Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 179–190
doi: 10.14794/ICAI.10.2017.179

How Hard is Bit-Precise Reasoning?*

Gergely Kovásznai

Eszterházy Károly University, Eger, Hungary
kovasznai.gergely@uni-eszterhazy.hu

Abstract

Formal verification of hardware and software is important in the case of critical systems, as shown by several accidents due to such problems as overflow, rounding error, etc. Bit-precise reasoning over bit-vectors is a fundamental tool for attacking such verification problems. But how hard is reasoning over bit-vector logics? It is essential to answer this question before inventing solving approaches for bit-vector logics.

The talk gives an overview on the complexity of bit-vector logics. Complexity depends on what bit-vector operators are used and whether quantifiers are permitted. For instance, the quantifier-free bit-vector logic (QF_BV) with all the common operators is NEXPTIME-complete, which seems extremely high considering the fact that QF_BV is commonly used in hardware verification. We will investigate how that complexity can be reduced to NP-completeness or PSPACE-completeness by restricting the set of operators.

Quantifiers are often used in software verification problems such as termination analysis and invariant synthesis. The bit-vector logic with quantifiers and uninterpreted functions (UFBV) is 2-NEXPTIME-complete. Interestingly, as it was proved recently, that logic without uninterpreted functions (BV) is AEXP(poly)-complete.

Keywords: bit-precise reasoning, bit-vector, logic, complexity

MSC: 68Q17, 68M15, 68N30

1. Introduction

Why does humanity still struggle with errors in computer systems? Or in other words, why so many bugs in our software and hardware? Do there not exist sophisticated development tools for avoiding such bugs already?

Yes, there does but today's tools do not necessarily scale to future systems. The main reason for that is that the complexity of our systems increases by time.

*The research was supported by the grant EFOP-3.6.1-16-2016-00001 "Complex improvement of research capacities and services at Eszterhazy Karoly University".

A well-known example is chip complexity which follows Moore's law. While we are currently capable to detect bugs in hardware designs such as the infamous floating-point division bug in 1994's Intel Pentium (see Section 2), which contained ca. 3 million transistors, it is much more difficult to do the same in today's processors which can contain ca. 3 billion transistors. Regarding software, the trend is similar in terms of complexity, not to mention that nowadays we need to debug distributed software systems. From the point of view of software developers, there is less and less time to write fault-tolerant code since the average development cycle time decreases. For instance, it took 1.5 years to release Mozilla Firefox 2.0 after version 1.0, nowadays new versions are released in ca. every 2 months.

In Section 2, we are going to get acquainted with a couple of historically infamous hardware and software bugs that caused a tremendous loss of money and, sometimes, a loss of human lives. Section 3 introduces bit-precise verification as a powerful tool for avoiding bugs. Of course, it is an important question if verifying hardware and software systems is feasible and, therefore, we are going to summarize in Section 4 what we currently know about the computational complexity of bit-precise reasoning in different settings.

2. Historical Accidents Caused by Bugs

There exist well-known historical accidents due to bugs in hardware or software. In this section, let us introduce a few of those.

Let us start with the infamous accident of NASA's Mars Climate Orbiter. This spacecraft was launched on December 11, 1998, and was scheduled to land on Mars on September 23, 1999. Shortly after starting the engines for landing, the signal was lost, 49 seconds earlier than predicted, and could not reacquired later on. The lander unit was lost, and probably burnt, in Mars' atmosphere. The investigation on the causes of the accident determined that the root cause was that a team of engineers used English units in preparing the data file for the software which, however, used metric units. Subsequent processing of the data by the navigation software underestimated the effect on the spacecraft trajectory [1]. The mission costed $125 million [20].

Another well-known accident from space industry is the explosion of the Ariane 5 launcher. After only 40 seconds of flight, at an altitude of 3700 m, the launcher veered off its flight path, broke up and exploded. The European Space Agency (ESA) set up an Inquiry Board to investigate what caused the failure. The board found that the root cause was an *integer overflow* in the software: a 64-bit floating point value was converted to a 16-bit signed integer, causing a value greater than what could be represented as a 16-bit integer [2]. The launcher was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and

[1] Mars Climate Orbiter Mishap Investigation Board, Phase I Report, 1999, `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`

[2] Inquiry Board (by ESA and CNES), Ariane 5: Flight 501 Failure, 1996, `https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`

its cargo were valued at $500 million. The recovery of the debris and the rocket fuel scattered over an area of 12 km^2 required lots of efforts as well.

A similar case from the recent past (2015) is an overflow bug in the software of Boeing Model 787. During laboratory testing, Boeing identified an issue with an internal counter in the software: if the electric generator control unit was continuously powered up during 248 days, it went to failsafe mode, resulting in a loss of electric power, regardless of flight phase. The investigation of the Federal Aviation Administration (FAA) found that the issue was caused by *integer overflow* [3]. Note that 2^{11} hundredths of a second equals to 248 days. Fortunately, the problem was discovered before any accident happened.

In the following accident, unfortunately, there were some casualties as well. The Instituto Oncológico Nacional in Panama provided treatment for cancer patients using radiotherapy. A computerized treatment planning system was used to calculate the resulting dose distributions and determine treatment times. As was found later, at least *one of the ways in which the data were entered for the software resulted a treatment time two times longer* than it should have been [4]. The result was that patients received a proportionately higher dose than that prescribed. The erroneous treatment protocol was used for 28 patients. By the time of the investigation, 8 patients had already died, among which at least 5 of the deaths were probably radiation related. Most of the 20 surviving patients suffered intestinal injuries.

The last accident that I am willing to mention did not cause casualties, to the best of my knowledge, but cost $450 million. In 1994, Intel discovered a *flaw in the floating-point division circuitry* of the early Pentium processors [5]. The cause of the error was the use of a lookup table with some cells being empty, resulting in slightly less precise numbers than the correct answer [13]. Although Intel claimed that the bug encountered for typical spreadsheet users once in 27 000 years, the company finally recalled the defective processors, costing ca. $450 million.

Such bugs and, as unfortunate consequences, accidents could be avoided if companies invested more on applicable techniques. In fact, this is exactly what the largest companies already do in order to prevent product recalls and, most importantly, loss of money and prestige. As the example of Boeing shows, testing is such a powerful technique. However, testing does not provide 100 % guarantee since executing the system being tested for all the possible inputs is typically infeasible. Verification, on the other hand, can provide such a guarantee for the types of bugs which you can define in advance.

[3] Federal Aviation Administration (FAA), Final Rule, 2015, `https://www.federalregister.gov/d/2015-10066`

[4] International Atomic Energy Agency, Investigation of an Accidental Exposure of Radiotherapy Patients in Panama, 2001, `https://www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf`

[5] Intel Corporation, Statistical Analysis of Floating Point Flaw in the Pentium Processor, 1994, copy at `http://users.minet.uni-jena.de/~nez/rechnerarithmetik_5/fdiv_bug/intel_white11.pdf`

3. Bit-Precise Verification

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

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

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

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

The SAT problem is meant the satisfiability checking of Boolean formulas. Although Boolean logic seems extremely simple, the computation complexity of SAT is very high. In fact, SAT was the first computational problem that was shown to be NP-*complete* by encoding any polynomial time-bounded non-deterministic Turing machine as a SAT instance [7]. Assuming P $\neq$ NP, SAT cannot be solved by a polynomial time (deterministic) algorithm in general.

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

tion of the Intel Core i7 processor design [15] and the analysis of cryptographic cyphers [23] by the help of SAT solvers.

It is a fairly natural idea to extend SAT solving with *background theories* such as integer or real arithmetic, or arrays. Needless to say that such an extension would have clear practical value since it would let our logical formulas contain atoms which, for instance, might evaluate some arithmetic expression over numbers or might check the value of array elements. *Satisfiability Modulo Theories (SMT)* is the decision problem of satisfiability checking of Boolean formulas with respect to some background theory and logic. The most common examples of *theories* are the integer numbers, the real numbers, the fixed-size bit-vectors, and the arrays. The *logics* that one could use might differ from each other in the linearity or non-linearity of *arithmetic*, the presence or absence of *quantifiers*, or in the presence or absence of *uninterpreted functions*. The *SMT-LIB* format [2], as the common input format for SMT solvers, defines the syntax for several such logics [6], such as QF_UFLIA as the quantifier-free logic of linear integer arithmetic with uninterpreted functions, or LRA as the logic of linear real arithmetic allowing quantification, or AUFLIA as the logic of linear integer arithmetic with quantifiers, uninterpreted functions and arrays.

In this paper, we are focusing on the background theory of *fixed-size bit-vectors*, also known as *words* or *sequences of bits*, i.e., Boolean values. The fundamental building blocks of bit-vector formulas are the *bit-vector variables* $x^{[n]}$ *and constants* $c^{[n]}$ *of certain bit-widths* n. To those variables and constants, different kinds of *bit-vector operators* can be applied [2, 1, 3, 4, 8, 11, 18], such as bitwise operators (*negation*, *and*, *or*, *xor*, etc.), relational operators (*equality*, *unsigned/signed less than*, *unsigned/signed less than or equal*, etc.), arithmetic operators (*addition*, *subtraction*, *multiplication*, *unsigned/signed division*, *unsigned/signed remainder*, etc.), shifts (*left shift*, *logical/arithmetic right shift*), *extraction*, *concatenation*, *zero/sign extension*, etc. For a detailed introduction into the syntax and semantics of bit-vector operators, we recommend the relevant parts of [18].

Bit-vector logics play an important role in many practical applications of computer science, most prominently in hardware and software verification, due to the fact that every piece of data in hardware or software has a given bit-width. Notable example are, for instance, the verification of Windows device drivers [19] and also of communication protocols of wireless sensor networks [10].

In *hardware verification*, the quantifier-free bit-vector logic QF_BV is commonly used in practice. If synthesis is also addressed, *uninterpreted functions* are added to this logic in order to allow to introduce custom signatures of function symbols on demand; the resulting logic is denoted by QF_UFBV.

In *software verification* where, for instance, loops are crucial components of the model to verify, quantifiers play an important role. BV denotes the bit-vector logic that allows quantification over the bit-vector variables. To be able to tackle verification tasks such as the termination analysis of loops and invariant synthesis, uninterpreted functions are also essential, resulting in the logic of UFBV.

[6] `http://smtlib.cs.uiowa.edu/logics.shtml`

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

The *computational complexity* of bit-blasting for the common bit-vector logics had not been clear for long time, although it is necessary to investigate for the sake of proving the *membership* of bit-vector logics in certain complexity classes. It is even more difficult to prove their *hardness* to those complexity classes, for the sake of investigating the precise characterization of the computation complexity of bit-vector logics.

4. Complexity of Bit-Vector Logics

The vast majority of bit-vector solvers rely on bit-blasting. This is a technique to translate a bit-vector formula to a Boolean formula whose Boolean variables represent the individual bits of the bit-vectors. Bit-blasting is known to be a polynomial reduction in the bit-width of the bit-vectors (regarding the commonly used bit-vector operators). Therefore it seems logical to say that bit-blasting is polynomial, and thus the satisfiability problem of a bit-vector logic is in the same complexity class as the underlying Boolean satisfiability problem. For instance, QF_BV should be in NP since SAT is in NP. It turned out that this reasoning failed in general. The reason for that originates in the way of how bit-widths, as scalars, are encoded in bit-vector formulas. In practice, they are encoded as decimal numbers in the input formula, i.e., they employ exponentially succinct encoding, and, therefore, bit-blasting could be exponential.

There exist a few complexity results for bit-vector logics in the literature, out of which some hold, but some do not. [3] correctly states that QF_BV is NP-hard and, similarly, [4] claims that QF_BV has an NP-complete sublogic, also known as a fragment. On the other hand, [5] incorrectly claims that the full fragment of QF_BV is NP-complete. For bit-vector logics with quantification, [24, 25] incorrectly propose that UFBV is NExpTime-complete. All those complexity results hold only if bit-widths are assumed to be encoded as unary numbers, which is, of course, neither the common nor the general case.

We showed for the first time that the commonly used bit-vector logics have higher complexity in general than the verification community had though before [16]. The higher complexity is due to the exponentially succinct, logarithmic encoding used in practice to represent the bit-widths of bit-vectors in the input formulas. In the paper, we investigate how complexity varies if we consider a logarithmic w.l.o.g. *binary encoding*. The paper shows that the binary encoding of bit-widths has at least as much expressive power as quantification. Table 1 summarizes our new complexity results in this paper [16], complemented by a result provided later by [14].

		quantifiers			
		no		*yes*	
		uninterpreted functions		uninterpreted functions	
		no	*yes*	*no*	*yes*
encoding	*unary*	NP	NP	PSPACE	NEXPTIME
	binary	NEXPTIME	NEXPTIME	AEXP(poly) [14]	2-NEXPTIME

Table 1: Completeness results for common bit-vector logics [16]

[16] proves that QF_BV with binary encoding is NEXPTIME-hard. For this, we picked an NEXPTIME-hard decision problem, the satisfiability problem of *Dependency Quantified Boolean Formulas* DQBF [21, 22], and gave a polynomial reduction from it to QF_BV. The proof cannot be done in a trivial way since the exponential many bits of bit-vectors should be somehow handled in a polynomial way. For this, we need to split the bit-vectors of bit-width 2^n into polynomial many chunks of exponential size. This can be done by applying the following special bitmasks, also known as the binary magic numbers [12]:

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

Such a binary magic number can be calculated as

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

[16, 18] show that that bit-mask can be "generated" by the following bit-vector expression of polynomial size:

$$b^{[2^n]} \ll (1 \ll i) \quad = \quad \sim b^{[2^n]} \tag{4.1}$$

Adding uninterpreted functions to QF_BV results in the logic QF_UFBV. As [16] proves, QF_UFBV has the same complexity as QF_BV, since all uninterpreted functions can be polynomially eliminated by introducing new bit-vector variables and Ackermann constraints.

Regarding quantified bit-vectors, [16] also proves that UFBV with binary encoding is 2-NEXPTIME-hard. The 2-NEXPTIME-hard decision problem we reduced to UFBV was the $2^{(2^n)}$-square tiling problem [6]. The $f(n)$-square tiling problem is about to place dominos on an $f(n) \times f(n)$ board, respecting certain horizontal and vertical matching conditions H and V, respectively. Any instance of the $2^{(2^n)}$-square tiling problem can be expressed as a UFBV formula

$$\lambda(0,0) = 0 \quad \wedge \quad \lambda\Big(2^{(2^n)}-1, 2^{(2^n)}-1\Big) = k-1$$
$$\wedge \bigwedge_{(t_1,t_2)\in H} h(t_1,t_2) \quad \wedge \bigwedge_{(t_1,t_2)\in V} v(t_1,t_2)$$

$$\wedge \quad \forall i^{[2^n]}, j^{[2^n]} \left(\begin{array}{lcl} \Big(j < 2^{(2^n)} - 1 & \Longrightarrow & h\big(\lambda(i,j), \lambda(i,j+1)\big) \Big) \\ & & \wedge \\ \Big(i < 2^{(2^n)} - 1 & \Longrightarrow & v\big(\lambda(i,j), \lambda(i+1,j)\big) \Big) \end{array} \right)$$

The formula contains two universally quantified bit-vector variable, i and j. The uninterpreted function $\lambda(i,j)$ represents the type of the tile in the cell at row index i and column index j. It is crucial to see that although the formula contains exponential bit-widths 2^n, they are encoded as binary numbers, i.e., by using n digits. Furthermore, the double-exponential scalars $2^{(2^n)} - 1$ can be represented as $\sim 0^{[2^n]}$. Thus, we gave a polynomial reduction of the $2^{(2^n)}$-square tiling problem to UFBV.

What happens with complexity if we do not allow the use of uninterpreted functions with quantified bit-vectors, i.e., what is the complexity of BV? It is fairly easy to see that BV is NExpTime-hard and is in ExpSpace, but we have never been able to prove if BV is complete for any of the complexity classes. Finally in 2016, Jonáš and Strejček proved that BV is complete for AEXP(poly) in [14], as shown in Table 1.

4.1. Fragments

After proving the computation complexity of certain logics, the natural question arises: Are there any practically reasonable fragments which have lower complexity?

[17, 18] investigates how the set of bit-vector operators used in formulas affects the computational complexity. We defined three fragments:

QF_BV$_{\ll c}$**:** only bitwise operators, equality, and shift by any constant c are allowed;

QF_BV$_{\ll 1}$**:** only bitwise operators, equality, and shift by $c = 1$ are allowed;

QF_BV$_{\mathsf{bw}}$**:** only bitwise operators and equality are allowed.

We proved all those fragments to be complete for certain complexity classes.

- The NExpTime-completeness of QF_BV$_{\ll c}$ directly follows from the proof of QF_BV being NExpTime-hard in [16], since the reduction we gave in that proof used only bitwise operations, equality and shift by constant.
- It is interesting that restricting shifts to be used only with $c = 1$ causes the complexity to drop to PSpace-completeness, as being proved for QF_BV$_{\ll 1}$ in [17].
- Finally, if no shifts are allowed to use, the resulting fragment QF_BV$_{\mathsf{bw}}$ becomes NP-complete [17].

Our paper [18] investigates possible extensions of the aforementioned fragments and their alternative characterizations. Speaking of $\mathrm{QF\_BV}_{\mathsf{bw}}$, it turns out that the set of bitwise operations and equality can be extended by *indexing* and *relational operations* without pulling out the fragment from NP. In a similar manner, $\mathrm{QF\_BV}_{\ll 1}$ stays in PSPACE even if we extend the set of bitwise operations, equality, and *left shift by 1* with any of the operations in Figure 1. It is even more interesting, as the figure shows, that the operations *right shift by 1*, *addition*, *subtraction*, and *multiplication by constant* can be used as *alternative base operations* instead of *left shift by 1*.

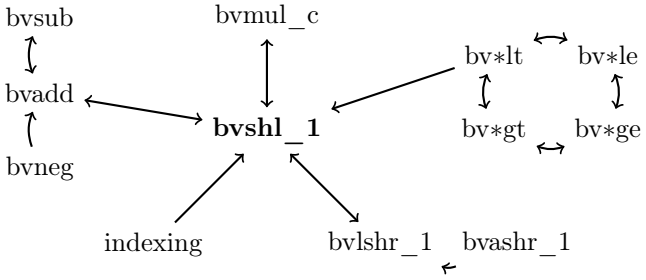

Figure 1: Extending $\mathrm{QF\_BV}_{\ll 1}$ with operations [18]

$\mathrm{QF\_BV}_{\ll c}$ stays in NEXPTIME even if we extend bitwise operations, equality, and *left shift by constant* with any of the operations in Figure 2. Any of *right shift by constant*, *extraction*, *concatenation*, and *multiplication* can serve as an alternative base operation instead of *left shift by constant*.

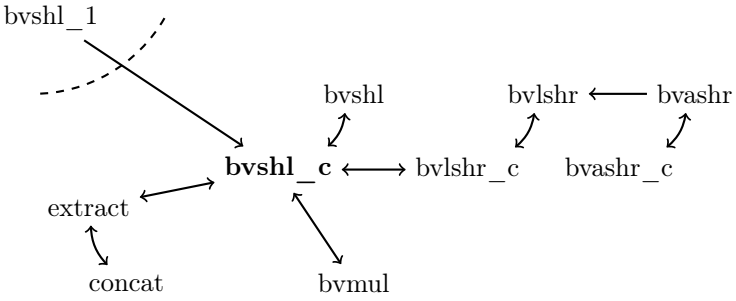

Figure 2: Extending $\mathrm{QF\_BV}_{\ll c}$ with operations [18]

[18] proposes new complexity results for fragments of *quantified* bit-vector logics as well. We already proved in [16] that UFBV is 2-NEXPTIME-complete, therefore the fragment $\mathrm{UFBV}_{\ll c}$, and all its alternative characterizations, have the same complexity. Interestingly, if we restrict shifts to be applied only by 1, the complexity

does not change, as opposed to the quantifier-free case. That is, both $\mathrm{UFBV}_{\ll c}$ and $\mathrm{UFBV}_{\ll 1}$ are 2-NExpTime-complete.

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

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

$\mathrm{QF\_UFBV}_{\mathcal{M}}$**:** In the SMT-LIB, *non-recursive macros* are basic tools. Such a macro provides an uninterpreted function and assigns a functional definition to it. We can formalize a QF_UFBV formula Φ with non-recursive macros as follows:

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

Here, $f_0, \ldots, f_m$ are the macros as uninterpreted functions and $d_0, \ldots, d_m$ are their functional definitions as bit-vector terms. Note that the macros' parameters are universally quantified variables and, therefore, the fragment $\mathrm{QF\_UFBV}_{\mathcal{M}}$ is basically a quantified bit-vector logic. We proved however that using non-recursive macros does not increase the complexity of QF_UFBV, i.e., $\mathrm{QF\_UFBV}_{\mathcal{M}}$ is NExpTime-complete.

5. Summary

We gave reasons and, also, historical evidences to show that the verification of software and hardware systems is of great importance. Bit-precise verification can verify systems on the bit-level and, therefore, is a powerful tool for avoiding low-level bugs such as overflow, arithmetic errors, rounding errors, indexing errors, etc., as well as higher-level ones such as infinite loops or infinite recursion in software. We gave a detailed survey on the computational complexity of satisfiability checking in different bit-vector logics. Furthermore, we summarized what we currently know about practically interesting fragments of those logics and in what extent computational complexity decreases for them, compared to the corresponding full fragments.

References

[1] C. W. Barrett, D. L. Dill, J. R. Levitt, A Decision Procedure for Bit-Vector Arithmetic, *Proc. 35th Design Automation Conference (DAC '98)*, 1998, pp. 522–527.

[2] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Department of Computer Science, The University of Iowa, 2017.

[3] N. Bjørner, M. C. Pichora, Deciding Fixed and Non-fixed Size Bit-vectors, *Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, Springer, Lecture Notes in Computer Science (LNCS), Vol. 1384, 1998, pp. 376–392.

[4] R. Bruttomesso, N. Sharygina, A Scalable Decision Procedure for Fixed-Width Bit-Vectors, *Proc. 2009 International Conference on Computer-Aided Design (ICCAD 2009)*, 2009, pp. 13–20.

[5] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, B. Brady, Deciding Bit-Vector Arithmetic with Abstraction, *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, Springer, Lecture Notes in Computer Science (LNCS), Vol. 4424, 2007, pp. 358–372.

[6] B. S. Chlebus, From Domino Tilings to a New Model of Computation, *Proc. 5th Symposium on Computation Theory*, Springer, Lecture Notes in Computer Science (LNCS), Vol. 208, 1984, pp. 24–33.

[7] S. A. Cook, The Complexity of Theorem-Proving Procedures, *Proc. 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, 1971, pp. 151-158.

[8] D. Cyrluk, O. Möller, H. Ruess, An Efficient Decision Procedure for a Theory of Fixed-Sized Bitvectors with Composition and Extraction, *Proc. 9th International Conference on Computer-Aided Verification (CAV '97)*, Springer, Lecture Notes in Computer Science (LNCS), Vol. 1254, 1997, pp. 60–71.

[9] M. Dowson, The Ariane 5 Software Failure, *ACM SIGSOFT Software Engineering Notes*, Vol. 22(2), 1997, pp. 84.

[10] Q. Duan, S. Al-Haj, E. Al-Shaer, Provable Configuration Planning for Wireless Sensor Networks, *Proc. 8th International Conference on Network and Service Management (CNSM 2012) and Workshop on Systems Virtualization Management (SVM 2012)*, IEEE, 2012, pp. 316–321.

[11] A. Franzén, Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT, PhD thesis, University of Trento, 2010.

[12] E. E. Freed, Binary Magic Numbers – Some Applications and Algorithms, *Dr. Dobb's Journal of Software Tools*, Vol. 8(4), 1983, pp. 24–37.

[13] T. R. Halfhill, The Truth Behind the Pentium Bug, *Byte*, Vol. 20(3), 1995, pp. 163-164.

[14] M. Jonáš, J. Strejček, On the Complexity of the Quantified Bit-Vector Arithmetic with Binary Encoded Bit-Widths, *arXiv preprint*, 2016, arXiv:1612.01263.

[15] R. Kaivola, R: Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S.Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, A. Naik, Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation, *Proc. 21st International Conference on Computer-Aided Verification*

(CAV 2009), Springer, Lecture Notes in Computer Science (LNCS), Vol. 5643, 2009, pp. 414–429.

[16] G. Kovásznai, A. Fröhlich, A. Biere, On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width, *Proc. 10th International Workshop on Satisfiability Modulo Theories (SMT 2012), aff. to IJCAR 2012*, 2012, pp. 44–55.

[17] A. Fröhlich, G. Kovásznai, A. Biere, More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding, *Proc. 8th International Computer Science Symposium in Russia (CSR 2013)*, Springer, Lecture Notes in Computer Science (LNCS), Vol. 7913, 2013, pp. 378–390.

[18] G. Kovásznai, A. Fröhlich, A. Biere, Complexity of Fixed-Size Bit-Vector Logics, *Theory of Computing Systems*, Vol. 59(2), 2016, pp. 323–376.

[19] L. M. de Moura, N. Bjørner, Applications and Challenges in Satisfiability Modulo Theories, *Proc. 2nd International Workshop on Invariant Generation (WING 2009), aff. to IJCAR 2010*, EPiC Series, EasyChair, Vol. 1, 2010, pp. 1–11.

[20] J. Oberg, Why The Mars Probe Went Off Course, *SPECTRUM Magazine*, 1999, `http://sunnyday.mit.edu/accidents/mco-oberg.htm`.

[21] G. Peterson, J. Reif, Multiple-Person Alternation, *Proc. 20th Annual Symposium on Foundations of Computer Science (FOCS 1979)*, IEEE, 1979, pp. 348–363.

[22] G. Peterson, J. Reif, S. Azhar, Lower Bounds for Multiplayer Noncooperative Games of Incomplete Information, *Computers & Mathematics with Applications*, Elsevier, Vol. 41(7), 2001, pp. 957–992.

[23] M. Soos, K. Nohl, C. Castelluccia, Extending SAT Solvers to Cryptographic Problems, *Proc. 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, Springer, Lecture Notes in Computer Science (LNCS), Vol. 5584, pp. 244–257.

[24] C. M. Wintersteiger, Y. Hamadi, L. M. de Moura, Efficiently solving quantified bit-vector formulas, *Proc. 10th Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*, 2010, pp. 239–246.

[25] C. M. Wintersteiger, Termination Analysis for Bit-Vector Programs, ETH Zurich, Switzerland, 2011.

[26] C. M. Wintersteiger, Y. Hamadi, L. M. de Moura, Efficiently solving quantified bit-vector formulas, *Formal Methods in System Design*, Vol. 42(1), 2013, pp. 3–23.