Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Propagation Based Local Search for Bit-Precise Reasoning

Aina Niemetz · Mathias Preiner · Armin Biere

Received: date / Accepted: date

Abstract Many applications of computer-aided verification require bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) solvers for the theory of quantifier-free fixed-size bit-vectors. The current state-of-the-art in solving bit-vector formulas in SMT relies on bit-blasting, where a given formula is eagerly translated into propositional logic (SAT) and handed to an underlying SAT solver. Bit-blasting is efficient in practice, but may not scale if the input size can not be reduced sufficiently during preprocessing. A recent score-based local search approach lifts stochastic local search (SLS) from the bit-level (SAT) to the word-level (SMT) without bit-blasting and proved to be quite effective on hard satisfiable instances, particularly in the context of symbolic execution. However, it still relies on brute-force randomization and restarts to achieve completeness. Guided by a completeness proof, we simplified, extended and formalized our propagation-based variant of this approach. We obtained a clean, simple and more precise algorithm that does not rely on score-based local search techniques and does not require brute-force randomization or restarts to achieve completeness. It further yields substantial gain in performance. In this article, we present and discuss our complete propagation based local search approach for bit-vector logics in SMT in detail. We further provide an extended and extensive experimental evaluation including an analysis of randomization effects.

Keywords Satisfiability Modulo Theories · Local Search · Bit-Vector Reasoning

Supported by Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

Aina Niemetz
Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
E-mail: aina.niemetz@jku.at

Mathias Preiner
Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
E-mail: mathias.preiner@jku.at

Armin Biere
Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria
E-mail: biere@jku.at

1 Introduction

A majority of applications in the field of hardware and software verification requires bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) solvers for the quantifier-free theory of fixed-size bit-vectors. In many of these applications, e.g., (constrained random) test case generation [33, 38, 40] or white box fuzz testing [19], a majority of the problems is satisfiable. For this kind of problems, local search procedures are useful even though they do not allow to determine unsatisfiability. Previous work [16,36] showed that local search approaches for bit-vector logics in SMT are orthogonal to other approaches, which suggests that they are in particular beneficial within a portfolio setting [36].

Current state-of-the-art SMT solvers for the quantifier-free theory of fixed-size bit-vectors [4, 13, 14, 31, 34] employ the so-called bit-blasting approach (e.g., [28]), where an input formula is eagerly translated to propositional logic (SAT) and handed to an underlying SAT solver. While efficient in practice, bit-blasting approaches heavily rely on rewriting and other techniques [9,10,11,12,15,17,18,22,23] to simplify the input during preprocessing and may not scale if the input size can not be reduced sufficiently. In [16], Fröhlich et al. proposed to attack the problem from a different angle and presented a score-based local search approach for bit-vector logics, which lifts stochastic local search (SLS) from the bit-level (SAT) to the word-level (SMT) without bit-blasting. In previous years, and in particular since the SAT challenge 2012 [2], a new generation of SLS for SAT solvers with very simple architecture [3] achieved remarkable results not only in the random but also in the combinatorial tracks of recent SAT competitions [1, 2, 7]. Previous attempts to utilize SLS techniques in SMT by integrating an SLS SAT solver into the DPLL(T)-framework of the SMT solver MathSAT [13] were not able to compete with bit-blasting [21]. In contrast, the word-level local search approach in [16] showed promising initial results. However, [16] does not fully exploit the word-level structure but rather simulates bit-level local search by focusing on single bit flips.

Hence, in [36], we proposed a propagation-based extension of [16], which introduced an additional strategy to propagate assignments from the outputs to the inputs. This significantly improved performance. Our results further suggested that these techniques may be beneficial in a sequential portfolio setting [39] in combination with bit-blasting. However, down-propagating assignments as presented in [36] utilizes inverse value computation only, which can get stuck if no inverse value can be found. In that case, [36] still falls back on score-based local search techniques and requires brute-force randomization and restarts to achieve completeness, as does [16]. Further, inverse value computation as presented in [36] is too restrictive for some operators and focusing only on inverse values when down-propagating assignments is incomplete and may inadvertently prune the search.

In this paper, guided by a formal completeness proof we present a simple, precise and complete local search variant of the procedure proposed in [36]. Our approach does not use score-based local search techniques as described in [16] but relies on propagation of assignments only. It further does not require brute-force randomization or restarts to achieve completeness. To determine propagation paths, we extend the concept of controlling inputs to the word-level, which allows to further prune the search. To propagate assignments down, we lift the concept of "backtracing" of Automatic Test Pattern Generation (ATPG) [30], which goes back to the PODEM algorithm [20], to the word-level. We further provide a formalization of backtracing for the bit-level and the word-level. Note that in contrast to backtracing in ATPG, our algorithm works with complete assignments. Existing algorithms for word-level ATPG [25, 26] are based on branch-bound, use neither backtracing nor complete assignments, and in general lack formal treatment.

We implemented our techniques in our SMT solver Boolector [34] and show that combining our propagation-based approach with bit-blasting within a sequential portfolio setting is beneficial in terms of performance. We provide an extensive experimental evaluation, including an analysis of randomization effects as a result of different seeds for the random number generator, in particular in comparison to the score-based local search approach in [16] as implemented in Boolector. Our results show that our techniques yield a substantial gain in performance.

This article extends and revises work presented earlier in [35]. We provide a more detailed description of the propagation-based local search approach introduced in [35], including extensive examples illustrating the core concepts of our approach. We further include a complete set of rules for determining assignments during backtracing. Our previous experimental evaluation of a sequential portfolio combination of our propagation-based technique with bit-blasting was a virtual experiment. For this paper, we implemented such a sequential portfolio combination within Boolector and provide an extensive experimental evaluation of our techniques. This evaluation includes an in-depth analysis of the performance of our propagation-based local search approach compared to the score-based local search approach presented in [16] and the evaluation of randomization effects of both techniques, which were not included in previous work.

2 Overview

Our propagation-based local search procedure is based on propagating target assignments from the outputs to the inputs and does not need to rely on restarts or brute-force randomization to achieve completeness. Local search procedures are in general incomplete in the sense that they do not allow to determine unsatisfiability. Hence, in the following, we restrict our notion of completeness to satisfiable input problems and use it synonymously to the more established property of *probabilistically approximately complete* (PAC) [24], which is commonly used in the AI community to discuss completeness properties of local search algorithms. It follows the traditional notion of non-deterministic computation of Turing machines, which entails that we treat probabilistic choices as non-deterministic choices [24].

The basic idea of our approach is illustrated in Fig. 1 and described more precisely in pseudo code in Fig. 2. It is applied to propositional formulas (the bit-level) and quantifier-free bit-vector formulas (the word-level) as follows.

Given a formula ϕ, we assume without loss of generality that ϕ is a directed acyclic graph (DAG) with a single root r (the so-called root constraint or output of ϕ). We use the letter σ to refer to complete but non-satisfying assignments to all inputs and operators in ϕ. We further identify complete satisfying assignments with the letter ω. Starting from a random but non-satisfying initial assignment σ_1 with $\sigma_1(r) = 0$, our goal is to reach a satisfying assignment ω with $\omega(r) = 1$ by iteratively changing the values of primary inputs. We identify $\omega(r) = 1$ as the *target* value of output r (line 3), denoted as $0 \rightsquigarrow 1$ in Fig. 1, and propagate this value along a path towards the primary inputs (lines 4-7). We also refer to this

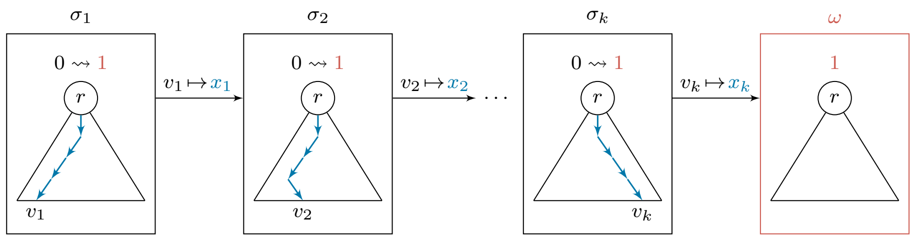


Fig. 1: Basic idea of our propagation-based local search approach. Starting from an unsatisfying assignment σ_1, we force root r to assume its target value $\omega(r) = 1$ and iteratively propagate this information towards the inputs until we find a solution ω.

```
 1  function sat ($r$, $\sigma$)
 2      while $\sigma(r) \neq 1$                  // while not satisfied
 3          $g := r$, $t := 1$                    // initialize path as root path
 4          while ¬leaf ($g$)                     // while current node is an operator
 5              $n :=$ child ($\sigma$, $t$, $g$)       // select backtracing node
 6              $x :=$ value ($\sigma$, $t$, $g$, $n$)  // select backtracing value
 7              $g := n$, $t := x$                // backtracing step (propagation)
 8          if ¬constant ($g$)                    // check if leaf is variable $v = g$
 9              $\sigma :=$ update ($\sigma$, $g$, $t$) // apply move to variable $v = g$
10      return true                               // return with *true* if satisfied
```

Fig. 2: The core sat procedure in pseudo-code.

process as "backtracing" [20]. Recursively propagating target value $\omega(r) = 1$ from the output to the primary inputs yields a new value $x_i \neq \sigma_i(v_i)$ for an input v_i (e.g., x_1 for v_1 in Fig. 1). By updating assignment σ_i on input v_i to $\sigma_{i+1}(v_i) = x_i$ (e.g., $\sigma_2(v_1) = x_1$ in Fig. 1) without changing the value of other primary inputs but recomputing consistent values for inner nodes (lines 8-9), we move from σ_i to σ_{i+1} and repeat this process until we reach a satisfying assignment, i.e., $\sigma_{i+1} = \omega$.

When down-propagating assignments, we identify path selection (line 5) and selecting the value to propagate (line 6) as the only sources of non-determinism. However, we aim to maximally reduce non-deterministic choices without sacrificing completeness. Hence, on the bit-level, path selection prioritizes controlling inputs w.r.t. the current assignment, a well-known concept from ATPG, while value selection for a selected input is uniquely defined. On the word-level, we introduce the corresponding new notion of essential inputs, which lifts the bit-level concept of controlling inputs to the word-level, and restrict value selection to the computation of what we refer to as consistent and inverse values.

As expected for local search, our propagation-based approach is not able to determine unsatisfiability. Thus the algorithm in Fig. 2 does not terminate in case that a given input formula is unsatisfiable. When determining satisfiability, however, our propagation-based local search approach is complete (PAC), i.e., there exists a non-deterministic choice of moves that leads to a solution.

In the following, we first introduce and formalize our propagation-based approach on the bit-level and prove its completeness. We then lift it to the word-level, and prove its completeness on the word-level. We further analyze randomization effects as result of using different seeds for the random number generator and show that our techniques yield substantial performance improvements, in particular in combination with bit-blasting within a sequential portfolio setting.

3 Bit-Level

For the sake of simplicity and without loss of generality we consider a fixed Boolean formula ϕ and restrict the set of Boolean operators to $\{\wedge, \neg\}$. We interpret ϕ as a single-rooted And-Inverter-Graph (AIG) [29], where an AIG is a DAG represented as a 5-tuple (r, N, G, V, E).

The set of *nodes* $N = G \cup V$ contains the single root node $r \in N$, and is further partitioned into a set of *gates* G and a set of *primary inputs* (or *variables*) V. We require that the set of variables is non-empty, i.e., $V \neq \emptyset$, and assume that the Boolean constants $\mathbb{B} = \{0, 1\}$, i.e., $\{\mathit{false}, \mathit{true}\}$, do not occur in N. This assumption is without loss of generality since every occurrence of *true* and *false* as input to a gate $g \in G$ can be eliminated through rewriting.

The set of *gates* $G = A \cup I$ consists of a set of *and*-gates A and a set of *inverter*-gates I. We write $g = n \wedge m$ if $g \in A$, and $g = \neg n$ if $g \in I$. We further refer to the children of a node $g \in G$ as its (gate) inputs (e.g., n or m). Let $E = E_A \cup E_I$ be the *edge* relation between nodes, with $E_A \colon A \to N^2$ and $E_I \colon I \to N$ describing edges from *and*- resp. *inverter*-gates to its input(s). We write $E(g) = (n, m)$ for $g = n \wedge m$ and $E(g) = n$ for $g = \neg n$ and further introduce the notation $g \to n$ for an edge between a gate g and one of its inputs n.

We define a *complete assignment* σ of the given fixed formula ϕ as a *complete* function $\sigma \colon N \to \mathbb{B}$. Similarly, a *partial assignment* α of formula ϕ is defined as a *partial* function $\alpha \colon N \to \mathbb{B}$. We say that a complete assignment σ is *consistent* on a gate $g \in I$ with $g = \neg n$ iff $\sigma(g) = \neg\sigma(n)$, and *consistent* on a gate $g \in A$ with $g = n \wedge m$ iff $\sigma(g) = \sigma(n) \wedge \sigma(m)$.

A complete assignment σ is *globally consistent* on ϕ (or just *consistent*) iff it is consistent on all gates $g \in G$. An assignment σ is *satisfying* if it is consistent (thus complete) and satisfies the root, i.e., $\sigma(r) = 1$. We use the letter ω to denote a satisfying assignment. A formula ϕ is satisfiable if it has a satisfying assignment. We use $\mathcal{C}$ to denote the set of *consistent* assignments, and $\mathcal{W}$ with $\mathcal{W} \subseteq \mathcal{C}$ to denote the set of *satisfying* assignments of formula ϕ.

Given two consistent assignments σ and σ', we say that σ' is obtained from σ by *flipping* the (assignment of a) variable $v \in V$, written as $\sigma \xrightarrow{v} \sigma'$, iff $\sigma(v) = \neg\sigma'(v)$ and $\sigma(u) = \sigma'(u)$ for all $u \in V \backslash \{v\}$. We also refer to flipping a variable as a *move.* Note that $\sigma'(g)$ for gates $g \in G$ is defined implicitly due to consistency of assignment σ' after fixing the values for the primary inputs V.

Given a set of variables V that can be flipped non-deterministically, let $S \colon \mathcal{C} \to \mathbb{P}(\mathcal{M})$ be a (local search) *strategy* that maps a consistent assignment to a set of possible moves $\mathcal{M} = V$. Note that in general, there exist different notions of strategy, e.g., as in the context of game theory or synthesis. In the context of local search, using the term "strategy" as defined above is well established, e.g., [24,37]. Further note that since a move corresponds to flipping a variable, the set of possible moves $\mathcal{M}$ corresponds to the set of variables V and is redundant on the bit-level. However, we use the same notation on the word-level where $\mathcal{M}$ captures the set of moves valid under a strategy as a set of input value pairs, since a word-level move requires to additionally identify the new value of an input.

A move $v \in V$ is *valid* under strategy S for assignment $\sigma \in \mathcal{C}$ if $v \in S(\sigma)$. Similarly, a sequence of moves $\mu = (v_1, \ldots, v_k) \in V^*$ of length $k = |\mu|$ with $v_1, \ldots, v_k \in V$ is *valid* under strategy S, iff there exists a sequence of consistent assignments $(\sigma_1, \ldots, \sigma_{k+1}) \in \mathcal{C}^*$ such that $\sigma_i \xrightarrow{v_i} \sigma_{i+1}$ and $v_i \in S(\sigma_i)$ for $1 \leq i \leq k$. In this case, assignment σ_{k+1} can be *reached* from assignment σ_1 under strategy S (with k moves), also written as $\sigma_1 \rightarrow^* \sigma_{k+1}$.

Definition 1 (Complete Strategy) If formula ϕ is satisfiable, then a strategy S is called *complete* iff for all consistent assignments $\sigma \in \mathcal{C}$ there exists a satisfying assignment $\omega \in \mathcal{W}$ such that ω can be reached from σ under S, i.e., $\sigma \rightarrow^* \omega$.

Given an assignment $\sigma \in \mathcal{C}$ and a satisfiable assignment $\omega \in \mathcal{W}$, let $\Delta(\sigma, \omega) = \{v \in V \mid \sigma(v) \neq \omega(v)\}$ be the set of variables with different values in σ and ω. Thus, $\mathrm{HD}(\sigma, \omega) = |\Delta(\sigma, \omega)|$ is the *Hamming Distance* between σ and ω on V.

Definition 2 (Distance-Reducing Strategy) A strategy S is (non-deterministically) *distance reducing*, if for all assignments $\sigma \in \mathcal{C} \backslash \mathcal{W}$ there exists a satisfying assignment $\omega \in \mathcal{W}$ and a move $\sigma \xrightarrow{v} \sigma'$ valid under S which reduces the Hamming Distance. That is, move $v \in \mathcal{M}$ is in $\Delta(\sigma, \omega)$, thus $\mathrm{HD}(\sigma, \omega) - \mathrm{HD}(\sigma', \omega) = 1$.

Obviously, any distance reducing strategy can reach a satisfying assignment (though not necessarily ω) within at most $\mathrm{HD}(\sigma, \omega)$ moves. This first observation is the key argument in the completeness proofs for our propagation based strategies (both on the bit-level and word-level).

Proposition 3 *A distance reducing strategy is also complete.*

In the following, our ultimate goal is to define a strategy that maximally reduces non-deterministic choices without sacrificing completeness. In the algorithm shown in Fig. 2, path selection (selecting the backtracing node in line 5) and value selection (selecting the backtracing value in line 6) while down-propagating assignments constitute the only sources of non-determinism. As we will show later, in contrast to value selection on the word-level, selecting a backtracing value on the bit-level is uniquely defined. When selecting a backtracing node on the bit-level, non-determinism can be reduced by utilizing the notion of *controlling inputs* from ATPG [30], which is defined as follows.

Definition 4 (Controlling Input) Let node $n \in N$ be an input of a gate $g \in G$, i.e., $g \rightarrow n$, and let σ be a complete assignment consistent on g. We say that input n is *controlling* under σ if for all complete assignments σ' consistent on g with $\sigma(n) = \sigma'(n)$ we have $\sigma(g) = \sigma'(g)$.

In other words, gate input n is *controlling* if the assignment of g (i.e., its output value) remains unchanged as long as the assignment of n does not change.

Given an assignment σ consistent on $g \in G$, we denote a *target value* t for gate g as $\sigma(g) \rightsquigarrow t$. On the bit-level, target value t is *implicitly* given through assignment σ as $t = \neg\sigma(g)$, i.e., t can not be reached as long as the controlling inputs of g remain unchanged. On the word-level, t may be any value $t \neq \sigma(g)$.

Example 5 Figure 3 shows all possible assignments σ consistent on a gate $g \in G$. At the outputs we denote current assignment $\sigma(g)$ and target value t as $\sigma(g) \rightsquigarrow t$ with $t = \neg\sigma(g)$, e.g., $0 \rightsquigarrow 1$. At the inputs we show their assignment under σ. All controlling inputs are indicated with an underline. Note that *and*-gate $g = n \wedge m$ has *no* controlling inputs for $\sigma(g) = 1$.

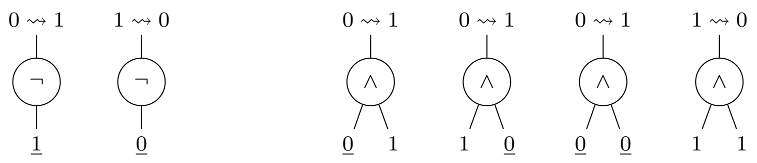


Fig. 3: An *inverter*-gate and an *and*-gate and their controlling (underlined) inputs. Given output values indicate the transition from current to target value.

We define a sequence of nodes $\pi = (n_1, \ldots, n_k) \in N^+$ as a *path* of length k with $k = |\pi|$ iff $n_i \rightarrow n_{i+1}$ for $0 < i < k$, also written as $n_1 \rightarrow \ldots \rightarrow n_k$. A path π is *rooted* if $n_1 = r$, and (fully) *expanded* if $n_k \in V$. We refer to $n_k \in V$ as the *leaf* of π in this case. As a restriction on ϕ, we require all nodes $n \in N$ to be reachable from the root r, i.e., there exists a path π such that $\pi = (r, \ldots, n)$. We further require all paths to be acyclic, i.e., for all $n \in N$ there exists no path $n \rightarrow^+ n$. Note that as a consequence of representing ϕ as a DAG, this is the case for any path in ϕ. Given a path $\pi = (\ldots, g)$ with gate $g \in G$ and $g \rightarrow n$, we say that $\pi.n = (\ldots, g, n)$ is an *extension* of path π with node n.

Definition 6 (Path Selection) Given a complete assignment $\sigma \in \mathcal{C}$ and a path $\pi = (\ldots, g)$ as above. *Gate input n can be selected* w.r.t. assignment σ to extend path π to $\pi.n$ if input n is controlling or if gate g has no controlling input.

Path selection based on the notion of controlling inputs as introduced above exploits observability don't cares as defined in the context of ATPG [30]. Similarly, we adopt the ATPG idea of backtracing [20, 30] as follows.

Definition 7 (Backtracing Step) Given a complete assignment $\sigma \in \mathcal{C}$ and a gate $g \in G$ with $g \rightarrow n$. A *backtracing step* w.r.t. assignment σ selects gate input n w.r.t. assignment σ as in Def. 6 and determines a *backtracing value* x for input n as follows. If $g = \neg n$, then $x = \sigma(g)$. Else, if $g = n \wedge m$, then $x = \neg\sigma(g)$.

As an important observation it turns out that performing a bit-level backtracing step always flips the value of the selected input under σ. For a selected input, the backtracing value is therefore always unique. This can be checked easily by considering all possible scenarios shown in Fig. 3.

Proposition 8 *A backtracing step yields as backtracing value $x = \neg\sigma(n)$.*

Example 9 Consider $g = n \wedge m$ and the assignment $\sigma = \{g \mapsto 0, n \mapsto 0,\ m \mapsto 1\}$ consistent on g as depicted in Fig. 3. Assume that $t = \neg\sigma(g) = 1$ is the target value of g, i.e., $\sigma(g) \rightsquigarrow t$ with $0 \rightsquigarrow 1$. We select n as the single controlling input of g (underlined), which yields backtracing value $x = \neg\sigma(n) = 1$.

A *trace* $\tau = (\pi, \alpha)$ is a rooted path $\pi = (n_1, \ldots, n_k)$ labelled with a partial assignment α, where α is defined exactly on $\{n_1, \ldots, n_k\}$. A trace (π, α) is (fully) *expanded*, if π is a fully expanded path, i.e., node $n_k \in V$ is the *leaf* of π and τ.

Let $\sigma \in \mathcal{C}\backslash\mathcal{W}$ be a complete consistent but non-satisfying assignment. Then the notion of extension is lifted from paths to traces as follows. Given a trace $\tau = (\pi, \alpha)$ with $\pi = (\ldots, g)$, $g \in G$ and $g \to n$. A backtracing (or *propagation*) step w.r.t. σ and target value $t = \neg\sigma(g) = \alpha(g)$ yields backtracing value $x = \neg\sigma(n) = \alpha'(n)$ and *extends* trace τ to $\tau' = (\pi', \alpha')$ (also denoted as $\tau \to \tau'$) if path $\pi' = \pi.n$ is an extension of π and $\alpha'(m) = \alpha(m)$ for all nodes m in π.

We define the *root trace* $\rho = ((r), \{r \mapsto 1\})$ as a trace that maps root r to its target value $\omega(r) = 1$. A *propagation trace* w.r.t. assignment σ is a (possibly partial) trace τ that starts from the root trace ρ and is extended by propagation steps w.r.t. assignment σ, denoted as $\rho \to^* \tau$.

Note that given a path π and σ, partial assignment α is redundant on the bit-level. However, we use the same notation on the word-level, where α captures updates to σ along π, which (in contrast to the bit-level) are not uniquely defined.

Definition 10 (Propagation Strategy) Given a non-satisfying but consistent assignment $\sigma \in \mathcal{C}\backslash\mathcal{W}$, the set of valid moves $\mathcal{P}(\sigma)$ for σ under propagation strategy $\mathcal{P}$ contains exactly the leafs of all expanded propagation traces w.r.t σ.

In the following, we present and prove the main Lemma of this paper, which then immediately gives completeness of strategy $\mathcal{P}$ in Theorem 12. It is reused for proving completeness of the extension of $\mathcal{P}$ to the word-level in Theorem 25.

Lemma 11 (Propagation Lemma) *Given a non-satisfying but consistent assignment $\sigma \in \mathcal{C}\backslash\mathcal{W}$, then for any satisfying assignment $\omega \in \mathcal{W}$, used as an oracle, there exists a fully expanded propagation trace τ w.r.t. σ with leaf $v \in \Delta(\sigma, \omega)$.*

Proof The basic idea of our completeness proof is to inductively extend the root trace ρ to traces $\tau = (\pi, \alpha)$, i.e., $\rho \to^* \tau$, through propagation steps, which all satisfy the (key) invariant

$$\alpha(n) = \omega(n) \neq \sigma(n) \quad \text{for all nodes } n \text{ in } \pi. \tag{1}$$

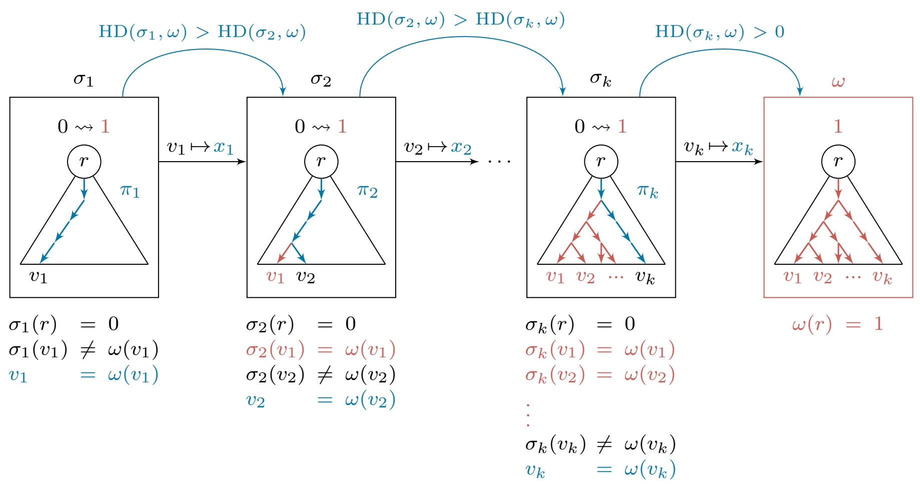
The root trace $\rho = ((r), \{r \mapsto \omega(r)\})$ obviously satisfies this invariant. Now, let $\tau = (\pi, \alpha)$ be a trace that satisfies the invariant but is *not fully expanded*, i.e., $\pi = (r, \ldots, g)$ with $g \in G$ and $\alpha(g) = \omega(g) \neq \sigma(g)$. Since $\sigma(g) \neq \omega(g)$ it follows that g has at least one input n with $\sigma(n) \neq \omega(n)$. If g has no controlling input, then by Def. 6 it is allowed to select n as an input with $\sigma(n) \neq \omega(n)$. Otherwise, input n is selected as any controlling input. In both cases we select $x = \omega(n) \neq \sigma(n)$ as backtracing value using Prop. 8. Trace τ is now extended by n with backtracing value x to τ', i.e., $\tau \to \tau'$, which in turn concludes the inductive proof of Equation (1). Any fully expanded propagation trace $\tau = (\pi, \alpha)$ with leaf $v \in V$, as generated above, also satisfies the invariant in Equation (1). Thus, we have $\alpha(v) = \omega(v) \neq \sigma(v)$ with $v \in \Delta(\sigma, \omega)$. $\square$

In essence, given assignment σ and ω as above, our propagation strategy propagates target value $\omega(r)$ from root r towards the primary inputs, ultimately producing a fully expanded propagation trace $\tau = (\pi, \alpha)$. In case of non-deterministic choices for extending the trace we use ω as an oracle to pick an input n with $\sigma(n) \neq \omega(n)$, which can be selected according to Def. 6. The oracle allows us to ensure that for all nodes $n \in \pi$, $\alpha(n) = \omega(n)$, which yields $\alpha(v) = \omega(v) \neq \sigma(v)$ and consequently $v \in \Delta(\sigma, \omega)$ for leaf v of τ. Thus, using Lemma 11, our propagation strategy turns out to be distance reducing, and therefore, according to Prop. 3, complete. Figure 4 illustrates the basic idea of our proof, which, in the following, serves as a basis for lifting our approach from the bit-level to the word-level.

Theorem 12 *Under the assumptions of the previous Lemma 11 we also get $v \in \mathcal{P}(\sigma)$ for leaf v. Thus, $\mathcal{P}$ is distance reducing and, as a consequence, complete.*

4 Word-Level

In the following, we only consider bit-vector expressions of fixed bit-width $w \in \mathbb{N}$. We denote a bit-vector expression n of width w as $n_{[w]}$, but will omit the bit-width if the context allows. We refer to the i-th bit of $n_{[w]}$ as $n[i]$ with $1 \leq i \leq w$ and, for the sake of simplicity, define bit indices as starting from 1 rather than 0. We interpret $n[1]$ as the least significant bit (LSB) and $n[w]$ as the most significant bit (MSB), and denote bit ranges over n from bit index j down to index i as $n[j:i]$. In string representations of bit-vectors, we interpret the bit at the far left index as the MSB and the bit at the far right index as the LSB. We further define *ctz* to be the common function that *counts the number of trailing zeroes* of a given bit-vector, i.e., the number of consecutive 0-bits starting from the LSB, e.g., $ctz(0101) = 0$ and $ctz(111100) = 2$. Similarly, *clz* is the common function to *compute the number of leading zeroes*, i.e., the number of consecutive 0-bits starting from the MSB, e.g., $clz(0101) = 1$ and $clz(111100) = 0$.



$\mathrm{HD}(\sigma_1,\omega) > \mathrm{HD}(\sigma_2,\omega)$
$\mathrm{HD}(\sigma_2,\omega) > \mathrm{HD}(\sigma_k,\omega)$
$\mathrm{HD}(\sigma_k,\omega) > 0$
σ_1
σ_2
σ_k
ω
$0 \rightsquigarrow 1$
$0 \rightsquigarrow 1$
$0 \rightsquigarrow 1$
1
r
π_1
π_2
π_k
$v_1 \mapsto x_1$
$v_2 \mapsto x_2$
$\dots$
$v_k \mapsto x_k$
v_1
$v_1 \; v_2$
$v_1 \; v_2 \; \cdots \; v_k$
$v_1 \; v_2 \; \cdots \; v_k$
$\sigma_1(r) = 0$
$\sigma_1(v_1) \neq \omega(v_1)$
$v_1 = \omega(v_1)$
$\sigma_2(r) = 0$
$\sigma_2(v_1) = \omega(v_1)$
$\sigma_2(v_2) \neq \omega(v_2)$
$v_2 = \omega(v_2)$
$\sigma_k(r) = 0$
$\sigma_k(v_1) = \omega(v_1)$
$\sigma_k(v_2) = \omega(v_2)$
$\vdots$
$\sigma_k(v_k) \neq \omega(v_k)$
$v_k = \omega(v_k)$
$\omega(r) = 1$

Fig. 4: The basic idea of our completeness proof. Using a satisfying assignment ω as an oracle, in each move $\sigma_i \to \sigma_{i+1}$ we down-propagate target value $x = \omega(n)$ for all nodes n in propagation path $\pi_i = (r, \dots, v_i)$, which yields update $x_i = \omega(v_i) = \sigma_{i+1}(v_i)$ and thus reduces the hamming distance $\mathrm{HD}(\sigma_i, \omega) > \mathrm{HD}(\sigma_{i+1}, \omega)$.

Operator	SMT-LIB	Arity	Bit-Width Output	Input			
$[j:i]$	`extract`	1	$j-i+1$	w	$-$	$-$	Extraction ($1 \leq i \leq j \leq w$)
$\sim$	`bvnot`	1	w	w	$-$	$-$	Bit-wise negation
$\&$	`bvand`	2	w	w	w	$-$	Bit-wise conjunction
$=$	`=`	2	1	w	w	$-$	Equality
$<$	`bvult`	2	1	w	w	$-$	Unsigned less than
$\ll$	`bvshl`	2	w	w	q	$-$	Logical shift left ($w = 2^q$)
$\gg$	`bvshr`	2	w	w	q	$-$	Logical shift right ($w = 2^q$)
$+$	`bvadd`	2	w	w	w	$-$	Addition
$\cdot$	`bvmul`	2	w	w	w	$-$	Multiplication
$\div$	`bvudiv`	2	w	w	w	$-$	Unsigned division
mod	`bvurem`	2	w	w	w	$-$	Unsigned remainder
$\circ$	`concat`	2	$p+q$	p	q	$-$	Concatenation
if-then-else	`ite`	3	w	1	w	w	Conditional

Table 1: The set of considered bit-vector operators ($w, p, q, i, j \in \mathbb{N}$).

For the sake of simplicity and without loss of generality we consider a fixed single-rooted quantifier-free bit-vector formula ϕ and interpret Boolean expressions as bit-vector expressions of bit-width one. The set of bit-vector operators is restricted to $\mathcal{O} = \{\&, \sim, =, <, \ll, \gg, +, \cdot, \div, \text{mod}, \circ, [:], \text{if-then-else}\}$ and interpreted according to Tab. 1. The selection of operators in $\mathcal{O}$ is rather arbitrary but provides a good compromise between effective and efficient word-level rewriting and compact encodings for bit-blasting approaches. It is complete, though, in the sense that all operators defined in SMT-LIB [5] (in particular signed operators) can be modeled in a compact way. Note that our methods are not restricted to single-rootedness or this particular selection of operators, and can easily be lifted to any other set of operators or the multi-rooted case.

We interpret formula ϕ as a single-rooted DAG represented as an 8-tuple $(r, N, \kappa, O, F, V, B, E)$. The set of *nodes* $N = O \cup V \cup B$ contains the single root node $r \in N$ of bit-width one, and is further partitioned into a set of *operator nodes* O, a set of *primary inputs* (or *bit-vector variables*) V, and a set of *bit-vector constants* $B \subseteq \mathbb{B}^*$, which are denoted in either decimal or binary notation if the context allows. The bit-width of a node is given by $\kappa\colon N \to \mathbb{N}$, thus $\kappa(r) = 1$. Operator nodes are interpreted as bit-vector operators via $F : O \to \mathcal{O}$, which in turn determines their arity and input and output bit-widths as defined in Tab. 1. The *edge* relation between nodes is given as $E = E_1 \cup E_2 \cup E_3$, with $E_i\colon O \to N^i$ describing the set of edges from unary, binary, and ternary operator nodes to its input(s), respectively. We again use the notation $o \to n$ for an edge between an operator node o and one of its inputs n.

We only consider well-formed formulas, where the bit-widths of all operator nodes and their inputs conform to the conditions imposed via interpretation F as defined in Tab. 1. For instance, we denote a bit-vector addition node o with inputs n and m as $o = n + m$, where $o \in O$ of arity 2 with $F(o) = +$, and therefore $\kappa(o) = \kappa(n) = \kappa(m)$. In the following, if more convenient we will use the functional notation $o = \diamond(n_1, \ldots, n_k)$ for operator node $o \in O$ of arity k with inputs $n_1, \ldots, n_k$ and $F(o) = \diamond$, e.g., $+(n, m)$. Note that the semantics of all operators in $\mathcal{O}$ correspond to their SMT-LIB counterparts listed in Tab. 1, with three exceptions. Given a logical shift operation $n \ll m$ or $n \gg m$, w.l.o.g. and as implemented in our SMT solver Boolector [34], we restrict bit-width $\kappa(n)$ to $2^{\kappa(m)}$. Further, as implemented by Boolector and other state-of-the-art SMT solvers, e.g., MathSAT [13] Yices [14] and Z3 [31], we define an unsigned division by zero to return the greatest possible value rather than introducing uninterpreted functions, i.e., $x \div 0 = \sim 0$. Similarly, $x \bmod 0 = x$.

A *complete assignment* σ of a given fixed ϕ is a complete function $\sigma\colon N \rightarrow \mathbb{B}^*$ with $\sigma(n) \in \mathbb{B}^{\kappa(n)}$, and a *partial assignment* is a partial function $\alpha\colon N \rightarrow \mathbb{B}^*$ with $\alpha(n) \in \mathbb{B}^{\kappa(n)}$. Given an operator node $o \in O$ with $o = \diamond(n_1, \ldots, n_k)$ and $\diamond \in \mathcal{O}$, a complete assignment σ is *consistent* on o if $\sigma(o) = f(\sigma(n_1), \ldots, \sigma(n_k))$ where $f\colon \mathbb{B}^{\kappa(n_1)} \times \cdots \times \mathbb{B}^{\kappa(n_k)} \rightarrow \mathbb{B}^{\kappa(o)}$ is determined by the semantics of operator $\diamond$ as defined in the SMT-LIB standard [5] (with the exceptions discussed above).

A complete assignment is (globally) *consistent* on ϕ (or just *consistent*), iff it is consistent on all bit-vector operator nodes $o \in O$ and $\sigma(b) = b$ for all bit-vector constants $b \in B$. A *satisfying* assignment ω is a complete and consistent assignment that satisfies the root, i.e., $\omega(r) = 1$. In the following, we will again use the letter $\mathcal{C}$ to denote the set of complete and consistent assignments, and the letter $\mathcal{W}$ with $\mathcal{W} \subseteq \mathcal{C}$ to denote the set of satisfying assignments of formula ϕ.

Given a bit-vector variable $v \in V$ with $\kappa(v) = w$ and assignments $\sigma, \sigma' \in \mathcal{C}$. We adopt the notion of obtaining assignment σ' from assignment σ by assigning a new value x to variable v with $x \in \mathbb{B}^w$ and $x \neq \sigma(v)$, written as $\sigma \xrightarrow{v \mapsto x} \sigma'$, which we refer to as a *move*. The set of word-level moves is thus defined as $\mathcal{M} = \{(v, x) \mid v \in V, x \in \mathbb{B}^{\kappa(v)}\}$, and accordingly, a word-level propagation strategy $\mathcal{P}$ is defined as a function $S\colon \mathcal{C} \mapsto \mathbb{P}(\mathcal{M})$, which maps a consistent assignment to a set of moves. We lift propagation strategy $\mathcal{P}$ from the bit-level to the word-level by first introducing our new notion of *essential inputs*, which lifts and extends the bit-level notion of controlling inputs to the word-level.

Definition 13 (Essential Inputs) Let $n \in N$ be an input of a bit-vector operator node $o \in O$, i.e., $o \rightarrow n$, and let σ be a complete assignment consistent on o. Further, let t be the *target value* of o, i.e., $\sigma(o) \rightsquigarrow t$, with $t \neq \sigma(o)$. We say that n is an *essential input* under σ w.r.t. target value t, if for all complete assignments σ' consistent on o with $\sigma(n) = \sigma'(n)$, we have $\sigma'(o) \neq t$.

In other words, an input n to an operator node o is *essential* w.r.t. some target value t, if o can not assume t as long as the assignment of n does not change. As an example, consider the bit-vector operators and their essential inputs under some consistent assignment σ w.r.t. some target value t as depicted in Fig. 5.

Example 14 Consider the bit-vector operators $\{+,\ \&,\ \ll,\ \cdot,\ \div,\ \mathrm{mod},\ \circ\}$ of bit-width 2 as depicted in Fig. 5. For an operator node o, at the outputs we denote given assignment $\sigma(o)$ and target value t as $\sigma(o) \rightsquigarrow t$ (e.g., $10 \rightsquigarrow 01$). At the inputs we show their assignment under σ. Essential inputs (under σ w.r.t. target value t) are indicated with an underline.

(a) Given $\mathbf{o} := \mathbf{n} + \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 11,\ n \mapsto 00,\ m \mapsto 11\}$.
Operator $+$ has no essential inputs, independent from σ and t.

(b) Given $\mathbf{o} := \mathbf{n}\ \&\ \mathbf{m}$ with $t = 01$ and $\sigma = \{o \mapsto 10,\ n \mapsto 10,\ m \mapsto 11\}$.
Input n is essential since $t\ \&\ \sigma(n) \neq t$ and thus, it is not possible to find a

value x for m such that $\sigma(n) \mathbin{\&} x = t$. Input m, however, is not essential since $t \mathbin{\&} \sigma(m) = t$.

(c) Given $\mathbf{o} := \mathbf{n} \ll \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 00,\ n \mapsto 00,\ m \mapsto 1\}$.
Input n is obviously essential, since shifting 00 can never result in the non-zero target value $t = 01$. Input m, however, is not essential, since it is possible to simply select, e.g., $x = 01$ for n such that $t = 10 = x \ll \sigma(m) = 01 \ll 1$.

(d) Given $\mathbf{o} := \mathbf{n} \cdot \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 00,\ n \mapsto 00,\ m \mapsto 10\}$.
Input n is essential since $t \neq 00$ but $\sigma(n) = 00$, and thus, it is not possible to find a value x for m such that $\sigma(n) \cdot x = t$. Input m, however, is not essential since we could pick, e.g., $x = 01$ for n to obtain $t = 10 = x \cdot \sigma(m) = 01 \cdot 10$.

(e) Given $\mathbf{o} := \mathbf{n} \div \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 01,\ n \mapsto 01,\ m \mapsto 01\}$.
Input n is essential since $\sigma(n) < t$, and thus, it is not possible to find a value x for m such that $\sigma(n) \div x = t$. Input m, however, is not essential, since we could pick, e.g., $x = 10$ to obtain $t = 10 = x \cdot \sigma(m) = 10 \cdot 01$.

(f) Given $\mathbf{o} := \mathbf{n} \bmod \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 00,\ n \mapsto 01,\ m \mapsto 01\}$.
Since $\sigma(n) = 01 < 10 = t$, it is not possible to find a value x for m such that $\sigma(n) \bmod x = t$. However, since $\sigma(m) = 01$ but $t \neq 00$, it is also not possible to find a value x for n such that $x \bmod \sigma(m) = t$. Hence, both inputs are essential.

(g) Given $\mathbf{o} := \mathbf{n} \circ \mathbf{m}$ with $t = 11$ and $\sigma = \{o \mapsto 01,\ n \mapsto 0,\ m \mapsto 1\}$.
Input n is essential since $\sigma(n) \neq t[2:2]$, and thus, it is not possible to find a value x for m such that $\sigma(n) \circ x = t$. Input m, however, is not essential since it already matches the corresponding slice of the target value.

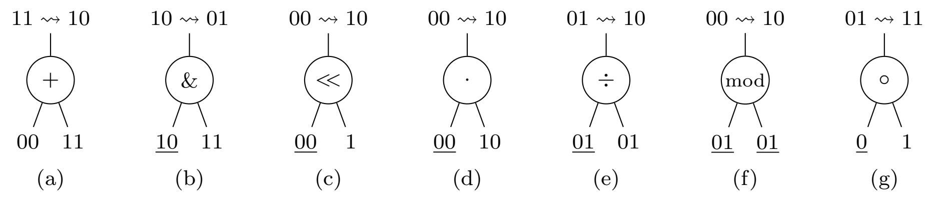


Fig. 5: Bit-vector operator nodes and examples for essential (underlined) inputs. Given output values indicate the transition from current to target value.

Note that bit-level expressions (AIGs) can be represented by bit-vectors of bit-width one, which can be interpreted as word-level Boolean expressions. In this sense, the notion of controlling inputs can also be applied to Boolean expressions on the word-level.

Proposition 15 *When applied to bit-level expressions, the notion of* essential inputs *exactly matches the notion of* controlling inputs*.*

Proof For applying the notion of essential inputs to bit-level expressions, consider the operator set $O = \{\neg, \wedge\} = G$ and operator $o \in G$ with $o \to n$. Target value $t \neq \sigma(o)$ as in Def. 13 implies $t = \neg\sigma(o)$ for operator o. This exactly matches the implicit definition of the target value of a Boolean operator on the bit-level. Now assume that input n is essential w.r.t. target value t. Then, if $\sigma(n) = \sigma'(n)$, by Def. 13 we have that $\sigma'(o) \neq t$, and therefore $\sigma'(o) = \neg t = \sigma(o)$, which exactly matches the notion of controlling inputs as in Def. 4. The other direction (applying the notion of controlling inputs to word-level Boolean expressions exactly matches the notion of essential inputs) works in the same way. $\square$

The definition of a (rooted and expanded) *path* as a sequence of nodes $\pi = (n_1, \ldots, n_k) \in N^*$ is lifted from the bit-level to the word-level in the natural way. Corresponding restrictions and implications of Section 3 apply. The notions of *path selection* and *path extension* are lifted to the word-level as follows.

Definition 16 (Path Extension) Given a path $\pi = (\ldots, o)$ with $o \in O$ and $o \rightarrow n$, we say that $\pi.n = (\ldots, o, n)$ is an extension of path π with node n.

Definition 17 (Path Selection) Given a complete consistent assignment $\sigma \in \mathcal{C}$, a path $\pi = (\ldots, o)$ as in Def. 16 above, and $\sigma(o) \rightsquigarrow t$, i.e., $t \neq \sigma(o)$, then *input n can be selected* w.r.t. σ and target value t to extend π to $\pi.n$ if n is essential or if o has no essential input (in both cases essential under σ w.r.t. t).

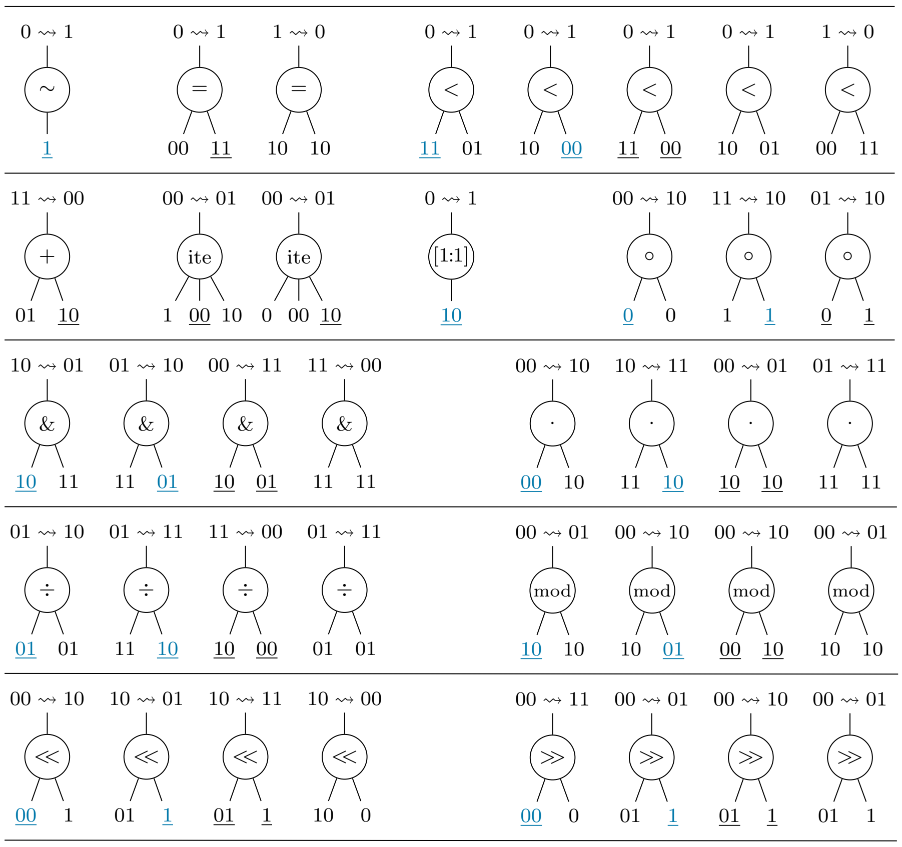


Fig. 6: Examples for all combinations of essential (underlined) inputs for all bit-vector operators in $\mathcal{O}$. Underlined blue cases indicate that this input is a single essential input. No input is essential for operators $=$, $+$, and if-then-else.

Figure 6 shows examples for all combinations of essential (underlined) and non-essential inputs for all bit-vector operators in $\mathcal{O}$. For an operator node o, an output value $\sigma(o) \rightsquigarrow t$ indicates the desired transition from current assignment $\sigma(o)$ to target value t, and an input value shows its assignment under σ. Underlined blue cases indicate that this input is a single essential input and will therefore always be selected. Any other case (both inputs are essential or no input is essential) represents a non-deterministic choice during path selection.

In contrast to value selection on the bit-level, where a backtracing step always yields the flipped assignment of the selected input as backtracing value, on the word-level, selecting a backtracing value is not uniquely defined but a source of non-determinism. We consider three variants of value selection, under the following assumptions. Let t be the target value of an operator node $o \in O$, and let $\sigma \in \mathcal{C}$ be a complete assignment such that $\sigma(o) \neq t$. Further, assume that input n with $o \to n$ is selected w.r.t. target value t and σ as in Def. 17 above.

Definition 18 (Random Value) Any value x with $\kappa(x) = \kappa(n)$ is called a *random value* for input n.

Definition 19 (Consistent Value) A random value x is a *consistent value* for input n w.r.t. target value t, if there is a complete assignment σ' consistent on operator node o with $\sigma'(n) = x$ and $\sigma'(o) = t$.

In other words, a value is consistent for an input, if it allows to produce the target value after changing values of other inputs if necessary. We compute a consistent value as backtracing value x for input n as described in Tab. 2.

However, in some cases, restricting the notion of consistent values even further may be beneficial. Consider the following motivating example.

Example 20 Consider a formula $\phi := 274177_{[65]} \cdot v = 18446744073709551617_{[65]}$. Computing $x = 18446744073709551617_{[65]} \div 274177_{[65]} = 67280421310721_{[65]}$ immediately concludes with a satisfying assignment for ϕ.

The chances to select $x = 67280421310721_{[65]}$ if consistent values for the multiplication operator are chosen as described in Tab. 2 are arbitrarily small. Hence, we also consider the notion of *inverse values*, which utilize the inverse of an operator.

Definition 21 (Inverse Value) A consistent value x is an *inverse value* for input n w.r.t. target value t and assignment σ, if there exists a complete assignment σ' consistent on operator node o with $\sigma'(n) = x$, $\sigma'(o) = t$ and $\sigma'(m) = \sigma(m)$ for all inputs m with $o \to m$ and $m \neq n$.

In other words, a value is an inverse value for input n, if it allows to produce the target value for an operator node without changing the assignment of its other inputs. Consequently, an inverse value for input n is also consistent. We compute an inverse value as backtracing value x for input n as described in Tab. 3-4.

Note that inverse value computation as initially presented in [36] is too restrictive for some operators, which is incomplete since it may inadvertently prune the search. We therefore require that inverse value computation allows to generate all possible values for all operators in $\mathcal{O}$, which is the case for the rules for inverse value computation as described in Tab. 3-4.

$\mathbf{o} \coloneqq \sim\mathbf{n}$	Then $x = {\sim}t$.
$\mathbf{o} \coloneqq \mathbf{n} = \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} = \mathbf{n}$	Any value x is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \mathbin{\&} \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \mathbin{\&} \mathbf{n}$	Let i be a bit index with $1 \leq i \leq \kappa(n)$. For all i, if $t[i] = 1$ then $x[i] = 1$, and else, $x[i]$ is set arbitrarily.
$\mathbf{o} \coloneqq \mathbf{n} < \mathbf{m}$	Any value x with $x < {\sim}0$ if $t = 1$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{m} < \mathbf{n}$	Any value x with $x \neq 0$ if $t = 1$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} + \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} + \mathbf{n}$	Any value x is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \cdot \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \cdot \mathbf{n}$	Any x with $\mathit{ctz}(t) \geq \mathit{ctz}(x)$ and $x = 0$ if $t = 0$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \div \mathbf{m}$	If $t = {\sim}0$ or $t = 0$, any value x with $x < {\sim}0$ if $t = 0$ is a consistent value for n. In any other case, let y be a random value with $y \neq 0$ such that $y \cdot t$ does not overflow. Then $x = y \cdot t$.
$\mathbf{o} \coloneqq \mathbf{m} \div \mathbf{n}$	If $t = {\sim}0$, then $x \in \{0, 1\}$ is a consistent value for n. Else, any value x such that $x \cdot t$ does not overflow is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \bmod \mathbf{m}$	If $t = {\sim}0$ then $x = {\sim}0$, and a random $x \geq t$, otherwise.
$\mathbf{o} \coloneqq \mathbf{m} \bmod \mathbf{n}$	If $t = {\sim}0$ then $x = 0$, and a random $x > t$, otherwise.
$\mathbf{o} \coloneqq \mathbf{n} \ll \mathbf{m}$	Let s be a random value with $0 \leq s \leq \mathit{ctz}(t)$, and let $w = \kappa(n)$. Then $x[i] = (t \gg s)[i]$ for $1 \leq i \leq w - s$, and all other bits $x[i]$ set arbitrarily for $w - s < i \leq w$.
$\mathbf{o} \coloneqq \mathbf{m} \ll \mathbf{n}$	Any x with $0 \leq x \leq \mathit{ctz}(t)$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \ll \mathbf{m}$	Let s be a random value with $0 \leq s \leq \mathit{ctz}(t)$, and let $w = \kappa(n)$. Then $x[i] = (t \gg s)[i]$ for $1 \leq i \leq w - s$, and all other bits $x[i]$ set arbitrarily for $w - s < i \leq w$.
$\mathbf{o} \coloneqq \mathbf{m} \ll \mathbf{n}$	Any x with $0 \leq x \leq \mathit{ctz}(t)$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \gg \mathbf{m}$	Let s be a random value with $0 \leq s \leq \mathit{clz}(t)$, and let $w = \kappa(n)$. Then $x[i] = (t \ll s)[i]$ for $s < i \leq w$, and all other bits $x[i]$ set arbitrarily for $1 \leq i \leq s$.
$\mathbf{o} \coloneqq \mathbf{m} \gg \mathbf{n}$	Any x with $0 \leq x \leq \mathit{clz}(t)$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \circ \mathbf{m}$	Let $p = \kappa(n)$ and $q = \kappa(m)$ and $w = \kappa(o) = p + q$. Then $x = t[w : q + 1]$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{m} \circ \mathbf{n}$	Let $p = \kappa(n)$ and $q = \kappa(m)$ and $w = \kappa(o) = p + q$. Then $x = t[q : 1]$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n}[\mathbf{j} : \mathbf{i}]$	Then a value x with $x[k] = t[k + i - 1]$ for $1 \leq k \leq j - i + 1$ and all other bits set arbitrarily is a consistent value for n.
$\mathbf{o} \coloneqq$ if $\mathbf{c}$ then $\mathbf{n}$ else $\mathbf{m}$ $\mathbf{o} \coloneqq$ if $\mathbf{c}$ then $\mathbf{m}$ else $\mathbf{n}$ $\mathbf{o} \coloneqq$ if $\mathbf{n}$ then $\mathbf{m_1}$ else $\mathbf{m_2}$	Any value x is a consistent value for n.

Table 2: *Consistent* value computation for all bit-vector operators in $\mathcal{O}$, where t is the target value of operator node $o \in O$, assignment $\sigma \in \mathcal{C}$ is a complete assignment s.t. $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. t and σ as in Def. 17.

$\mathbf{o} \coloneqq\ \sim \mathbf{n}$	Then $x = {\sim}t$.
$\mathbf{o} \coloneqq \mathbf{n} = \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} = \mathbf{n}$	If $t = 1$, then $x = \sigma(m)$. Else, any $x \neq \sigma(m)$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n}\ \&\ \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m}\ \&\ \mathbf{n}$	Let i be a bit index with $1 \leq i \leq \kappa(n)$. If there is an i with $t[i] = 1$ and $\sigma(m)[i] = 0$, then there exists no inverse value for n. Otherwise, for all i, if $t[i] = 1$ then $x[i] = 1$, or if $t[i] = 0$ and $\sigma(m)[i] = 1$ then $x[i] = 0$, and else, $x[i]$ is set arbitrarily.
$\mathbf{o} \coloneqq \mathbf{n} < \mathbf{m}$	If $t = 1$ and $\sigma(m) = 0$, then there exists no inverse value. Else, any x with $t = x < \sigma(m)$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} < \mathbf{n}$	If $t = 1$ and $\sigma(m) = {\sim}0$, then there exists no inverse value. Else, any x with $t = \sigma(m) < x$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n} + \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} + \mathbf{n}$	Then $x = t - \sigma(m) = t + (1 + {\sim}\sigma(m))$.
$\mathbf{o} \coloneqq \mathbf{n} \cdot \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \cdot \mathbf{n}$	If $t = \sigma(m) = 0$, any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $t \neq 0$ and $\sigma(m) = 0$, or if $\sigma(m) \neq 0$ with $ctz(t) < ctz(\sigma(m))$, there exists no inverse value. Otherwise, $ctz(t) \geq ctz(\sigma(m))$ and $\sigma(m) \neq 0$. Let $y = m \gg ctz(\sigma(m))$, thus y is odd. We compute y^{-1} as its multiplicative inverse modulo 2^w, e.g., via the Extended Euclidean algorithm (similar to word-level rewriting techniques that require solving for a variable, e.g. [18]), and determine x as $(t \gg ctz(\sigma(m))) \cdot y^{-1}$ except that all bits in $x[w : w - ctz(\sigma(m)) + 1]$ are set arbitrarily, with $w = \kappa(n)$.
$\mathbf{o} \coloneqq \mathbf{n} \div \mathbf{m}$	If $t = {\sim}0$ and $\sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $t = {\sim}0$ and $\sigma(m) \notin \{0, 1\}$, or if $t \neq {\sim}0$ and $\sigma(m) = 0$, or if $t \cdot \sigma(m)$ produces an overflow, there exists no inverse value for n. Else, if $t = {\sim}0$ and $\sigma(m) = 1$, then $x = {\sim}0$. In any other case, any x with $t = x \div \sigma(m)$ is an inverse value.
$\mathbf{o} \coloneqq \mathbf{m} \div \mathbf{n}$	If $t = \sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $t = 0$ and $\sigma(m) = {\sim}0$, or if $m < t$, then there exists no inverse value for n. If $t = \sigma(m) = {\sim}0$, then $x \in \{0, 1\}$, and if $t = {\sim}0$ and $\sigma(m) \neq {\sim}0$, then $x = 0$. Else, if $t = 0$ and $\sigma(m) \neq {\sim}0$, then any random $x > \sigma(m)$ is an inverse value. In any other case, any x with $t = \sigma(m) \div x$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n} \bmod \mathbf{m}$	If $\sigma(m) \leq t$, then there exists no inverse value. Else, we select a $y \neq 0$ such that neither in the multiplication nor the addition operation of $\sigma(m) \cdot y + t$ occurs an overflow. Then $x = \sigma(m) \cdot y + t$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} \bmod \mathbf{n}$	If $\sigma(m) < t$, or if $t \neq 0$ and $t = \sigma(m) - 1$, or if $\sigma(m) - t \leq t$, then there exists no inverse value for n. Else, if $\sigma(m) = t$, then $x = 0$ or any $x > t$ is an inverse value for n. In any other case, any $x = (\sigma(m) - t) \div y$ with $y > 0$ such that $(\sigma(m) - t) \bmod y = 0$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n} \ll \mathbf{m}$	If $\sigma(m) = 0$, then any x is an inverse value. Else, if $ctz(t) \geq ctz(m)$, then $x = t \gg \sigma(m)$ with all bits in $x[w : w - \sigma(m) + 1]$ with $w = \kappa(n)$ set arbitrarily. In any other case, there exists no inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} \ll \mathbf{n}$	If $t = \sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $ctz(m) \leq ctz(t)$, if $t = 0$, any $x \geq ctz(t) - ctz(m)$ is an inverse value for n, and else, $x = ctz(t) - ctz(m)$ is an inverse value if the remaining shifted bits in t match with the corresponding bits in $\sigma(m)$, i.e., if $t[w : x + 1] = \sigma(m)[w - x : 1]$ with $w = \kappa(n) = \kappa(o)$. In any other case, there exists no inverse value for n.

Table 3: *Inverse* value computation for $\sim$, $=$, $\&$, $<$, $+$, $\cdot$, $\div$, mod and $\ll$, where t is the target value of operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment s.t. $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. t and σ as in Def. 17.

$\mathbf{o} := \mathbf{n} \gg \mathbf{m}$	If $\sigma(m) = 0$, then any x is an inverse value. Else, if $clz(t) \geq clz(m)$ then $x = t \ll \sigma(m)$ with all bits in $x[\sigma(m) : 1]$ set arbitrarily. In any other case, there exists no inverse value for n.
$\mathbf{o} := \mathbf{m} \gg \mathbf{n}$	If $t = \sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $clz(m) \geq clz(t)$, if $t = 0$, then any $x \geq clz(t) - clz(m)$ is an inverse value for n, and else, $x = clz(t) - clz(m)$ is an inverse value, if the remaining shifted bits in t match with the corresponding bits in $\sigma(m)$, i.e., if $t[w - x : 1] = \sigma(m)[w : x + 1]$ with $w = \kappa(m) = \kappa(o)$. In any other case, there exists no inverse value for n.
$\mathbf{o} := \mathbf{n} \circ \mathbf{m}$ $\mathbf{o} := \mathbf{m} \circ \mathbf{n}$	Then any consistent value x is an inverse value for n.
$\mathbf{o} := \mathbf{n}[\mathbf{j} : \mathbf{i}]$	Then any consistent value x is an inverse value for n.
$\mathbf{o} :=$ if $\mathbf{c}$ then $\mathbf{n}$ else $\mathbf{m}$ $\mathbf{o} :=$ if $\mathbf{c}$ then $\mathbf{m}$ else $\mathbf{n}$	Then $x = t$.
$\mathbf{o} :=$ if $\mathbf{n}$ then $\mathbf{m_1}$ else $\mathbf{m_2}$	Then $x = {\sim}\sigma(n)$

Table 4: *Inverse* value computation for $\gg$, $\circ$, [:] and if-then-else, where t is the target value of operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment s.t. $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. t and σ as in Def. 17.

Definition 22 (Backtracing Step) Let $\sigma \in \mathcal{C}$ be a complete consistent assignment. Given an operator node $o \in O$ with $o \to n$ and a target value $t \neq \sigma(o)$, then a *backtracing step* selects input n of operator node o w.r.t. σ as in Def. 17 and selects a *backtracing value* x for n as a *consistent* (and optionally *inverse*) value w.r.t. σ and t if such a value exists, and a *random value* otherwise.

Note that it is not always possible to find an inverse value for input n, e.g., $o := n \,\&\, m$ with $\sigma = \{o \mapsto 00, n \mapsto 00, m \mapsto 00\}$ and $t = 01$. Further, even for operators that allow to always produce inverse values, e.g., operator $+$, doing so may lead to inadvertently pruning the search space, see Ex. 23 below.

Example 23 Consider formula $\phi := v + v + 2_{[2]} = 0_{[2]}$ with root $r := p_2 = 0_{[2]}$, where $p_2 := v + p_1$ and $p_1 := v + 2_{[2]}$, and a complete consistent assignment $\sigma_1 = \{v \mapsto 00,\ p_1 \mapsto 10,\ p_2 \mapsto 10,\ r \mapsto 0\}$, as shown in Fig. 7a. Let $t = 1$ be the target value of root r, i.e., our goal is to find a value for bit-vector variable v such that $p_2 = 00$, and thus, formula ϕ is satisfied. Assume that as in Fig. 7a-b, only inverse values are selected for $+$ operators during propagation. Down propagating target values along the path indicated by blue arrows in Fig. 7a, the move $v \mapsto 10 = \alpha_1(v)$ is generated, which consequently yields assignment $\sigma_2 = \{v \mapsto 10,\ p_1 \mapsto 00,\ p_2 \mapsto 10,\ r \mapsto 0\}$ as indicated in Fig. 7b. Selecting the other possible propagation path, the same move is produced. Thus, σ_2 is independent of which of the two paths is selected. Since $\sigma_2(r) \neq t$, target value t is again propagated down, which generates move $v \mapsto 00 = \alpha_2(v)$, again independently of which path is selected. With this, we move back to the initial assignment σ_1. Consequently, a satisfying assignment, e.g., $\omega(v) = 01$ or $\omega'(v) = 11$, can not be reached by only selecting inverse values. However, selecting a consistent but non-inverse value for p_1 as, e.g., in Fig. 7c, generates move $v \mapsto 01 = \alpha'_1(v)$, which yields a satisfying assignment $\omega = \{v \mapsto 01,\ p_1 \mapsto 11,\ p_2 \mapsto 00,\ r \mapsto 1\}$.

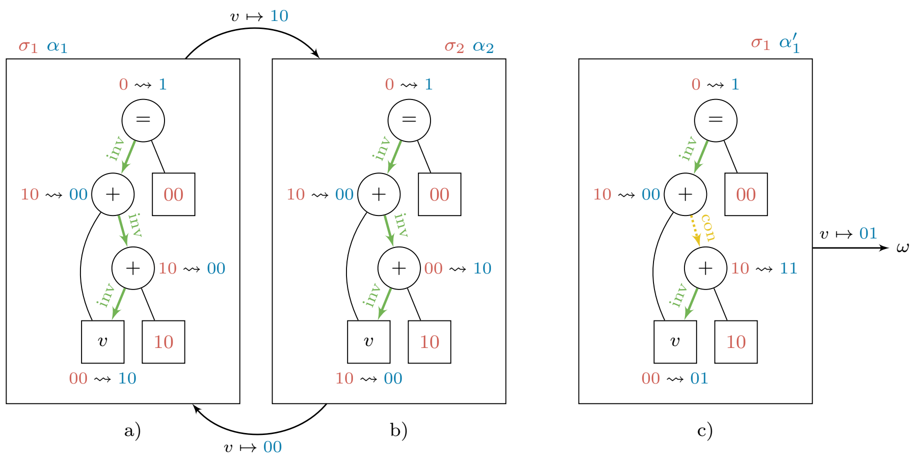


Fig. 7: Example illustrating the necessity of choosing between random and inverse values when down propagating assignments (backtracing). The output values indicate the (desired) transition from current to target value. Other values indicate the transition from current value to the inverse (inv) or consistent (con) value yielded by down propagating the target output value.

As shown in Ex. 23, a propagation strategy using only inverse values without further randomization is incomplete. Hence, when performing a backtracing step, we in general select some consistent non-inverse value, if no inverse value exists, and otherwise non-deterministically choose between consistent (but not necessarily inverse) and inverse values. Since all operators in $\mathcal{O}$ are *surjective* for our selected semantics (i.e., they can produce any target value, e.g., $\sim 0 \bmod 0 = \sim 0$), it is not necessary to select inconsistent random values. For other sets of operators, however, this might be necessary. For the sake of completeness we therefore included the selection of random values in the formal definition of backtracing steps.

Note that since on the bit-level the backtracing value for a selected input is uniquely determined (see Prop. 8), the issue of value selection is specific to the word-level. Further, when interpreting AIGs as word-level expressions, the notion of backtracing steps on the bit-level as in Def. 7 exactly matches the word-level notion as in Def. 22 using Prop. 15. As a side note, the problem of value selection during word-level backtracing and subsequent word-level propagation is similar to the problem of making a theory decision ("model assignment") and propagating this decision in MCSat [27, 32].

The *word-level propagation strategy* $\mathcal{P}$ is defined in exactly the same way as for the bit-level (see Def. 10) except that the word-level notion of backtracing based on essential inputs and consistent and inverse value selection (Def. 22) replaces bit-level backtracing based on controlling inputs (Def. 7), and the set of valid moves $\mathcal{P}(\sigma)$ contains not only the leafs of all expanded propagation traces but also their updated assignments, i.e., $(v, \alpha(v))$ for a leaf v. Further important concepts defined on the bit-level in Section 3 can be extended naturally to the word-level. These concepts include (expanded) paths and traces, leafs, and trace extension. We omit formal definitions accordingly.

Proposition 8, which is substantial for the bit-level proof of Lemma 11, does not directly apply on the word-level due to the more sophisticated selection of backtracing values. We lift Prop. 8 to the word-level as follows.

Proposition 24 *Let $\sigma \in \mathcal{C}$ be a complete consistent assignment, and let ω be a satisfying assignment $\omega \in \mathcal{W}$. Given operator node $o \in O$ with $o \rightarrow n$ and target value $t = \omega(o) \neq \sigma(o)$, i.e., $\sigma(o) \rightsquigarrow t$, then there exists a backtracing step w.r.t. assignment σ and target value t, which selects input n and backtracing value $x = \omega(n) \neq \sigma(n)$.*

Proof First, assume that operator node o has an essential input w.r.t. assignment σ. Then we select an arbitrary essential input n of o. Since target value $t = \omega(o) \neq \sigma(o)$, we get $\sigma(n) \neq \omega(n)$ by contraposition of Def. 13. Similarly, if o has no essential inputs, then we select n as an arbitrary input with $\sigma(n) \neq \omega(n)$, which has to exist since $\omega(o) \neq \sigma(o)$. In both cases, we can select $x = \omega(n) \neq \sigma(n)$ as backtracing value, which is consistent for operator node o w.r.t. assignment σ and target value t since ω is consistent. Picking a random value as backtracing value, which is the last case in Def. 22, can not occur under the given assumptions since, as already discussed, ω is consistent on o. $\square$

Using Prop. 24 instead of Prop. 8, the bit-level proof of Lemma 11 can then be lifted to the word-level by replacing every occurrence of gate g with operator node o, and the notion of "controlling" input with "essential" input.

Theorem 25 *Theorem 12 and Lemma 11 also apply on the word-level, and thus, propagation strategy $\mathcal{P}$ is also complete on the word-level.*

Note that even though Prop. 24 would allow us to restrict the selection of consistent and inverse backtracing values to be different from the current input node value, i.e., $x \neq \sigma(n)$, we do not enforce this property. Restricting value selection to a value $x \neq \sigma(n)$ interferes with path selection, in particular in the case where an input node is selected for which the current value is the only consistent or inverse value. We leave the exploration of this optimization to future work.

5 Experimental Evaluation

We implemented our propagation strategy within our SMT solver Boolector [34] and consider the following configurations.

(1) **Bb** The core Boolector engine, which implements a bit-blasting approach. This configuration is identical to the version that entered the QF_BV track of the SMT competition 2016 and uses (internal) version bbc of our SAT solver Lingeling [8] as back end solver.

(2) **Bsls** The score-based local search approach of [16] as implemented in Boolector [36], with random walks enabled. This approach lifts stochastic local search for SAT to the word-level and iteratively moves from a non-satisfying towards a satisfying assignment by flipping single bits or incrementing, decrementing and (bit-wise) negating the values of the primary inputs. Moves are in general selected as the best (improving) moves according to some score function, and

if no such move exists, a random value is chosen. If random walks are enabled, with a certain probability some random (and not necessarily the best) move is performed. This configuration mainly corresponds to the default configuration of [16] as implemented in Z3 [31] except for the score definition, which differs due to implementation issues (as described in [36]).

(3) **Paig** The bit-level configuration of our propagation-based approach, which operates on the AIG representation of a given input as bit-blasted by Boolector.

(4) **Pw** The word-level configuration of our propagation-based approach which directly operates on the given bit-vector formula, with inverse values prioritized over consistent values during backtracing with a probability of 99 to 1.

Note that the choice of rewriting and other simplification techniques applied prior to the actual decision procedure may considerably influence its performance. In order to provide the same basis for comparison and avoid skewed results due to differences in the rewriting and simplification techniques applied by Z3 [31] versus Boolector, we do not compare our propagation-based approach against the original implementation of [16] in Z3 but against our implementation of [16] in Boolector (configuration Bsls). All configurations of Boolector apply the same set of rewriting and simplification techniques in the same order.

Since [35] and in particular for the SMT competition 2016, we improved several core components of Boolector, which affects all the configurations above. The default configurations of Paig and Pw therefore show major improvements in comparison to [35]. In comparison to [36], the default configuration of Bsls, however, seems to perform worse. This is solely due to minor changes within the score-based local search engine of Boolector that affect the random number generator (RNG). We will show that the difference in the number of solved instances compared to [36] lies within the expected variance caused by randomization effects. Note that where not otherwise noted, in the default configuration of all local search configurations Bsls, Paig and Pw we will use a seed of value 0 for the RNG.

We compiled a set of in total 16436 benchmarks[1] and included all benchmarks with status *sat* or *unknown* in the QF_BV category of the SMT-LIB [6] benchmark library except those proved by Bb to be unsatisfiable within a time limit of 1200 seconds. We further excluded all benchmarks solved by Boolector via rewriting only. Note that our benchmark set is the same set we already used in [36] and [35]. Previously, all benchmarks in the Sage2 family that used non-SMT-LIBv2 compliant operators had to be explicitly excluded from the set above. However, since the SMT competition 2016, these benchmarks have been removed from SMT-LIB.

All experiments were performed on a cluster with 30 nodes of 2.83 GHz Intel Core 2 Quad machines with 8 GB of memory using Ubuntu 14.04.3 LTS. Each run is limited to use 7 GB of main memory. In terms of runtime we consider CPU time only. In case of a time out or memory out, the time limit is used as runtime.

Note that the results in [16] indicate that there still exists a considerable gap between the performance of state-of-the-art bit-blasting and word-level local search. However, the latter significantly outperforms bit-blasting on several instances. We therefore evaluated our local search configurations with regard to an application

[1] All experimental data of this evaluation can be found at `http://fmv.jku.at/fmsd16`.

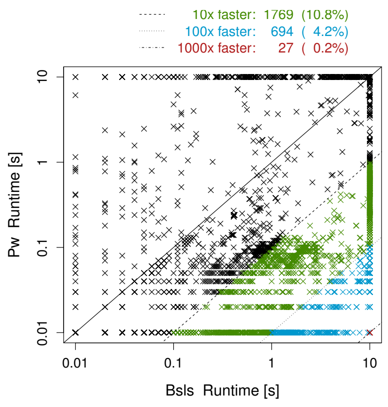
within a sequential portfolio setting and apply a limit of 1 and 10 seconds for the local search configurations, and a limit of 1200 seconds for the bit-blasting and the sequential portfolio configurations.

We evaluated our propagation-based strategy in comparison to the score-based local search approach in [16], in particular in terms of robustness with respect to randomization effects. We run a batch of 21 runs of each configuration Pw, Paig and Bsls with different seeds for the RNG of Boolector (one with default seed 0 and 20 with different random seeds) with a time limit of 10 seconds. Table 5 summarizes the results of configurations Bb, Bsls, Paig and Pw with a time limit of 10 seconds and default seed 0 for the local search configurations. As further illustrated in Fig. 8 and 9, overall, our word-level propagation strategy Pw clearly outperforms our bit-level propagation strategy Paig and the score-based local search approach Bsls.

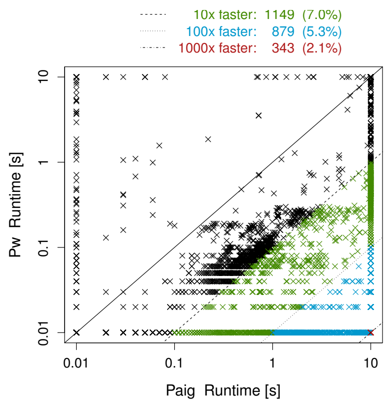
Figure 9 shows the results of Pw, Paig and Bsls over all 21 runs with different seeds in terms of number of solved instances and runtime as box-and-whiskers plots with the results of the runs with default seed 0 indicated with a red diamond. As a measure for robustness we use the standard deviation (SD) and the inter-quartile range (IQR), i.e., the distance between the lower quartile and the upper quartile, of the results of all 21 runs with different seeds, where lower values indicate a higher level of robustness. In terms of number of solved instances, for configuration Pw (SD: 17.88, IQR: 27) both the SD and the IQR is less than half of the SD and the IQR of Bsls (SD: 44.9, IQR: 60) and less than a third of the SD and IQR of Paig (SD: 62.6, IQR: 82). These results suggest that compared to Paig, both Pw and Bsls profit from directly working on the word-level, and overall, our word-level propagation-based strategy is indeed more robust with respect to randomization effects than the score-based local search approach of [16].

Even though overall Pw outperforms Paig and Bsls on some benchmarks in the families *sage*, *Sage2* and *stp_samples*, in comparison to Bsls (457 instances) and Paig (38 instances) configuration Pw seems to struggle. As an interesting observation, when bit-blasting the benchmarks in question, for the majority of benchmarks more than 50% of the bit-vector expressions contain bits that have been simplified to the Boolean constants $\{0, 1\}$ on the bit-level. Our bit-level strategy Paig operates on the bit-blasted AIG layer where all constant bits are eliminated via rewriting, and therefore always propagates target values that can actually be assumed. Our word-level strategy Pw, however, does not know which bits can be simplified to constant bits and may therefore determine and propagate target values that can never be assumed. Configuration Bsls, on the other hand, also does not have any explicit information on constant bits but considers them implicitly when exploring the neighborhood prior to performing a move since any neighbor with constant bits not matching their value will not result in score improvement.

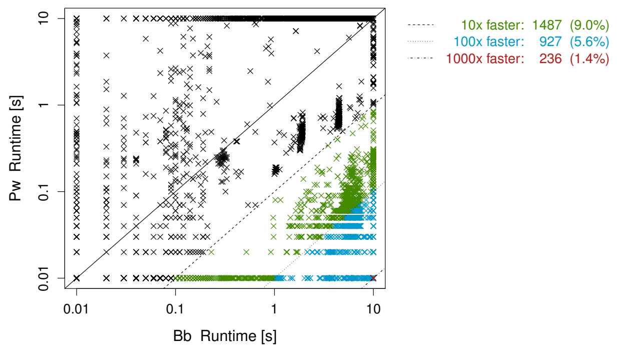
In an additional experiment, we evaluated the models of the 457 benchmarks on which Pw seems to have a disadvantage over Bsls and identified an interesting pattern. For more than 80% (374 instances) out of all 457 instances the assignment of more than 50% of the inputs was 0, and for 80% (293 instances) out of these instances, for more than 30% of the non-zero inputs only one bit was set to 1. Hence, since Bsls starts with an initial assignment where all inputs are set to 0, for this kind of benchmarks its focus on single bit flips allows to quickly move the initial assignment towards a satisfying assignment. For more than 60% of all 457 instances, Bsls required less than 50 moves (in comparison, the maximum



(a) Bsls vs. Pw



(b) Paig vs. Pw



(c) Bb vs. Pw

Fig. 8: Configurations Bsls, Paig and Bb versus Pw with a time limit of 10 seconds.

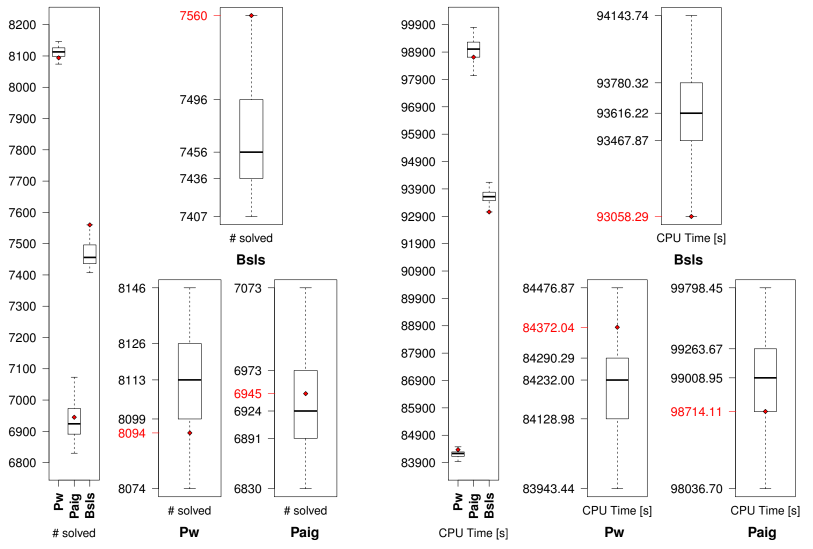


Fig. 9: Number of solved instances and runtime over 21 runs (with different seeds) of configurations Pw, Paig and Bsls with a time limit of 10 seconds.

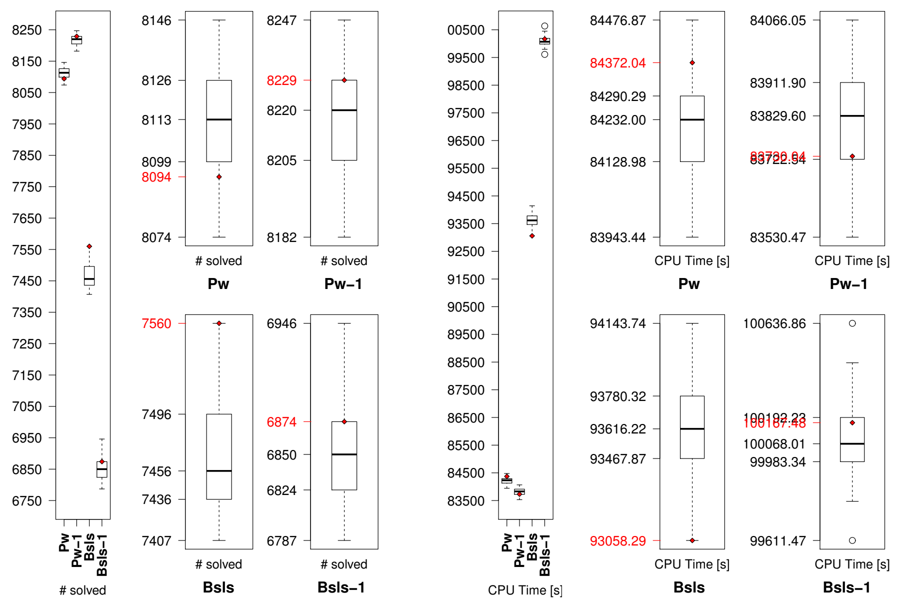


Fig. 10: Number of solved instances and runtime over 21 runs (with different seeds) of configurations Pw, Pw-1, Bsls and Bsls-1 with a time limit of 10 seconds.

number of moves for all solved instances is 3086). Configuration Pw starts with the same initial assignment as Bsls, however, as mentioned above, the fact that the majority of these 457 benchmarks contains a considerable amount of expressions with constant bits seems to handicap Pw. These results suggest that for this set of benchmarks the strategy of Bsls is advantageous over Pw and in particular profits from an initial assignment where all inputs are set to 0. Hence, in an additional experiment we introduce configurations **Bsls-1** and **Pw-1** where we initialized the inputs with all bits set to 1 rather than 0. Figure 10 shows the performance of Bsls-1 and Pw-1 in comparison to Bsls and Pw over 21 runs with different seeds (again, one with default seed 0 and 20 with different random seeds) with a time limit of 10 seconds. Table 6 further summarizes the results of Bsls, Bsls-1, Pw and Pw-1 with default seed 0. Overall, configuration Bsls obviously profits considerably from initializing the inputs with 0 since in comparison to Bsls the number of solved instances of Bsls-1 drops by almost 10%. In particular on the set of 457 benchmarks where Bsls had an advantage over our propagation-based strategy Pw, initializing the inputs with 1 resulted in Bsls-1 only solving 42 instances (9.2%) within a time limit of 10 seconds. Our propagation-based strategy, on the other hand, is much more robust than Bsls with respect to the input initialization value and seems to overall even profit from initializing the inputs with 1 rather than 0.

Figure 8c shows the performance of our propagation-based configuration Pw compared to our bit-blasting configuration Bb with a time limit of 10 seconds. As summarized in Table 5, even though there exists a considerable gap in the number of solved instances between Bb and Pw (within 10 seconds, Bb solves almost 2000

	Bb		**Bsls**		**Paig**		**Pw**	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	**48**	**3526.2**	0	3760.0	0	3760.0	0	3760.0
bench_ab (223)	223	0.2	**223**	**0.0**	**223**	**0.0**	**223**	**0.0**
bmc (22)	**20**	**58.5**	10	122.2	11	134.3	13	116.3
brummayerbiere (26)	5	228.2	25	13.5	12	184.8	**26**	**17.5**
calypto (13)	4	92.4	4	91.1	2	110.2	**5**	**93.3**
check2 (1)	1	0.0	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0	1	0.0
dwp_formulas (103)	103	0.4	**103**	**0.0**	**103**	**0.0**	**103**	**0.0**
fft (19)	**4**	**159.2**	0	190.0	0	190.0	1	180.5
float (126)	21	1124.9	0	1260.0	0	1260.0	**27**	**1033.7**
gulwani (6)	**5**	**28.3**	1	51.0	0	60.0	0	60.0
mcm (155)	**14**	**1477.1**	5	1523.1	5	1528.0	12	1440.2
pspace (21)	0	210.0	21	17.3	0	210.0	**21**	**1.6**
rubik (3)	**1**	**23.1**	0	30.0	0	30.0	0	30.0
RWS (20)	**14**	**84.5**	0	200.0	0	200.0	0	200.0
sage (6236)	**6236**	**2602.9**	5287	11117.4	4623	17036.3	5099	11615.9
Sage2 (6981)	**1564**	**60634.3**	613	64540.2	289	67648.1	526	64762.5
spear (1675)	1395	10587.1	1145	6848.8	1516	3516.0	**1668**	**205.2**
stp (1)	0	10.0	0	10.0	0	10.0	0	10.0
stp_samples (149)	**149**	**4.4**	120	523.4	129	217.9	104	546.5
tacas07 (3)	**3**	**11.5**	2	10.2	2	11.3	2	10.2
uclid (262)	261	741.1	2	2610.1	28	2467.0	**262**	**148.6**
VS3 (10)	0	100.0	0	100.0	0	100.0	0	100.0
wienand (4)	0	40.0	0	40.0	0	40.0	0	40.0
total (16436)	**10072**	**81744.5**	7560	93058.3	6945	98714.1	8094	84372.0

Table 5: Bit-blasting configuration Bb and the local search configurations Bsls, Paig and Pw with a time limit of 10 seconds grouped by benchmark families.

	Bsls		**Bsls-1**		**Pw**		**Pw-1**	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	0	3783.0	0	3786.0	0	3785.7	0	3785.2
bench_ab (223)	223	0.0	223	0.2	223	0.0	223	0.0
bmc (22)	10	122.8	10	121.9	13	116.9	12	117.6
brummayerbiere (26)	25	13.6	26	0.1	26	17.5	26	0.1
calypto (13)	4	91.4	3	100.5	5	93.7	5	89.8
check2 (1)	1	0.0	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0	1	0.0
dwp_formulas (103)	103	0.0	103	0.1	103	0.0	103	0.0
fft (19)	0	191.1	0	191.3	1	181.4	1	185.5
float (126)	0	1267.4	0	1267.5	27	1039.6	25	1064.8
gulwani (6)	1	51.3	0	60.5	0	60.3	0	60.4
mcm (155)	5	1531.8	3	1557.2	12	1448.6	12	1451.4
pspace (21)	21	17.3	21	0.9	21	1.6	21	0.0
rubik (3)	0	30.2	0	30.2	0	30.2	0	30.2
RWS (20)	0	201.2	0	201.0	0	201.0	0	201.1
sage (6236)	5287	11171.3	4910	14372.0	5099	11683.0	5096	11741.4
Sage2 (6981)	610	64912.0	619	65320.8	526	65149.0	667	63926.9
spear (1675)	1145	6879.4	895	9220.5	1668	205.7	1668	213.1
stp (1)	0	10.1	0	10.1	0	10.1	0	10.1
stp_samples (149)	120	525.2	36	1175.2	104	548.9	104	554.3
tacas07 (3)	2	10.3	2	10.4	2	10.2	2	10.3
uclid (262)	2	2625.3	21	2600.2	262	148.6	262	147.0
VS3 (10)	0	100.5	0	100.7	0	100.5	0	100.7
wienand (4)	0	40.2	0	40.3	0	40.2	0	40.3
totals	7560	93575.4	6874	100167.5	8094	84872.6	8229	83730.0

Table 6: Configurations Bsls and Pw versus Bsls-1 and Pw-1 with a time limit of 10 seconds grouped by benchmark families.

instances more than Pw), on 2650 benchmarks, Pw outperforms Bb by at least a factor of 10. In an additional experiment, we evaluated Pw with a time limit of 1200 seconds, which increases the number of solved instances compared to a time limit of 10 seconds by 7% (571 instances). These results suggest a combination of both configurations within a sequential portfolio setting [39], where our propagation-based strategy is run for a certain amount of time prior to invoking the bit-blasting engine. However, in practice, the number of propagation steps performed is a more reliable metric than the actual runtime of Pw within a sequential portfolio setting. In the following, we distinguish two sequential portfolio configurations.

(1) **Bb+Pw-virtual-Xs** A *virtual* sequential portfolio combination of Pw and Bb, where we assume that Pw is run exactly X seconds prior to invoking Bb.

(2) **Bb+Pw-X** The sequential portfolio combination of Pw and Bb as implemented in Boolector, where configuration Pw is run with a limit of X propagation steps prior to invoking Bb. Note that this configuration won the QF_BV division of the main track of the SMT competition 2016 with X=1000=1k.

Figure 11 illustrates the performance of a virtual sequential portfolio combination Bb+Pw-virtual-1s in comparison to the bit-blasting configuration Bb with a time limit of 1200 seconds, where we assume that configuration Pw is run for one second before falling back to the bit-blasting engine. Overall, configuration Bb+Pw-virtual-1s solves 63 instances more than Bb, and further outperforms Bb in terms of runtime by at least a factor of 10 on almost 2400 benchmarks.

Figure 12 shows the performance of the sequential portfolio combinations Bb+Pw-1k, Bb+Pw-10k, Bb+Pw-50k and Bb+Pw-100k in comparison to configuration Bb with a time limit of 1200 seconds, where Pw is run with a limit of 1 000, 10 000, 50 000 and 100 000 propagation steps before invoking the bit-blasting engine. With a limit of 1k propagation steps, configuration Bb+Pw-1k already solves 41 instances more than Bb. It further outperforms Bb in terms of runtime by at least a factor of 10 on more than 2400 benchmarks. Increasing the propagation step limit for configuration Pw to 10k, 50k and 100k further increases performance in term of runtime, with 2601 (Bb+Pw-10k), 2649 (Bb+Pw-50k) and 2657 (Bb+Pw-100k) instances solved by at least a factor of 10 faster than with configuration Bb. In terms of number of solved instances, configuration Bb+Pw-10k shows the best performance with a plus of 52 instances compared to Bb. Configurations Bb+Pw-50k and Bb+Pw-10k still solve 50 and 45 more instances than Bb, but lose instances compared to Bb+Pw-10k due to the increasing overhead introduced for those instances not solved within the given propagation step limit.

In an additional experiment with configurations Bb+Pw-1k and Bb+Pw-10k, we compiled a set of 21172 unsatisfiable benchmarks containing all QF_BV benchmarks in SMT-LIB with status *unsat* and determined the overhead introduced by Pw. With a total of 1237 seconds for configuration Bb+Pw-1k, the overhead for the unsatisfiable instances is negligible compared to the performance gain of almost 102k seconds on the satisfiable instances. For configuration Bb+Pw-10k, the overhead for the unsatisfiable instances is larger by a factor of 10 (10316 seconds), which is still an order of magnitude less than the performance gain of more than 116k seconds on the satisfiable instances.

Table 7 summarizes the results of configurations Bb, Bb+Pw-virtual-1s and Bb+Pw-10k, and gives a more detailed overview by benchmark family with a

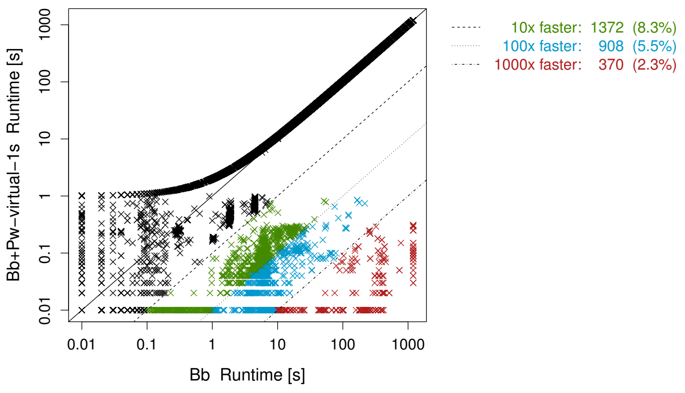


Fig. 11: Bb versus a *virtual* sequential portfolio configuration Bb+Pw-virtual-1s with a time limit of 1200 seconds.

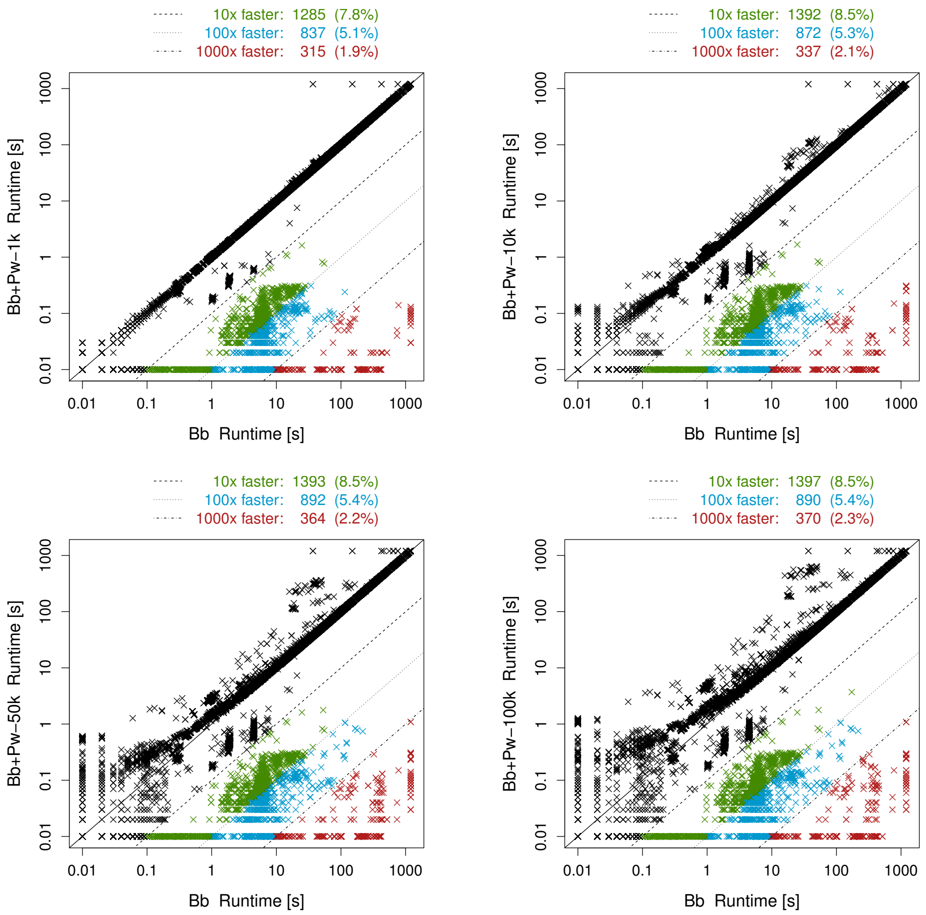


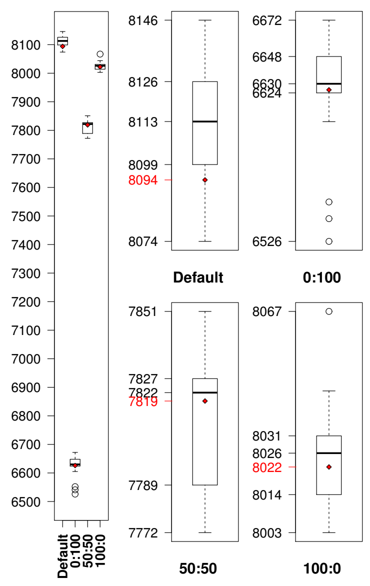
Fig. 12: Bb versus our sequential portfolio configurations Bb+Pw-1k, Bb+Pw-10k, Bb+Pw-50k and Bb+Pw-100k with a time limit of 1200 seconds.

	Bb		**Bb+Pw-10k**		**Bb+Pw-virtual-1s**	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	289	150600.3	287	155312.1	289	150889.3
bench_ab (223)	223	0.2	223	0.0	223	0.0
bmc (22)	22	63.6	22	83.8	22	71.6
brummayerbiere (26)	17	1759.8	26	46.7	26	97.9
calypto (13)	5	10423.7	5	10420.9	5	10423.5
check2 (1)	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0
dwp_formulas (103)	103	0.4	103	0.0	103	0.0
fft (19)	5	16924.2	5	16923.7	5	16928.6
float (126)	94	56417.8	94	56276.3	94	55497.3
gulwani (6)	6	47.6	6	48.3	6	53.6
mcm (155)	51	138276.6	51	138368.8	51	138273.6
pspace (21)	21	1964.5	21	1.6	21	1.6
rubik (3)	3	433.9	3	422.6	3	436.9
RWS (20)	18	3635.8	18	3649.8	18	3653.8
sage (6236)	6236	2602.9	6236	2492.5	6236	3622.2
Sage2 (6981)	5898	1853304.4	5940	1752833.2	5949	1738901.0
spear (1675)	1672	16202.9	1675	282.4	1675	278.3
stp (1)	1	18.9	1	20.9	1	19.9
stp_samples (149)	149	4.4	149	10.7	149	75.4
tacas07 (3)	3	11.5	3	6.2	3	6.0
uclid (262)	262	741.3	262	168.7	262	228.4
VS3 (10)	2	9859.6	2	9859.8	2	9861.6
wienand (4)	0	4800.0	0	4800.0	0	4800.0
total (16436)	**15082**	**2268094.4**	**15134**	**2152029.2**	**15145**	**2134120.5**

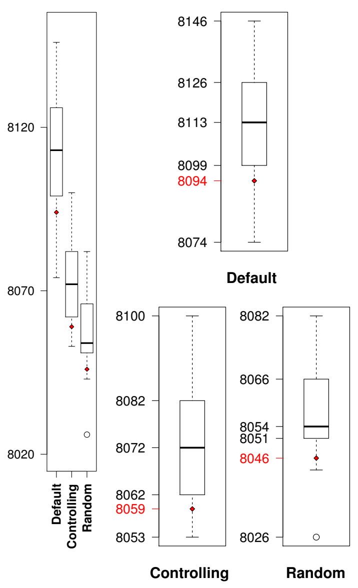
Table 7: Bit-blasting configuration Bb and sequential portfolio configurations Bb+Pw-10k and Bb+Pw-virtual-1s with a time limit of 1200 seconds.

time limit of 1200 seconds. As shown in Fig. 12, a propagation step limit of 100k (Bb+Pw-100k) almost corresponds to virtually limiting the runtime of Pw to 1 second (Bb+Pw-virtual-1s), in particular when considering the number of instances solved by at least a factor of 10 faster than Bb. A propagation limit of 10 000 (Bb+Pw-10k), however, yields the best results in terms of number of solved instances and the overall runtime.

Figure 13 shows the influence on randomization effects of our propagation-based strategy Pw in terms of the number of solved instances when introducing different levels of non-determinism during value selection and different path selection strategies over 21 runs with different seeds and a time limit of 10 seconds. In terms of value selection, the default configuration of Pw prioritizes inverse values over consistent values during backtracing with a probability of 99:1. As illustrated in Fig. 13a, decreasing this ratio, i.e., increasing the probability to choose consistent values over inverse values, increases the level of non-determinism of our backtracing algorithm, and as a consequence, the variance in terms of performance. The default ratio of 99:1 has a SD of 17.9 and decreasing the ratio of inverse to consistent values to 50:50 and 0:100 (consistent values only), the standard deviation increases to 23.5 and 38.9. When decreasing the level of non-determinism by increasing the ratio of inverse to consistent values to 100:0 (inverse values only), on the other hand, the SD drops to 14.1. Overall, as shown in Fig. 13a, a higher probability to choose inverse over consistent values also increases performance. However, as shown in Section 4, using inverse values only (ratio 100:0) is incomplete.



(a) Value Selection



(b) Path Selection

Fig. 13: Number of solved instances over 21 runs of configuration Pw with different levels of non-determinism during value selection (13a) and different path selection strategies (13b) and a time limit of 10 seconds.

In terms of path selection, not prioritizing inputs but choosing randomly corresponds to a maximum level of non-determinism. Prioritizing controlling inputs for Boolean operators already decreases non-determinism during path selection. However, utilizing essential inputs for all word-level operators decreases non-determinism even further. Figure 13b shows the influence of decreasing the level of non-determinism during path selection in terms of the number of solved instances over 21 runs with different seeds and a time limit of 10 seconds. By default, Pw prioritizes essential inputs for all word-level operators. Utilizing only controlling inputs of Boolean operators already decreases performance, and not prioritizing inputs but choosing randomly decreases performance even further. Prioritizing essential inputs for all word-level operators yields the best results.

6 Conclusion

In this paper, we presented our complete propagation-based local search strategy for the theory of quantifier-free fixed-size bit-vectors, which we previously presented in [35], in more detail.

We defined a complete set of rules for determining backtracing values when propagating assignments towards the primary inputs and provided extensive examples to illustrate the core concepts of our approach. We further provided a more extensive experimental evaluation, including an analysis of randomization effects caused by using different seeds for the random number generator. Motivated by the experimental results in [35], which showed the potential of a sequential portfolio combination of our propagation-based strategy and a state-of-the-art bit-blasting approach, we implemented this combination in our SMT solver Boolector. Our results confirm a considerable gain in performance.

Our procedure was evaluated on problems in the theory of quantifier-free bit-vectors in SMT. However, it is not restricted to bit-vector logics. Applying our strategy to other logics is probably the most intriguing direction for future work.

When combined with bit-blasting, our propagation-based techniques may learn properties of the input formula that might be useful for the bit-blasting engine. We leave learning and passing these properties to the bit-blasting engine to future work. Further, extending our propagation-based techniques by introducing strategies for conflict detection and resolution during backtracing as well as lemma generation to obtain an algorithm that is able to also prove unsatisfiability is another challenge for future work. A possible direction would be incorporating techniques from the MCSat for bit-vectors approach presented in [41].

Finally, we would like to thank Andreas Fröhlich and the reviewers for helpful comments, and Holger Hoos for fruitful discussions on the relation between non-deterministic completeness and the notion of probabilistically approximately complete (PAC).

References

1. Balint, A., Belov, A., Heule, M.J.H., Järvisalo, M. (eds.): SAT Competition 2013, *Dept. of Computer Science Series of Publications B*, vol. B-2013-1. University of Helsinki (2013)
2. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT challenge 2012 solver competition. Artif. Intell. **223**, 120–155 (2015)
3. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: SAT, *Lecture Notes in Computer Science*, vol. 7317, pp. 16–29. Springer (2012)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, *Lecture Notes in Computer Science*, vol. 6806, pp. 171–177. Springer (2011)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., Dept. of Computer Science, The University of Iowa (2015). Available at `www.SMT-LIB.org`
6. Barrett, C., Stump, A., Tinelli, C.: SMT-LIB. `www.SMT-LIB.org` (2010)
7. Belov, A., Heule, M.J.H., Järvisalo, M. (eds.): SAT Competition 2014, *Dept. of Computer Science Series of Publications B*, vol. B-2014-2. University of Helsinki (2014)
8. Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In: SAT Competition 2016 – Solver and Benchmark Descriptions, *Dept. of Computer Science Series of Publications B*, vol. B-2016-1, pp. 44–45. University of Helsinki (2016)
9. Brummayer, R.: Efficient SMT solving for bit-vectors and the extensional theory of arrays. Ph.D. thesis, Johannes Kepler University Linz (2009)
10. Bruttomesso, R.: RTL verification: from SAT to SMT(BV). Ph.D. thesis, University of Trento (2008)
11. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., Sebastiani, R.: A lazy and layered smt(BV) solver for hard industrial verification problems. In: CAV, *Lecture Notes in Computer Science*, vol. 4590, pp. 547–560. Springer (2007)
12. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The opensmt solver. In: TACAS, *Lecture Notes in Computer Science*, vol. 6015, pp. 150–153. Springer (2010)
13. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: TACAS, *Lecture Notes in Computer Science*, vol. 7795, pp. 93–107. Springer (2013)
14. Dutertre, B.: Yices 2.2. In: CAV, *Lecture Notes in Computer Science*, vol. 8559, pp. 737–744. Springer (2014)

15. Franzen, A.: Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT. Ph.D. thesis, University of Trento (2010)
16. Fröhlich, A., Biere, A., Wintersteiger, C.M., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: B. Bonet, S. Koenig (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA., pp. 1136–1143. AAAI Press (2015)
17. Ganesh, V.: Decision procedures for bit-vectors, arrays and integers. Ph.D. thesis, Stanford University (2007)
18. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV, *Lecture Notes in Computer Science*, vol. 4590, pp. 519–531. Springer (2007)
19. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS. The Internet Society (2008)
20. Goel, P.: An implicit enumeration algorithm to generate tests for combinational logic circuits. IEEE Transactions on Computers **30**(3), 215–222 (1981)
21. Griggio, A., Phan, Q., Sebastiani, R., Tomasi, S.: Stochastic local search for SMT: combining theory solvers with walksat. In: FroCoS, *Lecture Notes in Computer Science*, vol. 6989, pp. 163–178. Springer (2011)
22. Hadarean, L., Bansal, K., Jovanovic, D., Barrett, C., Tinelli, C.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: CAV, *Lecture Notes in Computer Science*, vol. 8559, pp. 680–695. Springer (2014)
23. Hansen, T.A.: A constraint solver and its application to machine code test generation. Ph.D. thesis, University of Melbourne (2012)
24. Hoos, H.H.: On the run-time behaviour of stochastic local search algorithms for SAT. In: AAAI/IAAI, pp. 661–666. AAAI Press / The MIT Press (1999)
25. Huang, C., Cheng, K.: Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques. In: DAC, pp. 118–123 (2000)
26. Iyer, M.A.: Race: A word-level atpg-based constraints solver system for smart random simulation. In: ITC, pp. 299–308. IEEE Computer Society (2003)
27. Jovanović, D., Barrett, C., de Moura, L.: The design and implementation of the model constructing satisfiability calculus. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pp. 173–180. IEEE (2013)
28. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science. An EATCS Series. Springer (2008)
29. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on CAD of Integrated Circuits and Systems **21**(12), 1377–1394 (2002)
30. Kunz, W., Stoffel, D.: Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques. Kluwer Academic Publishers, Norwell, MA, USA (1997)
31. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
32. de Moura, L.M., Jovanovic, D.: A model-constructing satisfiability calculus. In: VMCAI, *Lecture Notes in Computer Science*, vol. 7737, pp. 1–12. Springer (2013)
33. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine **28**(3), 13–30 (2007)
34. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. JSAT **9**, 53–58 (2015)
35. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: CAV (1), *Lecture Notes in Computer Science*, vol. 9779, pp. 199–217. Springer (2016)
36. Niemetz, A., Preiner, M., Biere, A., Fröhlich, A.: Improving local search for bit-vector logics in SMT with path propagation. In: DIFTS@FMCAD, pp. 1–10 (2015)
37. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: AAAI, pp. 337–343. AAAI Press / The MIT Press (1994)
38. Tillmann, N., Schulte, W.: Parameterized unit tests. In: ESEC/SIGSOFT FSE, pp. 253–262. ACM (2005)
39. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) **32**, 565–606 (2008)
40. Yuan, J., Pixley, C., Aziz, A.: Constraint-based verification. Springer (2006)
41. Zeljic, A., Wintersteiger, C.M., Rümmer, P.: Deciding bit-vector formulas with mcsat. In: SAT, *Lecture Notes in Computer Science*, vol. 9710, pp. 249–266. Springer (2016)