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

Gergely Kovácsznaí, 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_BV 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.

1 Supported by FWF, NFN Grant S11408-N23 (RiSE).
2 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.
2 Preliminaries

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

EPR, known as the Bernays-Schönfinkel class, is a \( \text{NExpTime} \)-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 \( i \)th bit of a bit-vector \( x^n \), \( 0 \leq i < n \), is represented by the atom \( p_2(bitInd_i) \), where \( bitInd_i \) is a constant. Note that for \( \text{QF}_{\text{BV}2} \), 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 \( \text{less}_k(bitInd_i) \) for all \( 0 \leq i < k \). Finally, not all the \( \text{QF}_{\text{BV}} \) 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 [8], 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

\texttt{bv2epr} takes a \( \text{QF}_{\text{BV}} \) 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 \texttt{bv2epr} can be seen in Fig. 1 consisting of the following modules:

Parser. The Parser is Boolector’s SMT2 parser.

\footnote{Bitwise operators could be handled in a similar way.}
**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 [S]: (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 $[\log_2 n]$-ary EPR2 predicate is introduced and assigned. For example, a term $x^{[32]}$ is represented by a 5-ary predicate $p_x$. Since $p_x$ is an EPR2 predicate, each of its arguments can be either 0, 1, or a universal variable. For instance, the atom $p_x(1,1,0,0,1)$ represents the 25th bit of $x$, since $25_{10} = 110012$. Using universal variables as arguments makes it possible to represent several bits by a single EPR2 formula; for instance, the atom $p_x(i_4, i_3, i_2, i_1, 0)$ represents all even bits of $x$.

**Bitwise Operators.** Translating bitwise operators is quite natural. We demonstrate the translation for *bitwise or* (denoted by $\lor$): Given a term $x^{[2^n]} \lor y^{[2^n]}$, 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 $x$.

\(^3\) 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:

\[
p_{or}(i_{n-1, \ldots, i_0}) \Leftrightarrow p_x(i_{n-1, \ldots, i_0}) \lor p_y(i_{n-1, \ldots, i_0})
\]

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

\[
\begin{align*}
\text{add}^{2^n} &= x^{2^n} \oplus y^{2^n} \oplus \text{cin}^{2^n} \quad (1) \\
\text{cin}^{2^n} &= \text{cout}^{2^n} \ll 1 \\
\text{cout}^{2^n} &= (x^{2^n} \& y^{2^n}) | (x^{2^n} \& \text{cin}^{2^n}) | (y^{2^n} \& \text{cin}^{2^n}) \quad (3)
\end{align*}
\]

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 I*, has to be handled differently.

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

\[
\begin{align*}
\text{succ}(i_{n-1, \ldots, i_3}, i_2, i_1, 0, i_{n-1, \ldots, i_3}, i_2, i_1, 1) \\
\text{succ}(i_{n-1, \ldots, i_3}, i_2, 0, i_{n-1, \ldots, i_3}, i_2, 1, 0) \\
\text{succ}(i_{n-1, \ldots, i_3}, 0, 1, 1, i_{n-1, \ldots, i_3}, 1, 0, 0) \\
\vdots
\end{align*}
\]

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

\[

\begin{align*}
\neg p_{cin}(0, \ldots, 0) \Rightarrow (p_{cin}(j_{n-1, \ldots, j_0}) \Leftrightarrow p_{cout}(i_{n-1, \ldots, i_0}))
\end{align*}

\]

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 \( i, 0 \leq i < n \), the \( i \)th bit of \( y \) is checked: if it is 1, then *left shift by \( 2^i \) has to be done.*

\[
\begin{align*}
\neg p_y(0, \ldots, 0) &\Rightarrow (p_{shl}(i_{n-1, \ldots, i_0}) \Leftrightarrow p_x(i_{n-1, \ldots, i_0})) \\
\text{succ}(i_{n-1, \ldots, i_0}, j_{n-1, \ldots, j_0}) &\Rightarrow (p_{shl}(j_{n-1, \ldots, j_0}) \Leftrightarrow p_x(i_{n-1, \ldots, i_0})) \\
\neg p_y(0, \ldots, 0, 1) &\Rightarrow (p_{shl}(i_{n-1, \ldots, i_0}) \Leftrightarrow p_x(i_{n-1, \ldots, i_0})) \\
\text{succ}(0, i_{n-1, \ldots, i_1}, 0, j_{n-1, \ldots, j_1}) &\Rightarrow (p_{shl}(j_{n-1, \ldots, j_1}, i_0) \Leftrightarrow p_{shl}(i_{n-1, \ldots, i_0})) \\
\vdots
\end{align*}
\]
Multiplication. The Translator applies a shift-and-add approach for translating a term $x^{2^n}y^{2^n}$. We generate $2^n$ subproducts of bit-width $2^n$, and represent all of them by a single $2n$-ary predicate $p_{mul}$: the $i$th bit of the $j$th 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 $x$.

\[
\neg p_y(1, \ldots, 1) \Rightarrow \neg p_{mul}(1, \ldots, 1, i_{n-1}, \ldots, i_0)
\]

\[
p_y(1, \ldots, 1) \Rightarrow (p_{mul}(1, \ldots, 1, i_{n-1}, \ldots, i_0) \Leftrightarrow p_x(i_{n-1}, \ldots, i_0))
\]

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

\[
\begin{align*}
\neg p_y(j_{n-1}, \ldots, j_0) \wedge & \quad \neg p_{shl}(j'_{n-1}, \ldots, j'_0, i_{n-1}, \ldots, i_0) \Rightarrow \neg p_{mul}(j_{n-1}, \ldots, j_0, i_{n-1}, \ldots, i_0) \\
\neg p_y(j_{n-1}, \ldots, j_0) \wedge & \quad \neg p_{add}(j'_{n-1}, \ldots, j'_0, i_{n-1}, \ldots, i_0) \Rightarrow \neg p_{mul}(j_{n-1}, \ldots, j_0, i_{n-1}, \ldots, i_0)
\end{align*}
\]

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/brummayverbiejerem3/mulhsbw represents instances of computing the high-order half of product problem, parameterized by the bit-width of multiplicands (bw); (b) QF_BV/bruttomesso/lfsr/lfsr1bwn formalizes the behaviour of a linear feedback shift register [9]. We further propose two new benchmark families that are not bit-width bounded: (a) add2bw describes how bit-vectors of bit-width $2bw$ can be added by using two adders for bit-vectors of bit-width $bw$. (b) addmulbw checks, whether the sum of two bit-vectors of bit-width $bw$ can differ from their product.

In order to demonstrate the exponential blow-up of bit-blasting, in contrast to our translation into EPR, we used the bit-blaster Synthebtor, part of the
Boolector distribution, to generate AIGER files and DIMACS (CNF) files out of BTOR files. Tab. 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.

---

### Table 1. Evaluation for the original SMT2 file

<table>
<thead>
<tr>
<th>bmark</th>
<th>bw</th>
<th>smt2</th>
<th>btor</th>
<th>Boolector</th>
<th>aig</th>
<th>cnf</th>
<th>Lingeling</th>
<th>epr</th>
<th>iProver</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfsr 2</td>
<td>6k</td>
<td>959</td>
<td>1K</td>
<td>TO</td>
<td>12K</td>
<td>205K</td>
<td>TO</td>
<td>55K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>6k</td>
<td>982</td>
<td>2K</td>
<td>TO</td>
<td>221K</td>
<td>4M</td>
<td>TO</td>
<td>78K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>6k</td>
<td>9K</td>
<td>0.2s</td>
<td>64K</td>
<td>258K</td>
<td>0.7s</td>
<td>56K</td>
<td>18.0s</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>7k</td>
<td>9K</td>
<td>1.2s</td>
<td>139K</td>
<td>545K</td>
<td>1.3s</td>
<td>61K</td>
<td>1m 14s</td>
</tr>
<tr>
<td></td>
<td>1023</td>
<td>7K</td>
<td>11K</td>
<td>5.1s</td>
<td>1M</td>
<td>5M</td>
<td>4.7s</td>
<td>74K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>8191</td>
<td>7K</td>
<td>18K</td>
<td>2m 37s</td>
<td>11M</td>
<td>43M</td>
<td>3m 10s</td>
<td>89K</td>
<td>TO</td>
</tr>
<tr>
<td>mulhs</td>
<td>8</td>
<td>947</td>
<td>1K</td>
<td>10.3s</td>
<td>3K</td>
<td>44K</td>
<td>9.0s</td>
<td>45K</td>
<td>1m 44s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>959</td>
<td>1K</td>
<td>TO</td>
<td>12K</td>
<td>205K</td>
<td>TO</td>
<td>55K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>882</td>
<td>2K</td>
<td>TO</td>
<td>221K</td>
<td>4M</td>
<td>TO</td>
<td>78K</td>
<td>TO</td>
</tr>
<tr>
<td>lfsr 2</td>
<td>6k</td>
<td>959</td>
<td>1K</td>
<td>TO</td>
<td>12K</td>
<td>205K</td>
<td>TO</td>
<td>55K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>6k</td>
<td>982</td>
<td>2K</td>
<td>TO</td>
<td>221K</td>
<td>4M</td>
<td>TO</td>
<td>78K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>6k</td>
<td>9K</td>
<td>0.2s</td>
<td>64K</td>
<td>258K</td>
<td>0.7s</td>
<td>56K</td>
<td>18.0s</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>7k</td>
<td>9K</td>
<td>1.2s</td>
<td>139K</td>
<td>545K</td>
<td>1.3s</td>
<td>61K</td>
<td>1m 14s</td>
</tr>
<tr>
<td></td>
<td>1023</td>
<td>7K</td>
<td>11K</td>
<td>5.1s</td>
<td>1M</td>
<td>5M</td>
<td>4.7s</td>
<td>74K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>8191</td>
<td>7K</td>
<td>18K</td>
<td>2m 37s</td>
<td>11M</td>
<td>43M</td>
<td>3m 10s</td>
<td>89K</td>
<td>TO</td>
</tr>
</tbody>
</table>

### Table 2. Evaluation for the simplified SMT2 file

<table>
<thead>
<tr>
<th>bmark</th>
<th>bw</th>
<th>smt2</th>
<th>btor</th>
<th>Boolector</th>
<th>aig</th>
<th>cnf</th>
<th>Lingeling</th>
<th>epr</th>
<th>iProver</th>
</tr>
</thead>
<tbody>
<tr>
<td>lfsr 2</td>
<td>6k</td>
<td>959</td>
<td>1K</td>
<td>TO</td>
<td>11K</td>
<td>197K</td>
<td>TO</td>
<td>77K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>6k</td>
<td>982</td>
<td>2K</td>
<td>TO</td>
<td>215K</td>
<td>4M</td>
<td>TO</td>
<td>110K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>6k</td>
<td>9K</td>
<td>0.1s</td>
<td>7K</td>
<td>53K</td>
<td>0.7s</td>
<td>13K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>7k</td>
<td>9K</td>
<td>3m 5s</td>
<td>549K</td>
<td>4M</td>
<td>1m 28s</td>
<td>21K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>1023</td>
<td>7K</td>
<td>11K</td>
<td>5.1s</td>
<td>1M</td>
<td>5M</td>
<td>4.7s</td>
<td>74K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>8191</td>
<td>7K</td>
<td>18K</td>
<td>2m 37s</td>
<td>11M</td>
<td>43M</td>
<td>3m 10s</td>
<td>89K</td>
<td>TO</td>
</tr>
<tr>
<td>mulhs</td>
<td>8</td>
<td>947</td>
<td>1K</td>
<td>9.8s</td>
<td>3K</td>
<td>42K</td>
<td>8.1s</td>
<td>63K</td>
<td>1m 48s</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>959</td>
<td>1K</td>
<td>TO</td>
<td>11K</td>
<td>197K</td>
<td>TO</td>
<td>77K</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>882</td>
<td>2K</td>
<td>TO</td>
<td>215K</td>
<td>4M</td>
<td>TO</td>
<td>110K</td>
<td>TO</td>
</tr>
</tbody>
</table>

---

1 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

7. bv2epr project page, [http://fmv.jku.at/bv2epr/](http://fmv.jku.at/bv2epr/)