Noname manuscript No.
(will be inserted by the editor)

Complexity of Fixed-Size Bit-Vector Logics

Gergely Kovásznai · Andreas Fröhlich · Armin Biere

Received: date / Accepted: date

Abstract Bit-precise reasoning is important for many practical applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. From the theoretical point of view, only few results on the complexity of fixed-size bit-vector logics have been published. Some of these results only hold if unary encoding on the bit-width of bit-vectors is used.

In our previous work [41], we have already shown that binary encoding adds more expressiveness to various fixed-size bit-vector logics with and without quantification. In a follow-up work [30], we then gave additional complexity results for several fragments of the quantifier-free case.

In this paper, we revisit our complexity results from [30, 41] and go into more detail when specifying the underlying logics and presenting the proofs. We give a better insight in where the additional expressiveness of binary encoding comes from. In order to do this, we bring together our previous work and propose several new complexity results for new fragments and extensions of earlier bit-vector logics. We also discuss the expressiveness of various bit-vector operations in more detail. Altogether, we provide the currently most complete overview on the complexity of common bit-vector logics.

This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE)

Gergely Kovásznai
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
E-mail: gergely.kovasznai@jku.at

Andreas Fröhlich
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
E-mail: andreas.froehlich@jku.at

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
E-mail: biere@jku.at

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 [9], MathSAT [12], STP [31], Z3 [22], and Yices [25].

The theory of *fixed-size bit-vector logics* is investigated in several scientific works [4,5,13,21,27], and even concrete formats for specifying such bit-vector problems exist, e.g., the SMT-LIB format [3] or the BTOR format [10]. Working with *non-fixed-size* bit-vectors has been considered for instance in [1,5], and more recently in [55,56], but is not further discussed in this paper. Most industrial applications (and examples in the SMT-LIB [1]) have fixed bit-width.

We investigate the *complexity* of solving *fixed-size bit-vector formulas*. Some papers propose such complexity results, e.g., in [4], the authors consider the common quantifier-free bit-vector logic and give an argument for NP-*hardness* of its satisfiability problem. In [13], a sublogic of the previous one is claimed to be NP-*complete*. Interestingly, in [14], 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 [59,60], the *quantified* case is addressed, and the satisfiability problem for this logic with uninterpreted functions is proved to be NExpTime-*complete*. However, the proof, similarly to the decision procedure in [14], only holds if we assume unary encoded bit-widths.

Parts of our paper already appeared as previous work [30,41]. 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 [19, Page 239, Footnote 3]). In practice, the more natural and exponentially more succinct *logarithmic* encoding is used, such as in the SMT-LIB [3] or the BTOR [10] format. We investigate how complexity varies if we consider either a unary or a binary encoding. Note that binary encoding, throughout the whole paper, can be replaced with any other logarithmic encoding.

The present paper extends our previous work in several ways. After giving a motivation for the use of binary encoded bit-vector logics in Section 2, we specify various fixed-size bit-vector logics in detail (Section 3). While our previous papers were referring to the common syntax and semantics used in other works, e.g., [4,5,10,13,21,27], but was never fully specified from the theoretical point of view, we now want to provide self-contained descriptions for the bit-vector logics that we are considering. Therefore, we introduce syntax and semantics for fixed-size bit-vector logics containing all common bit-vector operations as used in the SMT-LIB format.

After these preliminary definitions, we give a short overview of the existing complexity results for bit-vector logics with unary encoding in Section 4. We then introduce the concept of scalar-boundedness for bit-vector logics with

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

binary encoding in Section 5 and give improved versions of our complexity proofs for quantifier-free bit-vector logics in Section 6. Although our previous proofs from [30,41] are still valid, we modified and restructured our work to present those proofs in a clearer, easier-to-read, way. In Section 7, we look at the expressiveness of various bit-vector operations and analyze whether they can be used to extend some of the previously defined fragments or to give an alternative characterization of a given class.

We then revisit the quantified case in Section 8 and give new complexity results for fragments with restrictions on operations and the bit-widths of universal variables. Also, we provide a new complexity result for quantifier-free logics extended with non-recursive macros, which are allowed, for example, in the SMT-LIB format. Finally, we discuss practical considerations of our results in Section 9. A brief overview of related work is presented in [29,42]. We then explain how our theoretic contributions can help to improve practical SMT solving.

The Appendix contains examples that make some definitions and proofs easier to understand.

2 Motivation

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

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

To illustrate this effect on a practical example, consider the following bit-vector formula in SMT-LIB syntax [3]:

```
(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 [41], 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 [30,41], 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 [29,42] 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.

3 Preliminaries

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

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

3.1 SAT, QBF, and DQBF

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

The class of *Quantified Boolean Formulas* (QBF) is obtained by adding quantifiers to Boolean formulas. Each QBF ψ can be written in prenex normal form, i.e., as a closed formula $Q.\phi$ where Q is a *quantifier prefix* $\exists V_0 \forall V_1 \exists V_2 \forall V_3 \dots \forall V_{m-1} \exists V_m$, the V_is are pairwise disjoint sets of variables, and ϕ is a Boolean formula, which is called the *matrix* of ψ. A variable $v \in V_i$ depends on a variable $v' \in V_j$ iff $i > j$. This defines a total order on the

variables of ψ. 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 [48,57].

Instead of using totally ordered quantifiers, it is also possible to extend Boolean formulas with *Henkin quantifiers* [34]. 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 [50]. Again, a DQBF can always be expressed in prenex normal form, i.e., as a closed formula $Q'.\phi$, where Q' is a *quantifier prefix*

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

where each $u_{i,j}$ is a universally quantified variable, $m_i \in \mathbb{N}$, and the *matrix* ϕ is a Boolean formula. In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given and not implicitly defined by the order of the prefix (in contrast to QBF). The more general quantifier order makes DQBF more powerful than QBF and allows more succinct encodings. A DQBF is satisfiable 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 NEXPTIME-complete [49,50]. Although we did not formally specify the dependencies of universal variables, this can be done by the use of Herbrand functions [2].

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 NEXPTIME-complete logic used in reductions in the context of unary encoded bit-vector logics [59] is *Effectively Propositional Logic* (EPR) [45]. However, due to its simplicity, we consider DQBF to be a better choice for our purposes.

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 [55]:

A combinatorial circuit with n_i inputs and n_o outputs is a finite acyclic directed graph with exactly n_i vertices of in-degree zero and n_o vertices of out-degree zero. All vertices of a non-zero in-degree have a logical function assigned to them and are called gates. All vertices of in-degree one represent a NOT-gate and vertices of greater in-degrees are either AND- or OR-gates.

Given boolean values for the inputs, each gate can be evaluated in the natural way according to the logical function it represents. As already noted in the introduction, this kind of representation of a bit-vector formula is created during bit-blasting. For every combinatorial circuit, a corresponding set of n_o SAT formulas with n_i variables can be constructed naturally.

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

All the state variables are assumed to be provided with initial values stored in the flip-flops before the first clock cycle. The input variables need to be provided values from outside the system at every clock cycle and the output variables produce a new output at every clock cycle. A sequential circuit can be used to recognize languages. A word $w \in (\{0,1\}^{n_i})^+$ is said to be accepted by a sequential circuit SC with one output o, iff 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 [51,52,54].

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 [1,5,55,56], 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 ith bit of d is denoted by $d[i]$, where $i \in \mathbb{N}$ and $i < n$. Using vector notation, d is written as $\big(d[n-1], \ldots, d[1], d[0]\big)$, i.e., the most significant bit standing on the left-hand side and the least significant bit on the right-hand side. Sometimes we omit parentheses and commas.

Syntax and semantics of fixed-size bit-vector logics do not differ much in the literature [4,5,13,21,27]. Concrete formats for specifying bit-vector problems also exist, e.g., the SMT-LIB format [3] or the BTOR format [10]. 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.

3.3.1 Syntax

The main objective of this section is to define *bit-vector formulas*. As it turns out in Definition 2 and 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 1 of a *signature* captures these rules by providing three properties for each operator: (1) An operator is given an *arity*, which is a pair of numbers that specify the number of bit-vector operands and the number of scalar operands, respectively. For instance, the arithmetic operator *addition* has 2 bit-vector and 0 scalar operands, while *extraction* has 1 bit-vector and 2 scalar operands. (2) Since there usually exist restrictions on what kind of operands are legal to use with an operator, a signature has to specify a *condition* on the bit-widths and scalar values of operands. For instance, the operands of *addition* must be of the same bit-width; the scalar operands i, j of *extraction* must be less than the bit-width of the bit-vector operand and $i \geqslant j$. (3) A *bit-width* of the resulting bit-vector is assigned to each legal combination of bit-widths and scalar values of operands.

Definition 1 (Signature) A signature for an operator set Op is defined as a set $\Sigma_{Op} := \{\langle arity_o, cond_o, wid_o\rangle \mid o \in Op\}$, where

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

Table 1 shows the set of the most common operators provided by the SMT-LIB format [3] and the literature [4,5,13,21,27], 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 **Op** denote the *common operator set* given in Table 1. **Op** includes all bit-vector operators used in the SMT-LIB providing a collection of the most common bit-vector operators in software and hardware verification; other frameworks, like Boolector and Z3, provide additional useful operators, e.g., reduction operators and overflow operators. Let $\mathbf{\Sigma_{Op}}$ denote the *common signature* for **Op**. Note that Table 1 specifies some of the syntactic properties provided by $\mathbf{\Sigma_{Op}}$ in an implicit way: the arity is completely, the condition is partly implicit.

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

continued on next page

	continued from previous page			
signed division:	$bvsdiv\left(t_1{}^{[n]}, t_2{}^{[n]}\right)$		n	
s. remainder with rounding to 0:	$bvsrem\left(t_1{}^{[n]}, t_2{}^{[n]}\right)$		n	
s. remainder with rounding to $-\infty$:	$bvsmod\left(t_1{}^{[n]}, t_2{}^{[n]}\right)$		n	

Table 1 Syntax (signature) for common bit-vector operators

The simplest bit-vector expressions, or *terms*, are the variables and constants, as Definition 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 2 (Term) A bit-vector term t of bit-width $n \in \mathbb{N}^+$ is denoted by $t^{[n]}$. A term over a signature Σ_{Op} is defined inductively as follows:

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

Let us emphasize that, in a term, bit-widths are specified explicitly only for constants, variables, and uninterpreted functions. In all other cases, the bit-width is implicit, i.e., it can be derived from the bit-widths of the operands of operations. In the following, we may omit explicit bit-widths and parentheses if they can be concluded from the context.

Definition 3 (Formula) A *bit-vector formula* is an expression $Q.t^{[1]}$, where $t^{[1]}$ is a bit-vector term, Q is a *quantifier prefix* $Q_0 x_0{}^{[n_0]} Q_1 x_1{}^{[n_1]} \ldots Q_k x_k{}^{[n_k]}$, each $Q_i \in \{\forall, \exists\}$, and each $x_i{}^{[n_i]}$ is a bit-vector variable. We call t the *matrix* of the formula.

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

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 [4,11,13,21,27]. 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 8, the same separation of a Boolean level and a bit-vector level can be made in any bit-vector formula over $\mathbf{\Sigma_{Op}}$, assuming the common interpretation of $\mathbf{\Sigma_{Op}}$, defined in Section 3.3.2.

3.3.2 Semantics

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

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

In order to evaluate a term or formula, it is first necessary to interpret all the operators we use (Definition 4), and then to assign domain elements to free variables and to interpret uninterpreted functions (Definition 5).

Definition 4 (Interpretation) An interpretation of a signature Σ_{Op} is defined as a set $\widehat{Op}$ of functions, consisting of an $\hat{o}$ for each $o \in Op$, such that

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

where

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

Let $\widehat{\mathbf{Op}}$ denote the common interpretation of $\mathbf{\Sigma_{Op}}$, detailed in Table 2, based on [13,16,27] and the SMT-LIB. Note that Table 2 uses a notation that is introduced by the following definitions.

Definition 5 (Model) $M := \langle \alpha, \widehat{F} \rangle$ is a model for a formula Φ where

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

To facilitate the presentation, similar to [13,27], we define an auxiliary bijective meta-function $nat_n : D_n \mapsto [0, 2^n - 1]$. Given a bit-vector $d \in D_n$, $nat_n(d) := \sum_{i=0}^{n-1} 2^i d[i]$. We also introduce the inverse meta-function $bv_n := nat_n^{-1}$.

Definition 6 (Evaluation) Given a signature Σ_{Op}, a formula Φ over Σ_{Op}, an interpretation $\widehat{Op}$ of Σ_{Op}, and a model $M := \langle \alpha, \widehat{F} \rangle$ for Φ, Φ can be evaluated to either 0 or 1, by using the inductive definition of the evaluation function $[\![\cdot]\!]_M^{\widehat{Op}}$, as follows:

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

As mentioned before, the common interpretation $\widehat{\mathbf{Op}}$ is given in Table 2. In the table, we omit the interpretation and the model for evaluation. Furthermore, we use two abbreviations:

$$msb(t^{[n]}) := [\![t]\!][n-1]$$

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

bvnot:	$[\![\sim t^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i (\neg [\![t]\!][i])\right)$
bvand:	$[\![t_1{}^{[n]} \ \& \ t_2{}^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i ([\![t_1]\!][i] \wedge [\![t_2]\!][i])\right)$
bvor:	$[\![t_1{}^{[n]} \mid t_2{}^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i ([\![t_1]\!][i] \vee [\![t_2]\!][i])\right)$
bvxor:	$[\![t_1{}^{[n]} \oplus t_2{}^{[n]}]\!] := bv_n\left(\sum_{i=0}^{n-1} 2^i (\neg [\![t_1]\!][i] \Leftrightarrow [\![t_2]\!][i])\right)$
bvnand:	$[\![bvnand(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_1{}^{[n]} \ \& \ t_2{}^{[n]})]\!]$
bvnor:	$[\![bvnor(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_1{}^{[n]} \mid t_2{}^{[n]})]\!]$
bvxnor:	$[\![bvxnor(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_1{}^{[n]} \oplus t_2{}^{[n]})]\!]$
ite:	$[\![ite(t_1{}^{[1]}, t_2{}^{[n]}, t_3{}^{[n]})]\!] := \begin{cases} [\![t_2]\!] & \text{if } [\![t_1]\!] \\ [\![t_3]\!] & \text{otherwise} \end{cases}$
bvcomp:	$[\![t_1{}^{[n]} = t_2{}^{[n]}]\!] := bv_1(nat_n([\![t_1]\!]) = nat_n([\![t_2]\!]))$
bvult:	$[\![t_1{}^{[n]} <_{\mathbf{u}} t_2{}^{[n]}]\!] := bv_1(nat_n([\![t_1]\!]) < nat_n([\![t_2]\!]))$
bvule:	$[\![bvule(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![\sim (t_2 <_{\mathbf{u}} t_1)]\!]$
bvugt:	$[\![bvugt(t_1{}^{[n]}, t_2{}^{[n]})]\!] := [\![t_2 <_{\mathbf{u}} t_1]\!]$

continued on next page

continued from previous page

bvuge:	$[\![bvuge\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; [\![bvule\,(t_2, t_1)]\!]$
bvslt:	$[\![bvslt\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; bv_1 \left(\begin{array}{l} (msb\,(t_1) \wedge \neg msb\,(t_2)) \vee \\ ((msb\,(t_1) \Leftrightarrow msb\,(t_2)) \wedge [\![t_1 <_{\mathbf{u}} t_2]\!]) \end{array} \right)$
bvsle:	$[\![bvsle\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; [\![\sim bvslt\,(t_2, t_1)]\!]$
bvsgt:	$[\![bvsgt\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; [\![bvslt\,(t_2, t_1)]\!]$
bvsge:	$[\![bvsge\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; [\![bvsle\,(t_2, t_1)]\!]$
bvshl:	$[\![t_1{}^{[n]} \ll t_2{}^{[n]}]\!] \;:=\; bv_n\,(nat_n\,([\![t_1]\!]) \cdot 2^k \mod 2^n)$ where $k := nat_n\,([\![t_2]\!])$
bvlshr:	$[\![t_1{}^{[n]} \gg_{\mathbf{u}} t_2{}^{[n]}]\!] \;:=\; bv_n\,(\lfloor nat_n\,([\![t_1]\!])\,/2^k \rfloor)$ where $k := nat_n\,([\![t_2]\!])$
bvashr:	$[\![t_1{}^{[n]} \gg_{\mathbf{s}} t_2{}^{[n]}]\!] \;:=\; \begin{cases} [\![\sim(\sim t_1 \gg_{\mathbf{u}} t_2)]\!] & \text{if } msb\,(t_1) \\ [\![t_1 \gg_{\mathbf{u}} t_2]\!] & \text{otherwise} \end{cases}$
extract:	$[\![t^{[n]}\,[i:j]]\!] \;:=\; bv_{i-j+1}\,(\lfloor nat_n\,([\![t]\!])\,/2^j \rfloor \mod 2^i)$
concat:	$[\![t_1{}^{[m]} \circ t_2{}^{[n]}]\!] \;:=\; bv_{m+n}\,(2^n nat_m\,([\![t_1]\!]) + nat_n\,([\![t_2]\!]))$
zero_extend:	$[\![ext_{\mathbf{u}}\,(t^{[n]}, i)]\!] \;:=\; bv_{n+i}\,(nat_n\,([\![t]\!]))$
sign_extend:	$[\![sign\_extend\,(t^{[n]}, i)]\!] \;:=\; \begin{cases} bv_{n+i}\,(2^{n+i} - 2^n + nat_n\,([\![t]\!])) & \text{if } msb\,(t) \\ [\![ext_{\mathbf{u}}\,(t^{[n]}, i)]\!] & \text{otherwise} \end{cases}$
rotate_left:	$[\![rotate\_left\,(t^{[n]}, i)]\!] \;:=\; \begin{cases} [\![t]\!] & \text{if } n{=}1 \vee i{=}0 \\ [\![t\,[n{-}i{-}1:0] \circ t\,[n{-}1:n{-}i]]\!] & \text{otherwise} \end{cases}$
rotate_right:	$[\![rotate\_right\,(t^{[n]}, i)]\!] \;:=\; \begin{cases} [\![t]\!] & \text{if } n = 1 \vee i = 0 \\ [\![t\,[i-1:0] \circ t\,[n-1:i]]\!] & \text{otherwise} \end{cases}$
repeat:	$[\![repeat\,(t^{[n]}, i)]\!] \;:=\; \begin{cases} [\![t]\!] & \text{if } i = 1 \\ [\![t \circ repeat\,(t, i-1)]\!] & \text{otherwise} \end{cases}$
bvneg:	$[\![-t^{[n]}]\!] \;:=\; bv_n\,(2^n - nat_n\,([\![t]\!]))$
bvadd:	$[\![t_1{}^{[n]} + t_2{}^{[n]}]\!] \;:=\; bv_n\,(nat_n\,([\![t_1]\!]) + nat_n\,([\![t_2]\!]) \mod 2^n)$
bvsub:	$[\![t_1{}^{[n]} - t_2{}^{[n]}]\!] \;:=\; [\![t_1 + (-t_2)]\!]$
bvmul:	$[\![t_1{}^{[n]} \cdot t_2{}^{[n]}]\!] \;:=\; bv_n\,(nat_n\,([\![t_1]\!]) \cdot nat_n\,([\![t_2]\!]) \mod 2^n)$
bvudiv:	$[\![t_1{}^{[n]} /_{\mathbf{u}} t_2{}^{[n]}]\!] \;:=\; bv_n\,(\lfloor nat_n\,([\![t_1]\!])\,/nat_n\,([\![t_2]\!]) \rfloor)$
bvurem:	$[\![bvurem\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; [\![t_1 - (t_1 /_{\mathbf{u}} t_2) \cdot t_2]\!]$
bvsdiv:	$[\![bvsdiv\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; \begin{cases} [\![abs(t_1) /_{\mathbf{u}} abs(t_2)]\!] & \text{if } msb\,(t_1) = msb\,(t_2) \\ [\![-(abs(t_1) /_{\mathbf{u}} abs(t_2))]\!] & \text{otherwise} \end{cases}$
bvsrem:	$[\![bvsrem\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; \begin{cases} [\![-bvurem\,(abs(t_1), abs(t_2))]\!] & \text{if } msb\,(t_1) \\ [\![bvurem\,(abs(t_1), abs(t_2))]\!] & \text{otherwise} \end{cases}$
bvsmod:	$[\![bvsmod\,(t_1{}^{[n]}, t_2{}^{[n]})]\!] \;:=\; \begin{cases} [\![bvsrem\,(t_1, t_2)]\!] & \text{if } [\![bvsrem\,(t_1, t_2)]\!]{=}0 \\ & \quad \vee msb\,(t_1){=}msb\,(t_2) \\ [\![bvsrem\,(t_1, t_2) + t_2]\!] & \text{otherwise} \end{cases}$

Table 2 Semantics (interpretation) for common bit-vector operators

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

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

Definition 7 (Bit-blasting) *Bit-blasting*, or flattening [44], a bit-vector formula Φ means to construct an equisatisfiable Boolean formula ϕ. Φ and ϕ are *equisatisfiable* over an interpretation $\widehat{Op}$ iff the following condition holds: there exists a satisfying model for Φ over $\widehat{Op}$ iff there exists a satisfying assignment for ϕ.

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

3.3.3 Logics and Encodings

For the rest of this paper, we fix the operator set we use to **Op** with the signature $\mathbf{\Sigma_{Op}}$ (Table 1) and the interpretation $\widehat{\mathbf{Op}}$ (Table 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 [4,11,13,21,27]. Notice, however, that relational operations can occur not only at the Boolean level, but even below that, due to Definition 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 8 (Flat Form) A bit-vector formula Φ is in *flat form* iff it does not contain any nested relational operations.

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

In this 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 9 (Logic with Unary and Binary Encoding) Given a bit-vector logic $\mathcal{L}$, let $\mathcal{L}1$ and $\mathcal{L}2$ denote the logic $\mathcal{L}$ using unary and binary encoding on all the scalars in formulas, respectively.

In the rest of this 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 10 (Formula Size) Suppose we are given a bit-vector logic $\mathcal{L}$ and a formula $\Phi \in \mathcal{L}$, with $\Phi := Q_0 x_0{}^{[n_0]} Q_1 x_1{}^{[n_1]} \dots Q_k x_k{}^{[n_k]}.t^{[1]}$. The *size* of Φ is defined as $|\Phi| := \left|x_0{}^{[n_0]}\right| + \dots + \left|x_k{}^{[n_k]}\right| + \left|t^{[1]}\right|$.

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

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

4 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 [41].

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 [4] and [13]):

Proposition 1 QF_BV1 *is* NP*-complete.*[2]

[2] This kind of result is often called unary NP-completeness [32].

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 2 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., [44, Chapter 3.3.1]). Therefore, QF_UFBV1 can be polynomially translated into QF_BV1. The other direction follows from the fact that QF_BV1 $\subset$ QF_UFBV1.

Adding quantifiers to QF_BV1 yields the following complexity (see also [19]):

Proposition 3 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 $\subset$ BV1 (following the same argument as in Proposition 1).

Adding quantifiers to QF_UFBV1 increases complexity exponentially:

Proposition 4 UFBV1 *is* NEXPTIME*-complete (see [59]).*

Proof The *Effectively Propositional Logic* (EPR), is a common NEXPTIME-complete [45] logic, and can be reduced to UFBV1 [59, Theorem 7]. For completing the other direction, apply the reduction in [59, Theorem 7] combined with bit-blasting of the bit-vector operations.

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

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

Definition 11 (Scalar-Bounded Formula Set) An infinite set S of bit-vector formulas is *(polynomially) scalar-bounded*, iff there exists a polynomial function $p : \mathbb{N} \mapsto \mathbb{N}$ such that $\forall \Phi \in S.\ max_{\mathsf{scl}}(\Phi) \leqslant p(cnt_{\mathsf{scl}}(\Phi))$.

Proposition 5 *Given a scalar-bounded set S of formulas with binary encoded scalars, any $\Phi \in S$ grows polynomially when re-encoding the scalars to unary format.*

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

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

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

By applying this proposition to the logics of Section 3.3.3 together with the results from Section 4, we get:

Corollary 1 *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 $S \in$ NP. If $S \subseteq$ BV2, then $S \in$ PSpace. If $S \subseteq$ UFBV2, then $S \in$ NExpTime.*

6 Quantifier-Free Logics with Binary Encoding

Our main contribution in [30,41] 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 1, and continuing with three fragments of QF_BV2 in Theorem 2, 3, 4. All these theorems reference separate lemmas, which we introduce afterwards.

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

Theorem 1 QF_BV2 *is* NExpTime*-complete [41].*

Proof It is easy to see that QF_BV2 ∈ NExpTime, since a QF_BV2 formula can be translated exponentially to QF_BV1 ∈ NP (Proposition 1), by applying a simple unary re-encoding to all the scalars in the formula. NExpTime-hardness of QF_BV2 is a direct consequence of Lemma 1, in which a fragment of QF_BV2 is proved to be NExpTime-hard.

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

In [30], we investigated the complexity of the satisfiability problem for the following three fragments of QF_BV2, which only allow a restricted set of bit-vector operations in formulas:

- QF_BV2$_{\ll c}$: only bitwise operations, equality, and *left shift by constant*, i.e., $t^{[n]} \ll c^{[n]}$ where c is a constant, are allowed.
- QF_BV2$_{\ll 1}$: only bitwise operations, equality, and *left shift by 1*, i.e., $t^{[n]} \ll 1^{[n]}$, are allowed.
- QF_BV2$_{\mathsf{bw}}$: only bitwise operations and equality are allowed.

Theorem 2 QF_BV2$_{\ll c}$ *is* NExpTime*-complete [30].*

Proof In Lemma 1, we give a reduction from DQBF (which is NExpTime-complete) to QF_BV2$_{\ll c}$. This shows the NExpTime-hardness of QF_BV2$_{\ll c}$. The fact that QF_BV2$_{\ll c}$ ∈ NExpTime directly follows from Theorem 1.

Theorem 3 QF_BV2$_{\ll 1}$ *is* PSpace*-complete [30].*

Proof In Lemma 2, we give a reduction from QBF (which is PSpace-complete) to QF_BV2$_{\ll 1}$. This shows the PSpace-hardness of QF_BV2$_{\ll 1}$. In Lemma 3, we then prove PSpace-inclusion by giving a reduction from satisfiability for QF_BV2$_{\ll 1}$ to the *model checking* problem for sequential circuits. Symbolic model checking for sequential circuits is PSpace-complete as well [51,52,54].

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 9.

Theorem 4 QF_BV2$_{\mathsf{bw}}$ *is* NP*-complete [30].*

Proof Since *Boolean formulas* are a subset of QF_BV2$_{\mathsf{bw}}$, NP-hardness follows directly. To show that QF_BV2$_{\mathsf{bw}}$ ∈ NP, we give a reduction from QF_BV2$_{\mathsf{bw}}$ to a *scalar-bounded* set of formulas $S \subset$ QF_BV2 in Lemma 4. The claim then follows from Corollary 1.

As already hinted in Proposition 2, adding uninterpreted functions to all quantifier-free logics we discussed so far does not affect complexity. We formalize this in the following proposition:

Proposition 6 QF_UFBV2 *and* QF_UFBV2$_{\ll c}$ *are* NExpTime*-complete,* QF_UFBV2$_{\ll 1}$ *is* PSpace*-complete, and* QF_UFBV2$_{\mathsf{bw}}$ *is* NP*-complete [30, 41].*

Proof Apply the same arguments as were used in Proposition 2.

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

Lemma 1 DQBF *can be reduced to* QF_BV2$_{\ll c}$ *[30,41].*

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

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

Note that these bit masks correspond to the so-called *binary magic numbers* (or magic masks in [39, p. 141]), and can arithmetically be calculated in the following way (actually as the result of a geometric sum):

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

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

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

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

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

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

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

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

The reduction. Let $\psi := Q.\phi$ denote a DQBF with quantifier prefix Q and matrix ϕ. 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 ψ to a QF_BV2$_{\ll c}$ formula Φ by eliminating the quantifier prefix and translating the matrix ϕ as follows:

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

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

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

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

$$E_{m'} \;\&\; \sim U_m \;=\; (E_{m'} \ll 2^m) \;\&\; \sim U_m \tag{1}$$

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

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

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

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

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

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

$$[\![U_m]\!][b_i] \neq [\![U_m]\!][b_j] \quad \text{and} \tag{2a}$$

$$[\![U_k]\!][b_i] = [\![U_k]\!][b_j]\,, \ \forall k \neq m. \tag{2b}$$

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. (2a) and (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. (2a) holds, since, due to construction, U_m consists of 2^{n-1-m} concatenated bit-vector fragments $0 \ldots 01 \ldots 1 = R(0, 2^m)R(1, 2^m)$ (with both 2^m zeros and 2^m ones). Therefore, it is easy to see that

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

With a similar argument, we can show that Eqn. (2b) holds:

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

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

Now, consider all possible assignments to the universal variables of our original DQBF ψ. For a given assignment $\alpha \in \{0, 1\}^n$, the existence of such a previously defined b_j for every U_m and b_i allows us to iteratively find a b_α such that $\big([\![U_0]\!][b_\alpha], \ldots, [\![U_{n-1}]\!][b_\alpha] \big) = \alpha$. Thus, we have a *bijective mapping* from the *universal assignments* α for ψ to the *bit-indices* b_α for Φ_1. Up to this point, each bit-vector $E_{m'}$ can basically still take $2^{(2^n)}$ different values in Φ_1. The value of each individual bit $[\![E_{m'}]\!][b_\alpha]$ corresponds to the value that $e_{m'}$ takes under a given universal assignment $\alpha \in \{0, 1\}^n$. Note that, without any further restriction, there is no connection between the different bits of $E_{m'}$ and, therefore, the bit-vector represents an arbitrary Skolem-function for $e_{m'}$.

It may have different values for all universal assignments and thus would allow $e_{m'}$ to depend on all universal variables. Consequently, Φ_1 is satisfiable 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 $E_{m'}$s that correspond to Skolem-functions which do not respect the dependency scheme of ψ. Whenever $e_{m'}$ does not depend on a universal variable u_m, we add the constraint of Eqn. (1). In DQBF, independence can be formalized in the following way: $e_{m'}$ does not depend on u_m if $e_{m'}$ has to take the same value in the case of all pairs of universal assignments $\alpha, \beta \in \{0,1\}^n$ where $\alpha[k] = \beta[k]$ for all $k \neq m$. Exactly this is enforced by our constraint. Looking at the corresponding bit-indices b_α and b_β for α and β, respectively, our constraint for independence ensures that $[\![E]\!][b_\alpha] = [\![E]\!][b_\beta]$. More precisely, Eqn. (1) ensures that the positive and negative cofactors of the Skolem-function for $e_{m'}$ with respect to an independent variable u_m have the same value. Having added those constraints, Φ_2 is now respecting the dependency scheme and therefore Φ is satisfiable iff the original DQBF ψ is satisfiable.

Lemma 2 QBF *can be reduced to* QF_BV2$_{\ll 1}$ *[30].*

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

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

Let ${b_0}^{[2^n]}, \ldots, {b_{n-1}}^{[2^n]}$ be n initially unconstrained bit-vector variables. By adding certain constraints, we 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 $0 \leqslant c < 2^n$ in binary representation, the next row is equal to $c + 1$. The rows of B therefore represent a counter from 0 to 2^{n-1}. We can capture this fact by adding the following n constraints, with $m \in \{0, \ldots, n-1\}$:

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

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 similiar to the one in Lemma 1. An example can be found in Appendix B.

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

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

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

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

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

$$E_{m'} \;\&\; \sim 1 \;=\; E_{m'} \ll 1 \tag{3}$$

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

$$U'_m \;=\; \sim\big((U_m \ll 1) \oplus U_m\big) \tag{4}$$

$$E_{m'} \;\&\; U'_m \;=\; (E_{m'} \ll 1) \;\&\; U'_m \tag{5}$$

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

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

Again, each bit-index of Φ corresponds to the evaluation of ψ under a specific assignment to the universal variables $u_0, \ldots, u_{n-1}$, and, by construction of ${U_0}^{[2^n]}, \ldots, {U_{n-1}}^{[2^n]}$, all possible assignments are considered. Eqn. (4) creates a bit-vector ${U'_m}^{[2^n]}$ 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 1, 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. (4) does not have to be added multiple times, if several existential variables depend on the same universal variable. Eqn. (5) and Eqn. (3) ensure that the corresponding bits of ${E_{m'}}^{[2^n]}$ satisfy the dependency scheme of ψ by only allowing the value of $e_{m'}$ to change if an outer universal variable takes a different value. If $Deps(e_{m'}) = \{u_0, \ldots, u_{n-1}\}$, i.e., if $e_{m'}$ depends on all universal variables, Eqn. (4) evaluates to $U'_0 = 0^{[2^n]}$, and, as a consequence, Eqn. (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 [23]. See also [51] for a translation from QBF to sequential circuits.

Lemma 3 $\mathrm{QF\_BV2}_{\ll 1}$ *can be reduced to sequential circuits [30].*

Proof In [55,56], the authors give a polynomial translation from quantifier-free Presburger arithmetic with bitwise operations (QFPAbit [53]) 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 [38]. We can adopt their approach in order to construct a translation for $\mathrm{QF\_BV2}_{\ll 1}$. Related work, introducing an automata-based representation for Presburger Arithmetic (without bitwise operations), can be found in [61].

For the most part, the basic structure as well as the arguments used throughout the reduction are the same as in [55,56]. To keep the proof compact, we therefore focus on pointing out the changes compared to their earlier work and regularly refer to [55,56] for the technical details.

As mentioned, the main difference between QFPAbit and $\mathrm{QF\_BV2}_{\ll 1}$ is the fact that bit-vectors of arbitrary, non-fixed, size are allowed in QFPAbit while all bit-vectors contained in $\mathrm{QF\_BV2}_{\ll 1}$ have a fixed bit-width. We now give the reduction.

Given $\Phi \in \mathrm{QF\_BV2}_{\ll 1}$ in *flat form*, let $x^{[n]}, y^{[n]}$ denote bit-vector variables, $c^{[n]}$ a bit-vector constant, and ${t_1}^{[n]}, {t_2}^{[n]}$ bit-vector terms only containing bit-vector variables and bitwise operations. Following [55,56], we further assume w.l.o.g that Φ only consists of logical combinations of three types of *atomic expressions*: ${t_1}^{[n]} = {t_2}^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \ll 1^{[n]}$. Similar to generating a formula in *flat form* (Definition 8), it is easy to see that any $\mathrm{QF\_BV2}_{\ll 1}$ formula can be written like this with only linear growth in size by introducing Tseitin variables.

We then encode each equality in Φ into an individual *sequential circuit* separately. In the following, those are referred to as *atomic sequentical circuits*. Compared to [55,56], two modifications for the construction of an atomic sequential circuits are needed. First, we need to give a translation of $x = y \ll 1$ to sequential circuits. This can be done, for example, by using the sequential circuit for $x = 2 \cdot y$ in QFPAbit. The second modification relates to dealing with *fixed-size* bit-vectors. Let n be the bit-width of all bit-vectors in a given atomic expression. We extend each atomic sequential circuit to include a counter (circuit). The counter initially is set to 0 and is incremented by 1

in each clock cycle up to a value of n. When the counter reaches a value of n, the counter as well as the original atomic sequential circuit keep their value during all remaining cycles. In this way, their output also remains the same during all following cycles.

Using D-type flip-flops, as in the definition of sequential circuits in Section 3.2, this can be easily realized by adding a combinatorial part: Assume that the counter consists of k bits, represented by flip-flops $c_0, \ldots, c_{k-1}$ with outputs $o_0, \ldots, o_{k-1}$, respectively. Checking whether the counter has reached a value of n can be realized by a Boolean function $f(o_0, \ldots, o_{k-1})$, represented as a combinatorial circuit. 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 combinatorial circuit realizing the function

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

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

In contrast to the implementation described in [55], we further assume that the input streams for all variables start with the least significant bit. As already pointed out by the authors in [55], their choice was arbitrary and it is no more complicated to construct the circuits the other way around.

Finally, after constructing all atomic sequential circuits, their outputs are combined by logical gates following the Boolean structure of Φ, in the same way as for non-fixed bit-width in [55,56]. Due to the counters being part of the atomic sequential circuits, we ensure that for every input stream x_i, that represents a bit-vector variable of bit-width n_i, only the first n_i bits of x_i influence the result of the whole circuit.

Lemma 4 $\mathrm{QF\_BV2}_{\mathsf{bw}} \in \mathrm{NP}$ *[30].*

Proof To show that $\mathrm{QF\_BV2}_{\mathsf{bw}} \in \mathrm{NP}$, we give a reduction from $\mathrm{QF\_BV2}_{\mathsf{bw}}$ to a *scalar-bounded* set of formulas S. With $S \subset \mathrm{QF\_BV2}$, the claim then follows from Corollary 1. An example, that combines further results from Section 7.2, can be found in Appendix C.

Suppose we are given a formula $\Phi \in \mathrm{QF\_BV2}_{\mathsf{bw}}$ in *flat form* (Definition 8). We assume that any *inequality* ${t_1}^{[n]} \neq {t_2}^{[n]}$ in Φ is expressed by $\sim\left({t_1}^{[n]} = {t_2}^{[n]}\right)$. If Φ contains any constants $c^{[m]}$ where $c \neq 0$, we remove those constants in a (polynomial) pre-processing step. Let ${c_{\mathsf{max}}}^{[m]} := b_{k-1} \ldots b_1 b_0$ be the largest constant in Φ denoted in binary representation with $b_{k-1} = 1$

and arbitrary bits $b_{k-2}, \ldots, b_0$. We now replace each equality ${t_1}^{[n]} = {t_2}^{[n]}$, in Φ with

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

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

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

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

In a similar way, if $n > k$, ${t_{\mathrm{HI}1}}^{[n-k]} = {t_{\mathrm{HI}2}}^{[n-k]}$ represents the remaining $n-k$ rows of the original equality corresponding to the most significant bits. For constructing ${t_{\mathrm{HI}1}}^{[n-k]}$ and ${t_{\mathrm{HI}2}}^{[n-k]}$, (i) replace each occurrence of a variable ${x}^{[n]}$ with the variable ${x_{\mathrm{HI}}}^{[n-k]}$, and (ii) replace each constant ${c}^{[n]}$ with ${0}^{[n-k]}$.

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

We now construct a formula Φ' by reducing the bit-widths of all bit-vector terms in Φ. We use $cnt_{\mathsf{eq}}\,(\Phi)$ to denote the number of equalities in Φ. Each term ${t}^{[n]}$ in Φ is then replaced with a term ${t}^{[n']}$, with $n' := \min\{n, cnt_{\mathsf{eq}}\,(\Phi)\} \leqslant |\Phi|$. Apart from this, Φ' is exactly the same as Φ. As a consequence, $max_{\mathsf{scl}}\,(\Phi') \leqslant |\Phi|$. The set of formulas constructed in this way is scalar-bounded according to Definition 11.

To complete our proof, we now have to show that the proposed reduction is sound, i.e., out of every satisfying assignment to the bit-vector variables ${x_1}^{[n_1]}, \ldots, {x_k}^{[n_k]}$ for Φ we can also construct a satisfying assignment to ${x_1}^{[n'_1]}, \ldots, {x_k}^{[n'_k]}$ for Φ' and vice versa.

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

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

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

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

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

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

Remark 1 A similar proof can be found in [35,36]. While the focus of [35,36] 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 [35,36] 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.

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

On the other hand, it is also interesting to explore possible reductions of base operations to additional ones. We showed that the satisfiability problem for QF_BV2$_{\mathsf{bw}}$, i.e., when *bitwise operations* and *equality* are used as base operations, is NP-complete. Using *left shift by 1* or *left shift by constant* as an additional base operation makes the satisfiability problem PSPACE-hard (Lemma 2) or NEXPTIME-hard (Lemma 1), 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.

7.1 Notation

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

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

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

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

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

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

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

7.2 QF_BV2$_{\mathsf{bw}}$

Let us introduce the operation *indexing* $t^{[n]}[i]$, which is defined as $t\,[i:i]$, i.e., a special case of *extraction*. Although, in Section 7.4, we show that adding *extraction* makes the fragment NExpTime-hard, QF_BV2$_{\mathsf{bw}}$ can be extended with *indexing* without growth in complexity.

Theorem 5 QF_BV2$_{\mathsf{bw}}$ *extended by indexing is in* NP.

Proof To show this, we extend the proof of Lemma 4 by an additional preprocessing step even before removing the non-zero constants. Suppose we are given a formula $\Phi \in$ QF_BV2$_{\mathsf{bw}}$, 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 $I = \{i_1, \ldots, i_m\}$ with $i_l < i_{l+1}$, $\forall l \in \{1, \ldots, m-1\}$. After extracting those bit-indices from Φ, we explicitly encode the corresponding bits into Boolean variables, by translating Φ in a similar way as in Lemma 4. Consider three different kinds of terms in the following order:

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

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

as the number of explicitly specified indices smaller than n. Now, similar to Lemma 4, replace each equality ${t_1}^{[n]} = {t_2}^{[n]}$ with

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

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

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

As in Lemma 4, we use ${t_{1,i}}^{[1]} = {t_{2,i}}^{[1]}$ to express the ith row of the original equality. In the same way, ${t_i}^{[1]}$, being introduced for an *indexing*, represents the ith bit of t. The new terms $t_{1,i}$, $t_{2,i}$, and t_i are constructed in the same way as in Lemma 4.

Similarly, if $n > l'$, the expression ${t_{\text{REM}_1}}^{[n-l']} = {t_{\text{REM}_2}}^{[n-l']}$ represents the remaining $n - l'$ rows of the original equality corresponding to the indices that have not been extracted explicitly. Those terms are again constructed in the same way as in Lemma 4, except for the construction of new constants: each constant $c^{[n]}$ is replaced with a new constant $c_{\text{REM}}{}^{[n-l']}$ by setting the jth bit of c_{REM} to the value of the kth bit of c, for $k := \min\{k' \mid |\{1, \ldots, k'\} \setminus I| = j\}$.

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

Similarly, extending QF_BV2$_{\mathsf{bw}}$ with additional relational operations from Table 1 does not increase complexity, either.

Theorem 6 QF_BV2$_{\mathsf{bw}}$ *extended by relational operations from Table 1 is in* NP.

Proof We give a reduction for the relational operation *unsigned less than* (bvult). The remaining relational operations in Table 1 can be reduced in a similar way. Given $\Phi \in$ QF_BV2$_{\mathsf{bw}}$ (without indexing), additionally containing expressions ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$, we adopt the proof of Lemma 4 in three ways.

First, the elimination of constants has to be modified. Again, let $c_{\mathsf{max}} := b_{k-1} \dots b_1 b_0$ be the largest constant in Φ denoted in binary representation with $b_{k-1} = 1$ and arbitrary bits $b_{k-2}, \dots, b_0$. Without loss of generality, assume $n > k$. We now replace each relation ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ in Φ with

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

All expressions ${t_{1,i}}^{[1]}$, ${t_{2,i}}^{[1]}$, ${t_{\mathrm{HI}_1}}^{[n-k]}$, and ${t_{\mathrm{HI}_2}}^{[n-k]}$ are defined in the same way as it was done in Lemma 4.

Second, we need to use the number of all the relational operations $cnt_{\mathsf{rel}}(\Phi)$, when reducing the bit-widths in Φ.

The third modification is needed for constructing a satisfying assignment α' for the bit-width reduced formula Φ' out of the satisfying assignment α for Φ. When selecting the bit-index which is used as a witness for the evaluation of a given expression ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$, we choose the index of the most significant bit which is assigned to a different value in the two terms. As in Lemma 4, an arbitrary bit-index can be chosen if both terms are assigned to the same value.

Again, the reduction is only possible because there is no interaction between different bit-indices. While we only considered ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ in our proof, it is easy to see that it holds for all relational operations from Table 1. All unsigned operations can be replaced by ${t_1}^{[n]} <_{\mathbf{u}} {t_2}^{[n]}$ as in the definition of Table 1. For signed operations, an additional *if-then-else* constraint on the most significant bit is needed.

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

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

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

Corollary 2 QF_BV2$_{\mathsf{bw}}$ *extended by indexing together with relational operations from Table 1 is in* NP.

See Appendix C for an example.

7.3 QF_BV2$_{\ll 1}$

Figure 1 depicts our forthcoming results on extending QF_BV2$_{\ll 1}$ with operations. An edge (o_1, o_2) means that o_1 can be reduced to o_2, together with bitwise operations and equality. The vertex *bvshl_1* represents *left shift by 1*, and plays a central role as being a base operation in QF_BV2$_{\ll 1}$. The vertex *bvmul_c* represents *multiplication by constant*, and the four vertices to the right correspond to different kinds of unsigned and signed relational operations. All the other vertices are self-explanatory. Note that each operation which is mutually reachable with *bvshl_1*, namely *bvlshr_1*, *bvadd*, *bvsub*, and *bvmul_c*, can be used as an alternative base operation instead of *bvshl_1*.

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

Theorem 7 QF_BV2$_{\ll 1}$ *extended by indexing is in* PSPACE.

Proof The counter we introduced in our translation from QF_BV2$_{\ll 1}$ to sequential circuits (Lemma 3) can be used to return the value at a specific bit-index of a bit-vector.

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

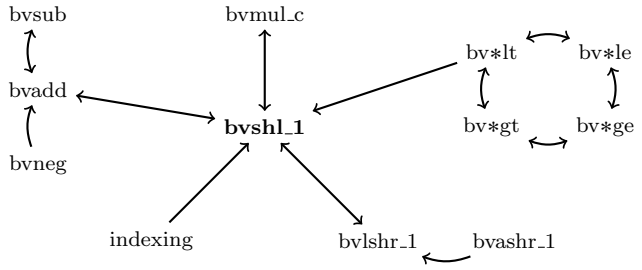

Fig. 1 Extending QF_BV2$_{\ll 1}$ with operations

Theorem 8 left shift by 1 *and* logical right shift by 1 *can be reduced to each other.*

Proof We give a direct translation:

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

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

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

We now look at arithmetic operations:

Theorem 9 QF_BV2$_{\ll 1}$ *extended with* linear modular arithmetic *is in* PSPACE.

Proof Addition can be expressed as follows:

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

Multiplication by constant can be splitted into several multiplications by 2, i.e., *left shift by 1*, and *addition*, similar to [55,56]. Given such a multiplication $t^{[n]} \cdot$

$c^{[n]}$, we introduce two sets of variables, s_i and x_i, $0 \leqslant i \leqslant \mathsf{L}c$. Each s_i represents $t \ll i$, and calculated by shifting s_{i-1} by 1. Note that only logarithmic many steps need to be performed. Each x_i represents the subresult in the ith step. By considering the individual bits of c, s_i either is or is not added to the previous subresult x_{i-1}. Finally, $x_{\mathsf{L}c}$ provides the required product.

$$t^{[n]} \cdot c^{[n]}$$

replace with: ${x_{\mathsf{L}c}}^{[n]}$

add assertions:

$${s_0}^{[n]} = t$$

$${s_i}^{[n]} = s_{i-1} \ll 1 \qquad , 0 < i \leqslant \mathsf{L}c$$

$${x_0}^{[n]} = \begin{cases} s_0 & \text{if } [\![c]\!][0] = 1 \\ 0 & \text{otherwise} \end{cases}$$

$${x_i}^{[n]} = \begin{cases} x_{i-1} + s_i & \text{if } [\![c]\!][i] = 1 \\ x_{i-1} & \text{otherwise} \end{cases} \qquad , 0 < i \leqslant \mathsf{L}c$$

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

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

7.4 QF_BV2$_{\ll c}$

Figure 2 depicts our forthcoming results on extending QF_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_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_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*, *bvlshr_c*, and *bvlshr*.

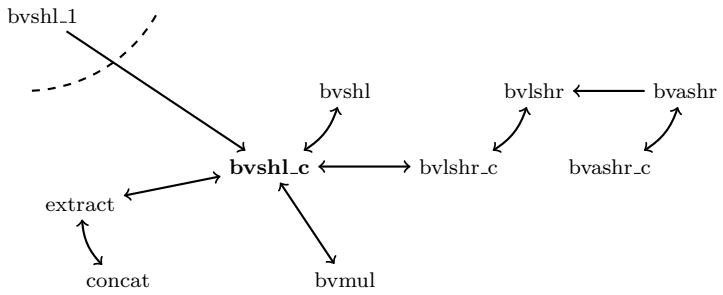

Fig. 2 Extending QF_BV2$_{\ll c}$ with operations

Theorem 10 bvshl_c *and* bvlshr_c *can be reduced to each other.*

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

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

,

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

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

Theorem 11 extraction, concatenation, *and* bvshl_c *can be reduced to each other.*

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

Proof First, consider extraction and concatenation:

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

,

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

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

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

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

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

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

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

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 12 multiplication *and* bvshl_c *can be reduced to each other.*

Proof First, we show how *bvshl_c* can be expressed by *bvmul*. Again, assume that $0 < c < n$. In this case, $t^{[n]} \ll c^{[n]}$ can be expressed as $t \cdot 2^c$. We can

construct 2^c, being an exponential number, as a bit-vector in $\mathsf{L}c$ steps using *exponentiation by squaring*. We introduce two sets of variables, p_i and x_i, $0 \leqslant i \leqslant \mathsf{L}c$. Each p_i represents the number $2^{(2^i)}$, and each x_i the subresult in the ith step. By considering the individual bits of c, the previous subresult x_{i-1} either is or is not multiplied by p_i. Finally, $x_{\mathsf{L}c}$ provides the value 2^c.

$$t^{[n]} \ll c^{[n]}$$

replace with: $t \cdot {x_{\mathsf{L}c}}^{[n]}$

add assertions:

$${p_0}^{[n]} = 2$$

$${p_i}^{[n]} = p_{i-1} \cdot p_{i-1} \quad , 0 < i \leqslant \mathsf{L}c$$

$${x_0}^{[n]} = \begin{cases} 2 & \text{if } [\![c]\!][0] = 1 \\ 1 & \text{otherwise} \end{cases}$$

$${x_i}^{[n]} = \begin{cases} x_{i-1} \cdot p_i & \text{if } [\![c]\!][i] = 1 \\ x_{i-1} & \text{otherwise} \end{cases} \quad , 0 < i \leqslant \mathsf{L}c$$

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

For the helper operation *binmagic*, which is in fact about constructing a *binary magic number*, we use the same notation and approach as in Lemma 1, where $m < n$:

$$\mathit{binmagic}\,(2^m,\ 2^n)$$

replace with: $x^{[2^n]}$

add assertion: $x \ll 2^m \ = \sim x$,

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

$$\mathit{selfconcat}\left(t^{[2^m]},\ 2^n\right)$$

replace with: ${x_n}^{[2^n]}$

add assertions:

$${x_m}^{[2^m]} = t$$

$${x_i}^{[2^i]} = x_{i-1} \circ x_{i-1} \quad , m < i \leqslant n$$

Halfshuffle applies a logarithmic translation, which is based on the generalization of a bit-vector operation called *half-shuffle* [58, Chpt. 7]. This generalized variant receives a bit-vector $t^{[2^m]}$ and produces the following bit-vector

of bit-width 2^n:

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

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

$$\textit{halfshuffle}\left(t^{[2^m]},\ 2^n\right)$$

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

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

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

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

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

$$\textit{expand}\left(t^{[2^m]},\ 2^n\right)$$

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

Now we are ready to propose how to express *multiplication* by simulating the common "shift and add" algorithm for integers. In a first step, one of the operands is multiplied independently by each digit of the other operand. Using base 2, this multiplication by a single digit can be expressed by a logical and-operation. Afterwards the results of the single-digit multiplications are shifted by the offset of the corresponding digit and finally added to give the result of the full multiplication. While this approach is straightforward in a naive implementation, we have to ensure only logarithmic many operations in the bit-width are used in our encoding. To achieve this, we generate bit-vectors

of quadratic bit-width $(\mathsf{P}n)^2$ out of our original operands of bit-width n, by applying *selfconcat* to the first one and *expand* to the second one. Using *bitwise and* on the two new bit-vectors, we directly get the results of all single-digit multiplications in one step. More precisely, the resulting bit-vector consists of $\mathsf{P}n$ groups of $\mathsf{P}n$ bits, each group representing the result of one single-digit multiplication. To add all $\mathsf{P}n$ partial results, a binary addition algorithm is used. Iteratively pairs of neighbouring groups are shifted relative to each others' offsets and then added to form one new group. The number of groups therefore is halved in each step, resulting in the final sum after $\log_2(\mathsf{P}n) = \mathsf{L}n$ steps. For a detailed example, see also Appendix E.

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

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

add assertions:

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

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

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

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 [41], 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.

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 13 UFBV2 *is* 2-NExpTime*-complete [41].*

Proof It is straightforward to see that UFBV2 $\in$ 2-NExpTime, since every UFBV2 formula can be translated exponentially to a formula in UFBV1 $\in$ NExpTime (Proposition 4), by applying a simple unary re-encoding to all the scalars in the formula. 2-NExpTime-hardness directly follows from Lemma 5.

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

Definition 12 (Domino System) A domino system is a tuple $\langle T, H, V, n\rangle$, where

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

Note that the above definition differs (but not substantially) from the classical one in [17]. Without loss of generality, we use sub-sequential natural numbers for identifying tiles. Similarly to [46,47], the size factor n, encoded in *unary* form, is part of the input. However, while a start tile α and a terminal tile ω is usually used, in our case 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 [17], a classical tiling problem is introduced, namely the *Square Tiling* problem, which can be defined as follows:

Definition 13 (Square Tiling) Given a domino system $\langle T, H, V, n\rangle$, an $f(n)$-square tiling is a mapping $\lambda : [0, f(n)-1] \times [0, f(n)-1] \mapsto T$ such that

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

In [17], a general theorem on the complexity of Domino Tiling problems is proved. More precisely, the $f(n)$-square tiling problem can be shown to be NTime$(f(n))$-complete. In particular, this implies that the $2^{(2^n)}$-square tiling problem is 2-NExpTime-complete.

Lemma 5 *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 $\mathsf{L}k$.
- Represent the horizontal and vertical matching conditions with the uninterpreted functions (actually predicates) $h^{[1]}(t_1{}^{[\mathsf{L}k]}, t_2{}^{[\mathsf{L}k]})$ and $v^{[1]}(t_1{}^{[\mathsf{L}k]}, t_2{}^{[\mathsf{L}k]})$, respectively.

– Represent the tiling with an uninterpreted function $\lambda^{[\mathsf{L}k]}(i^{[2^n]}, j^{[2^n]})$. λ returns the tile in the cell at the row index i and column index j. 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:

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

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

Similar to Section 6 and to our work in [30], we can now restrict the set of operations in UFBV2 to allow only bitwise operations, equality and *left shift by constant* (or *left shift by 1*). We refer to this logic as UFBV2$_{\ll c}$ (or UFBV2$_{\ll 1}$, in the case of *left shift by 1*). From a different point of view, it is also possible to consider this as an extension of QF_BV2$_{\ll c}$ and QF_BV2$_{\ll 1}$ by quantifiers and uninterpreted functions.

Since we can use bitwise operations, equality and *left shift by constant* to express all common operations, UFBV2$_{\ll c}$ remains 2-NExpTime-complete. However, in contrast to quantifier-free logics, we do not lose any expressiveness in UFBV2$_{\ll 1}$, either. We can see this already from the fact that we only used bitwise operations, equality and *addition* in Lemma 5. Since, as we pointed out in Section 7.3, *addition* can be reduced to bitwise operations, equality and *left shift by 1*, the following result follows immediately:

Corollary 3 UFBV2$_{\ll 1}$ *is* 2-NExpTime*-complete.*

Nevertheless, we want to formalize this in a proposition and give a constructive proof by showing how UFBV2$_{\ll c}$ can be reduced to UFBV2$_{\ll 1}$.

Proposition 7 UFBV2$_{\ll c}$ *can be reduced to* UFBV2$_{\ll 1}$.

Proof Let Φ denote a bit-vector formula, $x^{[n]}$, $y^{[n]}$ fresh bit-vector variables, and ${f_n}^{[n]}(\cdot,\cdot)$ a fresh uninterpreted function of arity 2, taking arguments of bit-width n. Replace each expression $t^{[n]} \ll c^{[n]}$ in Φ with ${f_n}^{[n]}\big(t^{[n]}, c^{[n]}\big)$,

extend the quantifier prefix of Φ with $\forall x^{[n]}, y^{[n]}$, and add the following two constraints to the matrix of Φ:

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

While the second constraint still contains *addition* to improve readability, this can be replaced by using *left shift by 1*, as described in Section 7.3.

Remark 2 This result is not very surprising if we consider the alternative characterizations of QF_BV2$_{\ll 1}$ and QF_BV2$_{\ll c}$ as given in Section 7. We showed that *addition* is equally expressive as *left shift by 1* and *multiplication* is equally expressive as *left shift by constant*. In *Peano arithmetic*, *multiplication* is defined by using *addition*, uninterpreted functions, and quantification. In the context of bit-vectors, this definition of multiplication can be expressed by introducing $\forall x^{[n]}, y^{[n]}$ to the quantifier prefix and adding the following constraints:

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

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

While we were also able to give some complexity results for BV2 in [41], it remains unclear whether BV2 is complete for any complexity class.

Proposition 8 BV2 $\in$ ExpSpace *and* BV2 *is* NExpTime*-hard [41].*

Proof Given a BV2 formula, a simple unary re-encoding can be used to give an exponential translation to BV1 $\in$ PSpace (Proposition 3). Therefore, BV2 $\in$ ExpSpace. Because of QF_BV2 $\subset$ BV2, NExpTime-hardness follows trivially.

8.2 Restricting the Bit-Width of Universal Variables

We now show that a completeness result can be obtained when a certain restriction to the bit-width of the universal variables is applied. For a given formula $\Phi \in$ BV2, let $max_{\mathsf{bw}(\exists)}(\Phi)$ and $max_{\mathsf{bw}(\forall)}(\Phi)$ denote the *maximal bit-width of all the existentially and universally quantified variables*, respectively. (We define $max_{\mathsf{bw}(\exists)}(\Phi) := 0$ and $max_{\mathsf{bw}(\forall)}(\Phi) := 0$ if Φ does not contain any existential or universal variables respectively.) Now we give a definition, similar to the one of scalar-boundedness in Definition 11:

Definition 14 (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 $p : \mathbb{N} \mapsto \mathbb{N}$ such that $\forall \Phi \in S.\ max_{\mathsf{bw}(\forall)}(\Phi) \leqslant p\left(\mathsf{L} max_{\mathsf{bw}(\exists)}(\Phi)\right)$.

Theorem 14 *If $S \subset$ UFBV2 (or $S \subset$ BV2) is universally bit-width bounded, then $S \in$ NExpTime.*

Proof Let $S \subset$ UFBV2 be universally bit-width bounded and let p_0 be the polynomial function that exists according to Definition 14. For any $\Phi_0 \in S$, let $n := |\Phi_0|$. We can assume that Φ_0 contains at most $k \leqslant n$ universal variables. Also, let $max_{\mathsf{scl}}(\Phi_0)$ and $cnt_{\mathsf{scl}}(\Phi_0)$ be defined in the same way as it was done in Section 5. This implies $max_{\mathsf{bw}(\exists)}(\Phi_0) \leqslant max_{\mathsf{scl}}(\Phi_0) \leqslant 2^n$ and $cnt_{\mathsf{scl}}(\Phi_0) \leqslant n$.

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

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

for some polynomial function p_1. Since Φ_1 does not contain any (universal) quantifiers, it can be polynomially translated to some $\Phi_2 \in$ QF_BV2, by replacing all uninterpreted functions of Φ_1 with bit-vector variables and adding at most quadratic many Ackermann constraints (as in Proposition 2). Therefore,

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

for some polynomial function p_2. In a last step, a unary re-encoding is applied to Φ_2 (similar to Proposition 1), resulting in $\Phi_3 \in$ QF_BV1. The size of Φ_3 is bounded by

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

for some polynomial function p_3. Therefore, $\Phi_3 \in$ QF_BV1 is only single-exponential in the size of Φ_0. Together with QF_BV1 $\in$ NP (Proposition 1), this shows that $S \in$ NExpTime.

We now define $\mathrm{BV}_{\mathsf{log}} \subset$ BV2 and $\mathrm{UFBV}_{\mathsf{log}} \subset$ UFBV2 as the set of all $\Phi \in$ BV2 and $\Phi \in$ UFBV2 with $max_{\mathsf{bw}(\forall)}(\Phi) \leqslant \mathsf{L} max_{\mathsf{bw}(\exists)}(\Phi) + 1$, respectively. These fragments are of special practical interest, because they can be used to express quantification over array indices if arrays are represented as bit-vectors. Arrays play an important role in automated Software Model Checking as, for example, done in the SAGE project by Microsoft [33]. Quantification over array indices is also discussed in [7], where the so-called *bounded array property fragment* is addressed.

Theorem 15 $\mathrm{BV}_{\mathsf{log}}$ *and* $\mathrm{UFBV}_{\mathsf{log}}$ *are* NExpTime*-complete.*

Proof It is easy to see that $\mathrm{BV}_{\mathsf{log}}$ and $\mathrm{UFBV}_{\mathsf{log}}$ are NExpTime-hard since both logics are an extension of QF_BV2, which is already NExpTime-hard (Proposition 6). The other direction is a consequence of Theorem 14, since $\mathrm{BV}_{\mathsf{log}}$ and $\mathrm{UFBV}_{\mathsf{log}}$ are universally bit-width bounded by definition.

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

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 [15,20]. The same obviously holds for QF_BV1.

However, as shown in Theorem 16, extending QF_UFBV2 (and even QF_BV2) with non-recursive macros does not give additional expressiveness, in terms of complexity. Let the subscript $\mathcal{M}$ denote the fact that, additionally, non-recursive macros can be used in our logic.

Definition 15 (Logic with Non-Recursive Macros) Given a bit-vector logic $\mathcal{L}$, let $\mathcal{L}_{\mathcal{M}}$ denote the set of all bit-vector formulas in the following form:

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

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

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

We now introduce a normal form, similar to the flat form in Definition 16, in order to obtain an upper bound for the growth in formula size when applying macro expansion.

Definition 16 (Functional Flat Form) A bit-vector formula Φ is in *function flat form* iff every uninterpreted function in Φ has only variables as arguments.

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

Theorem 16 QF_UFBV2$_{\mathcal{M}}$ *is* NExpTime*-complete.*

Proof NExpTime-hardness is obvious, since QF_UFBV2 $\subset$ QF_UFBV2$_{\mathcal{M}}$. Inclusion can be shown in a similar way as it is done in Theorem 14.

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

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

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

$$\begin{aligned} max_{\mathsf{scl}}(\Phi_1) &= max_{\mathsf{scl}}(\Phi_0) \leqslant 2^n \\ cnt_{\mathsf{scl}}(\Phi_1) &\leqslant l\big(n^{m+1}\big) \leqslant l\big(n^n\big) \leqslant l\big(2^{n \cdot \mathsf{L}n}\big) \end{aligned}$$

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

$$|\Phi_2| \leqslant max_{\mathsf{scl}}(\Phi_1) \cdot cnt_{\mathsf{scl}}(\Phi_1) + c \leqslant 2^n \cdot l\big(2^{n \cdot \mathsf{L}n}\big) + c$$

which is only single exponential in the size of Φ_0. This gives QF_UFBV2$_{\mathcal{M}} \in$ NExpTime.

9 Practical Considerations

As mentioned in Section 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 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.

9.1 Alternative Approaches

Instead of using bit-blasting, we can try to find alternative approaches for solving bit-vector formulas.

One possible approach is to polynomially translate bit-vector formulas to some other logic in the same complexity class. For example, target logics for QF_BV2$_{\ll c}$ (or general QF_BV2) are DQBF or EPR, which are both NExpTime-complete [45,49,50]. For QF_BV2$_{\ll 1}$, a translation to model checking for sequential circuits as given in Lemma 3 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 [28], their performance does not nearly reach the one of current EPR solvers as, e.g., iProver [40]. On the other hand, many efficient model checkers for sequential circuits in SMV or AIGER format exist.

In [42], we therefore gave a polynomial translation from QF_BV2 to EPR (this is in contrast to existing translations in [26,37], which are not guaranteed to be polynomial in the general case), and then did an experimental evaluation using iProver to solve the resulting EPR formulas. The overall results were rather mixed. While we were able to solve some formulas faster, SMT solvers performed better by orders of magnitude on most other problems considering runtime. Looking at the space requirements, iProver performed better in general. However, the gain was less significant than expected. An explanation for this can be found in the way iProver deals with EPR formulas. By solving propositional overapproximations and iteratively applying instantiations of predicates (the underlying concept is known as the Inst-Gen calculus), the formula can also grow exponentially. Of course this is no flaw in iProver but a direct consequence of the NExpTime-completeness of EPR and QF_BV2.

A different situation occurs when we look at QF_BV2$_{\ll 1}$. As seen in our introductory example, bit-blasting can still be exponential for formulas of this class. However, we know that it is possible to solve this kind of formulas in polynomial space, since QF_BV2$_{\ll 1} \in$ PSpace. In [29], we therefore presented a polynomial translation from QF_BV2$_{\ll 1}$ to SMV. Since current model checkers usually expect input in AIGER format, we then also translated our outputs to AIGER format using `smv2aig`, which is part of the AIGER library. Our experiments showed that, with growing bit-width, BDD-based model checkers (e.g., NuSMV [18] and IImc[5], using techniques described in [6,8], 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

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

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 [30, 41] and Section 6 can practically lead to improvements in state-of-the-art SMT solving. It is an interesting open problem to look at these results more closely and to integrate those concepts into SMT solvers in order to to increase their overall performance.

9.2 Benchmark Problems

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

In [42], we proposed two new sets of QF_BV2 benchmarks for our experiments on evaluating the performance of EPR solvers for quantifier-free bit-vector formulas. In connection with our experiments on using model checkers for efficiently solving restricted bit-vector formulas, we generated six more benchmark sets for QF_BV2$_{\ll 1}$ in [29].

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 6, all common bit-vector operations can be logarithmically expressed (in bit-width) by bitwise operations and equality in combination with *shift by constant*, *multiplication*, *concatenation*, or *slicing*. While we did not give direct translations for all common bit-vector operations in this work, we encoded most of them into SMT-LIB instances and used Boolector to verify their correctness for various bit-widths.

These benchmarks, together with those from [29,42], 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] [43].

10 Conclusion

We discussed the complexity of deciding various quantified and quantifier-free fixed-size bit-vector logics. In contrast to existing literature, where usually it is not distinguished between *unary* and *binary encoding* on scalars in formulas, we argued that it is important to make this distinction. Most of our results apply to the actually much more natural binary encoding as it is also used

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

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

in standard formats, e.g., in the SMT-LIB format. For this kind of logics, already the quantifier-free fragment without uninterpreted functions (QF_BV2) turned out to be NExpTime-complete [41].

In this paper, we extended our previous work from [30,41]. 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 [30,41] and extended those results in several ways. For quantifier-free logics, we combined our earlier work and restructured it to present several of our proofs in a clearer, easier-to-read way, with some small modifications in the proofs.

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

Considering quantified logics, it is still an open question whether BV2 is complete for any complexity class. However, we could give some new results for quantified logics with a restriction on the bit-width of universal variables. We introduced the notion of *universally bit-width bounded* problems and showed that this kind of problems are in NExpTime. This then allowed us to prove that $\mathrm{BV2}_{\mathsf{log}}$ is NExpTime-complete. Since bit-vector logics with arrays represented by bit-vectors are in this set if quantification is only allowed on array indices, this class is of particular practical interest.

For a last complexity theoretical result, we looked into $\mathrm{QF\_BV2}_{\mathcal{M}}$, the class of quantifier-free bit-vector logics extended with *non-recursive macros*, as allowed, e.g., in the SMT-LIB format. Again, we showed that this logic remains NExpTime-complete. Altogether, we provide the currently most complete overview on the complexity of common bit-vector logics.

To point out that our theoretical insights are also interesting from a practical point of view, we briefly discussed two approaches of solving bit-vector formulas not by bit-blasting but by using translations based on our complexity results. While bit-blasting is exponential in general, we proposed polynomial translations into EPR and SMV in recent practical work [29,42], 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 [24]. 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.

References

1. Abdelwaheb Ayari, David A. Basin, and Felix Klaedtke. Decision procedures for inductive boolean functions based on alternating automata. In *CAV*, volume 1855 of *LNCS*. Springer, 2000.
2. Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. Henkin quantifiers and boolean formulae. In *Proc. SAT'12*, 2012.
3. Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
4. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference*, pages 522–527, 1998.
5. Nikolaj Bjørner and Mark C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *TACAS*, volume 1384 of *LNCS*, pages 376–392. Springer, 1998.
6. Aaron R. Bradley. Sat-based model checking without unrolling. In *Proc. VMCAI'11*, pages 70–87, 2011.
7. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.
8. Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Proc. FMCAD'11*, pages 144–153, 2011.
9. Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.
10. Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: bit-precise modelling of word-level problems for model checking. In *Proc. 1st International Workshop on Bit-Precise Reasoning*, pages 33–38, New York, NY, USA, 2008. ACM.
11. Roberto Bruttomesso. *RTL Verification: From SAT to SMT(BV)*. PhD thesis, University of Trento, 2008.
12. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT SMT solver. In *CAV*, volume 5123 of *LNCS*, pages 299–303. Springer, 2008.
13. Roberto Bruttomesso and Natasha Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *ICCAD*, pages 13–20. IEEE, 2009.
14. Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 358–372, Berlin, Heidelberg, 2007. Springer-Verlag.
15. Uwe Bubeck and Hans Kleine Büning. Encoding nested boolean functions as quantified boolean formulas. *JSAT*, 8(1/2):101–116, 2012.
16. Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve, and Steven P. Miller. A bitvectors library for pvs. Technical report, NASA Langley Research Center, Hampton, Virginia, USA, August 1996.
17. Bogdan S. Chlebus. From domino tilings to a new model of computation. In *Symposium on Computation Theory*, volume 208 of *LNCS*. Springer, 1984.
18. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv version 2: An opensource tool for symbolic model checking. In *Proc. CAV02*, 2002.
19. Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *TACAS*, volume 6015 of *LNCS*. Springer, 2010.
20. Stephen Cook and Michael Soltys. Boolean programs and quantified propositional proof systems. *Bulletin of the Section of Logic*, 1999.
21. David Cyrluk, Oliver Mller, and Harald Rue. An efficient decision procedure for a theory of fixed-sized bitvectors with composition and extraction. In *Computer-Aided Verification (CAV 97)*, pages 60–71. Springer, 1997.

22. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
23. Francesco M. Donini, Paolo Liberatore, Fabio Massacci, and Marco Schaerf. Solving QBF with SMV. In *Proc. KR'02*, pages 578–589, 2002.
24. Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999. 530 pp.
25. Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.
26. Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In *FMCAD'10*, pages 137–144, 2010.
27. Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.
28. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. A DPLL algorithm for solving DQBF. In *Proc. POS'12*, 2012.
29. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. Efficiently solving bit-vector problems using model checkers. In *Proc. SMT'13*, 2013.
30. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In *Proc. CSR'13*, 2013.
31. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.
32. Michael R. Garey and David S. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, July 1978.
33. Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.
34. L. Henkin. Some remarks on infinitely long formulas. In *Infinistic Methods*, pages 167–183. Pergamon Press, 1961.
35. Peer Johannsen. Reducing bitvector satisfiability problems to scale down design sizes for RTL property checking. In *Proc. High-Level Design Validation and Test Workshop, 2001*, pages 123–128, 2001.
36. Peer Johannsen. *Speeding Up Hardware Verification by Automated Data Path Scaling*. PhD thesis, CAU Kiel, Germany, 2002.
37. Zurab Khasidashvili, Mahmoud Kinanah, and Andrei Voronkov. Verifying equivalence of memories using a first order logic theorem prover. In *FMCAD'09*, pages 128–135, 2009.
38. Nils Klarlund, Anders Mller, and Michael I. Schwartzbach. Mona implementation secrets. In *Proc. CIAA'00*, pages 182–194, 2000.
39. Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms*. Addison-Wesley, 2011.
40. Konstantin Korovin. iProver — an instantiation-based theorem prover for first-order logic (system description). In *Proc. IJCAR'08*, IJCAR '08. Springer, 2008.
41. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proc. SMT'12*, pages 44–55, 2012.
42. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. bv2epr: A tool for polynomially translating quantifier-free bit-vector formulas into epr. In *Proc. CADE'13*, 2013.
43. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Quantifier-free bit-vector formulas with binary encoding: Benchmark description. In A. Balint, A. Belov, M. Heule, and M. Järvisalo, editors, *Proc. SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 107–108. University of Helsinki, 2013.
44. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. Springer, 2008.
45. Harry R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.
46. Maarten Marx. Complexity of modal logic. In *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 139–179. Elsevier, 2007.

47. Matthias Niewerth and Thomas Schwentick. Two-variable logic and key constraints on data words. In *ICDT*, pages 138–149, 2011.
48. Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
49. G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information, 2001.
50. Gary L. Peterson and John H. Reif. Multiple-person alternation. In *FOCS*, pages 348–363. IEEE Computer Society, 1979.
51. Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.
52. Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, April 1970.
53. Tobias Schuele and Klaus Schneider. Verification of data paths using unbounded integers: automata strike back. In *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, HVC'06, pages 65–80, Berlin, Heidelberg, 2007. Springer-Verlag.
54. A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.
55. Andrej Spielmann and Viktor Kuncak. On synthesis for unbounded bit-vector arithmetic. Technical report, EPFL, Lausanne, Switzerland, February 2012.
56. Andrej Spielmann and Viktor Kuncak. Synthesis for unbounded bitvector arithmetic. In *International Joint Conference on Automated Reasoning (IJCAR)*, LNAI. Springer, 2012.
57. Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *STOC*, pages 1–9. ACM, 1973.
58. Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman, 2002.
59. Christoph M. Wintersteiger. *Termination Analysis for Bit-Vector Programs*. PhD thesis, ETH Zurich, Switzerland, 2011.
60. Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. In *Proc. FMCAD*, pages 239–246. IEEE, 2010.
61. Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *In Proc. Static Analysis Symposium, LNCS 983*, pages 21–32. Springer LNCS, 1995.

A Example: A Reduction of a DQBF to QF_BV2$_{\ll c}$

Consider the following DQBF:

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

This DQBF is *unsatisfiable*.

Using the reduction given in Lemma 1, this formula is translated to the following QF_BV2$_{\ll c}$ formula:

$$\begin{aligned}&\Big((X \mid Y \mid\!\sim U_0 \mid\!\sim U_1)\ \&\ (X \mid\!\sim Y \mid U_0 \mid\!\sim U_1 \mid\!\sim U_2)\ \&\ (X \mid\!\sim Y \mid\!\sim U_0 \mid\!\sim U_1 \mid U_2)\ \&\\ &(\sim X \mid Y \mid\!\sim U_0 \mid\!\sim U_2)\ \&\ (\sim X \mid\!\sim Y \mid U_0 \mid U_1 \mid\!\sim U_2)\Big)\ = \sim 0^{[8]}\ \ \wedge \\ &\bigwedge_{m \in \{0,1,2\}} U_m \ll 2^m\ = \sim U_m\ \ \wedge \\ &X\ \&\ \sim U_1\ =\ (X \ll 2^1)\ \&\ \sim U_1\ \ \wedge \\ &X\ \&\ \sim U_2\ =\ (X \ll 2^2)\ \&\ \sim U_2\ \ \wedge \\ &Y\ \&\ \sim U_0\ =\ (Y \ll 2^0)\ \&\ \sim U_0\end{aligned} \tag{6}$$

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

Recall that the notation $t^{[n]} \stackrel{\square}{=} d$ is an alternative for $[\![t^{[n]}]\!] = d$, assuming an appropriate model for t. By construction, $U_0 \stackrel{\square}{=} 01010101$, $U_1 \stackrel{\square}{=} 00110011$, and $U_2 \stackrel{\square}{=} 00001111$.

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

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

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

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

Furthermore, considering also the equality

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

results in

$$X\ \stackrel{\square}{=}\ (x_1, x_0, x_1, x_0, x_1, x_0, x_1, x_0)$$

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

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

In order to show that the formula (6) is unsatisfiable, let us evaluate the "clauses" in the formula:

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

By applying *bitwise and* to them, we get the bit-vector constrained by the formula (6):

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

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

B Example: A Reduction of a QBF to QF_BV2$_{\ll 1}$

Consider the following QBF:

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

This QBF is *satisfiable*.

Using the reduction given in Lemma 2, this formula is translated to the following QF_BV2$_{\ll 1}$ formula:

$$\begin{aligned} & \Big((U_2 \mid U_1 \mid\sim Z) \;\&\; (U_2 \mid\sim X \mid Y) \;\&\; (U_0 \mid\sim X \mid\sim Z) \;\& \\ & (U_1 \mid\sim Y \mid Z) \;\&\; (U_0 \mid\sim U_1 \mid Z) \Big) \;=\; \sim 0^{[8]} \;\wedge \\ & \bigwedge_{m \in \{0,1,2\}} \Big(\bigwedge_{0 \leqslant i < m} U_i \Big) \oplus U_m \;=\; U_m \ll 1 \;\wedge \\ & X \;\&\; \sim 1 \;=\; X \ll 1 \;\wedge \\ & \Big(U_2' \;=\; \sim \big((U_2 \ll 1) \oplus U_2 \big) \Big) \;\wedge\; \big(Y \;\&\; U_2' \;=\; (Y \ll 1) \;\&\; U_2' \big) \end{aligned} \tag{7}$$

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

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

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

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

$$X \;\stackrel{\square}{=}\; (x_0, x_0, x_0, x_0, x_0, x_0, x_0, x_0)$$

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

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

the following bit-vectors

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

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

$$Y \;\widehat{=}\; (y_4, y_4, y_4, y_4, y_0, y_0, y_0, y_0)$$

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.

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

In order to show that the formula (7) is satisfiable, let us evaluate the "clauses" in the formula:

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

By applying *bitwise and* to them, we get the bit-vector constrained by the formula (7):

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

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

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

Therefore,

$$\begin{aligned} U_0 \;\widehat{=}\; 01010101, \quad U_1 \;\widehat{=}\; 00110011, \quad U_2 \;\widehat{=}\; 00001111, \\ X \;\widehat{=}\; 00000000, \quad Y \;\widehat{=}\; 00000000, \quad Z \;\widehat{=}\; 00111111 \end{aligned}$$

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

C Example: Bit-Width Reduction of a QF_BV2$_{\mathsf{bw}}$ Formula with Indexing and Relational Operations

Let

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

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

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

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

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

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

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

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

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

by

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

respectively.

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

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

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

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

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

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

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

E Example: Multiplication of Two Bit-Vectors

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

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

${g_3}^{[4]}, {g_2}^{[4]}, {g_1}^{[4]}$, and ${g_0}^{[4]}$ are the bit groups representing the bit-vector which is the result of single-digit multiplication of ${t_1}^{[4]} = 0011$ with the single bits of ${t_2}^{[4]} = 0101$. Now, the neighbouring groups have to be shifted to their relative offsets and are added:

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

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

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

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

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

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.