 megabytes. In some cases, we randomly selected instances from the resulting sets. Table 16.2 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 tDQ and tPROVER on most benchmarks, although sKizzi and Nenofex terminate with memory out quite frequently. On some instances, tPROVER and Nenofex crash. tDQ 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#sat/uns</th>
<th>TO/MO</th>
<th>time</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefQBF</td>
<td>89 (19/70)</td>
<td>11/0</td>
<td>130.7</td>
<td></td>
</tr>
<tr>
<td>RAreQS</td>
<td>94 (17/77)</td>
<td>4/2</td>
<td>49.1</td>
<td></td>
</tr>
<tr>
<td>Nenofex</td>
<td>95 (19/76)</td>
<td></td>
<td>19.7</td>
<td>5</td>
</tr>
<tr>
<td>sKizzi</td>
<td>51 (11/40)</td>
<td>34/15</td>
<td>380.9</td>
<td></td>
</tr>
<tr>
<td>tDQ</td>
<td>95 (14/81)</td>
<td>5/0</td>
<td>81.9</td>
<td></td>
</tr>
<tr>
<td>tDQvisids</td>
<td>95 (14/81)</td>
<td>5/0</td>
<td>80.2</td>
<td></td>
</tr>
<tr>
<td>tPROVER</td>
<td>91 (14/77)</td>
<td>9/0</td>
<td>90.9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#sat/uns</th>
<th>TO/MO</th>
<th>time</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefQBF</td>
<td>59 (12/47)</td>
<td>103/0</td>
<td>586.1</td>
<td></td>
</tr>
<tr>
<td>RAreQS</td>
<td>63 (12/51)</td>
<td>99/0</td>
<td>572.0</td>
<td></td>
</tr>
<tr>
<td>Nenofex</td>
<td>26 (12/14)</td>
<td>0/136</td>
<td>487.9</td>
<td></td>
</tr>
<tr>
<td>sKizzi</td>
<td>48 (12/36)</td>
<td>79/35</td>
<td>526.8</td>
<td></td>
</tr>
<tr>
<td>tDQ</td>
<td>44 (12/32)</td>
<td>118/0</td>
<td>665.0</td>
<td></td>
</tr>
<tr>
<td>tDQvisids</td>
<td>42 (12/30)</td>
<td>120/0</td>
<td>666.8</td>
<td></td>
</tr>
<tr>
<td>tPROVER</td>
<td>26 (12/14)</td>
<td>135/0</td>
<td>762.8</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#sat/uns</th>
<th>TO/MO</th>
<th>time</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefQBF</td>
<td>50 (35/15)</td>
<td>50/0</td>
<td>457.3</td>
<td></td>
</tr>
<tr>
<td>RAreQS</td>
<td>33 (20/13)</td>
<td>0/67</td>
<td>248.2</td>
<td></td>
</tr>
<tr>
<td>Nenofex</td>
<td>18 (9/9)</td>
<td>0/82</td>
<td>564.7</td>
<td></td>
</tr>
<tr>
<td>sKizzi</td>
<td>18 (9/9)</td>
<td>43/39</td>
<td>614.8</td>
<td></td>
</tr>
<tr>
<td>tDQ</td>
<td>20 (8/12)</td>
<td>80/0</td>
<td>724.7</td>
<td></td>
</tr>
<tr>
<td>tDQvisids</td>
<td>27 (17/10)</td>
<td>73/0</td>
<td>658.7</td>
<td></td>
</tr>
<tr>
<td>tPROVER</td>
<td>19 (10/9)</td>
<td>76/5</td>
<td>725.8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#sat/uns</th>
<th>TO/MO</th>
<th>time</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefQBF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAreQS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nenofex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sKizzi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tDQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tDQvisids</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tPROVER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 16.2: Results for QBF Gallery 2013 benchmarks

16.7 Conclusion

In this paper, 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 tPROVER 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 tPROVER 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 heuristical level, is part of future
work. Specialized preprocessing techniques, e.g., related to those applied in sKizzo [Ben04] or for general QBF solvers [BLS11], as well as removing dependencies of existential variables by analyzing the propositional matrix [LB10a], 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 [BB13]; [ELW13]; [HSB14]; [JGMS13]; [JMS13]; [Nie+12]; [SS13]; [SS14]. 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 Sect. 16.2. 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 [JMS13]. 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.


Bibliography


