CDCL-inspired Word-level Learning for Bit-vector Constraint Solving

Zakaria Chihani, François Bobot, Sébastien Bardin

CDCL-inspired Word-level Learning for Bit-vector Constraint Solving*

Zakaria Chihani, François Bobot, Sébastien Bardin
CEA, LIST, Software Security Lab, Gif-sur-Yvette, France
(first.last@cea.fr)

Abstract

The theory of quantifier-free bitvectors is of paramount importance in software verification. The standard approach for satisfiability checking reduces the bitvector problem to a Boolean problem, leveraging the powerful SAT solving techniques and their conflict-driven clause learning (CDCL) mechanisms. Yet, this bit-level approach loses the structure of the initial bitvector problem. We propose a conflict-driven, word-level, combinable constraints learning for the theory of quantifier-free bitvectors. This work paves the way to truly word-level decision procedures for bitvectors, taking full advantage of word-level propagations recently designed in CP and SMT communities.

1 Introduction

Context. Since the early 2000's, there is a significant trend in the research community toward reducing verification problems to the satisfiability (SAT) problem of first-order logical formulas over well-chosen theories (e.g. bitvectors, arrays or floating-point arithmetic), leveraging the advances of modern powerful SAT and SMT solvers [16, 25, 20, 2]. Besides weakest-precondition calculi dating back to the 1970's [11], most major recent verification approaches follow this idea [7, 13, 14, 18].

In the context of verification, it is of paramount importance to efficiently and faithfully represent basic data types found in any programming language, such as arrays, floating points, integers and – as is the focus of this paper – bitvectors (*i.e.,* fixed-sized arrays of bits equipped with standard low-level machine instructions [16]).

The problem. The standard approach for satisfiability checking reduces the bitvector problem to a Boolean problem, leveraging the powerful SAT solving techniques and their conflict-driven clause learning (CDCL) mechanisms [21, 4]. Yet, this *bit-level approach* loses the structure of the initial bit-vector problem, disallowing any possible high-level simplification other than at preprocessing and, in some cases, yielding scalability

*Work partially funded by ANR under grants ANR-14-CE28-0020 and ANR-12-INSE-0002.

issues. Recent efforts in both the CP and SMT communities have demonstrated the potential benefits of *word-level reasoning* over bitvectors [1, 19, 6, 27, 8, 9] (where structural information "is not blasted into bits" [5]), but these works are mostly limited to the propagation step while the strength of modern SAT solvers rely on their learning mechanism [4]. Actually, two recent works [27, 28] do combine word-level propagation and learning for bitvectors, yet the learning is either bit-level [27] or assignment-dependent [28] in the vein of NoGood Learning [15].

Goal and challenge. Our goal is precisely to design a CDCL-like technique for the theory of quantifier-free bitvectors, such that it is truly word-level, and such that the learning stems from an interaction between the constraints involved in the conflict instead of a particular assignment to variables that lead to a conflict. This learning should be able to work together with recently developed word-level propagation techniques for bitvectors. Furthermore, in the context of verification where several theories can cooperate, it is also desirable that the technique can be combined with learning methods over other theories.

Proposal and contributions. We propose to lift bit-level learning mechanism to the word-level. This learning only makes sense if the decisions are allowed to be word-level, propagation is word-level, the conflict detection is word-level, the explanations are word-level and the sound rules of inference that derive knew knowledge from these explanations are also word-level. *Our very main contribution* is a CDCL-style, word-level learning over the theory of quantifier-free bitvectors, enjoying all good properties from the SAT-based CDCL: the learnt clauses *are part of the the theory of quantifier-free bitvectors theory*, they are *asserting*, they are *minimal* (only involve propagations that lead to the conflict and variables that actively participated in the conflict), and of course adding them is a *sound extension* of the original problem (they are implied by some of the original constraints). More specifically, to achieve this goal, we design the following original ingredients:

- Word-level explanations for propagations of logical and structural bitvector operations, through bitvector constraints, associated with masks indicating the positions that they fix in the variables (Sec. 3.1).
- A sound extension to word level of the Resolution rule, using bitvector variables and constraints (Sec. 3.2).
- An explanation selection algorithm that relies on bit-wise operations to minimize the number of learnt constraints (Sec. 3.3).
- Finally, our learning mechanism, being based on resolution, can be *combined* with resolution-based learning on other theories (notably arithmetic), allowing to learn also from simplifications (Sec. 3.4) used at propagation [6].

Discussion. This work paves the way to truly word-level decision procedures for bitvectors, taking full advantage of the word-level propagations recently designed in both CP [1, 19, 6, 27] and SMT communities [8, 9, 28]. The presentation of our technique is high-level. The many important design questions behind an efficient implementation (e.g., heuristics for clause deletion, using first versus last Unit Implication Point [25], explanation recording techniques [12] – forward, backward or clausal, etc.) are left as future work – but our current method is fully compatible with all of them.

2 Background

We briefly present the main characteristics of Conflict-Driven Clause Learning (CDCL), and give a high-level presentation of the theory of quantifier-free bitvectors.

2.1 Conflict-driven clause learning

A constraint is a formula dependant on some variables to which one wishes to find, if any, an assignment to the variables that makes the formula true. A clause is a special kind of constraint in the form of a disjunction of Boolean variables. We use the word "constraint" and "clause" interchangeably. A conflict is a state of affairs (empty domain for a variable, falsehood of an original constraint) that makes the problem unsatisfiable. Most solving algorithms follow a sequence: 1- propagate, 2- if there is a conflict then 3- do "some Process C", 4- else (once propagation is exhausted) make a decision (assign value to a variable, or restrict domain of a variable, or choose truth value of a constraint, etc.), then repeat. Originally, "Process C" was simply backtracking to the previous decision, but very early on, the problem of multi-occurring conflicts (or *trashing* [17]) lead to non-chronological backtracking algorithms. One of the most famous of these algorithm – the conflict-driven backjumping [23] – directly inspired [3] CDCL [21, 4], which deduces from a conflict some new facts (here: Boolean clauses) allowing to circumvent future similar conflicts, significantly short-cutting the search.

Learning techniques: requirements and properties. Learning algorithms are usually built on the following key elements [4]:

Explanations: to analyse the conflict, the solver needs to "know" the reasons for all the propagations that lead to that conflict. This implies that each propagation can generate some form of *explanation*, which is implied by the original set of clauses – either a theory lemma or some logical entailment (*explanation soundness*).

Selection: once a conflict is detected, the learning algorithm must be able to select, from those explanations, the ones that actively participated in the conflict.

Sound extension: to ensure that the learning algorithm only generates constraints (or clauses) that are implied by the original set of constraints, the relevant explanations that are selected are used to derive knew knowledge through *sound* rules of inference. In the case of CDCL, this rule is *Resolution* [24]. Thus adding a constraint that is a result of this inference is a *sound extension*.

Besides these three key elements a few other properties are also desirable. *(1) Theory inclusion*: if the learnt constraint is part of the theory of the original set of constraints, then there is no need to introduce extra machinery (any non-theory included constraint must be supplied with dedicated propagation, conflict detection, learning, etc.) in order to proceed with the solving. *(2) Asserting clause*: in order to shortcut the search, the added (learnt) constraint should force, immediately after (non-chronological) backtrack, a propagation that prevents the same decisions as those which generated the conflict. *(3) Minimality*: the number of the constraints added through learning can be significant, thus it is preferable to learn through information relevant to the conflict at hand, and nothing more, in order to minimize the size of the added knowledge. The selection algorithm should therefore discard what is not relevant to the conflict. However, if it is possible to avoid *more* conflicts (create more shortcuts) with the *same* additional space consumption (*e.g.,* the same constraint), that would be an advantage. *(4) Compositionality*: problems involving many theories are common and very useful to verification, it is therefore desirable that the different learning components of each theory can cooperate with each another. The learning we introduce here can naturally be used in a wider Resolution-based learning context.

2.2 Quantifier-free Bit-vector theory

Theory. A bitvector is a vector of binary values, 0 or 1 (as such, the representation of an integer can be considered a bitvector). The theory of quantifier-free bitvectors surrounds that data type with logical operations (bitwise disjunction ($|$), conjunction ($\&$), negation ($\div$), *etc.*), structural operations (shift ($\gg, \ll$), extraction ($\lfloor . \rceil_{i.j}$ with $j < i < \mathit{size} - 1$), concatenation ($::$), *etc.*), and modular arithmetic. A bitvector constraints is of the form $a\ \mathit{binop}\ b = c$, where *binop* is a binary operator, and of the form $\mathit{unop}\ a = b$, where *unop* is a unary operator. These constraints can be considered atomically or appear in a clause (a logical disjunction). We will present our algorithm in terms of unit clauses which atoms are bitvector constraints. Through compositionality with Boolean Resolution, however, it is possible to treat the wider clausal setting (Sec 3.2).

Bitvector domains inside a decision procedure. During a decision procedure, a bit in a given bitvector variable can have one of 4 states, depending on previous decisions and propagations: set (1), cleared (0), not yet known (?) and *conflicting* ($\perp$). A *conflicting bit* is a bit that was fixed (cleared or set) and that a propagation requires to be fixed to another value (resp. set or cleared). A *conflicting variable* is a variable containing at least one conflicting bit. In this paper, a conflict is detected when a conflicting bit is created, which is somewhat different from detecting conflict when a clause becomes unsatisfiable. Interestingly, these bitvector domains are shared by different word-level bitvector solvers from CP [1, 6, 27] and SMT [9, 28]. From a practical point of view, they can be implemented either by a list [1], a *run-length encoding* (a representation dependent on bit-alteration) [28] or (much more efficient) by a pair of integers, with a clever use of machine operations [19].

3 CDCL-inspired Word-level learning

Following the CDCL scheme (Section 2), we first introduce word-level explanations for word-level propagations over bitvectors (Sec. 3.1), we soundly extend resolution rules to bitvectors (Sec. 3.2), then we show how we analyse the conflict, select the explanations and derive new constraints from them through our extended resolution (Sec. 3.3). Finally, we extend the explanation language and illustrate the compositionality of learning through Resolution in order to encompass simplifications in the solver (Sec. 3.4).

3.1 Word-level explanations

We consider as a source for explanation the information propagated through the bitvector abstract domain (Sec. 2), as it is clear that word-level learning should be based on word-level decisions and propagations. Moreover, bitvector domains are already used in existing word-level bitvectors solvers from CP [1, 6, 27] and SMT [8, 9, 28].

The main idea here is to extend Boolean Resolution (the combination rule of CDCL) to bitvector. The first step is to find clause-like structures that are really bitvector constraints, yet still enjoying some of the properties of Boolean clauses. In the following, We reuse notations of bitvector domains from Section 2. We also use the usual notation b_i to mean the i^{th} bit of b (recall that indices start at 0 and that the least significant bit is the rightmost one).

Definition 3.1. *A word-level literal L (or simply literal when clear from context), when b is a bitvector variable, is given through the following rules:*

$$L ::= T \mid \overline{T} \qquad T ::= A \mid A \gg A \mid A \ll A \qquad A ::= b \mid \overline{b} \mid constant$$

where $\gg, \ll$ are logical shift operators. •

We say that a bitvector variable b *is set (resp. cleared) through* a list of literals $\mathcal{L}$ if its domain is set (resp. cleared) wherever (*i.e.,* on all positions where) the domains of *all* literals of $\mathcal{L}$ are set (resp. cleared), and we write $(\mathcal{L}) \rightharpoondown^1 b$ (resp. $(\mathcal{L}) \rightharpoondown^0 b$). For example, from a constraint $a \,\&\, b = c$, where a, b, c respectively have the abstract domains $[\![1??1]\!], [\![01??]\!], [\![??01]\!]$, we forward propagate into c, through $(c) \rightharpoondown^0 c$, clearing c_1, and $(a, b) \rightharpoondown^1 c$ setting c_0, and backward propagate through $(c) \rightharpoondown^1 a$ and $(c) \rightharpoondown^1 b$ setting a_2 and b_2, $(c, \overline{a}) \rightharpoondown^0 b$ clearing b_3. The constraint $a \gg n = b$ backward propagates using $(\overline{b \ll n}) \rightharpoondown^0 a$ and $(b \ll n) \rightharpoondown^1 a$.

Definition 3.2. *A word-level clause, or w-clause is of the form $L = \mathbb{1}$ where L is a bitwise disjunction of l-sized literals and $\mathbb{1}$ is a sequence of l ones (*i.e., *the bitvector that is set on every bit). By analogy with a Boolean clause, we write the constraint ($L = \mathbb{1}$) simply as the disjunction of literals L.* •

A w-clause is called a "propagation clause" when it is used in the explanation of a propagation. When a propagation takes place in the domain of a variable x through $(a^1, \cdots, a^n) \rightharpoondown^0 x$ (resp. $(b^1, \cdots, b^m) \rightharpoondown^1 x$), where a^i and b^i are (possibly shifted)

Table 1: Propagation constraints for &, $|$, $\oplus$, and shifts, where $\{\Diamond, \blacklozenge\} = \{\ll, \gg\}$

$x \mathbin{\&} y = z$	$x \mid y = z$	$x \oplus y = z$	$x \Diamond y = z$
$\overline{z} \mid x$	$\overline{x} \mid z$	$z \mid x \mid \overline{y}$	$\overline{z} \mid x \Diamond y$
$\overline{z} \mid y$	$\overline{y} \mid z$	$z \mid \overline{x} \mid y$	$z \mid \overline{x \Diamond y}$
$z \mid \overline{x} \mid \overline{y}$	$\overline{z} \mid x \mid y$	$\overline{z} \mid x \mid y$	$x \mid \overline{z \blacklozenge y}$
		$\overline{z} \mid \overline{x} \mid \overline{y}$	$\overline{x} \mid \overline{\overline{z} \blacklozenge y}$

literals, it can be explained through the w-clause $\overline{x} \mid a^1 \mid \cdots \mid a^n$ (resp. $x \mid \overline{b^1} \mid \cdots \mid \overline{b^m}$). Table 1 summarizes the w-clauses for the four main bitvector constraints.

Proposition 3.1 (Explanation soundness)**.** *The propagation clauses of Table 1 are implied by their respective bitvector constraint.*

Consider the (on the right) logical simplification using classical equivalencies and de Morgan rules of the bit-blasting encoding [16] (on the left) of the constraint $x \mathbin{\&} y = z$, where $v_0, \cdots, v_{l-1}$ are the Boolean variables representing the l long bitvector variable v:

$$\bigwedge_{i=0}^{l-1} ((x_i \wedge y_i) \longleftrightarrow z_i) \qquad \equiv \qquad \bigwedge_{i=0}^{l-1} ((\overline{x_i} \vee \overline{y_i} \vee z_i) \wedge (\overline{z_i} \vee x_i) \wedge (\overline{z_i} \vee y_i))$$

From this, it follows that each bitvector conjunction constraint implies three other constraints that are really w-clauses:

$$(x \mathbin{\&} y = z) \quad \Longrightarrow \quad (z \mid \overline{x} \mid \overline{y} = \mathbb{1}) \quad \wedge \quad (\overline{z} \mid x = \mathbb{1}) \quad \wedge \quad (\overline{z} \mid y = \mathbb{1})$$

A similar argument can be made for the other constraints of Table 1.

Note that *the same* w-clause can serve as a propagation explanation for *more than one* constraint. For example, $z \mid \overline{x} \mid y$ explains setting z wherever x is set and y is cleared from the constraints $z = x \mathbin{\&} \overline{y}$, $z = x \oplus y$, $x = y \mid z$, *etc.*

Explanation triples. To be useful, each explanation must record two pieces of information : *why* the propagation took place – the propagation w-clause, call it $\mathcal{C}$– and *when* it took place – by indicating the *decision level or depth* p of the propagation (the number of decisions that preceded this propagation).

In addition, because we deal with bitvector variables of which some bits may be *unaffected* by the propagation, we need a third piece of information m we call *impact mask* which is set for those bits of the variable that *were unknown before* and that were cleared or set through the propagation.

Definition 3.3. *An explanation triple* $(p, m, \mathcal{C})$ *explains the propagation at decision level* p *on a variable along the positions set in the mask* m *using the propagation clause* $\mathcal{C}$*. The set of explanations of a variable* x *is written* $\mathcal{E}(x)$*. We say that an explanation covers a bit (or a position) if its impact mask is set on that position.* •

We emphasize the following remarks. Firstly, because the impact mask only indicates the bits that were unknown and became fixed, *no two explanations can cover the same bit*: the order in which the propagations are done imposes this separation. For example, from the two constraints $e \ \& \ c = a, f \mid b = c$, with $a = [\![1110]\!]$, $b = [\![0111]\!]$ and e, f, c are unknown, at a decision level p, both constraints can set the two middle bits of c, through the propagation w-clauses $\overline{a}|c$ or $\overline{b}|c$ respectively. However, only one of them (the first to "fire") will be in an explanation that covers the two middle positions, *i.e.,* either $\{(p, \overline{a}|c, 1110), (p, \overline{b}|c, 0001)\} \subseteq \mathcal{E}(c)$ or $\{(p, \overline{a}|c, 1000), (p, \overline{b}|c, 0111)\} \subseteq \mathcal{E}(c)$.

Secondly, consider the constraint $C : x \mid y = z$, where x and z are unknown, $y = [\![00??]\!]$. After a decision at level p on some other variable, the propagations can follow this ordered sequence: x gets cleared by some other constraint ($x = [\![0000]\!]$), which clears z on some bits ($z = [\![00??]\!]$) through C, then y gets cleared by some other constraint ($y = [\![0000]\!]$), and then z gets completely cleared ($z = [\![0000]\!]$). Notice that the explanations for z will both have the same propagation w-clause: $(p, 1100, x \mid y \mid \overline{z}), (p, 0011, x \mid y \mid \overline{z})$. We choose to simplify such instances, *i.e.,* same propagation w-clause occurring *at the same decision level*, to $(p, 1111, x \mid y \mid \overline{z})$ by disjuncting their impact mask. Thus there is always a unique pair (p, C) for each decide depth and propagation w-clause in the set of explanations of a given variable.

3.2 Word-level resolution for w-clauses

A sequence of resolution steps involving disjoint set of clauses and variables can be carried out in parallel. Because a w-clause is essentially a group of Boolean clauses, they can subject to a word-level Resolution rule. Indeed, l resolution steps between clauses involving the Boolean units a_i, b_i, c_i (for $i = 0..l-1$) can be considered as a word-level rule of Resolution between w-clauses involving $l-$bit long variables a, b and c.

$$\frac{a_0 \vee c_0 \qquad \overline{c_0} \vee b_0}{a_0 \vee b_0}, \frac{a_1 \vee c_1 \qquad \overline{c_1} \vee b_1}{a_1 \vee b_1}, \cdots, \frac{a_{l-1} \vee c_{l-1} \qquad \overline{c_{l-1}} \vee b_{l-1}}{a_{l-1} \vee b_{l-1}} \equiv \frac{a|c \qquad \overline{c}|b}{a|b}$$

Proposition 3.2 (Soundness of inference)**.** *Word-level Resolution is a sound extension of binary Resolution.*

Word-level resolution on a shifted literal is less immediate. Consider that a shift constraint $a \gg n = b$ is equiprovable with the Boolean encoding $(\bigwedge_{i=0}^{l-1-n} a_{n+i} \longleftrightarrow b_i) \wedge (\bigwedge_{i=l-n}^{l-1} 0 \longleftrightarrow b_i)$, which implies $(\bigwedge_{i=0}^{l-1-n} a_{n+i} \vee \overline{b_i}) \wedge (\bigwedge_{i=l-n}^{l-1} \overline{b_i})$, which in turn is equivalent and justifies the propagation w-clause $a \gg n \mid \overline{b}$. The resolution can be extended to bitvector constraints involving shift. One can, for example resolve $a \gg n \mid \overline{b}$ with $\overline{a} \mid c$, whose encoding is $(\bigwedge_{i=0}^{l-1} \overline{a_i} \vee c_i)$ to obtain $c \gg n \mid \overline{b}$. Indeed, this bitvector resolution can be simply seen as a group of n Boolean resolution steps between $\overline{a_{n+i}} \vee c_{n+i}$ and $a_{n+i} \vee \overline{b_i}$, for $i = 0..l-1-n$, obtaining $(\bigwedge_{i=0}^{l-1-n} c_{n+i} \vee \overline{b_i})$, which in conjunction with $(\bigwedge_{i=l-n}^{l-1} \overline{b_i})$ is equivalent to $c \gg n \mid \overline{b}$. More generally we can use the resolution rules:

$$\frac{A \mid (y \gg n) \quad B \mid \overline{y}}{A \mid (B \gg n)} \qquad\qquad \frac{A \mid (y \ll n) \quad B \mid \overline{y}}{A \mid (B \ll n)}$$

The word-level resolution rules acting on two shifted literals are justified in the same way. Of course, such a resolution only makes sens if the two shifts preserve some bits that are common to (that come from the same position of) the two literals, for example the literals $y \ll n$ and $\overline{y} \gg m$, such that $n + m \geq \mathit{size(y)}$ have no bits in common. Fortunately, the selection algorithm presented in the next section, by construction, only selects w-clauses for which resolution is justified. The two literals can either have the same shift or opposing shift. The rules to apply depend on the length of each shift. In the following, $\{\lozenge, \blacklozenge\} = \{\ll, \gg\}$:

$$\frac{A \mid (y \lozenge n) \quad B \mid (\overline{y} \lozenge m) \quad n > m}{A \mid ((B \blacklozenge m) \lozenge n} \qquad \frac{A \mid (y \lozenge n) \quad B \mid (\overline{y} \lozenge m) \quad n \leq m}{B \mid ((A \blacklozenge n) \lozenge m}$$

Finally, resolution on two opposing shifted literals is as follows:

$$\frac{A \mid (y \gg n) \quad B \mid (\overline{y} \ll m) \quad n > m}{A \mid (B \gg (m+n)) \mid (2^m - 1) \ll (l - (n+m))} \qquad \frac{A \mid (y \gg n) \quad B \mid (\overline{y} \ll m) \quad n \leq m}{B \mid (A \ll (m+n)) \mid (2^n - 1) \ll m}$$

notice that these rules subsume the above Resolution rules (if $n = 0$ or $m = 0$).

The logical and structural operations on bitvectors all can be propagated using the basic blocks of conjunction, disjunction, negation and shift. For example, concatenation: $a :: b = (a \ll \|b\|) \mid b$, extraction: $\lfloor a \rceil_{i.j} = (a \gg j) \ \& \ \overline{(-1) \ll (i - j + 1)}$, in addition to involutive negation of these operations: $\overline{a :: b} = \overline{a} :: \overline{b}$ and $\overline{\lfloor a \rceil_{i.j}} = \lfloor \overline{a} \rceil_{i.j}$. If the literal's syntax is extended to include extraction and concatenations, one can also use the derivable rules of inference:

$$\frac{A \mid \lfloor x \rceil_{i.j} \quad \overline{x} \mid B}{A \mid \lfloor B \rceil_{i.j}} \qquad \frac{A \mid x :: C \quad \overline{x} \mid B}{A \mid B :: C} \qquad \frac{A \mid C :: x \quad \overline{x} \mid B}{A \mid C :: B}$$

However, for simplicity, we will present the conflict analysis and learning only through the basic blocks.

Normalizing. In order to obtain the actual w-clauses, we normalize using (non-exhaustively listed) rewrite rules such as:

$$\begin{array}{ll}
(A \# B) :: C \longmapsto (A :: C) \# (B :: C) & \lfloor (A \# B) \rceil_{i.j} \longmapsto \lfloor A \rceil_{i.j} \# \lfloor B \rceil_{i.j} \\
C :: (A \# B) \longmapsto (C :: A) \# (C :: B) & A \ \& \ B \longmapsto A \wedge B \\
(A \# B) \lozenge n \longmapsto A \lozenge n \# B \lozenge n & \overline{A \lozenge n} \longmapsto (2^n - 1) \blacklozenge (\|A\| - n) \mid \overline{A} \lozenge n
\end{array}$$

where $\# \in \{ \ \& \ , | \}$ and $\{\lozenge, \blacklozenge\} = \{\ll, \gg\}$.

Proposition 3.3. *The normalization rewrite rules form a terminating rewrite system.*

Sketch of proof: By decreasing distance between the root and the logical connectives.

Mixing theories. While this paper is concerned with purely bitvector constraints, it is worth noting that the above bitvector resolution rules can combine with usual resolution.

$$\frac{F \vee (a \mid c) \quad (\overline{c} \mid b) \vee G}{F \vee (a \mid b) \vee G}$$

In section 3.4, we will use this fact to explain simplifications and factorizations, and include them in our analysis.

Theorem 3.1 (Learning soundness)**.** *Any constraint obtained from the word-level resolution of w-clause (and Boolean resolution) is implied by the original problem and therefore constitutes a sound extension of it.*

3.3 Word-level Conflict analysis

We recall that a conflict is detected when a conflicting variable has a conflicting bit (Section 2.2). We now show how word-level analysis is carried out, isolating the explanations for that conflict from the irrelevant explanations, resulting in conflict-driven word-level learning that is expressed inside the theory of quantifier-free bitvectors, *i.e.,* through bitvector constraints.

When the propagation of a constraint requires to set (resp. clear) one or more bits of a variable that were previously cleared (resp. set), a conflict is detected on those bits. A mask that is set on those conflicting positions is called a *conflict mask*. For example, propagating from $a \ \& \ b = c$, with current domains $a = [\![?1111111]\!]$, $b = [\![1?111111]\!]$ and $c = [\![00?00000]\!]$ may generate a conflict through forward propagation resulting in $c = [\![001\bot\bot\bot\bot\bot]\!]$ or through backward propagation, resulting $a = [\![011\bot\bot\bot\bot\bot]\!]$ or $b = [\![101\bot\bot\bot\bot\bot]\!]$, in all these cases the conflict mask is 00011111 and the aborted propagation clause is $c \mid \overline{b} \mid \overline{a}$. This w-clause is called *conflict w-clause.*

This *grouped* detection exploits the word-level propagation as opposed to the conflict detection done through bit-blasting, one bit at a time. The main observation is the following: if *several* conflicting bits are generated through *the same propagations from the same variables*, then there is no need to learn *more than one* bitvector constraint for those bits. In general more than one w-clause is learnt, possibly in parallel, after a single conflict.

Main idea. For simplicity, we only discuss the algorithm for non-shifted literals (*i.e.,* variables) and we will then extend it. The algorithm starts from the conflict at decide level p caused by a propagation clause R on the positions set in a conflict mask m of some variable. The algorithm could be seen as applying Boolean CDCL in parallel on all the set bits in the conflict mask m. At some points the bits could have different explanations and so the mask would be split and the algorithm continues on two different inputs. Seeing each bit in isolation should be reminiscent of Boolean CDCL: each bit b_i of each variable b of R, such that the i^{th} bit of m is set, is necessarily fixed (a propagation through a w-clause on that bit has the same behaviour as a unit propagation through a Boolean clause: all other variables in the clause must be fixed). We call such bits – that participated in the conflict – the *culprit bits* of b. The values of the culprit bits were either fixed before p (pre-p) or through a propagation at p (@p). In the former case, there are no @p explanations that cover b_i ; in the latter case, there is exactly one explanation that covers it. We define the *explanation equivalence* ($=^{\mathcal{E}}$) of bits b_i and b_j such that $b_i =^{\mathcal{E}} b_j$ if they are either both pre-p or are both covered by the same @p explanation in $\mathcal{E}(b)$. This equivalence relation is the one that fixes the number of eventually learnt clauses: there is a new clause each time there is more than one equivalence class on the bits of a variable.

Definition 3.4. *A* reason R *is a w-clause that serves as an accumulator and is initialized with the conflict w-clause. Each reason is associated with an* analysed mask m*, initialized by the conflict mask. For each equivalence class* $E^{=}$ *of the culprit bits of each variable in* R*, a separate copy of* R *is created and associated with a new analysed mask which is set on the position of the bits in* $E^{=}$*. Once a variable has been investigated in this manner, it is* marked*, such that it is purged from future resolution steps of the algorithm. We call a* leaf *a reason whose literals are all marked (thus it is fully investigated). At the end of the analysis, all leaves are learnt. Finally, we call* set of reasons *the set of tuples* $(p, R, m^a, \mathcal{M})$ *containing the reason* R*, associated analysed mask* m^a*, decide depth* p*, with marked literals* $\mathcal{M}$*.* •

Algorithm 1 *analyse*$(p, R, m^a, \mathcal{M})$ (reason R, analysed mask m^a, decide depth p, with marked literals $\mathcal{M}$)

---

if $\exists x \in R, x \notin \mathcal{M}$ **then**
    if $\exists (p, C, mi) \in \mathcal{E}(x)$ s.t. $mi \mathbin{\&} m^a \neq 0$ **then**
        if $mi \mathbin{\&} m^a = m^a$ **then**
            return $\{(p, \mathit{Resol(R,C,x,}\mathcal{M}\mathit{)}, m^a, \{x\} \uplus \mathcal{M})\}$
        else
            $m^{in} = mi \mathbin{\&} m^a$
            $m^{out} = m^a \oplus m^{in}$
            $S = \mathit{Resol(R,C,x,}\mathcal{M}\mathit{)}$
            return $\{(p, S, m^{in}, \{x\} \uplus \mathcal{M}), (p, R, m^{out}, \mathcal{M})\}$
        end if
    else
        return $\{(p, R, m^a, \{x\} \uplus \mathcal{M})\}$
    end if
else
    return $\{\mathit{leaf(R)}\}$
end if

Conflict analysis algorithm. After a conflict, the set of reasons starts as a singleton containing the tuple formed with the conflict propagation R, the conflict mask m, the current decision level p and an empty set that will hold the marked literals, *i.e.*, $\{(p, R, m, \{\})\}$. The (parallelizable) algorithm consists of the repeated rewriting of the set of analysed reasons until all its elements become leaves, at which point they are all learnt.

$$\{(p, R, m^a, \mathcal{M})\} \uplus \mathcal{S} \longmapsto \mathit{analyse}(p, R, m^a, \mathcal{M}) \uplus \mathcal{S}$$

For now, we voluntarily omit shifted literals in Algorithm 1, (we will see shortly how to easily extend it for shifted literals). We comment this algorithm through the following example, where, at decide level (i-1), values are $c =$??11, $v = 1111$, $e = 0000$, $g = 00??$,

h =??00, and unknown for the rest.

$$(1.)d = \overline{c} \mid x \qquad (2.)e = b \oplus f \qquad (3.)f = \overline{d} \ \& \ v$$
$$(4.)g = f \ \& \ a \qquad (5.)h = f \ \& \ a \qquad (6.)a = b \ \& \ c$$

Then, at level i, we decide $d = 0000$. The following table lists the explanations stemming from that decision. We omit the decide level from the triples (all explanations are of the current decision level i) and replace it, for readability, with the index of the constraint that spawned the propagation they explain.

f	b	c	a
(3, $f \mid d \mid \overline{v}$, 1111)	(2, $b \mid e \mid \overline{f}$, 1111)	(1, $c \mid d$, 1100)	(4, $g \mid \overline{f} \mid \overline{a}$, 1100) (5, $h \mid \overline{f} \mid \overline{a}$, 0011)

At this point the 6^{th} constraint tries to propagate with w-clause $a \mid \overline{b} \mid \overline{c}$ and provokes a conflict on all bits. Conflict analysis then begins with the initial set of analysed reasons containing the tuple $\{(i, a \mid \overline{b} \mid \overline{c}, 1111, \{\})\}$.

Unfolding the algorithm. The algorithm selects the literal a (Line 1), and the first selected explanation for a is $(i, g \mid \overline{f} \mid \overline{a}, 1100)$ since 1100 & 1111 $\neq 0$. The test of Line 3 is not satisfied however because the impact mask of the propagation clause does not cover all the conflict mask. In the *else* branch, two masks are created: one, m^{in}, will be the analysed mask of the reason resulting from the resolution of the propagation clause and the current reason, the other, m^{out}, represent the conflict bits that must be explained elsewhere. The algorithm returns two tuples: $\{(i, g \mid \overline{f} \mid \overline{b} \mid \overline{c}, 1100, \{a\}), (i, a \mid \overline{b} \mid \overline{c}, 0011, \{\})\}$. The former will be rewritten along the left branch of Figure 1, the latter – where a is *not marked* – will select a again and this time chose the all covering explanation that comes from the 5^{th} constraint. Once there are no more variables to select from a reason (*i.e.,* all variables are marked, Line 11), that reason is marked as a leaf. Interestingly, because the learnt constraint has a clausal form, it serves as its own propagation clause.

Figure 1 summarizes the learning through conflict analysis. The nodes are the reasons and the analysed mask (with the root being the initial pair). Each edge labelled with a variable name represents a resolution step on that variable between the parent reason and the propagation clause of the explanation of that variable that covers the resulting analysed mask.

Notice how $\overline{c}$ stays only in one of the learnt w-clauses, because the right analysed mask is not covered by any current decision-level explanation of c, meaning that *the part* of c that *participated* in the conflict *predates* the decision, and indeed, initial value for c was ??11. The two learnt clauses propagate each 2 bits into d, avoiding the conflict.

This learning through interaction of constraints, independent of a particular assignment, has the advantageous side effect of preventing word-level conflicts from potentially bit-level conflicts, *i.e.,* even a conflict mask with a single set bit will result in a learnt w-clause that impacts all positions of its variables, thus avoiding *more than that exact* conflict with the *same amount of added* space.

$$\begin{tikzcd}[column sep=small]
& a \mid \bar{b} \mid \bar{c}\ ,\ 1111 \arrow[dl, "a"'] \arrow[dr, "a"] & \\
g \mid \overline{f} \mid \bar{b} \mid \bar{c},\ 1100 \arrow[d, "b"] & & h \mid \overline{f} \mid \bar{b} \mid \bar{c},\ 0011 \arrow[d, "b"] \\
g \mid \overline{f} \mid e \mid \bar{c},\ 1100 \arrow[d, "f"] & & h \mid \overline{f} \mid e \mid \bar{c},\ 0011 \arrow[d, "f"] \\
g \mid d \mid \overline{v} \mid e \mid \bar{c},\ 1100 \arrow[d, "c"] & & h \mid d \mid \overline{v} \mid e \mid \bar{c},\ 0011 \\
g \mid d \mid \overline{v} \mid e,\ 1100 & &
\end{tikzcd}$$

Figure 1: Conflict analysis through rewriting and resolution

Different strategies. The order in which the analysed reasons are rewritten by *analyse* can be tuned to obtain certain known strategies such as First-UIP or Last-UIP [25]. The determined strategy can be implemented by the order induced by a trail.

Treating structural operations. A slight addition to the above algorithm allows to treat shift operations. If the selected literal appears shifted by n, then the search for an explanation among the literal (line 3) uses the same shift of the impact mask. Line 1's test becomes $\exists a \gg n \in R$, and line 2's test becomes "s.t $(mi \gg n)$ & $m^a \neq 0$". Other structural operations are also adapted accordingly, or supported through the rewriting that uses shift operations and logical operations.

Taming non-asserting learnt clauses. To be useful, a learnt clause must be *asserting*, *i.e.,* force a propagation after a backtrack (or a back-jump) so as to avoid the same conflict. With our method, this is not always the case. Consider the example on 2-bits variables given on the left of Figure 2, where current assignment for a, b, c, v is respectively $0?, 0?, ?1, 00$. Then we decide that $d = 11$, we get (omitting impact mask and decide depth): from 3 that $a = 00$ with the propagation clause $\overline{a} \mid v \mid \overline{d}$; from 2, that $b = 01$ with $b \mid \overline{d \gg 1}$; from 1, we get $a = 01$ with $a \mid \bar{c} \mid \bar{b}$: conflict with mask 01. Then resolution gives us the clausal constraint $\bar{c} \mid v \mid \overline{d} \mid \overline{d \gg 1}$, the conclusion of the derivation on the right of Figure 2. After backtracking and applying the values for c and v, we get $?0 \mid \overline{d} \mid \overline{d \gg 1}$ which does not propagate. Indeed, if we look at the level of bits, this gives the clauses $\overline{c_1} \vee \overline{d_1}$ and $\overline{d_0} \vee \overline{d_1}$.

1. $a = c$ & b 2. $b = d \gg 1$ 3. $v = a$ & d	$\dfrac{b \mid \overline{d \gg 1} \qquad \dfrac{a \mid \bar{c} \mid \bar{b} \quad \bar{a} \mid v \mid \bar{d}}{\bar{c} \mid \bar{b} \mid v \mid \bar{d}}}{\bar{c} \mid v \mid \bar{d} \mid \overline{d \gg 1}}$

Figure 2: Non-asserting clauses justify partial decision on variables

The above clause provokes a problem because it relates *different* positions in the *same* variable (the variable d above appears both as a literal with and without shift). This sort of problem arises because a decision is allowed to be made on many bits of the same variable, unlike in bit-blasting where decisions are made on Boolean variables.

To account for these problematic cases, the constraint solver must include in the bitvector domain representation some indication that the incriminated variable (the one which bits are related by the clause, here d) can only be subject to partial decisions. This extra information can be a simple flag that, if set, forces a naive bit-by-bit decision, but a more sophisticated information can be generated using the conflict mask to derive decision masks associated to the variable, whereby grouped decisions are allowed on the bits indicated by that mask for that variable. *This way, the clause becomes asserting*, in the sense that the domain representation of the variable has been propagated in a way that prevents the conflict from arising again.

3.4 Extended explanations for learning from simplifications

The explanations presented so far fit well bitvector domain propagation, yet the solving process can be greatly improved by simplifications and factorizations [6]. We explain how the learning presented in this paper can be naturally extended to these extra propagations.

Extended explanations. We use $C \overset{\varepsilon}{\rightsquigarrow} E$ to mean that the constraint C can generate the explanation containing the propagation clause E and omit decide depth. We can now explain using regular Boolean clauses as well, where atoms are formed through bitvector constraints, equalities or disequalities. Some selected simplifications and factorizations are as follows (we consider the equality constraint $a = 0$ and the bitvector literal $\overline{a}$ to be equivalent):

- A logical shift constraint $a = a \gg c$ implies the nullity of a if $c > 0$. We allow each logical shift constraint to generate an explanation for such an update:
$$a_1 = a_2 \gg c \quad \overset{\varepsilon}{\rightsquigarrow} \quad (a_1 = a_2) \wedge (c > 0) \rightarrow (a_1 = a_2 = 0)$$
- A disjunction constraint $a = b_1 \mid b_2$ becomes an equality between a and b_1 when b_2 becomes 0, and *vice versa*, with explanations:
$$a = b_1 \mid b_2 \quad \overset{\varepsilon}{\rightsquigarrow} \quad b_i = 0 \rightarrow (a = b_j) \ \textit{with}\ \{i, j\} = \{1, 2\}$$
- The exclusive disjunction constraint $a = b \oplus c$ is equivalent to any permutation of its variables, and two $\oplus$ constraints with 2 common variables prompt an equality between their third respective variables, with the following explanation – where x, y, z (resp. x', y', z') is a permutation of a, b, c (resp. a', b', c'):
$$a = b \oplus c \wedge a' = b' \oplus c' \quad \overset{\varepsilon}{\rightsquigarrow} \quad (x = x') \wedge (y = y') \rightarrow (z = z')$$

The basic idea is then to combine our word-level resolution together with regular Boolean resolution in order to derive learnt clauses.

Example. We now give an example where, to keep it simple, the explanations cover all the conflict mask, such that only one clause is learnt in the end.

(1.)$x = x \gg y$	(2.)$z = x \mid t$	(3.)$z = a \oplus b$	(4.)$b = c \oplus t$
(5.)$a = c \gg y$	(6.)$a = v \ \& \ m$	(7.)$\overline{a} = v \mid n$	

We suppose current known values are $m = 1111$ and $n = 0000$, and we decide on $y = 0011$, (or 3, in decimal notation). Then, propagation steps are: $x = 0$ from 1 because $y = 3 > 0$, $z = t$ from 2 because x turned into the neutral element, then from 3 and 4 get $a = c$, and from that and $y = 3 > 0$ get $a = 0$, finally, through bitvector propagation $a = v \mathbin{\&} m$ propagates that $v = 0$ then $\overline{a} = v \mid n$ attempts to change that to $v = 1111$: conflict. The resolution derivation that allows learning is shown in Figure 3 (the numbers in the leaves are those of the constraints that prompted the respective explanation clause). This derivation uses both bitvector word-level resolution (marked with a double horizontal line) and regular Boolean resolution (marked with a single horizontal line). The conclusion of that derivation, *i.e.,* the learnt clause $(\overline{m}|n) \vee \neg(y > 0)$ (combining both bitvector and arithmetic atoms) excludes any value strictly greater than 0 from the concretization of y in the (current) event that $n = 0$ and $m = 1111$.

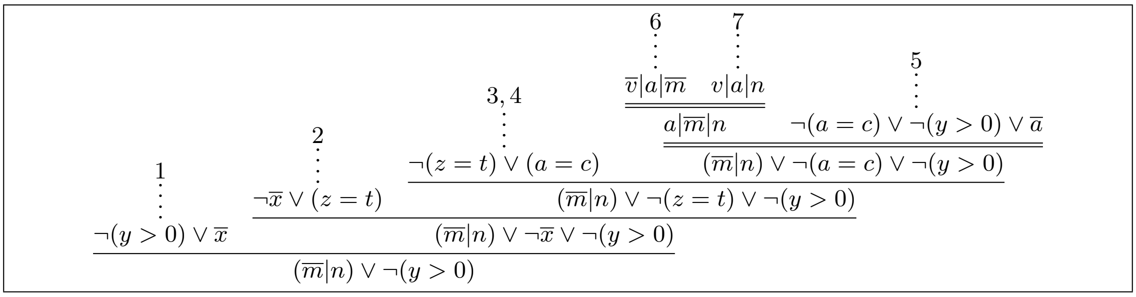

Figure 3: Learning through a combination of bitvector- and Boolean- resolution

4 Related work

We have already discussed at length the general CDCL approach [21, 4] as well as its straightforward instantiation to bit-level learning over bitvectors in Sections 1 and 2. We present now a few other learning methods from Constraint Programming (CP) – that can be in some ways adapted to world-level bitvector learning , as well as the very few learning methods dedicated to bitvectors.

Learning in Constraint Programming. There is a number of published work exploring learning capabilities in the context of a CP solver. *Lazy Clause Generation* [22] converts constraint propagators into clauses and feeds them to a SAT solver, showing significant results for scheduling problems. We are not aware of any treatment for bitvectors in this context but suspect that any such treatment would resort to bit-blasting (necessary to use a SAT solver in the background), which is precisely what we are trying to avoid. *(Generalized) NoGood Learning* techniques [10, 15] improve backtracking search in the context of Constraint Satisfaction Problems. Basically, (Generalized) NoGood learning allows to add partial assignments and non-assignments that are known to lead to conflict. This method extends naturally to word-level learning for bitvectors, yet our own approach is more general, as our learnt constraints do not mention particular assignments, they are solely derived from the relationship between the propagation clauses of the constraints that lead to a conflict. The HaifaCSP solver [26] is able to learn general constraints through a resolution-inspired method in a more general and concise way than nogood assignments. The approach uses a non-clausal learning that relies on direct inference between the constraints themselves as well as SAT inspired techniques, notably a similar heuristic to VSIDS [20] for variable ordering. It currently handles many of the constraints appearing in modern CP solvers, yet bitvector constraints are not handled. Being resolution-based able to handle integer constraints, this technique is a good candidate to test the limits of the ideas presented in this paper for theory learning cooperation.

Learning over bitvectors. There are two most closely related papers, mentioned in the introduction. First, Wang *et al.* describes a solver [27] that uses the same word-level propagations as we do but delegates bit-level learning to a SAT solver. Focusing on the learning scheme, which is the subject of our paper, the comparison between our work and this one is the same as with any bit-blasting learning approach.

The closest related work is a model constructing SAT solver [9] whose learning mechanism was extended [28] to the theory of quantified free bit-vectors, named mcBV and avoiding bit-blasting. The learning mechanism of mcBV resembles nogood learning in that it starts from a particular failing assignment. The difference is in mcBV's generalization method that starts from that assignment and weakens it as much as possible through successive flipping of bits from known to unknown. In contrast to our learning, which directly involves interactions between propagators, this learning is still tied to a particular assignment. Finally, the mcBV approach's handling of bounded arithmetic makes it a good candidate for combination with our approach.

One can also find a relation to the lazy layered [5] treatment of the theory of quantifier-free bitvectors, where bit-blasting is only invoked if necessary (the bits of a bitvector are all considered equal until those bits are affected differently – through a propagation or through a conflict – at which moment that vector is only blasted as much as necessary). Learning in that setting depends on where bit-blasting is done and is still, in essence, Boolean CDCL, even if each Boolean variable can potentially represent all bits of a variable (since those bits must be equal).

5 Conclusion

This paper presents a word-level CDCL-style learning mechanism for the theory of bitvectors, which is of paramount importance in software verification. The technique can be integrated in any world-level bitvector solver, whatever the underlying technology (CP [6] or natural-domain SMT [8]), as long as the deduction mechanism produces the required domain-based explanations and conflict detection. This work illustrates the effect of cross-fertilization between the CP and SMT community that brought forth many fruitful advances, most notably CDCL itself.

The immediate future work is, of course, the implementation of our ideas in a word-level solver for practical evaluation and the fine-tuning of all practical ingredients of an efficient CDCL-style learning mechanism, including clause deletion, variable ordering, and other learning-related heuristics.

References

[1] S. Bardin, P. Herrmann, and F. Perroud. "An Alternative to SAT-Based Approaches for Bit-Vectors". In: *TACAS*. 2010.

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. 2009.

[3] R. J. Bayardo Jr and R. Schrag. "Using CSP look-back techniques to solve real-world SAT instances". In: *AAAI/IAAI*. 1997.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh. "Conflict-driven clause learning SAT solvers". In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009).

[5] R. Bruttomesso et al. "A Lazy and Layered SMT (\ mathcal {BV}) Solver for Hard Industrial Verification Problems". In: *International Conference on Computer Aided Verification*. Springer. 2007.

[6] Z. Chihani, B. Marre, F. Bobot, and S. Bardin. "Sharpening Constraint Programming approaches for Bit-Vector Theory". In: *CPAIOR*. 2017.

[7] E. M. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *TACAS*. 2004.

[8] S. Cotton. "Natural domain SMT: A preliminary assessment". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2010.

[9] L. De Moura and D. Jovanović. "A model-constructing satisfiability calculus". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2013.

[10] R. Dechter. "Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition". In: *Artificial Intelligence* 41.3 (1990).

[11] E. W. Dijkstra. *A discipline of programming*. Vol. 1. Prentice-Hall Englewood Cliffs, 1976.

[12] T. Feydy, A. Schutt, and P. Stuckey. "Semantic learning for lazy clause generation". In: *TRICS workshop, held alongside CP*. 2013.

[13] P. Godefroid. "Test Generation Using Symbolic Execution". In: *FSTTCS*. 2012.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy abstraction". In: *POPL*. 2002.

[15] G. Katsirelos and F. Bacchus. "Generalized nogoods in CSPs". In: *AAAI*. Vol. 5. 2005.

[16] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* 1st ed. Springer Publishing Company, Incorporated, 2008.

[17] A. K. Mackworth. "Consistency in networks of relations". In: *Artificial intelligence* 8.1 (1977).

[18] K. L. McMillan. "Lazy Abstraction with Interpolants". In: *CAV*. 2006.

[19] L. D. Michel and P. Van Hentenryck. "Constraint Satisfaction over Bit-Vectors". In: *CP*. 2012.

[20] M. W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Design Automation Conference, DAC*. 2001.

[21] M. W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Design Automation Conference, DAC*. 2001.

[22] O. Ohrimenko, P. J. Stuckey, and M. Codish. "Propagation = Lazy Clause Generation". In: *CP*. 2007.

[23] P. Prosser. "Hybrid algorithms for the constraint satisfaction problem". In: *Computational intelligence* 9.3 (1993).

[24] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *JACM* 12 (1965).

[25] J. P. M. Silva and K. A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". In: *IEEE Trans. Computers* 48.5 (1999).

[26] M. Veksler and O. Strichman. "Learning general constraints in CSP". In: *Artificial Intelligence* 238 (2016).

[27] W. Wang, H. Søndergaard, and P. J. Stuckey. "A Bit-Vector Solver with Word-Level Propagation". In: *CPAIOR*. 2016.

[28] A. Zeljić, C. M. Wintersteiger, and P. Rümmer. "Deciding bit-vector formulas with mcSAT". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2016.