A Bit-vector to Integer Translation with bv2nat and nat2bv*

Max Barth[1,*], Matthias Heizmann[2]

[1]LMU Munich, Munich, Germany
[2]University of Stuttgart, Stuttgart, Germany

Abstract

In this paper we present a translation from bit-vector formulas to integer formulas. The translation uses the function symbols `bv2nat` and $\texttt{nat2bv}_k$ which are both utilized in the theory of fixed-width bit-vectors of the SMT-LIB [1] language to define the semantics of bit-vector operations. Our translation replaces bit-vector operations with their semantic definition. This facilitates a more modular application as bit-vector operations and their semantic definition have the same sort. As a postprocessing our translation replaces the composition $\texttt{bv2nat} \circ \texttt{nat2bv}_k$ with a modulo operation, and removes redundant modulo operations from the translation result. The evaluation of our translation shows that we are able to solve 9% more tasks, 10% faster and with 23% less memory usage compared to a closely related, up-to-date translation approach. Additionally, our translation supports the translation of quantified formulas and arrays over bit-vectors.

Keywords

Int-blasting, Bit-vectors, Translation from bit-vectors to integers, bv2nat and nat2bv, Translation of quantified formulas and arrays

1. Introduction

In many program languages, integer data types represent only a fixed number of values. A sequence of bits is utilized to represent an integer via two's complement or a binary encoding. We call such a sequence of bits a *bit-vector*. The SMT-LIB [1] theory of "FixedSizeBitVectors"[1] is well-suited for modeling such programming languages since it offers many function symbols that capture precisely the semantics operations that occur in programming languages.

However, the expressiveness of the bit-vector theory comes at a certain price. Due to their complex semantics, formulas of this theory are rather intractable and there are only few algorithms that handle these formulas directly. Typically, algorithms that work on bit-vector formulas first do a translation, either to propositional logic or to integer arithmetic. The translation to propositional logic is called bit-blasting [2, 3]. Bit-blasting translates each bit of a bit-vector into a propositional logical variable and translates each bit-vector operation into a propositional logical formula that captures exactly the semantics of that operation. The strength of bit-blasting is that we can utilize powerful SAT solvers for deciding statisfiability of the resulting formulas. The alternative to bit-blasting is the translation to integer arithmetic. A recent publication coined the term int-blasting [4] for this translation. Here, bit-vector variables are translated to integer variables and a comprehensive application of modulo operations makes sure that we can establish a connection between models for the bit-vector formulas and models for the integer formulas. In order to model bit-precise bit-vector operations, int-blasting can access individual bits via a combination of integer division (`div`) and modulo (`mod`) operations. E.g., if we translate a bit-vector variable x into an integer variable x' such that x is the binary encoding of x', we can access the third least-significant bit as follows: We divide x' by 4 and take the result modulo two. The result is one iff this bit was set to true. In order to improve the performance, int-blasting-based translations often work with approximations and increase their precision later if required.

SMT'24: 22st International Workshop on Satisfiability Modulo Theories, July 22–23, 2024, Montreal, Canada
*This project was supported by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY).
*Corresponding author.
0009-0002-7716-3898 (M. Barth); 0000-0003-4252-3558 (M. Heizmann)

[1]https://smt-lib.org/theories-FixedSizeBitVectors.shtml

Even though the state of the art to decide the satisfiability of bit-vector formulas is bit-blasting [2], there exist applications that need a translation from bit-vector formulas to formulas over integers. For example in software model checking many techniques only work on integers: loop acceleration [5, 6], invariant syntheses [7, 8] and syntheses of ranking functions [9, 10, 11]. While other techniques perform better on integers, especially Craig interpolation.

In this paper we present a translation from bit-vectors to integers that is closely related to the translation of [4]. In order to explain our translation and conceptual differences to [4] we use the functions $\texttt{nat2bv}_k$ and $\texttt{bv2nat}$ (see Section 3) which implement the binary encoding of natural numbers and its inverse function, respectively. The translation of [4] ensures the following relation between models of bit-vector formulas and the models of the resulting integer formulas: If the bit-vector model maps a term t to a bitvector value v then the integer model maps the translation result of t to $\texttt{bv2nat}(v)$. In order to ensure this relation, the translation of [4] adds constraints that require that integer variables are in a given range and modulo operations such that every integer term is in the same range as its corresponding bit-vector term.

Our translation only requires a less strict relation between models of bit-vector formulas and the models of the resulting integer formulas. If the integer model maps the translation result of the term t to an integer v' then the bit-vector model maps t to $\texttt{nat2bv}(v')$. I.e., we do not require that $\texttt{nat2bv}(v')$ is the binary encoding of v', we only require that $\texttt{nat2bv}(v')$ is the binary encoding of $\texttt{nat2bv}(v' \bmod k)$, where k is the bit-length of t. This conceptual difference to [4] allows us to omit constraints on the translated variables and it allows us to omit several modulo operations that the translation of [4] introduces.

The translation of [4] introduces modulo operations *after* each arithmetic operation. Our translation omits modulo operations after arithmetic operations but introduces modulo operations *before* each operation where we need that integer values are in a certain range. E.g., we translate $\texttt{bvult}$ to the less-than relation $<$ but apply a modulo operation to both operands. We call the translation of [4] *eager* int-blasting and our translation *lazy* int-blasting.

Example 1. For the bit-vector formula: $(\texttt{bvult}\ y\ (\texttt{bvadd}\ 0101\ (\texttt{bvmul}\ y\ 0011)))$. The result of the eager int-blasting is: $(<\ y'\ (\bmod\ (+\ 5\ (\bmod\ (\cdot\ y'\ 3)\ 2^4))\ 2^4)) \wedge (\geq\ y'\ 0) \wedge (<\ y'\ 2^4)$. The result of our lazy int-blasting is: $(<\ (\bmod\ y'\ 2^4)\ (\bmod\ (+\ 5\ (\cdot\ y'\ 3))\ 2^4))$.

Technically, our translation proceeds in two steps. In the first step, we inductively translate the formula and distinguish two cases. If the SMT-LIB semantic definition of the operation that we have to translate utilizes the functions $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$, we replace the operation with their semantic definition. For most of the other operations our translation is similar to the eager int-blasting. The result of our first translation step contains the functions $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$ only as a concatenation $\texttt{bv2nat}\circ\texttt{nat2bv}_k$. Our second translation step replaces this concatenation either by a modulo operation or by the identity function.

An additional contribution of this paper is that we present how we translate quantified formulas and arrays over bit-vectors into integers.

We have implemented our translation in two ways: as a wrapper script [12] for SMT solver and directly in the SMT solver SMTInterpol [13]. Through our evaluation of both implementations, we demonstrated that our translation does not produce incorrect results on the benchmark set. Moreover, we conducted a comparison between two settings of our translation implemented in SMTInterpol: one setting adds modulo operations lazily, while the other adds them eagerly. Our evaluation results revealed that SMTInterpol is capable of solving 9% more tasks, is 10% faster, and requires 23% less memory when modulo operations are added lazily compared to eagerly.

2. Related Work

There have been numerous approaches to translating from bit-vectors to integers. Some methods perform the translation during the verification process e.g. on the source level [14] or during invariant synthesis [15]. More closely related to our work are translations for SMT formulas [16, 17, 18, 19, 4].

However, our translation approach differs from the related work. Firstly, we incorporate modular arithmetic lazily to eliminate redundant modulo operations in the translation result. There exist simplification techniques to eliminate such redundant modulo operations. For example when the SMT solver Z3 is asked $\texttt{simplify}(\texttt{mod}\ (\cdot\ (\texttt{mod}\ (+\ x\ y)\ 256)\ z)\ 256)$, it returns $(\texttt{mod}\ (\cdot\ z\ (+\ x\ y))\ 256)$. However, since we do not add the redundant modulo operations in the first place, there is no need for such a simplification on our translation results. Secondly, no other approach utilizes function symbols with behavior similar to the functions $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$. We give a brief overview of the related approaches.

A similar concept to our lazy translation, but in a different research field, can be found in [20]. Where the authors apply modulo operations lazily during their translation from BTOR to C. Their evaluation shows that their lazy translation is faster in general, but not strictly better than their eager translation.

Griggio et al. introduces a layered SMT solver in [16], where on the higher layers, the solver performs translation from bit-vector formulas to integer formulas. This translation is for a fragment of the bit-vector functions, including arithmetic functions, extraction, concatenation, left shift, and relations. Rather than incorporating modulo operations to bound the integers, Griggio's approach utilizes an auxiliary variable, such as variable v in the expression $(t_1 + t_2 - 2^n \cdot v) \wedge (0 \leq v) \wedge (v \leq 1)$. Each integer term derived during the translation is within the bounds of its corresponding bit-vector term, effectively achieving an eager translation.

Backeman, Rummer, and Zeljic [18] introduce a new calculus for non-linear integer arithmetic, which, in certain cases, can eliminate quantifiers and extract Craig interpolants. Subsequently, they define a corresponding calculus for arithmetic bit-vector constraints. Both calculi allow for a flexible switch between bit-vectors and integers. Initially, integers are not bound by modular arithmetic; instead, the authors introduce an uninterpreted function symbol that represents the modulo operation. They note that the remainder operation tends to be a bottleneck for interpolation. If necessary, a definition for the uninterpreted function symbol can be added to precisely cover the remainder. In contrast to our approach with uninterpreted function symbols, their uninterpreted function is directly associated with the modulo operation and does not affect the sort of a term. Furthermore, their approach supports a translation of quantified formulas, but not of arrays over bit-vectors.

Recently, the first precise and complete translation for bit-vector formulas with bit-vectors of fixed size was published in [4]. This work is closely related to their translation for bit-vectors with parametric bit-width, proposed in their previous work [19]. The precise translation presented in [4] has been implemented in the cvc5 prover [21]. It employs a translation from bit-vector formulas to non-linear integer arithmetic formulas with uninterpreted functions and universal quantification. During the translation, modulo operations are added eagerly and without the use of uninterpreted function symbols. Our translation approach combines elements from the translation in [4] with the semantics of the theory of "FixedSizeBitVectors" defined in the SMT-LIB [1].

3. Preliminaries and Notations

The SMT-LIB [1] defines a many-sorted first-order logic with equality. In this paper we use the sorts, signatures Σ and theories defined in the SMT-LIB. In particular we use the sorts, signatures and theories of Booleans, fixed-size bit-vectors, integers, and arrays.

Bit-Vectors

The SMT-LIB defines a signature Σ_{Bv} and a theory called "FixedSizeBitVectors" for bit-vectors of fixed size. For every possible bit-vector size k, that is every positive integer greater than zero, Σ_{Bv} contains a unique sort σ_k. We call a term of sort $\sigma_k \in \Sigma_{\mathsf{Bv}}$ bit-vector of size k or bit-vector of width k. In the following let x be a bit-vector variable, c be a bit-vector constant, and t, t_1, and t_2 be bit-vector terms. Furthermore, let the width of bit-vectors x, c and t be k, the width of t_1 be k_1 and the width of t_2 be k_2. The signature Σ_{Bv} as defined in the SMT-LIB contains a set of bit-vector function symbols. We denote the extract function from i to j as $\texttt{extract}^i_j(t)$, where i and j are natural numbers with

$i, j \geq 0 \wedge i \geq j$. For the other bit-vector functions f we use the notation $f(t_1\ t_2)$ instead of the SMT-LIB notation $(f\ t_1\ t_2)$.

Integers

The SMT-LIB defines a signature Σ_{Int} and a Σ_{Int}-theory for mathematical integers. The sort $\sigma \in \Sigma_{\mathsf{Int}}$ is defined as the set of all integers. The integer signature Σ_{Int} consists of variables, constants and the usual functions and relations. Let constants c' and terms t', t'_1 and t'_2 all have sort Int. As notation for integer functions we write $(t'_1 + t'_2)$ instead of the SMT-LIB notation $(+\ t'_1\ t'_2)$.

In [4] the authors introduce a binary function symbols $\&_k(-,-)$, for every positive integer k. The functions $\&_k(-,-)$ are introduced to represents bit-wise *and*. Therefore, they extend the signature Σ_{Int} and define two theories for this extended signature. We will do the same in this paper and refer to [4] for details.

Functions `bv2nat` and $\texttt{nat2bv}_k$

In the theory of "FixedSizeBitVectors" the functions $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$ are defined. Given an arbitrary binary $b = (b_{k-1}, ..., b_i, ..., b_0)$ and its corresponding natural number n, the function $\texttt{bv2nat}$ is defined as follows: $\texttt{bv2nat}(b) := \sum_{i=0}^{k-1} b_i \cdot 2^i$. Furthermore, the function $\texttt{nat2bv}_k$ is defined as: $\texttt{nat2bv}_k(n) := (b_{k-1}, ..., b_i, ..., b_0)$, where $b_i = n \texttt{ div } 2^i \texttt{ mod } 2$.

Note, we do not extend our signatures and theories with $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$. Instead, we treat them as auxiliary functions and ensure they are eliminated in the translation result. For the sake of readability, we denote $\texttt{bv2nat}(t)$ and $\texttt{nat2bv}_k(t')$ as $t^{\texttt{bv2nat}}$ and $t'^{\texttt{nat2bv}_k}$, respectively.

4. Translation with `bv2nat` and $\texttt{nat2bv}_k$

Our bit-vector to integer translation maps from the set of Σ_{Bv}-formulas to the set of Σ_{Int}-formulas (extended with $\&_k(-,-)$). We say a translation "translates" a term or formula if it associates that term or formula with an element of its co-domain. Before we can define the translation, we need to define some auxiliary functions. First we define a variable mapping χ. Similar to the variable mapping defined in [4], χ maps a variable of sort bit-vector to a fresh variable of sort integer. We extend the definition of χ for arrays over bit-vectors and quantified variables.

Definition 4.1 (Variable Mapping χ)**.** Given a bit-vector formula ϕ, we define a one-to-one mapping χ as the following. For every variable and quantified variable x that occurs in ϕ, χ maps to a fresh variable x', such that if x is of sort Bv, then x' is of sort Int. If x is of sort Array with arguments of sort $s \in \{\mathsf{Bv}, \mathsf{Array}\}$ then x' is of sort Array with arguments of sort $s' \in \{\mathsf{Int}, \mathsf{Array}\}$ correspondingly. Finally, if x is of sort Bool, then x' is of sort Bool. We write χ maps x to x' as $\chi(x) = x'$.

In Table 1 we use the auxiliary function $uts_k(-)$ from [4]. For a bit-vector term t', $uts_k(t')$ is an abbreviation for the term $2 \cdot (t' \texttt{ mod } 2^{k-1})$, which transforms an unsigned bit-vector to a signed bit-vector. Initially, our translation interprets every bit-vector as unsigned. In the case of a signed relation, we enclose the arguments of the relation with the function uts_k. to ensures that the semantics of signed relation is preserved properly.

Finally, we define the translation function T that maps from Σ_{Bv}-formulas ϕ of the theory of bit-vectors to Σ_{Int}-formulas ψ of the theory of integers (extended with $\&_k(-,-)$). Therefore, we define a conversion functions C in Table 1 (column Lazy) and a replacement function R in Table 3. The translation function T is defined as:

$$T := \phi \mapsto R(C(\phi))$$

Our translation consist of two steps. In step one C replaces bit-vector formulas and terms by their semantic definition. The conversion functions C matches a term or formula e to a term or formula

in the first column. The match is then translated to the term or formula in the middle column named Lazy. For a direct comparison, we display the translation steps from the Eager translation in [4] in the third column in gray. Note, in [4] the authors translate a bit-vector variable v by adding constraints in the form of $(0 \leq \chi(v) < 2^k)$ to the translation result and do not surround the integer variable $\chi(v)$ with modulo as we do in the third column of Table 2. For readability reasons, we split the definition of function C into three functions: C, C_t and C'_t. Functions C_t and C'_t are both defined in Table 2. The conversion function uses $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$ to replace bit-vector formulas and terms by their semantic definition, but our signature Σ_{Int} and integer theory do not contain $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$. In the second step, we use the replacement function R to get rid of $\texttt{bv2nat}$ and $\texttt{nat2bv}_k$. Therefore, we either remove the concatenation $\texttt{bv2nat} \circ \texttt{nat2bv}_k$ or replace it with a modulo operation. In the tables defining C, C_t, and C'_t, we have indicated certain $\texttt{bv2nat}$ function calls in blue. If function $R(e)$ matches e to $(t'^{\texttt{nat2bv}_k})^{\texttt{bv2nat}}$ where $\texttt{bv2nat}$ is marked blue, then we replace e with $t' \bmod 2^k$. Otherwise, if $\texttt{bv2nat}$ is not marked blue we replace $(t'^{\texttt{nat2bv}_k})^{\texttt{bv2nat}}$ with t'.

So far we translate $\texttt{bvand}$ to the uninterpreted function symbol $\&_k(-,-)$ in C and do not treat it any further. For a more sophisticated translation of $\texttt{bvand}$ we refer to literature [4].

	Lazy	Eager
$C(e)$:		
$Match\, e$:		
$t_1 = t_2$	$C_t(t_1)^{\texttt{bv2nat}} = C_t(t_2)^{\texttt{bv2nat}}$	$C_t(t_1) = C_t(t_2)$
$\texttt{bvult}(t_1, t_2)$	$C_t(t_1)^{\texttt{bv2nat}} < C_t(t_2)^{\texttt{bv2nat}}$	$C_t(t_1) < C_t(t_2)$
$\texttt{bvule}(t_1, t_2)$	$C_t(t_1)^{\texttt{bv2nat}} \leq C_t(t_2)^{\texttt{bv2nat}}$	$C_t(t_1) \leq C_t(t_2)$
$\texttt{bvslt}(t_1, t_2)$	$uts_k(C_t(t_1)^{\texttt{bv2nat}}) < uts_k(C_t(t_2)^{\texttt{bv2nat}})$	$uts_k(C_t(t_1)) < uts_k C_t(t_2))$
$\texttt{bvsle}(t_1, t_2)$	$uts_k(C_t(t_1)^{\texttt{bv2nat}}) \leq uts_k(C_t(t_2)^{\texttt{bv2nat}})$	$uts_k(C_t(t_1)) \leq uts_k C_t(t_2))$
$\square(t_1, ...t_i)$	$\square(C(t_1), ..., C(t_i)) \; \square \in \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$	
$uts_k(t') := \; 2 \cdot (t' \bmod 2^{k-1}) - t'$		

Table 1
Definition of the Conversion Function C

4.1. Translation of Arrays and Quantified Formulas

For the translation of quantified formulas and arrays we extend our conversion functions C and C_t with the conversions outlined in Table 4. Therefore, let in Table 4 a and b be arrays over bit-vectors, where the bit-vectors that represent the indices of a and b have width k. Additionally, let i' be a quantified integer variable. The translation of quantified formulas utilizes the variable mapping $\chi(v) = v'$, where v is a quantified bit-vector variable of width k and v' is a quantified integer variable. Additionally, we ensure that a translated quantified formula is unsatisfiable for values of v' that are not within the bounds of v. Therefore, we add the bound $(0 \leq \chi(v) < 2^k)$ as constraints within the scope of the quantifier.

Arrays over bit-vectors a are translated to fresh arrays over integers with the help of the one-to-one mapping $\chi(a)$. Since arrays over integers have infinitely indices and arrays over bit-vectors don't, we have to add some limitations. First of all, we ensure to only read from and write to indices that are within the bounds of the corresponding bit-vector array. Secondly, we have to ensure that if two translated arrays are equal on every index in the range of the bit-vector array, then they are equal on every index. Therefore, we add a constraint to the translation result that evaluates to false if this condition is violated. This is achieved by the constraint function Lem in Table 4. Furthermore, we change the translation function to add constraints for every array equality: $T_{Array} := \phi \mapsto R(C(\phi)) \wedge \textsc{Lem}(\phi)$.

	Lazy	Eager
$C_t(e)$:		
$Match\, e$:		
$\texttt{concat}(t_1, t_2)$	$C'_t(e)^{\texttt{nat2bv}_{k_1+k_2}}$	$C'_t(e)$
$\texttt{extract}^i_j(t_1)$	$C'_t(e)^{\texttt{nat2bv}_{i-j+1}}$	$C'_t(e)$
else: e	$C'_t(e)^{\texttt{nat2bv}_k}$	$C'_t(e)$
$C'_t(e)$:		
$Match\, e$:		
x	$\chi(x)$	$\chi(x) \bmod 2^k$
c	$\sum_{i=0}^{k-1} c_i \cdot 2^i$ where c_i is the i-th bit of c	
$\texttt{bvneg}(t_1)$	$2^{k_1} - C_t(t_1)^{\texttt{bv2nat}}$	$2^{k_1} - C_t(t_1)$
$\texttt{bvmul}(t_1, t_2)$	$C_t(t_1)^{\texttt{bv2nat}} \cdot C_t(t_2)^{\texttt{bv2nat}}$	$(C_t(t_1) \cdot C_t(t_2)) \bmod 2^k$
$\texttt{bvadd}(t_1, t_2)$	$C_t(t_1)^{\texttt{bv2nat}} + C_t(t_2)^{\texttt{bv2nat}}$	$(C_t(t_1) + C_t(t_2)) \bmod 2^k$
$\texttt{bvsub}(t_1, t_2)$	$C_t(t_1)^{\texttt{bv2nat}} - C_t(t_2)^{\texttt{bv2nat}}$	$(C_t(t_1) - C_t(t_2)) \bmod 2^k$
$\texttt{bvudiv}(t_1, t_2)$	$\texttt{ite}(C_t(t_2)^{\texttt{bv2nat}} = 0,\ 2^k - 1,$ $C_t(t_1)^{\texttt{bv2nat}} \texttt{ div } C_t(t_2)^{\texttt{bv2nat}})$	$\texttt{ite}(C_t(t_2) = 0, 2^k - 1,$ $C_t(t_1) \texttt{ div } C_t(t_2))$
$\texttt{bvurem}(t_1, t_2)$	$\texttt{ite}(C_t(t_2)^{\texttt{bv2nat}} = 0,\ t_1,$ $C_t(t_1)^{\texttt{bv2nat}} \bmod C_t(t_2)^{\texttt{bv2nat}})$	$\texttt{ite}(C_t(t_2) = 0, C_t(t_1),$ $C_t(t_1) \bmod C_t(t_2))$
$\texttt{bvshl}(t_1, t_2)$	$\texttt{ite}(C_t(t_2)^{\texttt{bv2nat}} = 1,$ $2 \cdot C_t(t_1)^{\texttt{bv2nat}},$ ... $\texttt{ite}(C_t(t_2)^{\texttt{bv2nat}} = k-1,$ $2^{k-1} \cdot C_t(t_1)^{\texttt{bv2nat}},$ $0)...)$	$\texttt{ite}(C_t(t_2) = 1,$ $2 \cdot C_t(t_1) \bmod 2^k,$ ... $\texttt{ite}(C_t(t_2) = k-1,$ $2^{k-1} \cdot C_t(t_1) \bmod 2^k,$ $0)...)$
$\texttt{bvlshr}(t_1, t_2)$	$\texttt{ite}(C_t(t_2)^{\texttt{bv2nat}} = 1,$ $C_t(t_1)^{\texttt{bv2nat}} \texttt{div } 2,$ ... $\texttt{ite}(C_t(t_2)^{\texttt{bv2nat}} = k-1,$ $(C_t(t_1)^{\texttt{bv2nat}} \texttt{div } 2^{k-1}),$ $0)...)$	$\texttt{ite}(C_t(t_2) = 1,$ $C_t(t_1) \texttt{ div } 2,$ ... $\texttt{ite}((C_t(t_2) = k-1),$ $C_t(t_1) \texttt{ div } 2^{k-1},$ $0)...)$
$\texttt{concat}(t_1, t_2)$	$C_t(t_1)^{\texttt{bv2nat}} \cdot 2^{k_2} + C_t(t_2)^{\texttt{bv2nat}}$	$C_t(t_1) \cdot 2^{k_2} + C_t(t_2)$
$\texttt{extract}^i_j(t_1)$	$C_t(t_1)^{\texttt{bv2nat}} \texttt{ div } 2^j$	$C_t(t_1) \texttt{ div } 2^j \bmod 2^{i-j+1}$
$\texttt{bvnot}(t_1)$	$2^{k_1} - C_t(t_1)^{\texttt{bv2nat}} + 1$	$2^{k_1} - C_t(t_1) + 1$
$\texttt{bvand}(t_1, t_2)$	$\&_k(C_t(t_1)^{\texttt{bv2nat}}, C_t(t_2)^{\texttt{bv2nat}})$	$\&_k(C_t(t_1), C_t(t_2))$

Table 2
Definition of the Conversion Function C_t

$R(e)$:			
$Match\, e$:			
$(t'^{\texttt{nat2bv}_k})^{\texttt{bv2nat}}$	$\rightarrow$	$R(t') \bmod 2^k$	where bv2nat is marked blue
$(t'^{\texttt{nat2bv}_k})^{\texttt{bv2nat}}$	$\rightarrow$	$R(t')$	otherwise
else: $e(t'_1, ..., t'_n)$	$\rightarrow$	$e(R(t'_i), ..., R(t'_n))$	

Table 3
Definition of the Replacement Function R

	Lazy
$C(e)$: $Match\, e$: $\exists\, v.(e)$ $\forall\, v.(e)$	 $\exists\, C(v).(C(e) \wedge\ (0 \leq \chi(v) < 2^k))$ $\forall\, C(v).((0 \leq \chi(v) < 2^k)\ \Rightarrow C(e))$
$C_t(e)$: $Match\, e$: a $(\texttt{select}\ a\ i)$ $(\texttt{store}\ a\ i\ v)$	 $\chi(a)$ $(\texttt{select}\ C_t(a)\ C_t(i)^{\texttt{bv2nat}})$ $(\texttt{store}\ C_t(a)\ C_t(i)^{\texttt{bv2nat}}\ C_t(v))$
$\textsc{Lem}(e)$: $Match\, e$: $a = b$ $\square(t_1, ...t_i)$ else:	 $(\forall i'.(0 \leq i' < 2^k) \Rightarrow$ $((\texttt{select}\ C_t(a)\ i')\ \texttt{mod}\ 2^k =$ $(\texttt{select}\ C_t(b)\ i')\ \texttt{mod}\ 2^k))$ $\Rightarrow C_t(a) = C_t(b)$ $\bigwedge_{i=1}^{n} \textsc{Lem}(t_i) \qquad \square \in \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$ $\top$

Table 4
Extension for Quantified Formulas and Arrays

5. Implementation

We have two implementations for our lazy translation presented in Section 4. The first implementation is a wrapper script for SMT solvers, called Ultimate IntBlastingWrapper [12]. It is implemented in the Ultimate framework[2]. The wrapper script does not use the functions `bv2nat` and $\texttt{nat2bv}_k$ instead modulo operations are added directly. It supports a translation of bit-wise operations with all features described in [4], quantified formulas and arrays over bit-vectors.

The second implementation is in the SMT solver SMTInterpol[3]. This implementation is still work in progress. So far we use `bv2nat` and $\texttt{nat2bv}_k$ as uninterpreted functions, but the implementation does not support bit-wise operations, quantified variables and arrays yet. We implemented two settings to translate bit-vectors to integers. The first setting is called Lazy and it applies our lazy translation in Section 4. The second setting is called Eager and it applies a translation similar to [4]. Each setting applies the conversions in their respective column in Table 2. Additionally, we compare our settings Lazy and Eager with the original implementation (cvc5-int) of [4] in the SMT solver cvc5. We selected SMTInterpol as the SMT solver for our evaluation because, at the point of writing, it did not support bit-vectors, and we were already familiar with the tool. Unfortunately, it did not support non-linear integer arithmetic either. When SMTInterpol encounters a bit-wise operation or non-linear integer arithmetic we return an error.

6. Evaluation

We evaluate our two implementations to answer the following research questions:

- How does the performance of the approach Lazy and Eager compare?
- How does Lazy and Eager compare to int-blasting in cvc5?

6.1. Evaluation of Ultimate IntBlastingWrapper

We participated with Ultimate IntBlastingWrapper [12] at the latest SMT-COMP 2023. Ultimate IntBlastingWrapper competed in the Single Query Track on every logic that contains bit-

[2] https://ultimate.informatik.uni-freiburg.de and github.com/ultimate-pa/ultimate
[3] https://github.com/ultimate-pa/smtinterpol/tree/Intblasting

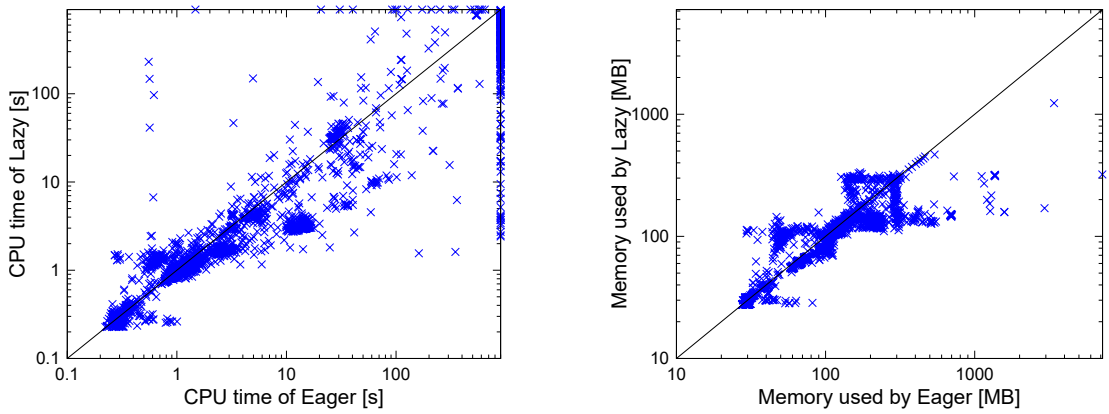

Figure 1: Comparing CPU time and Memory usage of Eager (x-axis) and Lazy (y-axis)

vectors. The results of Ultimate IntBlastingWrapper in the SMT-COMP 2023 showed wrong results on three benchmarks. Of these three, two are from the category QF_AUFBV and one from QF_ABV. The wrong results were caused by a mistake in the translation of equalities between arrays over bit-vectors. This has been fixed in commit: https://github.com/ultimate-pa/ultimate/commit/928447c7dc8c44e406f0a52121ccf96fcbe4d5b5.

6.2. Evaluation of Int-blasting in SMTInterpol

To evaluate our implementation in the SMTInterpol, we ran the settings Lazy and Eager and the implementation of int-blasting in cvc5 (cvc5-int) from the paper [4]. For cvc5-int we used the cvc5 options *--solve-bv-as-int=sum* and *--nl-ext-tplanes*. We run Lazy, Eager and cvc5-int on a randomly picked subset of the non-incremental QF_BV benchmarks from the SMT-LIB. From the set we excluded every benchmark where the expected result is *unknown*. There remained 12302 benchmarks.

Environment

We run our experiments on a cluster of machines with 33 GB of memory and an Intel Xeon E3-1230 v5 CPU with 8 processing units and a frequency of 3.40 GHz that run Ubuntu 22.04 (Linux kernel 5.15.0). To measure and limit resources we use Benchexec 3.18 [22]. We limit every run to 1 core, 15 min of CPU time and 15 GB of memory.

How does the performance of the approach Lazy and Eager compare?

	Lazy	Eager	cvc5-int
Correct	2961	2698	8409
SAT	662	425	2135
UNSAT	2299	2273	6274
Timeout	2124	2709	3769
Unsupported	7217	6895	-
Memory Out	-	-	124
Total	12302		

Table 5
Overview of the evaluation results

	Lazy	Eager
Correct	2679	
∑CPU	23000 s	25600 s
∑Memory	252 GB	327 GB

Table 6
Benchmarks where Lazy and Eager are correct

The evaluation results of Lazy and Eager are displayed in Table 5. We can see that SMTInterpol solves more benchmarks with Lazy than with Eager. Lazy creates 282 more correct results that is 9% (2698 of 2961). Among these, Eager times out on 280 cases, and on 2 cases, Eager reports that the formula contains non-linear integer arithmetic. On the other hand Eager returns a correct result on

19 benchmarks where Lazy times out. Furthermore, on 426 benchmarks Lazy returns an error where Eager times out. All errors in Table 5 are caused by either non-linear integer arithmetic or a bit-wise operation. To compare the CPU time and memory usage of Lazy and Eager, we analyze the 2679 benchmarks for which both settings return a correct result (see Table 6). When measuring the CPU time and memory used on these 2679 benchmarks, Lazy requires 10% less CPU time (23000 s out of 25600 s) and 23% less memory (252 GB out of 327 GB) to decide their satisfiability. For a more detailed view, we provide two scatter plots in Figure 1. Both scatter plots have logarithmic scales, the first shows the CPU time used and the second plot shows the memory usage. Every dot in the scatter plots below the line is in favor of Lazy. We observe that in many cases, the Lazy approach requires less CPU time and/or memory. Specifically, the CPU time and memory usage of Lazy tends to deviate less frequently and significantly from Eager. However, it is important to note that Lazy is not strictly superior to Eager.

How does Lazy and Eager compare to int-blasting in cvc5?

In Table 5, we observe that cvc5-int solves 8409 tasks. Out of these 8409 tasks Lazy times out on 553, returns an error on 5166 and solves correctly 2690. On the 2961 tasks where Lazy returns a correct result, cvc5-int times out 271 times and returns 2691 correct results. Among the 2006 benchmarks where Lazy, Eager, and cvc5-int all return a correct result, cvc5-int demonstrates significantly lower CPU time and memory usage. For solving these 2006 tasks, Lazy requires 30380 s, Eager requires 62800 s, and cvc5-int requires 12750 s. Regarding memory usage, Lazy consumes 246.7 GB, Eager consumes 321.4 GB, and cvc5-int consumes 16.5 GB.

The comparison between cvc5-int and our implementations shows that cvc5-int solves significantly more benchmarks. However, cvc5-int is not strictly superior, we solve 271 benchmarks on which cvc5-int times out.

7. Conclusion

We present a lazy translation from bit-vector formulas to integer formulas that utilizes the functions `bv2nat` and $\texttt{nat2bv}_k$. Our translation consists of a conversion function C, and a replacement function R. Conversion function C replaces bit-vector terms and formulas with their semantic definition, thus incorporating `bv2nat` and $\texttt{nat2bv}_k$ into the translation process This makes our translation more modular since bit-vector terms and their semantic definition have the same sort. Additionally, conversion function C supports a translation of quantified formulas and arrays over bit-vectors. Within replacement function R, we eliminate `bv2nat` and $\texttt{nat2bv}_k$ from the translation result and introduce modulo operations instead. This is done in such a way that the amount of redundant modulo operations is reduced. We implemented our lazy translation in the SMTInterpol and as wrapper script for SMT solver. Our evaluation of both implementations indicates the correctness of the lazy translation. Furthermore, it shows that SMTInterpol with our lazy translation is able to solve 9% more tasks, 10% faster and with 23% less memory usage than with a closely related, up-to-date eager translation approach.

References

[1] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Technical Report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[2] A. Niemetz, M. Preiner, Bitwuzla, in: C. Enea, A. Lal (Eds.), Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II, volume 13965 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 3–17. URL: https://doi.org/10.1007/978-3-031-37703-7_1. doi:10.1007/978-3-031-37703-7_1.

[3] L. Aniva, H. Barbosa, C. Barrett, M. Brain, V. Camillo, G. Kremer, H. Lachnitt, A. Mohamed, M. Mohamed, A. Niemetz, et al., Cvc5 at the smt competition 2023 (2023).

[4] Y. Zohar, A. Irfan, M. Mann, A. Niemetz, A. Nötzli, M. Preiner, A. Reynolds, C. Barrett, C. Tinelli, Bit-precise reasoning via int-blasting, in: B. Finkbeiner, T. Wies (Eds.), Verification, Model Checking, and Abstract Interpretation, Springer International Publishing, Cham, 2022, pp. 496–518.

[5] F. Frohn, A calculus for modular loop acceleration, in: A. Biere, D. Parker (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I, volume 12078 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 58–76. URL: https://doi.org/10.1007/978-3-030-45190-5_4. doi:10.1007/978-3-030-45190-5_4.

[6] P. Ganty, R. Iosif, F. Konecný, Underapproximation of procedure summaries for integer programs, Int. J. Softw. Tools Technol. Transf. 19 (2017) 565–584. URL: https://doi.org/10.1007/s10009-016-0420-7. doi:10.1007/s10009-016-0420-7.

[7] S. Gulwani, S. Srivastava, R. Venkatesan, Program analysis as constraint solving, in: R. Gupta, S. P. Amarasinghe (Eds.), Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, ACM, 2008, pp. 281–292. URL: https://doi.org/10.1145/1375581.1375616. doi:10.1145/1375581.1375616.

[8] M. Colón, S. Sankaranarayanan, H. Sipma, Linear invariant generation using non-linear constraint solving, in: W. A. H. Jr., F. Somenzi (Eds.), Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, volume 2725 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 420–432. URL: https://doi.org/10.1007/978-3-540-45069-6_39. doi:10.1007/978-3-540-45069-6_39.

[9] M. Colón, H. Sipma, Synthesis of linear ranking functions, in: T. Margaria, W. Yi (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings, volume 2031 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 67–81. URL: https://doi.org/10.1007/3-540-45319-9_6. doi:10.1007/3-540-45319-9_6.

[10] A. R. Bradley, Z. Manna, H. B. Sipma, Linear ranking with reachability, in: K. Etessami, S. K. Rajamani (Eds.), Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings, volume 3576 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 491–504. URL: https://doi.org/10.1007/11513988_48. doi:10.1007/11513988_48.

[11] A. Rybalchenko, Constraint solving for program verification: Theory and practice by example, in: T. Touili, B. Cook, P. B. Jackson (Eds.), Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, volume 6174 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 57–71. URL: https://doi.org/10.1007/978-3-642-14295-6_7. doi:10.1007/978-3-642-14295-6_7.

[12] M. Barth, M. Heizmann, Ultimate IntBlastingWrapper (2023). Available at https://smt-comp.github.io/2023/system-descriptions/UltimateIntBlastingWrapper%2BSMTInterpol.pdf.

[13] J. Christ, J. Hoenicke, A. Nutz, Smtinterpol: An interpolating SMT solver, in: A. F. Donaldson, D. Parker (Eds.), Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings, volume 7385 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 248–254. URL: https://doi.org/10.1007/978-3-642-31759-0_19. doi:10.1007/978-3-642-31759-0_19.

[14] Y. C. Liu, C. Pang, D. Dietsch, E. Koskinen, T. Le, G. Portokalidis, J. Xu, Source-level bitwise branching for temporal verification of lifted binaries, CoRR abs/2105.05159 (2021). URL: https://arxiv.org/abs/2105.05159. arXiv:2105.05159.

[15] A. Gurfinkel, A. Belov, J. Marques-Silva, Synthesizing safe bit-precise invariants, in: E. Ábrahám, K. Havelund (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, volume 8413 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 93–108. URL: https://doi.org/10.1007/

978-3-642-54862-8_7. doi:10.1007/978-3-642-54862-8_7.

[16] A. Griggio, Effective word-level interpolation for software verification, in: P. Bjesse, A. Slobodová (Eds.), International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, FMCAD Inc., 2011, pp. 28–36. URL: http://dl.acm.org/citation.cfm?id=2157662.

[17] T. Okudono, A. King, Mind the gap: Bit-vector interpolation recast over linear integer arithmetic, in: A. Biere, D. Parker (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I, volume 12078 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 79–96. URL: https://doi.org/10.1007/978-3-030-45190-5_5. doi:10.1007/978-3-030-45190-5_5.

[18] P. Backeman, P. Rümmer, A. Zeljic, Bit-vector interpolation and quantifier elimination by lazy reduction, in: N. S. Bjørner, A. Gurfinkel (Eds.), 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE, 2018, pp. 1–10. URL: https://doi.org/10.23919/FMCAD.2018.8603023. doi:10.23919/FMCAD.2018.8603023.

[19] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. W. Barrett, C. Tinelli, Towards bit-width-independent proofs in SMT solvers, in: P. Fontaine (Ed.), Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings, volume 11716 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 366–384. URL: https://doi.org/10.1007/978-3-030-29436-6_22. doi:10.1007/978-3-030-29436-6_22.

[20] D. Beyer, P. Chien, N. Lee, Bridging hardware and software analysis with btor2c: A word-level-circuit-to-c translator, in: S. Sankaranarayanan, N. Sharygina (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II, volume 13994 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 152–172. URL: https://doi.org/10.1007/978-3-031-30820-8_12. doi:10.1007/978-3-031-30820-8_12.

[21] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, cvc5: A versatile and industrial-strength SMT solver, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 415–442. URL: https://doi.org/10.1007/978-3-030-99524-9_24. doi:10.1007/978-3-030-99524-9_24.

[22] D. Beyer, S. Löwe, P. Wendler, Reliable benchmarking: Requirements and solutions, STTT 21 (2019) 1–29. doi:10.1007/s10009-017-0469-y.