BV2EPR: A Tool for Polynomially Translating Quantifier-Free Bit-Vector Formulas into EPR*

Gergely Kovásznai, Andreas Fröhlich, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. Bit-precise reasoning is essential in many applications of Satisfiability Modulo Theories (SMT). In recent years, efficient approaches for solving fixed-size bit-vector formulas have been developed. Most of these approaches rely on bit-blasting. In [1], we argued that bit-blasting is not polynomial in general, and then showed that solving quantifier-free bit-vector formulas (QF_BV) is NEXPTIME-complete. In this paper, we present a tool based on a new polynomial translation from QF_BV into Effectively Propositional Logic (EPR). This allows us to solve QF_BV problems using EPR solvers and avoids the exponential growth that comes with bit-blasting. Additionally, our tool allows us to easily generate new challenging benchmarks for EPR solvers.

1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for fixed-sized bit-vector logics are Boolector, MathSAT, STP, Z3, and Yices. All these solvers rely on *bit-blasting* in order to translate bit-vector formulas into propositional logic (SAT). The result is then checked by a SAT solver.

In practice, e.g. in the SMT-LIB [2], the BTOR [3], and the Z3 format, the bit-widths in bit-vector formulas are encoded as binary, decimal, or hexadecimal numbers, i.e., a *logarithmic encoding* is used. In [1], we proved that the encoding of bit-widths affects the complexity of the decision problem of bit-vector logics. In particular, logarithmic encoding makes the quantifier-free fragment QF_BV2 NEXPTIME-complete.[1] Thus, bit-blasting is *not polynomial* in general. For a polynomial reduction, the target logic has to be NEXPTIME-hard.

In this paper, we introduce our new tool BV2EPR. BV2EPR translates QF_BV formulas into Effectively Propositional Logic (EPR), which is NEXPTIME-complete [4], by using a new (polynomial) reduction. This is in contrast to existing translations in [5,6], which produce exponential EPR formulas in general, as we will point out in Sect. 2.1. We give some experimental results in Sect. 4 with the EPR solver iProver.

* Supported by FWF, NFN Grant S11408-N23 (RiSE).

[1] In [1], we introduced the notation QF_BV1 resp. QF_BV2 for QF_BV using a *unary* resp. a *logarithmic*, actually without loss of generality, *binary encoding*.

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 443–449, 2013.
© Springer-Verlag Berlin Heidelberg 2013

2 Preliminaries

We assume the usual syntax for QF_BV. A bit-vector term t of bit-width n ($n \in \mathbb{N}$, $n \geqslant 1$) is denoted by $t^{[n]}$. An atomic term can be either (a) a bit-vector *constant* $c^{[n]}$, where $c \in \mathbb{N}$, $0 \leqslant c < 2^n$; or (b) a bit-vector *variable* $v^{[n]}$. Compound terms and formulas can contain the usual bit-vector *operators* (c.f. SMT-LIB [2]), like e.g. bitwise operators, shifts, arithmetic operators, relational operators, etc. The decision problem for QF_BV is NExpTime-complete [1].

EPR, known as the Bernays-Schönfinkel class, is a NExpTime-complete fragment of first-order logic [4]. It corresponds to the set of first-order formulas that, written in prenex form, contain (a) no function symbol of arity greater than 0; and (b) no existential quantifier within the scope of a universal quantifier. After Skolemization, existential variables turn into constants (i.e., function symbols of arity 0), and quantifiers can be omitted. Consequently, an EPR *atom* can be defined as an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and each t_i is either a (universal) variable or a constant.

2.1 Existing Translations

In [5], encodings of hardware verification problems with bit-vectors into first-order logic are proposed. In particular, an encoding into EPR is given and called the *relational encoding* [6], since bit-vectors are modeled as unary predicates. These predicates are over bit-indices, represented by dedicated constants. For instance, the ith bit of a bit-vector $x^{[n]}$, $0 \leqslant i < n$, is represented by the atom $p_x(bitInd_i)$, where $bitInd_i$ is a constant. Note that for QF_BV2, such a translation might introduce exponentially many constants, since bit-widths like n are encoded logarithmically. The so-called *range-aware* relational encoding in [6], furthermore, introduces exponentially many assertions into the EPR formula in general, e.g., atoms $less_k(bitInd_i)$ for all $0 \leqslant i < k$. Finally, not all the QF_BV bit-vector operators are addressed by the relational encoding, but only *equality*[2]. All the *arithmetic operators* are assumed to be synthesized/bit-blasted in the verification front-end [6], potentially leading to an exponential blowup already before the actual encoding. In [5], an abstraction of *shifts* is proposed, which is, however, basically the same as bit-blasting. Consequently, the relational encoding is exponential in general, in constrast with our translation in Sect. 3.1.

3 The Tool

BV2EPR takes a QF_BV formula in SMT2 format as input, and outputs an EPR clause set in TPTP format. The tool is implemented in C and available at [7]. The architecture of BV2EPR can be seen in Fig. 1, consisting of the following modules:

Parser. The Parser is Boolector's SMT2 parser.

[2] Bitwise operators could be handled in a similar way.

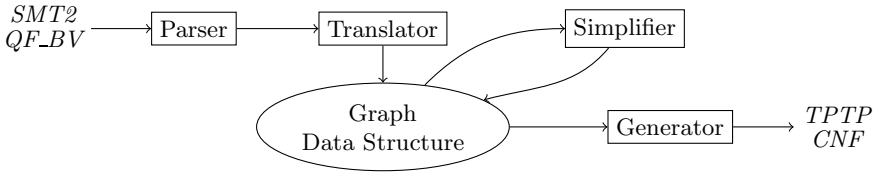

Fig. 1. The architecture of BV2EPR

Translator. The Translator provides an interface accessed by the Parser, in order to deal with the SMT2 QF_BV operators. This module builds a graph data structure, in which each bit-vector operation is modeled by an EPR predicate. Predicates are represented by shared nodes in the graph data structure. A node for a predicate p stores, besides other data, the functional definition of p as an EPR clause set. With each of these clauses, an argument list $i_{n-1}, \ldots, i_0$ for p is stored, indicating that this clause is part of the functional definition of the EPR atom $p(i_{n-1}, \ldots, i_0)$. Such a clause is realized as a list of EPR literals, each of which contains a reference to a predicate p' and an argument list for p'.

Simplifier. The graph constructed by the Translator is a good basis for various simplifications. Note that only polynomial simplification steps are acceptable. Among others, we implemented two kinds of simplification, both proposed in [8]: (a) *unused definition elimination* and (b) *non-growing definition inlining*.

Generator. Out of the (simplified) graph, this module generates a TPTP clause set. Since the graph might contain cycles, the Generator detects and avoids them. Due to the construction of the graph data structure, clauses can be extracted directly, i.e., no additional approach for clause generation is needed.

3.1 The Translator

We briefly sketch the (polynomial) reduction of QF_BV to EPR used by the Translator, without striving for completeness. As it will turn out, the target logic of this reduction is actually not general EPR, but rather its fragment which uses only two constants, 0 and 1. We call this fragment EPR2.[3] To each bit-vector term of bit-width n, a dedicated $\lceil \log_2 n \rceil$-ary EPR2 predicate is introduced and assigned. For example, a term $x^{[32]}$ is represented by a 5-ary predicate p_x. Since p_x is an EPR2 predicate, each of its arguments can be either 0, 1, or a universal variable. For instance, the atom $p_x(1, 1, 0, 0, 1)$ represents the 25th bit of x, since $25_{10} = 11001_2$. Using universal variables as arguments makes it possible to represent several bits by a single EPR2 formula; for instance, the atom $p_x(i_4, i_3, i_2, i_1, 0)$ represents all even bits of x.

Bitwise Operators. Translating bitwise operators is quite natural. We demonstrate the translation for *bitwise or* (denoted by $|$): Given a term $x^{[2^n]} \mid y^{[2^n]}$, where x and y are bit-vector terms, to which the predicates p_x and p_y have already been assigned, respectively. We need to specify each bit of

[3] The Herbrand universe of EPR2 can be considered as the Boolean domain.

the resulting bit-vector as the disjunction of the corresponding bits of x and y. We introduce a new predicate p_{or}, and give the following functional definition:

$$p_{or}(i_{n-1},\ldots,i_0) \;\Leftrightarrow\; p_x(i_{n-1},\ldots,i_0) \vee p_y(i_{n-1},\ldots,i_0)$$

Addition. Given a term $x^{[2^n]} + y^{[2^n]}$, let us first rewrite it to the following bit-vector equations, where $\oplus$ denotes *bitwise xor*, & *bitwise and*, and $\ll$ *left shift*.

$$add^{[2^n]} = x^{[2^n]} \oplus y^{[2^n]} \oplus cin^{[2^n]} \quad (1)$$
$$cin^{[2^n]} = cout^{[2^n]} \ll 1 \quad (2)$$
$$cout^{[2^n]} = (x^{[2^n]} \;\&\; y^{[2^n]}) \mid (x^{[2^n]} \;\&\; cin^{[2^n]}) \mid (y^{[2^n]} \;\&\; cin^{[2^n]}) \quad (3)$$

Note that Eqn. (1) and (3) only contain bitwise operators (and equality). Therefore, both can be translated into EPR2 as introduced previously. Only Eqn. (2), which contains *shift by 1*, has to be handled differently.

We introduce a helper predicate *succ* which will represent the fact that a bit-index j is the successor of a bit-index i, i.e., $j = i + 1$. Since i is represented by an EPR2 argument list $i_{n-1},\ldots,i_0$ and, similarly, j by $j_{n-1},\ldots,j_0$, the $2n$-ary predicate $succ(i_{n-1},\ldots,i_0,\ j_{n-1},\ldots,j_0)$ can be defined by n facts:

$$\begin{gathered} succ(i_{n-1},\ldots,i_3,i_2,i_1,0,\ \ i_{n-1},\ldots,i_3,i_2,i_1,\ 1) \\ succ(i_{n-1},\ldots,i_3,i_2,\ 0,1,\ \ i_{n-1},\ldots,i_3,i_2,\ 1,\ 0) \\ succ(i_{n-1},\ldots,i_3,\ 0,\ 1,1,\ \ i_{n-1},\ldots,i_3,\ 1,\ 0,\ 0) \\ \vdots \\ succ(0,1,\ldots,1,\ 1,0,\ldots,0) \end{gathered}$$

Using this helper predicate, Eqn. (2) can be translated into EPR2 as follows:

$$\begin{gathered} \neg p_{cin}(0,\ldots,0) \\ succ(i_{n-1},\ldots,i_0,\ j_{n-1},\ldots,j_0) \;\Rightarrow\; (p_{cin}(j_{n-1},\ldots,j_0) \;\Leftrightarrow\; p_{cout}(i_{n-1},\ldots,i_0)) \end{gathered}$$

This kind of adder can be adapted to represent other arithmetic operators like *unary minus* and *subtraction*. In BV2EPR, all the relational operators, like *equality* and *unsigned less than*, are also represented by such an adapted adder.

Shifts. Shifts are translated into EPR2 by applying *barrel shift*. For instance, given a term $x^{[2^n]} \ll y^{[2^n]}$, for all bit-indices i, $0 \leqslant i < n$, the ith bit of y is checked: if it is 1, then *left shift by* 2^i has to be done.

$$\begin{gathered} \neg p_y(0,\ldots,0) \;\Rightarrow\; \left(p^0_{shl}(i_{n-1},\ldots,i_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0)\right) \\ \left(\begin{array}{c} p_y(0,\ldots,0) \;\wedge \\ succ(i_{n-1},\ldots,i_0,\ j_{n-1},\ldots,j_0) \end{array}\right) \;\Rightarrow\; \left(p^0_{shl}(j_{n-1},\ldots,j_0) \Leftrightarrow p_x(i_{n-1},\ldots,i_0)\right) \\ \neg p_y(0,\ldots,0,1) \;\Rightarrow\; \left(p^1_{shl}(i_{n-1},\ldots,i_0) \Leftrightarrow p^0_{shl}(i_{n-1},\ldots,i_0)\right) \\ \left(\begin{array}{c} p_y(0,\ldots,0,1) \;\wedge \\ succ(0,i_{n-1},\ldots,i_1,\ 0,j_{n-1},\ldots,j_1) \end{array}\right) \;\Rightarrow\; \left(p^1_{shl}(j_{n-1},\ldots,j_1,i_0) \Leftrightarrow p^0_{shl}(i_{n-1},\ldots,i_0)\right) \\ \vdots \end{gathered}$$

Multiplication. The Translator applies a *shift-and-add* approach for translating a term $x^{[2^n]} \cdot y^{[2^n]}$. We generate 2^n subproducts of bit-width 2^n, and represent all of them by a single $2n$-ary predicate p_{mul}: the ith bit of the jth subproduct is represented by the atom $p_{mul}(j_{n-1}, \ldots, j_0,\ i_{n-1}, \ldots, i_0)$.

First, the $(2^n - 1)$th subproduct is computed, by checking the most significant bit of y: if it is 0, this subproduct is set to 0; otherwise, it is set equal to x.

$$\begin{aligned} \neg p_y(1, \ldots, 1) \ &\Rightarrow\ \neg p_{mul}(1, \ldots, 1,\ i_{n-1}, \ldots, i_0) \\ p_y(1, \ldots, 1) \ &\Rightarrow\ (p_{mul}(1, \ldots, 1,\ i_{n-1}, \ldots, i_0) \Leftrightarrow p_x(i_{n-1}, \ldots, i_0)) \end{aligned}$$

The jth subproduct, $0 \leqslant j < 2^n - 1$, is computed by checking the jth bit of y: if it is 0, then the $(j+1)$th subproduct has to be *shifted left by 1* (represented by the predicate p_{shl}); otherwise, the shifted subproduct and x have to be *added* (represented by p_{add}).

$$\begin{pmatrix} \neg p_y(j_{n-1}, \ldots, j_0) \ \wedge \\ succ(j_{n-1}, \ldots, j_0,\ j'_{n-1}, \ldots, j'_0) \end{pmatrix} \Rightarrow \begin{pmatrix} p_{mul}(j_{n-1}, \ldots, j_0,\ i_{n-1}, \ldots, i_0) \ \Leftrightarrow \\ p_{shl}(j'_{n-1}, \ldots, j'_0,\ i_{n-1}, \ldots, i_0) \end{pmatrix}$$

$$\begin{pmatrix} p_y(j_{n-1}, \ldots, j_0) \ \wedge \\ succ(j_{n-1}, \ldots, j_0,\ j'_{n-1}, \ldots, j'_0) \end{pmatrix} \Rightarrow \begin{pmatrix} p_{mul}(j_{n-1}, \ldots, j_0,\ i_{n-1}, \ldots, i_0) \ \Leftrightarrow \\ p_{add}(j'_{n-1}, \ldots, j'_0,\ i_{n-1}, \ldots, i_0) \end{pmatrix}$$

The final product is given by $p_{mul}(0, \ldots, 0,\ i_{n-1}, \ldots, i_0)$.

Polynomiality and Correctness. All above translation steps are polynomial in the input size since they are polynomial in the number of atoms and logarithmic in their bit-width. Formally showing correctness exceeds the scope of this paper and is part of future work. We also investigated correctness empirically by exhaustively testing consistency of the solving results by Boolector and BV2EPR+iProver, for each bit-vector operation, up to a certain bit-width.

4 Benchmarks and Experiments

Solving QF_BV formulas in general is NEXPTIME-complete [1]. However, certain families of QF_BV formulas are in NP, under certain restrictions on the bit-widths. We called this kind of families *bit-width bounded* [1]. Since solving EPR formulas is NEXPTIME-complete, our translation fits well to families which are *not* bit-width bounded. In [1], two examples of this kind were given: (a) `QF_BV/brummayerbiere3/mulhs`*bw* represents instances of computing the *high-order half of product* problem, parameterized by the bit-width of multiplicands (*bw*); (b) `QF_BV/bruttomesso/lfsr/lfsr`*t_bw_n* formalizes the behaviour of a *linear feedback shift register* [9]. We further propose two new benchmark families that are *not bit-width bounded*: (a) `add2n`*bw* describes how bit-vectors of bit-width 2*bw* can be added by using two adders for bit-vectors of bit-width *bw*. (b) `addmul`*bw* checks, whether the sum of two bit-vectors of bit-width *bw* can differ from their product.

In order to demonstrate the exponential blow-up of bit-blasting, in contrast to our translation into EPR, we used the bit-blaster Synthebtor, part of the

Table 1. Evaluation for the original SMT2 file

bmark	bw	smt2	btor	Boolector	aig	cnf	Lingeling	epr	iProver
mulhs	$8^{\ddagger}$	947	1K	10.3s	3K	44K	9.0s	45K	1m 44s
	$16^{\ddagger}$	959	1K	TO	12K	205K	TO	55K	TO
	$64^{\ddagger}$	982	2K	TO	221K	4M	TO	78K	TO
lfsr_2_bw_16	$63^{\ddagger}$	6K	9K	0.2s	64K	258K	0.7s	56K	18.0s
	$127^{\ddagger}$	7K	9K	1.2s	139K	545K	1.3s	61K	1m 14s
	1023	7K	11K	5.1s	1M	5M	4.7s	74K	TO
	8191	7K	18K	2m 37s	11M	43M	3m 10s	89K	TO
add2n	2^5	452	455	0.0s	3K	25K	0.1s	12K	1m 21s
	2^6	456	671	0.1s	7K	53K	0.7s	13K	TO
	2^{12}	484	8K	3m 5s	549K	4M	1m 28s	21K	TO
addmul	2^7	149	99	0.2s	174K	3M	2.4s	8K	0.1s
	2^9	149	99	2.7s	3M	58M	3m 22s	11K	0.1s
	2^{11}	151	103	TO	48M	1G	TO	13K	0.1s

Table 2. Evaluation for the simplified SMT2 file

bmark	bw	smt2	btor	Boolector	aig	cnf	Lingeling	epr	iProver
mulhs	$8^{\ddagger}$	2K	804	9.8s	3K	42K	8.1s	63K	1m 48s
	$16^{\ddagger}$	2K	956	TO	11K	197K	TO	77K	TO
	$64^{\ddagger}$	2K	1K	TO	215K	4M	TO	110K	TO
lfsr_2_bw_16	$63^{\ddagger}$	126K	59K	0.5s	81K	254K	0.9s	156K	3.0s
	$127^{\ddagger}$	126K	59K	0.6s	174K	540K	1.4s	158K	9.5s
	1023	126K	60K	7.0s	1M	5M	5.1s	165K	9m 21s
	8191	126K	67K	46.1s	13M	43M	TO	173K	TO
add2n	2^5	1K	575	0.0s	4K	25K	0.1s	17K	23.6s
	2^6	1K	671	0.1s	9K	53K	0.7s	18K	5m 0s
	2^{12}	2K	9K	2m 42s	711K	4M	1m 16s	32K	TO
addmul	2^7	239	75	0.2s	174K	3M	2.5s	8K	0.1s
	2^9	239	75	2.8s	3M	58M	1m 40s	11K	0.1s
	2^{11}	241	79	TO	48M	1G	TO	13K	0.1s

Boolector distribution, to generate AIGER files and DIMACS (CNF) files out of BTOR files. Tab. 1 summarizes these results, when *word-level rewriting* in Boolector is switched off. We give the file sizes (in bytes) in all formats and additionally provide the runtimes of Boolector (for SMT2), Lingeling (for CNF), and iProver (for EPR), using a timeout of 10 minutes.

In order to test the effect of word-level rewriting, we added a module to Boolector which reads an SMT2 file, performs rewriting, and outputs the simplified SMT2 file. In Tab. 2, we give the results for the simplified SMT2 files.

‡ Official SMT-LIB benchmarks.

5 Conclusion

We presented BV2EPR, a tool for polynomially translating QF_BV into EPR. The motivation for our tool lies in previous work [1], where we have shown QF_BV to be NEXPTIME-complete. Thus, bit-blasting QF_BV to SAT, as it is usually done in current SMT solvers, results in exponentially larger formulas in general. Previous translations from QF_BV into EPR also apply bit-blasting on certain operators and introduce exponentially many constants resp. constraints in the general case [5,6]. In contrast to this, the Translator used in BV2EPR always produces EPR formulas of polynomial size.

After discussing BV2EPR, we evaluated the size of the formulas produced by our tool and compared it to other commonly used formats. Our results show that the overhead in size is rather small when translating QF_BV into EPR, while all other formats often suffer from exponential blow-up as soon as the bit-widths in the input formula grow larger. However, our results also show that the runtime of iProver on the generated EPR formulas is usually worse compared to the runtime of Boolector on the original QF_BV formula or the one of Lingeling after bit-blasting has been applied. Nevertheless, the evaluation also shows that there exist benchmarks where iProver is faster. While it is probably still possible to improve EPR solvers on this kind of instances, formulas generated by BV2EPR can also help providing challenging benchmarks for current state-of-the-art solvers. The tool BV2EPR is available at [7].

References

1. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: Proc. SMT 2012, pp. 44–55 (2012)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: Proc. SMT 2010, Edinburgh, UK (2010)
3. Brummayer, R., Biere, A., Lonsing, F.: BTOR: bit-precise modelling of word-level problems for model checking. In: BPR 2008, pp. 33–38. ACM, New York (2008)
4. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. 21(3), 317–353 (1980)
5. Khasidashvili, Z., Kinanah, M., Voronkov, A.: Verifying equivalence of memories using a first order logic theorem prover. In: FMCAD 2009, pp. 128–135 (2009)
6. Emmer, M., Khasidashvili, Z., Korovin, K., Voronkov, A.: Encoding industrial hardware verification problems into effectively propositional logic. In: FMCAD 2010, pp. 137–144 (2010)
7. BV2EPR project page, `http://fmv.jku.at/bv2epr/`
8. Hoder, K., Khasidashvili, Z., Korovin, K., Voronkov, A.: Preprocessing techniques for first-order clausification. In: FMCAD 2012, pp. 44–51 (2012)
9. Bruttomesso, R., Sharygina, N.: A scalable decision procedure for fixed-width bit-vectors. In: Proc. ICCAD 2009, pp. 13–20. IEEE (2009)