University Library



A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Wang, W;Søndergaard, H;Stuckey, PJ

Title:
Wombit: A Portfolio Bit-Vector Solver Using Word-Level Propagation

Date:
2019-10-15

Citation:
Wang, W., Søndergaard, H. & Stuckey, P. J. (2019). Wombit: A Portfolio Bit-Vector Solver Using Word-Level Propagation. Journal of Automated Reasoning, 63 (3), pp.723-762. https://doi.org/10.1007/s10817-018-9493-1.

Persistent Link:
https://hdl.handle.net/11343/228886

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Wombit: A Portfolio Bit-Vector Solver Using Word-Level Propagation

Wenxi Wang · Harald Søndergaard · Peter J. Stuckey

Received: date / Accepted: date

Abstract We develop an idea originally proposed by Michel and Van Hentenryck of how to perform bit-vector constraint propagation on the word level. Most operations are propagated in constant time, assuming the bit-vector fits in a machine word. In contrast, bit-vector SMT solvers usually solve bit-vector problems by (ultimately) bit-blasting, that is, mapping the resulting operations to conjunctive normal form clauses, and using SAT technology to solve them. Bit-blasting generates intermediate variables which can be an advantage, as these can be searched on and learnt about. As each approach has advantages, it makes sense to try to combine them. In this paper, we describe an approach to bit-vector solving using word-level propagation with learning. We have designed alternative word-level propagators to Michel and Van Hentenryck's, and evaluated different variants of the approach. We have also experimented with different approaches to learning and back-jumping in the solver. Based on the insights gained, we have built a portfolio solver, Wombit, which essentially extends the STP bit-vector solver. Using machine learning techniques, the solver makes a judicious up-front decision about whether to use word-level propagation or fall back on bit-blasting.

Keywords bit-vector solver, bit-blasting, constraint propagation, word-level reasoning, machine learning, portfolio solvers

CR Subject Classification **Mathematics of computing**: Solvers · **Theory of computation**: Logic and verification · **Theory of computation**: Automated

W. Wang
Department of Electrical and Computing Engineering, The University of Texas at Austin, Austin TX 78712, USA. E-mail: wenxiw@utexas.edu

H. Søndergaard
School of Computing and Information Systems, The University of Melbourne, Vic. 3010, Australia. E-mail: harald@unimelb.edu.au

P. J. Stuckey
Faculty of IT, Monash University, 900 Dandenong Road, Caulfield East, Vic. 3145, Australia. E-mail: peter.stuckey@monash.edu

reasoning · **Mathematics of computing**: Discrete mathematics · **Hardware**: Functional verification—*Theorem proving and SAT solving*

1 Introduction

Decision procedures for quantifier-free fragments of first-order theories, also known as satisfiability modulo theory (SMT) solvers, find widespread use in many areas, such as program analysis and verification, security, and scheduling. For example, in predicate abstraction program verification, tools like BLAST [15] and CPAchecker [16] use SMT solvers to decide validity of formulas that capture the effects of program statements. Program verifiers such as Why3 [17] and Boogie2 [45] use Z3 [23] as back-end solver. In symbolic execution, KLEE [19] uses STP [31] as its underlying constraint solver. In automatic exploit generation [10], SMT solvers are used to report bugs if some path is feasible but violates some safety property. In rotating workforce scheduling [27], the problem is transformed using an SMT encoding and ultimately solved by the SMT solver.

Of the many first-order theories for which SMT solvers are available, one of the most useful is the theory of quantifier-free bit-vector logic (QF_BV). Since most time-critical and safety-critical software is built on fixed-width integers, it is vital to reason about fixed-width integers correctly and accurately in a software verification context. In programming languages such as C, fixed-width integers and fixed-width arithmetic operations are naturally seen as bit-vectors and bit-vector operations. Many program analysis and verification tools fail to take this fixed-width reality into account, instead assuming ideal integers, and this is a source of unsoundness. The QF_BV theory is a basis for sound reasoning in the context of fixed-width integer problems.

Most bit-vector SMT solvers [12, 53, 25, 31, 46] ultimately rely on bit-blasting to solve bit-vector constraints, that is, translating constraints to propositional logic form. This tends to cause two problems. First, bit-blasting may result in very large propositional formulas that even the most powerful current SAT solvers struggle to handle. Second, it disperses important word-level information during the encoding—much is obscured in translation. Here we investigate alternatives to bit-blasting, replacing it with word-level propagation entirely to produce a pure word-level bit-vector SMT solver.

The use of word-level propagation in the context of bit-vector solvers was suggested by Michel and Van Hentenryck [50] who view the problem as a Constraint Satisfaction Problem (CSP). Each variable is associated with a "bit-vector domain" which is progressively tightened using word-level constraint propagation rules. In Section 4 we explain and extend this idea. It has considerable appeal, because the propagation rules can be made to run in constant time (as long as the bit-vectors are not longer than the size of a machine register). An additional rule to check if a tightened domain remains valid also runs in constant time.

Importantly, we add a "learning" mechanism to the originally proposed method, since it seems plausible that explanations for the propagated bits can be used to advantage. Section 4 explains propagation and the generation of explanations for bit-vector operations in some detail.

We investigate design choices for word-level solvers in Section 5. Alternative ("decomposed") word-level propagators are proposed for some operations, based on insights in Warren's compendium [70] and we investigate the relative strengths and weaknesses of decomposed and composed propagators. In a learning solver we can generate explanations in a "forward" manner, as propagation progresses, as is done in a SAT solver, or we can generate them in a "backward" manner during conflict analysis, as in a typical SMT solver. Forward explanation is simpler to implement, while backward explanation may require less explanation work overall.

Another design choice comes from a potential benefit of word-level propagation which is deeper conflict analysis. Normally, using bit-blasting, conflict analysis starts as soon as the first conflict is found. In the word-level solver, we could do the same, to find the first conflict clause and backtrack to the level indicated by this conflict (we call this "standard backjumping"). Alternatively, with word-level propagation, we can discover several conflicts at once, corresponding to several backtrack levels. We choose the smallest of these levels, in order to backtrack to the highest level of the search tree and add all the learnt clauses along the way to prevent all the conflicts from happening again (we call this "multi-conflict backjumping").

To construct the solver we have extended MiniSAT [26] so that it can keep track of opportunities for word-level propagation and intersperse this kind of propagation with unit propagation. Our word-level propagators contribute to MiniSAT's powerful search and learning mechanism by providing clauses as explanations for word-level propagated bits. In this way, the word-level propagators become lazy clause generators [57] for a SAT solver extended with constraint programming technology [62].

To evaluate the potential of word-level solving, we have performed a sequence of experiments. We have compared different combinations of features against a naive bit-blaster, to identify the best combinations (Section 5.4). In a second stage (Section 6) we apply the word-level simplification used in STP to our word-level solvers as preprocessing, and then compare the result with STP. The reason we choose STP is its focus on bit-vector logic and few other theories, and its powerful word-level simplification method that can solve purely linear problems outright. Note that we do not do bit-blasting or any Boolean simplification in the word-level solvers.

Our experimental results suggest that STP and word-level solvers are somewhat complementary. There are many test cases which STP is best at solving, and other cases where a word-level solver wins. This suggests the possibility of developing a portfolio solver that can combine solvers so that we can gain the benefit from each. This raises the problem of identifying which cases favour which solver. To solve this, in Section 7 we use machine learning techniques to classify the cases for each solver. Using a classifier generated

by machine learning, we have implemented a portfolio solver called Wombit which outperforms the component solvers (STP and two word-level solvers) significantly. Wombit and its source code is available at Github: `https://github.com/wenxiw2/Wombit`.

This paper is mostly a distillation of a masters thesis [68]; it extends a previous paper [69], the focus of which was the idea of word-level propagation in a bit-vector solver. In this previous work we:

- created a word-level propagating (but bit-level explaining) constraint solver;
- designed algorithms for generation of explanations for word-level propagators (restricted to linear arithmetic and bit-twiddling operations); and
- provided an empirical evaluation of the word-level propagation approach, compared with the standard bit-blasting approach to these problems.

In this paper we expand on the previous work [69] as follows:

1. We have embedded the word-level solving techniques into the QF_BV solver STP, without changing the architecture of either STP or the word-level solver much (suggesting that similar embeddings are possible with other SMT solvers).
2. We have added word-level propagators for non-linear arithmetic, that is, multiplication, unsigned and signed division, unsigned remainder, signed remainder (sign follows dividend/divisor).
3. We have extended the evaluation by comparing the performance of the STP solver with that of two variants of word-level solvers. Since we now cover all operations, including non-linear arithmetic, this evaluation utilises all folders in the QF_BV category of the standard SMT-LIB2 benchmarks. The only test cases we have had to exclude are those in which the bit-width for bit-vector operations exceeds 64 (the size of our machine registers).
4. We have analysed the solvers' relative performances and strengths, in an attempt to identify input features that set the solvers apart.
5. From this we have developed a portfolio solver which, based on characteristics of its input, chooses one of the solvers (STP or one of the two word-level solvers) for the task, using a model generated by machine learning techniques.

2 Preliminaries

2.1 Bit-Vectors and the SMT-LIB2 Quantifier-Free Bit-Vector Theory

We use 0 and 1 for the truth values (false and true, respectively). We shall need to distinguish word-level logical operations from Boolean operations carefully. As bit-wise operations we use $\sim$, &, |, and $\oplus$, for bit complement, conjunction, disjunction, and exclusive or, respectively. As Boolean connectives, we use $\neg$, $\wedge$ and $\vee$ for negation, conjunction, and disjunction, respectively.

A *bit-vector* $x_{[w]}$ is a sequence of w binary digits (bits). The elements of the sequence are indexed from right to left, starting with index 0 : $x = x_{w-1}...x_1x_0$. In addition, we use $c_{[w]}$ to denote a constant bit-vector with length w, and use b to denote a Boolean variable.

The operations in the QF_BV category of SMT-LIB2 can be divided into three categories: logical constraints, structural constraints and arithmetic constraints.

Logical Constraints. Logical constraints include:

- bitwise negation: (**bvnot** $x_{[w]}$) = $y_{[w]}$
- bitwise conjunction: (**bvand** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$
- bitwise disjunction: (**bvor** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$
- bitwise exclusive or: (**bvxor** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$
- bitwise nand (negation of and): (**bvnand** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$
- bitwise nor (negation of or): (**bvnor** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$
- bitwise equivalence (negation of xor): (**bvxnor** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$

Arithmetic Constraints. Arithmetic constraints include (fixed-width) linear and non-linear constraints. (For presentation purposes we find it convenient to include the **ite** constraint in this category.) The constraints are:

- if-then-else: (**ite** b, $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$; equivalent to ($b \wedge z_{[w]} = x_{[w]}$) $\vee$ ($\neg b \wedge z_{[w]} = y_{[w]}$).
- addition: (**bvadd** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$
- unary minus: (**bvneg** $x_{[w]}$) = $y_{[w]}$; equivalent to $y_{[w]} = \sim x_{[w]} + 1$
- subtraction: (**bvsub** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$; equivalent to $z_{[w]} = x_{[w]} + \sim y_{[w]} + 1$
- reified equality: (**=** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = b; equivalent to $\Big(b \wedge (x_{[w]} = y_{[w]}) \wedge (x_{[w]} = z_{[w]}) \wedge ... \Big) \vee \Big(\neg b \wedge (x_{[w]} \neq y_{[w]}) \wedge (x_{[w]} \neq z_{[w]}) \wedge ... \Big)$.
- reified disequality: (**distinct** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = b; equivalent to $\Big(b \wedge (x_{[w]} \neq y_{[w]}) \wedge (x_{[w]} \neq z_{[w]}) \wedge ... \Big) \vee \Big(\neg b \wedge (x_{[w]} = y_{[w]}) \wedge (x_{[w]} = z_{[w]}) \wedge ... \Big)$.
- unsigned less than or equals: (**bvule** $x_{[w]}$, $y_{[w]}$) = b
- unsigned less than: (**bvult** $x_{[w]}$, $y_{[w]}$) = b
- unsigned greater than equals: (**bvuge** $x_{[w]}$, $y_{[w]}$) = b
- unsigned greater than: (**bvugt** $x_{[w]}$, $y_{[w]}$) = b
- corresponding signed inequality constraints: (**bvsle** $x_{[w]}$, $y_{[w]}$) = b, (**bvslt** $x_{[w]}$, $y_{[w]}$) = b, (**bvsge** $x_{[w]}$, $y_{[w]}$) = b, and (**bvsgt** $x_{[w]}$, $y_{[w]}$) = b. Signed inequality constraints can be translated into unsigned inequality constraints. For instance, b = (**bvslt** $x_{[w]}$, $y_{[w]}$) is equivalent to (**not** (**bvule** $y_{[w]}$, $x_{[w]}$)) $\oplus\, x_{w-1} \oplus y_{w-1}$.

The semantics of the non-linear arithmetic constraints are as follows:

- (**bvmul** $x_{[w]}$, $y_{[w]}$, $z_{[w]}$, ...) = $n_{[w]}$: multiplication modulo 2^w.
- (**bvudiv** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: unsigned division, truncating towards 0 (undefined if divisor is 0).
- (**bvurem** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: unsigned remainder after truncating division (undefined if divisor is 0).
- (**bvsdiv** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: two's complement signed division.

- (**bvsrem** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: two's complement signed remainder (sign follows dividend).
- (**bvsmod** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: two's complement signed remainder (sign follows divisor).

Structural Constraints. The structural constraints include:

- left shift: (**bvshl** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: left shift $x_{[w]}$ by the number of $y_{[w]}$ bit positions to get $z_{[w]}$, which is equivalent to the multiplication of $x_{[w]}$ by $2^{y_{[w]}}$.
- logical right shift: (**bvlshr** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: logical right shift $x_{[w]}$ by the number of $y_{[w]}$ bit positions to get $z_{[w]}$, which is equivalent to the unsigned division of $x_{[w]}$ by $2^{y_{[w]}}$.
- arithmetic right shift: (**bvashr** $x_{[w]}$, $y_{[w]}$) = $z_{[w]}$: like the logical right shift, except that the most significant bits ($z_{w-1}...z_{(w-y_{[w]})}$) of $z_{[w]}$ always copy the most significant bit of $x_{[w]}$ (x_{w-1}).
- left rotation: (**rotate_left** n, $x_{[w]}$) = $y_{[w]}$: left rotate $x_{[w]}$ by n bit positions to get $y_{[w]}$.
- right rotation: (**rotate_right** n, $x_{[w]}$) = $y_{[w]}$: right rotate $x_{[w]}$ by n bit positions to get $y_{[w]}$.
- concatenation: (**concat** $x_{[w_1]}$, $y_{[w_2]}$) = $z_{[w_1+w_2]}$: $z_{[w_1+w_2]}$ is the concatenation of bit-vectors x and y.
- repeat: (**repeat** n, $x_{[w]}$) = $y_{[w*n]}$: $y_{[w*n]}$ is the concatenation of n copies of $x_{[w]}$.
- extraction: (**extract** n, m, $x_{[w]}$) = $y_{[n-m+1]}$: $y_{[n-m+1]}$ is the extraction of bits n down to m from $x_{[w]}$.
- zero extension: (**zero_extend** n, $x_{[w]}$) = $y_{[w+n]}$: $y_{[w+n]}$ is the n-bit unsigned equivalent of bit-vector $x_{[w]}$.
- signed extension: (**sign_extend** n, $x_{[w]}$) = $y_{[w+n]}$: $y_{[w+n]}$ is the $(w+n)$-bit signed equivalent of bit-vector $x_{[w]}$.

2.2 Propagation-Based Constraint Solving

A common method employed in tackling combinatorial problems over finite domains is to enhance systematic search with cheap (but generally incomplete) reasoning techniques, such as local search and *constraint propagation*. Propagation assumes that each constraint variable has an associated *domain*: a set of possible values that the variable can take. The real information that resides in a domain is its complement—the values that the variable can definitely not take. Propagation is the process of taking a constraint and decreasing the domains of its variables, in a sound manner, according to what the constraint expresses. For example, if x currently has domain $\{3,4,5\}$ and y has domain $\{1,2,3,4\}$, the constraint $x < y$ allows us to tighten the domains to $\{3\}$ and $\{4\}$ respectively. The job of a *propagator* for the constraint is to perform this kind of tightening.

A propagation-based solver interleaves propagation and search, typically resorting to the latter only when the former fails to make progress. Unit

propagation is a well-known example in SAT solving, but the propagation concept generalises to other kinds of constraint satisfaction problems. We here take a propagator for a constraint c to be a contracting, monotone, idempotent function f on domains. That is, it makes the given domain smaller if it changes it at all (contraction); it does not squander information (monotonicity); and it performs in one attempt all the contraction that it is capable of (idempotence).[1]

A propagator usually does more than propagate information. Two important roles are conflict checking and explanation generation. A conflict occurs when the domain of some variable becomes empty. An explanation is an additional (logically redundant) constraint added for the purpose of pruning search.

From the definition of a propagator, it is clear that the identity function is a propagator, albeit not a very useful one. To get a handle on propagation *strength*, various "consistency" notions are used. An n-ary constraint $C(x_1,\ldots,x_n)$ is *domain consistent* with respect to the domain product $D_1\times\cdots\times D_n$ iff, for each variable x_i and value $v\in D_i$, the constraint $C(v_1,\ldots,v_{i-1},v,v_{i+1},\ldots,v_n)$ holds for some $v_1\in D_1,...,v_{i-1}\in D_{i-1},v_{i+1}\in D_{i+1},\ldots,v_n\in D_n$. That is, none of the domains $D_1,\ldots,D_n$ can be tightened based on $C(x_1,...,x_n)$.

In the context of bit-vector constraints, a weaker property of *bit-consistency* is proposed by Michel and Van Hentenryck [50]. It rests on a notion of domain which is equivalent to the domain of "trit-vectors" that we introduce in Section 4.1. In analogy with domain consistency, bit consistency guarantees that no free bit position in any bit-vector variable can be fixed, based on $C(x_1,...,x_n)$. The aim is for propagators (for the different kinds of constraints) to achieve bit consistency.

3 Overview of the Word-Level Solver

3.1 The Architecture and Solving Algorithm of MiniSAT

MiniSAT [26] is a small, complete, and efficient SAT solver which was designed with domain specific extension in mind. It performs a SAT solving algorithm as outlined in Algorithm 1 [26, 43]. Three important components of the architecture are:

- A *propagation queue* ($BPQueue(\ell)$) is used for unit propagation; it keeps track of the order of the assigned literals at the current level. The assigned literal is enqueued into the propagation queue once decided or implied, and is dequeued from the propagation queue to implicate new literals if possible.
- A *watcher list* ($BWatch(\ell)$) is constructed from the clauses of the input CNF formula, and adds the clauses to the related literal dynamically, based on the two watched literal scheme [51] for effective unit propagation.

[1] In some formalisations of propagation, the idempotence requirement is dropped, primarily for technical reasons—to ensure that the set of propagators is closed under function composition.

Algorithm 1 General algorithm for SAT solver and the propagation solvers

```
add the input into the system                                  ▷ initialization
if PROPAGATE() ≠ true then                                     ▷ top level conflict
    return UNSAT
while true do
    if PROPAGATE() = true then                                 ▷ no conflict
        if ¬decide() then                                      ▷ all variables are assigned
            return SAT
        else
            decide()
            dl ← dl + 1          ▷ Increment decision level due to new decision
    else                                                       ▷ conflict happens
        if top-level conflict found then
            return UNSAT
        else
            learnt_clause := conflict_analyze()
            backjump(learnt_clause)          ▷ dl is decremented due to backjumping
```

– A *reason array* (*Reasons*(*b*)) stores the antecedent clause (the *explanation*) for each implicated variable, for the purpose of conflict analysis. For a decided variable, and for the variable of a unit clause, the reason array holds *null*.

3.2 The Extended Architecture and Solving Algorithm

The extended architecture for our word-level SAT based solver is shown in Figure 1. The top part above the dashed line is largely the architecture of a typical SAT solver (we have used MiniSAT), but extended with a map from each individual bit of an integer to that integer. This is the $b \rightarrow w$ *map* box on the left, which we refer to as the word *origin* map. The input to the word-level solver can be a mixture of word-level formulas (bit-vector constraints, including arithmetic constraints) and propositional formulas in CNF. Only the CNF formulas are propagated through unit propagation inside the SAT solver. The word-level formulas are handled by the corresponding so-called "word-level propagators" (explained below) without any bit-blasting. The added components for word-level solving are:

– The *word-level watch list* (*WWatch*(*x*)) is constructed by the propagators of the input word-level operations, and adds the propagators to the related bit-vector statically at the beginning.
– The *word-level propagator queue* (*WPQueue*) is used for word-level propagation and keeps track of the order of the enqueued propagators at the current level, ensuring that an already enqueued propagator does not appear multiple times. In our implementation, the word-level propagator queue has a lower priority than the Boolean propagation queue.
– A *word-level propagator* performs word-level propagation and conflict checking. (The exact workings of these propagators will become clear in Section 4.) If a conflict is detected, the propagator generates a conflict clause ③; otherwise it sends the fixed literal(s) ① to the Boolean propagation

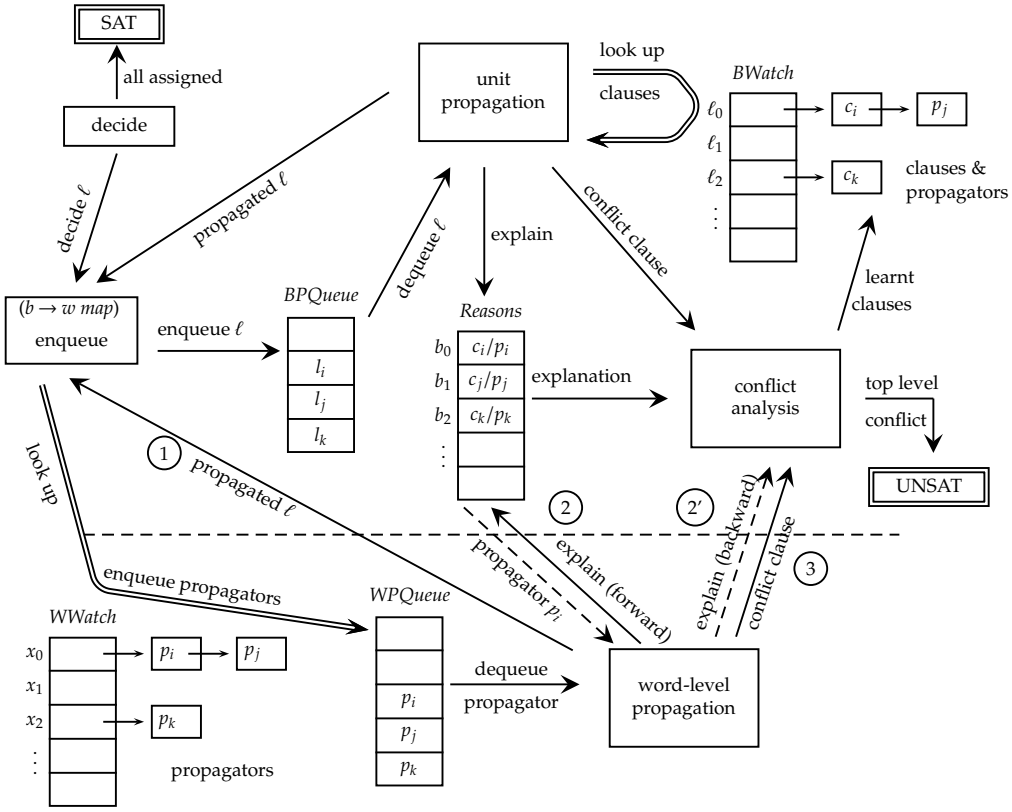


Fig. 1 Overall architecture: MiniSAT (top) and word-level mechanism (bottom)

queue and generates explanation(s) ②/②′ for why the literal(s) became true. (The ②/②′ distinction will be explained in Section 3.3.) A word-level propagator $p(b, x_1, \ldots, x_k)$ which involves a Boolean parameter b is not only statically stored in the word-level watch list, but also *statically* stored in the original Boolean watch list. For example, the propagator $p(b, x_1, x_2, x_3)$ for the if-then-else constraint $\mathbf{ite}(b, x_1, x_2) = x_3$ is in the word-level lists for x_1, x_2 and x_3, and also in the Boolean lists for b and $\neg b$.

The extended solving process is shown in Algorithm 2. We extend the enqueue function of MiniSAT by adding the word origin map $(b \to w)$ mentioned above. When a bit ℓ of an integer x is newly decided or propagated, this literal is enqueued into the Boolean propagation queue, and all the related word-level propagators of integer x in the word-level watch list are enqueued into the propagator queue.

When a literal is dequeued from the Boolean propagation queue, either the corresponding Boolean constraints in the Boolean watch list are invoked to do the unit propagation, or the propagators in the word-level watch list are

Algorithm 2 Extended solving process in word-level SAT based solver

```
function ENQUEUE(literal ℓ, clause C, prop p)
    BPQueue.enqueue(ℓ)
    b ← var(ℓ)                                   ▷ get the corresponding Boolean variable b
    Reasons[b] ← C/p      ▷ forward manner: add the explanation C for b to the reason array
                  ▷ backward manner: add the propagator p which fixed ℓ to the reason array
    if ℓ is in a bit-vector then
        x ← word(b)                              ▷ get the corresponding bit-vector x
        for p in WWatch(x) do
            if p is not in WPQueue then
                WPQueue.enqueue(p)               ▷ enqueue propagators not in WPQueue
function PROPAGATE( )
    clause confl ← null
    while confl = null do                                          ▷ no conflict
        while ¬BPQueue.isEmpty() ∧ confl = null do
            ℓ ← BPQueue.dequeue()
            if BWatch(l) is a propagator p then
                WPQueue.enqueue(p)
            else if BWatch(l) is a clause c then
                confl ← unit_prop(c)
        if confl = null then                     ▷ BPQueue is empty, no conflict
            p ← WPQueue.dequeue()
            confl ← word_prop(p)
    return confl
```

enqueued into the propagator queue. Only when the Boolean propagation queue is empty do we start to dequeue propagators from the propagator queue. We thus favour unit propagation since it is faster. Since previously learnt clauses are in the priority queue, previous conflicts can be avoided earlier this way.

As can be seen from Figure 1, the interactions between MiniSAT (top part) and the word-level mechanism (bottom part) are the propagated literals ①, explanations ②/②′, and the conflict clauses ③. Without these capabilities, word-level propagators cannot benefit from the learning capabilities of the SAT solver, including back-jumping and powerful adaptive search.

A broad-brush example will illustrate how the word-level propagators collaborate with MiniSAT. Suppose we need to solve a problem which includes the constraint $(x < y) \rightarrow (x < y + 1)$ (say x, y and 1 are bit-vectors of width 3). In the parsing phase we translate the constraint into four basic constraints by introducing two Boolean variables d and e, and one bit-vector variable z:

$$c_1 : y + 1 = z \qquad c_2 : d \leftrightarrow x < y \qquad c_3 : e \leftrightarrow x < z \qquad c_4 : d \rightarrow e$$

Unconstrained bit-level variables (x_0, x_1, x_2, y_0, y_1, y_2, z_0, z_1, z_2) are added to MiniSAT for x, y, and z respectively. In the origin map, each bit is mapped to its corresponding bit-vector. Suppose a decision assigns d to 1; so d is enqueued into *BPQueue*. Later, when d is dequeued, it will trigger unit propagation. Also, looking up d in *BWatch*, the propagator for the $d \rightarrow x < y$ constraint is found and enqueued into the propagator queue *WPQueue*. Later, when this

propagator is dequeued, it will perform the relevant word-level propagation. Now it may happen that, say, the bit x_1 is propagated by the propagator for $d \rightarrow x < y$. Then x_1 in turn is sent to the *BPQueue*, and the solver looks up the origin map to determine x_1's corresponding bit-vector x. It then enqueues all propagators whose arguments contain bit-vector x. In this case, propagators for $d \leftrightarrow x < y$ and $e \leftrightarrow x < z$ are enqueued. And so the process continues. It stops when all variables are assigned and a solution is generated, or a top-level conflict indicates the problem's unsatisfiability.

3.3 Forward and Backward Explanation Generation

An essential ingredient for conflict analysis in SAT solving is the generation of explanations—reasons why a given literal was implied. An explanation for a fixed literal is essential for conflict analysis in SAT solving. The explanations are used to build the implication graph, used to generate a learnt clause [51] when backtracking. Furthermore, the explanation is only demanded during the conflict analysis.

When a literal ℓ is generated by word-level propagation, there are two alternatives for explanation generation, as indicated in Figure 1. One possibility (labelled ② in the figure) is to generate the explanation clause c_i directly and send it to the *Reasons* array. This mirrors what is normally done in a SAT solver, and we call it "forward explanation": we generate the explanation eagerly, in a forward manner, as we do the propagation.

As the explanation is only needed for conflict analysis, we can, alternatively, generate it on demand [32] during the conflict analysis. This alternative is labelled ②′ in Figure 1. When ℓ is propagated, rather than send a clause to the reason array *Reasons*, we send the propagator p which fixed ℓ. It is entirely possible that ℓ is never involved in the conflict analysis, in which case we save the explanation generation cost. Otherwise, if an explanation is called for, *Reasons* will provide the propagator p, and we make p do the explanation generation for ℓ. The concrete clause (the explanation) is submitted to conflict analysis ②′. We call this method "backward explanation".

Compared to forward explanation, backward explanation can save explanation generation cost for literals not involved in conflict analysis. However, it does require revisiting the reason array when an explanation is requested. In Section 5.4, we evaluate the two methods of explanation generation.

4 Word-Level Propagators

4.1 Trit-Vectors and Word-Level Propagation

A "trit-vector" (for bit-width w) is a sequence of w elements taken from $\{0, 1, *\}$ which is used to denote the bit-vector during the reasoning. Here the $*$ represents an undetermined bit, so a trit-vector x corresponds to the cube

$(\bigwedge_{i\in I_0} \neg x_i) \wedge (\bigwedge_{i\in I_1} x_i)$, where I_0 is the set of index positions that hold a 0, and I_1 is the set of index positions that hold a 1.

In an implementation, the trit-vector can be represented by a pair $\langle \mathsf{lo}(x), \mathsf{hi}(x)\rangle$ of bit-vectors, with $\mathsf{lo}(x)$ and $\mathsf{hi}(x)$ representing the lower and upper bound of x respectively. More precisely,

$$\mathsf{lo}(x)_i = \begin{cases} 0 & \text{if } x_i = * \\ x_i & \text{otherwise} \end{cases} \qquad \mathsf{hi}(x)_i = \begin{cases} 1 & \text{if } x_i = * \\ x_i & \text{otherwise} \end{cases}$$

For example, trit-vector z= 011*0*11 is written $\langle 01100011, 01110111\rangle$ in this "lo-hi" form. The advantage of this form is that, as long as the bit-width of a trit-vector x is less than or equal to the size of machine registers, $\mathsf{lo}(x)$ and $\mathsf{hi}(x)$ can be treated as unsigned integers; in our example, z is $\langle 99, 119\rangle$. Supported by an implementation language (such as C) that can utilise word-level operations, we can rephrase bit-propagation on a trit-vector as word-level operations on its bounds. For a concrete example, consider y= *1110*** corresponding to $\langle 01110000, 11110111\rangle$, and the constraint $y = z$ (where z is as above). We can utilize the word-level rule: $\mathsf{lo}(y) = \mathsf{lo}(z) = \mathsf{lo}(y) \,|\, \mathsf{lo}(z)$, $\mathsf{hi}(y) = \mathsf{hi}(z) = \mathsf{hi}(y) \,\&\, \mathsf{hi}(z)$ to obtain the new lo-hi form of y (and z): $\langle 01110011, 01110111\rangle$ representing 01110*11. Instead of propagating the bits one by one, we effectively fix the bits y_7, y_1, y_0 and z_4 simultaneously with the word-level operations on the bounds.

The lo-hi form allows for invalid representations of trit-vectors. That happens when, for some x, a bit in $\mathsf{lo}(x)$ is 1 while the corresponding bit in $\mathsf{hi}(x)$ is 0. As will be seen, propagation can produce such invalid forms, but this happens when, and only when, an inconsistency is present in the current set of constraints. The simplified validity checking rule [50] is:

$$\mathsf{valid}(x) = \mathsf{\sim lo}(x) \,|\, \mathsf{hi}(x) \tag{1}$$

The result for a valid bit-vector lo-hi form should be a bit-vector consisting entirely of 1-bits with the same bit-width as the bit-vector variable; otherwise it is invalid, and the 0 bits in the result are the bits that cause the invalidity. For example, consider constraint $y = z$, and bit-vectors y= 1*00 and z= 00*1. First, we utilize the word-level rule to get the new lo-hi form of y (and z): $\langle 1001, 0000\rangle$. Then we use the validity checking rule on the new lo-hi form: $\mathsf{valid}(y) = \mathsf{valid}(z) = \mathsf{\sim lo}(y) \,|\, \mathsf{hi}(y) = 0110$.

The following predicates on trit-vectors will prove useful:

$$\mathsf{fixed}(x) \equiv \mathsf{lo}(x) = \mathsf{hi}(x) \qquad \mathsf{pos}(x) = \{\mathsf{lit}(x_i) \mid \mathsf{lo}(x_i) = 1\}$$
$$\mathsf{msb}(x_{[w]}) = x_{w-1} \qquad \mathsf{neg}(x) = \{\mathsf{lit}(x_i) \mid \mathsf{hi}(x_i) = 0\}$$
$$\mathsf{lit}(b) = \begin{cases} \ulcorner b \urcorner & \text{if } \mathsf{lo}(b) = 1 \\ \ulcorner \neg b \urcorner & \text{if } \mathsf{hi}(b) = 0 \end{cases} \qquad \mathsf{lits}(x) = \mathsf{pos}(x) \cup \mathsf{neg}(x)$$

We use $\mathsf{fixed}(x)$ to return a Boolean value indicating whether every bit in trit-vector x is fixed, that is, 0 or 1. We use $\mathsf{msb}(x_w)$ to denote the most significant bit of trit-vector x. We use $\mathsf{lit}(b)$ to return the *literal* corresponding to the Boolean bit b under the condition that b is fixed (hence the use of Quine corners). We

use $\mathsf{pos}(x)$ ($\mathsf{neg}(x)$) to return the set of literals fixed to 1 (respectively 0) in trit-vector x, and $\mathsf{lits}(x)$ to return the set of fixed literals in x. In algorithms, we take such a set of literals to mean the conjunction of the literals.

4.2 Word-Level Propagation with Bit-Level Explanation

Bit-blasting is the most common approach to bit-vector constraint solving. Bit-blasting rewrites all the word-level formulas into a large number of low-level propositional formulas although many of them may be redundant and never used in the solving process. Instead of doing bit-blasting, we use word-level propagation utilizing the lo-hi form. The propagators perform the propagation, and in our variant they also generate explanations in the form of clauses, for literals fixed by propagation. They can be seen as lazy clause generators, generating clauses only as these are needed.

The input of the word-level propagators are the domain of each bit-vector and sometimes Boolean variables (for example, the propagator for the $\mathbf{ite}(b, x_1, x_2) = y$ constraint). Inside the propagator, we utilize (and extend) the propagation rules introduced by Michel and Van Hentenryck [50] to do the propagation at the word level. In the following propagation rules, l and u denote the old lower and upper bound respectively; low and up denote the new lower and upper bound respectively. Note that a lower or upper bound is a bit-vector as shown in the following rules, and in an implementation language such as C, we need it to fit into a machine register. This is a limitation of our current word-level propagator—the bit width of its associated bit-vectors must be no longer than the size of a machine register (in our case 64 bits). This restriction can be lifted through translations that reflect standard implementation of arbitrary-precision arithmetic [63]; this, however, we have not done.

After each round of propagation for the bit-vector interval, the propagators apply validity checking (1) on the new intervals. After this, either some bits are propagated, or a conflict happened which means a conflict clause (or several conflict clauses) should be returned. Note that a conflict clause which stems from one bit set to b, is the same as the explanation why this bit would become fixed to $\neg b$. The explanations that we generate are presented in the following.

When there is no conflict, the propagators explain every propagated bit at the bit level. The explanation for the propagated bit is a set of literals which make up the reason for having fixed the propagated bit. Take the bitwise equality ($x = y$) as an example: when the ith bit of integer x_i is fixed to 1, we know that the reason is that y_i is already fixed to 1. So the clause $c : \neg y_i \vee x_i$ is the explanation why x_i is fixed to 1.

We next introduce the propagators for bit-vector constraints. Section 4.3 covers basic propagators, for logical constraints and structural constraints. Section 4.4 deals with propagation for arithmetic constraints, using a concept of *decomposition*. Section 4.5 briefly explores alternative ways of propagating reified constraints, using what we call composed propagators.

4.3 Basic Propagators

Both the propagation rules and the explanations are based on certain inference rules, as explained below. Since the explanation generation is straightforwardly done at the bit level, we concentrate on how to generate word-level propagation rules. The approach is general, so we simply illustrate it via an example. Full details can be found elsewhere [68].

Let us begin with the simplest constraint, bitwise equality, say, $y = x$. This constraint can only tighten the bounds for x and y, leaving them identical. Let the bounds for x be $[l^x, u^x]$ (before propagation), and let the bounds for y be $[l^y, u^y]$. The new, tighter, lower bound for x, which we denote low^x, is obtained by taking the disjunction of l^x and l^y,similarly the new, tighter upper bound is obtained through conjunction of the components' upper bounds. More precisely:

$$\begin{aligned} low^x &= l^x \mid l^y \\ up^x &= u^x \;\&\; u^y \\ low^y &= low^x \\ up^y &= up^x \end{aligned}$$

4.3.1 Logical Constraints

The propagation rules for the bitwise operators are generated by similar reasoning, and in an entirely systematic way. Take $z = x \,\&\, y$. Again, lower bounds for the involved variables can be tightened through disjunction, just as the upper bounds are tightened through conjunction. For a single bit, z_i of z, we have the inference rules that $z_i = 0$ iff $x_i = 0$ or $y_i = 0$, and $z_i = 1$ iff $x_i = 1$ and $y_i = 1$. The first rule entails that we may be able to tighten z's upper bound, in case either x or y (or both) have a 0 in some position i. The second entails that we may be able to tighten z's lower bound, in case x and y share a 1 in some position. Similarly, we can reason about the bounds for x and y. Altogether, expressed at the word level, we have:

Propagation rules for the $z = x \,\&\, y$ constraint:

$$\begin{aligned} low^x &= l^x \mid l^z \\ up^x &= u^x \,\&\, {\sim}({\sim}u^z \,\&\, l^y) \\ low^y &= l^y \mid l^z \\ up^y &= u^y \,\&\, {\sim}({\sim}u^z \,\&\, l^x) \\ low^z &= l^z \mid (l^x \;\&\; l^y) \\ up^z &= u^z \,\&\, u^x \,\&\, u^y \end{aligned}$$

Example 1 Consider $x = {**}00*$, $y = 1111*$, $z = 01{**}1$, and constraint $z = x \,\&\, y$. The lower and upper bounds for each bit-vector variable before the propagation are as shown below (left column). After propagation, we have the improved lower and upper bounds (middle column) and the result from the validity checking rules (right column).

$$\left.\begin{array}{ll} l^x = 00000 & low^x = 01001 \\ u^x = 11001 & up^x = 01001 \end{array}\right\} \ \mathsf{valid}(x) = 11111$$
$$\left.\begin{array}{ll} l^y = 11110 & low^y = 11111 \\ u^y = 11111 & up^y = 11111 \end{array}\right\} \ \mathsf{valid}(y) = 11111$$
$$\left.\begin{array}{ll} l^z = 01001 & low^z = 01001 \\ u^z = 01111 & up^z = 01001 \end{array}\right\} \ \mathsf{valid}(z) = 11111$$

Note how the constraint was found to be satisfiable, and, in this example, all bits ended up being determined. □

Explanations for the $z = x \,\&\, y$ constraint:

The constraint $z = x \,\&\, y$ gives rise to these explanations, per bit position i:

$$c_1 : x_i \vee \neg z_i \qquad c_2 : y_i \vee \neg z_i \qquad c_3 : \neg x_2 \vee \neg y_i \vee z_i$$

Propagators for other logical constraints follow the same pattern.

Propagation rules for the $z = x \oplus y$ constraint:

$$\begin{array}{l} low^x = l^x \mid (\sim u^z \;\&\; l^y) \mid (l^z \;\&\; \sim u^y) \\ up^x = u^x \;\&\; (u^z \mid u^y) \;\&\; (\sim (l^y \;\&\; l^z)) \\ low^y = l^y \mid (\sim u^z \;\&\; l^x) \mid (l^z \;\&\; \sim u^x) \\ up^y = u^y \;\&\; (u^z \mid u^x) \;\&\; (\sim (l^x \;\&\; l^z)) \\ low^z = l^z \mid (\sim u^x \;\&\; l^y) \mid (l^x \;\&\; \sim u^y) \\ up^z = u^z \;\&\; (u^x \mid u^y) \;\&\; (\sim (l^x \;\&\; l^y)) \end{array}$$

Explanations for the $z = x \oplus y$ constraint:

$$c_1 : x_i \vee y_i \vee \neg z_i \qquad c_2 : x_i \vee \neg y_i \vee z_i \qquad c_3 : \neg x_i \vee y_i \vee z_i \qquad c_4 : \neg x_i \vee \neg y_i \vee \neg z_i$$

The next example shows a case where a conflict happens in propagation.

Example 2 Consider x = 1*000, y = 0**1* and z = 01*00 and constraint $z = x \oplus y$. The lower and upper bounds for each bit-vector variable before the propagation are as shown below (left column). After the propagator is invoked to run the propagation rules, we get the new lower and upper bounds (middle column) and the result from the validity checking rules (right column).

$$\left.\begin{array}{ll} l^x = 10000 & low^x = 10010 \\ u^x = 11000 & up^x = 01000 \end{array}\right\} \ \mathsf{valid}(x) = \mathbf{0}11\mathbf{0}1$$
$$\left.\begin{array}{ll} l^y = 00010 & low^y = 10010 \\ u^y = 01111 & up^y = 01100 \end{array}\right\} \ \mathsf{valid}(y) = \mathbf{0}11\mathbf{0}1$$
$$\left.\begin{array}{ll} l^z = 01000 & low^z = 11010 \\ u^z = 01100 & up^z = 01100 \end{array}\right\} \ \mathsf{valid}(z) = \mathbf{0}11\mathbf{0}1$$

There are two conflicts, and the conflict clauses are $\neg x_4 \vee y_4 \vee z_4$ and $x_1 \vee \neg y_1 \vee z_1$. □

Propagation rules and explanations for all the remaining logical constraints can be found elsewhere [68].

4.3.2 Structural Constraints

We can take the structural constraints as variants of the bitwise equality constraints for bit manipulation. Therefore, the propagation rules for structural constraints are based on the propagation rules of the bitwise equality constraints but with different "masks" fixing the particular bits to be 1 or 0. The explanations for the structural constraints are also similar to the bitwise equality constraints but with some bit shift ($\ll$,$\gg_u$, $\gg_s$, *rotl*, *rotr*), or fixing some bit values ($\ll$, $\gg_u$, $\gg_s$, ext_u, ext_s).

We show the propagation rule and the explanation for the signed extension constraint $y = \textbf{sign_extend}(n, x)$ as an example. Example 3 below shows an instance where several bits can be fixed at the same time.

Propagation rules for the $y = \textbf{sign_extend}(n, x)$ constraint:

1. if $x_{w_x-1} = 0$:

$$\begin{aligned} mask_1 &= 0_{[n]}.1_{[w_x]} \\ mask_2 &= 0_{[w_x+n]} \\ low^x &= l^x \mid (l^y \;\&\; mask_1) \\ up^x &= u^x \;\&\; (u^y \;\&\; mask_1) \\ low^y &= (l^x \mid mask_2) \mid l^y \\ up^y &= (u^x \mid mask_2) \;\&\; u^y \end{aligned}$$

   The role of $mask_1$ is to extract the rightmost w_x bits from y. The role of $mask_2$ is to extend x to $w_x + n$ bit width.

2. if $x_{w_x-1} = 1$

$$\begin{aligned} mask_1 &= 0_{[n]}.1_{[w_x]} \\ mask_2 &= 1_{[n]}.0_{[w_x]} \\ low^x &= l^x \mid (l^y \;\&\; mask_1) \\ up^x &= u^x \;\&\; (u^y \;\&\; mask_1) \\ low^y &= low^x \mid mask_2 \\ up^y &= up^x \mid mask_2 \end{aligned}$$

   $mask_1$ is extracts the rightmost w_x bits of y. The role of $mask_2$ is to force the leftmost n bits of y to one.

3. If $x_{w_x-1} = *$:
   - if $y_i = 0$ (for some $i \geq w_x - 1$): fix $x_{w_x-1} = 0$, then apply rule 1.
   - else if $y_i = 1$ (for some $i \geq w_x - 1$): fix $x_{w_x-1} = 1$, then apply rule 2.
   - else

$$\begin{aligned} mask_1 &= 0_{[n]}.1_{[w_x]} \\ mask_2 &= 1_{[n]}.0_{[w_x]} \\ mask_3 &= 0_{[w_y]} \\ low^x &= l^x \mid (l^y \;\&\; mask_1) \\ up^x &= u^x \;\&\; (u^y \;\&\; mask_1) \\ low^y &= l^y \mid (l^x \mid mask_3) \\ up^y &= u^y \;\&\; (u^x \mid mask_2) \end{aligned}$$

The role of $mask_1$ is to extract the rightmost w_x bits from y. $mask_2$ and $mask_3$ are used to extend x to the same bit-width as y, without affecting the new lower and upper bound of y.

Explanations for the y = sign_extend(n,x) constraint:

- For y_i:

$$\begin{array}{llll} c_1 : x_{w_x-1} \vee \neg y_i & \text{and} & c_2 : \neg x_{w_x-1} \vee y_i & \text{if } i \geq w_x - 1 \\ c_1 : x_i \vee \neg y_i & \text{and} & c_2 : \neg x_i \vee y_i & \text{otherwise} \end{array}$$

- For x_i:

$$\begin{array}{llll} c_1 : x_i \vee \neg y_j & \text{and} & c_2 : \neg x_i \vee y_j \ \ (\text{for all} j \geq w_x - 1) & \text{if } i = w_x - 1 \\ c_1 : x_i \vee \neg y_i & \text{and} & c_2 : \neg x_i \vee y_i & \text{otherwise} \end{array}$$

Example 3 Consider $n = 3$, x = *0*0, and y = 1**1*1*. Since x_3 = *, we apply rule 3 of the propagation rule. Since there exists $y_3 = 1$, rule 2 is applied and the following new lower and upper bounds are obtained:

$$\left.\begin{array}{l} low^x = 1010 \\ up^x = 1010 \end{array}\right\} \mathsf{valid}(x) = 1111 \qquad \left.\begin{array}{l} low^y = 1111010 \\ up^y = 1111010 \end{array}\right\} \mathsf{valid}(y) = 1111111$$
$$new^x = \mathbf{1}0\mathbf{1}0 \qquad new^y = 1\mathbf{111}0\mathbf{1}0$$

The explanation for newly fixed $x_1 = 1$ is the clause: $x_1 \vee \neg y_1$; for newly fixed $x_3 = 1$ is the clause: $x_3 \vee \neg y_3$; for the newly fixed $y_0 = 0$ is the clause: $\neg y_0 \vee x_0$; for the newly fixed $y_2 = 0$ is the clause: $\neg y_2 \vee x_2$; for the newly fixed $y_4 = 1$ and $y_5 = 1$ is the clause: $y_i \vee \neg x_3$ (i = 4 and 5 respectively). □

Example 4 Consider $n = 2$, x = 00, and y = 100*. Using rule 1 we calculate low^y = 1000 and up^y = 0000. Hence $\mathsf{valid}(y)$ = 0111 leading to the conclusion that y = **sign_extend**$(2,x)$ is not satisfiable. □

Note that, for the shift constraints the number of the shift bits is not necessarily a constant; it can be a bit-vector variable. Take the general left shift operation ($x = y \ll z$) as an example. Here z is not a constant, but a bit-vector variable. Algorithm 3 shows how STP deals with this kind of constraint. Our approach mirrors that, but rather than resorting to bit-by-bit processing, we generate and process corresponding word-level constraints. Specifically, we generate a nested **ite** expression corresponding to the for loop of Algorithm 3.

Example 5 For bit width 4, the only interesting shift values are < 4, so only the z_0 and z_1 bits are of interest—if a more significant bit is set, the result of the shift will be 0. (In Algorithm 3, the variable *care_bits* records the number of interesting bits from z.) Hence we generate

$$x = \textbf{let } x' = \textbf{ite}(z_0, y \ll 1, y) \textbf{ in ite}(z_3 \vee z_2, 0, \textbf{ite}(z_1, x' \ll 2, x'))$$

In a context where we have derived, say, y = 0011 and z = 00*1, this effectively reduces to x = **ite**$(z_1, 1000, 0110)$. □

A way of reducing a propagator into several basic propagators ("decomposition") is now introduced.

Algorithm 3 Propagator for $x = y \ll z$

$care_bits \leftarrow \lceil \log_2 bitwid \rceil$
$previous_x \leftarrow y$
for $i \leftarrow 0$ **to** $care_bits - 1$ **do**
  $shift_amount \leftarrow 1 \ll i$
  $current_x \leftarrow \textbf{ite}(z_i, previous_x \ll shift_amount, previous_x)$
  $previous_x \leftarrow current_x$
$z_{rest_bits} \leftarrow z_{bitwid-1} \vee z_{bitwid-2} \vee \cdots \vee z_{care_bits}$
$x \leftarrow \textbf{ite}(z_{rest_bits}, 0, current_x)$

4.4 Decomposed Propagators

We now consider the important concept of *decomposition*, that is, how to break a target constraint into several basic constraints while introducing intermediate variables. We can then use the corresponding propagators for the basic constraints to solve the target constraint. The propagation rules and the explanation for the target constraint simply combine those of the basic constraints. We use decomposition for all the arithmetic constraints.

Consider the if-then-else constraint **ite**. We use it to illustrate how the decomposed propagators cooperate to solve a target constraint. We also give the specific decomposition rules we use for non-linear arithmetic constraints. The decomposition rules for the linear arithmetic constraints including addition, unary minus and subtraction constraints can be found elsewhere [69].

4.4.1 The If-Then-Else Constraint $z = \textbf{ite}(b, x, y)$

The constraint $z = \textbf{ite}^{\textbf{BV}}(bv, x, y)$ discussed by Michel and Van Hentenryck [50] is different from the one $z = \textbf{ite}(b, x, y)$ in SMT-LIB2, because of the first parameter: one is a bit-vector (bv) while the other is a Boolean variable (b). However, we can transform the constraint $z = \textbf{ite}(b, x, y)$ into the constraint $z = \textbf{ite}^{\textbf{BV}}(bv, x, y)$, by introducing a signed extension constraint and a new bit-vector variable bv. Note that within the propagator for signed extension, we treat Boolean variable b as a bit-vector with bit-width 1, while outside the propagator we treat b as a Boolean variable and put the propagator into a Boolean watch list as explained in Section 3. Assuming the bit-width is n:

$$\begin{aligned} bv &= \textbf{sign_extend}(n-1, b) \\ z &= \textbf{ite}(bv, x, y) \end{aligned}$$

Therefore, the propagator for constraint $z = \textbf{ite}(b, x, y)$ can be divided into two *decomposed* propagators: one is for the $\textbf{sign_extend}(n-1, b)$ constraint and the other is for the $\textbf{ite}(bv, x, y)$ constraint. The propagation rules and the explanations for the $z = \textbf{ite}(b, x, y)$ propagator are combinations of those for the two component propagators.

The propagation rules for $z = \mathbf{ite^{BV}}(bv, x, y)$ are as follows[2]:

$$\begin{aligned}
low^{bv} &= l^{bv} \mid (l^z \ \&\ (\sim u^y)) \mid ((\sim u^z) \ \&\ l^y) \\
up^{bv} &= u^{bv} \ \&\ (\sim l^z \mid u^x) \ \&\ (u^z \mid (\sim l^x)) \\
low^x &= l^x \mid (l^z \ \&\ (l^{bv} \mid (\sim u^y))) \\
up^x &= u^x \ \&\ (\sim((\sim u^z) \ \&\ (l^{bv} \mid l^y))) \\
low^y &= l^y \mid (l^z \ \&\ ((\sim u^{bv}) \mid (\sim u^x))) \\
up^y &= u^y \ \&\ (u^z \mid (u^{bv} \ \&\ (\sim l^x))) \\
low^z &= l^z \mid (l^{bv} \ \&\ l^x) \mid ((\sim u^{bv}) \ \&\ l^y) \mid (l^x \ \&\ l^y) \\
up^z &= u^z \ \&\ (\sim l^{bv} \mid u^x) \ \&\ (u^{bv} \mid u^y) \ \&\ (u^x \mid u^y)
\end{aligned}$$

The explanation for the $z = \mathbf{ite^{BV}}(bv, x, y)$ constraint:

$$\begin{aligned}
&c_1 : \neg bv_i \vee \neg x_i \vee z_i \quad c_2 : \neg bv_i \vee x_i \vee \neg z_i \quad c_3 : bv_i \vee \neg y_i \vee z_i \\
&c_4 : bv_i \vee y_i \vee \neg z_i \quad c_5 : \neg x_i \vee \neg y_i \vee z_i \quad c_6 : x_i \vee y_i \vee \neg z_i
\end{aligned}$$

The propagation rules and explanation for the signed extension propagator were given in Section 4.3.2.

Example 6 Consider x = 1*0**, y = *1*1*, z = 0***0, b = *, and constraint $z = \mathbf{ite}(b, x, y)$. The intermediate variable bv = *****, and the lower and upper bounds for each bit-vector variable before the propagation are:

$$\begin{aligned}
&l^{bv} = 00000 \quad u^{bv} = 11111 \qquad l^x = 10000 \ u^x = 11011 \\
&l^y = 01010 \quad u^y = 11111 \qquad l^z = 00000 \ u^z = 01110
\end{aligned}$$

Since some bits are fixed in x, y and z, we assume the propagator for $z = \mathbf{ite^{BV}}(bv, x, y)$ is enqueued in the propagation queue waiting to do its work. After the propagator is invoked, to run the propagation rules and the validity checking (which in this case indicates that there is no conflict), bv_4 and y_4 are newly fixed to 0:

$$\begin{aligned}
&\left.\begin{array}{r} low^{bv} = 00000 \\ up^{bv} = 01111 \end{array}\right\} \mathsf{valid}(bv) = 11111 \quad new^{bv} = \mathbf{0}**** \qquad
\left.\begin{array}{r} low^{y} = 01010 \\ up^{y} = 01111 \end{array}\right\} \mathsf{valid}(y) = 11111 \quad new^{y} = \mathbf{0}1*1* \\
&\left.\begin{array}{r} low^{x} = 10000 \\ up^{x} = 11011 \end{array}\right\} \mathsf{valid}(x) = 11111 \quad new^{x} = 1*0** \qquad
\left.\begin{array}{r} low^{z} = 00000 \\ up^{z} = 01110 \end{array}\right\} \mathsf{valid}(z) = 11111 \quad new^{z} = 0***0
\end{aligned}$$

The explanation for $bv_4 = 0$ is the clause $\neg bv_4 \vee \neg x_4 \vee z_4$, and for $y_4 = 0$ is the clause $\neg x_4 \vee \neg y_4 \vee z_4$. Because of the newly fixed bits, both the propagators for $bv = \mathbf{sign_extend}(4, b)$ and $z = \mathbf{ite^{BV}}(bv, x, y)$ are enqueued. When the signed extension propagator is invoked, the Boolean variable b is fixed to 0, and the remaining bits in bv are all fixed such that bv = 0**0000** (details are in Section 4.3.2). After that, the propagator for $z = \mathbf{ite^{BV}}(bv, x, y)$ is invoked again,

[2] The propagation rules for $z = \mathbf{ite^{BV}}(bv, x, y)$ differ substantially from those in [50] which are: $low^y = l^y \mid (l^z \ \&\ (l^{bv} \mid (\sim u^x)))$ and $up^y = u^y \ \&\ \sim(\sim u^z \ \&\ (l^{bv} \mid l^x))$ (using our notation). These rules are not quite correct. For example, for bit-width 1, let $z = 1$, $bv = 0$, and let x and y be unknown, then y should be fixed to 1. But under the rule above, $low^y = 0$, which is an error.

and we get the newly propagated bit-vectors: $bv = 00000$, $x = 1{*}0{**}$, $y = 01{*}1\mathbf{0}$, and $z = 0\mathbf{1}{*}\mathbf{1}0$. Therefore, in this propagation round, y_0 is fixed to 0 , and z_1 and z_3 are fixed to 1, with the explanation $bv_i \vee \neg y_i \vee z_i$ (i is 0, 1 and 3 respectively). □

4.4.2 Non-Linear Arithmetic Constraints

We deal with non-linear arithmetic constraints by first developing a propagator for multiplication, and then transforming all remainder and division constraints into multiplication and other basic constraints using decomposition. These transformations use the rules given by Limaye and Seshia [46]. Therefore, the task of solving non-linear arithmetic constraints is reduced to solving the multiplication constraint.

Assuming the bit-width is n, the multiplication constraint $z = x * y$ can be transformed in the following way:

$$\begin{aligned} z &= (x \ll (n-1)) * y_{n-1} + (x \ll (n-2)) * y_{n-2} + \ldots + (x \ll 1) * y_1 + x * y_0 \\ &= \mathbf{ite}(y_{n-1}, x \ll (n-1), 0) + \mathbf{ite}(y_{n-2}, x \ll (n-2), 0) + \ldots \\ &\qquad + \mathbf{ite}(y_1, x \ll 1, 0) + \mathbf{ite}(y_0, x, 0) \end{aligned}$$

Hence the multiplication propagator is decomposed into n **ite** propagators and $n-1$ addition propagators.

Example 7 Consider $x = 0{**}1$, $y = {*}10{*}$, $z = {*}1{*}1$, and constraint $z = x * y$. We get the following addition format according to the transformation rules above. We use the bit variable names to present the unknown bits (*).

$$\begin{array}{rcl} \begin{pmatrix} y_0 = 1 : 0 & x_2 & x_1 & 1 \\ y_0 = 0 : 0 & 0 & 0 & 0 \end{pmatrix} \\ 0 \quad 0 \quad 0 \\ x_1 \quad 1 \\ + \quad y_3 \\ \hline z_3 \quad 1 \quad z_1 \quad 1 \end{array}$$

Since $z_0 = 1$ and $x_0 = 1$, the addition propagator concludes $y_0 = 1$. Since there is no carry in from the second bit position, $x_2 = 0$ is also determined by the addition propagator. We also know that $x_1 = z_1$, and $y_3 = z_3$, so further information can be propagated once we fix some of these four bits. □

4.4.3 Reified Constraints

We base the decomposed propagator for reified equality constraint $b \leftrightarrow x = y$ on this observation [70]: $b = \mathsf{msb}(\sim((x-y)\,|\,(y-x)))$. We add intermediate variables to split this constraint into several basic constraints which can be processed by the word-level propagators already introduced. The explanation

for the reified equality constraint $b \leftrightarrow x = y$ is made up by those of the basic constraints—several small explanations with the intermediate literals involved.

$$m_1 = x - y;\ \ m_2 = -m_1;\ \ m_3 = m_1 \,|\, m_2;\ \ m_4 = \sim m_3;\ \ b = \mathsf{msb}(m_4)$$

The decomposed rule for the reified disequality constraint $b \leftrightarrow x \neq y$ is as follows: $b = \mathsf{msb}((x - y) \,|\, (y - x))$. The way of decomposing the propagator is similar to the reified equality constraint.

The decomposed propagator for an inequality constraint $b \leftrightarrow x \leq y$ is based on this observation [70]:

$$b = \mathsf{msb}((x \,|\, \sim y)\ \&\ ((x \oplus y) \,|\, \sim(y - x)))$$

The way to tackle an inequality constraint with decomposed propagators is the same as for the reified equality and disequality constraints. Other inequality constraints can be transformed into the unsigned less than or equals constraint $b \leftrightarrow x \leq_u y$, based on the semantics introduced in Section 2.1.

4.5 Composed Propagators

Besides using decomposed propagators, we also try an alternative way to solve the reified constraints which we call composed propagators. A composed propagator is a single compact function which is used to propagate the constraint in one go.

Take the reified equality constraint $b \leftrightarrow x = y$ as an example. The main algorithm for the propagator is shown as Algorithm 4. The propagator reuses the implementation of the propagators for $x = y$ and $x \neq y$ (here we call them "sub-propagators"), or checks that $x = y$ in the current domain in which case it explains b, or that $x \neq y$ in the current domain, in which case it explains $\neg b$. Detailed algorithms for all composed propagators can be found elsewhere [69].

5 Design Options

In this section we discuss three dimensions in the design space for a word-level propagation solver: two ways to create the propagators for reified constraints (composed vs decomposed), two ways to administer explanation (forward vs backward), and two approaches to conflict analysis (first vs highest decision level). In Section 5.4 we summarise the combinations we found to work best.

5.1 Propagators: Composed vs Decomposed

Sections 4.4 and 4.5 discussed composed and decomposed propagators to implement reified constraints. We now investigate the difference between the

Algorithm 4 Propagator for $b \leftrightarrow x = y$

```
function Prop_ReifEq(bit b, bit-vec x, y)
    if lo(b) = 1 then
        return Prop_Eq(x, y)
    else if hi(b) = 0 then
        return Prop_DisEq(x, y)
    else
        if fixed(x) ∧ fixed(y) ∧ lo(x) = lo(y) then                    ▷ x = y
            Explanation := lits(x) ∧ lits(y) → b
            Enqueue(b, Explanation)
        else
            z := lo(x) & ~hi(y) | ~hi(x) & lo(y)
            if z ≠ 0 then                                              ▷ x ≠ y
                choose i with z_i = 1
                Explanation := lit(x_i) ∧ lit(y_i) → ¬b
                Enqueue(¬b, Explanation)
        return true
```

two approaches in search and learning, and we compare the propagation strength of the two types of propagator.

Differences in Search and Learning. The composed propagators are often complex to implement. Moreover, the explanations generated by the propagators are often large, especially when the bit-width of the involved bit-vectors is large. For example, for the reified equality constraint $b \leftrightarrow x = y$, when $b = 0$, $x = 11010$, $y = 110*0$, we propagate bit $y_1 = 0$, and the explanation is $x_4 \wedge x_3 \wedge \neg x_2 \wedge x_1 \wedge \neg x_0 \wedge y_4 \wedge y_3 \wedge \neg y_2 \wedge \neg y_0 \rightarrow \neg y_1$. In some cases it can be worth splitting the complicated constraint into several smaller constraints thus decomposing it. Not only are the decomposed components easier to implement, but more importantly, the intermediate variables introduced can help make explanations shorter. From Section 4.3 we know that each explanation for a basic constraint contains at most three literals. On the other hand, the composed propagators are compact, while the decomposed propagators require communication among the decomposed components.

It is worth pointing out that the two kinds of propagator do not lead to identical search trees. The presence of intermediate variables introduced by the decomposition makes a considerable difference to activity based search, since there are new variables to search on and different initial activities. Therefore, on the one hand, the intermediate variables in the decomposed propagators allow it to "search in the middle"; on the other hand, they also enlarge the search space which may lead to greater effort to find a solution.

Differences in Propagation Strength

– **Composed Propagators**
  The composed propagators achieve bit-consistency, as defined in Section 2.2. Algorithm 4 and the similar algorithm for inequality constraints achieve bit-consistency. Hence if we want to show the composed propagators achieve bit-consistency, we only need to prove the sub-propagators are bit-consistent.

- **reified equality propagator** $b \leftrightarrow x = y$
  The propagation for the equality constraint $x = y$, obviously is bit-consistent since it operates bitwise. The propagation for the disequality constraint $x \neq y$ is also bit-consistent, since the propagation is on the bit-level and no more bits can be propagated according to the algorithm.
- **reified inequality propagator:** $b \leftrightarrow x \leq_u y$
  As proved by Michel and Van Hentenryck [50], the propagation for $x \leq_u y$ achieves bit-consistency. The propagation for the $x >_u y$ is similar to the one for $x \leq_u y$, hence it is also bit-consistent.

– **Decomposed Propagators**
  The corresponding decomposed propagators do not achieve bit-consistency. We show this next, through counter-examples.

Example 8 Consider first the propagator for reified equality, $b \leftrightarrow x = y$. Take x = **0*, y = **1*, and b = *. Clearly it can be deduced that $b = 0$. However, the propagation rules from Section 4.4.3 do not fix b, since the subtraction propagators for $x - y$ and $y - x$ are unable to fix any bits.

Now consider the propagator for reified inequality, $b \leftrightarrow x \leq_u y$. Take x = *1**, y = *0**, and $b = 1$. Clearly we can deduce $x_3 = 0$ and $y_3 = 1$. However, the propagation rules from Section 4.4.3 do not fix b, again because of the subtraction propagators' inability to fix bits. □

5.2 Explanation: Forward vs Backward

Normally in a SAT solver, for every fixed Boolean literal, a reason why it became *true* is required for conflict analysis. So normally, when we fix a Boolean literal in our word-level propagator, we return an explanation for it eagerly, so-called "forward explanation."

Another approach [32, 56] is to generate the explanation only during conflict analysis where the reason for a propagated literal is required. Compared to the forward explanation method, this has the advantage that explanations are only generated as needed. Furthermore, the "backward explanation" is good for our word-level propagator, since our propagators only have two parts: one is the propagation part, the other is the explanation generation part which is more time consuming. Therefore, backward explanation makes propagation faster, but possibly makes conflict analysis slower, which is the trade-off between these two approaches.

5.3 Conflict Analysis: First vs Highest Level

We observed in Section 4.1 that we can detect conflicts in many bit positions simultaneously. But how best to choose one for conflict analysis remains an open question. With SAT solving, as soon as the first conflict is found, conflict analysis is started, returning a learnt clause of the form $C \vee \ell$, where ℓ is the unique literal (UIP) at the current decision level, and the maximum

decision level in the remainder of the clause C determines the level to backjump to. One way to manage conflict analysis for word-level propagation is to choose the first conflict for conflict analysis, as for SAT. We call this "standard backjumping".

An alternative approach is to generate a conflict clause for each bit position that is in conflict. We can then add all the learnt clauses generated to the clause database and then jump to the highest decision level indicated by one of them. This has the advantage of generating more information from the failure, and potentially higher backjumps. We call this "multi-conflict backjumping".

Again there is a trade-off between these two approaches. On the one hand, standard backjumping promises faster conflict analysis; as multi-conflict backjumping needs to analyse several conflicts each time. On the other hand, multi-conflict backjumping can lead to backtracking to a higher level of the search tree than standard backjumping, with the potential to save on search.

5.4 Identifying the Best Options

In previous work [69] we experimented with the design options discussed above. The aim was to limit the experimental space, in case one design choice seemed clearly better than another, in practice.

First, we compared forward vs backward explanation, as well as standard vs multi-conflict backjumping. The conclusion [69] was that (1) backward explanation outperforms forward explanation significantly, especially when the test cases become time consuming, and that (2) multi-conflict backjumping outperforms standard backjumping (see Table 1 of [69]). Hence, we soon committed to backward explanation and multi-conflict backjumping.

Next we compared bit-blasting with word-level bit-vector solving using composed and decomposed propagators, experimenting with all possible combinations of composed (C) and decomposed (D) propagators for equality (eq) and for inequality constraints (le). In our experiments [69], the Deq+Dle word-level propagator was typically faster than bit-blasting on the easy cases, using less memory. For many difficult cases, the Ceq+Cle word-level propagator outperformed bit-blasting, and again used less memory (see Table 2 of [69]). In general, however, bit-blasting was found to have the most uniform and predictable behaviour.

We also counted the average number of conflicts found per time unit, to see to what extent the promise of parallel propagation and conflict checking is realised in practice (see Table 3 of [69]). For easy benchmarks, word-level propagation typically led to 50% more conflicts found per time unit. For hard benchmarks, the rate of conflict finding was more spectacular: ranging from 2.5 to 18 times the rate of the bit-blaster. Table 3 of [69] also gives the number of inspections (by which we mean a call to a unit or word-level propagator, whether it results in fixing new bits or not). For hard benchmarks, the table suggests that the rate of propagation per inspection is close to an order of magnitude larger for word-level propagation.

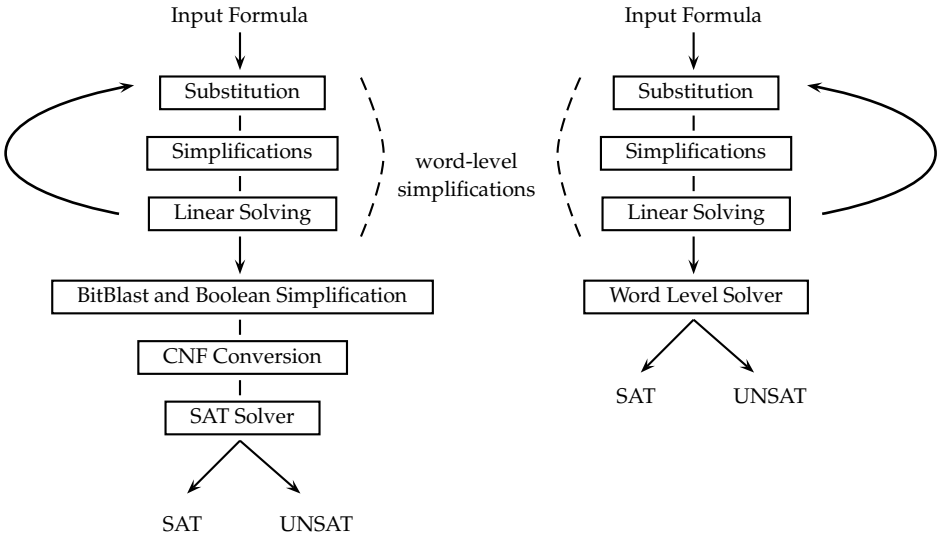


Fig. 2 The main structure of STP (left) and word-level solvers (right)

6 Experimental Evaluation

Having since extended the solvers to the entire QF_BV fragment of SMT-LIB2, we are now able to evaluate the solvers on larger sets of benchmarks, and compare against an established bit-vector logic solver. In this section we report on the evaluation. We have incorporated two word-level solvers into STP [31] and we compare the resulting bit-vector solvers against STP itself. The two word-level solvers used, Comp-W and Decomp-W, are the ones that use the Ceq+Cle and Deq+Dle propagation principles, respectively, together with backward explanation and multi-conflict backjumping.

STP has been developed over many years and been heavily used in many applications. For example, it has been central to the symbolic execution tool KLEE. It uses a number of "word-level simplifications" and these can be highly beneficial for our word-level solvers as well. STP applies an "on-the-fly" linear solver which enables many other simplification rules, and can solve purely linear problems outright. The main structure of STP for solving bit-vector constraints is shown in Figure 2 (left). Word-level simplification consists of three phases: substitution, simplification and linear solving. We have constructed variants that replace bit-blasting by word-level solving. The structure of the word-level solvers is shown in Figure 2 (right). Note that we do not do bit-blasting or any Boolean simplification in the word-level solver.

We compare the two word-level solvers (Comp-W and Decomp-W) with the original STP. The experimental data are the test cases from all the folders in the QF_BV category of SMT 2015 competition benchmarks,[3] except for the

[3] `https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=2641`

Table 1 STP vs two word-level solvers on the whole data set (times are in seconds)

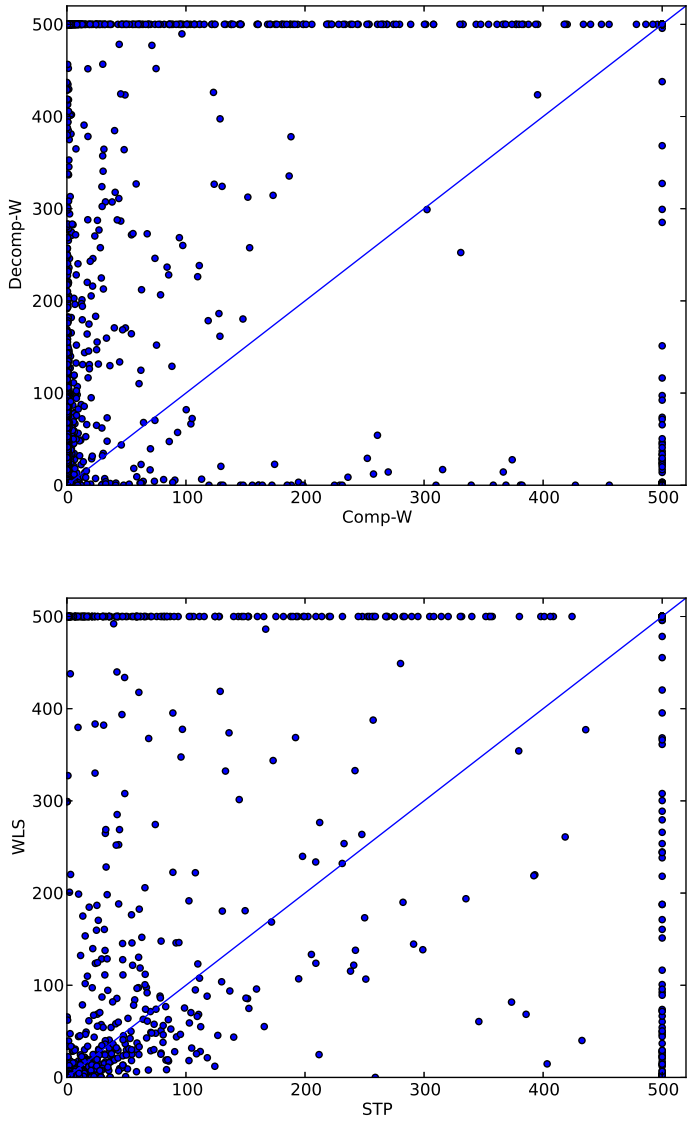
Problem		STP		Comp-W		Decomp-W	
name	number	time	TO	time	TO	time	TO
asp	492	**18372**	**269**	18819	283	16187	366
bench-ab	285	**0**	**0**	**0**	**0**	1	0
bmc-bv	131	**79**	**0**	140	0	935	0
bmc-bv-svcomp14	66	**146**	**2**	413	12	712	2
brummayerbiere	28	**407**	**4**	80	5	92	5
brummayerbiere2	31	**9**	**4**	111	5	74	5
brummayerbiere3	58	**809**	**21**	546	38	1091	38
brummayerbiere4	10	0	0	0	0	0	0
bruttomesso	314	**5760**	**100**	4921	141	7349	163
calypto	18	11	11	1	12	**30**	**10**
challenge	2	0	2	0	2	0	2
check2	6	0	0	0	0	0	0
crafted	21	0	0	0	0	0	0
dwp-formulas	332	**0**	**0**	**0**	**0**	2	0
ecc	8	**1**	**0**	0	4	**1**	**0**
fft	23	89	17	**259**	**16**	196	17
float	186	**8528**	**56**	6544	98	8995	118
galois	3	**0**	**2**	**0**	**2**	1	2
gulwani-pldi08	6	60	0	**47**	**0**	331	0
mcm	186	**4374**	**143**	5317	142	2486	164
pspace	21	4	0	4	0	4	0
rubik	7	**229**	**1**	380	1	80	2
sage	26607	**5485**	**1**	11642	86	16415	144
spear	1695	**4354**	**1**	787	11	47815	175
stp	1	10	0	**8**	**0**	0	1
stp-samples	426	**7**	**2**	**7**	**2**	8	2
tacas07	5	**342**	**1**	5	2	283	2
uclid	416	103	0	**97**	**0**	1424	7
uclid-contrib-smtcomp09	7	**479**	**5**	0	7	0	7
uum	8	109	6	**26**	**6**	287	6
VS3	11	433	10	**340**	**9**	385	10
wienand-cav2008	12	346	1	**61**	**1**	110	1
Total	31422	**50546**	**659**	50555	885	105296	1249
Overall time		**380046**		493055		729796	

ones in which the bit-width for the bit-vector operations is > 64 (the size of our machine registers). Totally, there are more than 30,000 test cases. The run time limit for each test case is 500 seconds. The experimental environment is a commodity computer with a Core-i7 CPU (2.7 GHz) and 5 GB RAM.

Table 1 shows the results. In the table, "time" means the total time in seconds for all the successful test cases in the folder; "TO" is the number of cases that timed out; "Total" is the total time of all successful test cases; "Overall time" is "Total" plus 500 seconds penalty for each unsuccessful case, which gives an overall "score". Table 1 highlights the best performances (according to the overall time) using boldface font.

The topmost scatter plot in Figure 3 shows Comp-W against Decomp-W performance across individual benchmarks, with each time-out plotted as 500

Fig. 3 Comp-W vs Decomp-W (top) and STP vs WLS (bottom)



seconds. The bottom plot likewise visualises STP performance against word-level solver performance. In this plot, as in Tables 2 and 3 below, we take the best case between Comp-W and Decomp-W, that is, 'WLS' indicates the best of the two word level solvers' performance in the given case. (Section 7 discusses how to automate the choice of approach.)

Table 1 (and the plots) show that STP outperforms the word-level solvers, overall. This is perhaps unsurprising, as STP has been developed over many years, and is well-engineered. STP also applies Boolean simplification after bit-blasting, which is not available to the word-level solvers. And, some benchmarks contain a large number of "pseudo bit-vector operations" with bit-width 1, which are actually Boolean operations; this favours bit-blasting.

Also note that the composed word-level solver outperforms the decomposed word-level solver in most cases. As noted, propagation by the composed propagators is stronger than by the decomposed ones. When the test cases become harder, composed propagators tend to infer more information.

However, we also find that, in many cases, the word-level solvers dramatically outperform STP, even though the overall run time for each folder is greater. To better understand what kind of test case favours word-level solvers and which favour STP, we have collected those cases that had significant solving time difference between the solvers (> 1 second), in total 2979 cases. We call this collection the *core set*. Small time differences are more likely to be attributed to irrelevant factors, such as I/O and machine workload, rather than to solving strategy, which is why we ignore the less-than-1-second cases. In the core set there are 825 test cases (less than 30%) where STP wins and 2103 cases (about 70%) where Comp-W wins. In the remaining cases, Decomp-W wins (51 cases, or less than 2%).

Table 2 lists the benchmark families where word-level solvers perform better, and Table 3 list cases where STP wins. To get a better understanding of which factors influence performance, we have collected various data. The tables provide the number of test cases solved in each family (num), together with various averages taken within each family: the bit-width of bit-vector constraints (bwid); the number of bit-vector constraints (bitvec); the number of Boolean operations (bool); the ratio of bit-vector operations to Boolean operations (vec/bool); the number of logical constraints (logic); the number of structural constraints (struc); the number of arithmetic constraints (arith); the number of comparison constraints (comp) including **=**, **distinct**, **bvule**, **bvult**, **bvuge**, **bvugt**, **bvsle**, **bvslt**, **bvsge** and **bvsgt**; the number of addition related constraints (adder) including **bvadd**, **bvneg** and **bvsub**; the number of multiplication related constraints (multip) including **bvmul**, **bvudiv**, **bvurem**, **bvsdiv**, **bvsrem** and **bvsmod**.

In Tables 2 and 3, we combine the cases where Comp-W wins with the cases where Decomp-W wins, to show the test cases where *some* word-level solver wins (WLS). We show the average solving time (in seconds) using STP and WLS respectively. Since more than half of the test cases appear in the "spear" family (which may bias the analysis), we also provide a summary row (sum(w/o spear)) which excludes the data from the spear family.

In terms of the solving time and the number of solved test cases in different families, the solving ability of STP and the word-level solvers are generally complementary. STP is significantly better at solving "brummayerbiere (2,3)", "float" and "sage" families. Word-level solving is significantly better for the "bruttomesso/core" and "spear" families. For the rest of the families, the

Table 2 Problems for which word-level solvers outperform STP

Problem	num	bwid	bitvec	bool	vec/bol	logic	struc	arith	comp	adder	multip	STP	WLS
asp	137	7	9301	15795	0.6	2	0	9299	5498	3801	0	125.3	65.6
bmc-bv	7	41	1207	2281	0.9	0	193	1014	846	20	6	2.0	1.6
bmc-bv-svcomp14	16	30	1362	1899	6.9	3	19	1340	502	12	10	4.3	3.3
brummayerbiere	2	39	255	1713	1.4	0	38	218	34	47	16	70.7	22.4
brummayerbiere2	2	12	29	85	0.4	14	7	8	2	0	2	252.7	33.7
brummayerbiere3	4	25	132	10	18.8	27	38	68	11	39	3	50.0	6.3
bruttomesso/core	131	7	1264	554	4.1	0	864	418	418	0	0	181.6	15.7
bruttomesso/lfsr	5	14	772	871	0.9	0	384	388	388	0	0	95.9	39.1
calypto	2	21	143	4	35.6	0	89	54	18	4	3	255.2	15.0
fft	2	4	1059	1161	0.9	0	549	510	138	372	0	251.2	45.2
float	25	22	2546	1613	1.5	423	915	1208	530	447	12	54.8	25.0
gulwani-pldi08	4	8	1234	400	3.1	3	0	1231	419	490	322	14.7	11.2
mcm	24	11	5563	2785	2.1	350	362	4851	2505	2205	7	204.2	112.4
sage	328	30	821	137	6.7	92	141	588	183	320	85	10.1	3.3
spear	1378	39	1640	425	3.7	0	111	1530	322	42	15	2.8	0.3
stp	1	30	19382	38789	0.5	688	660	18034	17956	59	19	10.4	8.4
tacas07	1	32	1186	1501	0.8	1	0	1185	1045	139	1	10.1	4.5
uclid	80	29	829	108	7.8	138	295	396	251	75.5	69	0.5	0.4
uum	1	9	302	192	1.6	69	155	78	71	0	7	108.9	25.6
VS3	2	6	2507	266	10.8	0	0	2507	386	960	1161	466.3	170.2
Wienand-cav2008	1	4	274	151	1.8	0	172	102	1	95	6	346.0	60.6
summary	2153	32	1998	1432	4.2	29	174	1797	670	353	27	27.1	7.8
sum (w/o spear)	775	21	2634	3222	4.9	79	284	2273	1289	905	49	70.3	21.3

Table 3 Problems for which STP outperforms word-level solvers

Problem	num	bwid	bitvec	bool	vec/bol	logic	struc	arith	comp	adder	multip	STP	WLS
asp	91	7	8619	12942	0.8	0	0	8619	4931	3687	0	95.6	261.9
bmc-bv	2	32	3611	28850	0.1	102	302	3208	805	1301	202	30.2	53.4
bmc-bv-svcomp14	17	27	725	757	23.6	48	88	588	215	63	15	3.6	13.0
brummayerbiere	4	42	360	114	3.2	87	71	203	45	83	0	66	132.0
brummayerbiere2	4	30	16	367	0.1	7	6	4	1	0	2	0.7	251.4
brummayerbiere3	33	23	460	195	11.2	176	87	196	50	14	8	18.5	263.9
bruttomesso/lfsr	66	44	923	977	1	0	450	473	461	308	0	63.1	436.4
bruttomesso/core	1	34	38	10	3.8	0	26	12	12	0	0	0.2	0.5
bruttomesso/simple-processor	25	10	226	126	1.7	20	72	134	114	20	0	13.1	293.0
calypto	1	6	57	226	0.3	0	2	55	18	0	0	0	0
ecc	4	32	73	476	0.2	3	6	65	40	4	1	0.1	0.2
fft	3	4	617	669	1	0	323	295	79	215	0	28.9	53.6
float	89	22	5710	3228	1.6	1164	1860	2685	1052	962	28	80.4	293.4
gulwani-pldi08	1	6	883	288	3.1	10	0	873	310	335	228	0.8	2.4
mcm	26	11	9010	4815	2	375	434	8200	3900	4178	0	133.5	235.3
rubik	4	3	1616	409	3.9	0	8	1609	199	0	0	56.8	254.4
sage	314	30	441	118	6.3	33	62	346	153	167	25	4.1	19.8
spear	104	41	3472	894	5.9	18	247	3208	679	108	17	3.1	41.0
tacas07	1	32	4601	6469	0.7	0	0	4601	3794	74	0	331.6	500
uclid_contrib_smtcomp09	2	20	2937	3133	1	23	254	2661	1696	69	0	239.5	500
uclid	32	28	829	108	7.7	137	295	401	252	79	70	0.5	0.8
summary	824	27	2640	2293	4.9	167	327	2146	995	755	19	33.5	139.9

winner is less clear. STP tends to win decisively when it wins. On the other hand, the word-level solvers win in the majority of cases. The summary rows (summary and sum (w/o spear)) of Tables 2 and 3 suggest, if anything, that STP handles test cases with more logical and structural constraints better, while the word-level solvers work better for test cases with more arithmetic constraints, especially multiplication related constraints (multip).

To test the hypothesis that the word-level solvers will handle difficult arithmetic constraints better, we generated three sets of benchmarks to stress test the solvers on non-linear constraints. In these sets, all bit-vector variables have width 64. Superscripts denote exponentiation, that is, v^n is $v \cdot v \cdots v$, with v occurring n times. (An expression such as $v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28}$ is shorthand for the obvious conjunction, $(v_1^{28} \leqslant v_2^{28}) \wedge (v_2^{28} \leqslant v_3^{28})$.)

- Set 1:
  $(v_1^{28} \leqslant v_2^{28}) \wedge (v_1^{28} \geqslant v_2^{28})$;
  $(v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28}) \wedge (v_1^{28} \geqslant v_2^{28} \geqslant v_3^{28})$;
  $(v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28} \leqslant v_4^{28}) \wedge (v_1^{28} \geqslant v_2^{28} \geqslant v_3^{28} \geqslant v_4^{28})$;
  $\vdots$
  from 2 variables to 20 variables (altogether 19 test cases).

- Set 2:
  $(v_1^{28} \leqslant v_2^{28}) \wedge (v_1^{28} \geqslant v_2^{28}) \wedge (v_1^{28} \neq v_2^{28})$;
  $(v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28}) \wedge (v_1^{28} \geqslant v_2^{28} \geqslant v_3^{28}) \wedge (v_1^{28} \neq v_2^{28}) \wedge (v_2^{28} \neq v_3^{28})$;
  $(v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28} \leqslant v_4^{28}) \wedge (v_1^{28} \geqslant v_2^{28} \geqslant v_3^{28} \geqslant v_4^{28}) \wedge (v_1^{28} \neq v_2^{28}) \wedge (v_2^{28} \neq v_3^{28}) \wedge (v_3^{28} \neq v_4^{28})$;
  $\vdots$
  from 2 variables to 20 variables (altogether 19 test cases).

- Set 3:
  $(v_1^{28} = v_2^{28} = v_3^{28}) \wedge (v_1^{28} \neq v_3^{28})$;
  $(v_1^{28} = v_2^{28} = v_3^{28} = v_4^{28}) \wedge (v_1^{28} \neq v_4^{28})$;
  $(v_1^{28} = v_2^{28} = v_3^{28} = v_4^{28} = v_5^{28}) \wedge (v_1^{28} \neq v_5^{28})$;
  $\vdots$
  from 3 variables to 20 variables (altogether 18 test cases).

- Set 4:
  $(v_1^{28} \leqslant v_2^{28}) \wedge (v_1^{28} \geqslant v_2^{28}) \wedge (v_1^{28} \neq v_2^{28})$;
  $(v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28}) \wedge (v_1^{28} \geqslant v_2^{28} \geqslant v_3^{28}) \wedge (v_1^{28} \neq v_3^{28})$;
  $(v_1^{28} \leqslant v_2^{28} \leqslant v_3^{28} \leqslant v_4^{28}) \wedge (v_1^{28} \geqslant v_2^{28} \geqslant v_3^{28} \geqslant v_4^{28}) \wedge (v_1^{28} \neq v_4^{28})$;
  $\vdots$
  from 2 variables to 20 variables (altogether 19 test cases).

– Set 5:
$(v_1^2 = v_2^2) \wedge (v_2^2 = v_3^2) \wedge (v_1^2 \neq v_3^2)$;
$(v_1^3 = v_2^3) \wedge (v_2^3 = v_3^3) \wedge (v_1^3 \neq v_3^3)$;
$(v_1^4 = v_2^4) \wedge (v_2^4 = v_3^4) \wedge (v_1^4 \neq v_3^4)$;
$\vdots$
with powers running from 2 to 28 (altogether 27 test cases).

Figure 4 shows that for test sets 1, 2, 3 and 5, the word-level solver outperforms STP significantly in speed, and it has a more uniform behaviour. In Sets 1, 2 and 3, as the number of the variables increase, the solving ability of STP declines dramatically. In Set 5, STP (not unexpectedly) shows multifarious behaviour as exponents grow. On the whole, word-level solving appears better at handling constraints that involve multiplication. Set 4, however, shows that word-level solving may still fail spectacularly where long chains of reasoning are required (compare Set 2 and Set 4).

All of these results point to the possibility of a portfolio solver that combines the three solvers, so that we can gain some benefit from each. However, our analysis so far is not sufficiently detailed to allow a reliable classification of input cases. Hence we turn to machine learning to try to classify input cases according to their suitability for the individual solvers.

7 Wombit: A Portfolio Solver

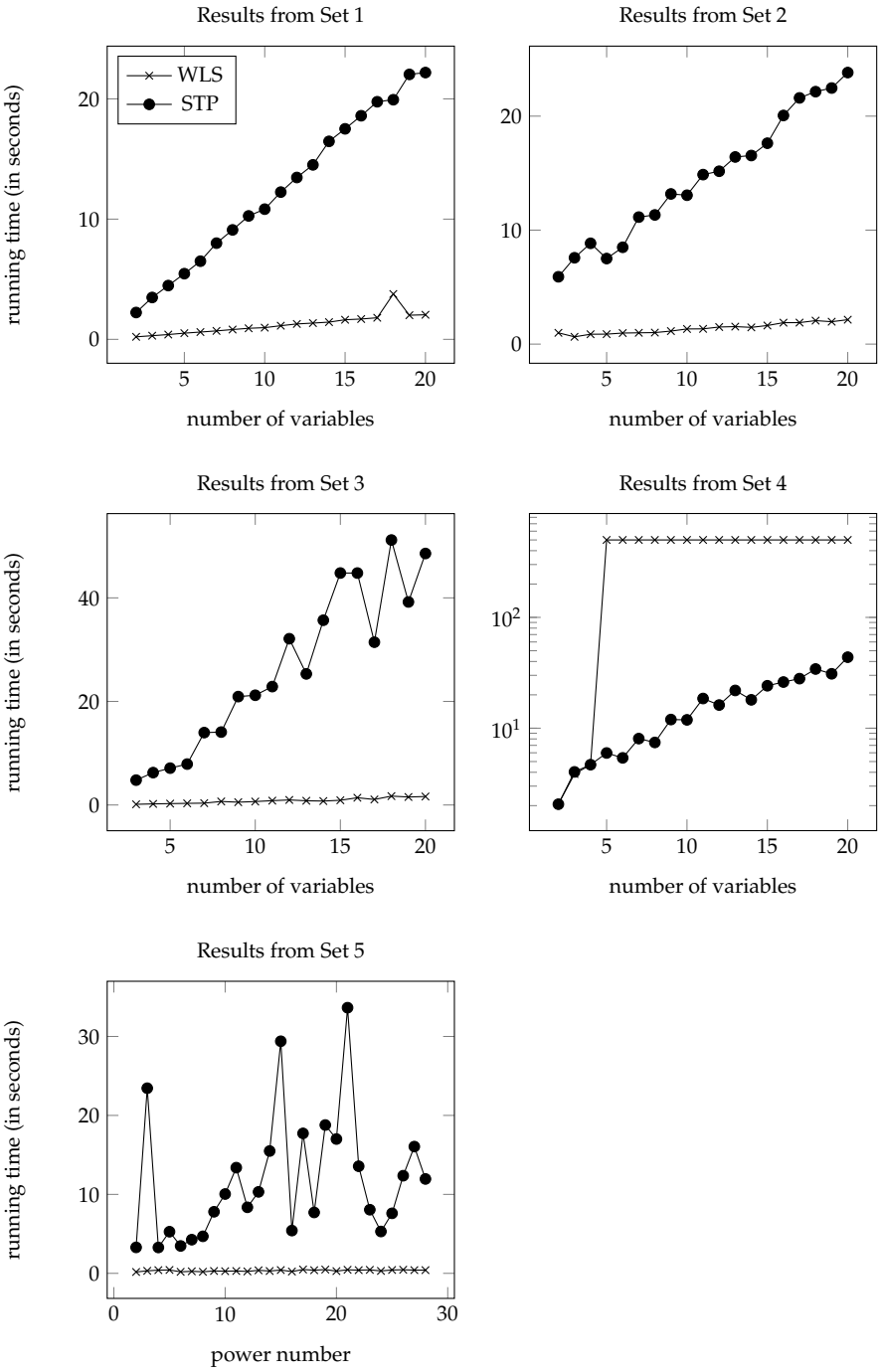
Wombit is a portfolio solver. It uses machine learning to predict which solver among STP, Comp-W, Decomp-W is best at solving a new unseen case. Section 7.1 recapitulates the basic concepts and ideas in machine learning, specially for classification problems. Sections 7.2 and 7.3 show the experimental result of the simulated portfolio solver and the actual portfolio solver called Wombit respectively, comparing with STP.

7.1 Classification in Machine Learning

Machine learning provides an automated method of data analysis, which can detect patterns in data, and use the uncovered patterns to predict future data [52]. Machine learning is usually divided into two main types: *supervised* learning and *unsupervised* learning. The supervised learning strategy is commonly used for classification problems. The goal is to learn a mapping from the inputs to one of the particular categories. If there are two categories, it is called *binary classification*. If there are more than two categories, it is called *multiclass classification*. We use the following terminology:

– **Training data**: data used for training the algorithm.
– **Test data**: data used for testing the algorithm.
– **Dataset**: contains the training data and the test data.

Fig. 4 Word-level solver vs STP on multiplication constraints evaluation



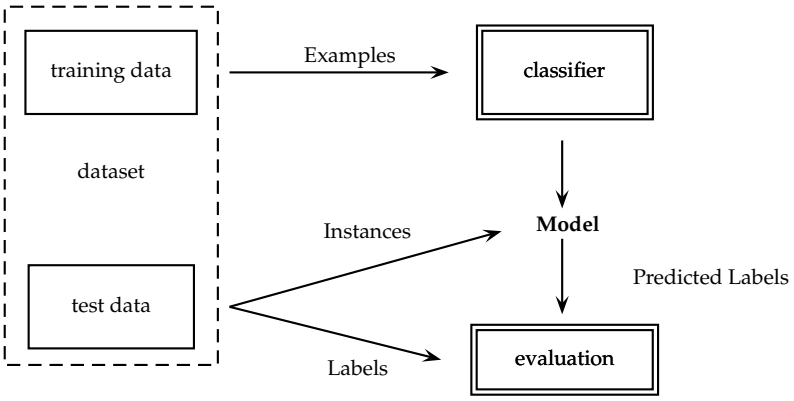


Fig. 5 Architecture of a supervised learning strategy

- **Instance**: a single training or test item, usually described in terms of its features.
- **Features**: attributes used to characterise an instance and determine its classification.
- **Label**: the category associated to an object.
- **Example**: an instance coupled with a label.
- **Model**: discovers the relationship between the features and the label.

The dataset can be divided into the training data which are used for training the model, and the test data which are used for evaluating the model.

7.1.1 Architecture of Supervised Learning Strategy

The supervised learning strategy for classification proceeds in four steps, as depicted in Figure 5. Firstly, we divide the labelled dataset into training and test data. Secondly, we need to choose an appropriate type of the machine learning algorithm adapted to the problem for the classifier. Thirdly, we use the learner which applies the selected algorithm to train a model on the training data. Finally, we use the labels predicted by the model and the actual labels of examples in the test data to evaluate the model, and get the prediction accuracy. Note that there are traditionally many kinds of algorithms applied in the learner for the classification problem, such as Logistic Regression [49], Support Vector Machines [20], Boosting [61], Decision Trees [59], Neural Networks [40] and others.

7.1.2 Cross Validation

To reduce the possibility of over-fitting, we use 10-fold cross-validation. In k-fold cross-validation, the original sample is randomly partitioned into k

equal-sized sub-samples called *folds* (typically, k=5 or 10). A model is trained on all folds but the i^{th} fold. The generated model is then used to test on the i^{th} fold, that is, to obtain the predicted labels for the i^{th} fold. This is done for each $i \in [1,k]$. The prediction accuracy for the whole dataset is the proportion of the correctly predicted labels among the labels in the whole dataset. The advantage of this method is that every example gets to be in a test set exactly once, and gets to be in a training set $k-1$ times. The variance of the resulting estimate is reduced as k is increased. For classification problems, one typically uses stratified k-fold cross-validation, so that each fold contains roughly the same proportions of the class labels.

7.2 A Simulated Portfolio Solver Based on Machine Learning

To use machine learning for classification we first need a significant dataset with good features to help accurately classify the cases. Second, we need to choose an appropriate machine learning method. We use the *core* set identified in Section 6 as our dataset. It consists of 2979 test cases with significant solving time difference between the solvers (> 1 second). We classify the dataset into three categories (labels): those where STP wins (825 cases), those where Comp-W wins (2103 cases), and those where Decomp-W wins (51 cases). We have chosen 30 features in total, naturally divided into four categories:

- the total number of bit-vector operations, the total number of logical and structural operations, the total number of arithmetic operations and the total number of Boolean operations;
- the number of constants and variables;
- the ratio of the bit-vector operations and the Boolean operations, and the ratio of the logical operations and the arithmetic operations; and
- the number of each specific bit-vector operation.

As machine learning method we choose the well-known C4.5 algorithm [60] for its proven performance. It builds a model called a decision tree. We did 10-fold cross validation on the collected dataset (core set), using 30 features. The prediction accuracy was 86%.

Using the prediction of which solver to use and the solving time of each solver shown in Table 1, we can simulate the solving time for a portfolio solver on the collected dataset (core set). Table 4 shows the result. Times are in seconds, and "TO" is the number of timed-out cases. We also show the optimal solving time by the virtual best solver (VBS). Compared to STP, the simulated portfolio solver uses less time and solves 35 more test cases.

7.3 Using the Machine Learning Model

Motivated by the promising result of the simulated solver, we have implemented a portfolio solver called Wombit using the decision tree model trained

Table 4 STP vs word-level solvers vs simulated solver (2979 test cases)

Solver	STP		Comp-W		Decomp-W		Simulated Solver		VBS	
	time (s)	TO	time (s)	TO	time (s)	TO	time (s)	TO	time (s)	TO
Total	49586	74	49612	300	103992	664	46291	39	45257	0
Overall time (s)	86458		199612		435992		65790		45257	

Table 5 STP vs Wombit (numbers in the last three columns are solving time in seconds)

Solver	cases	STP	Portfolio Solver	VBS
non_core_easy	27859	1087	1108	823
non_core_hard	585	292500	292500	292500
core	2979	86459	60523	45257
Total	31422	380046	354131	338580

on the whole core data set (the 2979 test cases). The accuracy is 92% on the core data set. We use Wombit to run on the whole data set which contains 31422 test cases in total. The experiment setup is exactly the same as in Section 6. The overall time result (in seconds) is shown in Table 5. We also show the optimal solving time by the virtual best solver (VBS). We split the whole data set into the core set which is the training set, and the non-core data set which contains cases without significant solving time difference. We further split the non-core data set into the easy set with solving time less than 4 seconds, and the hard set with those cases for which all solvers timed out.

From the result of the non-core easy data set, we can tell that there is little time overhead in deciding which solver to use. This is because we only need to traverse the input parse tree one more time to get the features and run the decision tree to get the suggestion of which solver to use. We also see that the main solving time difference comes from the core data set where 45 more cases are solved by Wombit, compared to STP. In general, Wombit outperforms each component solver (STP, Comp-W, Decomp-W) significantly.

7.4 Threats to Validity

The main concern about the experimental evaluation is whether *over-fitting* may have occurred, that is, whether the generated decision tree is designed to fit the training data so closely that it becomes inaccurate for other (untrained) data. If it has, then our conclusions about the advantages of the portfolio approach may not remain valid once the approach is applied more broadly.

As mentioned, the 10-fold cross validation scheme has been used to reduce the risk of over-fitting from the outset. Beyond that, we can hope to *detect* over-fitting (if present) as new datasets appear and we apply Wombit to those. However, new untrained data becomes available at a fairly slow rate. After all, SMT-LIB2 files are rather specialised data. It is not clear what it means for a file to be typical or representative. The competition benchmarks that we have used originate from a variety of sources and applications, but it is still possible that they exhibit an undesirable lack of variety.

Table 6 STP vs word-level solvers vs Wombit in recent test suites (Timeout: 500 seconds)

Problem	STP	Comp-W	Decomp-W	Wombit	
	time (s)	time (s)	time (s)	time (s)	Choice
BuchwaldFried-counter	0	0	0	0	-
BuchwaldFried-Sh132	500	500	500	500	Comp-W
BuchwaldFried-Or32.Or32	500	500	500	500	Comp-W
BuchwaldFried-Minus32	500	500	500	500	Comp-W
20170501-Heizmann	500	500	10	11	Decomp-W
20170531-Hansen-zero	0	0	0	0	STP
20170531-Hansen-zero0	0	0	0	0	STP
20170531-Hansen-zero1	0	0	0	0	STP
Overall time (s)	2000.0	2000.0	1510	1511	NA

The data used in our evaluation are from the 2015 SMT competition. For what it is worth, Table 6 shows how the different solvers perform on the benchmarks (8 new cases) that were added for the 2017 competition (after our learner was trained). The '-' indicates no choice was made by Wombit, which means the problem was already solved by the original STP's word-level simplifications. At least on this limited evidence, Wombit appears to make sensible decisions when presented with new cases.

8 Related Work

SAT and SMT solvers have been developed for several decades, and are applied in all kinds of areas. We refer to [56] for an overview of the SAT and SMT solvers. Constraint programming [48] also has a long history which can be traced back to Artificial Intelligence. Here we focus on the word-level reasoning for the bit-vector logic (Section 8.1), and the portfolio constraint solvers using machine learning techniques (Section 8.2).

8.1 Word-Level Reasoning for Bit-Vector Logic

Word-level reasoning on bit-vector logic is NEXPTIME-complete [42]. In spite of this, the problem has received much attention recently, albeit with limited progress. Current related work falls into one of or the combination [2] of three categories: reasoning based on lazy SMT techniques, reasoning based on constraint programming, reasoning based on linear programming, and reasoning based on stochastic local search.

8.1.1 Word-Level Reasoning Based on Lazy SMT Techniques

Hadarean et al. [34] propose a lazy bit-vector solver which is organized as a sequence of sub-solvers: two specialised bit-vector solvers at the word level, namely an Equality Solver and an Inequality Solver, an "in-processing" solver which can reduce the bit-blasted formulas when possible, and a bit-blasting solver at the end. If any of the solvers find unsatisfiability the whole process

ends and returns UNSAT. Similarly, if one of the solvers finds complete satisfiability the searching process stops and returns SAT. The lazy bit-vector solver can be seen as a semi-word-level lazy solver. The Equality Solver utilizes a variant of the incremental polynomial-time congruence-closure algorithm [43], while the Inequality Solver relies on an interesting incremental algorithm which maintains the least valuation of the inequality constraints. The advantage of this method is that specific algorithms can be used for solving the particular group of constraints. There are also disadvantages. Firstly, solving the equality and inequality constraints in separate groups may propagate little information since they cannot interact with other bit-vector constraints. Secondly, neither can express a conflict on bit-level which significantly affects the efficiency. Besides, the costs of the interaction between the Theory solver and the SAT solver cannot be ignored, even though a so-called justification heuristic engine is applied to reduce the cost.

Our word-level solvers, in contrast, process all bit-vector constraints using word-level propagators, with a learning mechanism that hooks straight into MiniSAT's learning. In this way, the word-level propagators are embedded in MiniSAT, so that unit propagation seamlessly interleaves with word-level propagation at practically no overhead.

8.1.2 Word-Level Reasoning Based on Constraint Programming

Bardin et al. [14] propose two kinds of word-level propagators based on two kinds of domains in the Constraint Logic Programming framework. One is called *Is*/*C* propagator which is built on the union of integer intervals plus congruence domain and is good at solving linear arithmetic constraints. The other is the *BL* (Bit-List) propagator which is built on a "bit-list" domain and runs in linear time to solve the bitwise constraints. Each variable is associated with a numerical domain *Is*/*C*, and a *BL* domain (which is similar to the dedicated domain proposed by Baray et al. [13]); and each constraint has two associated finite sets of propagators: *Is*/*C* and *BL* propagators. In addition, specific propagators are designed to ensure the consistency between the two kinds of domain.

Note that even though the *BL* domain is equivalent to our "trit-vector" presentation, the propagators for bitwise operations typically run in time $O(k)$ instead of constant time for our domain, since bit operations of the underlying architecture are not exploited. Furthermore, the implementation involves bit-blasting and the use of additional propagators to ensure the consistency of the two domains which also affects the efficiency. However, the work by Bardin et al. [14] has narrowed the gap between the word-level solving approach and the SAT solving approach considerably, and the gap continues to narrow. Chihani *et al.* [22] extend the work of Bardin et al. [14] by utilising the representational idea of Michel and Van Hentenryck [50] (what we have called lo-hi form) to build a CP-based bit-vector solver which enables channelling with other constraint domains, such as bounds constraints and global difference constraints [29]. Chihani, Bobot and Bardin [21] further argue

for *word-level* conflict analysis and learning, to replace the bit-learning that we have delegated to a SAT solver.

Constraint bound propagators for modular arithmetic constraints have been proposed by Gotlieb et al. [33] who utilize a domain called clockwise intervals (*CI*). The main idea is to find optimal bounds in bound-consistency filtering of modular integer computations. This approach often requires approximations and loses efficiency when it encounters general multiplication and division.

Zeljić et al. [72] propose a word-level bit-vector solver which is developed based on a Model-Constructing Satisfiability Calculus (mcSAT) proposed by Jovanović and de Moura [39, 24]. The novel part of their approach is that they apply a tailor-made conflict-driven learning strategy which exploits both the propositional and arithmetic properties of the bit-vector operations. However the cost of conflict generation is relatively high, and the propagation they use for the bit-vector constraints is simple bound propagation which might affect the propagation strength.

Note that no propagator in these CP approaches gives the explanation for the fixed literals which might impact the learning effect.

In addition, Wille et al. [71] look from a different angle in their SWORD tool which is a SAT solver which facilitates the word-level information to increase the performance of the SAT solver. They represent some sub-formulas in terms of modules defined over bit-vector operations which are handled similar to custom propagators in a constraint solver.

8.1.3 Word-Level Reasoning Based on Linear Programming

This approach is to transform the problem into linear programming constraints and is often applied in register transfer level (RTL) verification which is a well-known hardware verification problem.

Brinkmann et al. [18] propose a method to transform conjunctions of bit-vector equalities and inequalities into sets of integer linear arithmetic constraints and solve them at word-level with an integer linear programming solver. In particular, the bit-vector variables are translated into linear terms.

The mixed integer linear programming (MILP) approach proposed for RTL verification is LP-based SAT solving. Fallah et al. [28] have designed an approach called HSAT which generates linear arithmetic constraints for word-level operations and conjunctive normal form clauses for Boolean constraints. Zeng et al. [73] linearize both the word-level constraints and the Boolean constraints in integer linear constraint problems, in a unified MILP solver called LPSAT. For RTL verification on the word-level, the performance of LP solvers is often no better than SMT solvers [44]. SMT solvers are found to do complete verification checking in fewer iterations and in less time.

8.1.4 Word-Level Reasoning Based on Stochastic Local Search

Stochastic Local Search (SLS) is a heuristic method which has played an important role in AI [36]. It was first applied in SMT solvers by Fröhlich et al. [30] to solve bit-vector problems directly on the theory level. SLS is generally incomplete—it does not provide proof procedures for unsatisfiability. It can however be very efficient on satisfiable cases, including hard cases, and it is therefore highly attractive as a component of combined strategies. Niemetz et al. [55] extended the approach of [30], finding improved heuristics for neighbour selection, based on down propagation of assignments. Niemetz et al. [54] further simplified, extended and formalized the approach. The result is a bit-vector solver which is "probabilistically asymptotically complete" in the sense of [35]. It finds use in Boolector [53], as part of a solver portfolio, and its inclusion there has been found to be beneficial for performance [54].

Our approach is complete in the usual sense and aims to exploit word-level propagation while utilizing bit-level search. An SLS approach aims to exploit word-level search to facilitate the word-level reasoning. Both approaches subscribe to the idea that combining word-level reasoning with bit-blasting can yield significant solver improvements.

8.2 Portfolio CSP Solvers Using Machine Learning

The interest in algorithm selection and configuration for constraint satisfaction problem (CSP) solving is growing. The reason is that algorithm selection and configuration is crucial for the performance of a portfolio solver. Machine learning techniques are usually applied to solve this problem based on the features of the input case. We refer to [38, 41, 64] for overviews of the general portfolio approaches.

CPHydra [58] is the first portfolio CSP solver which applies a machine learning technique called *k*-nearest neighbour algorithm (*K-NN*) to exploit the instances of similarity and to schedule the component solvers. There are 36 extracted features including the static syntactic ones and the dynamic ones which are solver specific. And the dynamic features, generated by running a constraint solver called *Mistral* for a limited amount of time (typically two seconds), are modelling choices and search statistics, such as the number of constraint propagation calls and the number of nodes explored. Each instance contains the features and the solving time of each component solver. CPHydra combines machine learning with the idea of partitioning CPU-time between the component solvers to schedule the solvers and maximize the expected number of solved problem instances within a settled time limit. CPHydra won the 2008 International CSP Solver Competition.

SUNNY [4] is a lazy portfolio approach which uses the *K-NN* algorithm just as CPHydra. The novel idea in SUNNY is that it applies three heuristics to decide the order of the component solvers to be run, to minimize the average solving time of each instance. The first, denoted h_{sel} is for selecting the most

allocating a certain run time to each component solver; the third, denoted h_{sch} is for sorting the component solvers by increasing allocated solving time. Variants of SUNNY have been proposed, namely a sequential portfolio solver called sunny-cp [6], and a parallel solver called sunny-cp2 [5] which won the gold medal in the open category of MiniZinc Challenge [67] in 2015 and 2016.

In addition, Stojadinović et al. [66] propose a simplified *K-NN* based portfolio CSP solver which has a short training phase but achieves state-of-art performance. Loreggia et al. [47] introduce an automated way for generating an informative set of features by training a neural network on images extracted from problem instances. An evaluation of the portfolio approaches for CSPs is presented by Amadini [7, 3].

Some have looked at the problem from other angles. Arbelaez et al. [8, 9] use support vector machines (SVM) to dynamically adapt the search heuristics of a single CSP solver. Stojadinović et al. [65] and Hurley et al. [37] propose portfolio CSP approaches for selecting among different SAT encodings, instead of CSP solvers.

The most relevant work to Wombit is presented by Abdul Aziz et al. [11] who use a linear machine learning technique called Ridge regression to estimate the hardness of QF_BV SMT problems. The features selected are similar to ours which are all syntactic ones of the input SMT formula and cheap to get in hand. The result shows that the extracted features characterize the hardness of the problem well.

9 Conclusion

We have implemented a bit-vector word-level solver which can solve all the operations in the SMT-LIB2 QF_BV category. We have applied and also extended word-level propagation algorithms of Michel and Van Hentenryck [50] to produce an explaining word-level solver based on a SAT solver called MiniSAT, by generating the explanation for each fixed literal and the conflict clause when conflict happens.

We have proposed three dimensions in the design space for a word-level solver: two ways to create the propagators for reified constraints (composed propagator vs decomposed propagator); two ways to generate the explanation (forward explanation vs backward explanation); and two ways to do the conflict analysis (standard backjumping vs multi-conflict backjumping). We have given an empirical comparison of these design options, and an empirical comparison of the word-level propagation versus bit-blasting, the standard approach to these problems. Furthermore, we applied the word-level simplification in a practically used SMT solver called STP as the preprocessing of our word-level solvers, and did an empirical comparison with STP.

Motivated by the comparison with STP, we have built a portfolio solver called Wombit based on machine learning techniques. This solver combines the two word-level solvers with STP to try to gain the benefits from different approaches. Results show that, with careful engineering, a word-level

propagation approach can be competitive with, or a useful supplement to a bit-blasting/SAT solving approach.

For future work, it is worthwhile applying the word-level solving method to other state-of-the-art SMT solvers, such as Boolector [53], and comparing with those state-of-the-art SMT solvers. It may also be advantageous to let Wombit use the lazy decomposition approach proposed by Abío and Stuckey [1], as a supplement to the decision making of when to use SAT solving. Furthermore, a mixed way of doing word-level solving and bit-blasting could be considered, such as bit-blasting the logical constraints and focusing word-level solving on the arithmetic constraints. Another interesting line of research would be to combine word-level propagation with word-level search which we believe would make a significant difference. As we have shown, the word-level propagation can be seen as doing the bit propagation in parallel. If a word-level search is applied to guess a "word" instead of a single bit, the parallel benefit would be exploited much more effectively.

Acknowledgements We wish to thank the reviewers of this paper for very detailed and helpful comments and suggestions. We acknowledge support from the Australian Research Council through ARC Discovery Grant DP140102194.

References

1. Ignasi Abío and Peter J. Stuckey. Conflict directed lazy decomposition. In M. Milano, editor, *Principles and Practice of Constraint Programming: Proceedings of the 18th International Conference*, volume 7514 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2012.
2. Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In L. Perron and M. A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5015 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2008.
3. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving CSPs. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: Proceedings of the 10th International Conference*, Lecture Notes in Computer Science, pages 316–324. Springer, 2013.
4. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: A lazy portfolio approach for constraint solving. *Theory and Practice of Logical Programming*, 14(4–5):509–524, 2014.
5. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A multicore tool for constraint solving. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*, pages 232–238. AAAI Press, 2015.

6. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: A sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15)*, pages 1861–1867. ACM, 2015.
7. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An extensive evaluation of portfolio approaches for constraint satisfaction problems. *International Journal of Interactive Multimedia and Artificial Intelligence*, 3(7):81–86, 2016.
8. Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Online heuristic selection in constraint programming. In *Proceedings of the International Symposium on Combinatorial Search*, 2009. `https://hal.inria.fr/inria-00392752/`.
9. Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous search in constraint programming. In Y. Hamadi et al., editors, *Autonomous Search*, chapter 9, pages 219–243. Springer, 2011.
10. Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.
11. Mohammad Abdul Aziz, Amr Wassal, and Nevine Darwish. A machine learning technique for hardness estimation of QFBV SMT problems. In P. Fontaine and A. Goel, editors, *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT'12)*, volume 20 of *EPiC Series in Computing*, pages 57–66. EasyChair, 2013.
12. Domagoj Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.
13. Fabrice Baray, Philippe Codognet, Daniel Diaz, and Henri Michel. Code-based test generation for validation of functional processor descriptions. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 569–584. Springer, 2003.
14. Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An alternative to SAT-based approaches for bit-vectors. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2010.
15. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.
16. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification: Proceedings of the 23rd International Conference (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.
17. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64,

Wrocław, Poland, 2011.
18. Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the Asia and South Pacific Design Automation Conference and VLSI Design 2002*, pages 741–746. IEEE Computer Society, 2002.
19. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
20. Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, 2011.
21. Zakaria Chihani, François Bobot, and Sébastien Bardin. CDCL-inspired word-level learning for bit-vector constraint solving, 2017. HAL, URL `https://hal.archives-ouvertes.fr/hal-01531336`.
22. Zakaria Chihani, Bruno Marre, François Bobot, and Sébastien Bardin. Sharpening constraint programming approaches for bit-vector theory. In D. Salvagnin and M. Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming: Proceedings of the 14th International Conference*, volume 10335 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.
23. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
24. Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification Model Checking and Abstract Interpretation: Proceedings of the 14th International Conference*, volume 7737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013.
25. Bruno Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.
26. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT'04)*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer, 2004.
27. Christoph Erkinger. Rotating workforce scheduling as satisfiability modulo theories. Master's thesis, Vienna University of Technology, 2013.
28. Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. Functional vector generation for HDL models using linear programming and 3-satisfiability. In *Proceedings of the 35th Annual Design Automation Conference (DAC'98)*, pages 528–533. ACM, 1998.
29. Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey. Global difference constraint propagation for finite domain solvers. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of*

Declarative Programming (PPDP'08), pages 226–235. ACM, 2008.
30. Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pages 1136–1143. AAAI Press, 2015.
31. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.
32. Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In M. Carro and R. Pena, editors, *Practical Aspects of Declarative Languages (PADL'10)*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010.
33. Arnaud Gotlieb, Michel Leconte, and Bruno Marre. Constraint solving on modular integers. In *Proceedings of the Ninth International Workshop on Constraint Modelling and Reformulation (ModRef'10)*, 2010.
34. Liana Hadarean, Clark Barrett, Dejan Jovanović, Cesare Tinelli, and Kshitij Bansal. A tale of two solvers: Eager and lazy approaches to bit-vectors. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV'14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 680–695. Springer, 2014.
35. Holger Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pages 661–666. AAAI Press, 1999.
36. Holger Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2004.
37. Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In H. Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming: Proceedings of the 11th International Conference (CPAIOR'14)*, volume 8451 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2014.
38. Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: The state of the art, 2012. CoRR, `http://arxiv.org/abs/1211.0906`.
39. Dejan Jovanović and Leonardo de Moura. Cutting to the chase. *Journal of Automated Reasoning*, 51(1):79–108, 2013.
40. Bart Kosko. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Prentice-Hall, 1992.
41. Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35(3):48–60, 2014.
42. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, 59(2):323–376, 2016.
43. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

44. Sarvani Kunapareddy, Sriraj D. Turaga, and Solomon S. T. M. Sajjan. Comparison between LPSAT and SMT for RTL verification. In *Proceedings of the 2015 International Conference on Circuit, Power and Computing Technologies*, pages 1–5. IEEE Computer Society, 2015.
45. K. Rustan M. Leino. This is Boogie 2, 2008. Unpublished manuscript.
46. Rhishikesh S. Limaye and Sanjit A. Seshia. Beaver: An SMT solver for quantifier-free bit-vector logic. Master's thesis, University of California, Berkeley, 2010.
47. Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay A. Saraswat. Deep learning for algorithm portfolios. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1280–1286. AAAI Press, 2016.
48. Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.
49. P. McCullagh and J. A. Nelder. *Generalized Linear Models*. Chapman & Hall, second edition, 1989.
50. Laurent D. Michel and Pascal Van Hentenryck. Constraint satisfaction over bit-vectors. In M. Milano, editor, *Constraint Programming: Proceedings of the 2012 Conference*, volume 7514 of *Lecture Notes in Computer Science*, pages 527–543. Springer, 2012.
51. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.
52. Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
53. Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2015.
54. Aina Niemetz, Mathias Preiner, and Armin Biere. Propagation based local search for bit-precise reasoning. *Formal Methods in System Design*, 51(3):608–636, 2017.
55. Aina Niemetz, Mathias Preiner, Andreas Fröhlich, and Armin Biere. Improving local search for bit-vector logics in SMT with path propagation. In *Proceedings of the 4th International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS'15)*, page 10 pages, 2015.
56. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
57. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
58. Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science*, pages 210–216, 2008.
59. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

60. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
61. Robert E. Schapire. The boosting approach to machine learning: An overview. In D. D. Denison et al., editors, *Nonlinear Estimation and Classification*, pages 149–171. Springer, 2003.
62. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, 2008.
63. Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. BigNum: A portable and efficient package for arbitrary-precision arithmetic. Technical Report PRL 2, DEC Paris, 1989.
64. Kate A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, 41(1):6:1–6:25, 2009.
65. Mirko Stojadinović and Filip Marić. meSAT: Multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.
66. Mirko Stojadinović, Mladen Nikolić, and Filip Marić. Short portfolio training for CSP solving, 2015. CoRR, `https://arxiv.org/abs/1505.02070`.
67. Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.
68. Wenxi Wang. A bit-vector solver based on word-level propagation. Master's thesis, Computing and Information Systems, The University of Melbourne, 2016. `https://minerva-access.unimelb.edu.au/handle/11343/120613`.
69. Wenxi Wang, Harald Søndergaard, and Peter J. Stuckey. A bit-vector solver with word-level propagation. In C.-G. Quimper, editor, *Integration of AI and OR Techniques in Constraint Programming: Proceedings of the 13th International Conference*, volume 9676 of *Lecture Notes in Computer Science*, pages 374–391. Springer, 2016.
70. Henry S. Warren Jr. *Hacker's Delight*. Addison Wesley, 2003.
71. Robert Wille, Görschwin Fey, Daniel Große, Stephan Eggersglüß, and Rolf Drechsler. SWORD: A SAT like prover using word level information. In *VLSI-SoC: Advanced Topics on Systems on a Chip: A Selection of Extended Versions of the Best Papers of the Fourteenth International Conference on Very Large Scale Integration of System on Chip*, pages 1–17. Springer, 2009.
72. Aleksandar Zeljić, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding bit-vector formulas with mcSAT. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing (SAT 2016): Proceedings of the 19th International Conference*, volume 9710 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2016.
73. Zhihong Zeng, Priyank Kalla, and Maciej Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *Design, Automation and Test in Europe (DATE'01)*, pages 398–402. IEEE Press, 2001.