Abstraction of Bit-Vector Operations for BDD-based SMT Solvers[⋆]

Martin Jonáš and Jan Strejček

Masaryk University
Brno, Czech Republic
{xjonas,strejcek}@fi.muni.cz

Abstract. BDD-based SMT solvers have recently shown to be competitive for solving satisfiability of quantified bit-vector formulas. However, these solvers reach their limits when the input formula contains complicated arithmetic. Hitherto, this problem has been alleviated by approximations reducing efficient bit-widths of bit-vector variables. In this paper, we propose an orthogonal abstraction technique working on the level of the individual instances of bit-vector operations. In particular, we compute only several bits of the operation result, which may be sufficient to decide the satisfiability of the formula. Experimental results show that our BDD-based SMT solver Q3B extended with these abstractions can solve more quantified bit-vector formulas from the SMT-LIB repository than state-of-the-art SMT solvers Boolector, CVC4, and Z3.

1 Introduction

In the modern world, as the computer software becomes still more ubiquitous and complex, there is an increasing need to test it and formally verify its correctness. Several approaches to software verification, such as symbolic execution or bounded model checking, rely on the ability to decide whether a given first-order formula in a suitable logical theory is satisfiable. To this end, many of the verifiers use Satisfiability Modulo Theories (SMT) solvers, which can solve precisely the task of checking satisfiability of a first-order formula in a given logical theory. For describing software, the natural choice of a logical theory is the theory of *fixed-size bit-vectors* in which the objects are vectors of bits and the operations on them precisely reflect operations performed by computers. Moreover, in applications such as synthesis of invariants, ranking functions, or loop summaries, the formulas in question also naturally contain quantifiers [8, 17, 6, 11, 12].

It is therefore not surprising that the development of SMT solvers for quantified formulas in the theory of fixed-size bit-vectors has seen several advances in the recent years. In particular, the support for arbitrarily quantified bit-vector formulas has been implemented to existing solvers Z3 [18], Boolector [15], and CVC4 [14]. Moreover, new tools that aim for precisely this theory, such as the solver Q3B [9], were developed. Approaches of these tools fall into two categories:

[⋆] The research was supported by Czech Science Foundation, grant GA18-02177S.

Z3, Boolector, and CVC4 use variants of quantifier instantiation that iteratively produces quantifier-free formulas that can be solved by a solver for quantifier-free bit-vector formulas. On the other hand, the solver Q3B uses Binary Decision Diagrams (BDDs) to represent quantified bit-vector formulas and to decide their satisfiability.

However, BDDs have inherent limitations. For example, if a formula contains multiplication of two variables, the BDD that represents it is guaranteed to be exponential in size regardless the chosen order of variables. Similarly, if the formula contains complicated arithmetic, the produced BDDs tend to grow in size very quickly. The solver Q3B tries to alleviate this problem by computing approximations [9] of the original formula to reduce sets of values that can be represented by the individual variables and, in turn, to reduce sizes of the resulting BDDs. In particular, if the set of possible values of all existentially quantified variables is reduced and the formula is still satisfiable, the original formula must have been satisfiable. Conversely, if the set of possible values of all universally quantified variables is reduced and the formula is still unsatisfiable, the original formula must have been unsatisfiable.

Although the approximations allowed Q3B to remain competitive with state-of-the-art SMT solvers, the approach has several drawbacks. Currently, Q3B cannot solve satisfiability of simple formulas such as

$$\exists x, y \ \left((x < 2) \ \land \ (x > 4) \ \land \ (x \cdot y = 0))\right),$$

$$\exists x, y \ \left((x \ll 1) \cdot y = 1\right),$$

$$\exists x, y \ \left(x > 0 \ \land \ x \leq 4 \ \land \ y > 0 \ \land \ y \leq 4 \ \land \ x \cdot y = 0\right),$$

where all variables and constants have bit-width 32, and $\ll$ denotes bit-wise shift left. All these three formulas are unsatisfiable, but cannot be decided without approximations, because they contain non-linear multiplication. Moreover, they cannot be decided even with approximations, because they are unsatisfiable and contain no universally quantified variables that could be used to approximate the formula.

However, the three above-mentioned formulas have something in common: only a few of the bits of the multiplication results are sufficient to decide satisfiability of the formulas. The first formula can be decided unsatisfiable without computing any bits of $x \cdot y$ whatsoever. The second formula can be decided by computing only the least-significant bit of $(x \ll 1) \cdot y$ because it must always be zero. The third formula can be decided by computing 5 least-significant bits of $x \cdot y$, because they are enough to rule out all values of x and y between 1 and 4 as models.

With this in mind, we propose an improvement of BDD-based SMT solvers such as Q3B by allowing to compute only several bits of results of arithmetic operations. To achieve this, the paper defines abstract domains in which the operations can produce *do-not-know* values and shows that these abstract domains can be used to decide satisfiability of an input formula.

The paper is structured as follows. Section 2 provides necessary background and notations for SMT, bit-vector theory, and binary decision diagrams. Section 3 defines abstract domains for terms and formulas and shows how to use them to decide satisfiability of a formula. Section 4 introduces specific term and formula abstract domains that are used to compute only several bits from results of arithmetic bit-vector operations. Section 5 describes our implementation of these abstract domains in the SMT solver Q3B and the following Section 6 provides evaluation of this implementation both in comparison to the original Q3B and to other state-of-the-art SMT solvers.

2 Preliminaries

2.1 Bit-Vector Theory

This section briefly recalls the *theory of fixed sized bit-vectors* (*BV* or *bit-vector theory* for short). In the description, we assume familiarity with standard definitions of many-sorted logic, well-sorted terms, atomic formulas, and formulas. In the following, we denote the set of all well-sorted terms as $\mathcal{T}$ and the set of all well-sorted formulas as $\mathcal{F}$.

The bit-vector theory is a many-sorted first-order theory with infinitely many sorts corresponding to bit-vectors of various lengths. The BV theory uses only three predicates, namely *equality* ($=$), *unsigned inequality* of binary-encoded natural numbers ($\leq_u$), and *signed inequality* of integers in two's complement representation ($\leq_s$). The theory also contains various functions including *addition* ($+$), *multiplication* ($\cdot$), *unsigned division* ($\div$), *unsigned remainder* ($\%$), bit-wise *and* (bvand), bit-wise *or* (bvor), bit-wise *exclusive or* (bvxor), *left-shift* ($\ll$), *right-shift* ($\gg$), *concatenation* (concat), and *extraction* of n bits starting from position p ($\mathsf{extract}_p^n$). The signature of BV theory also contains constants $c^{[n]}$ for each bit-width $n > 0$ and a number $0 \leq c \leq 2^n - 1$. If a bit-width of a constant or a variable is not specified, we suppose that it is equal to 32. We denote set of all bit-vectors as $\mathcal{BV}$ and the set of all variables as *vars*.

For a valuation μ that assigns to each variable from *vars* a value in its domain, $[\![\_]\!]_\mu$ denotes the evaluation function, which assigns to each term t the bit-vector obtained by substituting variables in t by their values given by μ and evaluating all functions. Similarly, the function $[\![\_]\!]_\mu$ assigns to each formula φ the value obtained by substituting free variables in φ by values given by μ and evaluating all functions, predicates, logic operators etc. A formula φ is *satisfiable* if $[\![\varphi]\!]_\mu = 1$ for some valuation μ; it is *unsatisfiable* otherwise.

The precise definition of many-sorted logic can be found for example in Barrett et al. [3]. The precise description of bit-vector theory and its operations can be found for example in the paper describing complexity of quantified bit-vector theory by Kovásznai et al. [10].

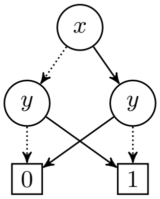

Fig. 1: BDD for $(x \,\mathrm{xor}\, y)$

2.2 Binary Decision Diagrams

A binary decision diagram (BDD) is a data structure that can succinctly represent Boolean functions. Formally, it is a binary directed acyclic graph that has at most two leaves, labelled by 0 and 1, and inner nodes labelled by formal arguments of the function. Each inner node has two children, called *high* and *low* children, that denote values 1 and 0, respectively, of the corresponding formal argument. Given a BDD that represents a Boolean function f, the value of f in a given assignment can be computed by traversing the BDD as follows: start in the root node; if the value of the argument corresponding to the current node is 1, continue to the high child, otherwise continue to the low child; continue with the traversal until reaching a leaf node and return its label. Given a BDD b and an assignment μ, we denote the result of the function represented by b as $[\![b]\!]_\mu$. For example, Figure 1 shows a BDD that represents a binary function $f(x, y) = (x \,\mathrm{xor}\, y)$. According to the traditional notation, the high children are marked by solid edges, the low children are marked by dotted edges. The trivial BDDs $\boxed{0}$ and $\boxed{1}$ represent functions *false* (0) and *true* (1), respectively.

Alternatively, binary decision diagrams can be used to represent a set of satisfying assignments (also called *models*) of a Boolean formula φ. Such a BDD represents a function that has Boolean variables of the formula φ as formal arguments and that evaluates to 1 in a given assignment iff the assignment is a model of the formula φ. In this view, the BDD of Figure 1 represents the set of assignments satisfying the formula $(x \wedge \neg y) \vee (\neg x \wedge y)$.

In this paper, we suppose that all binary decision diagrams are *reduced* and *ordered*. A BDD is *ordered* if for all pairs of paths in the BDD the order of common variables is the same. A BDD is *reduced* if it does not contain any inner node with the same high and low child. It has been shown that reduced and ordered BDDs (ROBDDs) are canonical – given a variable order, there is exactly one BDD for each given function [5].

Binary decision diagrams can be also used to represent an arbitrary *bit-vector function*, i.e., a function that assigns a bit-vector value to each assignment of bit variables. Such a function of a bit-width k (i.e., the produced bit-vectors have the bit-width k) can be represented by a vector of BDDs $\overline{b} = (b_i)_{0 \leq i < k}$. Result of this function for an assignment μ is then the bit-vector $([\![b_i]\!]_\mu)_{0 \leq i < k}$. For example, Figure 2 shows a vector of BDDs representing addition $x_2x_1x_0 + y_2y_1y_0$ of two bit-vectors of size 3. In the following text, we denote the set of all BDDs as $\mathtt{BDD}$

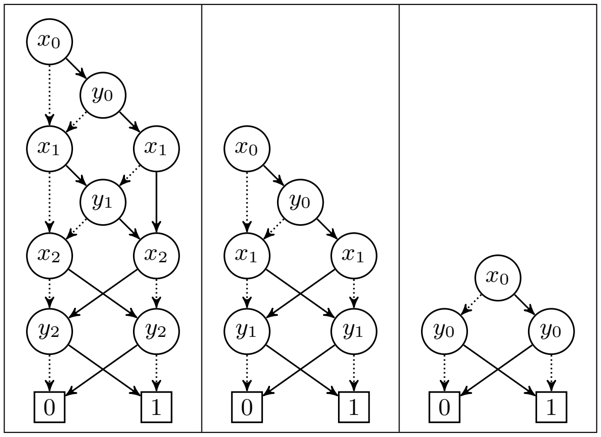

Fig. 2: Vector of BDDs representing the addition $x_2x_1x_0 + y_2y_1y_0$ of two bit-vectors of size 3. The least significant bit of the result is on the right.

and the set of all vectors of BDDs as `BDDvec`. We use the overlined symbols for both vectors of BDDs and bit-vectors.

2.3 Operations on Binary Decision Diagrams

It has been shown by Bryant [5] that given ROBDDs for Boolean functions f and g, one can compute a BDD for functions $f(\overline{x}) \wedge g(\overline{x})$ and $f(\overline{x}) \vee g(\overline{x})$ in polynomial time. A BDD for negation can be obtained by exchanging leaf nodes 0 and 1. Using these operations, a BDD for an arbitrary Boolean formula can be constructed by computing the corresponding BDDs for all subformulas from the smallest ones. Bryant has also described a function that modifies a given BDD by setting selected variables to given values. Using this function, it is possible to eliminate variable x from a given BDD representing $f(x, \overline{y})$ existentially or universally by computing the BDDs for $f(0, \overline{y}) \vee f(1, \overline{y})$ or $f(0, \overline{y}) \wedge f(1, \overline{y})$, respectively. We denote the functions for computing *conjunction*, *disjunction*, and *negation* of BDDs by the infix notations &, |, and !, respectively. Using these functions, we can define functions computing *equivalence* and *exclusive or* of two BDDs with the infix notations $\leftrightarrow$ and xor, respectively.

Further, given two vectors of BDDs that represent bit-vector functions f and g of the same bit-width, a vector of BDDs for the function $f(\overline{x}) + g(\overline{x})$, where $+$ is addition of two bit-vectors, can also be computed by using the basic logical operations on BDDs representing the individual bits. Listing 1.1 shows a pseudocode of this computation, which is implemented for example in the BDD package BuDDy[1]. Other arithmetical operations such as *multiplication*, *division*, or *remainder* can also be computed in this way, although the algorithms are more involved in these cases.

[1] `http://sourceforge.net/projects/buddy`

Listing 1.1: Pseudo-codes computing operations *addition* (+) and *equality* (=) on vectors $\overline{a} = (a_i)_{0 \leq i < k}$ and $\overline{b} = (b_i)_{0 \leq i < k}$ of BDDs.

```
bvec_add($\overline{a}$, $\overline{b}$)
{
   $\overline{result}$ $\leftarrow$ ($\boxed{0}, \boxed{0}, \ldots, \boxed{0}$) with the bit-width $k$;
   $carry$ $\leftarrow$ $\boxed{0}$;
   for $i$ from 0 to $k$ - 1 {
      $result_i$ $\leftarrow$ $a_i$ xor $b_i$ xor $carry$;
      $carry$ $\leftarrow$ $(a_i$ & $b_i) \mid (carry$ & $(a_i \mid b_i))$;
   }
   return $\overline{result}$;
}

bvec_eq($\overline{a}$, $\overline{b}$)
{
   $result$ $\leftarrow$ $\boxed{1}$;
   for $i$ from 0 to $k$ - 1 {
      $result$ $\leftarrow$ $result$ & $(a_i \leftrightarrow b_i)$;
   }
   return $result$;
}
```

Finally, given two vectors of BDDs that represent bit-vector functions f and g of the same bit-width, it is also possible to compute the BDD for their equality $f(\overline{x}) = g(\overline{x})$, the BDD for their unsigned inequality $f(\overline{x}) \leq_u g(\overline{x})$, and the BDD for their signed inequality $f(\overline{x}) \leq_s g(\overline{x})$. Listing 1.1 shows a pseudo-code computing the BDD for equality, which corresponds to the implementation in BuDDy.

Using these algorithms, it is possible to define a function `t2BDDvec`, which converts a bit-vector term to the vector of BDDs representing the function computed by the term. Consequently, it is possible to define a function `f2BDD`, which converts a bit-vector formula to the corresponding BDD.

3 Formula and Term Abstractions

Although it is often infeasible to compute functions `t2BDDvec` and `f2BDD` precisely, even an imprecise result can sometimes be enough to decide satisfiability of the input formula as illustrated in Introduction. In this section we describe notions of a *term abstract domain*, which captures an imprecise computation of `t2BDDvec`, and a *formula abstract domain*, which captures an imprecise computation of `f2BDD`. Generally, a term abstract domain defines a set of abstract objects A, a function α mapping terms to these abstract objects, and an evaluation function $[\![\_]\!]^A_\_$, which assigns to each abstract object a and a variable assignment μ the set $[\![a]\!]^A_\mu$ of bit-vectors represented by a.

Definition 1 (Term abstract domain). *A* term abstract domain *is a triple* $(A, \alpha, [\![\_]\!]^{A}_{\_})$*, where* A *is a set of* abstract objects*,* $\alpha \colon \mathcal{T} \to A$ *is an* abstraction function*, and* $[\![\_]\!]^{A}_{\_} \colon A \times \mathcal{BV}^{vars} \to 2^{\mathcal{BV}}$ *is an* abstract evaluation function*.*

As an example, consider the *precise* BDD *term abstract domain*, in which the corresponding vector of BDDs is assigned to each term. In particular, the precise BDD term abstract domain is the triple $(\mathtt{BDDvec}, \mathtt{t2BDDvec}, [\![\_]\!]^{\mathtt{BDDvec}}_{\_})$, where $[\![\overline{a}]\!]^{\mathtt{BDDvec}}_{\mu}$ is the singleton set $\{bv\}$ such that bv is the result of evaluation of vector $\overline{a}$ of BDDs in the assignment μ, i.e., $bv = [\![\overline{a}]\!]_{\mu}$. This abstract domain enjoys two interesting properties: for each term and assignment, the corresponding abstract object contains the correct result and it does not contain any incorrect result. These properties are called *completeness* and *soundness*.

Definition 2. *A term abstract domain* $(A, \alpha, [\![\_]\!]^{A}_{\_})$ *is* complete *if each term* $t \in \mathcal{T}$ *and each assignment* μ *satisfy* $[\![t]\!]_{\mu} \in [\![\alpha(t)]\!]^{A}_{\mu}$*. Conversely, it is* sound *if each* t *and* μ *satisfy* $[\![\alpha(t)]\!]^{A}_{\mu} \subseteq \{[\![t]\!]_{\mu}\}$*.*

Similarly to the term abstract domain, the *formula abstract domain* defines a set of abstract objects A, a function α mapping formulas to these abstract objects, and an evaluation function $[\![\_]\!]^{A}_{\_}$, which assigns to each abstract object a and a variable assignment μ the set $[\![a]\!]^{A}_{\mu} \subseteq \{0, 1\}$ of truth values associated to a.

Definition 3 (Formula abstract domain). *A* formula abstract domain *is a triple* $(A, \alpha, [\![\_]\!]^{A}_{\_})$*, where* A *is an arbitrary set of* abstract objects*,* $\alpha \colon \mathcal{F} \to A$ *is an* abstraction function*, and* $[\![\_]\!]^{A}_{\_} \colon A \times \mathcal{BV}^{vars} \to 2^{\{1,0\}}$ *is an* abstract evaluation function*.*

Definition 4. *A formula abstract domain* $(A, \alpha, [\![\_]\!]^{A}_{\_})$ *is* complete *if each formula* $\varphi \in \mathcal{F}$ *and each assignment* μ *satisfy* $[\![\varphi]\!]_{\mu} \in [\![\alpha(\varphi)]\!]^{A}_{\mu}$*. Conversely, it is* sound *if each* φ *and* μ *satisfy* $[\![\alpha(\varphi)]\!]^{A}_{\mu} \subseteq \{[\![\varphi]\!]_{\mu}\}$*.*

As in the case of terms, the precise computation of the BDD corresponding to a formula yields a *precise* BDD *formula abstract domain*, which is complete and sound. The precise BDD formula abstract domain is a triple $(\mathtt{BDD}, \mathtt{f2BDD}, [\![\_]\!]^{\mathtt{BDD}}_{\_})$, where $[\![a]\!]^{\mathtt{BDD}}_{\mu}$ is the singleton set $\{b\}$, where b is the result of evaluation of the BDD a in the assignment μ, i.e., $b = [\![a]\!]_{\mu}$.

In the following, we weaken the precise term and formula BDD abstract domains by dropping the requirement on the soundness, while still retaining the requirement of completeness. As the following theorem demonstrates, such an abstract domain can still be used for deciding satisfiability of the input formula.

Theorem 1. *Let* φ *be a formula and* $(A, \alpha, [\![\_]\!]^{A}_{\_})$ *be a complete formula abstract domain. If there exists an assignment* μ *such that* $[\![\alpha(\varphi)]\!]^{A}_{\mu} = \{1\}$*, the formula* φ *is satisfiable. On the other hand, if all assignments* μ *satisfy* $[\![\alpha(\varphi)]\!]^{A}_{\mu} = \{0\}$*, the formula is unsatisfiable.*

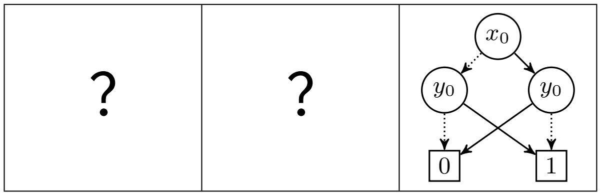

Fig. 3: Truncated result of addition of two three-bit bit-vectors.

Note that for abstract domains in which abstract objects are BDDs with the standard evaluation function, the check for existence of the assignment μ from the previous theorem is easy to implement. Such an assignment exists precisely if the leaf node 1 is reachable from the root of the BDD. Furthermore, if the BDD is reduced and ordered, this happens precisely if the BDD is not $\boxed{0}$. Conversely, all assignments μ satisfy $[\![\alpha(\varphi)]\!]^{A}_{\mu} = \{0\}$ iff the reduced and ordered BDD $\alpha(\varphi)$ is $\boxed{0}$.

4 Truncating Formula and Term Abstract Domains

This section describes a term abstract domain and a corresponding formula abstract domain that allow *truncating* results from bit-vector operations, i.e., computing only several bits from the result of arithmetic bit-vector operations.

In this whole section, we suppose that all formulas are in *negation normal form*, i.e., logical operations are conjunctions, disjunctions, and negations and negations are applied only on atomic subformulas. As traditional, we denote the literal $\neg(t_1 = t_2)$ as $t_1 \neq t_2$.

4.1 Truncating Term Abstract Domain

We introduce the *truncating term abstract domain* first. It is a complete but unsound term abstract domain, in which the terms are represented by vectors whose elements are BDDs, as in the precise term abstract domain, or *do-not-know values*. The do-not-know value, denoted as ?, represents an unknown value of the corresponding bit – it can be both 0 and 1.

For example, Figure 3 shows the result of computing only the least-significant bit of an addition of two bit-vectors $x_2x_1x_0 + y_2y_1y_0$. The value of this abstract object under the assignment $\{x \mapsto 001, y \mapsto 100\}$ is the set $\{001, 011, 101, 111\}$, since only the value of the least-significant bit is computed precisely.

Formally, the truncating term abstract domain is a triple

$$(\mathtt{tBDDvec}, \mathtt{t2tBDDvec}, [\![\_]\!]^{\mathtt{tBDDvec}}_{\_}),$$

where the set of abstract elements consists of vectors of BDDs and ? elements

$$\mathtt{tBDDvec} = \{(b_i)_{0 \leq i < k} \mid k \in \mathbb{N}, b_i \in \mathtt{BDD} \cup \{?\} \text{ for all } 0 \leq i < k\}$$

and the abstract evaluation function assigns to each $\bar{b} = (b_i)_{0\leq i<k} \in$ tBDDvec and an assignment μ the set of bit-vector values

$$[\![\bar{b}]\!]^{\mathsf{tBDDvec}}_{\mu} = \{(v_i)_{0\leq i<k} \mid \text{if } b_i = \text{?} \text{ then } v_i \in \{0,1\} \text{ else } v_i = [\![b_i]\!]_{\mu}, 0 \leq i < k\}.$$

There are many possible implementations of the t2tBDDvec function including the following two:

1. the number of computed bits is specified and other bits are set to ?,
2. the limit on BDD nodes in the result of the operation is specified and after reaching it, the remaining bits are set to ?.

In the following, we describe only the second option as our evaluation has shown that it outperforms the first one on the set of our benchmarks. Furthermore, it is easy to derive the implementation of the first option based on the description of the other option. In addition, we suppose that the limit on BDD nodes is fixed for the given domain. In the implementation, we use multiple abstract domains varying by the BDD node limit.

The function t2tBDDvec is computed recursively on the input term. The base case for the variables or constants is straightforward and it is the same as in the precise function t2BDDvec. On the other hand, the computation for bit-vector operations differs from t2BDDvec in two important aspects:

- The operations have to work correctly with ? elements. To achieve this, we modify the BDD operations &, |, and xor, which occurred in the computation of t2BDDvec. The handling of ? in the modified operations is similar to the definition of logical connectives in the three-valued logic and to the way bit-masks are computed in the SMT solver mcBV [19]. The modified BDD operations $\&_t$, $|_t$, and xor_t are computed as follows:

$$a \;\&_t\; b = \begin{cases} \boxed{0} & \textit{if } a = \boxed{0} \textit{ or } b = \boxed{0} \\ a \;\&\; b & \textit{if } a, b \notin \{\boxed{0}, ?\} \\ ? & \textit{otherwise} \end{cases}$$

$$a \;|_t\; b = \begin{cases} \boxed{1} & \textit{if } a = \boxed{1} \textit{ or } b = \boxed{1} \\ a \mid b & \textit{if } a, b \notin \{\boxed{1}, ?\} \\ ? & \textit{otherwise} \end{cases}$$

$$a \,\text{xor}_t\, b = \begin{cases} a \,\text{xor}\, b & \textit{if } a \neq ? \textit{ and } b \neq ? \\ ? & \textit{otherwise} \end{cases}$$

  Note that ? xor_t ? is not $\boxed{1}$ as each ? can represent a different value.
- Implementation of operations has to consider the given limit on the number of BDD nodes and set the bits that have not been computed precisely to ? after the limit has been reached. The example of modification of the original bvec_add that uses the node limit is shown in the Listing 1.2. The

Listing 1.2: Pseudo-code computing *truncated addition* of two tBDDvecs $\overline{a} = (a_i)_{0 \leq i < k}$ and $\overline{b} = (b_i)_{0 \leq i < k}$.

```
bvec_add_nodeLimit($\overline{a}$, $\overline{b}$, limit)
{
  $\overline{result}$ $\leftarrow$ $(\boxed{0}, \boxed{0}, \ldots, \boxed{0})$ with the bit-width $k$;
  $carry$ $\leftarrow$ $\boxed{0}$;
  for $i$ from 0 to $k$ - 1 {
    if (bddNodes($\overline{result}$) > limit) {
      $result_i$ $\leftarrow$ ?;
    } else {
      $result_i$ $\leftarrow$ $a_i$ $\mathrm{xor}_t$ $b_i$ $\mathrm{xor}_t$ $carry$;
      $carry$ $\leftarrow$ $(a_i\ \&_t\ b_i)\ |_t\ (carry\ \&_t\ (a_i\ |_t\ b_i))$;
    }
  }
  return $\overline{result}$;
}
```

implementations of other operations are similar. However, they differ in the order in which the precise bits are produced: during computation of addition and multiplication, the first computed precise bits are the least significant ones; during computation of division, the first computed precise bits are the most-significant ones. Therefore if the computation of addition or multiplication reaches the BDD node limit, remaining most-significant bits are set to ?, while for division least-significant bits are set to ?.

The set of values represented by the result of t2tBDDvec always contains the precise result of the given term because the t2tBDDvec can only make precise values imprecise by using ? elements. The truncating term abstract domain is therefore complete. However, it is not sound, as the abstract object can describe also incorrect results.

4.2 Truncating Formula Abstract Domain

We now define a formula abstract domain that uses results of truncated bit-vector operations. Intuitively, the abstract elements in this abstract domain are BDD pairs (b_{must}, b_{may}): the first one determines the assignments that satisfy the formula for all possible values of ? elements, and the second one determines the assignments that satisfy the formula for some values of ? elements.

Formally, the *truncating formula abstract domain* is a triple

$$(\mathsf{BDDpair}, \mathsf{f2BDDpair}, [\![\_]\!]^{\mathsf{BDDpair}}_{\_}),$$

where $\mathsf{BDDpair} = \mathsf{BDD} \times \mathsf{BDD}$ and the evaluation function assigns to each pair $(b_{must}, b_{may}) \in \mathsf{BDDpair}$ the set of Boolean values

$$[\![(b_{must}, b_{may})]\!]^{\mathsf{BDDpair}}_{\mu} = \{v \in \{0,1\} \mid [\![b_{must}]\!]_{\mu} \implies v \implies [\![b_{may}]\!]_{\mu}\}.$$

Observe that $[\![(b_{must}, b_{may})]\!]_{\mu}^{\mathtt{BDDpair}}$ is $\{0\}$ when $[\![b_{must}]\!]_{\mu} = [\![b_{may}]\!]_{\mu} = 0$, it is $\{1\}$ when $[\![b_{must}]\!]_{\mu} = [\![b_{may}]\!]_{\mu} = 1$, and it is $\{0,1\}$ when $[\![b_{must}]\!]_{\mu} = 0$, $[\![b_{may}]\!]_{\mu} = 1$. While the result would be $\emptyset$ in the remaining case $[\![b_{must}]\!]_{\mu} = 1$, $[\![b_{may}]\!]_{\mu} = 0$, this situation never happens for the result of the defined function `f2BDDpair`.

The function `f2BDDpair`(φ) is defined recursively as follows.

1. The formula φ is an atomic subformula or its negation, i.e., $\varphi \equiv t_1 \bowtie t_2$ for $\bowtie \in \{=, \neq, \leq_u, <_u, \leq_s, <_s\}$. The function `f2BDDpair` computes the pair (b_{must}, b_{may}) from `t2tBDDvec`(t_1) and `t2tBDDvec`(t_2) using modified algorithms for the corresponding operations on vectors of standard BDDs. For example, Listing 1.3 shows an algorithm for equality of `t2tBDDvec`(t_1) and `t2tBDDvec`(t_2) (compare with the algorithm for equality of vectors of standard BDDs presented in Listing 1.1). In this algorithm, the value b_{must} becomes 0 if there is `?` in any of the input vectors, because then the arguments may differ for some value of the `?`. On the other hand, the value b_{may} is the conjunction of equality of all pairs of corresponding bits that both have a known value. In particular, construction of b_{may} ignores the pairs of bits containing some `?` as it could be the case that equality holds for these bits. Listing 1.3 also shows the algorithm for computing disequality of `t2tBDDvec`(t_1) and `t2tBDDvec`(t_2). The algorithms for other relational symbols are similar.
2. The formula φ has the form $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$. Let (b^1_{must}, b^1_{may}) be the result of `f2BDDpair`(φ_1) and (b^2_{must}, b^2_{may}) be the result of `f2BDDpair`(φ_2). Then we define

$$\mathtt{f2BDDpair}(\varphi_1 \wedge \varphi_2) = ((b^1_{must} \,\&\, b^2_{must}),\ (b^1_{may} \,\&\, b^2_{may})),$$
$$\mathtt{f2BDDpair}(\varphi_1 \vee \varphi_2) = ((b^1_{must} \mid b^2_{must}),\ (b^1_{may} \mid b^2_{may})).$$

3. The formula φ has the form $\forall x.\, \varphi_1$ or $\exists x.\, \varphi_1$. Let (b^1_{must}, b^1_{may}) be the result of `f2BDDpair`(φ_1). Then we define

$$\mathtt{f2BDDpair}(\forall x.\, \varphi_1) = (\mathit{bdd\_forall}(x, b^1_{must}),\ \mathit{bdd\_forall}(x, b^1_{may})),$$
$$\mathtt{f2BDDpair}(\exists x.\, \varphi_1) = (\mathit{bdd\_exists}(x, b^1_{must}),\ \mathit{bdd\_exists}(x, b^1_{may})),$$

where the function $\mathit{bdd\_forall}(x, \_)$ eliminates the variable x universally and $\mathit{bdd\_exists}(x, \_)$ eliminates it existentially as explained in Section 2.3.

Example 1. Let t, r, s, u be bit-vector terms, for which we have computed only the least-significant bit as computation of the other bits was infeasible. Formally,

$$\mathtt{t2tBDDvec}(t) = (?, \ldots, ?, b_t), \qquad \mathtt{t2tBDDvec}(r) = (?, \ldots, ?, b_r),$$
$$\mathtt{t2tBDDvec}(s) = (?, \ldots, ?, b_s), \qquad \mathtt{t2tBDDvec}(u) = (?, \ldots, ?, b_u),$$

where b_t, b_r, b_s, b_u are BDDs.

Consider the formula $t = r$. The function `f2BDDpair` applied on this formula returns the pair $(\boxed{0}, b_t \leftrightarrow b_r)$. This pair says that every assignment satisfying the formula must also satisfy the BDD $b_t \leftrightarrow b_r$. Therefore, if $t = r$ is put in

Listing 1.3: Pseudo-codes computing *truncated equality* and *truncated disequality* of two tBDDvecs $\overline{a} = (a_i)_{0 \leq i < k}$ and $\overline{b} = (b_i)_{0 \leq i < k}$.

```
bvec_eq_trunc($\overline{a}$, $\overline{b}$)
{
  $result_{must}$ ← $\boxed{1}$;
  $result_{may}$ ← $\boxed{1}$;
  for $i$ from 0 to $k$ - 1 {
    if ($a_i$ == ? or $b_i$ == ?) {
      $result_{must}$ ← $\boxed{0}$;
    } else {
      $result_{must}$ ← $result_{must}$ & $(a_i \leftrightarrow b_i)$;
      $result_{may}$ ← $result_{may}$ & $(a_i \leftrightarrow b_i)$;
    }
  }
  return ($result_{must}$, $result_{may}$);
}

bvec_neq_trunc($\overline{a}$, $\overline{b}$)
{
  $result_{must}$ ← $\boxed{0}$;
  $result_{may}$ ← $\boxed{0}$;
  for $i$ from 0 to $k$ - 1 {
    if ($a_i$ == ? or $b_i$ == ?) {
      $result_{may}$ ← $\boxed{1}$;
    } else {
      $result_{must}$ ← $result_{must}$ | $(a_i \text{ xor } b_i)$;
      $result_{may}$ ← $result_{may}$ | $(a_i \text{ xor } b_i)$;
    }
  }
  return ($result_{must}$, $result_{may}$);
}
```

conjunction with another formula implying that $b_t \leftrightarrow b_r$ is equal to $\boxed{0}$, the whole conjunction can be decided to be unsatisfiable.

Consider the formula $s \neq u$. The function f2BDDpair now produces the pair $(b_s \text{ xor } b_u, \boxed{1})$. Intuitively, if an assignment satisfies $b_s \text{ xor } b_u$, it also satisfies the formula $s \neq u$, regardless the values of the remaining bits.

Finally, consider formulas $t = r \wedge s \neq u$ and $t = r \vee s \neq u$. The result of f2BDDpair($t = r \wedge s \neq u$) is $(\boxed{0}, b_t \leftrightarrow b_r)$, while the result of f2BDDpair($t = r \vee s \neq u$) is $(b_s \text{ xor } b_u, \boxed{1})$.

Similarly to the truncating term abstract domain, the truncating formula abstract domain is also complete, as the following theorem shows.

Theorem 2. *The truncating formula abstract domain is complete.*

Proof (sketch). It can be shown by induction on the structure of the formula that if $\texttt{f2BDDpair}(\varphi) = (b_{must}, b_{may})$, then the following must hold for each assignment μ:

$$[\![b_{must}]\!]_\mu \implies [\![\varphi]\!]_\mu \quad \text{and} \quad [\![\varphi]\!]_\mu \implies [\![b_{may}]\!]_\mu.$$

Therefore $[\![\varphi]\!]_\mu \in [\![\texttt{f2BDDpair}(\varphi)]\!]_\mu^{\texttt{BDDpair}}$ holds for each assignment μ, so the abstract domain is indeed complete. □

Since the truncating formula abstract domain is complete, Theorem 1 can be used to check satisfiability of a given formula φ. Consider b_{must} and b_{may} such that

$$\texttt{f2BDDpair}(\varphi) = (b_{must}, b_{may}).$$

Then if b_{must} is not $\boxed{0}$, the formula φ is satisfiable. On the other hand, if b_{may} is $\boxed{0}$, the formula φ is not satisfiable.

This satisfiability check solves the formulas mentioned in Introduction as the motivation for the described approach. For all three of the formulas, the b_{may} after computing at least 5 bits from results of the multiplication is $\boxed{0}$ and the formulas can be decided as unsatisfiable.

5 Implementation

We have implemented the described truncated abstract formula domain into the SMT solver Q3B, which is written in C++. The solver Q3B uses the package CUDD [16] for BDD representation and operations, and the implementation of bit-vectors and bit-vector operations for CUDD by P. Navrátil [13]. We have modified this implementation to support ? elements and to support computing truncated results of bit-vector operations and computing (b_{must}, b_{may}) for all bit-vector relation operators and logical operators $\wedge$ and $\vee$. The operations that introduce ? elements, when the precise result would contain too many BDD nodes, are *addition*, *multiplication*, and *division*. We have selected these operations as the original version of Q3B often has difficulties to handle them. The implementation is available at GitHub[2].

In contrast to the description of computing formula abstraction from the previous section, the implementation does not convert the formula to the negation normal form. Instead, during the traversal of the formula, the solver maintains the *polarity* of the current subformula and uses it to perform the appropriate abstraction of atomic subformulas.

5.1 Further Optimizations

The described approach cannot solve simple formulas as $x \cdot y \leq_u 2 \wedge x \cdot y \geq_u 4$. Even if the subterms $x \cdot y$ are computed abstractly, the information that the ? elements in the two vectors representing the two occurrences of $x \cdot y$ have been

[2] `https://github.com/martinjonas/Q3B/releases/tag/ictac2018`

the same is lost after computing BDD pairs for $x \cdot y \leq_u 2$ and $x \cdot y \geq_u 4$. Therefore, in the implementation, each multiplication and division is replaced by a fresh existentially quantified variable and the constraint specifying its relation to the multiplication or division, respectively, is added to the formula. For example, the previous formula is transformed to the equivalent formula

$$\exists m_{x,y}(\, m_{x,y} \leq_u 2 \,\land\, m_{x,y} \geq_u 4 \,\land\, m_{x,y} = x \cdot y).$$

This formula is decided as unsatisfiable even if $x \cdot y$ is computed with arbitrarily low precision. Note that the transformed formula is still not solved by the original version of Q3B as the solver starts to build the precise BDDs for all three conjuncts. Although this particular case could be solved by computing precise BDDs for $m_{x,y} \leq_u 2 \land m_{x,y} \geq_u 4$ and not for the third conjunct as the conjunction is already unsatisfiable, the proposed formula modification is more general.

Similar problem arises for example in the unsatisfiable formula $x \cdot y \leq_u 2 \land \forall z\, (z \cdot y \geq_u 4)$. This formula cannot be solved even after performing the above-mentioned transformation. The transformation yields the formula

$$\exists m_{x,y}(m_{x,y} \leq_u 2 \;\land\; m_{x,y} = x \cdot y \;\land\; \forall z\, \exists m_{z,y}\, (m_{z,y} \geq_u 4 \;\land\; m_{z,y} = z \cdot y)),$$

which can not be decided unsatisfiable even by using the abstractions, because the solver can not infer the relationship between variables $m_{x,y}$ and $m_{z,y}$. To solve such formula, the implementation adds a congruence subformula stating that $x = z \rightarrow m_{x,y} = m_{z,y}$ to the formula. This results in the formula

$$\begin{aligned}\exists m_{x,y}\big(\, & m_{x,y} \leq_u 2 \;\land\; m_{x,y} = x \cdot y \;\land \\ & \land\; \forall z\, \exists m_{z,y}\, (m_{z,y} \geq_u 4 \;\land\; m_{z,y} = z \cdot y \;\land\; (x = z \rightarrow m_{x,y} = m_{z,y}))\big),\end{aligned}$$

which can be decided unsatisfiable using the abstractions. Similarly to the previous transformation, the resulting formula is equivalent to the original one and its unsatisfiability can not be shown by the original solver without the abstractions, because it is infeasible to compute the precise BDD for the inner quantified subformula.

5.2 Combining Operation Abstractions and Formula Approximations

The solver Q3B employs formula approximations, which can in some cases help with solving input formulas with multiplication. This subsection elaborates on the interaction of these approximations with the newly implemented operation abstractions. Approximations of formulas are of two kinds: *underapproximation* and *overapproximation*. An underapproximation is a formula that logically entails the input formula; therefore if an underapproximation is satisfiable, the original formula is also satisfiable. On the other hand, an overapproximation is a formula that is logically entailed by the input formula; if an overapproximation is unsatisfiable, the original formula is also unsatisfiable.

Listing 1.4: Algorithm combining operation abstractions with underapproximation.

```
solve_underapproximation($\varphi$) {
  effBW $\leftarrow$ initialEffBW;
  nodeLimit $\leftarrow$ initialNodeLimit;
  while (true) {
    $\varphi_u$ $\leftarrow$ underApprox($\varphi$, effBW);
    (b_must, b_may) $\leftarrow$ solveAbstract($\varphi_u$, nodeLimit);
    if (b_must != $\boxed{0}$) return SAT;
    if (b_must == $\boxed{0}$ and $\varphi$ == $\varphi_u$) return UNSAT;
    if (b_must != b_may) {
      nodeLimit $\leftarrow$ increaseNodeLimit(nodeLimit);
    }
    else if ($\varphi$ != underApprox($\varphi$, effBW)) {
      effBW $\leftarrow$ increaseEffBW(effBW);
    }
  }
}
```

The formula approximations are performed on formulas in negation normal form by reducing the *effective bit-width* of selected variables by fixing some of their bits to chosen values. The underapproximations are obtained by decreasing effective bit-widths of all existentially quantified variables and the overapproximations are obtained by decreasing effective bit-widths of all universally quantified variables. Q3B tries to solve the input formula by solving the original formula, underapproximations of the formula, and overapproximations of the formula in parallel. We have integrated the proposed operation abstractions into the functions for solving underapproximations and overapproximations. The function solving the original formula can be adjusted to use operation abstractions as well, but the tool performs better if we keep this function unchanged.

Listing 1.4 shows the simplified implementation of solving underapproximations. The algorithm starts with the small initial values of the effective bit-width `effBW` for existential variables and the limit `nodeLimit` on the number of BDD nodes in the results of arithmetic operations. It repeatedly tries to solve the input formula and if the result is not determined, either the effective bit-width or the node limit is increased:

- if operation abstractions caused an imprecision, the node limit is increased;
- if the operation abstractions were precise, but the reduced effective bit-width could have caused imprecision, the effective bit-width is increased.

Currently, the initial effective bit-width is 1 and it is increased to $2, 4, 6, 8, \ldots$. The initial node limit is 1000 and the function `increaseNodeLimit()` multiplies it by 4 each time. The implementation for solving overapproximations is analogous.

Table 1: Numbers of benchmarks solved by the individual solvers divided by the satisfiability/unsatisfiability and the benchmark family.

	benchmark family	Boolector	CVC4	Z3	Q3B	Q3B+OA
UNSAT	heizmann	14	**107**	21	93	94
	preiner-keymaera	3919	3919	**3922**	3786	3906
	preiner-psyco	**62**	**62**	**62**	45	49
	preiner-scholl-smt08	**71**	37	68	52	69
	preiner-tptp	55	**56**	**56**	**56**	**56**
	preiner-ua	**137**	**137**	**137**	**137**	**137**
	wintersteiger-fixpoint	74	**75**	74	**75**	**75**
	wintersteiger-ranking	**20**	14	19	19	19
	Total UNSAT	4352	**4407**	4359	4263	4405
SAT	heizmann	17	18	13	19	**20**
	preiner-keymaera	**108**	78	**108**	104	104
	preinr-psyco	**131**	129	**131**	**131**	**131**
	preiner-scholl-smt08	248	215	203	239	**256**
	preiner-tptp	**17**	**17**	**17**	**17**	**17**
	preiner-ua	15	14	**16**	**16**	**16**
	wintersteiger-fixpoint	45	51	36	**54**	**54**
	wintersteiger-ranking	21	32	35	**40**	**40**
	Total SAT	602	554	559	620	**638**
	Total	4954	4961	4918	4883	**5043**

6 Experimental Evaluation

We have compared Q3B with our implementation of operation abstractions (referenced as Q3B+OA) against the original Q3B [9] and state-of-the-art SMT solvers Z3 [7], Boolector [15], and CVC4 [1]. We used Z3 in the version 4.6.0, Boolector in the version that entered SMT-COMP 2017, and CVC4 in the version presented in the paper by Niemetz et al. [14]. We evaluated all 5 solvers on all 5151 quantified bit-vector formulas from the SMT-LIB repository [2] used in SMT-COMP 2017. The used benchmarks are divided into three benchmark sets: benchmarks from the tool Ultimate Automizer by M. Heizmann (marked as *heizmann*), benchmarks that were created by converting integer and real arithmetic benchmarks to bit-vectors by M. Preiner (marked as *preiner*), and benchmarks from software and hardware verification by C. M. Wintersteiger (marked as *wintersteiger*). The benchmark sets *preiner* and *wintersteiger* are further divided into smaller families of benchmarks.

All experiments were performed on a Debian machine with two six-core Intel Xeon E5-2620 2.00GHz processors and 128 GB of RAM. Each benchmark run was limited to use 16 GB of RAM and 90 minutes of CPU time. All measured times are

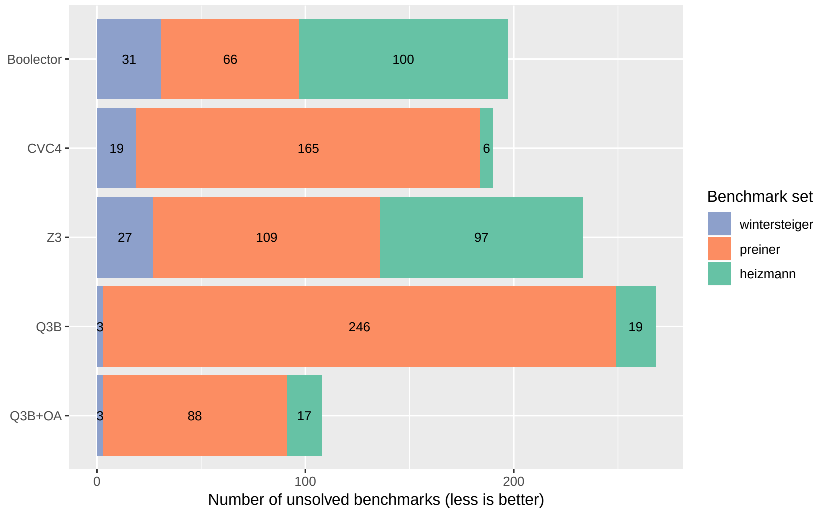

Fig. 4: The number of benchmarks unsolved by the individual solvers. The benchmarks are divided by the source of the benchmark.

CPU times. For reliable benchmarking we employed BenchExec [4], a tool that allocates resources for a program execution and measures their use precisely.

Table 1 shows numbers of solved benchmarks by the individual solvers. In total, Q3B+OA was able to solve 160 more benchmarks than the original version of Q3B. Moreover, Q3B+OA solved more benchmarks than other state-of-the art SMT solvers: 89 more than Boolector, 82 more than CVC4, and 125 more than Z3. We have also evaluated the effect of formula transformations described in Subsection 5.1. The transformations helped only in two families of benchmarks: in the family *preiner-keymaera* the optimizations were necessary to solve 116 out of 120 newly decided benchmarks; in the family *wintersteiger-fixpoint* the solver Q3B+OA solved 1 benchmark less without the optimizations.

From the opposite point of view, Figure 4 shows the number of benchmarks *unsolved* by the individual solvers. This graph shows that most of the benefit of abstractions is on formulas from the *preiner* benchmark set, where expensive operations like multiplication and division are frequently used.

Naturally, due to the repeated refinement of the abstractions, some benchmarks may require more solving time than without abstractions. In particular, there is one benchmark in the *heizmann* benchmark set that was solved by the original Q3B but not by Q3B+OA. Although this was the only such benchmark, the additional cost of abstractions is observable also on some benchmarks that were decided both with and without abstractions: computing abstractions slowed Q3B by more than 0.5 seconds on 115 benchmarks. On the other hand, there were 96 benchmarks decided by both versions of Q3B on which the version with abstraction was faster by more than 0.5 seconds.

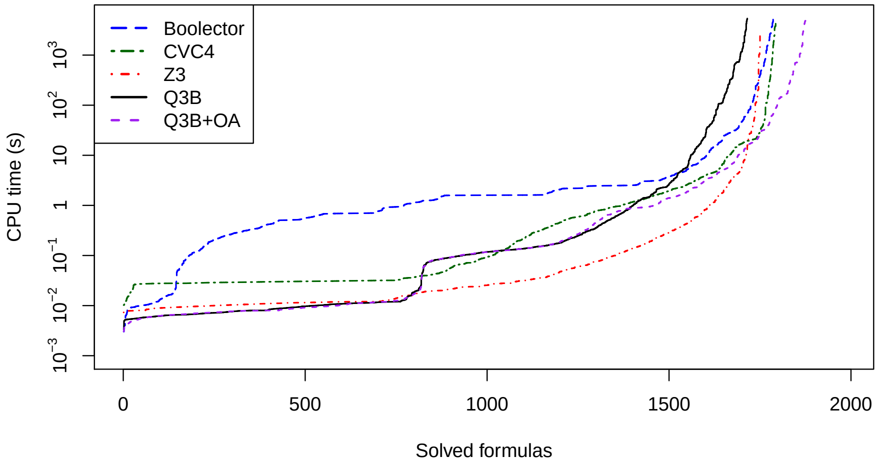

Fig. 5: Quantile plot of all solved non-trivial benchmarks from the SMT-LIB repository. Trivial benchmarks are those that all solvers solved within 0.1 s. The plot shows the number of non-trivial benchmarks (x-axis) that each solver was able to decide within a given CPU time limit (y-axis).

To compare the solving times of all solvers, Figure 5 shows quantile plots of solving times of *non-trivial benchmarks* for the individual solvers. We have filtered out 3168 trivial benchmarks, i.e., the benchmarks that were decided by all of the solvers in less than 0.1 s.

The detailed results of the evaluation, including the raw data files and further analyses, such as cross comparisons and scatter plots, are available at:

`http://fi.muni.cz/~xstrejc/ictac2018/`

7 Conclusions

We have presented operation abstractions that allow BDD-based SMT solvers to decide a formula by computing only some bits of results of arithmetic operations. The experimental evaluation shows that by using these abstractions, BDD-based SMT solver Q3B is able to solve more quantified bit-vector formulas from the SMT-LIB repository than state-of-the-art solvers Boolector, CVC4, and Z3.

In the implemented version, the solver computes overapproximations and underapproximations independently. It could be interesting to investigate whether sharing of the information obtained by an overapproximation with other parallel computations of the solver could improve the performance even more.

References

1. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.
2. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
3. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.
4. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and Resource Measurement. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings*, volume 9232 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 2015.
5. Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
6. Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013.
7. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
8. Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 120–135, 2009.
9. Martin Jonáš and Jan Strejček. Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2016.
10. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.
11. Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *LNCS*, pages 381–396. Springer, 2013.
12. Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. SymDIVINE: Tool for control-explicit data-symbolic state space exploration. In *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, pages 208–213, 2016.
13. Peter Navrátil. Adding Support for Bit-Vectors to BDD Libraries CUDD and Sylvan. Bachelor's thesis, Masaryk University, Faculty of Informatics, Brno, 2018.
14. Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. On solving quantified bit-vectors using invertibility conditions. *CoRR*, abs/1804.05025, 2018.

15. Mathias Preiner, Aina Niemetz, and Armin Biere. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 264–280. Springer, 2017.
16. Fabio Somenzi. CUDD: CU Decision Diagram Package Release 3.0.0. *University of Colorado at Boulder*, 2015.
17. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.
18. Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.
19. Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding bitvector formulas with mcsat. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 249–266, 2016.