








# Solving Quantified Bit-Vectors Using Invertibility Conditions

Aina Niemetz<sup>1</sup> , Mathias Preiner<sup>1</sup> ,  
Andrew Reynolds<sup>2</sup> , Clark Barrett<sup>1</sup> ,  
and Cesare Tinelli<sup>2</sup> 

<sup>1</sup> Stanford University, Stanford, USA  
niemetz@cs.stanford.edu

<sup>2</sup> The University of Iowa, Iowa City, USA



**Abstract.** We present a novel approach for solving quantified bit-vector formulas in Satisfiability Modulo Theories (SMT) based on computing symbolic inverses of bit-vector operators. We derive conditions that precisely characterize when bit-vector constraints are invertible for a representative set of bit-vector operators commonly supported by SMT solvers. We utilize syntax-guided synthesis techniques to aid in establishing these conditions and verify them independently by using several SMT solvers. We show that invertibility conditions can be embedded into quantifier instantiations using Hilbert choice expressions, and give experimental evidence that a counterexample-guided approach for quantifier instantiation utilizing these techniques leads to performance improvements with respect to state-of-the-art solvers for quantified bit-vector constraints.

## 1 Introduction

Many applications in hardware and software verification rely on Satisfiability Modulo Theories (SMT) solvers for bit-precise reasoning. In recent years, the quantifier-free fragment of the theory of fixed-size bit-vectors has received a lot of interest, as witnessed by the number of applications that generate problems in that fragment and by the high, and increasing, number of solvers that participate in the corresponding division of the annual SMT competition. Modeling properties of programs and circuits, e.g., universal safety properties and program invariants, however, often requires the use of *quantified* bit-vector formulas. Despite a multitude of applications, reasoning efficiently about such formulas is still a challenge in the automated reasoning community.

The majority of solvers that support quantified bit-vector logics employ instantiation-based techniques [8, 21, 22, 25], which aim to find conflicting ground instances of quantified formulas. For that, it is crucial to select good instantiations for the universal variables, or else the solver may be overwhelmed by the

---

This work was partially supported by DARPA under award No. FA8750-15-C-0113 and the National Science Foundation under award No. 1656926.

number of ground instances generated. For example, consider a quantified formula  $\psi = \forall x. (x + s \not\approx t)$  where  $x$ ,  $s$  and  $t$  denote bit-vectors of size 32. To prove that  $\psi$  is unsatisfiable we can instantiate  $x$  with all  $2^{32}$  possible bit-vector values. However, ideally, we would like to find a proof that requires much fewer instantiations. In this example, if we instantiate  $x$  with the symbolic term  $t - s$  (the inverse of  $x + s \approx t$  when solved for  $x$ ), we can immediately conclude that  $\psi$  is unsatisfiable since  $(t - s) + s \not\approx t$  simplifies to false.

Operators in the theory of bit-vectors are not always invertible. However, we observe it is possible to identify quantifier-free conditions that precisely *characterize* when they are. We do that for a representative set of operators in the standard theory of bit-vectors supported by SMT solvers. For example, we have proven that the constraint  $x \cdot s \approx t$  is solvable for  $x$  if and only if  $(-s \mid s) \& t \approx t$  is satisfiable. Using this observation, we develop a novel approach for solving quantified bit-vector formulas that utilizes invertibility conditions to generate symbolic instantiations. We show that invertibility conditions can be embedded into quantifier instantiations using Hilbert choice functions in a sound manner. This approach has compelling advantages with respect to previous approaches, which we demonstrate in our experiments.

More specifically, this paper makes the following *contributions*.

- We derive and present invertibility conditions for a representative set of bit-vector operators that allow us to model all bit-vector constraints in SMT-LIB [3].
- We provide details on how invertibility conditions can be automatically synthesized using syntax-guided synthesis (SyGuS) [1] techniques, and make public 162 available challenge problems for SyGuS solvers that are encodings of this task.
- We prove that our approach can efficiently reduce a class of quantified formulas, which we call *unit linear invertible*, to quantifier-free constraints.
- Leveraging invertibility conditions, we implement a novel quantifier instantiation scheme as an extension of the SMT solver CVC4 [2], which shows improvements with respect to state-of-the-art solvers for quantified bit-vector constraints.

*Related Work.* Quantified bit-vector logics are currently supported by the SMT solvers Boolector [16], CVC4 [2], Yices [7], and Z3 [6] and a Binary Decision Diagram (BDD)-based tool called Q3B [14]. Out of these, only CVC4 and Z3 provide support for combining quantified bit-vectors with other theories, e.g., the theories of arrays or real arithmetic. Arbitrarily nested quantifiers are handled by all but Yices, which only supports bit-vector formulas of the form  $\exists x \forall y. Q[x, y]$  [8]. For quantified bit-vectors, CVC4 employs counterexample-guided quantifier instantiation (CEGQI) [22], where concrete models of a set of ground instances and the negation of the input formula (the counterexamples) serve as instantiations for the universal variables. In Z3, model-based quantifier instantiation (MBQI) [10] is combined with a template-based model finding procedure [25]. In contrast to CVC4, Z3 not only relies on concrete counterexamples as candidates for quantifier instantiation but generalizes these counterexamples to generate symbolic

instantiations by selecting ground terms with the same model value. Boolec-tor employs a syntax-guided synthesis approach to synthesize interpretations for Skolem functions based on a set of ground instances of the formula, and uses a counterexample refinement loop similar to MBQI [21]. Other counterexample-guided approaches for quantified formulas in SMT solvers have been considered by Bjørner and Janota [4] and by Reynolds et al. [23], but they have mostly targeted quantified linear arithmetic and do not specifically address bit-vectors. Quantifier elimination for a fragment of bit-vectors that covers modular linear arithmetic has been recently addressed by John and Chakraborty [13], although we do not explore that direction in this paper.

## 2 Preliminaries

We assume the usual notions and terminology of many-sorted first-order logic with equality (denoted by  $\approx$ ). Let  $S$  be a set of *sort symbols*, and for every sort  $\sigma \in S$  let  $X_\sigma$  be an infinite set of *variables of sort*  $\sigma$ . We assume that sets  $X_\sigma$  are pairwise disjoint and define  $X$  as the union of sets  $X_\sigma$ . Let  $\Sigma$  be a *signature* consisting of a set  $\Sigma^s \subseteq S$  of sort symbols and a set  $\Sigma^f$  of interpreted (and sorted) function symbols  $f^{\sigma_1 \cdots \sigma_n \sigma}$  with arity  $n \geq 0$  and  $\sigma_1, \dots, \sigma_n, \sigma \in \Sigma^s$ . We assume that a signature  $\Sigma$  includes a Boolean sort **Bool** and the Boolean constants  $\top$  (true) and  $\perp$  (false). Let  $\mathcal{I}$  be a  $\Sigma$ -*interpretation* that maps: each  $\sigma \in \Sigma^s$  to a non-empty set  $\sigma^\mathcal{I}$  (the *domain* of  $\mathcal{I}$ ), with  $\mathbf{Bool}^\mathcal{I} = \{\top, \perp\}$ ; each  $x \in X_\sigma$  to an element  $x^\mathcal{I} \in \sigma^\mathcal{I}$ ; and each  $f^{\sigma_1 \cdots \sigma_n \sigma} \in \Sigma^f$  to a total function  $f^\mathcal{I}: \sigma_1^\mathcal{I} \times \dots \times \sigma_n^\mathcal{I} \rightarrow \sigma^\mathcal{I}$  if  $n > 0$ , and to an element in  $\sigma^\mathcal{I}$  if  $n = 0$ . If  $x \in X_\sigma$  and  $v \in \sigma^\mathcal{I}$ , we denote by  $\mathcal{I}[x \mapsto v]$  the interpretation that maps  $x$  to  $v$  and is otherwise identical to  $\mathcal{I}$ . We use the usual inductive definition of a satisfiability relation  $\models$  between  $\Sigma$ -interpretations and  $\Sigma$ -formulas.

We assume the usual definition of well-sorted terms, literals, and formulas as **Bool** terms with variables in  $X$  and symbols in  $\Sigma$ , and refer to them as  $\Sigma$ -terms,  $\Sigma$ -atoms, and so on. A *ground* term/formula is a  $\Sigma$ -term/formula without variables. We define  $\mathbf{x} = (x_1, \dots, x_n)$  as a tuple of variables and write  $Q\mathbf{x}\varphi$  with  $Q \in \{\forall, \exists\}$  for a *quantified* formula  $Qx_1 \cdots Qx_n \varphi$ . We use  $\text{Lit}(\varphi)$  to denote the set of  $\Sigma$ -literals of  $\Sigma$ -formula  $\varphi$ . For a  $\Sigma$ -term or  $\Sigma$ -formula  $e$ , we denote the *free variables* of  $e$  (defined as usual) as  $FV(e)$  and use  $e[\mathbf{x}]$  to denote that the variables in  $\mathbf{x}$  occur free in  $e$ . For a tuple of  $\Sigma$ -terms  $\mathbf{t} = (t_1, \dots, t_n)$ , we write  $e[\mathbf{t}]$  for the term or formula obtained from  $e$  by simultaneously replacing each occurrence of  $x_i$  in  $e$  by  $t_i$ . Given a  $\Sigma$ -formula  $\varphi[x]$  with  $x \in X_\sigma$ , we use Hilbert's *choice* operator  $\varepsilon$  [12] to describe *properties* of  $x$ . We define a *choice function*  $\varepsilon x. \varphi[x]$  as a term where  $x$  is bound by  $\varepsilon$ . In every interpretation  $\mathcal{I}$ ,  $\varepsilon x. \varphi[x]$  denotes some value  $v \in \sigma^\mathcal{I}$  such that  $\mathcal{I}[x \mapsto v]$  satisfies  $\varphi[x]$  if such values exist, and denotes an arbitrary element of  $\sigma^\mathcal{I}$  otherwise. This means that the formula  $\exists x. \varphi[x] \Leftrightarrow \varphi[\varepsilon x. \varphi[x]]$  is satisfied by every interpretation.

A *theory*  $T$  is a pair  $(\Sigma, I)$ , where  $\Sigma$  is a signature and  $I$  is a non-empty class of  $\Sigma$ -interpretations (the *models* of  $T$ ) that is closed under variable reassignment, i.e., every  $\Sigma$ -interpretation that only differs from an  $\mathcal{I} \in I$  in how it interprets

**Table 1.** Set of considered bit-vector operators with corresponding SMT-LIB 2 syntax.

Symbol	SMT-LIB syntax	Sort
$\approx, <_u, >_u, <_s, >_s$	$=, \text{bvult}, \text{bvugt}, \text{bvslt}, \text{bvsgt}$	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \text{Bool}$
$\sim, -$	$\text{bvnot}, \text{bvneg}$	$\sigma_{[n]} \rightarrow \sigma_{[n]}$
$\&,  , \ll, \gg, \gg_a$	$\text{bvand}, \text{bvor}, \text{bvshl}, \text{bvshr}, \text{bvashr}$	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$+, \cdot, \text{mod}, \div$	$\text{bvadd}, \text{bvmul}, \text{bvurem}, \text{bvudiv}$	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$\circ$	$\text{concat}$	$\sigma_{[n]} \times \sigma_{[m]} \rightarrow \sigma_{[n+m]}$
$[u : l]$	$\text{extract}$	$\sigma_{[n]} \rightarrow \sigma_{[u-l+1]}, 0 \leq l \leq u < n$

variables is also in  $I$ . A  $\Sigma$ -formula  $\varphi$  is  $T$ -satisfiable (resp.  $T$ -unsatisfiable) if it is satisfied by some (resp. no) interpretation in  $I$ ; it is  $T$ -valid if it is satisfied by all interpretations in  $I$ . A choice function  $\varepsilon x. \varphi[x]$  is  $(T)$ -valid if  $\exists x. \varphi[x]$  is  $(T)$ -valid. We refer to a term  $t$  as  $\varepsilon$ - $(T)$ -valid if all occurrences of choice functions in  $t$  are  $(T)$ -valid. We will sometimes omit  $T$  when the theory is understood from context.

We will focus on the theory  $T_{BV} = (\Sigma_{BV}, I_{BV})$  of fixed-size bit-vectors as defined by the SMT-LIB 2 standard [3]. The signature  $\Sigma_{BV}$  includes a unique sort for each positive bit-vector width  $n$ , denoted here as  $\sigma_{[n]}$ . Similarly,  $X_{[n]}$  is the set of *bit-vector variables* of sort  $\sigma_{[n]}$ , and  $X_{BV}$  is the union of all sets  $X_{[n]}$ . We assume that  $\Sigma_{BV}$  includes all *bit-vector constants* of sort  $\sigma_{[n]}$  for each  $n$ , represented as bit-strings. However, to simplify the notation we will sometimes denote them by the corresponding natural number in  $\{0, \dots, 2^{n-1}\}$ . All interpretations  $\mathcal{I} \in I_{BV}$  are identical except for the value they assign to variables. They interpret sort and function symbols as specified in SMT-LIB 2. All function symbols in  $\Sigma_{BV}^f$  are overloaded for every  $\sigma_{[n]} \in \Sigma_{BV}^s$ . We denote a  $\Sigma_{BV}$ -term (or *bit-vector term*)  $t$  of width  $n$  as  $t_{[n]}$  when we want to specify its bit-width explicitly. We use  $\max_{s[n]}$  or  $\min_{s[n]}$  for the *maximum* or *minimum signed value* of width  $n$ , e.g.,  $\max_{s[4]} = 0111$  and  $\min_{s[4]} = 1000$ . The width of a bit-vector sort or term is given by the function  $\kappa$ , e.g.,  $\kappa(\sigma_{[n]}) = n$  and  $\kappa(t_{[n]}) = n$ .

Without loss of generality, we consider a restricted set of bit-vector function symbols (or *bit-vector operators*)  $\Sigma_{BV}^f$  as listed in Table 1. The selection of operators in this set is arbitrary but complete in the sense that it suffices to express all bit-vector operators defined in SMT-LIB 2.

### 3 Invertibility Conditions for Bit-Vector Constraints

This section formally introduces the concept of an invertibility condition and shows that such conditions can be used to construct symbolic solutions for a class of quantifier-free bit-vector constraints that have a linear shape.

Consider a bit-vector literal  $x + s \approx t$  and assume that we want to solve for  $x$ . If the literal is *linear* in  $x$ , that is, has only one occurrence of  $x$ , a general solution for  $x$  is given by the inverse of bit-vector addition over equality:  $x = t - s$ . Computing the inverse of a bit-vector operation, however, is not always possible.

For example, for  $x \cdot s \approx t$ , an inverse always exists only if  $s$  always evaluates to an odd bit-vector. Otherwise, there are values for  $s$  and  $t$  where no such inverse exists, e.g.,  $x \cdot 2 \approx 3$ . However, even if there is no unconditional inverse for the general case, we can identify the condition under which a bit-vector operation is invertible. For the bit-vector multiplication constraint  $x \cdot s \approx t$  with  $x \notin FV(s) \cup FV(t)$ , the *invertibility condition* for  $x$  can be expressed by the formula  $(-s \mid s) \ \& \ t \approx t$ .

**Definition 1** (*Invertibility Condition*). Let  $\ell[x]$  be a  $\Sigma_{BV}$ -literal. A quantifier-free  $\Sigma_{BV}$ -formula  $\phi_c$  is an invertibility condition for  $x$  in  $\ell[x]$  if  $x \notin FV(\phi_c)$  and  $\phi_c \Leftrightarrow \exists x. \ell[x]$  is  $T_{BV}$ -valid.

An invertibility condition for a literal  $\ell[x]$  provides the *exact conditions* under which  $\ell[x]$  is solvable for  $x$ . We call it an “invertibility” condition because we can use Hilbert choice functions to express *all* such conditional solutions with a *single* symbolic term, that is, a term whose possible values are exactly the solutions for  $x$  in  $\ell[x]$ . Recall that a choice function  $\varepsilon y. \varphi[y]$  represents a solution for a formula  $\varphi[x]$  if there exists one, and represents an arbitrary value otherwise. We may use a choice function to describe inverse solutions for a literal  $\ell[x]$  with invertibility condition  $\phi_c$  as  $\varepsilon y. (\phi_c \Rightarrow \ell[y])$ . For example, for the general case of bit-vector multiplication over equality the choice function is defined as  $\varepsilon y. ((-s \mid s) \ \& \ t \approx t \Rightarrow y \cdot s \approx t)$ .

**Lemma 2.** If  $\phi_c$  is an invertibility condition for an  $\varepsilon$ -valid  $\Sigma_{BV}$ -literal  $\ell[x]$  and  $r$  is the term  $\varepsilon y. (\phi_c \Rightarrow \ell[y])$ , then  $r$  is  $\varepsilon$ -valid and  $\ell[r] \Leftrightarrow \exists x. \ell[x]$  is  $T_{BV}$ -valid.<sup>1</sup>

Intuitively, the lemma states that when  $\ell[x]$  is satisfiable (under condition  $\phi_c$ ), any value returned by the choice function  $\varepsilon y. (\phi_c \Rightarrow \ell[y])$  is a solution of  $\ell[x]$  (and thus  $\exists x. \ell[x]$  holds). Conversely, if there exists a value  $v$  for  $x$  that makes  $\ell[x]$  true, then there is a model of  $T_{BV}$  that interprets  $\varepsilon y. (\phi_c \Rightarrow \ell[y])$  as  $v$ .

Now, suppose that  $\Sigma_{BV}$ -literal  $\ell$  is again linear in  $x$  but that  $x$  occurs arbitrarily deep in  $\ell$ . Consider, for example, a literal  $s_1 \cdot (s_2 + x) \approx t$  where  $x$  does not occur in  $s_1$ ,  $s_2$  or  $t$ . We can solve this literal for  $x$  by recursively computing the (possibly conditional) inverses of all bit-vector operations that involve  $x$ . That is, first we solve  $s_1 \cdot x' \approx t$  for  $x'$ , where  $x'$  is a fresh variable abstracting  $s_2 + x$ , which yields the choice function  $x' = \varepsilon y. ((-s_1 \mid s_1) \ \& \ t \approx t \Rightarrow s_1 \cdot y \approx t)$ . Then, we solve  $s_2 + x \approx x'$  for  $x$ , which yields the solution  $x = x' - s_2 = \varepsilon y. ((-s_1 \mid s_1) \ \& \ t \approx t \Rightarrow s_1 \cdot y \approx t) - s_2$ .

Figure 1 describes in pseudo code the procedure to solve for  $x$  in an arbitrary literal  $\ell[x] = e[x] \bowtie t$  that is linear in  $x$ . We assume that  $e[x]$  is built over the set of bit-vector operators listed in Table 1. Function `solve` recursively constructs a symbolic solution by computing (conditional) inverses as follows. Let function `getInverse`( $x, \ell[x]$ ) return a term  $t'$  that is the inverse of  $x$  in  $\ell[x]$ , i.e., such that  $\ell[x] \Leftrightarrow x \approx t'$ . Furthermore, let function `getIC`( $x, \ell[x]$ ) return the invertibility condition  $\phi_c$  for  $x$  in  $\ell[x]$ . If  $e[x]$  has the form  $\diamond(e_1, \dots, e_n)$  with  $n > 0$ ,  $x$  must

<sup>1</sup> All proofs can be found in an extended version of this paper [19].

```

solve( $x, e[x] \bowtie t$ ):
  If  $e = x$ 
    If  $\bowtie \in \{\approx\}$  then return  $t$ 
    else return  $\varepsilon y. (\text{getIC}(x, x \bowtie t) \Rightarrow y \bowtie t)$ .
  else  $e = \diamond(e_1, \dots, e_i[x], \dots, e_n)$  with  $n > 0$  and  $x \notin FV(e_j)$  for all  $j \neq i$ .
    Let  $d[x'] = \diamond(e_1, \dots, e_{i-1}, x', e_{i+1}, \dots, e_n)$  where  $x'$  is a fresh variable.
    If  $\bowtie \in \{\approx, \not\approx\}$  and  $\diamond \in \{\sim, -, +\}$ 
      then let  $t' = \text{getInverse}(x', d[x'] \approx t)$  and return  $\text{solve}(x, e_i \bowtie t')$ 
    else let  $\phi_c = \text{getIC}(x', d[x'] \bowtie t)$  and return  $\text{solve}(x, e_i \approx \varepsilon y. (\phi_c \Rightarrow d[y] \bowtie t))$ .

```

**Fig. 1.** Function `solve` for constructing a symbolic solution for  $x$  given a linear literal  $e[x] \bowtie t$ .

occur in exactly one of the subterms  $e_1, \dots, e_n$  given that  $e$  is linear in  $x$ . Let  $d$  be the term obtained from  $e$  by replacing  $e_i$  (the subterm containing  $x$ ) with a fresh variable  $x'$ . We solve for subterm  $e_i[x]$  (treating it as a variable  $x'$ ) and compute an inverse `getInverse`( $x', d[x'] \approx t$ ), if it exists. Note that for a disequality  $e[x] \not\approx t$ , it suffices to compute the inverse over equality and propagate the disequality down. (For example, for  $e_i[x] + s \not\approx t$ , we compute the inverse  $t' = \text{getInverse}(x', x' + s \approx t) = t - s$  and recurse on  $e_i[x] \not\approx t'$ .) If no inverse for  $e[x] \bowtie t$  exists, we first determine the invertibility condition  $\phi_c$  for  $d[x']$  via `getIC`( $x', d[x'] \bowtie t$ ), construct the choice function  $\varepsilon y. (\phi_c \Rightarrow d[y] \bowtie t)$ , and set it equal to  $e_i[x]$ , before recursively solving for  $x$ . If  $e[x] = x$  and the given literal is an equality, we have reached the base case and return  $t$  as the solution for  $x$ . Note that in Fig. 1, for simplicity we omitted one case for which an inverse can be determined, namely  $x \cdot c \approx t$  where  $c$  is an odd constant.

**Theorem 3.** *Let  $\ell[x]$  be an  $\varepsilon$ -valid  $\Sigma_{BV}$ -literal linear in  $x$ , and let  $r = \text{solve}(x, \ell[x])$ . Then  $r$  is  $\varepsilon$ -valid,  $FV(r) \subseteq FV(\ell) \setminus \{x\}$  and  $\ell[r] \Leftrightarrow \exists x. \ell[x]$  is  $T_{BV}$ -valid.*

Tables 2 and 3 list the invertibility conditions for bit-vector operators  $\{\cdot, \text{mod}, \div, \&, |, \gg, \gg_a, \ll, \circ\}$  over relations  $\{\approx, \not\approx, <_u, >_u\}$ . Due to space restrictions we omit the conditions for signed inequalities since they can be expressed in terms of unsigned inequality. We omit the invertibility conditions over  $\{\leq_u, \geq_u\}$  since they can generally be constructed by combining the corresponding conditions for equality and inequality—although there might be more succinct equivalent conditions. Finally, we omit the invertibility conditions for operators  $\{\sim, -, +\}$  and literals  $x \bowtie t$  over inequality since they are basic bounds checks, e.g., for  $x <_s t$  we have  $t \not\approx \text{min}$ . The invertibility condition for  $x \not\approx t$  and for the extract operator is  $\top$ .<sup>2</sup>

<sup>2</sup> All the omitted invertibility conditions can be found in the extended version of this paper [19].

The idea of computing the inverse of bit-vector operators has been used successfully in a recent local search approach for solving quantifier-free bit-vector constraints by Niemetz et al. [17]. There, target values are propagated via inverse value computation. In contrast, our approach does not determine single inverse values based on concrete assignments but aims at finding symbolic solutions through the generation of conditional inverses. In an extended version of that work [18], the same authors present rules for inverse value computation over equality but they provide no proof of correctness for them. We define invertibility conditions not only over equality but also disequality and (un)signed inequality, and verify their correctness up to a certain bit-width.

### 3.1 Synthesizing Invertibility Conditions

We have defined invertibility conditions for all bit-vector operators in  $\Sigma_{BV}$  where no general inverse exists (162 in total). A noteworthy aspect of this work is that we were able to leverage syntax-guided synthesis (SyGuS) technology [1] to help identify these conditions. The problem of finding invertibility conditions for a literal of the form  $x \diamond s \bowtie t$  (or, dually,  $s \diamond x \bowtie t$ ) linear in  $x$  can be recast as a SyGuS problem by asking whether there exists a binary Boolean function  $C$  such that the (second-order) formula  $\exists C \forall s \forall t. ((\exists x. x \diamond s \bowtie t) \Leftrightarrow C(s, t))$  is satisfiable. If a SyGuS solver is able to synthesize the function  $C$ , then  $C$  can be used as the invertibility condition for  $x \diamond s \bowtie t$ . To simplify the SyGuS problem we chose a bit-width of 4 for  $x$ ,  $s$ , and  $t$  and eliminated the quantification over  $x$  in the formula above by expanding it to

$$\exists C \forall s \forall t. \left( \bigvee_{i=0}^{15} i \diamond s \bowtie t \right) \Leftrightarrow C(s, t)$$

Since the search space for SyGuS solvers heavily depends on the input grammar (which defines the solution space for  $C$ ), we decided to use two grammars with the same set of Boolean connectives but different sets of bit-vector operators:

$$O_r = \{\neg, \wedge, \approx, <_u, <_s, 0, \min_s, \max_s, s, t, \sim, -, \&, |\}$$

$$O_g = \{\neg, \wedge, \vee, \approx, <_u, <_s, \geq_u, \geq_s, 0, \min_s, \max_s, s, t, \sim, +, -, \&, |, \gg, \ll\}$$

The selection of constants in the grammar turned out to be crucial for finding solutions, e.g., by adding  $\min_s$  and  $\max_s$  we were able to synthesize substantially more invertibility conditions for signed inequalities. For each of the two sets of operators, we generated 140 SyGuS problems<sup>3</sup>, one for each combination of bit-vector operator  $\diamond \in \{\cdot, \text{mod}, \div, \&, |, \gg, \gg_a, \ll\}$  over relation  $\bowtie \in \{\approx, \not\approx, <_u, \leq_u, >_u, \geq_u, <_s, \leq_s, >_s, \geq_s\}$ , and used the SyGuS extension of the CVC4 solver [22] to solve these problems.

Using operators  $O_r$  ( $O_g$ ) we were able to synthesize 98 (116) out of 140 invertibility conditions, with 118 unique solutions overall. When we found more

<sup>3</sup> Available at <https://cvc4.cs.stanford.edu/papers/CAV2018-QBV/>.

**Table 2.** Conditions for the invertibility of bit-vector operators over (dis)equality. Those for  $\cdot$ ,  $\&$  and  $|$  are given modulo commutativity of those operators.

$\ell[x]$	$\approx$	$\not\approx$
$x \cdot s \bowtie t$	$(-s   s) \& t \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \bmod s \bowtie t$	$\sim(-s) \geq_u t$	$s \not\approx 1 \vee t \not\approx 0$
$s \bmod x \bowtie t$	$(t + t - s) \& s \geq_u t$	$s \not\approx 0 \vee t \not\approx 0$
$x \div s \bowtie t$	$(s \cdot t) \div s \approx t$	$s \not\approx 0 \vee t \not\approx \sim 0$
$s \div x \bowtie t$	$s \div (s \div t) \approx t$	$\begin{cases} s \& t \approx 0 & \text{for } \kappa(s) = 1 \\ \top & \text{otherwise} \end{cases}$
$x \& s \bowtie t$	$t \& s \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x   s \bowtie t$	$t   s \approx t$	$s \not\approx \sim 0 \vee t \not\approx \sim 0$
$x \gg s \bowtie t$	$(t \ll s) \gg s \approx t$	$t \not\approx 0 \vee s <_u \kappa(s)$
$s \gg x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \gg i \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \gg_a s \bowtie t$	$(s <_u \kappa(s) \Rightarrow (t \ll s) \gg_a s \approx t) \wedge$ $(s \geq_u \kappa(s) \Rightarrow (t \approx \sim 0 \vee t \approx 0))$	$\top$
$s \gg_a x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \gg_a i \approx t$	$(t \not\approx 0 \vee s \not\approx 0) \wedge$ $(t \not\approx \sim 0 \vee s \not\approx \sim 0)$
$x \ll s \bowtie t$	$(t \gg s) \ll s \approx t$	$t \not\approx 0 \vee s <_u \kappa(s)$
$s \ll x \bowtie t$	$\bigvee_{i=0}^{\kappa(s)} s \ll i \approx t$	$s \not\approx 0 \vee t \not\approx 0$
$x \circ s \bowtie t$	$s \approx t[\kappa(s) - 1 : 0]$	$\top$
$s \circ x \bowtie t$	$s \approx t[\kappa(t) - 1 : \kappa(s)]$	$\top$

than one solution for a condition (either with operators  $O_r$  and  $O_g$ , or manually) we chose the one that involved the smallest number of bit-vector operators. Thus, we ended up using 79 out of 118 synthesized conditions and 83 manually crafted conditions.

In some cases, the SyGuS approach was able to synthesize invertibility conditions that were smaller than those we had manually crafted. For example, we manually defined the invertibility condition for  $x \cdot s \approx t$  as  $(t \approx 0) \vee ((t \& -t) \geq_u (s \& -s) \wedge (s \not\approx 0))$ . With SyGuS we obtained  $((-s | s) \& t) \approx t$ . For some other cases, however, the synthesized solution involved more bit-vector operators than needed. For example, for  $x \bmod s \not\approx t$  we manually defined the invertibility condition  $(s \not\approx 1) \vee (t \not\approx 0)$ , whereas SyGuS produced the solution  $\sim(-s) | t \not\approx 0$ . For the majority of invertibility conditions, finding a solution did not require more than one hour of CPU time on an Intel Xeon E5-2637 with 3.5 GHz. Interestingly, the most time-consuming synthesis task (over 107h of CPU time) was finding condition  $((t + t) - s) \& s \geq_u t$  for  $s \bmod x \approx t$ . A small number of synthesized solutions were only correct for a bit-width of 4, e.g., solution  $(\sim s \ll s) \ll s <_s t$  for  $x \div s <_s t$ . In total, we found 6 width-dependent synthesized solutions, all of them for bit-vector operators  $\div$  and  $\bmod$ . For those, we used the manually crafted invertibility conditions instead.



**Table 3.** Conditions for the invertibility of bit-vector operators over unsigned inequality. Those for  $\cdot$ ,  $\&$  and  $|$  are given modulo commutativity of those operators.

$\ell[x]$	$<_u$	$>_u$
$x \cdot s \bowtie t$	$t \not\approx 0$	$t <_u -s \mid s$
$x \bmod s \bowtie t$	$t \not\approx 0$	$t <_u \sim(-s)$
$s \bmod x \bowtie t$	$t \not\approx 0$	$t <_u s$
$x \div s \bowtie t$	$0 <_u s \wedge 0 <_u t$	$\sim 0 \div s >_u t$
$s \div x \bowtie t$	$0 <_u \sim(-t \& s) \wedge 0 <_u t$	$t <_u \sim 0$
$x \& s \bowtie t$	$t \not\approx 0$	$t <_u s$
$x \mid s \bowtie t$	$s <_u t$	$t <_u \sim 0$
$x \gg s \bowtie t$	$t \not\approx 0$	$t <_u \sim s \gg s$
$s \gg x \bowtie t$	$t \not\approx 0$	$t <_u s$
$x \gg_a s \bowtie t$	$t \not\approx 0$	$t <_u \sim 0$
$s \gg_a x \bowtie t$	$(s <_u t \vee s \geq_s 0) \wedge t \not\approx 0$	$s <_s (s \gg \sim t) \vee t <_u s$
$x \ll s \bowtie t$	$t \not\approx 0$	$t <_u \sim 0 \ll s$
$s \ll x \bowtie t$	$t \not\approx 0$	$\bigvee_{i=0}^{\kappa(s)} (s \ll i) >_u t$
$x \circ s \bowtie t$	$t_x \approx 0 \Rightarrow s <_u t_s$ where $t_x = t[\kappa(t) - 1 : \kappa(x)]$ , $t_s = t[\kappa(s) - 1 : 0]$	$t_x \approx \sim 0 \Rightarrow s >_u t_s$
$s \circ x \bowtie t$	$s \leq_u t_s \wedge (s \approx t_s \Rightarrow t_x \not\approx 0)$ where $t_x = t[\kappa(x) - 1 : 0]$ , $t_s = t[\kappa(t) - 1 : \kappa(s)]$	$s \geq_u t_s \wedge s \approx t_s \Rightarrow t_x \not\approx \sim 0$

### 3.2 Verifying Invertibility Conditions

We verified the correctness of all 162 invertibility conditions for bit-widths from 1 to 65 by checking for each bit-width the  $T_{BV}$ -unsatisfiability of the formula  $\neg(\phi_c \Leftrightarrow \exists x. \ell[x])$  where  $\ell$  ranges over the literals in Tables 2 and 3 with  $s$  and  $t$  replaced by fresh constants, and  $\phi_c$  is the corresponding invertibility condition.

In total, we generated 12,980 verification problems and used all participating solvers of the quantified bit-vector division of SMT-competition 2017 to verify them. For each solver/benchmark pair we used a CPU time limit of one hour and a memory limit of 8 GB on the same machines as those mentioned in the previous section. We consider an invertibility condition to be verified for a certain bit-width if at least one of the solvers was able to report unsatisfiable for the corresponding formula within the given time limit. Out of the 12,980 instances, we were able to verify 12,277 (94.6%).

Overall, all verification tasks (including timeouts) required a total of 275 days of CPU time. The success rate of each individual solver was 91.4% for Boolector, 85.0% for CVC4, 50.8% for Q3B, and 92% for Z3. We observed that on 30.6% of the problems, Q3B exited with a Python exception without returning any result. For bit-vector operators  $\{\sim, -, +, \&, |, \gg, \gg_a, \ll, \circ\}$ , over all relations, and for operators  $\{\cdot, \div, \bmod\}$  over relations  $\{\not\approx, \leq_u, \leq_s\}$ , we were able to verify

all invertibility conditions for all bit-widths in the range 1–65. Interestingly, no solver was able to verify the invertibility conditions for  $x \bmod s <_s t$  with a bit-width of 54 and  $s \bmod x <_u t$  with bit-widths 35–37 within the allotted time. We attribute this to the underlying heuristics used by the SAT solvers in these systems. All other conditions for  $<_s$  and  $<_u$  were verified for all bit-vector operators up to bit-width 65. The remaining conditions for operators  $\{\cdot, \div, \bmod\}$  over relations  $\{\approx, >_u, \geq_u, >_s, \geq_s\}$  were verified up to at least a bit-width of 14. We discovered 3 conditions for  $s \div x \bowtie t$  with  $\bowtie \in \{\neq, >_s, \geq_s\}$  that were not correct for a bit-width of 1. For each of these cases, we added an additional invertibility condition that correctly handles that case.

We leave to future work the task of formally proving that our invertibility conditions are correct for all bit-widths. Since this will most likely require the development of an interactive proof, we could leverage some recent work by Ekici et al. [9] that includes a formalization in the Coq proof assistant of the SMT-LIB theory of bit-vectors.

## 4 Counterexample-Guided Instantiation for Bit-Vectors

In this section, we leverage techniques from the previous section for constructing symbolic solutions to bit-vector constraints to define a novel instantiation-based technique for quantified bit-vector formulas. We first briefly present the overall theory-independent procedure we use for quantifier instantiation and then show how it can be specialized to quantified bit-vectors using invertibility conditions.

We use a counterexample-guided approach for quantifier instantiation, as given by procedure  $\text{CEGQI}_{\mathcal{S}}$  in Fig. 2. To simplify the exposition here, we focus on input problems expressed as a single formula in prenex normal form and with up to one quantifier alternation. We stress, though, that the approach applies in general to arbitrary sets of quantified formulas in some  $\Sigma$ -theory  $T$  with a decidable quantifier-free fragment. The procedure checks via instantiation the  $T$ -satisfiability of a quantified input formula  $\varphi$  of the form  $\exists \mathbf{y} \forall \mathbf{x}. \psi[\mathbf{x}, \mathbf{y}]$  where  $\psi$  is quantifier-free and  $\mathbf{x}$  and  $\mathbf{y}$  are possibly empty sequences of variables. It maintains an evolving set  $\Gamma$ , initially empty, of quantifier-free instances of the input formula. During each iteration of the procedure’s loop, there are three possible cases: (1) if  $\Gamma$  is  $T$ -unsatisfiable, the input formula  $\varphi$  is also  $T$ -unsatisfiable and “unsat” is returned; (2) if  $\Gamma$  is  $T$ -satisfiable but not together with  $\neg\psi[\mathbf{y}, \mathbf{x}]$ , the negated body of  $\varphi$ , then  $\Gamma$  entails  $\varphi$  in  $T$ , hence  $\varphi$  is  $T$ -satisfiable and “sat” is returned. (3) If neither of previous cases holds, the procedure adds to  $\Gamma$  an instance of  $\psi$  obtained by replacing the variables  $\mathbf{x}$  with some terms  $\mathbf{t}$ , and continues. The procedure  $\text{CEGQI}$  is parametrized by a *selection function*  $\mathcal{S}$  that generates the terms  $\mathbf{t}$ .

**Definition 4** (*Selection Function*). A selection function takes as input a tuple of variables  $\mathbf{x}$ , a model  $\mathcal{I}$  of  $T$ , a quantifier-free  $\Sigma$ -formula  $\psi[\mathbf{x}]$ , and a set  $\Gamma$  of  $\Sigma$ -formulas such that  $\mathbf{x} \notin FV(\Gamma)$  and  $\mathcal{I} \models \Gamma \cup \{\neg\psi\}$ . It returns a tuple of  $\varepsilon$ -valid terms  $\mathbf{t}$  of the same type as  $\mathbf{x}$  such that  $FV(\mathbf{t}) \subseteq FV(\psi) \setminus \mathbf{x}$ .

CEGQI<sub>S</sub>( $\exists \mathbf{y} \forall \mathbf{x}. \psi[\mathbf{y}, \mathbf{x}]$ )  
 $\Gamma := \emptyset$   
Repeat:  
1. If  $\Gamma$  is  $T$ -unsatisfiable, then return “unsat”.  
2. Otherwise, if  $\Gamma' = \Gamma \cup \{-\psi[\mathbf{y}, \mathbf{x}]\}$  is  $T$ -unsatisfiable, then return “sat”.  
3. Otherwise, let  $\mathcal{I}$  be a model of  $T$  and  $\Gamma'$  and  $\mathbf{t} = \mathcal{S}(\mathbf{x}, \psi, \mathcal{I}, \Gamma)$ .  $\Gamma := \Gamma \cup \{\psi[\mathbf{y}, \mathbf{t}]\}$ .

**Fig. 2.** A counterexample-guided quantifier instantiation procedure CEGQI<sub>S</sub>, parameterized by a selection function  $\mathcal{S}$ , for determining the  $T$ -satisfiability of  $\exists \mathbf{y} \forall \mathbf{x}. \psi[\mathbf{y}, \mathbf{x}]$  with  $\psi$  quantifier-free and  $FV(\psi) = \mathbf{y} \cup \mathbf{x}$ .

**Definition 5.** Let  $\psi[\mathbf{x}]$  be a quantifier-free  $\Sigma$ -formula. A selection function is:

1. Finite for  $\mathbf{x}$  and  $\psi$  if there is a finite set  $\mathcal{S}^*$  such that  $\mathcal{S}(\mathbf{x}, \psi, \mathcal{I}, \Gamma) \in \mathcal{S}^*$  for all legal inputs  $\mathcal{I}$  and  $\Gamma$ .
2. Monotonic for  $\mathbf{x}$  and  $\psi$  if for all legal inputs  $\mathcal{I}$  and  $\Gamma$ ,  $\mathcal{S}(\mathbf{x}, \psi, \mathcal{I}, \Gamma) = \mathbf{t}$  only if  $\psi[\mathbf{t}] \notin \Gamma$ .

Procedure CEGQI<sub>S</sub> is refutation-sound and model-sound for any selection function  $\mathcal{S}$ , and terminating for selection functions that are finite and monotonic.

**Theorem 6 (Correctness of CEGQI<sub>S</sub>).** Let  $\mathcal{S}$  be a selection function and let  $\varphi = \exists \mathbf{y} \forall \mathbf{x}. \psi[\mathbf{y}, \mathbf{x}]$  be a legal input for CEGQI<sub>S</sub>. Then the following hold.

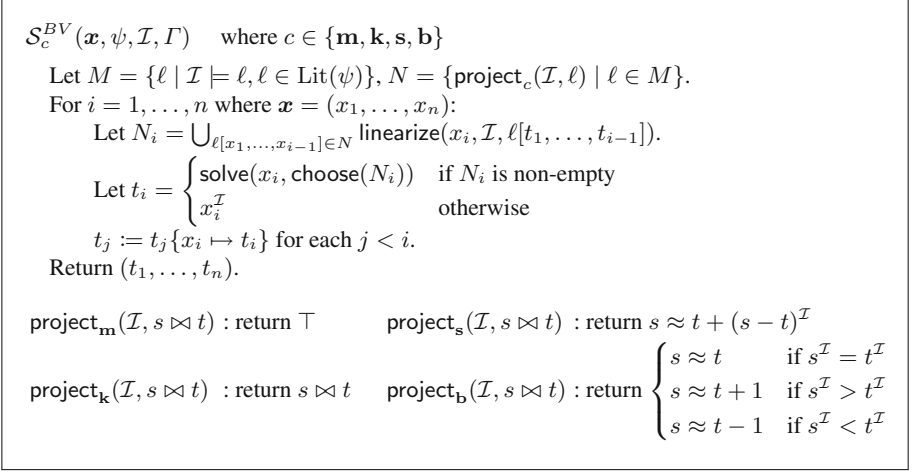
1. If CEGQI<sub>S</sub>( $\varphi$ ) returns “unsat”, then  $\varphi$  is  $T$ -unsatisfiable.
2. If CEGQI<sub>S</sub>( $\varphi$ ) returns “sat” for some final  $\Gamma$ , then  $\varphi$  is  $T$ -equivalent to  $\exists \mathbf{y}. \bigwedge_{\gamma \in \Gamma} \gamma$ .
3. If  $\mathcal{S}$  is finite and monotonic for  $\mathbf{x}$  and  $\psi$ , then CEGQI<sub>S</sub>( $\varphi$ ) terminates.

Thanks to this theorem, to define a  $T$ -satisfiability procedure for quantified  $\Sigma$ -formulas, it suffices to define a selection function satisfying the criteria of Definition 4. We do that in the following section for  $T_{BV}$ .

### 4.1 Selection Functions for Bit-Vectors

In Fig. 3, we define a (class of) selection functions  $\mathcal{S}_c^{BV}$  for quantifier-free bit-vector formulas, which is parameterized by a *configuration*  $c$ , a value of the enumeration type  $\{\mathbf{m}, \mathbf{k}, \mathbf{s}, \mathbf{b}\}$ . The selection function collects in the set  $M$  all the literals occurring in  $\Gamma'$  that are satisfied by  $\mathcal{I}$ . Then, it collects in the set  $N$  a *projected form* of each literal in  $M$ . This form is computed by the function `projectc` parameterized by configuration  $c$ . That function transforms its input literal into a form suitable for function `solve` from Fig. 1. We discuss the intuition for projection operations in more detail below.

After constructing set  $N$ , the selection function computes a term  $t_i$  for each variable  $x_i$  in tuple  $\mathbf{x}$ , which we call the *solved form* of  $x_i$ . To do that, it first



**Fig. 3.** Selection functions  $\mathcal{S}_c^{BV}$  for quantifier-free bit-vector formulas. The procedure is parameterized by a configuration  $c$ , one of either  $\mathbf{m}$  (model value),  $\mathbf{k}$  (keep),  $\mathbf{s}$  (slack), or  $\mathbf{b}$  (boundary).

constructs a set of literals  $N_i$  all linear in  $x_i$ . It considers literals  $\ell$  from  $N$  and replaces all previously solved variables  $x_1, \dots, x_{i-1}$  by their respective solved forms to obtain the literal  $\ell' = \ell[t_1, \dots, t_{i-1}]$ . It then calls function `linearize` on literal  $\ell'$  which returns a *set* of literals, each obtained by replacing all but one occurrence of  $x_i$  in  $\ell$  with the value of  $x_i$  in  $\mathcal{I}$ .<sup>4</sup>

*Example 7.* Consider an interpretation  $\mathcal{I}$  where  $x^{\mathcal{I}} = 1$ , and  $\Sigma_{BV}$ -terms  $a$  and  $b$  with  $x \notin FV(a) \cup FV(b)$ . We have that `linearize`( $x, \mathcal{I}, x \cdot (x + a) \approx b$ ) returns the set  $\{1 \cdot (x + a) \approx b, x \cdot (1 + a) \approx b\}$ ; `linearize`( $x, \mathcal{I}, x \geq_u a$ ) returns the singleton set  $\{x \geq_u a\}$ ; `linearize`( $x, \mathcal{I}, a \not\approx b$ ) returns the empty set.  $\triangle$

If the set  $N_i$  is non-empty, the selection function heuristically chooses a literal from  $N_i$  (indicated in Fig. 3 with `choose`( $N_i$ )). It then computes a solved form  $t_i$  for  $x_i$  by solving the chosen literal for  $x_i$  with the function `solve` described in the previous section. If  $N_i$  is empty, we let  $t_i$  is simply the value of  $x_i$  in the given model  $\mathcal{I}$ . After that,  $x_i$  is eliminated from all the previous terms  $t_1, \dots, t_{i-1}$  by replacing it with  $t_i$ . After processing all  $n$  variables of  $\mathbf{x}$ , the tuple  $(t_1, \dots, t_n)$  is returned.

The configurations of selection function  $\mathcal{S}_c^{BV}$  determine how literals in  $M$  are modified by the `project` <sub>$c$</sub>  function prior to computing solved forms, based on the current model  $\mathcal{I}$ . With the *model value* configuration  $\mathbf{m}$ , the selection function effectively ignores the structure of all literals in  $M$  and (because the set  $N_i$  is empty) ends up choosing the value  $x_i^{\mathcal{I}}$  as the solved form variable

<sup>4</sup> This is a simple heuristic to generate literals that can be solved for  $x_i$ . More elaborate heuristics could be used in practice.

$x_i$ , for each  $i$ . On the other end of the spectrum, the configuration  $\mathbf{k}$  keeps all literals in  $M$  unchanged. The remaining two configurations have an effect on how disequalities and inequalities are handled by `projectc`. With configuration  $\mathbf{s}$  `projectc` normalizes any kind of literal (equality, inequality or disequality)  $s \bowtie t$  to an equality by adding the *slack* value  $(s - t)^{\mathcal{I}}$  to  $t$ . With configuration  $\mathbf{b}$  it maps equalities to themselves and inequalities and disequalities to an equality corresponding to a *boundary point* of the relation between  $s$  and  $t$  based on the current model. Specifically, it adds one to  $t$  if  $s$  is greater than  $t$  in  $\mathcal{I}$ , it subtracts one if  $s$  is smaller than  $t$ , and returns  $s \approx t$  if their value is the same. These two configurations are inspired by quantifier elimination techniques for linear arithmetic [5, 15]. In the following, we provide an end-to-end example of our technique for quantifier instantiation that makes use of selection function  $\mathcal{S}_c^{BV}$ .

*Example 8.* Consider formula  $\varphi = \exists \mathbf{y}. \forall x_1. (x_1 \cdot a \leq_u b)$  where  $a$  and  $b$  are terms with no free occurrences of  $x_1$ . To determine the satisfiability of  $\varphi$ , we invoke  $\text{CEGQI}_{\mathcal{S}_c^{BV}}$  on  $\varphi$  for some configuration  $c$ . Say that in the first iteration of the loop, we find that  $\Gamma' = \Gamma \cup \{x_1 \cdot a >_u b\}$  is satisfied by some model  $\mathcal{I}$  of  $T_{BV}$  such that  $x_1^{\mathcal{I}} = 1$ ,  $a^{\mathcal{I}} = 1$ , and  $b^{\mathcal{I}} = 0$ . We invoke  $\mathcal{S}_c^{BV}((x_1), \mathcal{I}, \Gamma')$  and first compute  $M = \{x_1 \cdot a >_u b\}$ , the set of literals of  $\Gamma'$  that are satisfied by  $\mathcal{I}$ . The table below summarizes the values of the internal variables of  $\mathcal{S}_c^{BV}$  for the various configurations:

Config	$N_1$	$t_1$
$\mathbf{m}$	$\emptyset$	1
$\mathbf{k}$	$\{x_1 \cdot a >_u b\}$	$\varepsilon z. (a <_u -b \mid b) \Rightarrow z \cdot a >_u b$
$\mathbf{s}, \mathbf{b}$	$\{x_1 \cdot a \approx b + 1\}$	$\varepsilon z. ((-a \mid a) \& b + 1 \approx b + 1) \Rightarrow z \cdot a \approx b + 1$

In each case,  $\mathcal{S}_c^{BV}$  returns the tuple  $(t_1)$ , and we add the instance  $t_1 \cdot a \leq_u b$  to  $\Gamma$ . Consider configuration  $\mathbf{k}$  where  $t_1$  is the choice expression  $\varepsilon z. ((a <_u -b \mid b) \Rightarrow z \cdot a >_u b)$ . Since  $t_1$  is  $\varepsilon$ -valid, due to the semantics of  $\varepsilon$ , this instance is equivalent to:

$$((a <_u -b \mid b) \Rightarrow k \cdot a >_u b) \wedge k \cdot a \leq_u b \tag{1}$$

for fresh variable  $k$ . This formula is  $T_{BV}$ -satisfiable if and only if  $\neg(a <_u -b \mid b)$  is  $T_{BV}$ -satisfiable. In the second iteration of the loop in  $\text{CEGQI}_{\mathcal{S}_c^{BV}}$ , set  $\Gamma$  contains formula (1) above. We have two possible outcomes:

- (i)  $\neg(a <_u -b \mid b)$  is  $T_{BV}$ -unsatisfiable. Then (1) and hence  $\Gamma$  are  $T_{BV}$ -unsatisfiable, and the procedure terminates with “unsat”.
- (ii)  $\neg(a <_u -b \mid b)$  is satisfied by some model  $\mathcal{J}$  of  $T_{BV}$ . Then  $\exists z. z \cdot a \leq_u b$  is false in  $\mathcal{J}$  since the invertibility condition of  $z \cdot a \leq_u b$  is false in  $\mathcal{J}$ . Hence,  $\Gamma' = \Gamma \cup \{x_1 \cdot a >_u b\}$  is unsatisfiable, and the algorithm terminates with “sat”.

In fact, we argue later that quantified bit-vector formulas like  $\varphi$  above, which contain only one occurrence of a universal variable, require at most one instantiation before  $\text{CEGQI}_{\mathcal{S}_k^{BV}}$  terminates. The same guarantee does not hold with the other configurations. In particular, configuration  $\mathbf{m}$  generates the instantiation where  $t_1$  is 1, which simplifies to  $a \leq_u b$ . This may not be sufficient to show that  $\Gamma$  or  $\Gamma'$  is unsatisfiable in the second iteration of the loop and the algorithm may resort to *enumerating* a repeating pattern of instantiations, such as  $x_1 \mapsto 1, 2, 3, \dots$  and so on. This obviously does not scale for problems with large bit-widths.  $\triangle$

More generally, we note that  $\text{CEGQI}_{\mathcal{S}_k^{BV}}$  terminates with at most one instance for input formulas whose body has just one literal and a single occurrence of each universal variable. The same guarantee does not hold for instance for quantified formulas whose body has multiple disjuncts. For some intuition, consider extending the second conjunct of (1) with an additional disjunct, i.e.  $(k \cdot a \leq_u b \vee \ell[k])$ . A model can be found for this formula in which the invertibility condition  $(a <_u -b \mid b)$  is still satisfied, and hence we are not guaranteed to terminate on the second iteration of the loop. Similarly, if the literals of the input formula have multiple occurrences of  $x_1$ , then multiple instances may be returned by the selection function since the literals returned by `linearize` in Fig. 3 depend on the model value of  $x_1$ , and hence more than one possible instance may be considered in loop in Fig. 2.

The following theorem summarizes the properties of our selection functions. In the following, we say a quantified formula is *unit linear invertible* if it is of the form  $\forall x. \ell[x]$  where  $\ell$  is linear in  $x$  and has an invertibility condition for  $x$ . We say a selection function is *n-finite* for a quantified formula  $\psi$  if the number of possible instantiations it returns is at most  $n$  for some positive integer  $n$ .

**Theorem 9.** *Let  $\psi[x]$  be a quantifier-free formula in the signature of  $T_{BV}$ .*

1.  $\mathcal{S}_c^{BV}$  is a finite selection function for  $\mathbf{x}$  and  $\psi$  for all  $c \in \{\mathbf{m}, \mathbf{k}, \mathbf{s}, \mathbf{b}\}$ .
2.  $\mathcal{S}_m^{BV}$  is monotonic.
3.  $\mathcal{S}_k^{BV}$  is 1-finite if  $\psi$  is unit linear invertible.
4.  $\mathcal{S}_k^{BV}$  is monotonic if  $\psi$  is unit linear invertible.

This theorem implies that counterexample-guided instantiation using configuration  $\mathcal{S}_m^{BV}$  is a decision procedure for quantified bit-vectors. However, in practice the worst-case number of instances considered by this configuration for a variable  $x_{[n]}$  is proportional to the number of its possible values ( $2^n$ ), which is practically infeasible for sufficiently large  $n$ . More interestingly, counterexample-guided instantiation using  $\mathcal{S}_k^{BV}$  is a decision procedure for quantified formulas that are unit linear invertible, and moreover has the guarantee that at most one instantiation is returned by this selection function. Hence, formulas in this fragment can be effectively reduced to quantifier-free bit-vector constraints in at most two iterations of the loop of procedure  $\text{CEGQI}_{\mathcal{S}}$  in Fig. 2.

## 4.2 Implementation

We implemented the new instantiation techniques described in this section as an extension of CVC4, which is a DPLL( $T$ )-based SMT solver [20] that supports quantifier-free bit-vector constraints, (arbitrarily nested) quantified formulas, and support for choice expressions. For the latter, all choice expressions  $\varepsilon x. \varphi[x]$  are eliminated from assertions by replacing them with a fresh variable  $k$  of the same type and adding  $\varphi[k]$  as a new assertion, which notice is sound since all choice expressions we consider are  $\varepsilon$ -valid. In the remainder of the paper, we will refer to our extension of the solver as **cegqi**. In the following, we discuss important implementation details of the extension.

*Handling Duplicate Instantiations.* The selection functions  $\mathcal{S}_s^{BV}$  and  $\mathcal{S}_b^{BV}$  are not guaranteed to be monotonic, neither is  $\mathcal{S}_k^{BV}$  for quantified formulas that contain more than one occurrence of universal variables. Hence, when applying these strategies to arbitrary quantified formulas, we use a two-tiered strategy that invokes  $\mathcal{S}_m^{BV}$  as a second resort if the instance for the terms returned by a selection function already exists in  $\Gamma$ .

*Linearizing Rewrites.* Our selection function in Fig. 3 uses the function `linearize` to compute literals that are linear in the variable  $x_i$  to solve for. The way we presently implement `linearize` makes those literals dependent on the value of  $x_i$  in the current model  $\mathcal{I}$ , with the risk of overfitting to that model. To address this limitation, we use a set of equivalence-preserving rewrite rules whose goal is to reduce the number of occurrences of  $x_i$  to one when possible, by applying basic algebraic manipulations. As a trivial example, a literal like  $x_i + x_i \approx a$  is rewritten first to  $2 \cdot x_i \approx a$  which is linear in  $x_i$  if  $a$  does not contain  $x_i$ . In that case, this literal, and so the original one, has an invertibility condition as discussed in Sect. 3.

*Variable Elimination.* We use procedure `solve` from Sect. 3 not only for selecting quantifier instantiations, but also for eliminating variables from quantified formulas. In particular, for a quantified formula of the form  $\forall x \mathbf{y}. \ell \Rightarrow \varphi[x, \mathbf{y}]$ , if  $\ell$  is linear in  $x$  and `solve`( $x, \ell$ ) returns a term  $s$  containing no  $\varepsilon$ -expressions, we can replace this formula by  $\forall \mathbf{y}. \varphi[s, \mathbf{y}]$ . When  $\ell$  is an equality, this is sometimes called destructive equality resolution (DER) and is an important implementation-level optimization in state-of-the-art bit-vector solvers [25]. As shown in Fig. 1, we use the `getInverse` function to increase the likelihood that `solve` returns a term that contains no  $\varepsilon$ -expressions.

*Handling Extract.* Consider formula  $\forall x_{[32]}. (x[31 : 16] \not\approx a_{[16]} \vee x[15 : 0] \not\approx b_{[16]})$ . Since all invertibility conditions for the `extract` operator are  $\top$ , rather than producing choice expressions we have found it more effective to eliminate extracts via rewriting. As a consequence, we independently solve constraints for *regions* of quantified variables when they appear underneath applications of `extract` operations. In this example, we let the solved form of  $x$  be  $y_{[16]} \circ z_{[16]}$  where  $y$  and  $z$  are fresh variables, and subsequently solve for these variables in  $y \approx a$  and  $z \approx b$ . Hence, we may instantiate  $x$  with  $a \circ b$ , a term that we would not have found by considering the two literals independently in the negated body of the formula above.

## 5 Evaluation

We implemented our techniques in the solver **cegqi** and considered four configurations **cegqi<sub>c</sub>**, where **c** is one of **{m, k, s, b}**, corresponding to the four selection function configurations described in Sect. 4. Out of these four configurations, **cegqi<sub>m</sub>** is the only one that does not employ our new techniques but uses only model values for instantiation. It can thus be considered our base configuration. All configurations enable the optimizations described in Sect. 4.2 when applicable. We compared them against all entrants of the quantified bit-vector division of the 2017 SMT competition SMT-COMP: Boolector [16], CVC4 [2], Q3B [14] and Z3 [6]. With the exception of Q3B, all solvers are related to our approach since they are instantiation-based. However, none of these solvers utilizes invertibility conditions when constructing instantiations. We ran all experiments on the StarExec logic solving service [24] with a 300s CPU and wall clock time limit and 100 GB memory limit.

We evaluated our approach on all 5,151 benchmarks from the quantified bit-vector logic (BV) of SMT-LIB [3]. The results are summarized in Table 4. Configuration **cegqi<sub>b</sub>** solves the highest number of unsatisfiable benchmarks (4,399), which is 30 more than the next best configuration **cegqi<sub>s</sub>** and 37 more than

**Table 4.** Results on satisfiable and unsatisfiable benchmarks with a 300s timeout.

unsat	Boolector	CVC4	Q3B	Z3	<b>cegqi<sub>m</sub></b>	<b>cegqi<sub>k</sub></b>	<b>cegqi<sub>s</sub></b>	<b>cegqi<sub>b</sub></b>
h-uauto	14	12	93	24	10	103	105	<b>106</b>
keymaera	3917	3790	3781	<b>3923</b>	3803	3798	3888	3918
psyco	<b>62</b>	<b>62</b>	49	<b>62</b>	<b>62</b>	39	<b>62</b>	61
scholl	57	36	13	<b>67</b>	36	27	36	35
tptp	55	52	<b>56</b>	<b>56</b>	<b>56</b>	<b>56</b>	<b>56</b>	<b>56</b>
uauto	<b>137</b>	72	131	<b>137</b>	72	72	135	<b>137</b>
ws-fixpoint	74	71	<b>75</b>	74	<b>75</b>	74	<b>75</b>	<b>75</b>
ws-ranking	16	8	18	<b>19</b>	15	11	12	11
<b>Total unsat</b>	4332	4103	4216	4362	4129	4180	4369	<b>4399</b>
sat	Boolector	CVC4	Q3B	Z3	<b>cegqi<sub>m</sub></b>	<b>cegqi<sub>k</sub></b>	<b>cegqi<sub>s</sub></b>	<b>cegqi<sub>b</sub></b>
h-uauto	15	10	<b>17</b>	13	16	<b>17</b>	16	<b>17</b>
keymaera	<b>108</b>	21	24	<b>108</b>	20	13	36	75
psyco	131	<b>132</b>	50	131	<b>132</b>	60	<b>132</b>	129
scholl	<b>232</b>	160	201	204	203	188	208	211
tptp	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>
uauto	14	14	15	<b>16</b>	14	14	14	14
ws-fixpoint	45	49	<b>54</b>	36	45	51	49	50
ws-ranking	19	15	<b>37</b>	33	33	31	31	32
<b>Total sat</b>	<b>581</b>	418	415	558	480	391	503	545
<b>Total (5151)</b>	4913	4521	4631	4920	4609	4571	4872	<b>4944</b>



the next best external solver, Z3. Compared to the instantiation-based solvers Boolector, CVC4 and Z3, the performance of **cegqi<sub>b</sub>** is particularly strong on the h-uauto family, which are verification conditions from the Ultimate Automizer tool [11]. For satisfiable benchmarks, Boolector solves the most (581), which is 36 more than our best configuration **cegqi<sub>b</sub>**.

Overall, our best configuration **cegqi<sub>b</sub>** solved 335 more benchmarks than our base configuration **cegqi<sub>m</sub>**. A more detailed runtime comparison between the two is provided by the scatter plot in Fig. 4. Moreover, **cegqi<sub>b</sub>** solved 24 more benchmarks than the best external solver, Z3. In terms of uniquely solved instances, **cegqi<sub>b</sub>** was able to solve 139 benchmarks that were not solved by Z3, whereas Z3 solved 115 benchmarks that **cegqi<sub>b</sub>** did not. Overall, **cegqi<sub>b</sub>** was able to solve 21 of the 79 benchmarks (26.6%) not solved by any of the other solvers. For 18 of these 21 benchmarks, it terminated after considering no more than 4 instantiations. These cases indicate that using symbolic terms for instantiation solves problems for which other techniques, such as those that enumerate instantiations based on model values, do not scale.

Interestingly, configuration **cegqi<sub>k</sub>**, despite having the strong guarantees given by Theorem 9, performed relatively poorly on this set (with 4,571 solved instances overall). We attribute this to the fact that most of the quantified formulas in this set are not unit linear invertible. In total, we found that only 25.6% of the formulas considered during solving were unit linear invertible. However, only a handful of benchmarks were such that *all* quantified formulas in the problem were unit linear invertible. This might explain the superior performance of **cegqi<sub>s</sub>** and **cegqi<sub>b</sub>** which use invertibility conditions but in a less monolithic way.

For some intuition on this, consider the problem  $\forall x.(x > a \vee x < b)$  where  $a$  and  $b$  are such that  $a > b$  is  $T_{BV}$ -valid. Intuitively, to show that this formula is unsatisfiable requires the solver to find an  $x$  between  $b$  and  $a$ . This is apparent when considering the dual problem  $\exists x.(x \leq a \wedge x \geq b)$ . Configuration **cegqi<sub>b</sub>** is capable of finding such an  $x$ , for instance, by considering the instantiation  $x \mapsto a$  when solving for the boundary point of the first disjunct. Configuration **cegqi<sub>k</sub>**, on the other hand, would instead consider the instantiation of  $x$  for two terms that witness  $\varepsilon$ -expressions: some  $k_1$  that is never smaller than  $a$ , and some  $k_2$  that is never greater than  $b$ . Neither of these terms necessarily resides in between  $a$  and  $b$  since the solver may subsequently consider models where  $k_1 > b$  and  $k_2 < a$ . This points to a potential use for invertibility conditions that solve multiple literals simultaneously, something we are currently investigating.

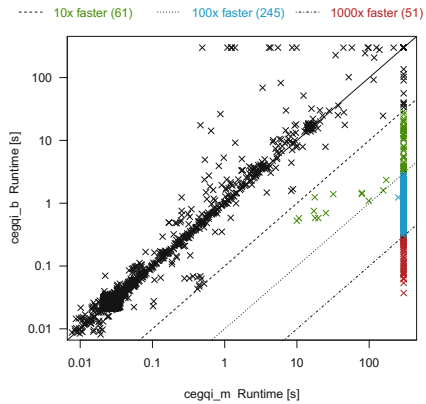


Fig. 4. Configuration **cegqi<sub>m</sub>** vs. **cegqi<sub>b</sub>**.

## 6 Conclusion

We have presented a new class of strategies for solving quantified bit-vector formulas based on invertibility conditions. We have derived invertibility conditions for the majority of operators in a standard theory of fixed-width bit-vectors. An implementation based on this approach solves over 25% of previously unsolved verification benchmarks from SMT LIB, and outperforms all other state-of-the-art bit-vector solvers overall.

In future work, we plan to develop a framework in which the correctness of invertibility conditions can be formally established independently of bit-width. We are working on deriving invertibility conditions that are optimal for linear constraints, in the sense of admitting the simplest propositional encoding. We also are investigating conditions that cover additional bit-vector operators, some cases of non-linear literals, as well as those that cover multiple constraints. While this is a challenging task, we believe efficient syntax-guided synthesis solvers can continue to help push progress in this direction. Finally, we plan to investigate the use of invertibility conditions for performing quantifier elimination on bit-vector constraints. This will require a procedure for deriving concrete witnesses from choice expressions.

## References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghthaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 1–8 (2013)
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_14](https://doi.org/10.1007/978-3-642-22110-1_14). <http://dl.acm.org/citation.cfm?id=2032305.2032319>
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)
4. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, 24–28 November 2015, pp. 15–27 (2015)
5. Cooper, D.C.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 91–100. Edinburgh University Press (1972)
6. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24). <http://dl.acm.org/citation.cfm?id=1792734.1792766>
7. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_49](https://doi.org/10.1007/978-3-319-08867-9_49)

8. Dutertre, B.: Solving exists/forall problems in yices. In: Workshop on Satisfiability Modulo Theories (2015)
9. Ekici, B., et al.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63390-9\\_7](https://doi.org/10.1007/978-3-319-63390-9_7)
10. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02658-4\\_25](https://doi.org/10.1007/978-3-642-02658-4_25)
11. Heizmann, M., et al.: Ultimate automizer with an on-demand construction of Floyd-Hoare automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 394–398. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54580-5\\_30](https://doi.org/10.1007/978-3-662-54580-5_30)
12. Hilbert, D., Bernays, P.: Grundlagen der Mathematik. Die Grundlehren der mathematischen Wissenschaften. Verlag von Julius Springer, Berlin (1934)
13. John, A.K., Chakraborty, S.: A layered algorithm for quantifier elimination from linear modular constraints. *Formal Methods Syst. Des.* **49**(3), 272–323 (2016). <https://doi.org/10.1007/s10703-016-0260-9>
14. Jonáš, M., Strejček, J.: Solving quantified bit-vector formulas using binary decision diagrams. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 267–283. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-40970-2\\_17](https://doi.org/10.1007/978-3-319-40970-2_17)
15. Loos, R., Weispfenning, V.: Applying linear quantifier elimination (1993)
16. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. *J. Satisfiability Boolean Model. Comput.* **9**, 53–58 (2014). (published 2015)
17. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 199–217. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41528-4\\_11](https://doi.org/10.1007/978-3-319-41528-4_11)
18. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. *Formal Methods Syst. Des.* **51**(3), 608–636 (2017). <https://doi.org/10.1007/s10703-017-0295-6>
19. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: On solving quantified bit-vectors using invertibility conditions. eprint [arXiv:cs.LO/1804.05025](https://arxiv.org/abs/1804.05025) (2018)
20. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. ACM* **53**(6), 937–977 (2006)
21. Preiner, M., Niemetz, A., Biere, A.: 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, 22–29 April 2017, Proceedings, Part I, pp. 264–280 (2017)
22. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-21668-3\\_12](https://doi.org/10.1007/978-3-319-21668-3_12)
23. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods Syst. Des.* **51**(3), 500–532 (2017). <https://doi.org/10.1007/s10703-017-0290-y>

24. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08587-6\\_28](https://doi.org/10.1007/978-3-319-08587-6_28)
25. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. *Formal Methods Syst. Des.* **42**(1), 3–23 (2013)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

