Truncating Abstraction of Bit-Vector Operations for BDD-based SMT Solvers[☆,☆☆]

Martin Jonáš[a,*], Jan Strejček[a]

[a] *Faculty of Informatics, Masaryk University*
Botanická 68a, 602 00, Brno, Czech Republic

Abstract

During the last few years, BDD-based SMT solvers proved to be competitive in deciding satisfiability of quantified bit-vector formulas. However, these solvers usually do not perform well on input formulas with complicated arithmetic. Hitherto, this problem has been alleviated by approximations reducing effective bit-widths of bit-vector variables. In this paper, we propose an orthogonal abstraction technique that works 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 SMT solvers Boolector, CVC4, and Z3.

Keywords: SMT solving, bit-vector theory, operation abstraction, Q3B

1. Introduction

Complexity of computer systems as well as their importance for human society grow unceasingly. A possible way to improve reliability of these systems is by application of formal verification. Several approaches to formal verification, such as symbolic execution or bounded model checking, rely on the ability to decide whether a given formula over some logical theory is satisfiable. To this end, many 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 the 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

[☆] This is an extended version of a paper presented at ICTAC 2018 [12].
[☆☆] Authors of this work are supported by the Czech Science Foundation grant GA23-06506S.
[*] Corresponding author
Email addresses: `martin.jonas@mail.muni.cz` (Martin Jonáš), `strejcek@mail.muni.cz` (Jan Strejček)

functions, or loop summaries, the arising formulas also naturally contain quantifiers [9, 21, 7, 15, 16].

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 [22], Boolector [19], and CVC4 [18]. Moreover, new tools that aim for precisely this theory, such as the solver Q3B [12, 13], were developed. Approaches of these tools fall into two categories. Solvers Z3, Boolector, and CVC4 apply variants of quantifier instantiation [22, 19, 18] that iteratively produces quantifier-free formulas that can be solved by a solver for quantifier-free bit-vector formulas. The solver Q3B uses Binary Decision Diagrams (BDDs) [5] to decide satisfiability of quantified bit-vector formulas.

In principle, the BDD-based approach builds a BDD that represents all models of the given bit-vector formula. The BDD is constructed in a bottom-up manner, i.e., the BDD for the formula is computed from BDDs for its subformulas. Recall that a model of a bit-vector formula is an assignment of bit-vectors to free bit-vector variables that makes the formula valid. As BDDs work with Boolean variables, each bit-vector variable x of bit-width k is identified with a sequence $x_{k-1}x_{k-2}\ldots x_1x_0$ of k Boolean variables corresponding to individual bits of x, where x_0 corresponds to the least-significant bit. The approach that constructs a BDD for all models suffers from the well-known issues of BDDs: the size of a BDD can heavily depend on the chosen ordering of Boolean variables and there exist formulas for which *all* corresponding BDDs are prohibitively big, regardless the variable ordering. For example, consider the formula $x \cdot y = z$ with multiplication and free bit-vector variables x, y, z of bit-width k. Any BDD that represents all models of this formula is guaranteed to have exponentially many nodes in k regardless of the chosen order of Boolean variables [6]. In practice, if a formula contains complicated arithmetic, the constructed BDDs tend to grow in size very quickly.

The solver Q3B tries to reduce sizes of constructed BDDs in several ways [10]. One of them is the computation of a suitable ordering of Boolean variables from the formula structure (see the original paper [10] for details). Another way is to restrict the domains of some bit-vector variables in order to use fewer Boolean variables. We say that we reduce the *effective bit-width* of these variables. For example, consider the mentioned formula $x \cdot y = z$ where all variables have bit-width 32. We can reduce the effective bit-width of each variable to 2 by fixing the 30 most-significant bits to 0. We obtain the formula

$$0\ldots0x_1x_0 \;\cdot\; 0\ldots0y_1y_0 \;=\; 0\ldots0z_1z_0$$

with only 6 Boolean variables. Note that the computation of a BDD representing all models of this formula is instantaneous, while it is infeasible for the original formula. As an alternative to fixing k most significant bits to 0, Q3B can also restrict a variable domain by the assumption that k most significant bits have the same value. For example, application of this restriction to all variables in the considered formula for $k = 31$ yields the formula

$$x_1 \dots x_1 x_0 \ \cdot \ y_1 \dots y_1 y_0 \ = \ z_1 \dots z_1 z_0$$

also with 6 Boolean variables. Again, a BDD representing all models of this formula can be computed immediately. Both modified formulas can be seen as underapproximations of the original formula as every model of the modified formulas directly translates to a model of the original formula. As both underapproximations are satisfiable, any of them can be used to conclude that the original formula is also satisfiable. Note that we can restrict the domain of each bit-vector variable separately. Given a bit-vector formula in negation normal form, Q3B computes underapproximations by restricting the domains of all free and existentially quantified variables. If an underapproximation is satisfiable, so is the original formula. Analogously, Q3B also computes overapproximations by restricting the domains of all universally quantified variables. If an overapproximation is unsatisfiable, so is the original formula.

Although the approximations made Q3B competitive with state-of-the-art SMT solvers, the approach has several drawbacks. For example, the original version of Q3B cannot solve satisfiability of simple formulas such as

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

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

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

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. Unfortunately, the mentioned approximations do not help as the formulas are unsatisfiable and contain no universally quantified variables that could be used to overapproximate the formula in order to prove its unsatisfiability.

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 since $(x < 2) \land (x > 4)$ alone in unsatisfiable. 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 while the right side of the equality states that it is one. 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 reasoning about partial results of selected operations including non-linear multiplications. In these partial results, some bits are set to a distinguished *do-not-know* value. Formally, the paper defines abstract domains in which the operations can handle and produce *do-not-know* values and shows that these abstract domains can be used to decide satisfiability of an input formula.

Abstract domains with *do-not-know* values have the same goal as the approximation techniques based on effective bit-width reductions: to decide satisfiability of the given quantified bit-vector formula without construction of large BDDs. While the effective bit-width reduction lower the number of Boolean variables that can appear in these BDDs, the *do-not-know* values are used when BDDs representing some term exceed a given number of BDD nodes. The two approaches are in fact complementary and we also show how to combine them.

The paper is structured as follows. Section 2 provides necessary background and notations for SMT, bit-vector theory, and binary decision diagrams. Section 3 presents a general definition of abstract domains for terms and formulas and shows how they can help decide satisfiability of a formula. Section 4 introduces truncating term and formula abstract domains that compute only several bits from results of arithmetic bit-vector operations. Section 5 describes our implementation of these abstract domains and their deployment in the SMT solver Q3B. The section also explains some heuristics that further improve the performance of Q3B, especially in combination with the truncating abstract domains. Finally, Section 6 provides a comparison of Q3B extended with the truncating abstract domains against the original Q3B and other SMT solvers for quantified bit-vector formulas.

A preliminary version of this paper has been presented at ICTAC 2018 [12]. The current version is extended in several directions. We consider a richer version of bit-vector logic that includes also *if-then-else* construct. Further, we provide more details about the technique including the proofs omitted in the preliminary version and algorithms computing bit-vector multiplication and unsigned inequality in truncating abstract domains. Moreover, we have added some heuristics further improving performance of our tool. These heuristics are explained in the section devoted to implementation and deployment of truncating abstract domains, namely in Subsections 5.1 and 5.2. The experimental evaluation has been done again with the latest versions of relevant SMT solvers and with the latest version of benchmarks from the SMT-LIB repository[1]. The evaluation now considers two additional Q3B configurations to show the contribution of truncating abstract domains and other introduced heuristics to the overall performance improvements. The comparison of Q3B extended with the truncating abstract domains and other heuristics against the original Q3B is shown in more details.

2. Preliminaries

First of all, we recall the basics of bit-vector theory and binary decision diagrams. Then we remind some algorithms for BDD operations and explain how they can be used to decide satisfiability of bit-vector formulas.

[1]We used the current versions of the solvers and benchmarks that were available at the time of the major revision of this paper, i.e., December 2020.

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 terms, atomic formulas, and formulas of a many-sorted logic. If the reader is interested, these definitions can be found for example in Barrett and Tinelli [3].

The bit-vector theory is a many-sorted first-order theory with infinitely many sorts corresponding to bit-vectors of various lengths. The length of a bit-vector is traditionally called its *bit-width.* The BV theory uses only three predicates, namely *equality* ($=$), *unsigned inequality* of bit-vectors interpreted as binary-encoded natural numbers ($\leq_u$), and *signed inequality* of bit-vectors interpreted as integers in two's complement representation ($\leq_s$). The theory contains various binary 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* ($\gg$), and *concatenation* (concat). All predicates and binary functions except concatenation are applied to two terms of the same bit-width. The theory also includes several unary functions including *extraction* of n bits starting from position p ($\mathsf{extract}_p^n$), *extension with n zeroes* as the most-significant bits ($\mathsf{zeroExtend}_n$), and *extension with n copies of the most-significant bit* ($\mathsf{signExtend}_n$). Further, the signature of BV theory contains constants $c^{[n]}$ for each bit-width $n > 0$ and a number $0 \leq c \leq 2^n - 1$. Finally, the theory also uses the *if-then-else* construct $\mathtt{ite}(\varphi, t_1, t_2)$ that evaluates as term t_1 if φ holds and as t_2 otherwise. We denote the set of all bit-vectors as $\mathcal{BV}$, the set of all bit-vector variables as *Vars*, the set of all terms as $\mathcal{T}$, and the set of all formulas as $\mathcal{F}$.

For an assignment μ that assigns to each variable from *Vars* a value in its domain, $[\![\_]\!]_\mu$ denotes the evaluation function which assigns to each term or formula the value obtained by substituting all free variables x by their values $\mu(x)$ and evaluating all functions, predicates, logic operators etc. A formula φ is *satisfiable* if $[\![\varphi]\!]_\mu = 1$ for some assignment μ; it is *unsatisfiable* otherwise.

If φ is a formula and t, s are terms, $\varphi[t \mapsto s]$ denotes the formula φ with all occurrences of the term t simultaneously substituted by the term s. If μ is an assignment, x a variable, and v a value in the domain of v, $\mu[x \mapsto v]$ denotes the assignment identical to μ for all variables except x, to which it assigns v.

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. [14].

2.2. Binary Decision Diagrams

A *binary decision diagram* (BDD) is a data structure that can succinctly represent Boolean functions. Formally, it is a rooted directed acyclic graph that has at most two leaves (i.e., nodes without any outgoing edge), which are labelled by 0 and 1. Each inner node is labelled by a Boolean variable and it has two outgoing edges called *high* and *low*, which are related to the potential values 1 and 0 of the corresponding variable, respectively. A BDD maps each assignment μ of Boolean variables either to 0 or to 1 by the following process:

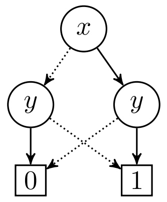


Figure 1: A BDD for $x \iff y$.

1. Start in the root node.
2. Repeat this step until a leaf is reached: Let x be the variable labelling the current node. If $\mu(x) = 1$ then follow the high edge. If $\mu(x) = 0$ then follow the low edge.
3. Return the value 0 or 1 labelling the reached leaf.

The value to which a BDD b maps an assignment μ is denoted by $[\![b]\!]_\mu$.

An example of a BDD is provided in Figure 1. According to the traditional notation, high edges are drawn by solid lines and low edges are drawn by dotted lines. The BDD on the figure maps to 1 precisely the assignments where the values of x and y are the same. Hence, it represents the Boolean function $x \iff y$. The trivial BDDs $\boxed{0}$ and $\boxed{1}$ represent constant Boolean functions *false* (0) and *true* (1), respectively.

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 the variables that occur on both of the paths is the same. A BDD is *reduced* if there is no inner node with high and low edges leading to the same node. It has been shown that reduced and ordered BDDs are *canonical*, i.e., given a variable ordering, there is exactly one reduced and ordered 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 Boolean variables. Such a function of a bit-width k (i.e., the produced bit-vector has the bit-width k) can be represented by a vector of k BDDs $\overline{b} = (b_{k-1}, \ldots, b_1, b_0) = (b_i)_{0 \leq i < k}$, where b_0 represents the least-significant bit of the function result. The result for an assignment μ is then the bit-vector $[\![\overline{b}]\!]_\mu = ([\![b_i]\!]_\mu)_{0 \leq i < k}$. 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 BDD and the set of all vectors of BDDs as $\overline{\mathsf{BDD}}$. We use the overlined symbols for vectors of BDDs.

2.3. Operations on Binary Decision Diagrams

It has been shown by Bryant [5] that, given a pair of BDDs for Boolean functions f and g, one can compute a BDD for function $f \wedge g$ and a BDD for function $f \vee g$ in polynomial time. We denote the BDD operations corresponding

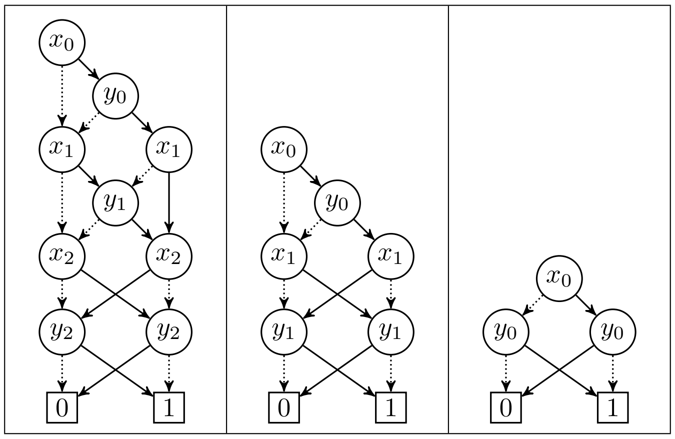


Figure 2: Vector of BDDs representing the addition $x_2x_1x_0 + y_2y_1y_0$ of two bit-vectors of bit-width 3.

to *conjunction* and *disjunction* by the infix operators & and |, respectively. A BDD for the *negation* of f can be obtained from a BDD for f simply by swapping the labels 0 and 1 of leaves. We use the operator ! to denote this BDD operation.

Using these basic BDD operations, we can easily define more operations. In the following, we employ the operations

- *equivalence* denoted as $a \leftrightarrow b$, which abbreviates $(a \,\&\, b) \mid (!a \,\&\, !b)$,
- *exclusive or* denoted as $a \operatorname{xor} b$, which abbreviates $(a \,\&\, !b) \mid (!a \,\&\, b)$, and
- *if-then-else* denoted as $(a \,?\, b : c)$, which abbreviates $(a \,\&\, b) \mid (!a \,\&\, c)$.

Bryant [5] has also described an operation that modifies a given BDD b by setting a selected Boolean variable y to a given value $v \in \{0, 1\}$. Let $b_{y \mapsto v}$ denote the result of this operation. With this operation, we can define two operations corresponding to *existential* and *universal quantification*. For quantification of a Boolean variable y in a BDD b, we set

- $bdd_{\forall}(y, b)$ to be an abbreviation for $b_{y \mapsto 0} \,\&\, b_{y \mapsto 1}$ and
- $bdd_{\exists}(y, b)$ to be an abbreviation for $b_{y \mapsto 0} \mid b_{y \mapsto 1}$.

Then we extend these two operations to handle a bit-vector variable $x = x_{k-1}x_{k-2} \ldots x_0$ as their first argument. Formally,

- $bdd_{\forall}(x, b)$ stands for $bdd_{\forall}(x_{k-1}, bdd_{\forall}(x_{k-2}, \ldots, bdd_{\forall}(x_0, b) \ldots))$ and
- $bdd_{\exists}(x, b)$ stands for $bdd_{\exists}(x_{k-1}, bdd_{\exists}(x_{k-2}, \ldots, bdd_{\exists}(x_0, b) \ldots))$.

Note that the operations above are not computed by the compositions of the basic operations in practice, but are usually implemented by dedicated algorithms.

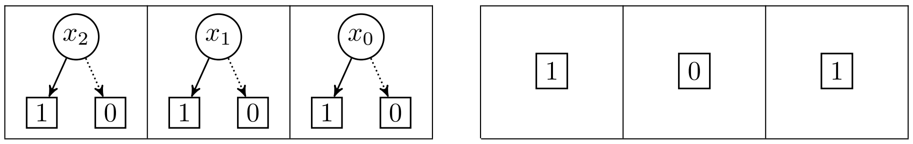


Figure 3: Vectors of BDDs representing variable $x_2x_1x_0$ of bit-width 3 (left) and constant $5^{[3]} = 101$ (right).

With the mentioned BDD operations, we can implement a function $\texttt{fo2BDD} : \mathcal{F} \to \texttt{BDD}$ that translates a given bit-vector formula φ into a BDD $\texttt{fo2BDD}(\varphi)$ that represents exactly all models of the formula, i.e., for each assignment μ it holds that $[\![\varphi]\!]_\mu = [\![\texttt{fo2BDD}(\varphi)]\!]_\mu$. This function provides a straightforward satisfiability check: a formula φ is satisfiable if and only if the BDD $\texttt{fo2BDD}(\varphi)$ maps at least one assignment to 1. For ordered and reduced BDDs, this is always the case unless the BDD is $\boxed{0}$. In the rest of this section, we intuitively describe the algorithm computing `fo2BDD`.

A given bit-vector formula is translated to a corresponding BDD in a bottom-up manner. The translation of terms is performed by an auxiliary function $\texttt{te2}\overline{\texttt{BDD}} : \mathcal{T} \to \overline{\texttt{BDD}}$ which translates a given term t to a vector of BDDs that represents the bit-vector function corresponding to t. Translation of variables and constants is straightforward, as illustrated on Figure 3. Let t_1 and t_2 be two terms of the same bit-width, which are already translated to vectors of BDDs $\overline{a}, \overline{b} \in \overline{\texttt{BDD}}$, respectively. Then the vector of BDDs for term $t_1 + t_2$ can be constructed by the function $\texttt{bvec\_add(}\overline{a}\texttt{, }\overline{b}\texttt{)}$ presented in Listing 1. The listing also provides the function $\texttt{bvec\_mul(}\overline{a}\texttt{, }\overline{b}\texttt{)}$, which constructs the vector of BDDs for the term $t_1 \cdot t_2$. The algorithm starts with the result set to 0 and, for each bit b_i of the second argument $\overline{b}$, the result stays the same if b_i is 0 and $2^i \cdot \overline{a}$ is added to the result if b_i is 1. Note that the inner `for`-cycle responsible for this addition ignores the first i least-significant bits as $2^i \cdot \overline{a}$ has 0 in these bits. The computation of $2^i \cdot \overline{a}$ is performed by iteratively shifting $\overline{a}$ left by one bit using the function `bvec_shl`, which shifts the vector of BDDs in its first argument by the number of bits given by the second argument and fills the vacant least-significant bits with the BDD $\boxed{0}$. All other functions of the bit-vector theory such as *unsigned division*, *bitwise and*, *concatenation*, or *extraction* are implemented similarly. A term of the form $\texttt{ite}(\varphi, t_1, t_2)$ is translated to a vector of BDDs by the function $\texttt{bvec\_ite(}a\texttt{, }\overline{b}\texttt{, }\overline{c}\texttt{)}$ provided also in Listing 1, where the BDD a represents the models of φ and $\overline{b}, \overline{c} \in \overline{\texttt{BDD}}$ correspond to t_1, t_2, respectively.

When terms t_1 and t_2 of the same bit-width are translated to the corresponding vectors of BDDs, we can compute the BDD for their *equality* $t_1 = t_2$, the BDD for their *unsigned inequality* $t_1 \leq_u t_2$, and the BDD for their *signed inequality* $t_1 \leq_s t_2$. Listing 2 shows the algorithm `bvec_eq` for *equality* and `bvec_leq` for *unsigned inequality*. The algorithm for *signed inequality* is similar. Finally, if we have a BDD b_1 for φ_1 and a BDD b_2 for φ_2, we can compute BDDs for the formula $\neg\varphi_1$ as $!b_1$, the formula $\varphi_1 \wedge \varphi_2$ as b_1 & b_2, the formula $\varphi_1 \vee \varphi_2$ as $b_1 \mid b_2$, the formula $\forall x\, (\varphi_1)$ as $bdd_\forall(x, b_1)$, and the formula $\exists x\, (\varphi_1)$ as $bdd_\exists(x, b_1)$.

Listing 1: Functions bvec_add and bvec_mul implementing *addition* (+) and *multiplication* ($\cdot$) on vectors $\overline{a} = (a_i)_{0\leq i<k}$ and $\overline{b} = (b_i)_{0\leq i<k}$ of BDDs, and function bvec_ite implementing *if-then-else* (ite) construct on BDD a and vectors $\overline{b} = (b_i)_{0\leq i<k}$ and $\overline{c} = (c_i)_{0\leq i<k}$ of BDDs.

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

bvec_mul($\overline{a}$, $\overline{b}$)
{
  $\overline{result}$ ← ($\boxed{0}, \boxed{0}, \ldots, \boxed{0}$) with the bit-width $k$;
  for $i$ from $0$ to $k-1$ {
    $\overline{added}$ ← bvec_add($\overline{result}, \overline{a}$);
    for $j$ from $i$ to $k-1$ {
      $result_j$ ← $(b_i \mathbin{?} added_j : result_j)$;
    }
    $\overline{a}$ ← bvec_shl($\overline{a}, 1$);
  }
  return $\overline{result}$;
}

bvec_ite($a$, $\overline{b}$, $\overline{c}$)
{
  for $i$ from $0$ to $k-1$ {
    $result_i$ ← $(a \mathbin{?} b_i : c_i)$;
  }
  return $\overline{result}$;
}
```

Note that all the mentioned BDD operations including the algorithms of Listings 1 and 2 are implemented for example in the BDD package BuDDy[2]. Hence, one can easily implement the functions te2$\overline{\text{BDD}}$ and fo2BDD translating bit-vector terms and formulas to the corresponding (vectors of) BDDs, respectively. Let us note that these implementations are mutually recursive: fo2BDD calls te2$\overline{\text{BDD}}$ to translate the subterms of the input formula to the corresponding vectors of BDDs and te2$\overline{\text{BDD}}$ calls fo2BDD to translate the first argument of a term ite(φ, t_1, t_2) to the corresponding BDD.

Unfortunately, the computation of fo2BDD(φ) is infeasible even for some short formulas φ mentioned in Introduction as it aims to construct prohibitively large BDDs.

[2]http://sourceforge.net/projects/buddy

Listing 2: Functions `bvec_eq` and `bvec_leq` implementing *equality* ($=$) and *unsigned inequality* ($\leq_u$) of vectors $\overline{a} = (a_i)_{0\leq i<k}$ and $\overline{b} = (b_i)_{0\leq i<k}$ of BDDs.

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

bvec_leq($\overline{a}$, $\overline{b}$)
{
  $\mathit{result}$ ← $\boxed{1}$;
  for $i$ from 0 to $k-1$ {
    $\mathit{result}$ ← (!$a_i$ & $b_i$) | ($\mathit{result}$ & ($a_i \leftrightarrow b_i$))
  }
  return $\mathit{result}$;
}
```

3. Term and Formula Abstractions

Although it is often infeasible to compute $\mathtt{fo2BDD}(\varphi)$ precisely, even an imprecise result can sometimes be enough to decide satisfiability of φ as illustrated in Introduction. In this section, we describe general notions of a *term abstract domain*, which captures an imprecise computation of $\mathtt{te2}\overline{\mathtt{BDD}}$, and a *formula abstract domain*, which captures an imprecise computation of $\mathtt{fo2BDD}$. 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* $[\![\_]\!]^A_\_ \colon A \times \mathcal{BV}^{\mathit{Vars}} \to 2^{\mathcal{BV}}$ *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. Formally, the precise BDD term abstract domain is the triple $(\overline{\mathtt{BDD}}, \mathtt{te2}\overline{\mathtt{BDD}}, [\![\_]\!]^{\overline{\mathtt{BDD}}}_\_)$, where $[\![\overline{a}]\!]^{\overline{\mathtt{BDD}}}_\mu$ is the singleton set $\{bv\}$ such that bv is the bit-vector obtained by evaluation of vector $\overline{a}$ of BDDs with the assignment μ, i.e., $bv = [\![\overline{a}]\!]_\mu$. Note that this abstract domain serves only as an artificial example as it does not bring any real abstraction. Nevertheless, it enjoys two interesting properties: for each term t and assignment μ, the corresponding abstract object evaluates to the set of values that contains the correct value $[\![t]\!]_\mu$ and it does not contain any incorrect value. These properties are called *completeness* and *soundness*, respectively.

Definition 2. *A term abstract domain* $(A, \alpha, [\![\_]\!]^A_\_)$ *is* complete *if each term*

$t \in \mathcal{T}$ and each assignment μ satisfy $[\![t]\!]_\mu \in [\![\alpha(t)]\!]^A_\mu$. 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 a 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 $[\![\varphi]\!]_\mu \in [\![\alpha(\varphi)]\!]^A_\mu$. 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 the triple $(\mathtt{BDD}, \mathtt{fo2BDD}, [\![\_]\!]^{\mathtt{BDD}}_\_)$, where $[\![a]\!]^{\mathtt{BDD}}_\mu$ is the singleton set $\{b\}$, where $b \in \{0, 1\}$ 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.*

Proof. Suppose that there is an assignment μ such that $[\![\alpha(\varphi)]\!]^A_\mu = \{1\}$. Since the abstract domain is complete, we know that $[\![\varphi]\!]_\mu \in \{1\}$. Therefore $[\![\varphi]\!]_\mu = 1$ and φ is indeed satisfiable.

For the second claim, suppose that all assignments μ satisfy $[\![\alpha(\varphi)]\!]^A_\mu = \{0\}$. Again, from the completeness we know that $[\![\varphi]\!]_\mu \in \{0\}$ for all assignments μ. Therefore $[\![\varphi]\!]_\mu = 0$ for any assignment μ and φ is indeed unsatisfiable. □

One can define various abstract domains in the context of bit-vector logic, for example, a complete term abstract domain that tracks only bits with a known fixed value (0 or 1). Another example of a complete term abstract domain is a domain that computes an interval containing all possible values of a given term. These abstractions are computationally cheap but very coarse.

Abstract domains used in practice have usually some parameters that enable their applications to find the balance between precision and performance. For example, the approximations mentioned in Introduction, which reduce effective bit-width, can be expressed as an abstract domain with parameters. Namely, the parameters specify which bits will be fixed and how they will be fixed. The following section presents other adjustable abstract domains, namely the truncating term and formula abstract domains.

4. Truncating Term and Formula Abstract Domains

This section describes a term abstract domain and a formula abstract domain that allow *truncating* results of bit-vector operations, i.e., computing only several bits from the results of arithmetic bit-vector operations. As formulas contain terms and terms can contain formulas due to the construct $\mathtt{ite}(\varphi, t_1, t_2)$, the definitions of the two abstract domains are interdependent. Therefore we first define the *truncating formula abstract domain* without precise specification of its abstraction function. Then we introduce the *truncating term abstract domain* and discuss its properties. After that, we return back to the truncating formula abstract domain to present its abstraction function and prove its completeness. Note that although we present the two abstraction functions separately, if the functions should be defined fully formally, they have to be mutually recursive.

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

4.1. Truncating Formula Abstract Domain: Part One

In our truncating approach, some bits of bit-vectors may not be computed and their precise values may be unknown. Intuitively, the truncating formula abstract domain interprets the unknown bits simultaneously in two ways: as a pessimist and as an optimist. If the truth value of some atomic formula depends on unknown bits, the pessimist assumes that the atomic formula is false while the optimist assumes that it is true. More precisely, the abstract objects of the *truncating formula abstract domain* are BDD pairs (b_{must}, b_{may}), where b_{must} (the pessimist) determines the assignments that satisfy the formula for all possible values of unknown bits and b_{may} (the optimist) determines the assignments that satisfy the formula for at least one value of the unknown bits. Hence, b_{must} represents a subset of all formula models while b_{may} represents a superset of all formula models.

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

$$(\mathsf{BDDpair}, \mathsf{fo2BDDpair}, [\![\_]\!]^{\mathsf{BDDpair}}_{\_}),$$

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

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

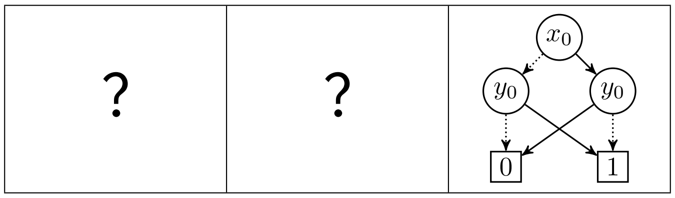


Figure 4: Truncated result of addition $x_2x_1x_0 + y_2y_1y_0$ of two bit-vectors of bit-width 3.

Observe that $[\![(b_{must}, b_{may})]\!]_\mu^{\mathsf{BDDpair}}$ 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 $[\![b_{must}]\!]_\mu = 0$, $[\![b_{may}]\!]_\mu = 1$. The result would be $\emptyset$ in the remaining case $[\![b_{must}]\!]_\mu = 1$, $[\![b_{may}]\!]_\mu = 0$, but this situation never happens for the BDD pairs produced by the abstraction function `fo2BDDpair`.

4.2. Truncating Term Abstract Domain

In the *truncating term abstract domain*, 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 any of 0 and 1.

For example, Figure 4 shows the result of computing only the least-significant bit of an addition of two bit-vectors $x_2x_1x_0+y_2y_1y_0$ (compare to Figure 2). 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

$$(\overline{\mathsf{BDD?}}, \mathtt{te2}\overline{\mathsf{BDD?}}, [\![\_]\!]_\_^{\overline{\mathsf{BDD?}}}),$$

where the set $\overline{\mathsf{BDD?}}$ of abstract objects consists of vectors of BDDs and ? elements:

$$\overline{\mathsf{BDD?}} = \{(b_i)_{0 \le i < k} \mid k > 0, b_i \in \mathsf{BDD} \cup \{?\} \text{ for all } 0 \le i < k\}.$$

Further, the abstract evaluation function $[\![\_]\!]_\_^{\overline{\mathsf{BDD?}}}$ assigns to each $\bar{b} = (b_i)_{0 \le i < k} \in \overline{\mathsf{BDD?}}$ and an assignment μ the set of bit-vector values

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

There are multiple possible versions of the abstraction function $\mathtt{te2}\overline{\mathsf{BDD?}}$ including the following two:

1. the number of precisely computed bits is fixed and the remaining bits are set to ?,

2. the limit on the number of BDD nodes in the result of the operation is specified and after reaching it, the remaining bits are not computed and are set to ?.

We focus purely on the second option as our preliminary evaluation has shown that it outperforms the first one. Furthermore, it is easy to derive the implementation of the first option based on the description of the second option.

We suppose that the limit on BDD nodes is fixed for the given domain. Note that our procedure deciding satisfiability uses multiple abstract domains varying by the BDD node limit (see Section 5 for details).

The function $\mathtt{te2}\overline{\mathtt{BDD?}}$ is in fact very similar to the function $\mathtt{te2}\overline{\mathtt{BDD}}$ described in Subsection 2.3, which translates a term to a vector of BDDs precisely representing the bit-vector function given by the term. For variables and constants, both functions produce the same vectors of BDDs (illustrated with Figure 3). For more complex terms, the abstraction function $\mathtt{te2}\overline{\mathtt{BDD?}}$ is also computed recursively, but it differs from the precise function $\mathtt{te2}\overline{\mathtt{BDD}}$ in two important aspects.

1. The computation has to work correctly with ? elements. To achieve this, we modify the BDD operations &, |, $\leftrightarrow$, xor, and $(\_\,?\,\_\,:\,\_)$, which are the building blocks of the $\mathtt{te2}\overline{\mathtt{BDD}}$ computation. 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 [23]. The modified BDD operations $\&_t$, $|_t$, $\leftrightarrow_t$, xor_t, and $(\_\,?\,\_\,:\,\_)_t$ are computed as follows:

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

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

$$a \leftrightarrow_t b \;=\; \begin{cases} a \leftrightarrow b & \text{if } a, b \neq ? \\ ? & \text{otherwise} \end{cases}$$

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

$$(a\,?\,b : c)_t \;=\; \begin{cases} b & \text{if } a = \boxed{1} \text{ or } b = c \\ c & \text{if } a = \boxed{0} \\ (a\,?\,b : c) & \text{if } a \notin \{\boxed{0}, \boxed{1}, ?\} \text{ and } b, c \neq ? \text{ and } b \neq c \\ ? & \text{otherwise} \end{cases}$$

Note that $? \leftrightarrow_t ?$ and $? \,\mathrm{xor}_t\, ?$ are left unknown as each ? can represent a different value.

2. For terms with bit-vector arithmetic functions, the computation 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. Listing 3 provides algorithms that compute these *truncating* versions of addition and multiplication (compare to the original functions given in Listing 1). Both algorithms use the function `bddNodes`, which returns the total number of BDD nodes in a given vector of BDDs and `?` elements. In truncating addition, individual bits of the result are computed precisely until the node limit is reached and the remaining bits are set to `?`. The algorithm for multiplication is modified in a similar way: once the limit is reached, all bits of the result that could be modified in the rest of the precise algorithm are set to `?`. The algorithm uses function `bvec_shl_trunc`, which is a straightforward extension of `bvec_shl` to vectors $\overline{\texttt{BDD?}}$. The algorithms for other truncating bit-vector functions are similar. However, they may differ in the order in which the precise bits are produced: during the computation of addition and multiplication, the first precisely computed bits are the least significant ones; during the computation of division, the first precisely computed bits are the most-significant ones. Therefore if truncating addition or multiplication reaches the BDD node limit, the remaining most-significant bits are set to `?`, while truncating division sets to `?` the remaining least-significant bits.

Listing 3 also shows the function `bvec_ite_trunc` that computes the value of $\texttt{te2}\overline{\texttt{BDD?}}$ for terms $\texttt{ite}(\varphi, t_1, t_2)$. The function does not contain any test against the BDD node limit as the $\texttt{ite}(\_, \_, \_)$ construct is not an arithmetic operation and the size of its output is comparable to the size of its inputs. The first two arguments of $\texttt{bvec\_ite\_trunc}(a_{must}\texttt{,}\ a_{may}\texttt{,}\ \overline{b}\texttt{,}\ \overline{c})$ form the BDD pair (a_{must}, a_{may}) corresponding to the formula φ in the truncating formula abstract domain and the other two arguments $\overline{b}, \overline{c} \in \overline{\texttt{BDD?}}$ correspond to t_1, t_2, respectively. The BDD pair (a_{must}, a_{may}) is computed by the function $\texttt{fo2BDDpair}(\varphi)$ defined later. Here we assume that the values represented by the BDD pair always contain the correct value of φ, i.e., $[\![\varphi]\!]_\mu \in [\![(a_{must}, a_{may})]\!]_\mu^{\texttt{BDDpair}}$ for each assigment μ. Observe that the BDD a_{must} xor a_{may} represents the assignments μ for which the values of a_{must} and a_{may} differ and thus the BDD pair does not determine the value of $[\![\varphi]\!]_\mu$. The algorithm `bvec_ite_trunc` computes the bits of the result one by one. If $(a_{must} \text{ xor } a_{may})\ \ \&_t\ \ (b_i \text{ xor}_t\ c_i)$ is the BDD $\boxed{0}$, there is no assignment for which the value of φ is not determined by the BDD pair and i-th bit produced by the *then* branch differs from the i-th bit produced by the *else* branch. In other words, for each assignment μ, it either holds that $[\![a_{must}]\!]_\mu = [\![a_{may}]\!]_\mu = [\![\varphi]\!]_\mu$ or $[\![b_i]\!]_\mu = [\![c_i]\!]_\mu$ and thus the value of the i-th bit can be computed as $(a_{must}\,?\,b_i : c_i)_t$. On the other hand, if there is an assignment for which the BDD pair does not determine the value of φ and the results of the branches differ, we cannot produce a precise result and the algorithm thus sets the i-th bit to `?`.

The definition of $\texttt{te2}\overline{\texttt{BDD?}}$ directly guarantees that this function produces vectors where each element is either `?` or the BDD precisely representing the

Listing 3: Functions bvec_add_trunc and bvec_mul_trunc implement truncating versions of *addition* (+) and *multiplication* $(\cdot)$ on vectors $\overline{a} = (a_i)_{0\leq i<k}$ and $\overline{b} = (b_i)_{0\leq i<k}$ in $\overline{\mathsf{BDD?}}$. Function bvec_ite_trunc implements the ite(_, _, _) construct on the BDD pair a_{must}, a_{may} representing the value of its first argument in the truncating formula abstract domain and on two vectors $\overline{b} = (b_i)_{0\leq i<k}$ and $\overline{c} = (c_i)_{0\leq i<k}$ in $\overline{\mathsf{BDD?}}$.

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

bvec_mul_trunc($\overline{a}$, $\overline{b}$, nodeLimit)
{
  $\overline{result}$ ← ($\boxed{0}, \boxed{0}, \dots, \boxed{0}$) with the bit-width $k$;
  for $i$ from $0$ to $k-1$ {
    $\overline{added}$ ← bvec_add_trunc($\overline{result}, \overline{a}$);
    for $j$ from $i$ to $k-1$ {
      $result_j$ ← $(b_i \, ? \, added_j : result_j)_t$;
      if (bddNodes($\overline{result}$) > nodeLimit) {
        for $m$ from $i+1$ to $k-1$ {
          $result_m$ ← ?;
        }
        return $\overline{result}$;
      }
    }
    $\overline{a}$ ← bvec_shl_trunc($\overline{a}, 1$);
  }
  return $\overline{result}$;
}

bvec_ite_trunc($a_{must}$, $a_{may}$, $\overline{b}$, $\overline{c}$)
{
  for $i$ from $0$ to $k-1$ {
    if ((($a_{must}$ xor $a_{may}$) $\&_t$ ($b_i$ xor$_t$ $c_i$)) = $\boxed{0}$) {
      $result_i$ ← $(a_{must} \, ? \, b_i : c_i)_t$;
    } else {
      $result_i$ ← ?;
    }
  }
  return $\overline{result}$;
}
```

corresponding bit of a given term. This is formally stated by the following theorem.

Theorem 2. *Let $t \in \mathcal{T}$ be a term such that for each subterm* $\texttt{ite}(\varphi, t_1, t_2)$ *of t and for each assigment μ it holds $[\![\varphi]\!]_\mu \in [\![\texttt{fo2BDDpair}(\varphi)]\!]_\mu^{\texttt{BDDpair}}$. Then each element of* $\texttt{te2}\overline{\texttt{BDD?}}(t)$ *is either* ? *or it is equal to the corresponding element of* $\texttt{te2}\overline{\texttt{BDD}}(t)$.

The theorem is employed in the next subsection, where we prove completeness of the truncating term abstract domain and the truncating formula abstract domain. Note that the truncating term abstract domain is not sound, as an abstract object can describe also incorrect results.

4.3. Truncating Formula Abstract Domain: Part Two

The truncating formula abstract domain $(\texttt{BDDpair}, \texttt{fo2BDDpair}, [\![\_]\!]_{\_}^{\texttt{BDDpair}})$ is defined in Subsection 4.1 except for its abstraction function $\texttt{fo2BDDpair}\colon \mathcal{F} \to \texttt{BDDpair}$. The function maps formulas to BDD pairs (b_{must}, b_{may}). We define $\texttt{fo2BDDpair}(\varphi)$ inductively as follows.

1. Assume that φ is an atomic formula or its negation, i.e., $\varphi \equiv t_1 \bowtie t_2$ for $\bowtie \in \{=, \neq, \leq_u, <_u, \leq_s, <_s\}$. The function fo2BDDpair computes the pair (b_{must}, b_{may}) from $\texttt{te2}\overline{\texttt{BDD?}}(t_1)$ and $\texttt{te2}\overline{\texttt{BDD?}}(t_2)$ using modified algorithms for the corresponding predicates on vectors of standard BDDs. For example, Listing 4 shows an algorithm for *equality* of vectors in $\overline{\texttt{BDD?}}$ (compare to the original function for equality of vectors of standard BDDs presented in Listing 2). 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 4 also shows the algorithms computing *disequality* and *unsigned inequality* of vectors in $\overline{\texttt{BDD?}}$. The algorithms for other predicates are similar.

2. Now assume that 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 $\texttt{fo2BDDpair}(\varphi_1)$ and (b^2_{must}, b^2_{may}) be the result of $\texttt{fo2BDDpair}(\varphi_2)$. Then we define

$$\texttt{fo2BDDpair}(\varphi_1 \wedge \varphi_2) = ((b^1_{must} \,\&\, b^2_{must}),\ (b^1_{may} \,\&\, b^2_{may})),$$
$$\texttt{fo2BDDpair}(\varphi_1 \vee \varphi_2) = ((b^1_{must} \mid b^2_{must}),\ (b^1_{may} \mid b^2_{may})).$$

3. Finally, assume that φ has the form $\forall x\,(\varphi_1)$ or $\exists x\,(\varphi_1)$. Let (b^1_{must}, b^1_{may}) be the result of $\texttt{fo2BDDpair}(\varphi_1)$. Then we define

$$\texttt{fo2BDDpair}(\forall x\,(\varphi_1)) = (bdd_\forall(x, b^1_{must}),\ bdd_\forall(x, b^1_{may})),$$
$$\texttt{fo2BDDpair}(\exists x\,(\varphi_1)) = (bdd_\exists(x, b^1_{must}),\ bdd_\exists(x, b^1_{may})).$$

Listing 4: Functions bvec_eq_trunc, bvec_neq_trunc, and bvec_leq_trunc implementing *equality*, *disequality*, and *unsigned inequality* on vectors $\overline{a} = (a_i)_{0 \leq i < k}$ and $\overline{b} = (b_i)_{0 \leq i < k}$ in $\overline{\mathsf{BDD?}}$.

```
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 ($result_{must}$, $result_{may}$);
}

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

Example 1. *Let t, r, s, u be bit-vector terms of an identical bit-width, for which we have computed only the least-significant bit. Formally,*

$$\texttt{te2}\overline{\texttt{BDD?}}(t) = (?, \ldots, ?, b_t), \qquad \texttt{te2}\overline{\texttt{BDD?}}(r) = (?, \ldots, ?, b_r),$$
$$\texttt{te2}\overline{\texttt{BDD?}}(s) = (?, \ldots, ?, b_s), \qquad \texttt{te2}\overline{\texttt{BDD?}}(u) = (?, \ldots, ?, b_u),$$

where b_t, b_r, b_s, b_u are BDDs.

Consider the formula $t = r$. The function `fo2BDDpair` *applied on this formula returns the pair $(\boxed{0}, b_{may})$, where b_{may} is $b_t \leftrightarrow b_r$. The pair says that an assignment may satisfy the formula $t = r$ only if it satisfies $b_t \leftrightarrow b_r$. Therefore, if $t = r$ is put in conjunction with another formula implying that $b_t \leftrightarrow b_r$ is equal to $\boxed{0}$, the whole conjunction can be decided as unsatisfiable.*

Consider the formula $s \neq u$. The function `fo2BDDpair` *now produces the pair $(b_s \operatorname{xor} b_u, \boxed{1})$. Intuitively, if an assignment satisfies $b_s \operatorname{xor} b_u$, it also satisfies the formula $s \neq u$, regardless the values of the remaining bits of s and u.*

Further, consider the formula $t = r \wedge s \neq u$. The result of `fo2BDDpair` *applied to this formula is computed as $(\boxed{0} \,\&\, (b_s \operatorname{xor} b_u),\ (b_t \leftrightarrow b_r) \,\&\, \boxed{1})$, which can be simplified to $(\boxed{0}, b_t \leftrightarrow b_r)$. This BDD pair contains neither b_s nor b_u.*

Finally, consider the formula $t = r \vee s \neq u$. The result of `fo2BDDpair`$(t = r \vee s \neq u)$ *is computed as $(\boxed{0} \mid (b_s \operatorname{xor} b_u),\ (b_t \leftrightarrow b_r) \mid \boxed{1})$, which is clearly equivalent to $(b_s \operatorname{xor} b_u, \boxed{1})$. This BDD pair contains neither b_t nor b_r.*

Now we prove the completeness of both truncating abstract domains.

Theorem 3. *The truncating term abstract domain and the truncating formula abstract domain are complete.*

Proof. To prove the completeness of the abstract domains, we need to show that for every term $t \in \mathcal{T}$, every formula $\varphi \in \mathcal{F}$, and every assignment μ it holds

$$[\![t]\!]_\mu \in [\![\texttt{te2}\overline{\texttt{BDD?}}(t)]\!]_\mu^{\overline{\texttt{BDD?}}} \quad \text{and} \quad [\![\varphi]\!]_\mu \in [\![\texttt{fo2BDDpair}(\varphi)]\!]_\mu^{\texttt{BDDpair}}.$$

In the proof, we use that fact that the latter condition is equivalent to the following claim where `fo2BDDpair`$(\varphi) = (b_{must}, b_{may})$.

$$[\![b_{must}]\!]_\mu \implies [\![\varphi]\!]_\mu \implies [\![b_{may}]\!]_\mu$$

As formulas contain terms and terms can contain formulas thanks to the `ite`$(\_,\_,\_)$ construct, we prove completeness of both abstract domains together by a structural induction.

The base case are the terms and formulas without any `ite`$(\_,\_,\_)$ subterms. For each term t without any `ite`$(\_,\_,\_)$ subterms, Theorem 2 says that each element of $\texttt{te2}\overline{\texttt{BDD?}}(t)$ is either ? or it is equal to the corresponding element of $\texttt{te2}\overline{\texttt{BDD}}(t)$. Together with the definition of $[\![\_]\!]_\_^{\overline{\texttt{BDD?}}}$, we immediately get that for each assignment μ it holds $[\![\texttt{te2}\overline{\texttt{BDD}}(t)]\!]_\mu \in [\![\texttt{te2}\overline{\texttt{BDD?}}(t)]\!]_\mu^{\overline{\texttt{BDD?}}}$. As $[\![t]\!]_\mu = [\![\texttt{te2}\overline{\texttt{BDD}}(t)]\!]_\mu$, we have $[\![t]\!]_\mu \in [\![\texttt{te2}\overline{\texttt{BDD?}}(t)]\!]_\mu^{\overline{\texttt{BDD?}}}$.

For formulas φ without any `ite`$(\_,\_,\_)$ subterms, we prove the base case by a nested structural induction. The first item below proves the base case of the nested induction (i.e., the claim holds for atomic formulas) and the subsequent items employ induction hypothesis (i.e., the assumption that the claim holds for proper subformulas of the current formula).

1. Assume that φ is an atomic subformula or its negation, i.e., $\varphi \equiv t_1 \bowtie t_2$ for $\bowtie \in \{=, \neq, \leq_u, <_u, \leq_s, <_s\}$. Terms t_1, t_2 were translated by $\texttt{te2}\overline{\texttt{BDD?}}$ to vectors in $\overline{\texttt{BDD?}}$, where each element of these vectors is either `?` or a BDD representing precisely the corresponding bit (see Theorem 2).

   We first consider the case for $\varphi \equiv t_1 = t_2$. The value of `fo2BDDpair`(φ) is computed by function `bvec_eq_trunc` of Listing 4. We compare this function with function `bvec_eq` of Listing 2 which computes the precise result `fo2BDD`(φ). In other words, the BDD *result* produced by `bvec_eq` satisfies $[\![\mathit{result}]\!]_\mu = [\![\varphi]\!]_\mu$. If both of the arguments of function `bvec_eq_trunc` contain no `?` element, then its output clearly satisfies $\mathit{result}_{must} = \mathit{result}_{may} = \mathit{result}$ and we are done. If any argument of function `bvec_eq_trunc` contains some `?`, then $\mathit{result}_{must} = \boxed{0}$ and thus $[\![\mathit{result}_{must}]\!]_\mu \implies [\![\varphi]\!]_\mu$. Further, result_{may} is computed in the same way as *result*, only some "conjuncts" of the form $a_i \leftrightarrow b_i$ are skipped. Therefore, $[\![\varphi]\!]_\mu = [\![\mathit{result}]\!]_\mu \implies [\![\mathit{result}_{may}]\!]_\mu$. The situation is dual for $\varphi \equiv t_1 \neq t_2$ and `bvec_neq_trunc`.

   For $\varphi \equiv t_1 \leq_u t_2$, we compare the functions `bvec_leq` of Listing 2 and `bvec_leq_trunc` of Listing 4. Consider variables result_{must} and result_{may} from `bvec_leq_trunc`. We show that

   $$[\![\mathit{result}_{must}]\!]_\mu \implies [\![\mathit{result}]\!]_\mu \implies [\![\mathit{result}_{may}]\!]_\mu \qquad (*)$$

   for each assignment μ is an invariant of the for loop, where *result* refers to the value of the variable from the corresponding iteration of `bvec_leq`. The claim clearly holds before the first iteration of the loop, as all three of the BDDs are $\boxed{1}$. We now show that the invariant is preserved by each loop iteration. Assume that (*) holds for each assignment μ before an iteration. Let us denote as $\mathit{result}' = (!a_i \;\&\; b_i) \mid (\mathit{result} \;\&\; (a_i \leftrightarrow b_i))$ the value of *result* after an iteration of `bvec_leq`. We consider all three cases from the pseudocode of `bvec_leq_trunc`. If $a_i \neq\ ?$ and $b_i \neq\ ?$, the BDDs $(!a_i \;\&\; b_i)$ and $(a_i \leftrightarrow b_i)$ in `bvec_leq_trunc` are the same as in `bvec_leq`. Therefore, both

   $$[\![(!a_i \;\&\; b_i) \mid (\mathit{result}_{must} \;\&\; (a_i \leftrightarrow b_i))]\!]_\mu \implies [\![\mathit{result}']\!]_\mu \quad \text{and}$$
   $$[\![\mathit{result}']\!]_\mu \implies [\![(!a_i \;\&\; b_i) \mid (\mathit{result}_{may} \;\&\; (a_i \leftrightarrow b_i))]\!]_\mu$$

   must hold for each assignment μ due to monotonicity of & and |. For the second case, if $a_i = \boxed{1}$ or $b_i = \boxed{0}$, then $(!a_i \;\&\; b_i)$ computed by `bvec_leq` is $\boxed{0}$. Therefore result' is equal to $\mathit{result} \;\&\; (a_i \leftrightarrow b_i)$. Together with the assumption (*) this gives $[\![\mathit{result}']\!]_\mu \implies [\![\mathit{result}]\!]_\mu \implies [\![\mathit{result}_{may}]\!]_\mu$ for each assignment μ. As a result, $[\![\boxed{0}]\!]_\mu \implies [\![\mathit{result}']\!]_\mu \implies [\![\mathit{result}_{may}]\!]_\mu$ holds for each assignment μ. In the final case, $[\![\boxed{0}]\!]_\mu \implies [\![\mathit{result}']\!]_\mu \implies [\![\boxed{1}]\!]_\mu$ clearly holds.

The argumentation for the cases $t_1 <_u t_2$, $t_1 \leq_s t_2$, and $t_1 <_s t_2$ is analogous.

2. Now assume that φ has the form $\varphi_1 \wedge \varphi_2$. The definition of `fo2BDDpair` says that

$$\texttt{fo2BDDpair}(\varphi_1 \wedge \varphi_2) = ((b^1_{must} \mathbin{\&} b^2_{must}),\ (b^1_{may} \mathbin{\&} b^2_{may})),$$

where (b^1_{must}, b^1_{may}) is the result of $\texttt{fo2BDDpair}(\varphi_1)$ and (b^2_{must}, b^2_{may}) is the result of $\texttt{fo2BDDpair}(\varphi_2)$. Let μ be an arbitrary assignment. We know from the induction hypothesis that $[\![b^1_{must}]\!]_\mu \implies [\![\varphi_1]\!]_\mu$ and $[\![b^2_{must}]\!]_\mu \implies [\![\varphi_2]\!]_\mu$. As the BDD operation & corresponds to conjunction, we have $[\![b^1_{must} \mathbin{\&} b^2_{must}]\!]_\mu = [\![b^1_{must}]\!]_\mu \wedge [\![b^2_{must}]\!]_\mu$ and thus $[\![b^1_{must} \mathbin{\&} b^2_{must}]\!]_\mu \implies [\![\varphi_1]\!]_\mu$ and $[\![b^1_{must} \mathbin{\&} b^2_{must}]\!]_\mu \implies [\![\varphi_2]\!]_\mu$. This immediately implies

$$[\![b^1_{must} \mathbin{\&} b^2_{must}]\!]_\mu \implies [\![\varphi_1]\!]_\mu \wedge [\![\varphi_2]\!]_\mu = [\![\varphi_1 \wedge \varphi_2]\!]_\mu.$$

The induction hypothesis also gives us $[\![\varphi_1]\!]_\mu \implies [\![b^1_{may}]\!]_\mu$ and $[\![\varphi_2]\!]_\mu \implies [\![b^2_{may}]\!]_\mu$. As $[\![\varphi_1 \wedge \varphi_2]\!]_\mu = [\![\varphi_1]\!]_\mu \wedge [\![\varphi_2]\!]_\mu$, we have $[\![\varphi_1 \wedge \varphi_2]\!]_\mu \implies [\![b^1_{may}]\!]_\mu$ and $[\![\varphi_1 \wedge \varphi_2]\!]_\mu \implies [\![b^2_{may}]\!]_\mu$. As & implements conjunction, we can conclude that

$$[\![\varphi_1 \wedge \varphi_2]\!]_\mu \implies [\![b^1_{may}]\!]_\mu \wedge [\![b^2_{may}]\!]_\mu = [\![b^1_{may} \mathbin{\&} b^2_{may}]\!]_\mu.$$

The case when the formula φ has the form $\varphi_1 \vee \varphi_2$ is analogous.

3. Assume that φ has the form $\forall x\,(\varphi_1)$. The definition of `fo2BDDpair` says that

$$\texttt{fo2BDDpair}(\forall x\,(\varphi_1)) = (\mathit{bdd}_\forall(x, b^1_{must}),\ \mathit{bdd}_\forall(x, b^1_{may})),$$

where (b^1_{must}, b^1_{may}) is the result of $\texttt{fo2BDDpair}(\varphi_1)$. Let μ be an arbitrary assignment. We want to show that $[\![\mathit{bdd}_\forall(x, b^1_{must})]\!]_\mu \implies [\![\forall x\,(\varphi_1)]\!]_\mu$ and $[\![\forall x\,(\varphi_1)]\!]_\mu \implies [\![\mathit{bdd}_\forall(x, b^1_{may})]\!]_\mu$.

For the first implication, assume that $[\![\mathit{bdd}_\forall(x, b^1_{must})]\!]_\mu = 1$ and v is an arbitrary bit-vector of the same bit-width as x. We want to show that $[\![\varphi_1]\!]_{\mu[x \mapsto v]} = 1$. The assumption and the definition of operation $\mathit{bdd}_\forall$ imply that $[\![b^1_{must}]\!]_{\mu[x\mapsto v]} = 1$. The induction hypothesis says that $[\![b^1_{must}]\!]_{\mu[x\mapsto v]} \implies [\![\varphi_1]\!]_{\mu[x\mapsto v]}$ and thus $[\![\varphi_1]\!]_{\mu[x\mapsto v]} = 1$.

For the second implication, assume that $[\![\forall x\,(\varphi_1)]\!]_\mu = 1$. We want to prove that $[\![\mathit{bdd}_\forall(x, b^1_{may})]\!]_\mu = 1$. In other words, we need to show that $[\![b^1_{may}]\!]_{\mu[x\mapsto v]} = 1$ for an arbitrary bit-vector v of the same bit-width as x. The assumption implies that $[\![\varphi_1]\!]_{\mu[x\mapsto v]} = 1$ and the induction hypothesis says that $[\![\varphi_1]\!]_{\mu[x\mapsto v]} \implies [\![b^1_{may}]\!]_{\mu[x\mapsto v]}$. As a direct consequence, we get $[\![b^1_{may}]\!]_{\mu[x\mapsto v]} = 1$.

The case when the formula φ has the form $\exists x\,(\varphi_1)$ is again analogous.

It remains to prove the induction step of the outer induction. Let t be an arbitrary term and φ be an arbitrary formula. Due to the induction hypothesis, we can assume that for each their subterm $\mathtt{ite}(\varphi', t_1, t_2)$ and for each assigment μ it holds $[\![\varphi']\!]_\mu \in [\![\mathtt{fo2BDDpair}(\varphi')]\!]_\mu^{\mathsf{BDDpair}}$. This matches the assumption of Theorem 2 and thus we can prove $[\![t]\!]_\mu \in [\![\mathtt{te2}\overline{\mathtt{BDD?}}(t)]\!]_\mu^{\overline{\mathsf{BDD?}}}$ and $[\![\varphi]\!]_\mu \in [\![\mathtt{fo2BDDpair}(\varphi)]\!]_\mu^{\mathsf{BDDpair}}$ the same arguments as in the base case. $\square$

Since the truncating formula abstract domain is complete, we can use Theorem 1 to check satisfiability of a given formula φ. Assume that $\mathtt{fo2BDDpair}(\varphi) = (b_{must}, b_{may})$. If b_{must} is not $\boxed{0}$, then there exists an assignment μ such that $[\![b_{must}]\!]_\mu = 1$ and thus the formula φ is satisfiable. Furthermore, if b_{may} is $\boxed{0}$, then $[\![b_{may}]\!]_\mu = 0$ for each assignment μ and thus φ is unsatisfiable.

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

5. Implementation and Deployment

We have implemented the described truncating formula abstract domain into the SMT solver Q3B [13], which is written in C++. The solver Q3B uses the package CUDD [20] for BDD representation and operations, and the implementation of bit-vectors and bit-vector operations for CUDD by P. Navrátil [17]. We have modified this implementation to support ? elements in input of all operations over vectors of BDDs. Further, we have modified all algorithms for relation operators, logical operators, and quantifier processing to work with BDD pairs (b_{must}, b_{may}) rather than with individual BDDs. 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 without operation abstractions often has difficulties to handle them. The other bit-vector operations do not introduce new ? elements, but they can produce them on inputs containing some ? elements. The implementation supports all the bit-vector operations defined by SMT-LIB standard, is open-source and available on GitHub[3].

The implementation also contains several improvements, which use the returned BDD b_{may} even in the cases where the abstraction cannot be used to decide the satisfiability of the input formula. First of these is checking the potential models described by b_{may} as explained in Subsection 5.1. Further, Subsection 5.2 shows that b_{may} can be also used to identify some bits whose values are implied by the input formula.

Furthermore, Subsection 5.3 introduces formula modifications that add new variables for results of multiplications and divisions and their respective congruences to further amplify the positive effect of the truncating abstractions.

[3]`https://github.com/martinjonas/Q3B`

Finally, Subsection 5.4 shows how to combine the proposed operation abstractions with the approximations of bit-widths of variables [10], which were already implemented in Q3B.

5.1. Checking Possible Models

The satisfiability check based on truncating abstraction, which is explained at the end of Section 4, returns a definite answer only if b_{must} is not $\boxed{0}$ or if b_{may} is $\boxed{0}$. In the former case the formula is decided as satisfiable; in the latter case it is decided as unsatisfiable. Even if neither of the two conditions holds, b_{may} can sometimes be used to decide the satisfiability of the formula: one can extract a model from b_{may} and check whether it satisfies φ. If it does, the input formula is satisfiable; if it does not, the result is unknown.

This checking of potential models is supported in our implementation, but it is more involved. The problem is that the potential model contains values only for free variables and top-level existential variables, and substituting the model values for those variables in φ results in a (closed) formula φ_{subst} that can still contain nested universal and existential quantifiers. Such a formula cannot be directly evaluated to yield 1 or 0 and has to be decided as another query to a solver for quantified formulas. However, satisfiability check of φ_{subst} can be potentially expensive and we want to ensure that the model checking finishes quickly. Therefore, in our implementation, we compute only b'_{must} for φ_{subst} using a very low limit on the number of BDD nodes. Note that this computation will usually finish in a short time. If the resulting b'_{must} is not $\boxed{0}$, then φ_{subst} is satisfiable, and thus also the input formula is satisfiable. In the opposite case, b'_{must} is $\boxed{0}$ and the potential model is discarded, because it is either not a model of the input formula or could not be validated quickly.

Example 2. *Consider the following formula φ in which the variable x is free and all the variables have bit-width 32:*

$$x >_u 1 \quad \wedge \quad \forall y \exists z (y = x \cdot z).$$

The subformula $y = x \cdot z$ implies that the least-significant bits of the variables must satisfy $y_0 \leftrightarrow (x_0 \wedge z_0)$. If we translate φ to (b_{must}, b_{may}) such that only the least-significant bit of $x \cdot z$ is computed precisely, then $b_{must} = \boxed{0}$ and b_{may} corresponds to the formula $(x >_u 1) \wedge (\forall y_0 \exists z_0 (y_0 \leftrightarrow (x_0 \wedge z_0)))$, which is equivalent to the formula $(x >_u 1) \wedge x_0$. In other words, b_{may} says that every x satisfying the formula must represent an odd number greater than one.

We cannot decide satisfiability of φ from this pair of BDDs. However, we can take a model of b_{may}, say $x = 3$, and substitute it to φ. We obtain the formula $\varphi_{subst} \equiv \forall y \exists z (y = 3 \cdot z)$. The BDD $b'_{must} = \boxed{1}$ corresponding to the formula φ_{subst} can be computed even for a small node limit. This implies that φ_{subst} is satisfiable, and thus φ is also satisfiable and $x = 3$ is its model.

A dual approach can be used for closed formulas that contain universally quantified variables on the top-level of the formula. If a BDD b_{must} for such formula is $\boxed{0}$, one can identify a potential countermodel, i.e., an assignment of values to the top-level universal variables that makes the formula unsatisfiable. Such a potential countermodel can then be checked against the original formula.

Since no countermodel can be computed directly from the result $b_{must} = \boxed{0}$, in the implementation we have an option to negate an input formula if it is closed and has a universal top-level quantifier. This reduces the problem to the one described in the beginning of this subsection: if the negation of the closed input formula is satisfiable, the original is unsatisfiable and vice versa. Note that although this option is supported by our solver, it is not enabled by default and it is not used in the experiments.

5.2. Learning From Overapproximations

The implementation uses b_{may} not only for generating potential models, but also for identifying values of some bits that are necessary to satisfy the formula. Since b_{may} represents a superset of all the satisfying assignments, if the value of some bit b is 1 in all the represented assignments, this bit must be 1 if the input formula should be satisfied. This also works analogously if the value of a bit is 0 in all the represented assignments. After identifying all such bits, we replace them by their implied values, which preserves satisfiability of the formula, and we try solving the resulting formula with an increased node limit. In the implementation, these bits and their implied values are identified using the CUDD function `FindEssential()`.

For example, consider the formula $x \leq_u 30 \wedge x \cdot y = 0$ where x, y are 32-bit variables. The precise BDD for the subformula $x \cdot y = 0$ is exponentially large. But because the BDD for the subformula $x \leq_u 30$ can be computed precisely, the aforementioned procedure can identify that the most-significant 27 bits of x must be zero from the result of b_{may}. Therefore, the formula can be modified to $(00\dots0x_4x_3x_2x_1x_0 \leq_u 30) \wedge (00\dots0x_4x_3x_2x_1x_0 \cdot y = 0)$. For this formula, the computation of the precise BDD is feasible.

Now consider the formula $x \leq_s 4 \ \wedge \ x \geq_s -4 \ \wedge \ x \cdot y = 0$. Although the most-significant 29 bits of x must be either all 0 (if x is non-negative) or all 1 (if x is negative), the aforementioned procedure cannot identify this, as these bits have more possible values. Therefore, we have also implemented a procedure that identifies which successive bits of a variable must be equal for the formula to be true. In the example, we can thus identify that the most-significant 29 bits of x must be all the same and therefore we can represent all these bits by a single Boolean variable in the following iterations and start the solving again. The pseudocode for the procedure `get_implied_eqns` that identifies equalities implied by b_{may} is presented in Listing 5. The idea of this procedure is straightforward: if a conjunction of b_{may} with a xor b is $\boxed{0}$, there is no satisfying assignment of the formula with different values of a and b. And thus the values of a and b must have the same value if the formula should be satisfied.

5.3. Adding New Variables and Congruences

The abstractions by themselves cannot directly 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

Listing 5: Function `get_implied_eqns` computes equivalences of the successive bits of variables that are implied by the input formula. Function `vars(`φ`)` returns the set of all bit-vector variables in φ and `bitWidth(`v`)` returns the bit-width of variable v.

```
get_implied_eqns($\varphi$, $b_{may}$)
{
  $impliedEqualities \leftarrow \emptyset$;
  foreach $v$ in vars($\varphi$) {
    foreach $i$ from 0 to bitWidth($v$) - 2 {
      if (($b_{may}$ & ($v_i$ xor $v_{i+1}$)) == $\boxed{0}$) {
        $impliedEqualities \leftarrow impliedEqualities \cup \{(v_i, v_{i+1})\}$;
      }
    }
  }
  return $impliedEqualities$;
}
```

information that the corresponding ? elements in the two vectors representing the two occurrences of $x \cdot y$ have been 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

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

This formula is decided as unsatisfiable even if $x \cdot y$ is computed with arbitrarily low precision. Let us note that this particular case could be solved by the original Q3B without operation abstractions by computing precise BDD only for the subformula $m_{x,y} \leq_u 2 \wedge m_{x,y} \geq_u 4$ (and not for the third conjunct) as this conjunction is already unsatisfiable.

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

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

which cannot be decided unsatisfiable even by using the abstractions, because without computing the multiplication precisely, the solver cannot infer the relationship between variables $m_{x,y}$ and $m_{z,y}$. To solve such formula, our implementation adds a congruence subformula stating that $(x = z) \rightarrow (m_{x,y} = m_{z,y})$ to the formula. This results in the formula

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

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 cannot be shown by the original solver without the abstractions, because it is infeasible to compute the precise BDD for the inner quantified subformula.

We now precisely describe the implemented formula modifications. For simplicity, suppose that the input formula is closed. The modifications proceed in two steps:

1. In the first step, we introduce constants $m_{x,y}$ for each subterm $x{\cdot}y$, where x and y are bit-vector variables. Namely, we recursively traverse the formula and identify all subformulas of the form $Q_1 x\,(\psi)$, where ψ has a subformula of the form $Q_2 y\,(\rho)$ and ρ contains a subterm $x \cdot y$ for $Q_1, Q_2 \in \{\exists, \forall\}$ and bit-vector variables x, y. We replace all such subformulas ρ by

$$\rho' \quad \equiv \quad \exists m_{x,y}\,(\rho[x \cdot y \mapsto m_{x,y}] \;\wedge\; m_{x,y} = x \cdot y),$$

   where $m_{x,y}$ is a fresh variable.

   Note that a general approach would be to introduce new variables for multiplications of *arbitrary* terms; not just variables. However, our preliminary experiments have shown that adding new variables only for multiplications of variables is often sufficient for real-world formulas.

2. In the resulting formula, we add the subformulas expressing the congruences for the variables $m_{x,y}$. We iterate through all the modified subformulas $\rho' \equiv \exists m_{x,y}\,(\rho[x{\cdot}y \mapsto m_{x,y}] \wedge m_{x,y} = x{\cdot}y)$ such that the occurrence of the subformula ρ' is in scope of some newly introduced variable $m_{z,v}$ such that x, y, z, v have the same bit-width. Based on the *syntactic* equality of the variables x, y, z, and v, we then perform one of the following modifications:

   - If $x \neq z$ and $y = v$, the body of the quantified subformula ρ' is conjoined with the formula $(x = z) \to (m_{x,y} = m_{z,y})$.
   - If $x = z$ and $y \neq v$, the body of the quantified subformula ρ' is conjoined with the formula $(y = v) \to (m_{x,y} = m_{x,v})$.
   - If $x \neq z$ and $y \neq v$, the body of the quantified subformula ρ' is conjoined with the formula $(x = z \wedge y = v) \to (m_{x,y} = m_{z,v})$.

The analogous procedure is also implemented for terms $x \div y$ with *division*. Similar transformation is not implemented for *addition* as it was not helpful according to our preliminary experiments. Note that these formula modifications increase both the number of the variables in the formula and its size, so the resulting BDD can in the worst case be exponentially larger than the BDD for the original formula. However, we did not observe this kind of behavior in practice and the overhead was outweighed by the benefits.

5.4. Combining Operation Abstractions and Formula Approximations

As mentioned in Introduction, the solver Q3B employs formula approximations, which can in some cases help with solving harder input formulas including formulas with multiplication. These formula approximations are based on fixing bits of the formula variables and are orthogonal to the newly introduced operation abstractions. On the one hand, there are formulas, such as the three formulas from Introduction, where fixing bits of the variables cannot help with deciding satisfiability of the formula and operation abstractions have to be employed. On the other hand, consider the formula $x \cdot y = z$ from Introduction. As has been explained, this formula can be decided as satisfiable by considering the underapproximation in which most of the bits of all three variables are assigned to constant 0. The introduced operation abstractions alone, however, cannot decide satisfiability of this formula. This happens because unless all bits of $x \cdot y$ are computed precisely, the resulting BDD pair satisfies $b_{must} = \boxed{0}$ and $b_{may} \neq \boxed{0}$. Generally speaking, formula approximations tend to be useful when satisfiability of the formula depends on a small subsets of domains of the variables. On the other hand, operation abstractions tend to be useful when satisfiability of the formula depends on a small number of bits from the results of the hard operations.

Since both of these cases happen in practice, and often both happen in a single formula, this subsection elaborates on the interaction of the previously-implemented approximations with the newly implemented operation abstractions.

Recall that approximations of formulas are of two kinds: *underapproximation* and *overapproximation*. An underapproximation is basically a formula that logically entails the input formula; therefore if an underapproximation is satisfiable, the original formula is also satisfiable. Analogously, an overapproximation is a formula that is logically entailed by the input formula; if an overapproximation is unsatisfiable, the original formula is also unsatisfiable. In Q3B, the formula approximations are performed on formulas in negation normal form by reducing the *effective bit-width* of selected bit-vector variables by fixing some of their bits to chosen values. The underapproximations are obtained by decreasing effective bit-widths of all free and existentially quantified variables to a given value and the overapproximations are obtained by decreasing effective bit-widths of universally quantified variables to a given value.

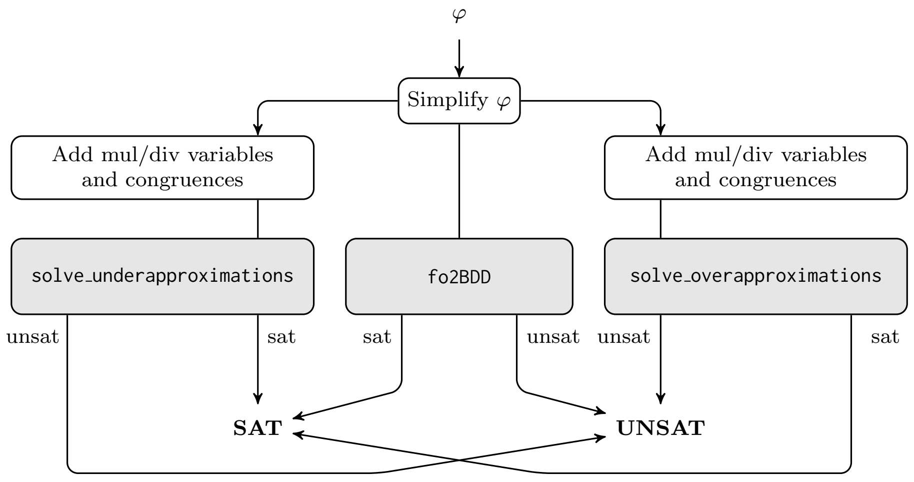
The combination of formula approximations and operation abstractions relies on the following simple theorem.

Theorem 4. *Let φ be an arbitrary formula. Let $\underline{\varphi}$ be an underapproximation of φ, i.e., $\underline{\varphi} \models \varphi$, and (b_{must}, b_{may}) be the abstract object corresponding to $\underline{\varphi}$. Then if $b_{must} \neq \boxed{0}$, the formula φ is satisfiable.*

Analogously, let $\overline{\varphi}$ be an overapproximation of φ, i.e., $\varphi \models \overline{\varphi}$, and (b_{must}, b_{may}) be the abstract object corresponding to $\overline{\varphi}$. Then if $b_{may} = \boxed{0}$, the formula φ is not satisfiable.

Proof. Directly follows from definitions of underapproximations, overapproximations, and from the proof of Theorem 3. □

Q3B first simplifies the input formula using simplification rules designed for quantified formulas [11] and other general simplification rules. Then it starts



φ
Simplify φ
Add mul/div variables and congruences
Add mul/div variables and congruences
solve_underapproximations
fo2BDD
solve_overapproximations
unsat
sat
sat
unsat
unsat
sat
SAT
UNSAT

Figure 5: High-level overview of SMT solving approach used by Q3B. The three shaded areas are executed in parallel and the first result is returned.

solving the simplified formula, underapproximations of this formula, and overapproximations of this formula in parallel. The first result of these three threads is then returned to the user. The overall solving approach of Q3B is depicted on Figure 5. 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 6 shows the simplified implementations of solving underapproximations and overapproximations. The algorithm `solve_underapproximations` starts with the small initial values of the effective bit-width `effBW` for free and 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 before another try:

- if operation abstractions caused an imprecision, the node limit is increased;
- if the BDD pair returned by `fo2BDDpair` was 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 $2, 4, 6, 8, \ldots$. Both the initial node limit and the limit for model checking is 1000 and the function `increaseNodeLimit()` multiplies the node limit by 4 each time. The implementation `solve_overapproximations` for solving overapproximations is analogous. Note that the implementation that increases effective bit-with first and the node limit second is also possible. However, the implementation proposed above produced best results in our preliminary experiments.

Listing 6: Functions solve_underapproximation and solve_overapproximation combining operation abstractions with approximations.

```
solve_underapproximations($\varphi$)
{
  effBW $\leftarrow$ initialEffBW;
  nodeLimit $\leftarrow$ initialNodeLimit;
  while (true) {
    $\varphi_u \leftarrow$ underApprox($\varphi$, effBW);
    ($b_{must}$, $b_{may}$) $\leftarrow$ fo2BDDpair($\varphi_u$, nodeLimit);
    if ($b_{must}$ != $\boxed{0}$) return SAT;
    if ($b_{may}$ == $\boxed{0}$ and $\varphi$ == $\varphi_u$) return UNSAT;
    if ($b_{must}$ != $b_{may}$) {
      nodeLimit $\leftarrow$ increaseNodeLimit(nodeLimit);
    }
    if ($b_{must}$ == $b_{may}$ and $\varphi$ != $\varphi_u$) {
      effBW $\leftarrow$ increaseEffBW(effBW);
    }
  }
}

solve_overapproximations($\varphi$)
{
  effBW $\leftarrow$ initialEffBW;
  nodeLimit $\leftarrow$ initialNodeLimit;
  while (true) {
    $\varphi_o \leftarrow$ overApprox($\varphi$, effBW);
    ($b_{must}$, $b_{may}$) $\leftarrow$ fo2BDDpair($\varphi_o$, nodeLimit);
    if ($b_{may}$ == $\boxed{0}$) return UNSAT;
    if ($b_{must}$ != $\boxed{0}$ and $\varphi$ == $\varphi_o$) return SAT;

    candidateModel $\leftarrow$ getSatisfyingAssignment($b_{may}$);
    $\varphi_{subst}$ $\leftarrow$ substitute($\varphi$, candidateModel);
    ($b'_{must}$, $b'_{may}$) $\leftarrow$ fo2BDDpair($\varphi_{subst}$, modelCheckNodeLimit);
    if ($b'_{must}$ != $\boxed{0}$) return SAT;

    impliedBitsAndEqns $\leftarrow$ getImpliedBitsAndEquations($b_{may}$);
    $\varphi$ $\leftarrow$ substitute($\varphi$, impliedBitsAndEqns);

    if ($b_{must}$ != $b_{may}$) {
      nodeLimit $\leftarrow$ increaseNodeLimit(nodeLimit);
    }
    if ($b_{must}$ == $b_{may}$ and $\varphi$ != $\varphi_o$) {
      effBW $\leftarrow$ increaseEffBW(effBW);
    }
  }
}
```

In addition, the function `solve_overapproximations` uses the improvements mentioned in Subsections 5.1 and 5.2 in each iteration before increasing the effective bit-width or the node limit. These techniques are not implemented in the `solve_underpproximations` function, because they apply only to overapproximated results. Before calling the functions `solve_underapproximations` and `solve_overapproximations`, we also first transform the formula by adding variables and congruences for the results of multiplications and divisions, as described in Subsection 5.3. These are not used for the precise solver, because for a solver without abstractions they help in only a small number of cases and generally increase the solving time.

6. Experimental Evaluation

This section describes the setup of our experiments and their results.

6.1. Setup

We have evaluated efficiency of the implemented operation abstractions in version 1.0 of the SMT solver Q3B[4] [13]. We have tested 4 different configurations of Q3B:

Q3B is the configuration that uses only the approximations of variable bit-widths,

Q3B+OA is the configuration that uses approximations of variable bit-widths and the described abstractions of bit-vector operations,

Q3B+Cong is the configuration that uses the approximations of variable bit-widths and introduces variables $m_{x,y}$ and their respective congruence subformulas as described in Subsection 5.3,

Q3B+OA+Cong is the configuration that uses approximations of variable bit-widths, abstractions of bit-vector operations, and introduces variables $m_{x,y}$ and their respective congruence subformulas.

Moreover, all four of these configurations check candidate models (Subsection 5.1) and learn from overapproximations (Subsection 5.2). The two configurations Q3B+OA and Q3B+OA+Cong use the algorithm from Subsection 5.4 to combine variable approximations with operation abstractions. The other two configurations use only variable approximations and only increase the effective bit-width during the refinement.

We have further compared these four configurations of Q3B against the SMT solvers Z3 [8], Boolector [19], and CVC4 [1]. We used Z3 in the version 4.8.9, Boolector in the version 3.2.1, and CVC4 in the version 1.8. All the solvers were executed in their default configurations. We evaluated all 7 solvers on all

[4]Commit `12ded255950a8fa8d81c7656a035b4df070c05e3`.

5846 quantified bit-vector formulas from the SMT-LIB repository [2] used in SMT-COMP 2020. The used benchmarks are divided into eight benchmark sets: two sets of benchmarks from the tool Ultimate Automizer by M. Heizmann (marked as *heizmann-ua-17* and *heizmann-ua-19*), unsatisfiable benchmarks that were created by M. Preiner et al. to verify bit-vector invertibility conditions [18] used by the SMT solver CVC4 (marked as *preiner-cav18*), benchmarks that were created by converting integer and real arithmetic benchmarks to bit-vectors by M. Preiner (marked as *preiner-keymaera*, *preiner-psyco*, *preiner-scholl-smt08*, *preiner-tptp*, and *preiner-ua*), benchmarks corresponding to verification conditions for software optimizations (*llvm13-smtlib*), and benchmarks from software and hardware verification by C. M. Wintersteiger (marked as *wintersteiger*). The total numbers of benchmarks in these benchmark families are shown in the column *Total* in Table 1.

Note that the described operation abstractions are not specific to *quantified* formulas, but also work for *quantifier-free* formulas. However, our preliminary experiments show that although the abstractions slightly improve the performance also on quantifier-free formulas, our BDD-based SMT solver is not competitive with state-of-the-art SMT solvers for quantifier-free formulas based on bit-blasting and efficient SAT solvers. The benefit of using BDDs comes with quantifiers, which are much easier for BDD-based SMT solvers than for traditional SMT solvers based on quantifier instantiation. We therefore focus solely on formulas with quantifiers.

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 60 minutes of CPU time. All measured times are CPU times as wall times would give a indisputable advantage to solvers that use multiple parallel threads, i.e., Boolector and Q3B. For reliable benchmarking we employed BenchExec [4], a tool that allocates resources for a program execution and measures their use precisely.

6.2. Results

Table 1 shows the numbers of solved benchmarks by the individual solvers. In total, Q3B+OA was able to solve 39 more benchmarks than the original version of Q3B. The configuration Q3B+OA+Cong, which also introduces new variables and congruences for the results of multiplications and divisions as described in Subsection 5.3, was able to solve 115 additional benchmarks (all from *preiner-keymaera*) compared to Q3B+OA. Moreover, the configuration Q3B+OA+Cong solved more benchmarks than other SMT solvers: 172 more than Boolector, 228 more than CVC4, and 177 more than Z3. The column Q3B+Cong of Table 1 also shows that the formula transformations that add new variables for results of multiplications and divisions and the respective congruences by themselves help only to solve 3 additional formulas from *heizmann-ua-17* compared to the configuration Q3B. In other words, these transformations are effective only in combination with the introduced operation abstractions.

From the opposite point of view, Figure 6 shows the number of benchmarks *unsolved* by the individual solvers. This graph shows that most of the benefit of

Table 1: Numbers of benchmarks solved by the individual solvers divided by the satisfiability/unsatisfiability and the benchmark family. The column *Total* also shows the total number of benchmarks for the given category.

Result	Family	Total	Boolector	CVC4	Z3	Q3B	Q3B+OA	Q3B+Cong	Q3B+OA+Cong
UNSAT	heizmann-ua-17	110	11	**110**	17	104	107	107	107
	heizmann-ua-19	73	13	55	35	**72**	**72**	**72**	**72**
	llvm13-smtlib	6	1	**6**	5	1	1	1	1
	preiner-cav18	591	557	**568**	555	566	565	566	565
	preiner-keymaera	3924	3917	3921	**3923**	3786	3792	3786	3907
	preiner-psyco	62	**62**	60	**62**	56	59	56	59
	preiner-scholl-smt08	76	67	45	68	59	**70**	59	**70**
	preiner-tptp	56	53	**56**	**56**	**56**	**56**	**56**	**56**
	preiner-ua	137	**137**	**137**	**137**	**137**	**137**	**137**	**137**
	wintersteiger	95	85	89	91	**94**	93	**94**	93
	Total UNSAT	5130	4903	5047	4949	4931	4952	4934	**5067**
SAT	heizmann-ua-17	21	17	19	14	18	**20**	18	**20**
	heizmann-ua-19	15	**15**	14	**15**	**15**	**15**	**15**	**15**
	llvm13-smtlib	3	2	**3**	**3**	0	0	0	0
	preiner-keymaera	108	106	34	**108**	104	104	104	104
	preiner-psyco	132	131	131	**132**	125	125	125	125
	preiner-scholl-smt08	259	**249**	110	204	233	**249**	233	**249**
	preiner-tptp	17	**17**	**17**	**17**	**17**	**17**	**17**	**17**
	preiner-ua	16	**16**	14	**16**	**16**	**16**	**16**	**16**
	wintersteiger	86	67	78	60	**82**	**82**	**82**	**82**
	Total SAT	657	620	420	569	610	**628**	610	**628**
	Total	5846	5523	5467	5518	5541	5580	5544	**5695**

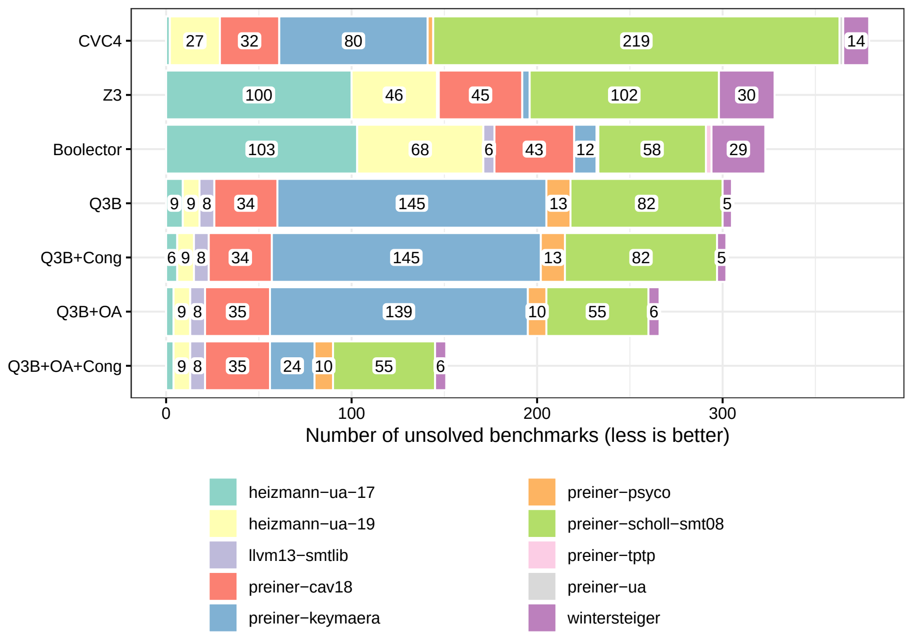


Figure 6: The number of benchmarks unsolved by the individual solvers. The benchmarks are divided by the source of the benchmark. For better readability, the numbers of unsolved benchmarks less than 5 are not explicitly spelled out in the plot.

abstractions is on formulas from the *preiner-keymaera* and *preiner-scholl-smt08* benchmark sets, 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 are 4 formulas solved by Q3B and not Q3B+OA. The additional cost of abstractions is observable also on some benchmarks that were decided both with and without abstractions: computing abstractions slowed Q3B down by more than 0.5 second on 127 benchmarks. On the other hand, there were 90 benchmarks on which the version with abstractions was faster by more than 0.5 second.

The addition of congruences also entails some computational cost, since the modified formulas have more bit-vector variables and are larger overall. Although there were no formulas that were solved by Q3B+OA and not solved by Q3B+OA+Cong, with respect of times there are 62 formulas that were solved by Q3B+OA by at least 0.5 second faster than by Q3B+OA+Cong. On the other hand, the configuration Q3B+OA+Cong was able to solve 96 formulas by at least 0.5 second faster than Q3B+OA.

The running times of Q3B configurations are compared in three scatter plots. Figure 7 compares the base Q3B against Q3B+OA+Cong. We also compare

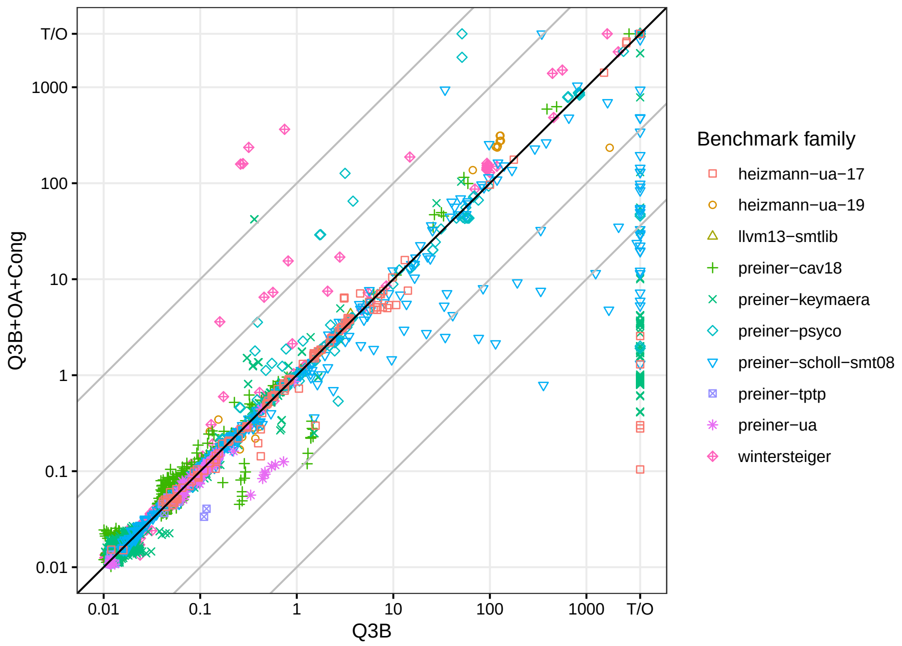


Figure 7: Scatter plot of CPU times of the configuration Q3B and the best-performing configuration Q3B+OA+Cong. Each point represents one benchmark and its color and shape indicate the benchmark family.

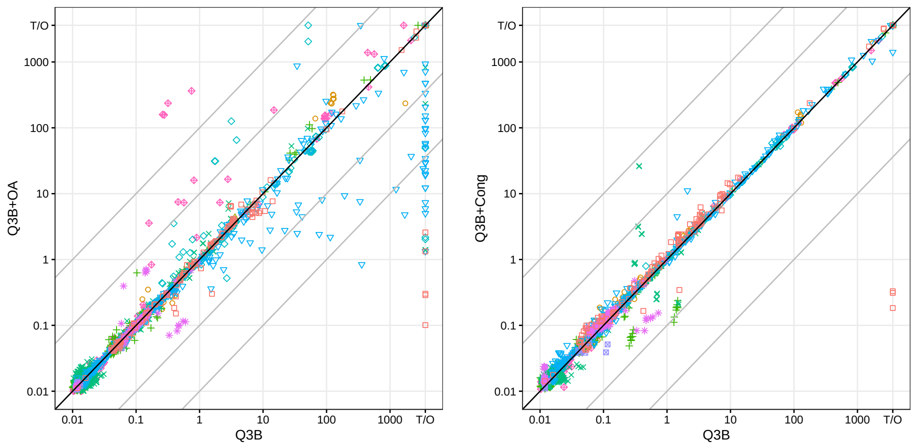


Figure 8: Scatter plots of CPU times of the configuration Q3B against the configurations Q3B+OA (left) and Q3B+Cong (left). Each point represents one benchmark and its color and shape indicate the benchmark family; the colors and shapes are the same as in Figure 7.

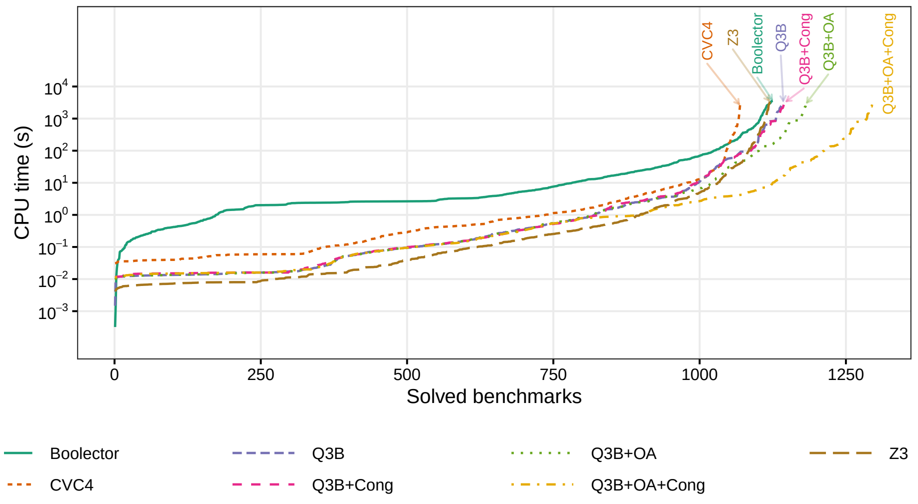


Figure 9: Cactus plot of all solved non-trivial benchmarks from the SMT-LIB repository. Trivial benchmarks are those that all solvers solved within 2 s. The plot shows the number of non-trivial benchmarks (x-axis) that each solver was able to decide within a given CPU time limit (y-axis).

Q3B against the configurations where only operation abstractions are enabled or only congruences are added. Namely, Figure 8 shows the effect of enabling only operation abstractions without congruences (left) and the effect of enabling only congruences without operation abstractions (right).

To compare the solving times of all solvers, Figure 9 shows a cactus plot of solving times of *non-trivial benchmarks* for the individual solvers. We have removed 4398 trivial benchmarks, i.e., the benchmarks that were decided by all of the solvers in less than 2 s.

We investigated how the operation abstractions affect the numbers of iterations of the abstraction refinement loop required to decide the given formula. The numbers of iterations of the refinement loop with and without operation abstractions are compared in Figure 10. The plot shows that enabling operation abstractions often leads to the increase of the number of iterations. This is not surprising, as both the abstraction and its refinements are more fine-grained. However, each iteration of the algorithm with operation abstractions is potentially cheaper than the iteration without the operation abstractions, which can compensate the increased number of iterations. This is clear from the previously-presented results, which show that the solver with operation abstractions can decide more formulas in general.

Finally, we investigate the effect of our techniques on the number of benchmarks that were solved by *any* of the solvers. Although the benefit is not so large as far as the number of solved benchmarks is concerned, the operation abstractions improve the runtime. The virtual-best solver from Boolector, CVC4,

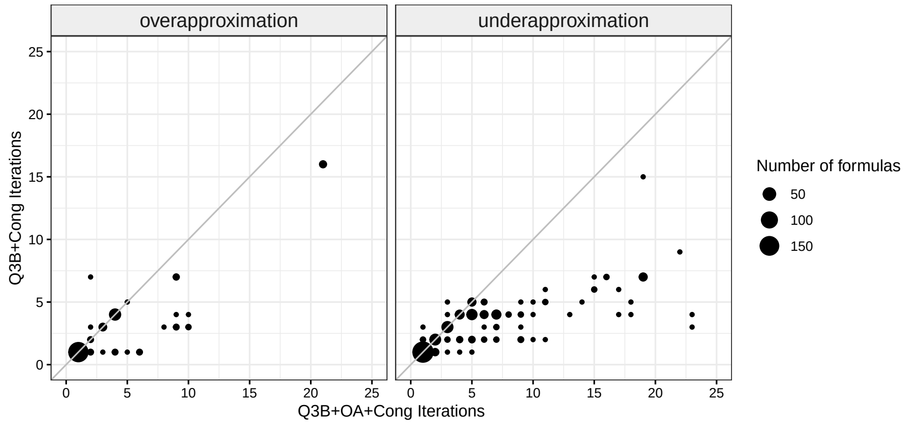


Figure 10: Scatter plots of the numbers of iterations of the abstraction refinement loops `solve_overapproximations` and `solve_underapproximations`, with and without the operation abstractions. The results are divided according to the type of the approximation that decided the given benchmark: the left subplot contains formulas decided by `solve_overapproximations` and the right subplot those that were decided by `solve_underapproximations` within the portfolio solver.

Z3, and Q3B can solve 5784 benchmarks in 272 minutes of wall time. After adding also the configuration Q3B+OA+Cong into the list of the solvers, the virtual-best solver solves 5787 benchmarks in 230 minutes of wall time. Two of the newly-solved benchmarks are from the family *preiner-scholl-smt08*; the third one is from *preiner-keymaera*. All three of these benchmarks were solved also by Q3B+OA, but neither was solved by Q3B+Cong.

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

`http://fi.muni.cz/~xjonas/papers/tcs_operation_abstractions/`

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 with the help of these abstractions, BDD-based SMT solver Q3B is able to solve more quantified bit-vector formulas from the SMT-LIB repository than the SMT solvers Boolector, CVC4, and Z3.

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, 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 and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[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] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Martin Jonáš and Jan Strejček. On simplification of formulas with unconstrained variables and quantifiers. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th*

International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings, volume 10491 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2017.

[12] Martin Jonáš and Jan Strejček. Abstraction of Bit-Vector Operations for BDD-Based SMT Solvers. In *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, volume 11187 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2018.

[13] Martin Jonáš and Jan Strejček. Q3B: An Efficient BDD-based SMT Solver for Quantified Bit-Vectors. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 64–73. Springer, 2019.

[14] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.

[15] 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.

[16] 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.

[17] Peter Navrátil. Adding Support for Bit-Vectors to BDD Libraries CUDD and Sylvan. Bachelor's thesis, Masaryk University, Faculty of Informatics, Brno, 2018.

[18] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Solving quantified bit-vectors using invertibility conditions. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 236–255, 2018.

[19] 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.

[20] Fabio Somenzi. CUDD: CU Decision Diagram Package Release 3.0.0. *University of Colorado at Boulder*, 2015.

[21] 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.

[22] 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.

[23] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding Bit-Vector 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.