

# Ternary Propagation-Based Local Search for more Bit-Precise Reasoning

Aina Niemetz    
Stanford University

Mathias Preiner    
Stanford University

**Abstract**—Current state of the art for reasoning about quantifier-free bit-vector constraints in Satisfiability Modulo Theories (SMT) is a technique called bit-blasting, an eager translation into propositional logic (SAT). While efficient in practice, it may not scale for large bit-widths when the input size cannot be sufficiently reduced with preprocessing techniques. A recent propagation-based local search procedure was shown to be effective on hard satisfiable instances, in particular in combination with bit-blasting in a sequential portfolio setting. However, a major weakness of this approach is its obliviousness to bits that can be simplified to constant values. In this paper, we generalize propagation-based local search with respect to such constant bits to ternary values. We further extend the procedure to handle more bit-vector operators, and introduce heuristics for more precise inverse value computation via bound tightening for inequality constraints. We provide an extensive experimental evaluation and show that the presented techniques yield a considerable improvement in performance.

## I. INTRODUCTION

Satisfiability Modulo Theories (SMT) solvers for the theory of fixed-width bit-vectors provide bit-precise reasoning for many applications in hardware and software verification. In particular the quantifier-free fragment of this theory has received a lot of interest in recent years, as witnessed by the high and increasing number of participants in the corresponding divisions of the annual SMT competition [35]. Current state of the art for solving quantifier-free bit-vector formulas in SMT is a technique called bit-blasting, where the input formula is first simplified and then eagerly translated into propositional logic (SAT). While efficient in practice, it does not necessarily scale for large bit-widths, in particular if the size of the input cannot be sufficiently reduced during preprocessing.

In [24], we attacked the problem from a different angle and proposed a complete propagation-based local search procedure for quantifier-free bit-vector formulas. It is based on propagating target values from the outputs to the inputs, does not require bit-blasting, brute-force randomization or restarts, and lifts the concept of *backtracing* of Automatic Test Pattern Generation (ATPG) [19] to the word-level. Even though it only allows to determine satisfiability (as expected for local search), it is particularly effective in a sequential portfolio [36] combination with bit-blasting. One of its main weaknesses, however, is its obliviousness to bits that can be simplified to constant values [22]. For example, consider

This work was supported in part by DARPA (award no. FA8650-18-2-7861) and ONR (award no. N68335-17-C-0558).

a formula  $(1110 \& x) \not\approx 0000$ , where the left operand of the *bitwise and* ( $\&$ ) operation forces its least significant bit (LSB) to constant 0. The procedure in [24] is oblivious to this information and may select invalid target values for  $(1110 \& x)$  where the LSB is set to 1. Propagating such values that are invalid due to constant bits and can therefore never be assumed may introduce significant overhead.

In this paper, we generalize the propagation-based local search approach presented in [24] with respect to constant bits to ternary values. We extract constant bit information from the bit-level circuit representation of the input formula, use ternary bit-vectors to represent this information and propagate target values with respect to these constant bits. This allows us to propagate more precise target values since we can guarantee that we only propagate target values that can actually be assumed. We show in our experiments that this considerably reduces redundant work and improves performance.

Down-propagating values as in [24] utilizes inverse value (and its less restrictive variant, consistent value) computation. Computing inverse values is, however, not always possible. For example, finding an inverse value for  $x$  in multiplication  $x \cdot s$  such that it produces value  $t$  given value  $s$ , i.e.,  $x \cdot s \approx t$ , is only possible if the value of  $t$  has at least as many right-most zeroes in its binary representation as the value of  $s$ , i.e., if the *invertibility condition*  $(s \mid -s) \& t \approx t$  is true. A consistent value for  $x$ , on the other hand, is any value that produces  $t$  disregarding value  $s$ , i.e., there exists a value  $v$  such that  $x \cdot v \approx t$ . Finding consistent values for  $x$  is in general always possible. When considering constant bits in  $x$ , however, inverse and consistent value computation is further restricted, and the latter becomes conditional. In [24] we defined invertibility conditions without considering constant bits in  $x$  in pseudocode, which we then formalized and verified in [26]. In this paper, we provide and verify invertibility conditions and *consistency conditions* with respect to constant bits in the operand to solve for. We further extend the set of natively supported bit-vector operators, and introduce heuristics for more precise inverse value computation via bound tightening for inequality constraints. To summarize, this paper makes the following *contributions*.

- We introduce the notion of *consistency condition*. We further derive and present invertibility conditions and consistency conditions with respect to constant bits for a representative set of bit-vector operators that allows us to model all bit-vector operators defined in SMT-LIB [4].



This article is licensed under a Creative Commons Attribution 4.0 International License

- We verify the correctness of all presented conditions up to a certain bit-width.
- We present a (probabilistically approximately) complete [17] generalization of the propagation-based local search procedure in [24] with respect to constant bits.
- We extend the set of bit-vector operators from [24] with bit-wise xor, signed less than, sign extension and arithmetic right shift, and provide invertibility and consistency conditions modulo constant bits for all of them.
- We introduce two heuristics for inequality predicates that allow us to infer more precise inverse values based on tightening bounds with respect to its operands and satisfied top-level inequalities.

*Related Work.* In previous years, a new generation of SAT solvers implementing Stochastic Local Search (SLS) achieved remarkable results in SAT competitions [2, 3, 6]. Hybrid combinations of SLS and CDCL [30] SAT procedures aim to get more than the best out of both worlds by tightly integrating SLS strategies into the CDCL approach, with promising results in last year’s SAT race [8, 31, 33]. Attempts to utilize SLS techniques in SMT by integrating an SLS SAT solver into the DPLL(T)-framework of the SMT solver MathSAT [12], on the other hand, were not able to compete with bit-blasting [16]. In [15], Fröhlich et al. lifted stochastic local search (SLS) from the bit-level to the word-level without bit-blasting, with promising results. Their approach, however, does not fully exploit the word-level structure but rather simulates bit-level local search by focusing on single bit flips. In [25], we proposed a propagation-based extension of [15], which introduced an additional strategy to propagate assignments from the outputs to the inputs. Our propagation-based local search approach in [24] expands on this idea and does not employ any SLS strategies. Invertibility conditions have been formalized, verified and utilized for quantified bit-vector formulas to generate symbolic instantiations in [26]. Recently, in [10] the concept of invertibility conditions has been lifted to the theory of floating-points by means of Syntax-Guided Synthesis (SyGuS) [1].

## II. PRELIMINARIES

We assume the usual notions and terminology of many-sorted first-order logic with equality (denoted by  $\approx$ ) (see, e.g., [14, 20]). We will focus on the quantifier-free fragment of the theory of fixed-size bit-vectors  $T_{BV} = (\Sigma_{BV}, I_{BV})$  as defined by the SMT-LIB 2 standard [4]. The signature  $\Sigma_{BV}$  includes a unique sort  $\sigma_{[w]}$  for each bit-width  $w$ , function symbols overloaded for every  $\sigma_{[w]}$ , all *bit-vector constants* of sort  $\sigma_{[w]}$  for each  $w$ , and a sort Bool and the Boolean constants  $\top$  (true) and  $\perp$  (false). We further assume that  $\Sigma_{BV}$  includes the Boolean operators  $\neg$  (not) and  $\wedge$  (and). Without loss of generality, we will interpret Boolean expressions as bit-vector expressions of size one. The non-empty class of  $\Sigma_{BV}$ -interpretations  $I_{BV}$  (the *models* of  $T_{BV}$ ) interpret sort and functions symbols as specified in SMT-LIB 2.

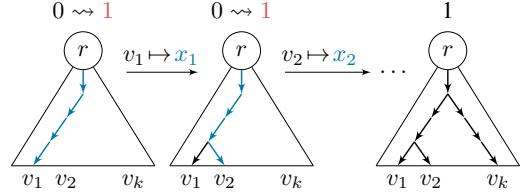


Fig. 1: Basic idea of propagation-based local search.

We denote a  $\Sigma_{BV}$ -term (or *bit-vector term*)  $a$  of width  $w$  as  $a_{[w]}$  when we want to specify its bit-width explicitly. The width of a bit-vector sort or term is given by the function  $\kappa$ , e.g.,  $\kappa(\sigma_{[w]}) = w$  and  $\kappa(t_{[w]}) = w$ . We will omit the bit-width from the notation when it is clear from the context.

We represent a bit-vector constant  $c_{[w]}$  as a bit-string of 0s and 1s, with the most significant bit (MSB) as the left-most bit  $c[msb]$  at index  $msb = w - 1$ , and the least significant bit (LSB) as the right-most bit  $c[lsb]$  at index  $lsb = 0$ . We use  $smax_{[w]}$  or  $smin_{[w]}$  for the *maximum* or *minimum signed value* of width  $w$ , e.g.,  $smax_{[4]} = 0111$  and  $smin_{[4]} = 1000$ , and  $ones_{[w]}$  for the maximum unsigned value, e.g.,  $ones_{[4]} = 1111$ . We refer to the bit at index  $i$  of a bit-vector  $t$  as  $t[i]$  and use  $ctz(t)$  to denote the *count of trailing zeros* of a bit-vector  $t$ . Similarly,  $clz(t)$  and  $clo(t)$  denote the count of *leading zeros* and *leading ones* in  $t$ . When interpreting  $t$  as signed value, we use  $cnt(t)$  to denote  $clo(t)$  when  $t[msb] \approx 1$ , and  $clz(t)$  when  $t[msb] \approx 0$ . We further use function  $min$  to determine the unsigned minimum value of two bit-vectors, and functions  $addo$  and  $mulo$ , which return true if the addition and multiplication of two bit-vectors overflows, respectively.

Without loss of generality, for a given input formula we consider a restricted set of bit-vector function symbols (or *bit-vector operators*) as listed in Table I. The selection in this set is arbitrary but complete in the sense that it suffices to express all bit-vector operators defined in SMT-LIB 2. This means that our approach is not restricted to this particular set of operators and can be lifted to any other set of bit-vector operators.

Note that we extend the set of operators considered in [24] with  $<_s$ ,  $\oplus$ ,  $\gg_a$  and sign extension. Further, we sometimes use the logical connectives  $\vee$  (or),  $\Rightarrow$  (implication) and  $\Leftrightarrow$  (if and only if), and the bit-vector operators  $\neq$  (disequality),  $|$  (bit-wise or) and (un)signed inequalities  $\leq_u$ ,  $\geq_u$ ,  $\leq_s$  and  $\geq_s$  as shorthand when convenient.

## III. PROPAGATION-BASED LOCAL SEARCH

The basic idea of the propagation-based local search procedure for quantifier-free bit-vector constraints as presented in [24] is illustrated in Figure 1 and lifts the concept of backtracking from ATPG [19] from the bit-level to the word-level. The procedure iteratively moves from a non-satisfying to a satisfying assignment by propagating target values from the outputs towards the inputs as follows.

Given a quantifier-free bit-vector formula as a directed acyclic graph (DAG) with a single root node  $r$ . We start from a random initial assignment to the inputs that does

not satisfy  $r$ , i.e., root  $r$  evaluates to 0, and assume that its target value is 1 (this desired transition from actual to target value is indicated in Figure 1 by  $0 \rightsquigarrow 1$ ). Starting from the root, this target value is then propagated along a single path towards an input, and this process is repeated until the root is satisfied, i.e., a solution is found. Down-propagation of target values is performed as a down-traversal where each traversal step represents a *propagation step*. In each propagation step, we first select the *propagation path*, i.e., the operand of the current node (representing a bit-vector operation) for which we want to compute the next target value. After selecting the propagation path, we then select the *propagation value* (the new target value) for the selected operand.

Propagation path and value selection are the two main sources of *non-determinism* of this procedure. Paths are non-deterministically selected with a preference to *essential inputs*. The concept of essential inputs was introduced in [24] to lift the notion of *controlling* inputs from the bit-level to the word-level, e.g., for  $00 \cdot 01 \approx t$  with target value  $t = 10$ , the left operand of the multiplication is essential since  $t$  can not be assumed without changing its value. Target values are determined via *inverse* and (its less strict variant) *consistent value computation*, and non-deterministically chosen if multiple possible values exist. An *inverse value* for an operand  $x$  allows to immediately produce a given target value assuming that the current value of the other operand (if any) does not change, e.g., for  $x \cdot 10 \approx t$  with target value  $t = 10$ , both 01 and 11 are inverse values for  $x$ . A *consistent value* for  $x$ , on the other hand, allows to produce a given target value after changing the value of the other operand (if necessary), e.g., for  $x \cdot 00 \approx t$  with target value  $t = 10$ , any value greater than zero is a consistent value for  $x$ . Notice that for every bit-vector literal  $x \diamond s \approx t$  (with  $\diamond$  a bit-vector operator as listed in Table I), any inverse value is also a consistent value for  $x$ .

When down-propagating values, inverse value computation can only be applied if such an inverse value exists. Computing a consistent value, on the other hand, is always possible (if constant bits are not considered, as in [24]). Inverse value computation is usually preferred over consistent value computation if possible. If no inverse value exists, the procedure falls back to consistent value computation. However, even if an inverse value exists, it is necessary to non-deterministically select between inverse and consistent value computation in order to guarantee completeness, as shown in [24].

Propagation-based local search as in [24] is not able to determine unsatisfiability, as expected for local search. However, when determining satisfiability it is probabilistically approximately complete (PAC) [17], i.e., it is guaranteed to (eventually) find a solution if there exists one.

#### IV. PROPAGATING CONSTANT BITS

In this section, we generalize the propagation-based local search procedure as presented in [24] with respect to constant bits to ternary values. Figure 2 describes the generalized algorithm in pseudocode, with all parts of the original algorithm (as given in [24]) that are affected by the generalization indicated

Symbol	SMT-LIB Syntax	Sort
$\approx, <_u, <_s$	=, bvult, bvslt	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \text{Bool}$
$\sim$	bvnot	$\sigma_{[w]} \rightarrow \sigma_{[w]}$
$\&, \oplus$	bvand, bvxor	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \sigma_{[w]}$
$\ll, \gg, \gg_a$	bvshl, bvlshr, bvashr	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \sigma_{[w]}$
$+, \cdot$	bvadd, bvmul	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \sigma_{[w]}$
$\text{mod}, \div$	bvurem, bvdiv	$\sigma_{[w]} \times \sigma_{[w]} \rightarrow \sigma_{[w]}$
$\circ$	concat	$\sigma_{[w]} \times \sigma_{[m]} \rightarrow \sigma_{[w+m]}$
$\langle m \rangle$	sign_extend	$\sigma_{[w]} \rightarrow \sigma_{[m+w]}$
$[u : l]$	extract ( $l \leq u < w$ )	$\sigma_{[w]} \rightarrow \sigma_{[u-l+1]}$

TABLE I: Set of considered bit-vector operators.

```

function sat( $r, \mathcal{A}_t, \mathcal{A}_b$ ):
1   while  $\mathcal{A}_b(r) \not\approx 1$ :
2      $n = r, t = 1$ 
3     while  $\neg \text{isLeaf}(n)$ :
4        $n_x = \text{select}(n, t, \mathcal{A}_b)$ 
5       if  $\neg \text{isConsistent}(n, n_x, t, \mathcal{A}_t)$ :
6         break // conflict
7          $v = \text{value}(n, n_x, t, \mathcal{A}_t, \mathcal{A}_b)$ 
8          $t = v, n = n_x$ 
9       if  $\neg \text{const}(n)$ :
10         $\mathcal{A}_b = \mathcal{A}_b\{n \mapsto t\}$ 
11   return true

```

Fig. 2: The propagation-based local search algorithm generalized with respect to constant bits to ternary values. Parts affected and lifted are indicated in blue.

in blue (with boxed line numbers). In the following, we first introduce notation, and then describe how to lift all relevant parts of the procedure to ternary values.

Without loss of generality and as in [24], we assume that a quantifier-free bit-vector formula  $\phi$  is represented as a single-rooted DAG with root  $r$  of bit-width one. Its set of nodes  $\mathcal{N}$  includes  $r$  and is partitioned into a set of *bit-vector operations* and a set of *leaf* nodes, the latter consisting of bit-vector constants and bit-vector variables (the *primary inputs*).

In the following, given a bit-vector literal  $x \diamond s \approx t$  or  $x \diamond s \approx t$  (with  $\diamond$  a bit-vector operator as listed in Table I), we will use  $t$  for the target value of the bit-vector operation,  $x$  to identify the operand we compute a value for, and  $s$  for the value of the other operand (if any).

We define a *binary bit-vector* as introduced above, and a *ternary bit-vector*  $x$  as a vector of three-valued bits where each bit can assume the values true (1), false (0) and undetermined ( $\bullet$ ). We either use a string representation or a range-based representation for  $x$ , where the latter is a pair of binary bit-vectors  $\langle x^{lo}, x^{hi} \rangle$  that determine the lower and upper bound of  $x$ , respectively. If  $x[i] = \bullet$ , then  $x^{lo}[i] = 0$  and  $x^{hi}[i] = 1$ , and  $x[i]$  otherwise. For example, a ternary bit-vector of size 4 with a true MSB and all other bits undetermined is represented as  $1\bullet\bullet\bullet$  when represented as a string, and as the pair of two binary bit-vectors  $\langle 1000, 1111 \rangle$  when represented as a bit-vector range. In the following, we only consider *valid* range

representations for ternary bit-vectors, i.e., pairs  $\langle x^{lo}, x^{hi} \rangle$  for which the validity check ( $\sim x^{lo} \mid x^{hi} \approx \text{ones}$ ) from [21] evaluates to true. For example, a range  $\langle 1100, 1000 \rangle$  is invalid since  $x^{lo}[2] >_u x^{hi}[2]$ , and thus  $\sim 1100 \mid 1000 \approx 1011 \not\approx 1111$ . We use function  $\text{valid}(x^{lo}, x^{hi})$  to check if  $x$  is valid.

In the following, we will use  $x$  for ternary bit-vectors, and  $s, t$  and  $v$  for binary bit-vectors. Further, to simplify notation, we will frequently use bit-vector literal patterns  $\diamond x \approx t$  and  $x \diamond s \approx t$  ( $s \diamond x \approx t$ ) for unary and binary literals, where we mix ternary and binary bit-vectors. We use  $x$  in these patterns as a placeholder, which represents constant bits in the operand we want to compute a binary bit-vector value for. Further, we sometimes give definitions only for one binary case (e.g.,  $x \diamond s \approx t$ ) when the other case is treated symmetrically.

We define assignment  $\mathcal{A}_b : \mathcal{N} \mapsto \{0, 1\}^+$  of formula  $\phi$  as a complete function that maps nodes  $n \in \mathcal{N}$  to *binary bit-vector* values. We use  $\mathcal{A}_b\{n \mapsto t\}$  to update node  $n$  to map to the new binary bit-vector value  $t$ , and assume that such an update propagates with respect to the semantics of the operators listed in Table I, e.g.,  $\mathcal{A}_b(n_x + n_s) = \mathcal{A}_b(n_x) + \mathcal{A}_b(n_s)$  for  $n_x, n_s \in \mathcal{N}$ . We further define assignment  $\mathcal{A}_t : \mathcal{N} \mapsto \{0, 1, \bullet\}^+$  as a complete function that represents constant bits and maps nodes  $n \in \mathcal{N}$  to *ternary bit-vectors*. Assignment  $\mathcal{A}_t$  is precomputed as described in Section IV-B.

**Definition 1** (Matching Constant Bits). *Given a ternary bit-vector  $x$  represented as a pair of binary bit-vectors  $\langle x^{lo}, x^{hi} \rangle$ . A binary bit-vector  $v$  matches the constant bits in  $x$  if and only if  $(x^{hi} \& v \approx v) \wedge (x^{lo} \mid v \approx v)$ .*

We use function  $mcb$  to check for matching constant bits, i.e.,  $mcb(x, v)$  (alternatively,  $mcb(x^{lo}, x^{hi}, v)$ ) is true if  $v$  matches the constant bits in  $x$ .

Given a bit-vector operation  $\diamond n_x$  or  $n_x \diamond n_s$  ( $n_s \diamond n_x$ ) with operand  $n_s \in \mathcal{N}$ , operand  $n_x \in \mathcal{N}$  the operand to solve for, and  $\diamond$  an operator as defined in Table I. As a first step, we lift the notion of *random*, *inverse* and *consistent values* from [24] to consider constant bits in  $n_x$  as follows.

**Definition 2** (Random Value). *A binary bit-vector  $v$  is a random value for a ternary bit-vector  $x = \mathcal{A}_t(n_x)$  if  $\kappa(v) \approx \kappa(x) \wedge mcb(x, v)$ .*

**Definition 3** (Consistent Value). *Given a bit-vector literal  $\diamond x \approx t$  or  $x \diamond s \approx t$ , with  $x = \mathcal{A}_t(n_x)$  a ternary bit-vector representing constant bits in  $n_x$ , and  $s = \mathcal{A}_b(n_s)$  and  $t$  binary bit-vectors. Given a target value  $t$ , a random value  $v$  is a consistent value for  $x$  if (there exists a binary bit-vector value  $s'$  such that)  $\diamond v \approx t$  or  $v \diamond s' \approx t$  evaluates to  $\top$ .*

**Definition 4** (Inverse Value). *Given a bit-vector literal and  $x$ ,  $s$  and  $t$  as above. Given a target value  $t$  (and a value  $s$ ), a consistent value  $v$  is called an inverse value for  $x$  if  $\diamond v \approx t$  or  $v \diamond s \approx t$  evaluates to  $\top$ .*

As an example, consider a ternary bit-vector  $x_{[2]} = 1\bullet$  with  $x^{lo} = 10$  and  $x^{hi} = 11$ . For  $x \cdot 11 \approx 01$ ,  $v = 11$  is an *inverse* value for  $x$ . For  $x \cdot 00 \approx 01$ , there exists no inverse value, but  $v = 11$  is a *consistent* value for  $x$  since  $11 \cdot s' \approx 01$  with

$s' = 11$ . A random value for  $x$  that is neither an inverse value nor a consistent value for both examples is  $v = 10$ .

Given a binary bit-vector operation  $n = n_x \diamond n_s$  with  $n \in \mathcal{N}$  and  $\diamond$  an operator as defined in Table I. We lift the notion of *essential input* of  $n$  from [24] to consider constant bits in its operands  $n_x, n_s \in \mathcal{N}$  as follows.

**Definition 5** (Essential Input). *Let  $n_x$  be an operand of a node  $n \in \mathcal{N}$  with  $n = n_x \diamond n_s$  and  $\mathcal{A}_b(n) = \mathcal{A}_b(n_x) \diamond \mathcal{A}_b(n_s)$ . Further, let  $t$  be the target value of  $n$ . We say that  $n_x$  is an essential input with respect to  $t$  if there exists no value  $v$  for  $n_s$  with respect to constant bits in  $n_s$  such that  $\mathcal{A}_b(n_x) \diamond v \approx t$ .*

For example, consider inequality  $n_x <_u n_s$  with target value  $t = 1$ ,  $\mathcal{A}_b(n_x) = 1011$ ,  $\mathcal{A}_b(n_s) = 1000$  and  $\mathcal{A}_t(n_s) = 100\bullet$ . When only considering the current assignment of  $n_x$  and  $n_s$  in  $\mathcal{A}_b$ , neither of the operands is essential—operand  $n_x$  would be if  $\mathcal{A}_b(n_x) = \text{ones}_{[4]}$ , and  $n_s$  if  $\mathcal{A}_b(n_s) = 0_{[4]}$ . However, considering constant bits in  $n_s$ , operand  $n_x$  is essential since  $n_s$  can not assume a value greater than  $\mathcal{A}_b(n_x)$ . More generally, for any  $n_x <_u n_s$  with  $\mathcal{A}_t(n_x) = \langle x^{lo}, x^{hi} \rangle$  and  $\mathcal{A}_t(n_s) = \langle s^{lo}, s^{hi} \rangle$ ,  $n_x$  is essential if  $\mathcal{A}_b(n_x) \approx \text{ones} \vee \mathcal{A}_b(n_x) \geq_u s^{hi}$ , and  $n_s$  is essential if  $\mathcal{A}_b(n_s) \approx 0 \vee x^{lo} \geq_u \mathcal{A}_b(n_s)$ .

As an interesting general observation, an operand  $n_x$  is *essential* if there exists *no inverse value* for the other operand  $n_s$  given  $\mathcal{A}_b(n_x)$ ,  $\mathcal{A}_t(n_s)$  and target value  $t$ . Operands to unary bit-vector operations are always essential.

**Algorithm Overview.** Function *sat* in Figure 2 determines the satisfiability of a given input formula, and takes as input root  $r$ , an initial assignment  $\mathcal{A}_b$ , and a precomputed fixed assignment  $\mathcal{A}_t$ . Assignment  $\mathcal{A}_b$  is updated in each iteration of the outer loop (lines 1–8), whereas  $\mathcal{A}_t$  is determined before function *sat* is called. Each iteration of the outer loop represents a *move*, i.e., the down propagation of target value  $t = 1$  for root  $r$  along a single path until a primary input is reached, which then triggers an update of assignment  $\mathcal{A}_b$  (lines 9–10) where input  $n$  is mapped to the new value  $t$ . Each iteration of the inner loop (lines 3–8) represents a single down propagation (a *propagation step*), which mainly consists of two phases: path selection (line 4) and value selection (line 7).

**Path Selection.** In each propagation step, on line 4, we first select the next leg of the propagation path as follows. For a bit-vector operation  $n$ , function *select* first determines which of its operands are essential with respect to target value  $t$  and constant bits in the other operand (if any). If all or none of the operands are essential, we non-deterministically select one of them. Else, the essential operand is selected.

To determine if an operand  $n_x$  is essential, we check if it is possible to find an inverse value for the other operand  $n_s$  given  $t$ ,  $\mathcal{A}_b(n_x)$  and  $\mathcal{A}_t(n_s)$ , i.e., we check if the corresponding invertibility condition when solved for  $n_s$  is false. For example, for  $n_x + n_s$  with  $t = 11$ ,  $\mathcal{A}_b(n_x) = 10$  and  $\mathcal{A}_t(n_s) = \bullet 0$ ,  $n_x$  is essential since  $mcb(\bullet 0, 11 - 10)$  is false.

**Value Selection.** After selecting the path, we then propagate  $t$  as target value for  $n$  to its selected operand  $n_x$  by computing an *inverse* or *consistent* value for  $n_x$ . An inverse value,

however, does not always exist, even when constant bits in the selected operand are not considered. And while it is always possible to find a consistent value when constant bits are not considered, this is not the case when they are. We therefore generalize the notion of *invertibility conditions* [24, 26] and introduce the new notion of *consistency conditions* to determine if there exists a value for  $n_x$  with respect to  $\mathcal{A}_t(n_x)$  such that target value  $t$  can be assumed. These conditions are utilized when selecting the propagation value as follows.

In each propagation step in Figure 2, before selecting a value with respect to constant bits for operand  $n_x$  (line 7), function *isConsistent* tests the corresponding *consistency condition* for bit-vector operation  $n$  and its operand  $n_x$  with respect to target value  $t$  and  $x = \mathcal{A}_t(n_x)$  (line 5). If this determines that no consistent value exists, the current target value for  $n$  can never be assumed (notice that every inverse value is also consistent) and we stop the current down propagation by breaking out of the inner loop to restart from the root (line 6). Note that this is in contrast to the original procedure, where it was *always* possible to find a consistent value for  $n_x$ .

If the consistency condition is true, in function *value* on line 5, we select a consistent value if no inverse value exists, i.e., if the *invertibility condition* for  $n$  with respect to  $n_x$ ,  $t$ ,  $s = \mathcal{A}_b(n_s)$  (if any other operand  $n_s$ ) and  $x = \mathcal{A}_t(n_x)$  is false. Else, we non-deterministically choose between inverse and consistent values (with a preference for inverse values). As shown in [24], the latter non-deterministic choice between inverse and consistent values (as opposed to always choosing inverse values if possible) is necessary for the sake of completeness. Note that if multiple possible inverse or consistent values exist, we non-deterministically select one of them.

**Invertibility Conditions.** Given target value  $t$  for bit-vector operation  $n$  and the current assignment  $s = \mathcal{A}_b(n_s)$  of its operand other than  $n_x$  (if any), computing an *inverse* value is in general *not* always possible, even when not considering constant bits in  $n_x$ . As in [26], we refer to the exact condition under which an inverse value can be computed for  $x$  given  $s$  and  $t$  as *invertibility condition* (*IC*), e.g., for bit-vector literal  $x \diamond s \approx t$  we have that  $\forall s, t. (IC(s, t) \Leftrightarrow \exists y. (y \diamond s \approx t))$ . We lift this to consider constant bits in  $n_x$  by interpreting  $x$  as a ternary bit-vector  $x = \mathcal{A}_t(n_x)$ , and yield generalized invertibility conditions for all operators in Table I as given in Tables II–III. Thus, for literal  $x \diamond s \approx t$  we now have that

$$\forall x, s, t. (IC(x, s, t) \Leftrightarrow \exists y. (y \diamond s \approx t \wedge mcb(x, y))). \quad (1)$$

The unary case is defined analogously. Note that invertibility conditions without considering constant bits in  $n_x$  were first given in pseudocode in [24], and formalized and verified for up to 65 bits in [26]. In Tables II and III, we indicate the part of an invertibility condition that is the condition without considering constant bits in  $n_x$  in blue. For cases that do not include such a condition, this condition is  $\top$ . For example, for  $x \cdot s \approx t$ , the blue part of the invertibility condition ensures that  $ctz(s) \leq_u ctz(t)$ , and the remainder determines if possible solutions match constant bits in  $x$ .

**Consistency Conditions.** Computing a *consistent* value when *not* considering constant bits is *always* possible, and thus in the procedure in [24], it was never possible to encounter a case where no inverse *and* no consistent value exists. In contrast, when considering constant bits in  $n_x$ , it is *not always* possible to determine a consistent value for  $n_x$ , e.g., for  $\bullet 0 \cdot s \approx 01$  there is no value that  $x$  can assume such that  $t$  can be produced for some  $s$ . We therefore introduce the new notion of *consistency condition* when considering constant bits in  $n_x$  as follows.

**Definition 6. (Consistency Condition)** Given a bit-vector literal  $\diamond x \approx t$  or  $x \diamond s \approx t$ , we refer to the exact condition under which a consistent value can be computed for  $x$  given  $t$  as *consistency condition* (*CC*).

For unary operations, any invertibility condition is also a consistency condition. For  $x \diamond s \approx t$ , we have that

$$\forall x, t. (CC(x, t) \Leftrightarrow \exists y, s. (y \diamond s \approx t \wedge mcb(x, y))) \quad (2)$$

and the other binary case is defined analogously. The consistency conditions with respect to constant bits in  $x$  for bit-vector operators in Table I are given in Tables IV and V.

**Synthesizing Conditions.** Previous work utilized SyGuS techniques to synthesize invertibility conditions for bit-vector [26] and floating-point [10] literals. For this work, we adopted the SyGuS approach from [26] to find invertibility and consistency conditions with respect to constant bits. We encoded Equations 1 and 2 as SyGuS problems to synthesize functions *IC* and *CC* and defined a general grammar that includes all bit-vector operators from Table I (excl. concatenation and sign extension), common logical connectives and the additional operators *mcb*, *clz*, *ctz*, and *clo*. For the invertibility condition problems, we further added the corresponding condition that must hold without considering constant bits (indicated in blue in Tables II–III) as pre-condition. In total, we generated 30 (15 invertibility, 15 consistency) SyGuS problems and used the SyGuS solver in CVC4 [29] with a time limit of 7200 seconds and 8GB memory limit. Overall, CVC4 was able to synthesize 10 conditions (3 invertibility and 7 consistency conditions). Unfortunately, all three invertibility conditions were trivial and we were not successful in synthesizing any complex invertibility conditions. Of the seven consistency conditions, on the other hand, three were significantly simpler than the manually crafted ones (marked with  $\star$  in Table V).

**Completeness (PAC).** As in [24], the two main sources of non-determinism of our procedure are path and value selection when down-propagating target values. However, we now aim to only propagate target values that can actually be assumed, i.e.,  $mcb(\mathcal{A}_t(n_x), \mathcal{A}_b(n_x)) = \top$ . Path selection still implements the same strategy as in [24], i.e., essential inputs are selected over non-essential inputs. The notion of essential input has been lifted to constant bits, however, this only excludes values that can never be assumed. Generalizing value selection to ternary values, as intended, significantly changes the behavior of the algorithm compared to [24]. Since we compute consistent and inverse values with respect to

$(x \approx s) \approx t$	$(t \Rightarrow mcb(x, s)) \wedge (\sim t \Rightarrow (x^{hi} \not\approx x^{lo} \vee x^{hi} \not\approx s))$
$(x <_u s) \approx t$	$(t \Rightarrow s \not\approx 0 \wedge x^{lo} <_u s) \wedge (\sim t \Rightarrow (x^{hi} \geq_u s))^*$
$(s <_u x) \approx t$	$(t \Rightarrow s \not\approx \text{ones} \wedge x^{hi} >_u s) \wedge (\sim t \Rightarrow x^{lo} \leq_u s)$
$(x <_s s) \approx t$	$(t \Rightarrow (s \not\approx \text{smin} \wedge ((x[msb] \approx 0 \wedge x^{lo} <_s s) \vee (x[msb] \in \{1, \bullet\} \wedge (\text{smin} \mid x^{lo}) <_s s))) \wedge (\sim t \Rightarrow ((x[msb] \approx 1 \wedge x^{hi} \geq_s s) \vee (x[msb] \in \{0, \bullet\} \wedge (\text{smax} \& x^{hi}) \geq_s s))))$
$(s <_s x) \approx t$	$(t \Rightarrow (s \not\approx \text{smax} \wedge ((x[msb] \approx 1 \wedge s <_s x^{hi}) \vee (x[msb] \in \{0, \bullet\} \wedge s <_s (\text{smax} \& x^{hi})))) \wedge (\sim t \Rightarrow (((x[msb] \approx 0 \wedge s \geq_s x^{lo}) \vee (x[msb] \in \{1, \bullet\} \wedge s \geq_s (\text{smin} \mid x^{lo}))))))$

TABLE II: Invertibility conditions for bit-vector predicates modulo constant bits in  $x$ .

constant bits and break on conflict when no consistent value exists, it is guaranteed that every target value that is propagated down all the way to the primary inputs can be assumed with respect to constant bits in the inputs. As a consequence, we only exclude target values that can never be part of a satisfying assignment of the input formula. Our procedure is thus still probabilistically approximately complete (PAC), following the same line of argument as in [24].

#### A. Verifying Invertibility and Consistency Conditions

The invertibility and consistency conditions in Tables II–V are utilized in our procedure to determine whether a given target value can be down-propagated. Incorrect conditions will not result in unsoundness of the procedure, but may affect completeness (PAC). To verify the correctness of these conditions we check for each literal and bit-width up to 65 if the negation of the corresponding quantified formula as defined above is unsatisfiable. For unary literals, consistency conditions are also invertibility conditions and we only have to check the unsatisfiability of

$$\exists x^{lo}, x^{hi}, t. \text{valid}(x^{lo}, x^{hi}) \wedge \neg(\text{IC}(x, t) \Leftrightarrow \exists y. (\diamond y \approx t \wedge mcb(x^{lo}, x^{hi}, y))).$$

Note that we do not test the conditions for extracts since they can essentially be reduced to the checks for equality, which makes tests for all combinations of upper and lower indices redundant. Further, in order to keep the number of queries manageable, we only check for signed extensions of up to 4 bits. For binary literals, we check each of the formulas

$$\begin{aligned} &\exists x^{lo}, x^{hi}, s, t. \text{valid}(x^{lo}, x^{hi}) \wedge \\ &\neg(\text{IC}(x, s, t) \Leftrightarrow \exists y. (y \diamond s \approx t \wedge mcb(x^{lo}, x^{hi}, y))) \\ &\quad \exists x^{lo}, x^{hi}, t. \text{valid}(x^{lo}, x^{hi}) \wedge \\ &\quad \neg(\text{CC}(x, t) \Leftrightarrow \exists y, s. (y \diamond s \approx t \wedge mcb(x^{lo}, x^{hi}, y))) \end{aligned}$$

The other binary case is defined analogously. Note that for the sake of simplicity, we only use operands of the same bit-width (from 1 to 65) for concatenation. Concatenation can again be seen as a special case of equality, i.e.,  $x \circ s \approx t$  can be interpreted as  $x \circ s \approx t_x \circ t_s$ , and the check can be reduced to checking the condition  $\text{IC}((x \approx t_x) \approx 1) \wedge s \approx t_s$ . Hence,

$x + s \approx t$	$mcb(x, t - s)$
$x \cdot s \approx t$	$(-s \mid s) \& t \approx t \wedge (s \approx 0 \vee ((\text{odd}(s) \Rightarrow mcb(x, t \cdot s^{-1})) \wedge (\neg \text{odd}(s) \Rightarrow mcb(x \ll c, y \ll c))))$ with $c = ctz(s)$ and $y = (t \gg c) \cdot (s \gg c)^{-1}$
$x \bmod s \approx t$	$\sim(-s) \geq_u t \wedge ((s \approx 0 \vee t \approx \text{ones}) \Rightarrow mcb(x, t)) \wedge ((s \not\approx 0 \wedge t \not\approx \text{ones}) \Rightarrow \exists y. (mcb(x, s \cdot y + t) \wedge \neg mulo(s, y) \wedge \neg addo(s \cdot y, t)))$
$s \bmod x \approx t$	$(t + t - s) \& s \geq_u t \wedge (s \approx t \Rightarrow (x^{lo} \approx 0 \vee x^{hi} >_u t)) \wedge (s \not\approx t \Rightarrow \exists y. (mcb(x, y) \wedge y >_u t \wedge (s - t) \bmod y \approx 0))$
$x \div s \approx t$	$(s \cdot t) \div s \approx t \wedge (t \approx 0 \Rightarrow x^{lo} <_u s) \wedge ((t \not\approx 0 \wedge s \not\approx 0) \Rightarrow \exists y. (mcb(x, y) \wedge (\neg c \Rightarrow y <_u s \cdot t + 1) \wedge (c \Rightarrow y \leq_u \text{ones})))$ with $c = mulo(s, t + 1) \vee addo(t, 1)$
$s \div x \approx t$	$s \div (s \cdot t) \approx t \wedge (t \not\approx \text{ones} \Rightarrow x^{hi} >_u 0) \wedge ((s \not\approx 0 \vee t \not\approx 0) \Rightarrow (s \div x^{hi} \leq_u t) \wedge \exists y. (mcb(x, y) \wedge (t \approx \text{ones} \Rightarrow y \geq_u 0 \wedge y \leq_u s \div t) \wedge (t \not\approx \text{ones} \Rightarrow y >_u t + 1 \wedge y \leq_u s \div t)))$
$x \& s \approx t$	$t \& s \approx t \wedge ((s \& x^{hi}) \& c) \approx (t \& c)$ with $c = \sim(x^{lo} \oplus x^{hi})$
$x \oplus s \approx t$	$mcb(x, s \oplus t)$
$x \ll s \approx t$	$(t \gg s) \ll s \approx t \wedge mcb(x \ll s, t)$
$s \ll x \approx t$	$ctz(s) \leq_u ctz(t) \wedge (t \not\approx 0 \Rightarrow s \ll c \approx t) \wedge (t \approx 0 \Rightarrow (x^{hi} \geq_u c \vee s \approx 0)) \wedge (t \not\approx 0 \Rightarrow mcb(x, c))$ with $c = ctz(t) - ctz(s)$
$x \gg s \approx t$	$(t \ll s) \gg s \approx t \wedge mcb(x \gg s, t)$
$s \gg x \approx t$	$clz(s) \leq_u clz(t) \wedge (t \not\approx 0 \Rightarrow s \gg c \approx t) \wedge (t \approx 0 \Rightarrow (x^{hi} \geq_u c \vee s \approx 0)) \wedge (t \not\approx 0 \Rightarrow mcb(x, c))$ with $c = clz(t) - clz(s)$
$x \gg_a s \approx t$	$(s <_u \kappa(s) \Rightarrow ((t \ll s) \gg_a s \approx t)) \wedge (s \geq_u \kappa(s) \Rightarrow (t \approx \text{ones} \vee t \approx 0)) \wedge mcb(x \gg_a s, t)$
$s \gg_a x \approx t$	$s[msb] \approx 0 \Rightarrow IC(s \gg x = t) \wedge s[msb] \approx 1 \Rightarrow IC(\sim s \gg x = \sim t)$
$x \circ s \approx t$	$s \approx t^s \wedge mcb(x, t^s)$ with $t^s = t[msb : \kappa(s)]$ and $t^s = t[\kappa(s) - 1 : lsb]$
$s \circ x \approx t$	$s \approx t^s \wedge mcb(x, t^x)$ with $t^s = t[msb : \kappa(s)]$ and $t^x = t[\kappa(s) - 1 : lsb]$
$x \langle n \rangle \approx t$	$(t^n \approx 0 \vee t^n \approx \text{ones}) \wedge mcb(x, t^n)$ with $t^n = t[msb : \kappa(x) - 1]$ and $t^n = t[\kappa(x) - 1 : lsb]$
$x[u : l] \approx t$	$mcb(x[u : l], t)$

TABLE III: Invertibility conditions for non-predicate bit-vector operators modulo constant bits in  $x$ .

$(x \approx s) \approx t$	$\top$
$(x <_u s) \approx t$	$\sim t \vee x^{lo} \not\approx \text{ones}$
$(s <_u x) \approx t$	$\sim t \vee x^{hi} \not\approx 0$
$(x <_s s) \approx t$	$\sim t \vee (x^{lo} \approx x^{hi} \Rightarrow x^{lo} \not\approx \text{smax})$
$(s <_s x) \approx t$	$\sim t \vee (x^{lo} \approx x^{hi} \Rightarrow x^{lo} \not\approx \text{smin})$

TABLE IV: Consistency conditions for bit-vector predicates modulo constant bits in  $x$ .

\* Errata: In the original publication of this paper, the red subterm of this condition was incorrectly given as  $s \geq_u x \approx t$ . However, the implementation and verification used the correct term  $x^{hi} \geq_u s$ . See verification conditions for case bvuge (`verify_ic_bvuge_x_s_.smt2`) in the artifact.

$x + s \approx t$	$\top$
$x \cdot s \approx t$	$(t \not\approx 0 \Rightarrow x^{hi} \not\approx 0) \wedge (\text{odd}(t) \Rightarrow x^{hi}[\text{lsb}] \not\approx 0) \wedge (\neg\text{odd}(t) \Rightarrow \exists y. (\text{mcb}(x, y) \wedge \text{ctz}(t) \geq_u \text{ctz}(y)))$
$x \bmod s \approx t$	$(t \approx \text{ones} \Rightarrow \text{mcb}(x, \text{ones})) \wedge (t \not\approx \text{ones} \Rightarrow (t >_u (\text{ones} - t) \Rightarrow \text{mcb}(x, t)) \wedge (t \leq_u (\text{ones} - t) \Rightarrow (\text{mcb}(x, t) \vee \exists y. (\text{mcb}(x, y) \wedge y >_u 2 \cdot t))))$
$\star s \bmod x \approx t$	$(x^{lo} \gg (t \div x^{hi})) \approx x^{lo}$
$x \div s \approx t$	$(t \not\approx \text{ones} \Rightarrow x^{hi} \geq_u t) \wedge (t \approx 0 \Rightarrow x^{lo} \not\approx \text{ones}) \wedge ((t \not\approx 0 \wedge t \not\approx \text{ones} \wedge t \not\approx 1 \wedge \neg\text{mcb}(x, t)) \Rightarrow (\neg\text{mulo}(2, t) \wedge \exists y, o. (\text{mcb}(x, y \cdot t + o) \wedge y \geq_u 1 \wedge o \leq_u c \wedge \neg\text{mulo}(y, t) \wedge \neg\text{addo}(y \cdot t, o))))$ with $c = \min(y - 1, x^{hi} - y \cdot t)$
$s \div x \approx t$	$(t \approx \text{ones} \Rightarrow (\text{mcb}(x, 0) \vee \text{mcb}(x, 1))) \wedge (t \not\approx \text{ones} \Rightarrow (\neg\text{mulo}(x^{lo}, t) \wedge \exists y. (y >_u 0 \wedge \text{mcb}(x, y) \wedge \neg\text{mulo}(y, t))))$
$x \& s \approx t$	$t \& x^{hi} \approx t$
$x \oplus s \approx t$	$\top$
$x \ll s \approx t$	$\exists y. (y \leq_u \text{ctz}(t) \wedge \text{mcb}(x \ll y, t))$
$\star s \ll x \approx t$	$((\text{ones} \ll x^{lo}) \wedge t) \approx t$
$x \gg s \approx t$	$\exists y. (y \leq_u \text{clz}(t) \wedge \text{mcb}(x \gg y, t))$
$\star s \gg x \approx t$	$((\text{ones} \gg x^{lo}) \wedge t) \approx t$
$x \gg_a s \approx t$	$(t \approx 0 \vee t \approx \text{ones}) \Rightarrow \exists y. (y[\text{msb}] \approx t[\text{msb}] \wedge \text{mcb}(x, y)) \wedge (t \not\approx 0 \wedge t \not\approx \text{ones}) \Rightarrow (\exists y. (c \Rightarrow y \leq_u \text{clo}(t) \wedge \sim c \Rightarrow y \leq_u \text{clz}(t) \wedge \text{mcb}(x \gg_a y, t)))$ with $c = (t \ll y)[\text{msb}] \approx 1$
$s \gg_a x \approx t$	$t \approx 0 \vee t \approx \text{ones} \vee \exists y. (c \Rightarrow y <_u \text{clo}(t) \wedge \sim c \Rightarrow y <_u \text{clz}(t)) \wedge \text{mcb}(x, y)$ with $c = t[\text{msb}] \approx 1$
$x \circ s \approx t$	$\text{mcb}(x, t[\text{msb} : \kappa(s)])$
$s \circ x \approx t$	$\text{mcb}(x, t[\text{msb} - \kappa(s) : \text{lsb}])$
$x \langle n \rangle \approx t$	$IC(x \langle n \rangle = t)$
$x[u : l] \approx t$	$IC(x[u : l] = t)$

TABLE V: Consistency conditions for non-predicate bit-vector operators modulo constant bits in  $x$ . Conditions marked with  $\star$  are conditions synthesized with SyGuS.

it is not necessary to check the condition for concatenation for all possible combinations of bit-widths of the operands.

We split the conditions for the predicates by the value of  $t$  and generated in total 3575 quantified bit-vector verification problems for 55 conditions (30 invertibility and 25 consistency conditions). To verify these problems, we used our SMT solver Bitwuzla [23] and the solvers CVC4 [5] and Z3 [13]. Note that we had to exclude Q3B [18] due to disagreements with all three other solvers on 2/3 of the commonly solved instances. We used a time limit of 3600 seconds and a memory limit of 8GB and ran this verification task on a cluster with Intel Xeon CPU E5-2620 CPUs with 2.1GHz and 128GB memory.

We consider a condition to be verified for a certain bit-width, if all solvers that don't run into the time limit agree on its status, and the status is unsat. Overall we were able to verify 2867 out of 3575 instances (80.2%). For operators  $\{\approx, <_u, \&, \oplus, \ll, \gg, \gg_a, +, \circ, \langle \rangle, [:]\}$  we were able to verify *all invertibility conditions*, and for operators  $\{\approx, <_u, \&, +, \cdot, \circ\}$

we were able to verify *all consistency conditions* for all bit-widths up to 65. For  $x <_s s$ , no solver was able to verify the invertibility condition for bit-width 36, and for  $x \oplus s$  no solver was able to verify the consistency condition for bit-widths 32, 49, 52 and 58. The remaining conditions were verified at least for bit-widths up to (and including) 7.

Verifying the correctness of the presented invertibility and consistency conditions up to some bit-width establishes a certain level of trust but does not prove that they are correct for all possible bit-widths. Proving the correctness for all bit-widths is more involved since it requires bit-width independent proofs [27] and is left to future work.

### B. Computing Assignment $\mathcal{A}_t$

Assignment  $\mathcal{A}_t : \mathcal{N} \mapsto \{0, 1, \bullet\}^+$  maps each node  $n \in \mathcal{N}$  to a ternary bit-vector, which represents constant bits in  $n$ . We determine these constant bits upfront by utilizing the And-Inverter Graph (AIG) circuit representation of the input formula. Rewriting on the AIG layer during the translation [11] allows to simplify gates to constants, which are then mapped back to the word-level and represented as the constant bits of the corresponding ternary bit-vectors in  $\mathcal{A}_t$ .

Bit-blasting the input formula to AIGs introduces additional overhead, both in terms of time and memory, in particular for large bit-widths. In [21], the authors proposed word-level propagators based on ternary bit-vectors for a limited set of bit-vector operators, which was later extended in [34]. These propagators might allow to determine constant bits without the additional overhead of bit-blasting to AIGs. We leave utilizing these propagators to compute  $\mathcal{A}_t$  to future work.

## V. EXTENSIONS

Tables III and V include invertibility conditions and consistency conditions for the bit-vector operators  $\oplus, \gg_a, <_s$  and sign extension, which are not considered in [24]. Instead, they are rewritten in terms of a smaller set of base operators. For example, signed bit-vector operators are encoded by means of unsigned operations only, and bit-wise operations are mostly expressed in terms of  $\&$  and  $\sim$ . As a consequence, the overall size of the formula (in terms of number of nodes) increases. This can have a negative impact on our local search procedure, since the number of paths that need to be considered when propagating target values potentially increases. Further, eliminating bit-vector operators can introduce multiple occurrences of their operands, which can make it harder to find a value that is part of a satisfying assignment. For example, the bit-vector exclusive or operation  $t_1 = x \oplus s$  can be represented as  $t_2 = ((x | s) \& \sim(x \& s))$ . Selecting an inverse value for  $x$  in  $t_1$  only requires one propagation step, whereas for  $x$  in  $t_2$  we have to find a value that is also consistent with  $x | s$  and  $\sim(x \& s)$ , which may take multiple propagation steps.

We extended the set of operators in [24] to natively support bit-vector operators  $\oplus, \gg_a, <_s$ , and sign extension since they are widely used in SMT-LIB benchmarks. Other operators such as signed division and remainder operators do not occur

as frequently and we leave the native support for these operators to future work.

### A. Tightening Bounds for Inequalities

Given a bit-vector inequality literal  $x \diamond s \approx t$  ( $s \diamond x \approx t$ ) with  $\diamond \in \{<_u, <_s\}$ , an inverse value for  $x$  is a random value within a certain range. The lower (upper) bound is determined by  $s$ , whereas the other bound is at least the (un)signed minimum (at most the (un)signed maximum) value, depending on constant bits in  $x$ . For example, for  $x <_u s$  with target value  $t = 1$ , the range of possible inverse values  $v$  for  $x$  is  $x^{lo} \leq_u v <_u s$ . When such a range is large and only few values within this range are part of a satisfying assignment, randomly picking the right value can have a very low probability. For example, for  $x_0 <_u s$  with target value  $t = 1$  and  $x_0 = x_1 \langle w \rangle$ , we first compute an inverse value  $v_0$  for  $x_0$  within range  $x^{lo} \leq_u v_0 <_u s$ , and then propagate  $v_0$  to  $x_1 \langle w \rangle$ . The sign extension of  $x_1$  requires that the  $w + 1$  left-most bits of  $v_0$  are either  $0_{[w+1]}$  or  $\text{ones}_{[w+1]}$ , i.e., the value of bit  $v_0[\kappa(x_1) - 1]$  determines the  $w$  left-most bits. However, this information is not known when computing an inverse value for  $x_0$  since we do not consider its kind. As a consequence, we may select inverse values where the  $w + 1$  left-most bits are neither  $\text{ones}_{[w+1]}$  or  $0_{[w+1]}$ , which will immediately produce a conflict in the next propagation step.

In the following, we discuss heuristics that address this weakness and further tighten the bounds based on the currently satisfied top-level inequality constraints.

**Inequalities with Sign Extension.** Consider an unsigned inequality over sign extension  $x \langle w \rangle <_u s$  with target value  $t = 1$ . We can define the following two ranges when computing an inverse value  $v$  for  $x \langle w \rangle$ .

$$\text{ones}_{[w+1]} \circ 0_{[\kappa(x)-1]} \leq_u v <_u s \quad (3)$$

$$0 \leq_u v <_u \min(s, 0_{[w]} \circ \text{smin}_{[\kappa(x)]}) \quad (4)$$

Each of these ranges can only be considered if it is valid, i.e., if the lower bound is strictly less than the upper bound. Further, range (3) is only applicable if  $x[\text{msb}] \in \{1, \bullet\}$ , and range (4) if  $x[\text{msb}] \in \{0, \bullet\}$ . Picking an inverse value  $v$  with  $mcb(x, v)$  from any of these two ranges guarantees that the  $w + 1$  left-most bits are either  $0_{[w+1]}$  or  $\text{ones}_{[w+1]}$ . Similar ranges can be derived for  $t = 0$  and  $<_s$ .

**Satisfied Inequality Constraints.** An additional, more general way to tighten the bounds of inverse value computation for an inequality literal with operand  $n_x$  is to determine these bounds with respect to other inequality constraints on  $n_x$  that are currently satisfied in  $\mathcal{A}_b$ . We consider all satisfied inequalities on  $n_x$  that are conjuncts reachable from the root. If this results in an invalid range, i.e., the lower bound is greater than the upper bound, we fall back to computing a consistent value without this bound tightening strategy.

We only consider this heuristic for inverse values and not for consistent values in order to maintain completeness. For example, consider formula  $n_x <_u 100 \wedge n_a <_u n_x$  with  $\mathcal{A}_b(n_x) = 110$  and  $\mathcal{A}_b(n_a) = 101$ . Assignment  $\mathcal{A}_b$  satisfies

inequality  $n_a <_u n_x$ , but falsifies  $n_x <_u 100$ . We select node  $n_x <_u 100$ , assume  $x <_u 100 \approx 1$  with  $x = \mathcal{A}_t(n_x)$ , and determine 100 as upper and  $\mathcal{A}_b(n_x)$  as lower bound of the inverse value for  $x$ . Since this range is invalid, we fall back to computing a consistent value. If we compute a consistent value with the bound tightening strategy above, we would ignore the upper bound and use  $\mathcal{A}_b(n_x)$  as lower bound. However, this would result in getting stuck in computing consistent values greater than  $\mathcal{A}_b(n_x)$ , which will never satisfy  $n_x <_u 100$  and would therefore be incomplete.

Note that in our implementation, we currently only consider inequality constraints that have the same signedness as the inequality we currently compute an inverse for. Further, this heuristic can be generalized to apply to inverse value computation in general (not only for inequality literals), which requires to incorporate ranges into all inverse value computations. We leave these extensions to future work.

## VI. EVALUATION

We implemented our techniques in our SMT solver Bitwuzla [23], which is the successor of our SMT solver Boolector [28]. It supports the theories of arrays, bit-vectors, floating-points and uninterpreted functions and their combinations. We first evaluate our generalized procedure and the proposed extensions in comparison to the base procedure presented in [24]. We then show the performance of a sequential portfolio combination of our procedure with state-of-the-art bit-blasting as implemented in Bitwuzla. We performed all experiments on a cluster with Intel Xeon CPU E5-2620 CPUs with 2.1GHz and 128GB memory. We use an 8GB memory limit for each solver/benchmark pair and count memory out as time out. We consider the following configurations:

- 1) **base** The propagation-based local search procedure presented in [24], which serves as a baseline for our propagation-based local search configurations.
- 2) **prop-c** Our ternary propagation-based local search procedure (Section IV).
- 3) **prop-c+** Configuration **prop-c** with additional propagators for  $\oplus$ ,  $\gg_a$ , and sign extension enabled.
- 4) **prop-cb+** Configuration **prop-c+** with all bound tightening heuristics from Section V and  $<_s$  propagator enabled.
- 5) **bb** The bit-blasting engine of Bitwuzla with CaDiCaL [8] version 1.2.1, CryptoMiniSat [32] version 5.7.0, Kissat [9] version sc2020 (winner of SAT competition 2020), and Lingeling [7] version bcj as SAT back ends.
- 6) **bb-prop-cb+** Sequential portfolio of **bb** and **prop-cb+**, where **prop-cb+** is run prior to invoking **bb** with a limit of 10k propagation steps and 2M steps for updating  $\mathcal{A}_b$ .

We evaluated configurations **base**, **prop-c**, **prop-c+**, and **prop-cb+** on all 14,382 QF\_BV benchmarks from SMTLIB with status “sat”. We ran each configuration with 20 different seeds for the random number generator and a time limit of 60 seconds. Figure 3 shows the number of solved instances of **base**, **prop-c**, **prop-c+**, and **prop-cb+** over all 20 runs with different seeds as box-and-whiskers plots. The box of a plot shows the interquartile range (IQR), and the

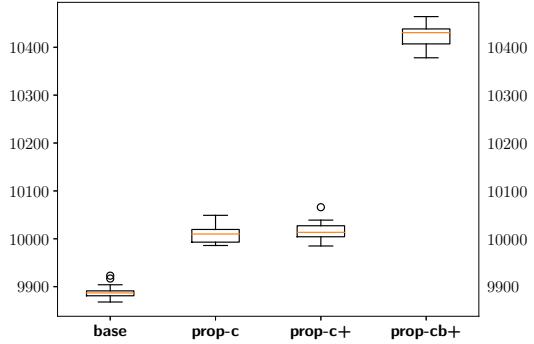


Fig. 3: Number of solved instances over 20 runs (with different seeds) of configurations **base**, **prop-c**, **prop-c+**, and **prop-cb+**.

orange line indicates the median value over all runs of a configuration. The ends of the whiskers indicate minimum and maximum values excluding outliers, which are shown as circles. IQR measures the distance between the lower and upper quartile. Additionally, we also determine the median absolute deviation (MAD), which is a measure for how much one run deviates from the median. Configuration **prop-c** (IQR: 26.5, MAD: 13.4) clearly outperforms **base** (IQR: 10.3, MAD: 9.0) with +120 (median) solved instances. Enabling additional propagators for  $\oplus$ ,  $\gg_a$ , and sign extension in **prop-c+** (IQR: 23.0, MAD: 14.5) increases the number of solved instances by +3 (median) in comparison to **prop-c**. Enabling the bound tightening heuristics achieves the best results, with over 400 additional solved instances (median) compared to **prop-c+** (IQR: 31.3, MAD: 18.9).

Value computation in **prop-c** is expected to propagate more precise values than **base**, i.e., we expect the number of moves required to solve a problem to decrease. In an additional experiment, we compare the runs of **base** and **prop-c** that are closest to their median on commonly solved instances. As expected, configuration **prop-c** requires 70% less moves, 63% less propagations and 44% less updates of  $\mathcal{A}_b$  than **base** while being 9% faster in terms of solving time.

By enabling the additional operators  $\oplus$ ,  $\gg_a$ , and sign extension (as discussed in Section V), we observed that the median of configuration **prop-c+** increased by 3 solved instances. Further, enabling  $<_s$  for configuration **prop-c+** resulted in a considerable loss of 262 median solved instances. This is due to the *uclid* benchmark family, which contains many signed inequalities that effectively define (small) ranges over positive/unsigned values only. For these instances, rewriting  $<_s$  in terms of  $<_u$  thus significantly reduces the number of possible values for its operands<sup>1</sup>. However, natively handling  $<_s$  in combination with our bound tightening heuristics in configuration **prop-cb+** solves all 262 *uclid* benchmarks. Generally, natively handling different sets of operators yields (sometimes significantly) different results. Identifying a minimal set that

<sup>1</sup>Bitwuzla rewrites  $a <_s b$  to  $(a[msb] > b[msb]) \vee (a[msb] \approx b[msb] \wedge a[msb - 1 : 0] <_u b[msb - 1 : 0])$ .

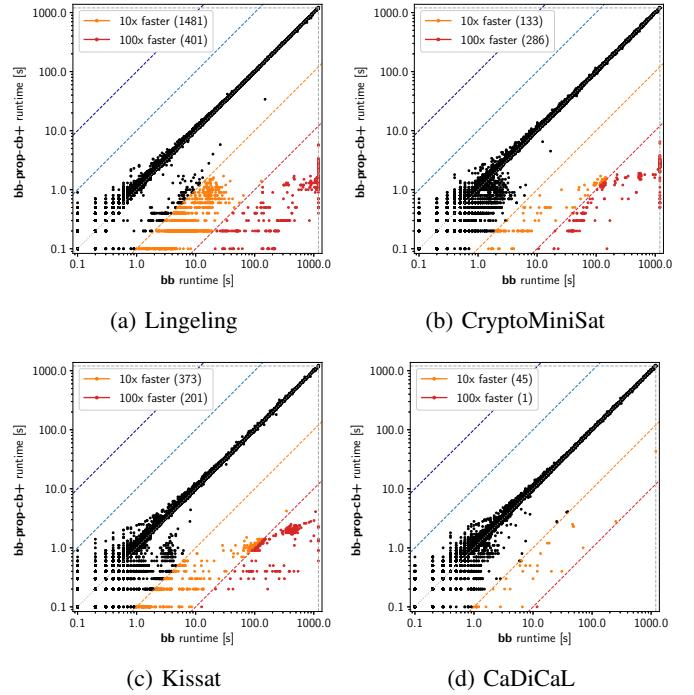


Fig. 4: **bb** versus **bb-prop-cb+** with Lingeling (4a), Crypto-MiniSat (4b), Kissat (4c), and CaDiCaL (4d) with a time limit of 1200 seconds.

allows for best performance may simplify the implementation and is an interesting direction for future work.

Figure 4 shows the performance of the sequential portfolio combination **bb-prop-cb+** in comparison to configuration **bb** with a time limit of 1200 seconds on *all* QF\_BV benchmarks (41,713 total). We compare **bb-prop-cb+** against bit-blasting with CaDiCaL, CryptoMiniSat, Kissat, and Lingeling as SAT back ends. Our sequential portfolio combination clearly compensates weaknesses of CryptoMiniSat, Kissat, and Lingeling on satisfiable instances. The bit-blasting engine with CaDiCaL as a back end significantly improves over the other configurations, but **bb-prop-cb+** still improves over the configuration with CaDiCaL in terms of runtime. Overall, the overhead introduced on unsatisfiable instances is negligible.

All experimental data is available at <https://bitwuzla.github.io/papers/fmcad2020>.

## VII. CONCLUSION

We have presented a generalization of propagation-based local search for quantifier-free bit-vector formulas with respect to constant bits to ternary values. We have derived and verified invertibility and consistency conditions modulo constant bits for a majority of the bit-vector operators defined in SMT-LIB 2. We have shown that our approach yields more precise value propagation and considerably improves the performance.

Our sequential portfolio utilizes propagation-based local search and improves over pure bit-blasting. When falling back to the bit-blasting engine, however, it does not share any information, which is an interesting direction for future work.

## REFERENCES

- [1] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.
- [2] A. Balint, A. Belov, M. J. H. Heule, and M. Järvisalo, editors. *SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2013.
- [3] A. Balint, A. Belov, M. Järvisalo, and C. Sinz. Overview and analysis of the SAT challenge 2012 solver competition. *Artificial Intelligence*, 223:120–155, 2015.
- [4] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
- [5] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
- [6] A. Belov, M. J. H. Heule, and M. Järvisalo, editors. *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2014.
- [7] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018. In M. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*, pages 13–14. University of Helsinki, 2018.
- [8] A. Biere. CaDiCaL at the SAT Race 2019. In M. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.
- [9] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, page 49, 2020. To appear.
- [10] M. Brain, A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli. Invertibility conditions for floating-point formulas. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 116–136. Springer, 2019.
- [11] R. Brummayer and A. Biere. Local Two-Level And-Inverter Graph Minimization without Blowup. In *2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia, October 2006. Proceedings*, 2006.
- [12] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 SMT solver. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.
- [13] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [14] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.
- [15] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi. Stochastic local search for satisfiability modulo theories. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 1136–1143. AAAI Press, 2015.
- [16] A. Griggio, Q. Phan, R. Sebastiani, and S. Tomasi. Stochastic local search for SMT: combining theory solvers with walksat. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2011.
- [17] H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *AAAI/IAAI*, pages 661–666. AAAI Press / The MIT Press, 1999.
- [18] M. Jonás and J. Strejcek. Q3B: an efficient bdd-based SMT solver for quantified bit-vectors. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 64–73. Springer, 2019.
- [19] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
- [20] M. Manzano. Introduction to many-sorted logic. In *Many-sorted logic and its applications*, pages 3–86. John Wiley & Sons, Inc., New York, NY, USA, 1993.
- [21] L. D. Michel and P. V. Hentenryck. Constraint satisfaction over bit-vectors. In M. Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 527–543. Springer, 2012.
- [22] A. Niemetz. *Bit-Precise Reasoning Beyond Bit-Blasting*. PhD thesis, Johannes Kepler University Linz, 2017.
- [23] A. Niemetz and M. Preiner. Bitwuzla at the SAT-COMP 2020. *CoRR*, abs/2006.01621, 2020.
- [24] A. Niemetz, M. Preiner, and A. Biere. Propagation based local search for bit-precise reasoning. *Formal Methods in System Design*, 51(3):608–636, 2017.
- [25] A. Niemetz, M. Preiner, A. Biere, and A. Fröhlich. Improving local search for bit-vector logics in SMT with path propagation. In *Proceedings of the Fourth International Workshop on Design and Implementation of Formal Tools and Systems, Austin, TX, USA, September 26-27, 2015*, pages 1–10, 2015.
- [26] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli. Solving quantified bit-vectors using invertibility conditions. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 236–255. Springer, 2018.
- [27] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. W. Barrett, and C. Tinelli. Towards bit-width-independent proofs in SMT solvers. In P. Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2019.
- [28] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2 , btormc and boolector 3.0. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.
- [29] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.
- [30] J. P. M. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.
- [31] M. Soos and A. Biere. CryptoMiniSat 5.6 with YalSAT at the SAT Race 2019. In M. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2019.
- [32] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

- [33] M. Soos, B. Selman, and H. Kautz. CryptoMiniSat 5.6 with WalkSAT at the SAT Race 2019. In M. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 12–13. University of Helsinki, 2019.
- [34] W. Wang, H. Søndergaard, and P. J. Stuckey. Wombit: A portfolio bit-vector solver using word-level propagation. *J. Autom. Reasoning*, 63(3):723–762, 2019.
- [35] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.
- [36] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.