Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams*

Martin Jonáš and Jan Strejček

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{martin.jonas, strejcek}@mail.muni.cz

Abstract. We describe a new approach to deciding satisfiability of quantified bit-vector formulas using binary decision diagrams and approximations. The approach is motivated by the observation that the binary decision diagram for a quantified formula is typically significantly smaller than the diagram for the subformula within the quantifier scope. The suggested approach has been implemented and the experimental results show that it decides more benchmarks from the SMT-LIB repository than state-of-the-art SMT solvers for this theory, namely Z3 and CVC4.

1 Introduction

During the last decades, the area of *Satisfiability Modulo Theories* (SMT) [6] solving has undergone steep development in both theory and practice. Achieved advances of SMT solving opened new research directions in program analysis and verification, where SMT solvers are now seen as standard tools.

Common programming languages provide basic datatypes of fixed size. Program variables of these datatypes naturally correspond to variables of the bit-vector logic, which can easily express bit-wise operations or arithmetic overflows. In spite of this natural correspondence, most SMT-based program analysis techniques model program variables by variables in the theory of integers. This may look a bit strange considering the fact that the satisfiability problem for the theory of integers is undecidable whenever an arbitrary usage of addition and multiplication is allowed, while the same problem is decidable for the bit-vector theory. The reasons for using the integer logic instead of the bit-vector logic are twofold. First, the satisfiability problem is NEXPTIME-complete even for formulas of the *quantifier-free fragment of the bit-vector logic* (QF_BV) with binary encoding of bit-vector sizes [21]. In this paper, we consider formulas with quantifiers and without uninterpreted functions. The precise complexity of the problem for this logic is an open question: it is known to be NEXPTIME-hard [21] and trivially solvable in EXPSPACE. Second and from the practical point of view more important, the SMT solvers for the theory of integers are often more efficient than the solvers for the theory of fixed-size bit-vectors.

While there are several SMT solvers for QF_BV formulas, only few of them can decide the *quantified bit-vector (*BV*) logic*. In particular, the logic is supported by CVC4 [3] and Z3 [16]. Relatively modest support of this logic from

* The research was supported by Czech Science Foundation, grant GBP202/12/G061.

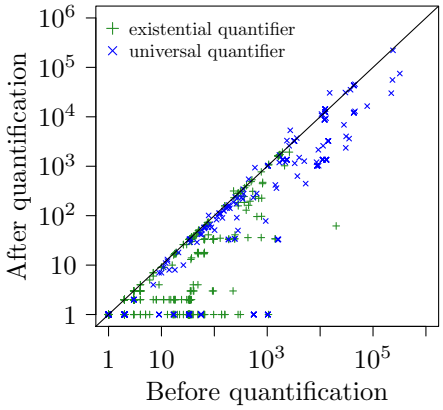


Fig. 1: Comparison of sizes (measured by the number of BDD nodes) of BDDs corresponding to all quantified subformulas in SMT-LIB benchmarks for BV logic, before and after quantification.

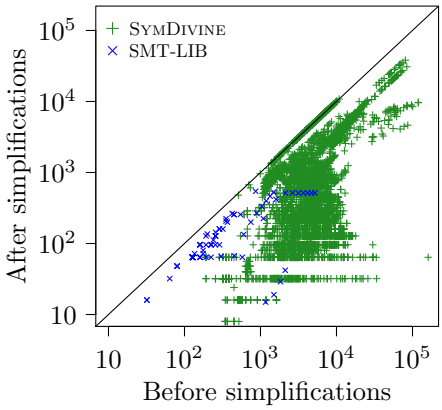


Fig. 2: Effect of simplifications on the number of bit variables in the formulas of the SMT-LIB and SymDivine benchmarks. Formulas simplified to *true* or *false* are not represented.

developers of SMT solvers is definitely not a consequence of a low demand from potential users. For example, in the program analysis community, BV formulas are suitable for description of various properties of program loops like loop invariants, ranking functions, or loop summaries [22], or to describe properties of symbolic representations of sets of program states, such as inclusion [7].

While current solvers for BV logic rely on *model-based quantifier instantiation* [25], we present a new algorithm based on *Binary Decision Diagrams* (BDDs) and approximations. BDDs have been previously used to implement satisfiability decision procedures for the propositional logic, however state-of-the-art CDCL-based solvers usually achieve much better performance. The main disadvantage of BDDs is low scalability: the size of a BDD corresponding to a propositional formula can be exponential in the number of propositional variables, and when a BDD becomes too large, some operations are very slow. Employment of BDDs in SMT solving makes more sense when formulas with quantifiers are considered: quantification usually reduces size of a BDD as it decreases the number of BDD variables. This can be documented by Figure 1, which compares the BDD sizes for formulas before and after existential or universal quantification.

There already exist some BDD-based tools deciding validity of quantified boolean formulas with the performance similar to state-of-the-art solvers for this problem [2, 23].

Our BDD-based algorithm for satisfiability of the BV logic consists of three main components:

– Formula simplifications, which reduce the number of variables in the formula and push quantifiers downwards in the syntax tree of the formula (which later helps to keep intermediate BDDs smaller as they are build in the bottom-up

manner). Formula simplifications can reduce some formulas to *true* or *false* and thus immediately decide their satisfiability.
- Construction of BDD using a specific variable ordering. The ordering has a significant influence on the BDD size.
- Formula approximations, which reduce the width of bit-vector variables in the formula and thus lead to smaller BDDs. Unsatisfiability of a formula over-approximation implies unsatisfiability of the original formula and an analogous statement holds for satisfiability of an under-approximation.

We present a minor contribution in each component. The main contribution of the paper is the fact that the algorithm based on the three parts can compete with leading SMT solvers for the BV logic, which participated in the BV category of SMT-COMP 2015 [1], namely Z3 and CVC4.

In the next section, we recall the definition of BV logic and BDDs, and briefly explain the main idea of the model-based quantifier instantiation technique employed by CVC4 and Z3. The proposed algorithm including the three main components is presented in Section 3. Section 4 is devoted to the implementation of the algorithm and to experimental results showing separately the effect of formula simplification, variable ordering, and approximations. The section also provides an experimental comparison of our solver with Z3 and CVC4. The paper closes with conclusions and intended directions of future work.

2 Preliminaries

2.1 Quantified Bit-Vector Formulas

In what follows, we assume a knowledge of the multi-sorted first-order logic and the model theory [18, 19]. Let $\mathbb{N}$ denote the set of positive integers.

The *bit-vector logic* is a multi-sorted first-order logic with the set of sort symbols $S = \{\texttt{bitvec}_i \mid i \in \mathbb{N}\}$, where $\texttt{bitvec}_i$ represents the sort of bit-vectors of length i, the set of function symbols

$$\begin{aligned} F = &\{c^i_{[n]} \mid n, i \in \mathbb{N}\} \cup \bigcup_{n\in\mathbb{N}} \{0_{[n]}, 1_{[n]}, \ldots, (2^n-1)_{[n]}\} \cup \\ &\cup \bigcup_{n\in\mathbb{N}} \{\texttt{not}_{[n]}, \texttt{and}_{[n]}, \texttt{or}_{[n]}, \texttt{shl}_{[n]}, \texttt{shr}_{[n]}, -_{[n]}, +_{[n]}, \times_{[n]}, /_{[n]}, \%_{[n]}\} \cup \\ &\cup \{\texttt{concat}_{[m,n]} \mid m, n \in \mathbb{N}\} \cup \{\texttt{extract}_{[n,i,j]} \mid n, i, j \in \mathbb{N}, i \leq j < m\}, \end{aligned}$$

and the set of the predicate symbols $P = \{=_{[n]}, <_{[n]} \mid n \in \mathbb{N}\}$. Arities of function and predicate symbols are described in Table 1.

The syntax of bit-vector formulas is defined in the standard way. Every formula can be transformed into the *negation normal form* (NNF), where negation is applied only to atomic subformulas and implication is not used at all.

A structure M is said to be a *model* for formula φ, if the formula φ is true in M and if M interprets all function and predicate symbols according to Table 1.

Symbol	Arity	Interpretation
$0_{[n]}, 1_{[n]}, \ldots$	$\texttt{bitvec}_n$	natural number constants
$c^1_{[n]}, c^2_{[n]}, \ldots$	$\texttt{bitvec}_n$	uninterpreted constants
$\texttt{not}_{[n]}$	$\texttt{bitvec}_n \to \texttt{bitvec}_n$	bit-wise negation
$\texttt{and}_{[n]}, \texttt{or}_{[n]}$	$\texttt{bitvec}_n \times \texttt{bitvec}_n \to \texttt{bitvec}_n$	bit-wise and, or
$\texttt{shl}_{[n]}, \texttt{shr}_{[n]}$	$\texttt{bitvec}_n \times \texttt{bitvec}_n \to \texttt{bitvec}_n$	bit-wise shift left, right
$-_{[n]}$	$\texttt{bitvec}_n \to \texttt{bitvec}_n$	two-complement negation
$+_{[n]}, \times_{[n]}$	$\texttt{bitvec}_n \times \texttt{bitvec}_n \to \texttt{bitvec}_n$	addition, multiplication
$/_{[n]}, \%_{[n]}$	$\texttt{bitvec}_n \times \texttt{bitvec}_n \to \texttt{bitvec}_n$	unsigned division, remainder
$\texttt{concat}_{[m,n]}$	$\texttt{bitvec}_m \times \texttt{bitvec}_n \to \texttt{bitvec}_{m+n}$	concatenation
$\texttt{extract}_{[m,i,j]}$	$\texttt{bitvec}_m \to \texttt{bitvec}_{j-i+1}$	extraction from i-th to j-th bit
$=_{[n]}, <_{[n]}$	$\texttt{bitvec}_n \times \texttt{bitvec}_n$	equality, unsigned less than

Table 1: Function and predicate symbols of the bit-vector logic.

Precise description of function and predicate symbols interpretation can be found in [4]. A closed formula is said to be *satisfiable* if it has a model.

We omit subscripts representing the sorts from the function and predicate symbols if the bit-width can be inferred from the context. If the sort of a variable or a constant is not specified, it is assumed to be $\texttt{bitvec}_{32}$. We also write $a, b, c, \ldots$ instead of uninterpreted constants $c^1, c^2, c^3, \ldots$. For example, $\forall x\,(x < a)$ denotes the formula $\forall x_{[32]}\,(x_{[32]} <_{[32]} c^1_{[32]})$. We write $\varphi[x_1, \ldots, x_n]$ for a formula φ, which may contain free variables $x_1, \ldots, x_n$. If $\varphi[x_1, \ldots, x_n]$ is a formula and $t_1, \ldots, t_n$ are terms of corresponding sorts, then $\varphi[t_1, \ldots, t_n]$ is the result of simultaneous substitution of free variables $x_1, \ldots, x_n$ in the formula φ by terms $t_1, \ldots, t_n$, respectively.

2.2 Model-Based Quantifier Instantiation

Satisfiability of the quantifier-free fragment of the bit-vector logic is traditionally solved by eager or lazy reduction to a propositional formula (bit-blasting) and subsequent call of a SAT solver. In the following, we describe the *model-based quantifier instantiation* algorithm [25], which is used by existing solvers for the full bit-vector logic.

Given a closed formula with quantifiers, the first step is to convert the formula to the negation normal form and apply Skolemization to obtain equisatisfiable formula of the form

$$\varphi \;\wedge\; \forall x_1, x_2, \ldots, x_n\,(\psi[x_1, \ldots, x_n]),$$

where φ and ψ are quantifier-free formulas. Then the QF_BV solver is invoked to check the satisfiability of the formula φ. If φ is unsatisfiable, then the entire formula is unsatisfiable. If φ is satisfiable, the QF_BV solver returns its model

M and another call to the QF_BV solver is made to determine whether M is also a model of $\forall x_1, x_2, \ldots, x_n\,(\psi)$. This is achieved by asking the solver whether the formula $\neg\widehat{\psi}$ is satisfiable, where $\widehat{\psi}$ is the formula ψ with uninterpreted constants replaced by their corresponding values in M. If $\neg\widehat{\psi}$ is not satisfiable, then the structure M is indeed a model of the formula $\forall x_1, x_2, \ldots, x_n\,(\psi)$, therefore the entire formula is satisfiable and M is its model. If $\neg\widehat{\psi}$ is satisfiable, we get values $v_1, \ldots, v_n$ such that $\neg\widehat{\psi}[v_1, \ldots, v_n]$ holds. To rule out M as a model, the instance $\psi[v_1, \ldots, v_n]$ of the quantified formula is added to the quantifier-free part, i.e. the formula φ is modified to

$$\varphi' \;\equiv\; \varphi \wedge \psi[v_1, \ldots, v_n],$$

and the procedure is repeated.

Example 1. *Consider the formula* $3 < a \;\wedge\; \forall x\,(\neg(a = 2 \times x))$. *The subformula* $3 < a$ *is satisfiable and* $a = 4$ *is its model. However, it is not a model of the formula* $\forall x\,(\neg(a = 2 \times x))$, *since the* QF_BV *solver called on the formula* $\neg(\neg(4 = 2 \times x))$ *returns* $x = 2$ *as a model. The next step is to decide the satisfiability of the formula* $3 < a \;\wedge\; \neg(a = 2 \times 2)$. *This formula is satisfiable and* $a = 5$ *is its model. Moreover, it is also a model of* $\forall x\,(\neg(a = 2 \times x))$ *as* $\neg(\neg(5 = 2 \times x))$ *is unsatisfiable. Hence, the input formula is satisfiable and* $a = 5$ *is its model.*

This algorithm is trivially terminating, since there is only a finite number of distinct models M of φ. However, in some cases exponentially many such models have to be ruled out before the solver is able to find a correct model or decide unsatisfiability of the whole formula. To overcome this issue, state-of-the-art SMT solvers do not use just instances of the form $\psi[v_1, \ldots, v_n]$ with concrete values, but employ heuristics such as E-matching [15, 17] or symbolic quantifier instantiation [25] to choose instances with ground terms which can potentially rule out more spurious models and thus significantly reduce the number of iterations of the algorithm. In practice, suitable ground terms substituted for quantified variables are selected only from subterms of the input formula. This strategy brings some drawbacks. For example, the formula

$$a = 2^4 \times b \;+\; 2^4 \times c \;\wedge\; \forall x\,(\neg(a = 2^4 \times x))$$

is unsatisfiable as the subformula $\forall x\,(\neg(a = 2^4 \times x))$ is true precisely when the value of a is not a multiple of 2^4, while $a = 2^4 \times b + 2^4 \times c$ implies that a is a multiple of 2^4. The quantifier instantiation can prove the unsatisfiability easily by using the instance $\psi[b+c]$ of $\psi[x] \equiv \neg(a = 2^4 \times x)$. However, the current tools do not consider this instance as $b+c$ is not a subterm of the formula. As a result, current tools can not decide satisfiability of this formula within a reasonable time limits.

2.3 Binary Decision Diagrams

A *binary decision diagram (BDD)* is a data structure proposed by Bryant [12] to succinctly represent all satisfying assignments of a boolean formula.

A BDD is a rooted directed acyclic graph with inner nodes labeled by boolean variables of the formula and two leaf nodes 0 and 1. Every inner node has two outgoing edges, one labeled with *true* and the other with *false*. Every assignment of boolean variables determines a path from the root to a leaf: from every inner node we follow the edge labeled with the truth value assigned to the variable corresponding to the node. The BDD represents all assignments that determine paths to leaf 1. Fixing an order in which variables can occur on paths from the root yields an *Ordered Binary Decision Diagram (OBDD)* and merging identical subgraphs of an OBDD and deleting every node whose two children are identical yields a *Reduced Ordered Binary Decision Diagram (ROBDD)*. The main advantage of ROBDDs is that for the fixed variable order every set of assignments corresponds to a unique ROBDD [12]. In the following, BDD always stands for ROBDD.

A BDD for a boolean formula can be built from BDDs for atomic subformulas in a bottom-up manner. Application of negation corresponds to switching the leaf nodes 0 and 1. For binary operators, there is a function `Apply` that gets an operator and two BDDs corresponding to the operands and produces the desired BDD. Using this function, one can also build a BDD representing a quantified boolean formula: if B is a BDD representing a formula φ, then the BDD for $\forall x\,(\varphi)$ is obtained by $\texttt{Apply}(\wedge, B[x \leftarrow \mathit{true}], B[x \leftarrow \mathit{false}])$ and the BDD for $\exists x\,(\varphi)$ by $\texttt{Apply}(\vee, B[x \leftarrow \mathit{true}], B[x \leftarrow \mathit{false}])$.

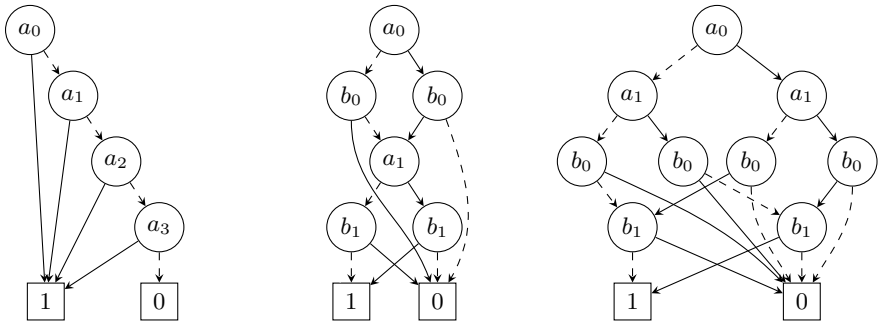
A BDD can also represent a set of all models of a BV formula. It is sufficient to decompose every bit-vector variable and every uninterpreted constant of bit-width n into n boolean variables and perform operations on individual bits. For example, all models of the formula $\forall x\,(\neg(a = 2^4 \times x))$ are represented by the BDD of Figure 3a, where the 32 bits of the uninterpreted constant a are denoted by boolean variables $a_0, a_1, \ldots, a_{31}$ in order from the least significant to the most significant bit. Boolean variables arising from bit-vector variables and uninterpreted constants are called *bit variables* henceforth. As usual, instead of labelling edges as *true* and *false*, edges are drawn as solid and dashed, respectively. Note that every unsatisfiable formula is represented by the BDD with the single node 0. By the BDD size we mean the number of its nodes.

3 Our Approach

This section first describes three main parts of our algorithm, namely formula simplifications, bit variable ordering for BDD construction, and approximations. Subsequently, the main algorithm is presented.

3.1 Formula Simplifications

As in most of modern SMT solvers, the first step of deciding satisfiability is simplification of the input formula. Besides trivial simplifications (e.g. $\varphi \wedge \varphi$ reduces to φ), we apply the following simplification rules.



(a) $\forall x\,(\neg(a = 2^4 \times x))$ (b) $a_{[2]} = b_{[2]}$ (c) $a_{[2]} = b_{[2]}$

Fig. 3: Examples of BDDs representing bit-vector formulas.

Miniscoping. Miniscoping [19] is a technique reducing the scope of universal quantifier over disjunctions whenever one disjunct has no free occurrences of the quantified variable, and over conjunctions by distributivity (existential quantifiers are handled analogously). The simplification rules are as follows:

$$\forall x\,(\varphi[x] \vee \psi) \;\rightsquigarrow\; \forall x\,(\varphi[x]) \vee \psi \qquad \forall x\,(\varphi[x] \wedge \psi[x]) \;\rightsquigarrow\; \forall x\,(\varphi[x]) \wedge \forall x\,(\psi[x])$$
$$\exists x\,(\varphi[x] \wedge \psi) \;\rightsquigarrow\; \exists x\,(\varphi[x]) \wedge \psi \qquad \exists x\,(\varphi[x] \vee \psi[x]) \;\rightsquigarrow\; \exists x\,(\varphi[x]) \vee \exists x\,(\psi[x])$$

Destructive Equality Resolution. Destructive equality resolution (DER) [25] eliminates a universally quantified variable x in a formula $\forall x\,\overline{Qy}\,(\neg(x = t) \vee \varphi[x])$, where t is a term that does not contain the variable x, and $\overline{Qy}$ is a sequence of variable quantifications. The formula is equivalent to $\forall x\,\overline{Qy}\,(x = t \rightarrow \varphi[x])$ and hence also to $\overline{Qy}\,(\varphi[t])$. The simplification rule is formulated as follows:

$$\forall x\,\overline{Qy}\,(\neg(x = t) \vee \varphi[x]) \quad\rightsquigarrow\quad \overline{Qy}\,(\varphi[t])$$

Constructive Equality Resolution. Constructive equality resolution (CER) is a dual version of DER. As far as we know, it was not considered before as solvers for quantified formulas typically work with formulas after Skolemization and thus without any existential quantifiers. CER can be formulated as the following simplification rule, where t and $\overline{Qy}$ have the same meaning as above:

$$\exists x\,\overline{Qy}\,(x = t \wedge \varphi[x]) \quad\rightsquigarrow\quad \overline{Qy}\,(\varphi[t])$$

Theory-Related Simplifications. We also perform several simplifications related to the interpretation of the function and predicate symbols in the BV logic. Examples of such simplifications are reductions $a_{[n]} + (-a_{[n]}) \;\rightsquigarrow\; 0_{[n]}$, $a_{[n]} \times 0_{[n]} \;\rightsquigarrow\; 0_{[n]}$, $a_{[n]}$ `and` $0_{[n]} \;\rightsquigarrow\; 0_{[n]}$, or $\texttt{extract}_{[n,i,j]}(0_{[n]}) \;\rightsquigarrow\; 0_{[j-i+1]}$.

Note that all mentioned simplification rules have no effect on models of the formula and thus they have no direct effect on the resulting BDD. However, a simplified formula has simpler subformulas and thus the intermediate BDDs are often smaller and the computation of the resulting BDD is faster.

3.2 Bit Variable Ordering

When constructing a BDD, one has to specify an order of BDD variables. In our case, BDD variables precisely correspond to bit variables. The order of these variables has a significant effect on the BDD size and its construction time. In some cases, the size of a BDD for a formula is linear in the number of BDD variables with one variable ordering, but exponential with another ordering.

For example, consider the formula $\phi_1 \equiv a_{[n]} = b_{[n]}$ for arbitrary $n \in \mathbb{N}$ and let $a_0, a_1, \ldots, a_{n-1}$ be the bits of a and $b_0, b_1, \ldots, b_{n-1}$ be the bits of b. We define two orderings:

$\leq_1$ All bit variables are ordered according to their significance (from the least to the most significant) and variables with the same significance are ordered by the order of the first occurrences of the corresponding bit-vector variables in the formula. For the considered formula ϕ_1, we get:

$$a_0 \leq_1 b_0 \leq_1 a_1 \leq_1 b_1 \leq_1 \ldots \leq_1 a_{n-1} \leq_1 b_{n-1}$$

$\leq_2$ Bit variables are ordered by the order of the first occurrences of the corresponding bit-vector variables in the formula and bit variables corresponding to the same bit-vector variable are ordered according to their significance (from the least to the most significant). For the considered formula, we get:

$$a_0 \leq_2 a_1 \leq_2 \ldots \leq_2 a_{n-1} \leq_2 b_0 \leq_2 b_1 \leq_2 \ldots \leq_2 b_{n-1}$$

The BDD for ϕ_1 using the ordering $\leq_1$ has $3n+2$ nodes, while the BDD for the same formula and $\leq_2$ has $3 \cdot 2^n - 1$ nodes. Figures 3b and 3c show these BDDs for $n = 2$ and orderings $\leq_1$ and $\leq_2$, respectively.

These orderings can lead to opposite results with other formulas. For example, the size of the BDD for the formula

$$\phi_2 \equiv (c^1_{[2]} = c^2_{[2]}\ \texttt{shr}\ 1_{[2]}) \wedge (c^3_{[2]} = c^4_{[2]}\ \texttt{shr}\ 1_{[2]}) \wedge \ldots \wedge (c^{2n-1}_{[2]} = c^{2n}_{[2]}\ \texttt{shr}\ 1_{[2]})$$

using the ordering $\leq_1$ is $2^{n+2} - 1$, while it is only $4n + 2$ for $\leq_2$. In general, choosing the optimal variable ordering is an NP-complete problem [10]. In the following, we introduce an ordering $\leq_3$ combining advantages of $\leq_1$ and $\leq_2$.

Let V be the set of bit-vector variables and uninterpreted constants appearing in an input formula φ. Elements $x, y \in V$ are *dependent*, written $x \sim y$, if they both appear in some atomic subformula of φ. Let $\simeq$ be the equivalence on V defined as the transitive closure of $\sim$. Every $v \in V$ then defines an equivalence class $[v]_\simeq$ of transitively dependent elements.

$\leq_3$ Bit variables are first ordered according to $\leq_1$ within corresponding equivalence classes of $\simeq$ and the equivalence classes are then ordered by the first occurrences of BV variables in φ. In particular for $u \not\simeq v$, $u_i \leq_3 v_j$ if there is a BV variable in $[u]_\simeq$, which occurs in φ before all BV variables of $[v]_\simeq$.

Note that for both formulas ϕ_1, ϕ_2 mentioned above, $\leq_3$ coincides with the better of the orderings $\leq_1$ and $\leq_2$.

In addition to the initial variable ordering, there are several techniques that dynamically reorder the BDD variables to reduce the BDD size. We use *sifting* [24] as usually the most successful one [20].

3.3 Approximations

For some BV formulas, e.g. formulas containing non-linear multiplication, the size of the BDD representation is exponential for every possible variable ordering [13]. Fortunately, satisfiability of these formulas can be often decided using their over-approximations or under-approximations. Given a formula φ, its *under-approximation* is any formula $\underline{\varphi}$ that logically entails φ, and its *over-approximation* is any formula $\overline{\varphi}$ logically entailed by φ. Clearly, every model of $\underline{\varphi}$ is also a model of φ and if an under-approximation $\underline{\varphi}$ is satisfiable, so is the formula φ. Similarly, if an over-approximation $\overline{\varphi}$ is unsatisfiable, so is φ.

The model-based quantifier instantiation presented in Section 2.2 can be seen as a technique based on iterative over-approximation refinement: the formulas φ, $\varphi \wedge \psi[\bar{v}]$, ... are over-approximations of $\varphi \wedge \forall \bar{x}\,(\psi[\bar{x}])$. A different concepts of approximations can be found in SMT solvers for QF_BV formulas. For example, the SMT solver UCLID over-approximates a formula in the negation normal form by replacing some subformulas with fresh uninterpreted constants [14]. Further, SMT solvers UCLID and Boolector under-approximate a formula by restricting the value of m most significant bits of a bit-vector variable while leaving the remaining bits unchanged [11,14]. The number of bit variables used to represent the bit-vector variable or uninterpreted constant is called its *effective bit-width*. This approach inspired both over- and under-approximation used in our algorithm.

Let $a_{[n]}$ be a variable or an uninterpreted constant of bit-width n and $e \in \mathbb{N}$ be its desired effective bit-width. If $e \geq n$, we leave $a_{[n]}$ unchanged. Otherwise, we consider four different ways to reduce the effective bit-width of $a_{[n]}$ to e:

zero-extension uses the effective bit-width to represent the e least significant bits and sets the $n - e$ most significant bits to 0.

sign-extension also uses the effective bit-width to represent the e least significant bits and sets the $n - e$ most significant bits to the value of the e-th least significant bit.

right zero-extension uses the effective bit-width to represent the e most significant bits and sets the $n - e$ least significant bits to 0.

right sign-extension also uses the effective bit-width to represent the e most significant bits and sets the $n - e$ least significant bits to the value of the e-th most significant bit.

All considered extensions are illustrated in Figure 4. The first two extensions are taken directly from [14], while the other two are original. One can easily see that each extension reduces the domain of $a_{[n]}$ to a different subdomain of size 2^e. Another extensions are suggested in [11], e.g. *one-extension* defined analogously to the zero-extension. Our choice of considered extensions is motivated by exploration of values near corner cases as well as by reduction of BDD size. In particular, we do not consider one-extension because it produces only few zero bits which are desired as they tend to reduce the size of BDDs for multiplication.

In the following, the term *extension* always refers either to zero-extension, or to sign-extension. In an over-approximation, we apply a selected reduction to all

Fig. 4: Reductions of $a_{[6]} = a_5a_4a_3a_2a_1a_0$ to 3 effective bits.

universally quantified variables. Given a formula φ and $e \in \mathbb{N}$, let $\overline{\varphi}_e$ denote the formula φ where the effective bit-width of each universally quantified variable is reduced to e by the chosen extension. Further, $\overline{\varphi}_{-e}$ denotes the formula obtained by application of the right counterpart of the chosen extension.

In an under-approximation, we apply the selected reduction to all existentially quantified variables and uninterpreted constants. Given a formula φ and $e \in \mathbb{N}$, let $\underline{\varphi}_e$ and $\underline{\varphi}_{-e}$ denote the formula φ where the effective bit-width of each existentially quantified variable and uninterpreted constant is reduced to e by the chosen extension or its right counterpart, respectively.

The following theorem establishes that, for each formula φ in the negation normal form, $\overline{\varphi}_e$, $\overline{\varphi}_{-e}$ are over-approximations (and analogously for under-approximations). The theorem can be easily proven by an induction on the structure of the formula φ.

Theorem 1. *For every formula φ in the NNF and any $e \in \mathbb{N}$, it holds:*

1. *If M is a model of φ, then M is also a model of $\overline{\varphi}_e$ and $\overline{\varphi}_{-e}$.*
2. *If M is a model of $\underline{\varphi}_e$ or $\underline{\varphi}_{-e}$, then M is also a model of φ.*

Corollary 1. *For every formula φ in the NNF and any $e \in \mathbb{N}$, it holds:*

1. *If the formula $\overline{\varphi}_e$ or $\overline{\varphi}_{-e}$ is unsatisfiable, so is the formula φ.*
2. *If the formula $\underline{\varphi}_e$ or $\underline{\varphi}_{-e}$ is satisfiable, so is the formula φ.*

3.4 The Algorithm

In this section, we present the complete algorithm deciding satisfiability of BV formulas. In the algorithm, we use a procedure `ConvertToBDD` which converts a formula to the corresponding BDD recursively on the formula structure. For a given input formula φ, the algorithm proceeds in the following steps:

1. Simplify the formula φ using the rules discussed in Section 3.1 up to the fix-point and convert it to the negation normal form. If the result is *true*, return `SAT`. If the result is *false*, return `UNSAT`.
2. Take the simplified formula in NNF φ' and compute a chosen ordering $\leq$ as described in Section 3.2. This ordering will be used as the initial ordering in the procedure `ConvertToBDD`.
3. Call `ConvertToBDD`(φ') to compute the BDD corresponding to φ'. If the root node of the BDD has label 0, return `UNSAT`. Otherwise return `SAT`.
4. If the procedure `ConvertToBDD` called in the previous step has not finished within 0.1 seconds, additionally run in parallel:

(a) *Under-approximations*: Sequentially compute `ConvertToBDD`($\underline{\varphi'}_i$) for $i = 1, -1, 2, -2, 4, -4, 6, -6, \ldots$ until reaching the greatest bit-width of a bit-vector variable in φ'. If any of the resulting BDDs has a root node distinct from the leaf 0, return `SAT`.
(b) *Over-approximations*: Sequentially compute `ConvertToBDD`($\overline{\varphi'}_i$) for $i = 1, -1, 2, -2, 4, -4, 6, -6, \ldots$ until reaching the greatest bit-width of a bit-vector variable in φ'. If any of the produced BDDs has a root node labeled by 0, return `UNSAT`.

The algorithm is parametrized by the choice of an ordering and reductions for approximations. Regardless these parameters, the algorithm is sound and complete. The decision to start the solvers using approximations after 0.1 second is based on our experiments. In practice, the procedure `ConvertToBDD` may need exponential time and memory and thus the algorithm may not finish within reasonable limits.

4 Implementation and Experimental Results

We have implemented the presented algorithm in an experimental SMT solver called Q3B. The implementation is written in C++, relies on the BDD package BuDDy[1], and uses the API of Z3 to parse the input formula in the SMT-LIB 2.5 format [4] and to perform some formula simplifications. As the BuDDy package does not support allocation of multiple BDD instances, we run separate processes for the base solver and for computing over- and under-approximations. The execution of these three processes is controlled by a Python wrapper.

We have evaluated our solver on two sets of BV formulas. The first set consists of all 191 formulas in the category BV of the SMT-LIB benchmark repository [5]. The second set contains 5 461 formulas generated by the model checker SymDivine [7] when run on verification tasks from SV-COMP [8]. These formulas correspond to checking equivalence of two symbolic states of the verified program. In total, SymDivine generated 1 462 500 formulas. For tasks with more than 25 generated formulas, we randomly picked 25 formulas to keep the number of formulas reasonable.

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 3 processor cores, 4 GB of RAM and 20 minutes of CPU time (if not stated otherwise). All measured times are CPU times. For reliable benchmarking we employed BenchExec [9], a tool that allocates specified resources for a program execution and measures their use precisely.

All used benchmarks and detailed experimental results are available at `http://www.fi.muni.cz/~xstrejc/sat2016.tar.gz`. Q3B is available under the MIT License and hosted at GitHub: `https://github.com/martinjonas/Q3B`.

In the following, we demonstrate the effect of formula simplifications on the formulas and the effect of various algorithm parameters on its efficiency. At the end, we compare our solver with the best parameters against CVC4 and Z3.

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

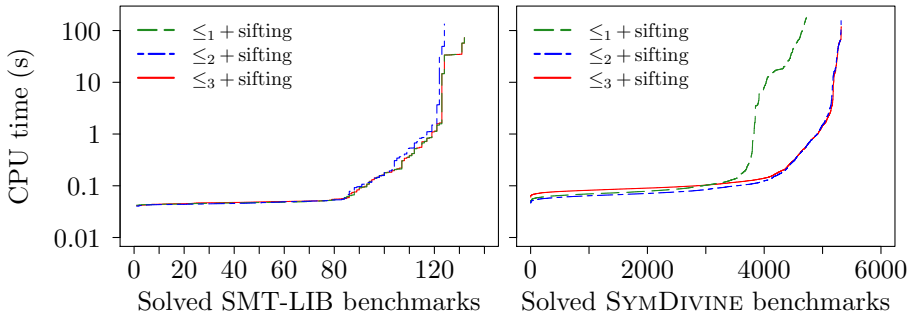


Fig. 5: Quantile plot of the number of solved benchmarks for each of three described initial variable orderings.

Formula Simplifications. Considered formula simplifications reduced 108 of 191 SMT-LIB benchmarks and 300 of 5 461 SymDivine benchmarks to *true* or *false*. Additionally, 1 276 SymDivine benchmarks were reduced to a quantifier-free formulas, which is not the case for any SMT-LIB benchmark. Figure 2 shows the number of bit variables (i.e. the sum of bit-widths of all bit-vector variables and uninterpreted constants in the formula) of each formula before and after simplification.

Variable Ordering. To compare the effect of BDD variable orderings $\leq_1$, $\leq_2$, and $\leq_3$ defined in Section 3.2, we run our tool with each of these initial orderings on all considered benchmarks. Recall that sifting method is used for dynamic variable reordering. The solver has been executed without approximations (to ensure that approximations will not hide the effect of the initial ordering) and with CPU time limited to 3 minutes. The results are shown in Figure 5.

When SMT-LIB are considered, the worst performing initial ordering is $\leq_2$. The results for $\leq_1$ and $\leq_3$ are almost identical as nearly all bit-vector variables in these benchmarks are mutually transitively dependent. For SymDivine benchmarks, initial ordering $\leq_1$ performs the worst. The results for $\leq_2$ and $\leq_3$ are very similar, as SymDivine formulas usually contain a large number of mutually independent groups of variables. The solver using $\leq_3$ decided 3 more formulas than the solver using $\leq_2$. To sum up, since now we always use the ordering $\leq_3$ as it provides better overall performance than $\leq_1$ and $\leq_2$.

Note that the solver runs usually faster when executed without sifting, as the dynamic reordering causes some computational overhead. However, with sifting it decides 2 more SMT-LIB benchmarks and 9 more SymDivine benchmarks.

Approximations. To compare the effect of the considered effective bit-width reductions, we run the solver once with approximations based on (right) zero-extension, and again with approximations based on (right) sign-extension. Quantile plots in Figure 6 show results for zero-extension and sign-extension on SMT-LIB benchmarks. The results are presented separately for satisfiable and unsat-

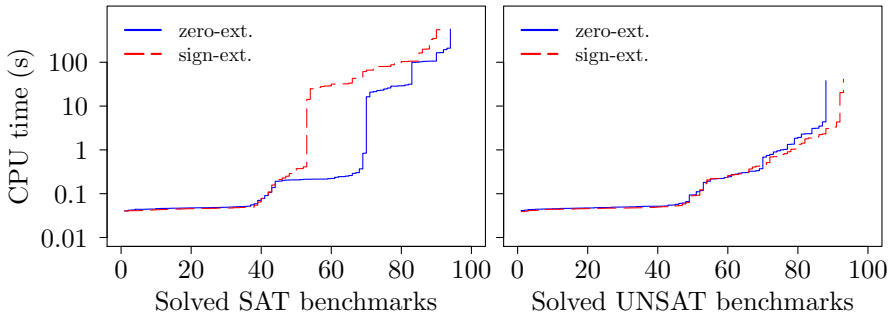


Fig. 6: Quantile plot of the number of solved SMT-LIB benchmarks using approximation via sign-extension and zero-extension compared by the CPU time.

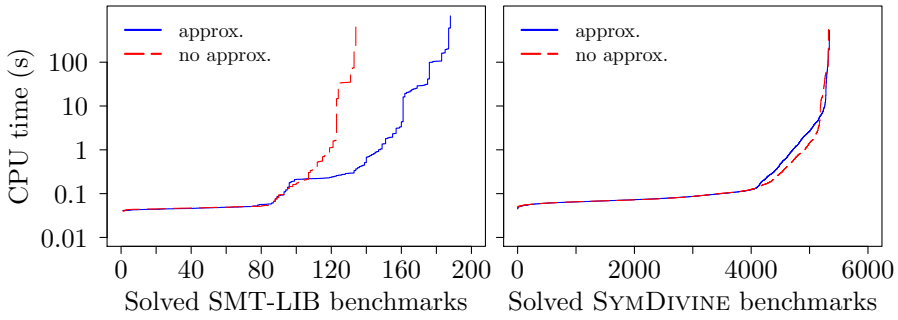


Fig. 7: Quantile plot of the number of solved benchmarks with and without approximations compared by the CPU time.

isfiable formulas. On satisfiable formulas, approximation using zero-extension performs better and can decide 3 more satisfiable formulas. On the contrary, on unsatisfiable formulas sign-extension performs better and can decide 5 formulas more. Corresponding plots for SymDivine formulas are not presented, since the difference in CPU times was insignificant. Based on this observation and the fact that satisfiability can be decided by an under-approximation and unsatisfiability by an over-approximation, the default bit-width reduction method in our solver is zero-extension for under-approximations and sign-extension for over-approximations.

Further, to show the contribution of approximations, we compare the solver using the proposed algorithm as described in the section 3.4 against the same algorithm without approximations. Figure 7 shows quantile plots corresponding to measured CPU times. With approximations, the solver was able to decide 54 more SMT-LIB formulas. The difference is less significant when SymDivine formulas are considered, as they mostly do not contain difficult arithmetic; only 15 more of SymDivine formulas were decided using approximations.

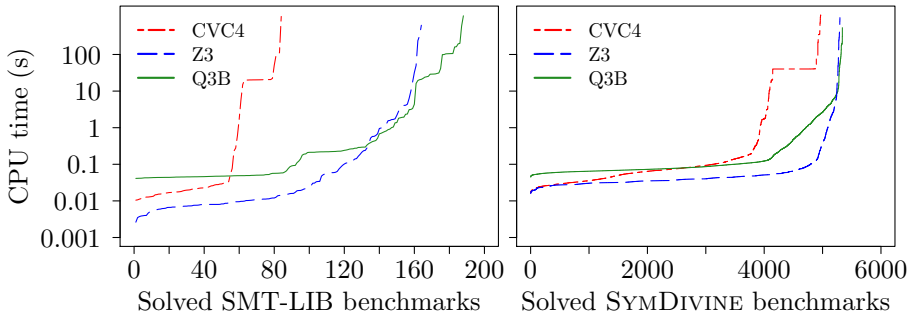


Fig. 8: Quantile plot of the number of benchmarks which CVC4, Q3B, and Z3 solved compared by the CPU time.

Comparison. Finally, we compare our solver (with the parameters selected by the previous experiments) to the current stable versions of leading SMT solvers for BV logic, namely to the version 4.4.1 of Z3 [16] and the version 1.4 of CVC4 [3]. We also tested the latest development version of CVC4 (2016-02-25), but it decided some SymDivine benchmarks incorrectly. The solver Z3 was executed with the default settings, CVC4 was executed with settings supplied for the SMT-competition, where the benchmarks from the SMT-LIB benchmark repository are used.

Table 2 shows summary results of the solvers CVC4, Z3, and Q3B on the two benchmark sets. Additionally, Table 3 shows for each pair of solvers the number of formulas which were decided by one solver, but not by the other one. Out of the 191 SMT-LIB benchmarks, CVC4 solves 84 benchmarks, Z3 decides 164 benchmarks, and our solver can decide 188 benchmarks. Out of the 5 461 SymDivine benchmarks, CVC4 decides 4 969 benchmarks, Z3 solves 5 297 benchmarks, and our solver Q3B decides 5 339 benchmarks. To sum up, in the number of decided benchmarks Q3B outperforms both CVC4 and Z3. Moreover, only 1 of all considered formulas was solved by Z3 and not by Q3B, and no formula was solved by CVC4 and not by Q3B.

Further, quantile plots of Figure 8 show numbers of input formulas each of the solvers was able to decide within different CPU time limits. Note that the y axis has the logarithmic scale. On the easy instances, our experimental solver can not compete with highly optimized solvers as CVC4 and Z3. The initial delay of Q3B is caused by an overhead of a process creation within the Python wrapper. However, as the instances become harder, the difference in solving times decreases. In particular, Q3B solves more SMT-LIB benchmarks than CVC4 whenever the CPU time limit is longer than 0.05 s and more than Z3 for any CPU time limit over 0.39 s. For SymDivine benchmarks, these thresholds are 0.08 s for CVC4 and 8.72 s for Z3. Note that Q3B uses 3 parallel processes and hence its wall times are usually three times shorter than presented CPU times, while wall times are the same as CPU times for Z3 and CVC4.

	SMT-LIB				SymDivine			
	sat	unsat	unknown	timeout	sat	unsat	unknown	timeout
CVC4	29	55	32	75	1 124	3 845	2	490
Z3	71	93	5	22	1 135	4 162	22	142
Q3B	94	94	0	3	1 137	4 202	0	122

Table 2: For each benchmark set and each solver, the table provides the numbers of formulas decided as satisfiable (*sat*), unsatisfiable (*unsat*), or undecided with the result unknown or because of an error (*unknown*), or a *timeout*.

	SMT-LIB			SymDivine		
	CVC4	Z3	Q3B	CVC4	Z3	Q3B
CVC4	–	0	0	–	21	0
Z3	80	–	1	349	–	0
Q3B	104	25	–	370	42	–

Table 3: For each pair of solvers, the table shows the number of benchmarks, which were solved by the solver in the corresponding row, but not by the solver in the corresponding column.

5 Conclusions

We presented a new SMT solving algorithm for quantified bit-vector formulas. While current SMT solvers for this logic typically rely on model-based quantifier instantiation and an SMT solver for quantifier-free bit-vector formulas, our algorithm is based on BDDs (with a specific initial variable ordering) and approximations. We have implemented the algorithm and experimental results indicate that our approach can compete with state-of-the-art SMT solvers CVC4 and Z3. In fact, it decides more formulas than the mentioned solvers.

We plan to further develop the algorithm and the tool. In particular, we plan to add a support for arrays and uninterpreted functions as these are useful for modelling some features of computer programs. We would also like to investigate possible approximations of bit-vector operations and predicates, or to develop some fine-grained methods for a targeted approximation refinement.

References

1. The 10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015). `http://smtcomp.sourceforge.net/2015/`. 2015.
2. Gilles Audemard and Lakhdar Sais. SAT based BDD solver for quantified boolean formulas. In *16th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2004*, pages 82–89, 2004.
3. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification, CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
4. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.
5. Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
6. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
7. Petr Bauch, Vojtěch Havel, and Jiří Barnat. LTL Model Checking of LLVM Bitcode with Symbolic Data. In *MEMICS 2014*, volume 8934 of *LNCS*, pages 47–59. Springer, 2014.
8. Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*, volume 9035 of *LNCS*, pages 401–416. Springer, 2015.
9. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings*, pages 160–178, 2015.
10. Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *Computers, IEEE Transactions on*, 45(9):993–1002, 1996.
11. Robert Brummayer and Armin Biere. Effective Bit-Width and Under-Approximation. In *Computer Aided Systems Theory, EUROCAST 2009*, volume 5717 of *LNCS*, pages 304–311. Springer, 2009.
12. Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
13. 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.
14. Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *LNCS*, pages 358–372. Springer, 2007.
15. Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT Solvers. In *Automated Deduction, CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
16. Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
17. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, 2005.

18. Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, 2001.
19. John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 1st edition, 2009.
20. Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, 2009.
21. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of Fixed-Size Bit-Vector Logics. *Theory of Computing Systems*, 7913:1–54, 2015.
22. 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.
23. Oswaldo Olivo and E. Allen Emerson. A More Efficient BDD-Based QBF Solver. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming, CP 2011*, volume 6876 of *LNCS*, pages 675–690. Springer, 2011.
24. Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993*, pages 42–47, 1993.
25. Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.