Solving bitvectors with MCSAT: explanations from bits and pieces

Stéphane Graham-Lengrand, Dejan Jovanović, and Bruno Dutertre

SRI International, USA

Abstract. We present a decision procedure for the theory of fixed-sized bitvectors in the MCSAT framework. MCSAT is an alternative to CDCL(T) for SMT solving and can be seen as an extension of CDCL to domains other than the Booleans. Our procedure uses BDDs to record and update the sets of feasible values of bitvector variables. For explaining conflicts and propagations, we develop specialized word-level interpolation for two common fragments of the theory. For full generality, explaining conflicts outside of the covered fragments resorts to local bit-blasting. The approach is implemented in the Yices 2 SMT solver and we present experimental results.

1 Introduction

Model-constructing satisfiability (MCSAT) [21, 22, 26] is an alternative to the CDCL($\mathcal{T}$) scheme [30] for Satisfiability Modulo Theories (SMT). While CDCL($\mathcal{T}$) interfaces a CDCL SAT solver [25] with black-box decision procedures, MCSAT integrates first-order reasoning into CDCL directly. Like CDCL, MCSAT alternates between search and conflict analysis. In the search phase, MCSAT assigns values to first-order variables and propagates unit consequences of these assignments. If a conflict occurs during search, e.g., when the domain of a first-order variable is empty, MCSAT enters conflict analysis and learns an explanation, which is a symbolic representation of what was wrong with the assignments causing the conflict. As in CDCL, the learned clause triggers backtracking from which search can resume. Decision procedures based on MCSAT have demonstrated strong performance in theories such as non-linear real [26] and integer arithmetic [21]. These theories are relatively well-behaved and provide features such as quantifier elimination and interpolation—the building blocks of conflict resolution in MCSAT.

We describe an MCSAT decision procedure for the theory of bitvectors ($\mathcal{BV}$). In contrast to arithmetic, the complexity of $\mathcal{BV}$ in terms of syntax and semantics, combined with the lack of word-level interpolation and quantifier elimination, makes the development of $\mathcal{BV}$ decision procedures (MCSAT or not) very difficult. The state-of-the art $\mathcal{BV}$ decision procedures are all based on a "preprocess and bitblast" pipeline [14,24,27]: they reduce the $\mathcal{BV}$ problems to a pure SAT problem by reducing the word-level semantics to bit-level semantics. Exceptions to the bitblasting approach do exist, such as [5, 18], which generally do not perform as well as bitblasting except on small classes of crafted examples, and the MCSAT approach of [32], which we discuss below and in the conclusion.

An MCSAT decision procedure must provide two theory-specific reasoning mechanisms.

First, the procedure must maintain a set of values that are feasible for each variable. This set is updated during the search. It is used to propagate variable values and to detect a conflict when the set becomes empty. Finding a suitable representation for domains is a key step in integrating a theory into MCSAT. We represent variable domains with Binary Decision Diagrams (BDDs) [6]. BDDs can represent any set of bitvector values. By being canonical, they offer a simple mechanism to detect when a domain becomes a singleton—in which case MCSAT can perform a theory propagation—and when a domain becomes empty–in which case MCSAT enters conflict analysis. In short, BDDs offer a generic mechanism for proposing and propagating values, and for detecting conflicts. In contrast, previous work by Zeljić et al. [32] represents bitvector domains using *intervals* and *patterns*, which cannot represent every set of bitvector values precisely; they over-approximate the domains.

Second, once a conflict has been detected, the procedure must construct a symbolic explanation of the conflict. This explanation must rule out the partial assignment that caused the conflict, but it is desirable for explanations to generalize and rule out larger parts of the search space. For this purpose, previous work [32] relied on incomplete abstraction techniques (replace a value by an interval; extend a value into a larger set by leaving some bits unassigned), and left open the idea of using interpolation to produce explanations. Instead of aiming for a uniform, generic explanation mechanism, we take a modular approach. We develop efficient word-level explanation procedures for two useful fragments of $\mathcal{BV}$, based on interpolation. Our first fragment includes bitvector equalities, extractions, and concatenations where word-level explanations can be constructed through model-based variants of classic equality reasoning techniques (e.g., [5, 11, 12]). Our second fragment is a subset of linear arithmetic where explanations are constructed by interval reasoning in modular arithmetic. When conflicts do not fit into either fragment, we build an explanation by bitblasting and extracting an unsat core. Although this fallback produces theory lemmas expressed at the bit-level, it is used only as a last resort. In addition, this bitblasting-based procedure is local and limited to constraints that are relevant to the current conflict; we do not apply bitblasting to the full problem.

Section 2, is an overview of MCSAT. It also presents the BDD approach and general considerations for conflict explanation. Section 3 describes our interpolation algorithm for equality with concatenation and extraction. Section 4 presents our interpolation method for a fragment of linear bitvector arithmetic. Section 5 presents the normalization technique we apply to conflicts in the hope of expressing them in that bitvector arithmetic fragment. Section 6 presents an evaluation of the approach, which we implemented in the Yices 2 solver [13].[1]

[1] This paper extends preliminary results presented at the SMT workshop [15, 16] and includes a full implementation and experimental evaluation.

2 A General Scheme for Bitvectors

By $\mathcal{BV}$, we denote the theory of quantifier-free fixed-sized bitvectors, a.k.a. QF_BV in SMT-LIB [1]. A first-order term u of $\mathcal{BV}$ is sorted as either a Boolean or a bitvector of a fixed *length* (a.k.a. *bitwidth*), denoted $|u|$. Its set of variables (a.k.a. uninterpreted constants) is denoted $\mathsf{var}(u)$. This paper only uses a few $\mathcal{BV}$ operators. The concatenation of bitvector terms t and u is denoted $t \circ u$; the binary predicates $<^{\mathsf{u}}$, $\leq^{\mathsf{u}}$ denote unsigned comparisons, and $<^{\mathsf{s}}$, $\leq^{\mathsf{s}}$ denote signed comparisons. In such comparisons, both operands must have the same bitwidth. If n is the bitwidth of u, and l and h are two integer indices such that $0 \leq l < h \leq n$, then $u[h{:}l]$, extracts $h{-}l$ bits of u, namely the bits at indices between l and $h{-}1$ (included). We write $u[{:}l]$, $u[h{:}]$, and $u[l]$ as abbreviations for $u[n{:}l]$, $u[h{:}0]$, and $u[l{+}1{:}l]$, respectively. Our convention is to have bitvector indices start from the right-hand side, so that bit 0 is the right-most bit and 0011[2:] is 11. We use standard notations for bitvector arithmetic, which coincides with arithmetic modulo 2^w where w is the bitwidth. We sometimes use integer constants e.g., 0, 1, -1 for bitvectors when the bitwidth is clear. We use the standard (quantifier-free) notions of *literal*, *clause*, *cube*, and *formula* [31].

A *model* of a $\mathcal{BV}$ formula Φ is an assignment that gives a bitvector (resp. Boolean) value to all bitvector (resp. Boolean) variables of Φ, in such a way that Φ evaluates to true, under the standard interpretation of Boolean and bitvector symbols. To simplify the presentation, we assume in this paper that there are no Boolean variables, although they are supported in our implementation.

2.1 MCSAT Overview

MCSAT searches for a model of an input quantifier-free formula by building a partial assignment—maintained in a *trail*—and extends the concepts of unit propagation and consistency to first-order terms and literals [21,22,26]. Reasoning is implemented by theory-specific plugins, each of which has a partial view of the trail. In the case of $\mathcal{BV}$, the bitvector plugin sees in the trail an assignment $\mathcal{M}$ of the form $x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$ that gives values to bitvector variables, and a set of bitvector literals $L_1, \ldots, L_t$, called *constraints*, that must be true in the current trail. MCSAT and its bitvector plugin maintain the invariant that none of the literals L_i evaluates to false under $\mathcal{M}$; either L_i is true or some variable of L_i has no value in $\mathcal{M}$. To maintain this invariant, they detect *unit inconsistencies*: We say that literal L_i is *unit in y* if y is the only unassigned variable of L_i, and that a trail is *unit inconsistent* if there is a variable y and a subset $\{C_1, \ldots, C_m\}$ of $\{L_1, \ldots, L_t\}$, called a *conflict*, such that every C_j is unit in y and the formula $\exists y \bigwedge_{i=1}^{m} C_i$ evaluates to false under $\mathcal{M}$. In such a case, y is called the *conflict variable* and $C_1, \ldots, C_m$ are called the *conflict literals*.

When such a conflict is detected, the current assignment, or partial model, $\mathcal{M}$ cannot be extended to a full model; some values assigned to $x_1, \ldots, x_n$ must be revised. As in CDCL, MCSAT backtracks and updates the current assignment by learning a new clause that explains the conflict. This new clause must not contain other variables than $x_1, \ldots, x_n$ and it must rule out the current assignment. For some theories, this *conflict explanation* can be built by quantifier elimination. More generally, we can build an explanation from an *interpolant*.

Definition 1 (Interpolant). *A clause I is an* interpolant[2] *for formula F at model $\mathcal{M}$ assigning values to $x_1, \ldots, x_n$, if (1) $F \Rightarrow I$ is valid (in $\mathcal{BV}$), (2) The variables in I are in $\{x_1, \ldots, x_n\} \cap \mathsf{var}(F)$, and (3) I evaluates to false in $\mathcal{M}$.*

Given an interpolant I for the conjunction $\bigwedge_{i=1}^{m} C_i$ of the conflict literals (or equivalently, for $\exists y \bigwedge_{i=1}^{m} C_i$) at the current model $\mathcal{M}$, the conflict explanation is clause $(\bigwedge_{i=1}^{m} C_i) \Rightarrow I$. Our main goal is constructing such interpolants in $\mathcal{BV}$.

2.2 BDD Representation and Conflict Detection

To detect conflicts, we must keep track of the set of feasible values for every unassigned variable y. These sets are frequently updated during search so an efficient representation is critical. The following operations are needed:

- updating the set when a new constraint becomes unit in y,
- detecting when the set becomes empty,
- selecting a value from the set.

For $\mathcal{BV}$, Zeljić et al. [32] represent sets of feasible values using both intervals and bit patterns. For example, the set defined by the interval $[0000, 0011]$ and the pattern ???1 is the pair $\{0001, 0011\}$ (i.e., all bitvectors in the interval whose low-order bit is 1). This representation is lightweight and efficient but it is not precise. Some sets are not representable exactly. We use Binary Decision Diagrams (BDD) [6] over the bits of y. The major advantage is that BDDs provide an exact implementation of any set of values for y. Updating sets of values amounts to computing the conjunction of BDDs (i.e., set intersection). Checking whether a set is empty and selecting a value in the set (if it is not), can be done efficiently by, respectively, checking whether the BDD is false, and performing a top-down traversal of the BDD data structure. There is a risk that the BDD representation explodes but this risk is reduced in our context since each BDD we build is for a single variable (and most variables do not have too many bits). We use the CUDD package [10] to implement BDDs.

2.3 Baseline Conflict Explanation

Given a conflict as described previously, the clause $(x_1 \not\simeq v_1) \vee \cdots \vee (x_n \not\simeq v_n)$, which is falsified by model $\mathcal{M}$ only, is an interpolant for $\bigwedge_{i=1}^{m} C_i$ at $\mathcal{M}$ according to Definition 1. This gives the following trivial conflict explanation:

$$C_1 \wedge \cdots \wedge C_m \Rightarrow (x_1 \not\simeq v_1) \vee \cdots \vee (x_n \not\simeq v_n)$$

We seek to generalize model $\mathcal{M}$ with a formula that rules out bigger parts of the search space than just $\mathcal{M}$. A first improvement is replacing the constraints by a *core* $\mathcal{C}$, that is, a minimal subset of $\{C_1, \ldots, C_n\}$ that evaluates to false in $\mathcal{M}$.[3]

[2] This is the same as the usual notion of (reverse) interpolant between formulas if we see $\mathcal{M}$ as the formula $F_{\mathcal{M}}$ defined by $(x_1 \simeq v_1) \wedge \cdots \wedge (x_n \simeq v_n)$: the interpolant is implied by F, it is inconsistent with $F_{\mathcal{M}}$, and its variables occur in both F and $F_{\mathcal{M}}$.

[3] In our implementation, we construct $\mathcal{C}$ using the QuickXplain algorithm [23].

To produce the interpolant I, we can bitblast the constraints $C_1, \ldots, C_m$ and solve the resulting SAT problem *under the assumptions* that each bit of $x_1, \ldots, x_n$ is true or false as indicated by the values $v_1, \ldots, v_n$. Since the SAT problem encodes a conflict, the SAT solver will return an *unsat core*, from which we can extract bits of $v_1, \ldots, v_n$ that contribute to unsatisfiability. This generalizes $\mathcal{M}$ by leaving some bits unassigned, as in [32].

This method is general. It works whatever the constraints $C_1, \ldots, C_m$, so we use it as a default procedure. The bitblasting step focuses on constraints that are unit in y, which typically leads to a much smaller SAT problem than bitblasting the whole problem from the start. However, the bitblasting approach can still be costly and it may produce weak explanations.

Example 1. Consider the constraints $\{x_1 \not\simeq x_2,\ \ x_1 \simeq y,\ \ x_2 \simeq y\}$ and the assignment $x_1 \mapsto 1001, x_2 \mapsto 0101$. The bitblasting approach might produce explanation $(x_1 \simeq y \wedge x_2 \simeq y) \Rightarrow (x_1[3] \Rightarrow x_2[3])$. After backtracking, we might similarly learn that $(x_2[3] \Rightarrow x_1[3])$. In this way, it will take eight iterations to learn enough information to represent the high-level explanation:

$$(x_1 \simeq y \wedge x_2 \simeq y) \Rightarrow x_1 \simeq x_2 \ \ .$$

A procedure that can produce $(x_1 \simeq x_2)$ directly is much more efficient.

3 Equality, Concatenation, Extraction

Our first specialized interpolation mechanism applies when constraints $\mathcal{C} = \{C_1, \ldots, C_m\}$ belong to the following grammar:

$$\begin{array}{lrl} \text{Constraints} & C ::= & t \simeq t \mid t \not\simeq t \\ \text{Terms} & t\ ::= & e \mid y[h{:}l] \mid t \circ t \end{array}$$

where e ranges over any bitvector terms such that $y \notin \mathsf{var}(e)$. Without loss of generality, we can assume that $\mathcal{C}$ is a core. We split $\mathcal{C}$ into a set of equalities $E = \{a_i \simeq b_i\}_{i \in \mathfrak{E}}$ and a set of disequalities $D = \{a_i \not\simeq b_i\}_{i \in \mathfrak{D}}$.

Slicing. Our first step rewrites $\mathcal{C}$ into an equivalent *sliced* form. This computes the *coarsest-base slicing* [5, 11] of equalities and disequalities in $\mathcal{C}$. The goal of this rewriting step is to split the variables into slices that can be treated as independent terms. The terms in coarsest-base slicing are either of the form $y[h{:}l]$ (slices), or are *evaluable terms* e with $y \notin \mathsf{var}(e)$.

Example 2. Consider the constraints $E = \{x_1[4{:}0] \simeq x_1[8{:}4], y[6{:}2] \simeq y[4{:}0]\}$ and $\{y[4{:}0] \not\simeq x_1[8{:}4]\}$ over variables y of length 6, and x_1 of length 8. We cannot treat $y[6{:}2]$ and $y[4{:}0]$ as independent terms because they overlap. To break the overlap, we introduce slices: $y[6{:}4]$, $y[4{:}2]$, and $y[2{:}0]$. Equality $y[6{:}2] \simeq y[4{:}0]$ is rewritten to $(y[6{:}4] \simeq y[4{:}2]) \wedge (y[4{:}2] \simeq y[2{:}0])$. Disequality $y[4{:}0] \not\simeq x_1[8{:}4]$ is rewritten to $(y[4{:}2] \not\simeq x_1[8{:}6]) \vee (y[2{:}0] \not\simeq x_1[6{:}4])$. The final result is

$$E_s = \{\ x_1[4{:}2] \simeq x_1[8{:}6]\ ,\ x_1[2{:}0] \simeq x_1[6{:}4]\ ,\ y[6{:}4] \simeq y[4{:}2]\ ,\ y[4{:}2] \simeq y[2{:}0]\ \}\ ,$$
$$D_s = \{\ (y[4{:}2] \not\simeq x_1[8{:}6]) \vee (y[2{:}0] \not\simeq x_1[6{:}4])\ \}.$$

Algorithm 1 E-graph with value management

```
1: function E_GRAPH($E_s, \mathcal{M}$)
2:     INITIALIZE($\mathcal{G}$)    ▷ each evaluable term or slice is its own component
3:     for $t_1 \simeq t_2 \in E_s$ do
4:         $t'_1 \leftarrow$ REP($t_1, \mathcal{G}$)    ▷ get representative for $t_1$'s component
5:         $t'_2 \leftarrow$ REP($t_2, \mathcal{G}$)    ▷ get representative for $t_2$'s component
6:         if $y \notin \mathsf{var}(t'_1)$ and $y \notin \mathsf{var}(t'_2)$ and $[\![t'_1]\!]_{\mathcal{M}} \neq [\![t'_2]\!]_{\mathcal{M}}$ then
7:             raise_conflict($E \Rightarrow t'_1 \simeq t'_2$)    ▷ $D$ must be empty
8:         $t_3 \leftarrow$ SELECT($t'_1, t'_2$)    ▷ select representative for merged component
9:         $\mathcal{G} \leftarrow$ MERGE($t_1, t_2, t_3, \mathcal{G}$)    ▷ merge the components with representative $t_3$
10:    return $\mathcal{G}$
```

Explanations. After slicing, we obtain a set E_s of equalities and a set D_s that contains disjunctions of disequalities. We can treat each slice as a separate variable, so the problem lies within the theory of equality on a *finite domain.*

We first analyze the conflict with equality reasoning against the model, as shown in Algorithm 1. We construct the E-graph $\mathcal{G}$ from E_s [12], while also taking into account the partial model $\mathcal{M}$ that triggered the conflict. The model can evaluate terms e such that $y \notin \mathsf{var}(e)$ to values $[\![e]\!]_{\mathcal{M}}$, and those can be the source of the conflict. To use the model for evaluating terms, we maintain two invariants during E-graph construction:

1. If a component contains an evaluable term c, then the representative of that component is evaluable.
2. Two evaluable terms c_1 and c_2 in the same component must evaluate to the same value, otherwise this is the source of the conflict.

The E-graph construction can detect and explain basic conflicts between the equalities in E and the current assignment.

Example 3. Let r_1, r_2, r_3 be bit ranges of the same width. Let E be such that $E_s = \{x_1[r_1] \simeq y[r_3],\ x_2[r_2] \simeq y[r_3]\}$, and let $D = \emptyset$. Consider the model $\mathcal{M} := x_1 \mapsto 0\ldots0, x_2 \mapsto 1\ldots1$. Then, E_GRAPH($E_s, \mathcal{M}$) produces the conflict clause $E \Rightarrow x_1[r_1] \simeq x_2[r_2]$.

If the E-graph construction does not raise a conflict, then $\mathcal{M}$ is compatible with the equalities in E_s. Since $\mathcal{C}$ conflicts with $\mathcal{M}$, the conflict explanation must involve D_s. To obtain an explanation, we decompose each disjunct $C \in D_s$ into $(C_{E_s} \vee C_{\mathcal{M}} \vee C_{\mathsf{interface}} \vee C_{\mathsf{free}})$ as follows.

– C_{E_s} contains disequalities $t_1 \not\simeq t_2$ such that t_1 and t_2 have the same E-graph representatives; such disequalities are false because of the equalities in E_s.
– $C_{\mathcal{M}}$ contains disequalities $t_1 \not\simeq t_2$ such that t_1 and t_2 have distinct representatives t'_1 and t'_2 with $[\![t'_1]\!]_{\mathcal{M}} = [\![t'_2]\!]_{\mathcal{M}}$; these are false because of $\mathcal{M}$.
– $C_{\mathsf{interface}}$ contains disequalities $t_1 \not\simeq t_2$ such that t_1 and t_2 have distinct representatives t'_1 and t'_2, t'_1 is evaluable and t'_2 is a slice; we can still satisfy $t_1 \not\simeq t_2$ by picking a good value for y; we say t'_1 is an *interface term.*
– C_{free} contains disequalities $t_1 \not\simeq t_2$ such that t_1 and t_2 have distinct slices as representatives; we can still satisfy $t_1 \not\simeq t_2$ by picking a good value for y.

Algorithm 2 Disequality conflict

```
1: function DIS_CONFLICT($D_s, \mathcal{M}, \mathcal{G}$)
2:     $S \leftarrow \emptyset$                                ▷ where we collect interface terms
3:     $C_0 \leftarrow \emptyset$          ▷ where we collect the disequalities that evaluate to false
4:     for $C \in D_s$ do
5:         $C^{\mathsf{rep}}_{\mathcal{M}} \leftarrow \bigvee\{\text{REP}(t_1,\mathcal{G}) \not\simeq \text{REP}(t_2,\mathcal{G}) \mid (t_1 \not\simeq t_2) \in C_{\mathcal{M}}\}$
6:         if IS_EMPTY($C_{\mathsf{interface}}$) and IS_EMPTY($C_{\mathsf{free}}$) then
7:             raise_conflict($E \wedge D \Rightarrow C^{\mathsf{rep}}_{\mathcal{M}}$)
8:         else
9:             $C_0 \leftarrow C_0 \vee C^{\mathsf{rep}}_{\mathcal{M}}$    ▷ we collect the disequalities made false in the model
10:            for $t_1 \not\simeq t_2 \in C_{\mathsf{interface}}$ with $y \not\in \mathsf{var}(\text{REP}(t_1,\mathcal{G}))$ do
11:                $S \leftarrow S \cup \{\text{REP}(t_1,\mathcal{G})\}$              ▷ we collect the interface term
12:    $C_{\neq} \leftarrow \bigvee\{t_1 \simeq t_2 \mid [\![t_1]\!]_{\mathcal{M}} \neq [\![t_2]\!]_{\mathcal{M}},\ t_1, t_2 \in S\}$
13:    $C_{=} \leftarrow \bigvee\{t_1 \not\simeq t_2 \mid [\![t_1]\!]_{\mathcal{M}} = [\![t_2]\!]_{\mathcal{M}},\ t_1 \neq t_2,\ t_1, t_2 \in S\}$
14:    return $E \wedge D \Rightarrow C_0 \vee C_{\neq} \vee C_{=}$
```

The disjuncts in D_s take part in the conflict either when (i) one of the clauses in D_s is false because $C_{\mathsf{interface}}$ and C_{free} are both empty; or (ii) the finite domains are too small to satisfy the disequalities in $C_{\mathsf{interface}}$ and C_{free}, given the values assigned in $\mathcal{M}$. In either case, we can produce a conflict explanation with Algorithm 2.

In a type (i) conflict, the algorithm produces an interpolant $C^{\mathsf{rep}}_{\mathcal{M}}$ that is derived from a single element of D_s. Because we assume that $\mathcal{C}$ is a core, a type (i) conflict can happen only if D_s is a singleton. Here is how the algorithm behaves on such a conflict:

Example 4. Let r_1 and r_2 be bit ranges of the same length, let r_3, r_4, r_5 be bit ranges of the same length. Assume E_s contains

$$\{\ x_1[r_1] \simeq y[r_1]\ ,\ x_2[r_2] \simeq y[r_2]\ ,\ y[r_3] \simeq y[r_5]\ ,\ y[r_4] \simeq y[r_5]\ \},$$

and assume D_s is the singleton $\{\ (y[r_1] \not\simeq y[r_2] \vee y[r_3] \not\simeq y[r_4])\ \}$. Let $\mathcal{M}$ map x_1 and x_2 to $0 \ldots 0$ and assume $y[r_5]$ is the E-graph representative for component

$$\{\ y[r_3], y[r_4], y[r_5]\ \}.$$

The unique clause of D_s contains two disequalities:

– The first one, $y[r_1] \not\simeq y[r_2]$, belongs to $C_{\mathcal{M}}$ because the representatives of $y[r_1]$ and $y[r_2]$, namely $x_1[r_1]$ and $x_2[r_2]$, both evaluate to $0 \ldots 0$.
– The second one, $y[r_3] \not\simeq y[r_4]$,belongs to C_{E_s} because the representatives of $y[r_3]$ and $y[r_4]$ are both $y[r_5]$,

As $C_{\mathsf{interface}}$ and C_{free} are empty, Algorithm 2 outputs $E \wedge D \Rightarrow x_1[r_1] \not\simeq x_2[r_2]$.

For a conflict of type (ii), the equalities and disequalities that hold in $\mathcal{M}$ between the interface terms make the slices of y require more values than there exist. So the produced conflict clause includes (the negation of) all such equalities and disequalities. An example can be given as follows:

Example 5. Assume E (and then E_s) is empty and assume D_s is

$$\{\ x_2[0] \not\simeq x_2[1] \vee y[0] \not\simeq y[1]\ ,\ x_1[0] \not\simeq y[0]\ ,\ x_1[1] \not\simeq y[1]\ \}$$

Let $\mathcal{M}$ map x_1 and x_2 to 00. Then DIS_CONFLICT$(D_s, \mathcal{M}, \mathcal{G})$ behaves as follows:

– In the first clause, call it C, the first disequality is in $C_{\mathcal{M}}$, as the two sides are in different components but evaluate to the same value; so C_0 becomes $\{ \ x_2[0] \not\simeq x_2[1] \ \}$; the second disequality features two slices and is thus in C_{free}; The clause is potentially satisfiable and we move to the next clause.
– The second clause contains a single disequality that cannot be evaluated (since $y[0]$ is not evaluable in $\mathcal{M}$). Term $x_1[0]$ is added to S. The clause is potentially satisfiable so we move to the next clause.
– The third clause of D_s is similar. It contains a single disequality that cannot be evaluated. The interface term $x_1[1]$ is added to S.

Since all clauses of D_s have been processed, the conflict is of type (ii). Indeed, $y[0]$ must be different from 0 because of the second clause, $y[1]$ must also be different from 0 because of the third clause, but $y[0]$ and $y[1]$ must be different from each other because of the first clause. Since both $y[0]$ and $y[1]$ have only one bit, there are only two possible values for these two slices, so the three constrains are in conflict. Algorithm 2 produces the conflict clause

$$D \Rightarrow (\ x_2[0] \not\simeq x_2[1] \vee x_1[0] \not\simeq x_1[1] \ ).$$

The disequality $x_2[0] \not\simeq x_2[1]$ is necessary because, if it were true in $\mathcal{M}$, we would not have to satisfy $y[0] \not\simeq y[1]$ and therefore $y \leftarrow 11$ would work. Disequality $x_1[0] \not\simeq x_1[1]$ is also necessary because, if it were true in $\mathcal{M}$, say with $x_1 \leftarrow 01$ (resp. $x_1 \leftarrow 10$), then $y \leftarrow 11$ (resp. $y \leftarrow 00$) would work.

Correctness of the method relies on the following lemma, whose proof can be found in [17].

Lemma 1 (The produced clauses are interpolants).
1. *If Algorithm 1 reaches line 7, $t'_1 \simeq t'_2$ is an interpolant for $E \wedge D$ at $\mathcal{M}$.*
2. *If Algorithm 2 reaches line 7, $C^{\mathsf{rep}}_{\mathcal{M}}$ is an interpolant for $E \wedge D$ at $\mathcal{M}$.*
3. *If it reaches line 14, $C_0 \vee C_{\neq} \vee C_{=}$ is an interpolant for $E \wedge D$ at $\mathcal{M}$.*

4 A Linear Arithmetic Fragment

Our second specialized explanation mechanism applies when constraints $\mathcal{C} = \{C_1, \ldots, C_m\}$ belong to the following grammar:

$$\begin{array}{lll} \text{Constraints} & C ::= a \mid \neg a & \\ \text{Atoms} & a \ ::= e_1 + t \leq^{\mathsf{u}} e_2 + t \mid e_1 \leq^{\mathsf{u}} e_2 + t \mid e_1 + t \leq^{\mathsf{u}} e_2 \\ \text{Terms} & t \ ::= y[h{:}] \mid t[{:}l] \mid t + e_1 \mid -t \mid 0_k \circ t \mid t \circ 0_k \end{array}$$

where e_1 and e_2 range over *evaluable* bitvector terms (i.e., $y \notin \mathsf{var}(e_1) \cup \mathsf{var}(e_2)$), and 0_k is 0 on k bits. We can represent variable y as the term $y[|y|{:}]$. This fragment of bitvector arithmetic is *linear* in y and there can be only one occurrence of y in terms. Constraints in Section 3 are then outside this fragment in general.

Let $\mathcal{A}$ be $\exists y(C_1 \wedge \cdots \wedge C_m)$, and $\mathcal{M}$ be the partial model involved in the conflict. The interpolant for $\mathcal{A}$ at model $\mathcal{M}$ is (roughly) produced as follows:

1. For each constraint C_i, $1 \leq i \leq m$, featuring a (necessarily unique) lower-bits extract $y[w_i{:}]$, we compute a *condition cube* c_i satisfied by $\mathcal{M}$ and a

Atom a	**Forbidden interval that** a **(resp.** $\neg a$**) specifies for** t			
	I_a	$I_{\neg a}$	**Condition** $c_a/c_{\neg a}$	
$e_1 + t \leq^{\mathsf{u}} e_2 + t$	$[-e_2\,;\,-e_1[$	$[-e_1\,;\,-e_2[$	$e_1 \not\simeq e_2$	1
	$[0\,;0[$	full	$e_1 \simeq e_2$	2
$e_1 \leq^{\mathsf{u}} e_2 + t$	$[-e_2\,;e_1 - e_2[$	$[e_1 - e_2\,;\,-e_2[$	$e_1 \not\simeq 0$	3
	$[0\,;0[$	full	$e_1 \simeq 0$	4
$e_1 + t \leq^{\mathsf{u}} e_2$	$[e_2 - e_1 + 1\,;\,-e_1[$	$[-e_1\,;e_2 - e_1 + 1[$	$e_2 \not\simeq -1$	5
	$[0\,;0[$	full	$e_2 \simeq -1$	6

Table 1: Creating the forbidden intervals

forbidden interval I_i of the form $[l_i\,;u_i[$, where l_i and u_i are evaluable terms, such that $c_i \Rightarrow (C_i \Leftrightarrow (y[w_i{:}] \notin I_i))$ is valid.

2. We group the resulting intervals $(I_i)_{1\leq i\leq m}$ according to their bitwidths: if $\mathcal{S}_w$ is the set of intervals forbidding values for $y[w{:}]$, $1 \leq w \leq |y|$, then under condition $\bigwedge_{i=1}^{m} c_i$ formula $\mathcal{A}$ is equivalent to $\exists y(\bigwedge_{w=1}^{|y|}\ (\ y[w{:}] \notin \bigcup_{I\in\mathcal{S}_w} I\ ))$.
3. We produce a series of constraints $d_1,\ldots,\ d_p$ that are satisfied by $\mathcal{M}$ and that are inconsistent with $\bigwedge_{w=1}^{|y|}\ (\ y[w{:}] \notin \bigcup_{I\in\mathcal{S}_w} I\ )$. The interpolant will be $(\bigwedge_{i=1}^{m} c_i \wedge \bigwedge_{i=1}^{p} d_i) \Rightarrow \bot$: it is implied by $\mathcal{A}$, and evaluates to false in $\mathcal{M}$.

4.1 Forbidden Intervals

An *interval* takes the form $[l\,;u[$, where the lower bound l and upper bound u are evaluable terms of some bitwidth w, with l included and u excluded. The notion of interval used here is considered modulo 2^w. We do not require $l \leq^u u$ so an interval may "wrap around" in $\mathbb{Z}/2^w\mathbb{Z}$. For instance, the interval $[1111\,;0001[$ contains two bitvector values, namely, 1111 and 0000. If l and u evaluate to the same value, then we consider $[l\,;u[$ to be empty (as opposed to the full domain, which we denote by full^w or just full). Notation $t \in I$ stands for literal $\top$ if I is full and literal $t{-}l <^u u{-}l$ if I is $[l\,;u[$. The value in model $\mathcal{M}$ of an evaluable term e (resp. evaluable cube c, interval I) is denoted $[\![e]\!]_{\mathcal{M}}$ (resp. $[\![c]\!]_{\mathcal{M}}$, $[\![I]\!]_{\mathcal{M}}$).

Given a constraint C with unevaluable term t, we produce an interval I_C of forbidden values for t according to the rules of Table 1. A side condition literal c_C identifies when the lower and upper bounds would coincide, in which case the interval produced is either empty or full. For every row of the table, the formula $c_C \Rightarrow (C \Leftrightarrow t \notin I_C)$ is valid in $\mathcal{BV}$. Given a partial model $\mathcal{M}$, we convert C to such an interval by selecting the row where $[\![c_C]\!]_{\mathcal{M}} = \mathsf{true}$.

Example 6.

6.1 Assume C_1 is literal $\neg(x_1 \leq^{\mathsf{u}} y)$ and $\mathcal{M} = \{x_1 \mapsto 0000\}$. Then line 4 of Table 1 applies, and I_{C_1} is interval full with condition $x_1 \simeq 0$.

6.2 Assume C_1 is $\neg(y \simeq x_1)$, C_2 is $(x_1 \leq^{\mathsf{u}} x_3 + y)$, C_3 is $\neg(y - x_2 \leq^{\mathsf{u}} x_3 + y)$, and $\mathcal{M} = \{x_1 \mapsto 1100, x_2 \mapsto 1101, x_3 \mapsto 0000\}$. Then by line 5, $I_{C_1} = [x_1\,;x_1 + 1[$ with trivial condition $(0 \not\simeq -1)$, by line 3, $I_{C_2} = [-x_3\,;x_1 - x_3[$ with condition $(x_1 \not\simeq 0)$, and by line 1, $I_{C_3} = [x_2\,;\,-x_3[$ with condition $(-x_2 \not\simeq x_3)$.

$$
\begin{array}{llll}
\mathsf{forbid}(\,t\,,\ [0\,;0[\,,\ c\,) := (1,\ [0\,;0[,\ c) & & \mathsf{forbid}(\,0_k \circ t\,,\ I\,,\ c\,) := \mathsf{utrim}_k(\,t\,,\ I\,,\ c\,) \\
\mathsf{forbid}(\,t\,,\ \mathsf{full}\,,\ c\,) := (1,\ \mathsf{full},\ c) & & \mathsf{forbid}(\,t \circ 0_k\,,\ I\,,\ c\,) := \mathsf{dtrim}_k(\,t\,,\ I\,,\ c\,) \\
\mathsf{forbid}(\,y[w{:}]\,,\ I\,,\ c\,) := (w,\ I,\ c) & & \qquad\qquad \text{when } I \text{ is not } [0\,;0[\text{ nor full} \\
\mathsf{forbid}(\,t[{:}w]\,,\ [l\,;u[\,,\ c\,) := \mathsf{forbid}(\,t\,,\ [l \circ 0_w\,; u \circ 0_w[\,,\ c\,) & & \\
\mathsf{forbid}(\,t+c\,,\ [l\,;u[\,,\ c\,) := \mathsf{forbid}(\,t\,,\ [l{-}c\,; u{-}c[\,,\ c\,) & & \\
\mathsf{forbid}(\,-t\,,\ [l\,;u[\,,\ c\,) := \mathsf{forbid}(\,t\,,\ [1{-}u\,; 1{-}l[\,,\ c\,) & &
\end{array}
$$

$$
\mathsf{utrim}_k(\,t\,,\ [l\,;u[\,,\ c\,) := \begin{cases}
\mathsf{forbid}(\,t\,,\ [l'\,;u'[\,,\ c \wedge c_l \wedge c_u\,) & \text{if } [l'\,;u'[\text{ is not } [0\,;0[\\
(1,\ \mathsf{full},\ c \wedge c_l \wedge c_u \wedge c') & \text{if } [l'\,;u'[\text{ is } [0\,;0[\text{ and } [\![c']\!]_{\mathcal{M}} \text{ is true} \\
(1,\ [0\,;0[,\ c \wedge c_l \wedge c_u \wedge \neg c') & \text{if } [l'\,;u'[\text{ is } [0\,;0[\text{ and } [\![c']\!]_{\mathcal{M}} \text{ is false}
\end{cases}
$$

where l' is $l[w{:}]$ (resp. 0_w) and c_l is a_l (resp. $\neg a_l$) if $[\![a_l]\!]_{\mathcal{M}}$ is true (resp. false),
u' is $u[w{:}]$ (resp. 0_w) and c_u is a_u (resp. $\neg a_u$) if $[\![a_u]\!]_{\mathcal{M}}$ is true (resp. false),
a_l is $l[{:}w] \simeq 0_k$, a_u is $u[{:}w] \simeq 0_k$, c' is $(0_{k+w} \in [l\,;u[)$, and w is $|t|$.

$$
\mathsf{dtrim}_k(\,t\,,\ [l\,;u[\,,\ c\,) := \begin{cases}
\mathsf{forbid}(\,t\,,\ [l'\,;u'[\,,\ p \wedge c_l \wedge c_u\,) & \text{if } [l'\,;u'[\text{ is not } [0\,;0[\\
(1,\ \mathsf{full},\ c \wedge c_l \wedge c_u \wedge c') & \text{if } [l'\,;u'[\text{ is } [0\,;0[\text{ and } [\![c']\!]_{\mathcal{M}} \text{ is true} \\
(1,\ [0\,;0[,\ c \wedge c_l \wedge c_u \wedge \neg c') & \text{if } [l'\,;u'[\text{ is } [0\,;0[\text{ and } [\![c']\!]_{\mathcal{M}} \text{ is false}
\end{cases}
$$

where l' is $l[{:}k]$ (resp. $l[{:}k]{+}1$) and c_l is a_l (resp. $\neg a_l$) if $[\![a_l]\!]_{\mathcal{M}}$ is true (resp. false),
u' is $u[{:}k]$ (resp. $u[{:}k]{+}1$) and c_u is a_u (resp. $\neg a_u$) if $[\![a_u]\!]_{\mathcal{M}}$ is true (resp. false),
a_l is $l[k{:}] \simeq 0_k$, a_u is $u[k{:}] \simeq 0_k$, c' is $(u' \circ 0_k \in [l\,;u[)$, and w is $|t|$.

Fig. 1: Transforming the forbidden intervals

Given the supported grammar, term t contains a unique subterm of the form $y[w{:}]$. We transform I_C into an interval of forbidden values for $y[w{:}]$ by applying procedure $\mathsf{forbid}(\,t\,,\ I_C\,,\ c_C\,)$ shown in Figure 1, which proceeds by recursion on t. Its specification is given below, and correctness is proved by induction on t.

Lemma 2 (Correctness of forbidden intervals). *Assuming cube c is true in $\mathcal{M}$, then $\mathsf{forbid}(\,t\,,\ I\,,\ c\,)$ returns a triple (w, I', c') such that c' is a cube that is true in $\mathcal{M}$, and both $c' \Rightarrow c$ and $c' \Rightarrow (t \notin I \Leftrightarrow y[w{:}] \notin I')$ are valid in $\mathcal{BV}$.*

Running $\mathsf{forbid}(\,t_{C_i}\,,\ I_{C_i}\,,\ c_{C_i}\,)$ for all constraints C_i, $1{\le}i{\le}m$, produces a family of triples $(w_i, I'_i, c'_i)_{1 \le i \le m}$ such that, for each i, formula $c'_i \Rightarrow (C_i \Leftrightarrow (y[w_i{:}] \notin I'_i))$ is valid in $\mathcal{BV}$ and c'_i is true in $\mathcal{M}$.

4.2 Interpolant

First, assume that one of the triples obtained above is of the form (w, full, c), coming from constraint C. As the interval forbids the full domain of values for $y[w{:}]$, we produce conflict clause $C \wedge c \Rightarrow \bot$. This formula is an interpolant for $\mathcal{A}$ at $\mathcal{M}$. This is illustrated in Example 7.1.

Example 7.

7.1 In Example 6.1 where C_1 is literal $\neg(x_1 \le^{\mathsf{u}} y)$ and $\mathcal{M} = \{x_1 \mapsto 0000\}$, the interpolant for $\neg(x_1 \le^{\mathsf{u}} y)$ at $\mathcal{M}$ is $(x_1 \simeq 0) \Rightarrow \bot$.

7.2 Example 6.2 does not contain a full interval. Model $\mathcal{M}$ satisfies the three conditions $c_1 := (0 \not\simeq -1)$, $c_2 := (x_1 \not\simeq 0)$ and $c_3 := (-x_2 \not\simeq x_3)$, and the

bitwidth	w_1	$>$	w_2	$> \cdots >$	w_j
Interval layer	w_1-intervals $\mathcal{S}_1 = \{I_{1.1}, I_{1.2}, \ldots\}$		w_2-intervals $\mathcal{S}_2 = \{I_{2.1}, I_{2.2}, \ldots\}$	$\ldots$ $\ldots$	w_j-intervals $\mathcal{S}_j = \{I_{j.1}, I_{j.2}, \ldots\}$
Forbidding values for	$y[w_1{:}]$		$y[w_2{:}]$	$\ldots$	$y[w_j{:}]$

Fig. 2: Intervals collected from $C_1 \wedge \cdots \wedge C_m$

intervals $I_1 = [x_1\,; x_1+1[$, $I_2 = [-x_3\,; x_1-x_3[$, and $I_3 = [x_2\,; -x_3[$, evaluate to $[\![I_1]\!]_\mathcal{M} = [1100\,;1101[$, $[\![I_2]\!]_\mathcal{M} = [0000\,;1100[$, and $[\![I_3]\!]_\mathcal{M} = [1101\,;0000[$, respectively. Note how $\bigcup_{i=1}^{3}[\![I_i]\!]_\mathcal{M}$ is the full domain.

Assume now that no interval is full (as in Example 7.2). We group the triples (w, I, c) into different *layers* characterized by their bitwidths w: I will henceforth be called a w*-interval*, restricting the feasible values for $y[w{:}]$, and c_I denotes its associated condition in the triple. Ordering the groups of intervals by decreasing bitwidths $w_1 > w_2 > \cdots > w_j$, as shown in Figure 2, $\mathcal{S}_j$ denotes the set of produced w_j-intervals. The properties satisfied by the triples entail that

$$\mathcal{A} \wedge (\textstyle\bigwedge_{i=1}^{j} \bigwedge_{I\in\mathcal{S}_i} c_I) \Rightarrow \mathcal{B}$$

is valid, where $\mathcal{B}$ is $\exists y \bigwedge_{i=1}^{j}(y[w_i{:}] \notin \bigcup_{I\in\mathcal{S}_i} I)$. And formula $(\bigwedge_{i=1}^{j} \bigwedge_{I\in\mathcal{S}_i} c_I) \Rightarrow \mathcal{B}$ is false in $\mathcal{M}$. To produce an interpolant, we replace $\mathcal{B}$ by a quantifier-free clause.

The simplest case is when there is only one bitwidth $w = w_1$: the fact that $\mathcal{B}$ is falsified by $\mathcal{M}$ means that $\bigcup_{I\in\mathcal{S}_1}[\![I]\!]_\mathcal{M}$ is the full domain $\mathbb{Z}/2^w\mathbb{Z}$. Property "$\bigcup_{I\in\mathcal{S}_1} I$ is the full domain" is then expressed symbolically as a conjunction of constraints in the bitvector language. To compute them, we first extract a sequence $I_1, \ldots, I_q$ of intervals from the set $\mathcal{S}_1$, originating from a subset $\mathcal{C}$ of the original constraints $(C_i)_{i=1}^{m}$, and such that the sequence $[\![I_1]\!]_\mathcal{M}, \ldots, [\![I_q]\!]_\mathcal{M}$ of *concrete* intervals leaves no "hole" between an interval of the sequence and the next, and goes round the full circle of domain $\mathbb{Z}/2^w\mathbb{Z}$: the sequence forms a circular chain of linking intervals. This chain can be produced by a standard coverage extraction algorithm (see, e.g., [17]). Formula $\mathcal{B} := \exists y(y[w{:}] \notin \bigcup_{I\in\mathcal{S}_1} I)$ is then replaced by $(\bigwedge_{i=1}^{q} u_i \in I_{i+1}) \Rightarrow \bot$, where u_i is the upper bound of I_i and I_{q+1} is I_1. Each interval has its upper bound in the next interval $(u_i \in I_{i+1})$, i.e., intervals do link up with each other. The conflict clause is then

$$(\mathcal{C} \wedge (\textstyle\bigwedge_{i=1}^{q} c_{I_i}) \wedge (\bigwedge_{i=1}^{q} u_i \in I_{i+1})) \Rightarrow \bot$$

Example 8. For Example 7.2, the coverage-extraction algorithm produces the sequence I_1, I_3, I_2, i.e., $[x_1\,; x_1+1[$, $[x_2\,; -x_3[$, $[-x_3\,; x_1-x_3[$. The linking constraints are then $d_3 := (x_1+1) \in I_3$, $d_2 := (-x_3) \in I_2$, and $d_1 := (x_1-x_3) \in I_1$, and the interpolant is $d_3 \wedge d_2 \wedge d_1 \Rightarrow \bot$.[4]

When several bitwidths are involved, the intervals must "complement each other" at different bitwidths so that no value for y is feasible. For a bitwidth w_i, the union of the w_i-intervals in model $\mathcal{M}$ may not necessarily cover the full

[4] We omit c_1, c_2, c_3 here, since they are subsumed by d_1, d_2, d_3, respectively.

Algorithm 3 Producing the interpolant with multiple bitwidths

```
1: function COVER($(\mathcal{S}_1, \ldots, \mathcal{S}_j), \mathcal{M}$)
2:     output ← $\emptyset$                          ▷ output initialized with the empty set of constraints
3:     longest ← LONGEST($\mathcal{S}_1, \mathcal{M}$)                          ▷ longest interval identified
4:     baseline ← longest.upper                          ▷ where to extend the coverage from
5:     while $[\![\text{baseline}]\!]_\mathcal{M} \notin [\![\text{longest}]\!]_\mathcal{M}$ do
6:         if $\exists I \in \mathcal{S}_1, [\![\text{baseline}]\!]_\mathcal{M} \in [\![I]\!]_\mathcal{M}$ then
7:             $I$ ← FURTHEST_EXTEND(baseline, $\mathcal{S}_1, \mathcal{M}$)
8:             output ← output ∪ $\{c_I$, baseline $\in I\}$          ▷ adding $I$'s condition and linking constraint
9:             baseline ← $I$.upper                    ▷ updating the baseline for the next interval pick
10:        else                    ▷ there is a hole in the coverage of $\mathbb{Z}/2^{w_1}\mathbb{Z}$ by intervals in $\mathcal{S}_1$
11:            next ← NEXT_COVERED_POINT(baseline, $\mathcal{S}_1, \mathcal{M}$)          ▷ the hole is [baseline ; next[
12:            if $[\![\text{next}]\!]_\mathcal{M} - [\![\text{baseline}]\!]_\mathcal{M} <^{\mathsf{u}} 2^{w_2}$ then
13:                $I$ ← [next[$w_2$:] ; baseline[$w_2$:][          ▷ it is projected on $w_2$ bits and complemented
14:                output ← output ∪ {next−baseline $<^{\mathsf{u}} 2^{w_2}$} ∪ COVER($((\mathcal{S}_2 \cup I), \mathcal{S}_3, \ldots, \mathcal{S}_j), \mathcal{M}$)
15:                baseline ← next                    ▷ updating the baseline for the next interval pick
16:            else                    ▷ intervals of bitwidths $\leq w_2$ must forbid all values for $y[w_2{:}]$
17:                return COVER($(\mathcal{S}_2, \ldots, \mathcal{S}_j), \mathcal{M}$)                    ▷ $\mathcal{S}_1$ was not needed
18:    return output ∪ {baseline ∈ longest}                    ▷ adding final linking constraint
```

domain (i.e., $\bigcup_{I \in \mathcal{S}_i} [\![I]\!]_\mathcal{M}$ may be different from $\mathbb{Z}/2^{w_i}\mathbb{Z}$). The coverage can leave "holes", and values in that hole are ruled out by constraints of other bitwidths. To produce the interpolant, we adapt the coverage-extraction algorithm into Algorithm 3, which takes as input the sequence of sets $(\mathcal{S}_1, \ldots, \mathcal{S}_j)$ as described in Figure 2, and produces the interpolant's constraints $d_1, \ldots, d_p$, collected in set output. The algorithm proceeds in decreasing bitwidth order, starting with w_1, and calling itself recursively on smaller bitwidths to cover the holes that the current layer leaves uncovered (termination of that recursion is thus trivial). For every hole that $\bigcup_{I \in \mathcal{S}_1} [\![I]\!]_\mathcal{M}$ leaves uncovered, it must determine how intervals of smaller bitwidths can cover it.

Algorithm 3 relies on the following ingredients:

- LONGEST($\mathcal{S}, \mathcal{M}$) returns an interval among $\mathcal{S}$ whose concrete version $[\![I]\!]_\mathcal{M}$ has maximal length;
- I.upper denotes the upper bound of an interval I;
- FURTHEST_EXTEND($a, \mathcal{S}, \mathcal{M}$) returns an interval $I \in \mathcal{S}$ that furthest extends a according to $\mathcal{M}$ (technically, an interval I that $\leq^{\mathsf{u}}$-maximizes $[\![I.\text{upper} - a]\!]_\mathcal{M}$ among those intervals I such that $[\![a]\!]_\mathcal{M} \in [\![I]\!]_\mathcal{M}$).
- If no interval in $\mathcal{S}$ covers a in $\mathcal{M}$, NEXT_COVERED_POINT($a, \mathcal{S}, \mathcal{M}$) outputs the lower bound l of an interval in $\mathcal{S}$ that $\leq^{\mathsf{u}}$-minimizes $[\![l - a]\!]_\mathcal{M}$.

Algorithm 3 proceeds by successively moving a concrete bitvector value baseline around the circle $\mathbb{Z}/2^{w_1}\mathbb{Z}$. The baseline is moved when a symbolic reason why it is a forbidden value is found, in a while loop that ends when the baseline has gone round the full circle. If there is at least one interval in $\mathcal{S}_1$ that covers baseline in $\mathcal{M}$ (l. 6), the call to FURTHEST_EXTEND(baseline, $\mathcal{S}_1, \mathcal{M}$) succeeds, and output is extended with condition c_I and (baseline $\in I$) (l. 8). If not, a hole has been discovered, whose extent is given by NEXT_COVERED_POINT(baseline, $\mathcal{S}_1, \mathcal{M}$) (l. 11). If the hole is bigger than 2^{w_2} (i.e., $2^{w_2} \leq^{\mathsf{u}} [\![\text{next}-\text{baseline}]\!]_\mathcal{M}$), then the intervals of layers w_2 and smaller must rule out every possible value for $y[w_2{:}]$, and the w_1-intervals were not needed (l. 17). If on the contrary the hole is smaller (i.e., $[\![\text{next}-\text{baseline}]\!]_\mathcal{M} <^{\mathsf{u}} 2^{w_2}$), then the w_1-interval [baseline ; next[is projected

$u_1 <^{\mathsf{s}} u_2$	$\rightsquigarrow$	$\neg(u_2 \le^{\mathsf{s}} u_1)$		$u_1 \le^{\mathsf{s}} u_2$	$\rightsquigarrow$	$u_1{+}2^{\lvert u_1\rvert-1} \le^{\mathsf{u}} u_2{+}2^{\lvert u_2\rvert-1}$	
$u_1 <^{\mathsf{u}} u_2$	$\rightsquigarrow$	$\neg(u_2 \le^{\mathsf{u}} u_1)$		$u_1 \simeq u_2$	$\rightsquigarrow$	$u_1 - u_2 \le^{\mathsf{u}} 0$	
$u[h{:}l]$	$\rightsquigarrow$	$u[h{:}][{:}l]$		$u[{:}l][h{:}]$	$\rightsquigarrow$	$u[h{+}l{:}][{:}l]$	
$(u_1{\circ}u_2)[{:}l]$	$\rightsquigarrow$	$u_1[{:}l{-}\lvert u_2\rvert]$	if $\lvert u_2\rvert \le l$	$(u_1{\circ}u_2)[h{:}]$	$\rightsquigarrow$	$u_2[h{:}]$	if $h \le \lvert u_2\rvert$
$(u_1{\circ}u_2)[{:}l]$	$\rightsquigarrow$	$u_1 \circ u_2[{:}l]$	if not	$(u_1{\circ}u_2)[h{:}]$	$\rightsquigarrow$	$u_1[h{-}\lvert u_2\rvert{:}] \circ u_2$	if not
$2^n{\times}u$	$\rightsquigarrow$	$u[\lvert u\rvert{-}n{:}] \circ 0_n$	$(n < \lvert u\rvert)$	$(u_1{+}u_2)[h{:}]$	$\rightsquigarrow$	$u_1[h{:}] + u_2[h{:}]$	
$\text{bvnot}(u)$	$\rightsquigarrow$	$-(u+1)$		$(u_1{\times}u_2)[h{:}]$	$\rightsquigarrow$	$u_1[h{:}]{\times}u_2[h{:}]$	
$\pm\text{-ext}_k(u)$	$\rightsquigarrow$	$(0_k{\circ}(u{+}2^{\lvert u\rvert-1})){-}(0_k{\circ}2^{\lvert u\rvert-1})$		$(-u)[h{:}]$	$\rightsquigarrow$	$-u[h{:}]$	
$u_1{\circ}u_2$	$\rightsquigarrow$	$(u_1{\circ}0_{\lvert u_2\rvert}) + (0_{\lvert u_1\rvert}{\circ}u_2)$					

Fig. 3: Rewriting rules

as a w_2-interval $I := [\mathsf{baseline}[w_2{:}]\,;\mathsf{next}[w_2{:}][$ that needs to be covered by the intervals of bitwidth w_2 and smaller. This is performed by a recursive call on bitwidth w_2 (l. 14); the fact that only hole I needs to be covered by the recursive call, rather than the full domain $\mathbb{Z}/2^{w_2}\mathbb{Z}$, is implemented by adding to $\mathcal{S}_2$ in the recursive call the complement $[\mathsf{next}[w_2{:}]\,;\mathsf{baseline}[w_2{:}][$ of I. The result of the recursive call is added to the output variable, as well as the fact that the hole must be small. The final interpolant is $(\bigwedge_{d\in\mathsf{output}} d) \Rightarrow \bot$. An example of run on a variant of Example 6.2 is given in [17].

5 Normalization

As implemented in Yices 2, MCSAT processes a conflict by first computing the conflict core with BDDs, and then normalizing the constraints using the rules of Figure 3. In the figure, u, u_1 and u_2 stand for any bitvector terms, $\pm\text{-ext}_k(u)$ is the *sign-extension* of u with k bits, and $\text{bvnot}(u)$ is the *bitwise negation* of u. The bottom left rule is applied with lower priority than the others (as upper-bits extraction distributes over $\circ$ but not over $+$) and only if exactly one of $\{u_1, u_2\}$ is evaluable (and not 0). In the implementation, $u[|u|{:}0]$ is identified with u, $\circ$ is associative, and $+$, $\times$ are subject to ring normalization. This is helped by the internal (flattened) representation of concatenations and bitvector polynomials in Yices 2. Normalization allows the specialized interpolation procedure to apply at least to the following grammar:[5]

$$\begin{array}{lll} \text{Atoms} & a ::= e_1 + t \lessdot e_2 + t \mid e_1 \lessdot e_2 + t \mid e_1 + t \lessdot e_2 \mid e_1 \lessdot e_2 \\ \text{Terms} & t \ ::= t[h{:}l] \mid t + e_1 \mid -t \mid e_1 \circ t \mid t \circ e_1 \mid \pm\text{-ext}_k(t) \end{array}$$

where $\lessdot \in \{\le^{\mathsf{u}}, <^{\mathsf{u}}, \le^{\mathsf{s}}, <^{\mathsf{s}}, \simeq\}$. Rewriting can often help further, by eliminating occurrences of the conflict variable (thus making more subterms evaluable) and increasing the chances that two unevaluable terms t_1 and t_2 become syntactically equal in an atom $e_1{+}t_1 \lessdot e_2{+}t_2$.[6] Finally, we cache evaluable terms to avoid recomputing conditions of the form $y \notin \mathsf{var}(e)$. These conditions are needed to determine whether the specialized procedures apply to a given conflict core.

[5] $e_1 \lessdot e_2$ is accepted since it either constitutes the interpolant or it can be ignored.

[6] For this reason we normalize evaluable subterms of, e.g., t_1 and t_2.

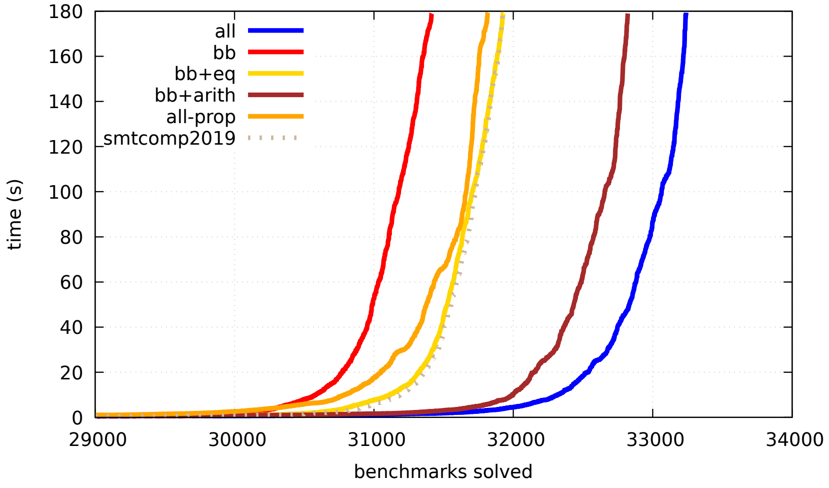

Fig. 4: Evaluation of the MCSAT solver and the effect of different explainer combinations and propagation. Each curve shows the number of benchmarks that the solver variant can solve against the time.

6 Experiments

We implemented our approach in the MCSAT solver within Yices 2 [13]. To evaluate its effectiveness, and the impact of the different modules, we ran the MCSAT solver with different settings on the 41,547 QF_BV benchmarks available in the SMT-LIB library [1]. We used a three-minute timeout per instance. Each curve in Figure 4 shows the number of solved instances for each solver variant; all: the procedures of Sections 3 and 4, with the bitblasting baseline when these do not apply; bb: only the bitblasting baseline; bb+eq: procedure of Section 3 plus the baseline; bb+arith: procedure of Section 4 plus the baseline; all-prop is the same as all but with no propagation of bitvector assignments during search. For reference, we also included the version of the Yices 2 MCSAT solver that entered the 2019 SMT competition[7], marked as smtcomp2019.

The solver combining all explainer modules solved 33,236 benchmarks before timeout, 14,174 of which are solved by pure simplification, and 19,062 of which actually rely on MCSAT explanations. 14,313 of those are solved without ever calling the default bitblasting baseline (only the dedicated explainers of Sections 3 and 4 are used), while the other 4,749 instances are solved by a combination of the three explainers.

The results show that both equality and arithmetic explainers contribute to the effectiveness of the overall solver, individually and combined. A bit more than half of the problem instances involving MCSAT explanations are fully within the scope of the two dedicated explainers. Of course these explainers are still useful beyond that half, in combination with the bitblasting explainer. The results also

[7] https://smt-comp.github.io/2019/

show that the eager MCSAT value propagation mechanism introduced in [21] is important for effective solving in practice.

For comparison, we also ran two solvers CDCL($\mathcal{T}$) solvers based on bitblasting on the same benchmarks and with the same timeout. We picked Yices 2 [13] (version 2.6.1) and Boolector [29] (version 3.2.0) and we used the same backend SAT solver for both, namely CaDiCaL [7]. Yices 2 solved 40,962 instances and Boolector solved 40,763 instances. We found 794 instances in the SMTLib benchmarks where our MCSAT solver was faster than Boolector by more than 2 sec. The `pspace/ndist*` and `pspace/shift1add*` instances are trivial for MCSAT (solved in less than 0.25 sec. each), while Boolector hit our 3-minute timeout on all `ndist.a.*` instances and all but 3 `shift1add*` ones. The `brummayerbiere4` instances are trivial for MCSAT (solved in less than 0.03 sec.) while Boolector ran out of memory in our experimentation (except for one instance). Instances with a significant runtime difference in favour of MCSAT are among `spear/openldap_v2.3.35/*` and `brummayerbiere/bitrev*` (MCSAT is systematically better), `float/mult*` (MCSAT is almost systematically better), `float/div*`, `asp/SchurNumbers/*`, `20190311-bv-term-small-rw-Noetzli/*`, and `Sage2/*`. MCSAT is almost systematically faster on `uclid/catchconv/*` and faster on more than half of `spear/samba_v3.0.24/*`.

Using an alternative MCSAT approach to bitvector solving, Zeljić et al. reported that their solver could solve 23704 benchmarks from a larger set of 49971 instances with a larger timeout of 1200s [32].[8] We have not managed to reproduce the results of Zeljić's solver on our Linux server for direct comparison.

To debug the implementation of our explainers, every conflict explanation that is produced when solving in debug mode is sent on-the-fly to (non-MCSAT) Yices 2, which checks the validity of the clause by bitblasting. In debug mode, every normalization we perform with the rules of Section 5 is also sent to Yices 2 to prove the equality between the original term and the normalized term. Performance benchmarking was only done after completing, without any red flag, a full run of MCSAT in debug mode on the 41,547 QF_BV benchmarks instances.

7 Discussion and Future Work

The paper presents ongoing work on building an MCSAT solver for the theory of bitvectors. We have presented two main ideas for the treatment of $\mathcal{BV}$ in MCSAT, that go beyond the approach proposed by Zeljić et al. [32].

First, by relying on BDDs for representing feasible sets, our design keeps the main search mechanism of MCSAT generic and leaves fragment-specific mechanisms to conflict explanation. The explanation mechanism is selected based on the constraints involved in the conflict. BDDs are also used to minimize the conflicts, which increases the chances that a dedicated explanation mechanism can be applied. BDDs offer a propagation mechanism that differs from those in [32] in that the justification for a propagated assignment is computed lazily, only

[8] The additional 8424 benchmarks have since been deleted from the SMT-LIB library as duplicates.

when it is needed in conflict analysis. Computing the conflict core at that point effectively recovers justification of the propagations.

Second, we propose explanation mechanisms for two fragments of the theory: the core fragment of $\mathcal{BV}$ that includes equality, concatenation and extraction; and a fragment of linear arithmetic. Compared to previous work on coarsest-base slicing, such as [5], our work applies the slicing on the conflict constraints only, rather than the whole problem. This should in general make the slices coarser, which we expect to positively impact efficiency. Our work on explaining arithmetic constraints is novel, notwithstanding the mechanisms studied by Janota and Wintersteiger [19] that partly inspired our Table 1 but addressed a smaller fragment of arithmetic outside of the context of MCSAT.

We have implemented the overall approach in the Yices 2 SMT solver. Experiments show that the overall approach is effective on practical benchmarks, with all the proposed modules adding to the solver performance. MCSAT is not yet competitive with bitblasting, but we are making progress. The main challenge is devising efficient word-level explanation mechanisms that can handle all or a least a large fragment of $\mathcal{BV}$. Finding high-level interpolants in $\mathcal{BV}$ is still an open problem and our work on MCSAT shows progress for some fragments of the bitvector theory. For MCSAT to truly compete with bitblasting, we will need interpolation methods that cover larger classes of constraints.

A key step in that direction is to extend the bitvector arithmetic explainer so that it handles multiplications by constants, then multiplication by evaluable terms, and, finally, arbitrary multiplications. Deeper integration of fragment-specific explainers could potentially help explaining *hybrid* conflicts that involve constraints from different fragments. To complement the explainers that we are developing, we plan to further explore the connection between interpolant generation and the closely related domain of quantifier elimination, particularly those techniques by John and Chakraborty [20] for the bitvector theory. The techniques by Niemetz et al. [28] for solving quantified bitvector problems using *invertibility conditions* could also be useful for interpolant generation in MCSAT.

Future work also includes relating our approach to the report by Chihani, Bobot, and Bardin [8], which aims at lifting the CDCL mechanisms to the word level of bitvector reasoning, and therefore seems very close to MCSAT. Finally, we plan to explore integrating our MCSAT treatment of bitvectors with other components of SMT-solvers, whether in the context of MCSAT or in different architectures. An approach for this is the recent framework of *Conflict-Driven Satisfiability* (CDSAT) [3, 4], which precisely aims at organizing collaboration between generic theory modules.

Acknowledgments The authors thank Aleksandar Zeljić for fruitful discussions. This material is based upon work supported in part by NSF grants 1528153 and 1816936, and by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, or SSC Pacific.

References

1. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2010), www.SMT-LIB.org 3, 14
2. Biere, A., Bloem, R. (eds.): Proc. of the 26th Int. Conf. on Computer Aided Verification (CAV'14), LNCS, vol. 8559. Springer-Verlag (Jul 2014). https://doi.org/10.1007/978-3-319-08867-9 17, 18
3. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Conflict-driven satisfiability for theory combination: Transition system and completeness. J. of Automated Reasoning **64**(3), 579–609 (2019). https://doi.org/10.1007/s10817-018-09510-y 16
4. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Satisfiability modulo theories and assignments. In: de Moura, L. (ed.) Proc. of the 26th Int. Conf. on Automated Deduction (CADE'17). LNAI, vol. 10395, pp. 42–59. Springer-Verlag (Aug 2017) 16
5. Bruttomesso, R., Sharygina, N.: A scalable decision procedure for fixed-width bit-vectors. In: Proc. of the 2009 Int. Conf. on Computer-Aided Design (ICCAD'09). pp. 13–20. ICCAD'09, ACM Press (2009). https://doi.org/10.1145/1687399.1687403 1, 2, 5, 16
6. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on **100**(8), 677–691 (1986) 2, 4
7. CaDiCaL Simplified Satisfiability Solver, http://fmv.jku.at/cadical/ 15
8. Chihani, Z., Bobot, F., Bardin, S.: CDCL-inspired Word-level Learning for Bit-vector Constraint Solving (Jun 2017), https://hal.archives-ouvertes.fr/hal-01531336, preprint 16
9. Chockler, H., Weissenbacher, G. (eds.): Proc. of the 30th Int. Conf. on Computer Aided Verification (CAV'18), LNCS, vol. 10982. Springer-Verlag (Jul 2018). https://doi.org/10.1007/978-3-319-96142-2, https://doi.org/10.1007/978-3-319-96142-2 18
10. CUDD: the CU Decision Diagram package, https://github.com/ivmai/cudd 4
11. Cyrluk, D., Möller, O., Rueß, H.: An efficient decision procedure for the theory of fixed-sized bit-vectors. In: Grumberg, O. (ed.) Proc. of the 9th Int. Conf. on Computer Aided Verification (CAV'97). LNCS, vol. 1254, pp. 60–71. Springer-Verlag (1997). https://doi.org/10.1007/3-540-63166-6_9 2, 5
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM (JACM) **52**(3), 365–473 (2005) 2, 6
13. Dutertre, B.: Yices 2.2. In: Biere and Bloem [2], pp. 737–744. https://doi.org/10.1007/978-3-319-08867-9_49 2, 14, 15
14. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) Proc. of the 19th Int. Conf. on Computer Aided Verification (CAV'07). LNCS, vol. 4590, pp. 519–531. Springer-Verlag (2007). https://doi.org/10.1007/978-3-540-73368-3 1
15. Graham-Lengrand, S., Jovanović, D.: An MCSAT treatment of bit-vectors. In: Brain, M., Hadarean, L. (eds.) 15th Int. Work. on Satisfiability Modulo Theories (SMT 2017) (Jul 2017) 2
16. Graham-Lengrand, S., Jovanović, D.: Interpolating bit-vector arithmetic constraints in MCSAT. In: Sharygina, N., Hendrix, J. (eds.) 17th Int. Work. on Satisfiability Modulo Theories (SMT 2019) (Jul 2019) 2
17. Graham-Lengrand, S., Jovanović, D., Dutertre, B.: Solving bitvectors with MCSAT: explanations from bits and pieces (long version). Tech. rep., SRI International (Apr 2020), https://arxiv.org/abs/2004.07940 8, 11, 13

18. Hadarean, L., Bansal, K., Jovanović, D., Barrett, C., Tinelli, C.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: Biere and Bloem [2], pp. 680–695. https://doi.org/10.1007/978-3-319-08867-9 1
19. Janota, M., Wintersteiger, C.M.: On intervals and bounds in bit-vector arithmetic. In: King, T., Piskac, R. (eds.) Proc. of the 14th Int. Work. on Satisfiability Modulo Theories (SMT'16). CEUR Workshop Proceedings, vol. 1617, pp. 81–84. CEUR-WS.org (Jul 2016), http://ceur-ws.org/Vol-1617/paper8.pdf 16
20. John, A.K., Chakraborty, S.: A layered algorithm for quantifier elimination from linear modular constraints. Formal Methods in System Design **49**(3), 272–323 (Dec 2016). https://doi.org/10.1007/s10703-016-0260-9, https://doi.org/10.1007/s10703-016-0260-9 16
21. Jovanović, D.: Solving nonlinear integer arithmetic with MCSAT. In: Bouajjani, A., Monniaux, D. (eds.) Proc. of the 18th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'17). LNCS, vol. 10145, pp. 330–346. Springer-Verlag (Jan 2017). https://doi.org/10.1007/978-3-319-52234-0_18 1, 3, 15
22. Jovanović, D., Barrett, C., de Moura, L.: The design and implementation of the model constructing satisfiability calculus. In: Proc. of the 13th Int. Conf. on Formal Methods In Computer-Aided Design (FMCAD'13). FMCAD Inc. (Oct 2013), http://www.cs.nyu.edu/~barrett/pubs/JBdM13.pdf 1, 3
23. Junker, U.: Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI'01 Workshop on Modelling and Solving problems with constraints (2001) 4
24. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag (2016). https://doi.org/10.1007/978-3-662-50497-0, https://doi.org/10.1007/978-3-662-50497-0 1
25. Marques Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., Maaren, H.V., Walsh, T. (eds.) Handbook of S atisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009) 1
26. de Moura, L.M., Jovanovic, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Proc. of the 14th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'13). LNCS, vol. 7737, pp. 1–12. Springer-Verlag (Jan 2013). https://doi.org/10.1007/978-3-642-35873-9_1 1, 3
27. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014), https://satassociation.org/jsat/index.php/jsat/article/view/120 1
28. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler and Weissenbacher [9], pp. 236–255. https://doi.org/10.1007/978-3-319-96142-2_16, https://doi.org/10.1007/978-3-319-96142-2_16 16
29. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler and Weissenbacher [9], pp. 587–595. https://doi.org/10.1007/978-3-319-96145-3_32, https://doi.org/10.1007/978-3-319-96145-3_32 15
30. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. of the ACM Press **53**(6), 937–977 (2006). https://doi.org/10.1145/1217856.1217859 1

31. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier and The MIT Press (2001) 3
32. Zeljić, A., Wintersteiger, C.M., Rümmer, P.: Deciding bit-vector formulas with mcsat. In: Creignou, N., Berre, D.L. (eds.) Proc. of the 19th Int. Conf. on Theory and Applications of Satisfiability Testing (RTA'06). LNCS, vol. 9710, pp. 249–266. Springer-Verlag (Jul 2016). https://doi.org/10.1007/978-3-319-40970-2_16 2, 4, 5, 15