PolySAT: Word-level Bit-vector Reasoning in Z3

Jakob Rath[1], Clemens Eisenhofer[1], Daniela Kaufmann[1], Nikolaj Bjørner[2], and Laura Kovács[1]

[1] TU Wien, Vienna, Austria, `first.last@tuwien.ac.at`
[2] Microsoft Research, Redmond, USA, `nbjorner@microsoft.com`

Abstract. PolySAT is a word-level decision procedure supporting bit-precise SMT reasoning over polynomial arithmetic with large bit-vector operations. Addressing challenges of verified software, PolySAT integrates the theoretical development of SMT-based calculi with a proof of concept implementation and empirical evaluation. The PolySAT calculus extends conflict-driven clause learning modulo theories with two key components: (i) a bit-vector plugin to the equality graph, and (ii) a theory solver for bit-vector arithmetic with non-linear polynomials. PolySAT implements dedicated procedures to extract bit-vector intervals from polynomial inequalities. For conflict analysis and resolution, PolySAT comes with on-demand lemma generation over non-linear bit-vector arithmetic. PolySAT is integrated into the SMT solver Z3 and has applications in model checking and smart contract verification where bit-blasting techniques on multipliers/divisions do not scale.

Keywords: SMT Solving · Bit-vector Theory · Word-level Reasoning · Software Verification

1 Introduction

Bit-vector reasoning plays a central role in applications of system verification, enabling for example efficient bounded model checking [11], bit-precise memory handling [22], or proving safety of decentralized financial transactions [1]. Although one may argue that, because bit-vectors are bounded, bit-vector reasoning is simpler than proving arithmetic properties over the integers or reals, showing (un)satisfiability of bit-vector problems is inherently expensive due to complex arithmetic operations over large bit-widths [19].

Related works. State-of-the-art satisfiability modulo theories (SMT) solvers handle bit-vector operations by *bit-blasting* [20], i.e., translating bit-vector formulas into propositional ones that can be solved by ordinary propositional satisfiability (SAT) solvers. While the core idea of translating bit-vector operations to SAT formulas is quite natural, several variants of such translations arose. Some methods apply heavy preprocessing before bit-blasting, see STP [16], whereas others use over- and under-approximations to simplify solving, such as Boolector [26] and uclid [8]. Alternatively, other approaches bit-blast only relevant parts of the input, as developed in MathSAT [10] and cvc5 [2, 18].

Yet, the bit-blasting strategy performs poorly when multiplications are involved. As a result, stochastic local search, as in Bitwuzla [25] or Z3 [14], and int-blasting, as in cvc5 [31], have been developed. Local search works very well for satisfiable instances, but in general does not terminate for unsatisfiable (unsat) problems. On the contrary, int-blasting tends to work better for unsat formulas.

PolySAT – Our contribution. In this paper, we propose PolySAT, a *word-level reasoning procedure* integrated into SMT solving as a theory solver. PolySAT is based on conflict-driven clause learning modulo theories (CDCL(T)), providing thus *an alternative to bit-blasting.*

While our work builds on previous research on bit-vector slicing [7], forbidden intervals [17], and fixing bits [30], we extend these efforts as follows. We generalize forbidden intervals to non-unit coefficients (Section 5), while in [17] forbidden intervals are extracted only from constraints with unit coefficients. We further introduce theory lemmas to partially handle non-linear conflicts (Section 6), wheras in [17] such conflicts are deferred to bit-blasting. Finally, PolySAT uses intervals to track viable values and detect conflicts, whereas in [17], forbidden intervals are used only to construct a lemma after a conflict has been detected, and in [30], viable values are tracked by a combination of fixed bits and a single interval consisting of a lower and upper bound. We integrate bit-vector slicing from [7] and [17] as a plugin into the main e-graph of the SMT solver (Section 3.1).

In our setting, we consider bit-vectors as elements of the ring $\mathbb{Z}/2^w\mathbb{Z}$. Informally, arithmetical operations on bit-vectors can be seen as the respective integer operations, where the result is evaluated "$\bmod 2^w$". Yet, due to modulo/bounded arithmetic, many properties of the integers (such as, there is no maximal element and no zero-divisors) do not hold over bit-vectors. Nevertheless, with PolySAT we support bit-vector arithmetic without bit-blasting.

PolySAT – Illustrative example. We illustrate the benefits of PolySAT using the next example.

Example 1. Consider the bit-vector constraints with large bit-width w:

$$\begin{aligned} xy + y &>_{\mathsf{u}} y + 3 \qquad & 1 &= 3x + 6yz + 3z^2 \\ 6 &= 2y + z \qquad & 0 &= (2y+1)\ \&\ x \end{aligned}$$

where "&" denotes the bit-wise *and* operation and $>_{\mathsf{u}}$ refers to unsigned comparison. PolySAT proves this set of bit-vector constraints to be `unsat`, without using bit-blasting as follows.

We initially guess the assignment $x = 0$, simplifying the first constraint to $y >_{\mathsf{u}} y + 3$. We pick the assignment $y = 2^w - 2$ which is feasible w.r.t. the inequality. Hence, the constraint $6 = 2y + z$ simplifies to $z = 10$, which conflicts with the constraint $1 = 3x + 6yz + 3z^2$. We backtrack, apply variable elimination upon y on the two equality constraints, and learn the *new equation* $3x + 18z = 1$. From the bit-wise &-constraint, we derive that x is even, as $2y + 1$ is odd. This conflicts with the learned equation, as the learned equation $3x + 18z = 1$ implies that x is odd. Hence, PolySAT concludes that the given constraints are `unsat`.

PolySAT – Main improvements. With PolySAT, we bring the following main improvements to word-level reasoning over bit-vectors.

- We adjust the concept of forbidden intervals [17] to track viable values in PolySAT (Section 4);
- We extract bit-vectors intervals from polynomial (non-linear) inequalities (Section 5);
- We introduce lemmas on-demand for detecting and resolving non-linear conflicts in PolySAT (Section 6).
- We implement PolySAT directly in the SMT solver Z3 [24] and evaluate our work on challenging examples (Section 7).

Paper outline. We discuss required preliminaries in Section 2 and provide an overview of PolySAT in Section 3. We describe our main methodological contributions in Sections 4–6 and present our experimental evaluation in Section 7. Section 8 concludes our work.

2 Preliminaries

For a given number of bits $w > 0$, we consider bit-vectors of size w as elements of the ring $\mathbb{Z}/2^w\mathbb{Z}$ (algebraic representation), or equivalently as strings of length w over $\{0, 1\}$ (binary representation). Throughout the paper, we write w for the size of related bit-vectors, when it is clear from the context. In other cases, we denote the size of x by $|x|$ explicitly.

For conversion from bit-vectors to integers, unless explicitly stated otherwise, we default to the *unsigned* interpretation of bit-vectors, i.e., choose the representatives $\{0, 1, \ldots, 2^w - 1\}$ for elements of $\mathbb{Z}/2^w\mathbb{Z}$. Negative constants such as -1 stand for their equivalent $2^w - 1$.

We write $x \leq_{\mathsf{u}} y$ for unsigned comparison of bit-vectors, and use $x \leq_{\mathsf{s}} y$ to denote signed comparison. For simplicity of notation, we use "=" for both object-level equality and meta-level equality.

The basic building blocks of PolySAT constraints are *polynomials*, i.e., multiplications and additions of bit-vector variables and constants. We emphasize bit-vector multiplication by writing "$\cdot$" explicitly.

We write $x[i]$ for the i-th bit of the bit-vector x, where $x[0]$ denotes the least significant bit. Let $x +\!\!+ y$ denote the concatenation of x and y.

We write $x[h{:}l]$, with $0 \leq l \leq h < w$, for the *sub-slice* ranging from bit h to bit l inclusively, i.e., $x[h{:}l] = x[h] +\!\!+ x[h-1] +\!\!+ \cdots +\!\!+ x[l]$. We call the sub-slices $x[i{:}0]$ the *prefixes* of x.

We use half-open *wrapping* intervals over the domain $\mathbb{Z}/2^w\mathbb{Z}$. That is, for $l > h$ we define $[l; h[\;:= [0; h[\;\cup\; [l; 2^w[$. Then, $t \in [l; h[$ is equivalent to the bit-vector inequality $t - l <_{\mathsf{u}} h - l$.

3 PolySAT in a Nutshell

PolySAT serves as a decision procedure for bit-vector constraints and is developed as a theory solver within the SMT solver Z3 [24]. An overview of the

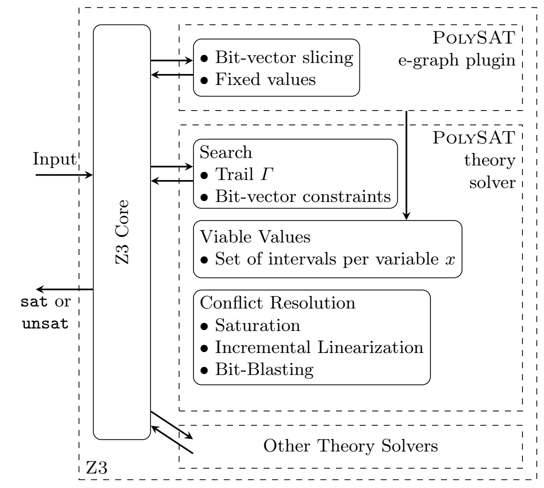


Fig. 1: PolySAT Integration

$p \leq_{\mathsf{u}} q$	unsigned inequality
$\Omega^*(p, q)$	multiplicative overflow
$x = p \mathbin{\&} q$	bit-wise *and*
$x = p \mid q$	bit-wise *or*
$x = p \ll q$	left shift
$x = p \gg q$	logical right shift
$x = p \gg_{\mathsf{a}} q$	arithmetic right shift

Fig. 2: Primitive Constraints

$p <_{\mathsf{u}} q$	$\rightsquigarrow$	$\neg(q \leq_{\mathsf{u}} p)$
$p \leq_{\mathsf{s}} q$	$\rightsquigarrow$	$p + 2^{w-1} \leq_{\mathsf{u}} q + 2^{w-1}$
$p = q$	$\rightsquigarrow$	$p - q \leq_{\mathsf{u}} 0$
$\Omega^+(p, q)$	$\rightsquigarrow$	$p + q <_{\mathsf{u}} p$
$p[i]$	$\rightsquigarrow$	$2^{w-1} \leq_{\mathsf{u}} 2^{w-i-1} p$
$p - q$	$\rightsquigarrow$	$p + (2^w - 1)q$
$\sim p$	$\rightsquigarrow$	$-p - 1$

Fig. 3: Derived Constraints

PolySAT architecture is given in Figure 1, with further details on key ingredients in Sections 4–6.

In a nutshell, PolySAT consists of two inter-connected components that interact for theory solving in an SMT setting:

1. A *bit-vector plugin to the equality graph, in short e-graph [12, 29]*. This plugin handles structural constraints that involve multiple bit-widths (concatenation, extraction) and determines canonical sub-slices of bit-vectors. The PolySAT e-graph plugin also propagates assigned values across bit-vector slices.
2. A *theory solver*, which handles the remaining constraints by translating them into *polynomial constraints* (Figure 2) and builds on information from the e-graph plugin to search for a satisfiable assignment (Sections 4–6).

From its e-graph, PolySAT receives Boolean assignments to bit-vector constraints, and equality propagations between bit-vector terms. In return, the theory solver of PolySAT produces a satisfying assignment, or a conflicting subset of the received constraints. We next discuss these two components, and then focus on the theory solving aspects of PolySAT in Sections 4–6.

3.1 E-graph Plugin

In SMT solving, an e-graph [12,29] is typically shared between theory solvers. The primary purpose of the e-graph is to infer equalities that follow from congruence reasoning. For PolySAT, the e-graph is extended with theory reasoning for bit-vectors. Theory reasoning is dispatched when the e-graph merges two terms of bit-vector sort. PolySAT determines of bits of variables that are fixed by certain types of constraints ("fixed bits") and performs constant propagation over bit-vector extraction and concatenation. Furthermore, the PolySAT e-graph establishes equalities between bit-vector ranges. For example, it infers that $x[5{:}4] = x[1{:}0]$ from the equation $x[5{:}2] = x[3{:}0]$.

We note that congruence reasoning for bit-vectors was also considered in [6, 7, 23]. Moreover, e-graphs are also used for constant propagation in [17]. The PolySAT integration of theory plugins to the e-graph structure is generic and not specific to bit-vectors.

3.2 Theory Solver

The *propositional search* is driven by the CDCL(T) core of the SMT solver [4, 27]. PolySAT receives Boolean assignments to bit-vector constraints and equality propagations between bit-vector terms. Both of them are translated into primitive constraints (cf. Figure 2) and tracked by the *trail* $\mathit{\Gamma}$. PolySAT maintains the invariant that each element of $\mathit{\Gamma}$ is justified by previous elements, and that each constraint and variable is assigned at most once in $\mathit{\Gamma}$.

Value search in PolySAT assigns viable values (see Section 4) to bit-vector variables, which are communicated back to the SMT solver core as variable assignment constraints.

Constraints. Overall, PolySAT fully supports the standardized bit-vector logic of SMT-LIB [3]. Extraction and concatenation are handled by PolySAT's e-graph plugin, while other bit-vector constraints are passed to its theory solver. Figure 2 depicts the *primitive constraints*, where p, q are bit-vector polynomials and x is a bit-vector variable. Other constraints are either internally reduced to primitive constraints as shown in Figure 3, or axiomatized upfront.

For example, to internalize the (unsigned) division $x \,/\, y$, PolySAT introduces fresh variables $q \coloneqq x/y$ and $r \coloneqq x \% y$ for the quotient and remainder, respectively. The main axiom is $x = qy + r$, but for correctness in bit-vector logic, four more axioms are required:

$$\begin{array}{ll} \neg\mathit{\Omega}^*(q, y) & \quad y \neq 0 \rightarrow r <_{\mathsf{u}} y \\ \neg\mathit{\Omega}^+(qy, r) & \quad y = 0 \rightarrow q = -1 \end{array}$$

where $\neg\mathit{\Omega}^+(qy, r)$ means that the addition $qy + r$ does not overflow, which can be implemented, e.g., as the constraint $qy \leq_{\mathsf{u}} -r - 1$.

Constraints of the form $x = n$, where x is a variable and n is a bit-vector constant, are called *variable assignments*. Bit-vector terms p and constraints c can be evaluated w.r.t. the current trail $\mathit{\Gamma}$, that is, we substitute the variable assignments in $\mathit{\Gamma}$ into p and c, respectively, and simplify. As a shorthand, we write $\widehat{p}$ for the evaluation of p under the current trail.

PolySAT uses rewriting to simplify different syntactic forms of equivalent constraints. In particular, we normalize several forms of equations that may appear in modular arithmetic. For instance, the constraints $p \leq_{\mathsf{u}} 0$, $p <_{\mathsf{u}} 1$, and $2^w - 1 \leq_{\mathsf{u}} p - 1$, are all normalized to $p = 0$.

Constraint Solving. The PolySAT theory solver uses a waterfall model of refinements to generate lemmas on demand, using the following steps:

1. *Propagation*: Value propagation is triggered when a variable is assigned a value (Section 4.1).
2. *Viable Interval Conflict*: If propagation tightens the feasible intervals of a variable to the empty set, the solver yields an interval conflict (Section 4.3).
3. *Case Split on Viable Candidates*: If no further propagation is possible, and there are no interval conflicts, the solver picks a value for the next unassigned variable, if any. It produces a literal $x = n$ for the CDCL solver to case split on, with a preference to the phase $x = n$ over $x \neq n$. The constant n is chosen to be outside the ranges of infeasible intervals stored for x so far (Section 4.2).
4. *Saturation Lemmas*: Saturation lemmas let us propagate consequences from non-linear constraints (Section 6.1).
5. *Incremental Linearization*: Our solver includes incremental linearization rules for the cases where variables are 0, 1, -1, or powers of two (Section 6.2).
6. *Bit-blasting*: As a final resort, PolySAT admits bit-blasting rules (Section 6.3).

The first three steps above (steps 1, 2, 3) operate on linear constraints, or rather, a *linear abstraction* of the original constraints, where non-linear monomials are treated as variables themselves. If no conflicts arise from the linear abstraction, then any conflicting non-linear constraints are handled by the latter stages (steps 4, 5, 6 above).

A conflict at any stage will cause PolySAT to return a conflict lemma to the SMT solver core, which will then backtrack and continue with search. When control is passed to PolySAT the next time, theory solving in PolySAT will begin again in the above step 1 of constraint solving.

4 Tracking Viable Values

In the following, we discuss the key ingredients of the theory solving component of PolySAT. A crucial part of the PolySAT theory solver tracks for each bit-vector variable x an over-approximation of the set of feasible values under the current trail Γ, which we call the *viable* values of x. Specifically, the set of viable values is represented as a set of *forbidden intervals*, each of which excludes a certain range of values of x, and is justified by constraints in the current trail Γ.

In PolySAT, we adapt forbidden intervals from [17] and use intervals for propagating and querying viable values of variables (Sections 4.1–4.2), and resolving respective conflicts (Section 4.3). Our approach extends [17] by computing intervals when the coefficient of x is not a power of two (Section 5.1), or when the coefficients are different on both sides of an inequality (Section 5.2).

4.1 Value Propagation

PolySAT extracts forbidden intervals from inequalities and overflow constraints c that are linear in x under the current trail Γ. Formally, we determine an inter-

Algorithm 1: PolySAT Viable Value Query

```
Input  : Set of forbidden intervals $\mathcal{I}$, set $C$ of constraints
Output : Viable value $x_0$, or a conflict
1  $x_0 \leftarrow x_{prev}$                                ▷ Start at previous viable value
2  $\mathcal{J} \leftarrow \langle\rangle$                  ▷ Justification (sequence of visited intervals)
3  loop
4      while $\exists I \in \mathcal{I}$ such that $x_0 \in I$ do
5          Choose such an $I \in \mathcal{I}$ with smallest bit-width
6          $\mathcal{J} \leftarrow \langle \mathcal{J}; I \rangle$
7          $x_0 \leftarrow forward(x_0, I)$
8          if $isConflict(\mathcal{J})$ then return Conflict $\mathcal{J}$
9      if $x_0$ does not violate any $c \in C$ then return $x_0$
10     $\mathcal{I} \leftarrow \mathcal{I} \cup \{computeInterval(C, x_0)\}$
```

val $[l; u[$ and side conditions $c_1, \ldots, c_n$ that hold under Γ such that

$$c \wedge c_1 \wedge \cdots \wedge c_n \implies x \notin [l; u[.$$

Intervals are ordered by their starting points, and we drop intervals that are fully contained in other intervals. Section 5 explains how intervals are obtained from constraints.

Value propagation in PolySAT is triggered when a variable is assigned a value, or in other words, the solver is presented with a literal $x = n$, where n is a value. Propagation is limited to linear occurrences of variables. For example, if x is assigned 2, then from $x + y \geq_{\mathsf{u}} 10$, the non viable intervals for y are updated to $y \notin [-2; 8[$. On the other hand, for $xz + y \geq_{\mathsf{u}} 10$, where x occurs in a non-linear term, there is no propagation. Non-linear propagation in PolySAT is currently side-stepped because we noticed that it produced very weak lemmas from viable interval conflicts. Non-linear conflicts are therefore handled separately, see Section 6.

4.2 Viable Value Query

To find a viable value for variable x, we collect the forbidden intervals $\mathcal{I}$ over the prefixes $x[k{:}0]$ of x for $0 \leq k < w$. In this context, iff an interval $I \in \mathcal{I}$ is an interval for $x[k{:}0]$, we say I *has bit-width* $k + 1$. In addition, we consider intervals for variables that are equivalent to a prefix of x, as determined by the current state of the e-graph.

In addition to forbidden intervals, we keep track of the set C of constraints that are linear in x. We then invoke Algorithm 1 to either find a value for x or detect a conflict. We resolve limitations of [17] by using intervals to track viable values and detect conflicts in Algorithm 1 as follows.

Algorithm 1 starts out with the previous viable value x_{prev} of x, initially set to 0. Then, in the loop of Algorithm 1, we check whether any of the known intervals $\mathcal{I}$ contain the current candidate value x_0 of x. If that is not the case, then the current value x_0 is compatible with the intervals in $\mathcal{I}$. We additionally test x_0 for admissibility against the set C of constraints (line 9 of Algorithm 1). If none of these constraints are violated, the candidate value x_0 is returned as viable value for x. Otherwise (line 10 of Algorithm 1), $\mathit{computeInterval}(C, x_0)$ extracts a new interval that covers x_0 (cf. Section 5) and the search for a viable value of x continues. If, on the other hand, the current value x_0 of x is contained in some forbidden interval, we choose an interval I of minimal bit-width among these (line 5 of Algorithm 1) and record it in the list $\mathcal{J}$ of justifications (line 6 of Algorithm 1).

The candidate value x_0 of x is updated to $\mathit{forward}(x_0, I)$, the first value after x_0 that is not covered by I (line 7 of Algorithm 1). If a conflict is detected (line 8 of Algorithm 1), the justifications $\mathcal{J}$ are returned for further processing (see Section 4.3).

4.3 Interval Conflict

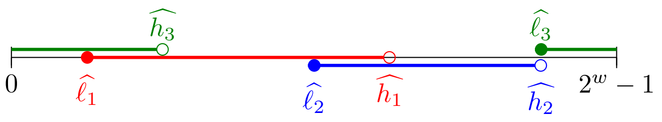
We detect conflicts by examining the list of justifications $\mathcal{J}$ after appending a new interval I to $\mathcal{J}$. The condition $\mathit{isConflict}(\mathcal{J})$ in Algorithm 1 is true iff the latest interval I has already been visited previously, and no interval of larger bit-width has occurred in between. Let $I_1, \ldots, I_{n+1}$ denote this subsequence of intervals, where $I_1 = I_{n+1} = I$, and let $I_i = [l_i; h_i[$. To block the current assignment to x, PolySAT creates a conflict lemma from $I_1, \ldots, I_{n+1}$ and reports it to its SMT core. For simplicity, we only explain here the case where all intervals have same bit-width.

The PolySAT conflict lemma is used to capture the following fact: the union of $I_1, \ldots, I_n$ covers the full domain $\mathbb{Z}/w\mathbb{Z}$, and the intervals have been chosen such that each upper bound h_i in contained in the next interval I_{i+1}. In other words, as long as $h_i \in I_{i+1}$ holds, for all i, and the intervals are valid for x, there can be no feasible value for x in PolySAT. While the PolySAT conflict lemma is similar to the one of [17], we note that [17] succinctly represents constraints $h_i \in I_{i+1}$ when multiple bit-widths are involved; this is not the case with PolySAT as we avoid using extract-expressions.

Since the constraints $h_i \in I_{i+1}$ in PolySAT do not contain x itself, they are useful for formulating a conflict lemma. Let C_i denote the set consisting of the constraint and side conditions of I_i. Then, the PolySAT conflict lemma is

$$\bigwedge_{i=1}^{n} C_i \wedge \bigwedge_{i=1}^{n} h_i \in I_{i+1} \implies \bot.$$

To illustrate the idea of conflict lemma generation in PolySAT, consider three intervals $[l_1; h_1[$, $[l_2; h_2[$, $[l_3; h_3[$ whose concrete evaluation under the current trail Γ covers the full domain by forming the following configuration:



Assuming the three intervals are justified by constraints C_1, C_2, C_3, respectively, the PolySAT conflict lemma is

$$\bigwedge C \wedge h_1 \in [l_2; h_2[\wedge h_2 \in [l_3; h_3[\wedge h_3 \in [l_1; h_1[\implies \bot,$$

where $C \coloneqq C_1 \cup C_2 \cup C_3$.

5 Computing Intervals

We now describe how forbidden intervals are extracted from a constraint $c \in C$ that is linear in the variable x under consideration. Intervals may be computed on demand, relative to a given candidate value (sample point) x_0 of x: the goal is then to find a maximal interval around x_0 of x-values that are excluded by c. In practice, we note the intervals are often not strictly maximal, but as large as reasonably possible to compute.

5.1 Linear Inequality with Equal Coefficients

Given the inequality constraint $px + q \leq_{\mathsf{u}} rx + s$ that is linear in x. In the cases where either p or r evaluate to 0 or both to the same value a, the inequality constraint is equivalent to an interval constraint [17], according to the following table, and subject to side conditions $p = \widehat{p}$ and $r = \widehat{r}$:

Constraint under Γ	Forbidden Interval	Condition
$ax + \widehat{q} \leq_{\mathsf{u}} \widehat{s}$	$ax \notin [s - q + 1; -q[$	$s \neq -1$
$\widehat{q} \leq_{\mathsf{u}} ax + \widehat{s}$	$ax \notin [-s; q - s[$	$q \neq 0$
$ax + \widehat{q} \leq_{\mathsf{u}} ax + \widehat{s}$	$ax \notin [-s; -q[$	$q \neq s$

Assume we have $ax \in [l; h[$. Yet, we want to extract an interval on x, rather than on ax.

Case $a = \pm 1$: The case $a = 1$ trivially leads to such an interval. In the case $a = -1$ (i.e., $2^w - 1$), the transformation $-x \in [l; h[\Leftrightarrow x \in [1 - h; 1 - l[$ is applied.

Case $a = \alpha 2^k$ (reducing the bit-width): Consider the case where a is divisible by 2^k for some $k > 0$. Due to the factor 2^k, the upper k bits of x do not influence the value of the constraint. In this case, we consider an interval for the prefix $x[w - k - 1{:}0]$ of x:

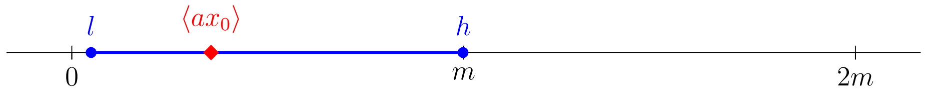
$$\alpha 2^k x \notin [l; h[\iff \begin{cases} \alpha x[w-k-1{:}0] \notin [l'; h'[& \text{if } l' \neq h' \\ 0 \notin [l; h[& \text{otherwise} \end{cases}$$

where $\beta' \coloneqq \lceil \frac{\beta}{2^k} \rceil \bmod 2^{w-k}$ for $\beta \in \{l, h\}$.

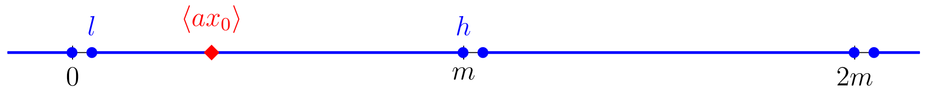
Other values of a: For other values of a, in general, multiple disjoint intervals exist. We extract intervals around a sample point x_0 on demand, i.e., given concrete values $a, x_0, l, h \in \mathbb{Z}/2^w\mathbb{Z}$ such that $ax_0 \in [l; h[$, the task is to compute the maximal x-interval $[x_l; x_h[$ such that $ax \in [l; h[$ for all $x \in [x_l; x_h[$. To compute x_l and x_h, we move the problem into the integers $\mathbb{Z}$ and work with non-wrapping intervals. Operations until the end of this subsection are therefore to be understood as operations in $\mathbb{Z}$.

Let w be a fixed bit-width and let $m := 2^w$. Assume values $a, x_0, l, h \in \mathbb{Z}$ are given such that $1 \le a < m$, $-m < l \le h < m$, and $ax_0 \bmod m \in [l; h]$. Furthermore, the length of the interval should be less than m, i.e., $h - l + 1 < m$ (otherwise the computation is unnecessary because the corresponding modular interval covers the whole domain). The goal is to find the minimal x_l and the maximal x_h such that $ax \bmod m \in [l; h]$ for all $x \in [x_l; x_h]$.

Let $k_0 \in \mathbb{Z}$ such that $l \le ax_0 + k_0 m \le h$. To simplify notation, define $\langle x \rangle := x + k_0 m$. The initial configuration is illustrated by the following diagram:

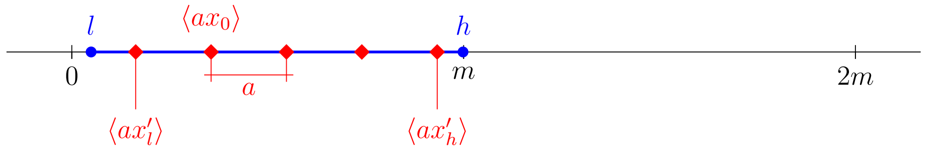


Since we are ultimately interested in the modular interval $[l; h] \bmod m$ over $\mathbb{Z}/m\mathbb{Z}$, we consider the set of all representatives of elements of that interval, i.e., the union of $[l; h] + im$ for all $i \in \mathbb{Z}$, as depicted in the following diagram.

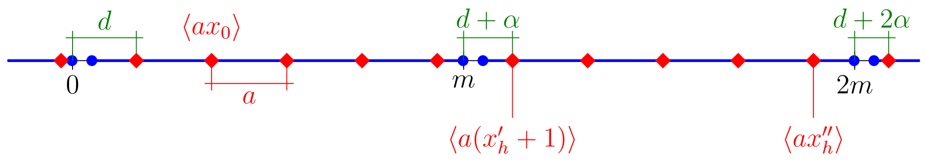


The underlying idea of our procedure is to look at each interval representative $[l; h] + im$ separately (intuitively, as a region where no overflow occurs) and take advantage of periodicity after each overflow.

In the first step, we compute the minimal x'_l and the maximal x'_h such that $l \le \langle ax \rangle \le h$ for all $x \in [x'_l; x'_h]$. Intuitively, $[x'_l; x'_h]$ is the maximal x-interval around x_0 such that no overflow occurs among the corresponding multiples of a.



However, the interval $[x'_l; x'_h]$ is often far from optimal, causing repeated queries over the same constraint in Alg. 1. In case of the upper bound, this means that $\langle a(x'_h + 1) \rangle$ is contained in the next interval representative $[l; h] + m$. The following diagram illustrates the multiples of a across several interval representatives.



The situation in the second interval $[l;h]+m$ is very similar to the initial setting. However, the multiples of a (depicted by red diamonds) have shifted by some amount α relative to the interval.

In the example illustrated in the diagrams we have $\alpha < 0$, i.e., with each overflow, the multiples of a drift to the left (relative to the interval). With different parameters, $\alpha = 0$ (no drift) and $\alpha > 0$ (drift to the right) are also possible.

For $\alpha < 0$, we keep overflowing until the leftmost multiple of a drifts outside the interval. For $\alpha > 0$, similarly for the rightmost multiple of a (in this case, the final considered interval will be irregular in the sense that it contains one fewer multiple of a).

In case $\alpha = 0$, the situation for each interval representative is exactly the same, and we conclude no upper bound x_h exists (which means the final x-interval over $\mathbb{Z}/m\mathbb{Z}$ will be the full domain).

We have described our method to compute the upper bound x_h. The lower bound x_l can be computed analogously. In fact, PolySAT reduces the computation of x_l to the computation of x_h by mirroring the initial configuration and the result across 0. Let f denote the procedure for calculating x_h, i.e., $x_h = f(x_0, a, l, h, m)$. Then $x_l = -f(-x_0, a, -h, -l, m)$.

Even though this method works well in practice, some limitations remain. The interval extension ends as soon as one of the red diamonds is outside the blue interval. This is by specification, but it does mean that this method is only helpful when the gap between blue intervals (i.e., $m - (h - l)$) is less than the distance between red diamonds (i.e., a).

5.2 Linear Inequality with Different Coefficients

Consider an inequality c of the form $px + q \leq_{\mathsf{u}} rx + s$ with $\widehat{p} \neq \widehat{r}$. Here, we need to find the largest x-interval around a sample point x_0 where c is satisfied. The corresponding problem is easily solved over infinite domains, such as rationals, by computing the intersection point of the left- and right-hand side of the inequality. The interval then extends from the intersection point towards infinity.

However, in modular arithmetic, the left-hand side and the right-hand side of c do not represent continuous lines; instead, they wrap around at 2^w. As such, the solution is not necessarily a single interval; the desired intervals extend from an intersection point to the next wraparound point. PolySAT computes and returns the interval containing x_0. This method works best when the coefficients $\widehat{p}$ and $\widehat{q}$ of x are near 0 or 2^w. More details are given in Appendix A.

5.3 Projecting Intervals to Sub-Slices

Since value assignments are propagated eagerly across bit-vector slices by the e-graph component of PolySAT, in some cases, a bit-vector variable is assigned to a value that contradicts an interval on a super-slice of the variable. Such contradictions may also be caused by the e-graph, because it does not take into account intervals when merging nodes.

Let $x := y \mathbin{+\!\!+} z$ s.t. $|y| = u$ and $|z| = v$. Given the forbidden interval $x \notin [l;h[$, then $2^v y + z \notin [l;h[$. We learn intervals for y and z via the following PolySAT lemmas.

Lemma 1 (General Intervals). *In case no fixed value is known for the other sub-slice, it is possible to learn an interval as long as $[l;h[$ is big enough.*

$$\mathit{len}([l;h[) \geq 2^u \implies y \notin [l_y; h_y[\tag{1}$$
$$\mathit{len}([l;h[) > 2^{u+v} - 2^v \implies z \notin [l_z; h_z[\tag{2}$$

where $l_y := \lceil \frac{l}{2^v} \rceil \bmod 2^v$, $h_y := \lfloor \frac{h}{2^v} \rfloor$, $l_z := l \bmod 2^v$, *and* $h_z := h \bmod 2^v$.

Lemma 2 (Specific Intervals). *If the other sub-slice has a fixed value, a larger interval can be projected [17, Figure 1].*

$$z = n \wedge l_y \neq h_y \implies y \notin [l_y, h_y[\tag{3}$$
$$z = n \wedge l_y = h_y \wedge h_y 2^v + n \in [l;h[\implies \bot \tag{4}$$
$$y = n \wedge l_z \neq h_z \implies z \notin [l_z; h_z[\tag{5}$$
$$y = n \wedge l_z = h_z \wedge n 2^v \in [l;h[\implies \bot \tag{6}$$

where ($\beta \in \{l, h\}$)

$$\beta_y := \left\lceil \frac{(\beta - n) \bmod 2^{u+v}}{2^v} \right\rceil \bmod 2^u, \qquad \beta_z := \begin{cases} \beta \bmod 2^v & \text{if } \lfloor \frac{\beta}{2^v} \rfloor = n, \\ 0 & \text{otherwise.} \end{cases}$$

These projections are applied iteratively in PolySAT to derive intervals for arbitrary sub-slices. At each step, a choice is made between Lemmas 1–2, depending on whether a fixed value is available at the required decision level.

Example 2. We can use the above to find an interval I such that $x = 0 \mathbin{+\!\!+} y \mathbin{+\!\!+} z \wedge z[15{:}8] = 123 \wedge x \notin [300007; 0[$ implies $y \notin I$, where $|x| = 64$ and $|y| = |z| = 16$.
- First, apply (5) to obtain $y \mathbin{+\!\!+} z \notin [300007; 0[$.
- Next, with (1) we obtain $y \mathbin{+\!\!+} z[15{:}8] \notin [1253; 0[$.
- Finally, with (3) we obtain $y \notin [5; 0[$.

6 Non-Linear Conflicts

Non-linear conflicts are handled in PolySAT by saturation, incremental linearization, and bit-blasting. Saturation, incremental linearization and bit-blasting are postponed until all variables are assigned values and there are no conflicts detected by propagating bounds on linear constraints.

6.1 Saturation Lemmas

Saturation lemmas propagate consequences from non-linear constraints. The consequences are considered “simpler”, when they are linear or if they contain fewer variables. Saturation lemmas, given in Lemmas 3–6, are added by PolySAT if their non-linear constraints are in the assertion trail and they evaluate to false under the current assignment in Γ.

Lemma 3 (Saturation Modulo Multiplication Inequalities). *We give an excerpt of possible saturation rules. An extended list can be found in Appendix B.*

$$\begin{array}{lcllll}
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(p,x) & \vee\, p <_{\mathsf{u}} q & & \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(-q,x) & \vee\, p <_{\mathsf{u}} q & & \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(q,-x) & \vee\, p >_{\mathsf{u}} q & \vee\, p = 0 & \\
px <_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(-p,-x) & \vee\, p >_{\mathsf{u}} q & \vee\, p = 0 & \\
px \leq_{\mathsf{u}} qx & \Longrightarrow & \Omega^*(p,x) & \vee\, p \leq_{\mathsf{u}} q & \vee\, x = 0 & \\
px + s \leq_{\mathsf{u}} q & \Longrightarrow & \Omega^*(p,x) & \vee\, \Omega^+(px,s) & \vee\, pr \leq_{\mathsf{u}} q & \vee\, x <_{\mathsf{u}} r \\
p \leq_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,x) & \vee\, pq \leq_{\mathsf{u}} r & & \\
p \leq_{\mathsf{u}} x \wedge qx <_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,x) & \vee\, pq <_{\mathsf{u}} r & & \\
p \leq_{\mathsf{u}} qx \wedge x \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,r) & \vee\, p \leq_{\mathsf{u}} qr & & \\
p <_{\mathsf{u}} qx \wedge x \leq_{\mathsf{u}} r & \Longrightarrow & \Omega^*(q,r) & \vee\, p <_{\mathsf{u}} qr & &
\end{array}$$

Note that these rules do not require $x \notin p, q, r, s$, so they can be applied even when the degree of x is larger than 1.

Obtaining Saturation Lemmas. Since bit-vector arithmetic does not match the intuition of standard arithmetic, it can take a lot of effort to come up with saturation lemmas manually. We have therefore employed some automation to discover the rules given in Lemma 3. We start with the constraint on the LHS of the rule (e.g., $px <_{\mathsf{u}} qx$) and generate a set of constraints that we want to allow in the RHS. We then add the constraints for a small fixed bit-width to Z3 and employ the MARCO algorithm [21] to find the minimal unsatisfiable subsets (MUS). Each MUS corresponds to a valid lemma; however, to be useful as saturation lemmas, we filter the candidates such that the RHS is simpler in some sense. Finally, we verify manually that the lemmas generalize to arbitrary bit-widths.

Next, we can connect overflow constraints with multiplications or decompose them to linear inequalities.

Lemma 4 (Overflow Saturation).

$$\begin{array}{lcl}
\neg\Omega^*(p,q) \wedge q \neq 0 & \Longrightarrow & p \leq_{\mathsf{u}} p \cdot q \\
\bar{0}p \cdot \bar{0}q \geq_{\mathsf{u}} 2^w & \Longrightarrow & \Omega^*(p,q) \\
\Omega^*(p,q) \wedge \neg\Omega^*(r,s) & \Longrightarrow & p >_{\mathsf{u}} r \vee q >_{\mathsf{u}} s \\
\Omega^*(p,q) \wedge p \geq_{\mathsf{u}} q & \Longrightarrow & p \geq_{\mathsf{u}} \lceil\sqrt{2^w}\rceil \\
\neg\Omega^*(p,q) \wedge p \geq_{\mathsf{u}} q & \Longrightarrow & q <_{\mathsf{u}} \lfloor\sqrt{2^w}\rfloor
\end{array}$$

where $\bar{0}p$ and $\bar{0}q$ stands for a zero-extension with at least one bit of p and q, respectively. Note that here $w = |p| = |q| > 1$, since for $w = 1$ multiplication overflow is impossible.

Variables can in some cases be resolved, producing constraints that are free of resolved variables.

Lemma 5 (Saturation Modulo Equalities).

$$\begin{array}{lll} ax+b=0 \wedge cx+d=0 & \Longrightarrow & ad-bc=0 \\ ax+b=0 \wedge c[x] & \Longrightarrow & c[-b\cdot a^{-1}] \quad \textit{if } a \textit{ is odd} \end{array}$$

where $c[x]$ may be any constraint containing x. Note that the multiplicative inverse a^{-1} of a in $\mathbb{Z}/2^w\mathbb{Z}$ exists if and only if a is odd.

Finally, let us define the *parity* of a bit-vector x as the largest number $i \in \{0,\dots,w\}$ such that 2^i divides x. The parity of a bit-vector can be constrained by a linear inequality, where $\mathrm{parity}(p) \geq i \Longleftrightarrow p2^{w-i}=0$ for $0 < i \leq w$.

Lemma 6 (Parity Saturation). *Parity inequalities can be used to constrain values of multipliers.*

$$\begin{array}{lll} p\cdot q=0 & \Longrightarrow & \mathrm{parity}(p)+\mathrm{parity}(q)\geq w \\ p\cdot q=1 & \Longrightarrow & \mathrm{parity}(p)=0 \\ p\cdot q=q & \Longrightarrow & \mathrm{parity}(p-1)+\mathrm{parity}(q)\geq w \\ \multicolumn{3}{l}{\mathrm{parity}(p\cdot q)=\min(w,\mathrm{parity}(p)+\mathrm{parity}(q))} \end{array}$$

6.2 Incremental Linearization

PolySAT includes incremental linearization rules for the cases where variables are 0, 1, -1, or powers of two. Note that our vocabulary of incremental linearization lemmas is considerably smaller than what is used for non-linear integer arithmetic [9], but it is also materially different as it operates over modular semantics of bit-vector operations. Notably, we do not include here inferences for deriving ordering constraints, such as $a > b \wedge c > 0 \implies ac > bc$, which holds for integers, but not for bit-vectors. Note that Lemma 3 includes ordering constraints, but only for the cases where relevant uses of multiplication do not overflow.

Lemma 7 (Incremental Linearization).

$$\begin{array}{lll} p=0 & \Longrightarrow & p\cdot q=0 \\ p=1 & \Longrightarrow & p\cdot q=q \\ p=-1 & \Longrightarrow & p\cdot q=-q \\ p=2^k & \Longrightarrow & p\cdot q=2^k q \quad (k=1,\dots,w-1) \\ p\cdot q=1 & \Longrightarrow & p=1 \vee \Omega^*(p,q) \\ p\cdot q=q & \Longrightarrow & p=1 \vee q=0 \vee \Omega^*(p,q) \end{array}$$

6.3 Bit-blasting Rules

As a final resort, PolySAT admits bit-blasting. A product $x := p\cdot q$ can be equivalently represented as $\sum_i 2^i p[i]q$. The other primitive operations (bit-wise *and*, bit-wise *or*, left shift, logical and arithmetic right shift) are unfolded using blasting as follows.

Lemma 8 **($x \coloneqq p \,\&\, q$).** *Bit-wise* and *"&" is handled using standard axioms, that fall back to bit-blasting at each index i if the basic algebraic properties hold, but x still does not evaluate to the bit-wise* and *of p, q.*

$$\begin{array}{lll} \top & \implies x \leq_{\mathsf{u}} p & \\ p = 0 & \implies x = 0 & \\ p = -1 & \implies x = q & \\ p = q & \implies x = p & \\ p[i] \land q[i] & \implies x[i] & \textit{for each } 0 \leq i < w \\ x[i] & \implies p[i] & \textit{for each } 0 \leq i < w \end{array}$$

Note that we do not list symmetric rules, e.g., $x \leq_{\mathsf{u}} q$.

Bit-wise *or* is handled analogously. For shift operations, we split on the value of the second argument. PolySAT also performs partial bit-blasting for multiplication overflow predicates. Details may be found in Appendix B.

7 Experiments

We evaluated our PolySAT prototype[3] against recent versions of several state-of-the-art SMT solvers on the following four benchmark sets: the category `QF_BV` from SMT-LIB [3] (release 2023, non-incremental); the BV2SMV benchmarks featuring large bit-widths [15]; 14 benchmarks from smart contract verification related to the Certora prover [1]; and a set of benchmarks from the Alive2 compiler verification project [22]. Note that the STP solver [16] does not support the logic `QF_UFBV` used by some of the Certora benchmarks.

Our experiments were performed on a TU Wien cluster, where each compute node contains two AMD Epyc 7502 processors, each of which has 32 CPU cores running at 2.5 GHz. Each compute node is equipped with 1008 GiB of physical memory that is split into eight memory nodes of 126 GiB each, with eight logical CPUs assigned to each node. We used `runexec` from the benchmarking framework BenchExec [5] to assign each benchmark process to a different CPU core and its corresponding memory node. Further, we used GNU Parallel [28] to schedule benchmark processes in parallel.

Our results are summarized in Table 1 and indicate that PolySAT is comparable to the other word-level approaches on the BV2SMV benchmark set, however in general, more work is needed. Concerning the Alive2 benchmarks that were solved by Yices2-mcsat but not by PolySAT, we found that in all but three cases Yices2-mcsat did not use any interval reasoning for conflicts/propagation; rather, Yices2-mcsat relied mostly on a fallback to bit-blasting. As PolySAT does not yet have such a fallback, this result suggests our bit-blasting rules (Section 6.3) alone are not enough.

[3] Available at `https://github.com/Z3Prover/z3/tree/poly`. This paper refers to commit `16fb86b636047fd79ad5827f768b6f26d8812948`. To select PolySAT for bit-vector solving, add the following options: `sat.smt=true tactic.default_tactic=smt smt.bv.solver=1`.

		SMT-LIB		BV2SMV		Smart Contracts		Alive2	
		sat	unsat	sat	unsat	sat	unsat	sat	unsat
Bit-blasting	Bitwuzla [25]	17 745	27 203	32	115	1	3	39	3 954
	cvc5 [2]	16 417	25 922	31	114	0	4	39	2 722
	STP [16]	17 462	27 011	24	115	-	-	39	2 893
	Yices2 [13]	17 589	26 600	24	107	0	3	39	1 519
	Z3 [24]	16 112	25 597	29	94	0	3	39	1 514
Word-lvl	cvc5-IntBlast [31]	11 251	24 376	32	64	1	9	5	1 047
	Yices2-mcsat [17]	14 155	22 396	24	101	1	4	23	2 562
	Z3-IntBlast	10 912	24 371	28	56	1	5	30	921
	Z3-PolySAT	7 297	20 080	28	63	0	3	0	21
	Total	46 191		192		14		12 951	

Table 1: Number of problems solved within 60 s for several benchmark sets. The upper five solvers are based on bit-blasting, while the lower four solvers use word-level techniques.

Nevertheless, PolySAT complements Z3 with word-level bit-vector reasoning. Our experimental analysis found that PolySAT solved 135 problems that Z3 did not solve and 404 problems that Z3-IntBlast did not solve (40 of which neither Z3 nor Z3-IntBlast solved). Further combinations of complementary approaches of word-level reasoning with bit-blasting is a promising direction to explore.

8 Conclusion

We introduced PolySAT, a general purpose word-level bit-vector solver, to overcome the scalability issue of bit-blasting over large bit-vectors. PolySAT integrates into CDCL(T)-based SMT solving, generalizes interval-based reasoning, and performs incremental linearization of constraints. PolySAT is implemented in the SMT solver Z3 and complements bit-vector reasoning in Z3.

Acknowledgements. We thank Mooly Sagiv and Alexander Nutz for thorough discussions on PolySAT applications. This work was partially supported by the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College SecInt, the FWF SFB project SpyCoDe F8504, the FWF ESPRIT grant 10.55776/ESP666, and the Amazon Research Award 2023 QuAT.

References

1. Albert, E., Grossman, S., Rinetzky, N., Rodríguez-Núñez, C., Rubio, A., Sagiv, M.: Taming Callbacks for Smart Contract Modularity. Proc. ACM Program. Lang. 4(OOPSLA), 1–30 (2020). https://doi.org/10.1145/3428277, https://doi.org/10.1145/3428277

2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Proc. of TACAS. pp. 415–442 (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
4. Bayardo, Jr., R.J., Schrag, R.: Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In: Proc. of AAAI and IAAI. pp. 203–208 (1997)
5. Beyer, D., Löwe, S., Wendler, P.: Reliable Benchmarking: Requirements and Solutions. J. on Software Tools for Technology Transfer **21**(1), 1–29 (2017)
6. Bjørner, N.S., Pichora, M.C.: Deciding Fixed and Non-fixed Size Bit-vectors. In: Proc. of TACAS. pp. 376–392 (1998). https://doi.org/10.1007/BFB0054184, https://doi.org/10.1007/BFb0054184
7. Bruttomesso, R., Sharygina, N.: A Scalable Decision Procedure for Fixed-width Bit-vectors. In: Proc. of ICCAD. pp. 13–20 (2009). https://doi.org/10.1145/1687399.1687403, https://doi.org/10.1145/1687399.1687403
8. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding Bit-Vector Arithmetic with Abstraction. In: Proc. of TACAS. pp. 358–372 (2007). https://doi.org/10.1007/978-3-540-71209-1_28, https://doi.org/10.1007/978-3-540-71209-1_28
9. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on Solving Nonlinear Integer Arithmetic with Incremental Linearization. In: Proc. of SAT. pp. 383–398 (2018). https://doi.org/10.1007/978-3-319-94144-8_23, https://doi.org/10.1007/978-3-319-94144-8_23
10. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Proc. of TACAS. pp. 93–107 (2013). https://doi.org/10.1007/978-3-642-36742-7_7
11. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Proc. of TACAS. pp. 168–176 (2004)
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a Theorem Prover for Program Checking. J. ACM **52**(3), 365–473 (2005). https://doi.org/10.1145/1066100.1066102, https://doi.org/10.1145/1066100.1066102
13. Dutertre, B.: Yices 2.2. In: Proc. of CAV. pp. 737–744 (2014). https://doi.org/10.1007/978-3-319-08867-9_49, https://doi.org/10.1007/978-3-319-08867-9_49
14. Fröhlich, A., Biere, A., Wintersteiger, C.M., Hamadi, Y.: Stochastic Local Search for Satisfiability Modulo Theories. In: Proc. of AAAI. pp. 1136–1143 (2015), http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9896
15. Fröhlich, A., Kovásznai, G., Biere, A.: Efficiently Solving Bit-Vector Problems Using Model Checkers. In: Proc. of Workshop on SMT. pp. 6–15 (2013), https://fmv.jku.at/bv2smv/
16. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Proc. of CAV. pp. 519–531 (2007). https://doi.org/10.1007/978-3-540-73368-3_52, https://doi.org/10.1007/978-3-540-73368-3_52
17. Graham-Lengrand, S., Jovanovic, D., Dutertre, B.: Solving Bitvectors with MCSAT: Explanations from Bits and Pieces. In: Proc. of IJCAR. pp. 103–121 (2020). https://doi.org/10.1007/978-3-030-51074-9_7, https://doi.org/10.1007/978-3-030-51074-9_7

18. Hadarean, L., Bansal, K., Jovanovic, D., Barrett, C.W., Tinelli, C.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: Proc. of CAV. LNCS, vol. 8559, pp. 680–695. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_45
19. Kovásznai, G., Fröhlich, A., Biere, A.: Complexity of Fixed-Size Bit-Vector Logics. Theory Comput. Syst. **59**(2), 323–376 (2016). https://doi.org/10.1007/s00224-015-9653-1, https://doi.org/10.1007/s00224-015-9653-1
20. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer (2008). https://doi.org/10.1007/978-3-540-74105-3
21. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints An Int. J. **21**(2), 223–250 (2016). https://doi.org/10.1007/S10601-015-9183-0, https://doi.org/10.1007/s10601-015-9183-0
22. Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: Bounded Translation Validation for LLVM. In: Proc. of PLDI. p. 65–79 (2021). https://doi.org/10.1145/3453483.3454030, https://doi.org/10.1145/3453483.3454030
23. Möller, M.O., Rueß, H.: Solving Bit-Vector Equations. In: Proc. of FMCAD. pp. 36–48 (1998). https://doi.org/10.1007/3-540-49519-3_4, https://doi.org/10.1007/3-540-49519-3_4
24. de Moura, L.M., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Proc. of TACAS. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
25. Niemetz, A., Preiner, M.: Bitwuzla. In: Proc. of CAV. pp. 3–17 (2023). https://doi.org/10.1007/978-3-031-37703-7_1, https://doi.org/10.1007/978-3-031-37703-7_1
26. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Proc. of CAV. pp. 587–595 (2018). https://doi.org/10.1007/978-3-319-96145-3_32
27. Silva, J.P.M., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999). https://doi.org/10.1109/12.769433
28. Tange, O.: GNU Parallel 20240122 ('Frederik X') (Jan 2024). https://doi.org/10.5281/zenodo.10558745, https://doi.org/10.5281/zenodo.10558745, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.
29. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and Extensible Equality Saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434304, https://doi.org/10.1145/3434304
30. Zeljic, A., Wintersteiger, C.M., Rümmer, P.: Deciding Bit-Vector Formulas with mcSAT. In: Proc. of SAT. pp. 249–266 (2016). https://doi.org/10.1007/978-3-319-40970-2_16, https://doi.org/10.1007/978-3-319-40970-2_16
31. Zohar, Y., Irfan, A., Mann, M., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-precise Reasoning via Int-blasting. In: Proc. of VMCAI. pp. 496–518 (2022). https://doi.org/10.1007/978-3-030-94583-1_24

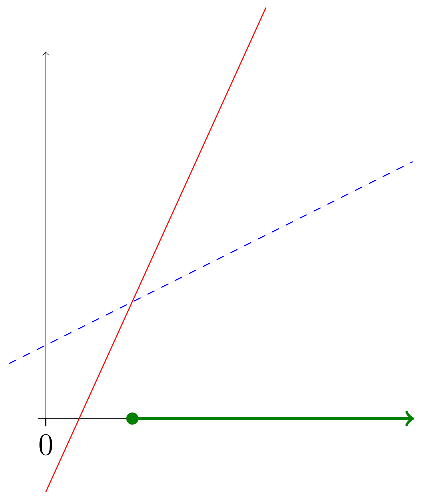
A Computing Intervals – Linear Inequality with Different Coefficients

This section extends the discussion in Section 5.2, providing further details on how interval bounds are analysed and computed.

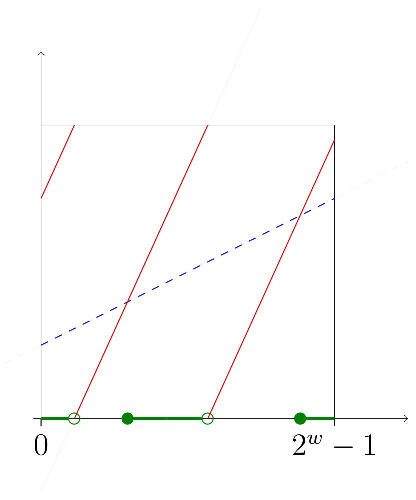
Linear inequalities with different coefficients Consider an inequality c of the form $px + q \leq_{\mathsf{u}} rx + s$ with $\widehat{p} \neq \widehat{r}$. Here, we need to find the largest x-interval around a sample point x_0 where c is satisfied. As Figure 4a shows for an example, the corresponding problem is easily solved over infinite domains, such as rationals, by computing the intersection point of the left- and right-hand side of the inequality. The interval extends from the intersection point towards infinity.

However, in modular arithmetic, the left-hand side and the right-hand side of c do not represent continuous lines; instead, they wrap around at 2^w as seen in Figure 4b. The intervals extend from an intersection point to the next wraparound point. We compute and return the interval containing x_0.

We note that PolySAT computes only the intersection/wraparound points nearest to x_0. In some configurations, the gap between one interval to the next (i.e., between the green lines in Figure 4b) does not contain an integer, which means the obtained x-interval is not maximal. This method works best when the coefficients $\widehat{p}$ and $\widehat{q}$ of x are near 0 or 2^w.



(a) In $\mathbb{Q}$.



(b) In $\mathbb{Z}/2^w\mathbb{Z}$.

Fig. 4: Example for extracting intervals from an inequality constraint $px + q \leq_{\mathsf{u}} rx + s$ with different variable coefficients. The blue dashed line plots $\widehat{p}x + \widehat{q}$, and the red continuous line is $\widehat{r}x + \widehat{s}$.

Computing intersection and wraparound points In order to work out the above intuition more precisely, consider the inequality c of the form $px + q \leq_{\mathsf{u}} rx + s$ with $p, q, r, s \in \mathbb{Z}/2^w\mathbb{Z}$ such that $p \neq 0$, $r \neq 0$ and $p \neq r$. Let $x_0 \in \mathbb{Z}/2^w\mathbb{Z}$ be a sample value that violates the constraint, i.e., such that $(px_0 + q) \bmod 2^w > (rx_0 + s) \bmod 2^w$ (to avoid confusion, we write "mod" operations in this section explicitly).

The goal is to find a maximal x-interval around x_0 whose elements all violate the constraint, i.e., we want to find the minimal x_l and the maximal x_h such that $x_l \leq x_0 \leq x_h$ and $(px + q) \bmod 2^w > (rx + s) \bmod 2^w$ for all $x \in [x_l; x_h]$.

In the following, we explain our method for extracting such intervals, however, we cannot yet guarantee to obtain a maximal interval in all cases. As illustrated in Figure 4, we extrapolate the left-hand side (LHS) and the right-hand side (RHS) of the constraint using standard arithmetic until the next overflow point, and extract the maximal interval that can be obtained without overflow.

Let us define the abbreviations $a \coloneqq (px_0+q) \bmod 2^w$ and $b \coloneqq (rx_0+s) \bmod 2^w$. From now on, we view p, q, r, s, a, b as values over the rationals $\mathbb{Q}$ by choosing the representative in the interval $[0; 2^w[$.

To compute a safe upper bound $x_h = x_0 + \delta_h$, we find the maximal $\delta_h \in \mathbb{Z}$ satisfying the following conditions:

- $\delta_h \geq 0$, i.e., it should be an *upper* bound,
- $\forall x.(0 \leq x \leq \delta_h \to 2^w > a + px > b + rx \geq 0)$, i.e., the LHS and RHS do not overflow within the interval and the constraint is violated for all values,
- $x_0 + \delta_h < 2^w$, i.e., the upper bound does not overflow.

After several transformations, we obtain the formula

$$\delta_h = \min\left(\left\{2^w - x_0, \left\lceil\frac{2^w - a}{p}\right\rceil\right\} \cup \left\{\left\lceil\frac{a-b}{r-p}\right\rceil \;\middle|\; r > p\right\}\right) - 1.$$

Similarly, we obtain a safe lower bound $x_l = x_0 - \delta_l$, by finding the maximal $\delta_l \in \mathbb{Z}$ such that:

- $\delta_l \geq 0$ (it should be a *lower* bound),
- $\delta_l \leq x_0$ (lower bound does not overflow),
- $\forall x.(0 \leq x \leq \delta_l \to 2^w > a - px > b - rx \geq 0)$.

A sequence of transformations leads us to the formula

$$\delta_l = \min\left(\left\{x_0 + 1, \left\lceil\frac{b+1}{r}\right\rceil\right\} \cup \left\{\left\lceil\frac{a-b}{p-r}\right\rceil \;\middle|\; p > r\right\}\right) - 1.$$

Remark 1. At the beginning of this section, we embedded the coefficients p, q from $\mathbb{Z}/2^w\mathbb{Z}$ into $\mathbb{Q}$ by choosing the representative in the interval $[0; 2^w[$. However, whenever p or q is a large value near 2^w we may obtain better bounds by interpreting them as negative numbers, i.e., choose the representative in the interval $[-2^w; 0[$ instead. To obtain a uniform formula, we can simply plug in $p - 2^w$ and $q - 2^w$ for p and q (or just one of them), respectively, in the formulas above. In total, this gives us four different ways to estimate each bound. Since each of these computations finds a safe bound, we choose the best among them.

Strict inequalities Finally, if we want to compute such bounds for a strict inequality $px + q <_{\mathsf{u}} rx + s$, we only have to change the strictness of one inequality in our initial conditions, i.e., replace $a \pm px > b \pm rx$ by $a \pm px \geq b \pm rx$. In the final formulas, this manifests as replacing $a - b$ in the numerator by $a - b + 1$; otherwise, the results are unchanged.

B Additional Lemmas

Lemma 9. *Extended version of Lemma 3.*

$$
\begin{array}{llllll}
px <_{\mathsf{u}} qx & \implies & p \neq q & & & \\
px <_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(p, x) & \vee\, p <_{\mathsf{u}} q & & \\
px <_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(-q, x) & \vee\, p <_{\mathsf{u}} q & & \\
px <_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(q, -x) & \vee\, p >_{\mathsf{u}} q & \vee\, p = 0 & \\
px <_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(-p, -x) & \vee\, p >_{\mathsf{u}} q & \vee\, p = 0 & \\
px \leq_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(p, x) & \vee\, p \leq_{\mathsf{u}} q & \vee\, x = 0 & \\
px \leq_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(-q, x) & \vee\, p \leq_{\mathsf{u}} q & \vee\, x = 0 & \vee\, q = 0 \\
px \leq_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(q, -x) & \vee\, p \geq_{\mathsf{u}} q & \vee\, x = 0 & \vee\, p = 0 \\
px \leq_{\mathsf{u}} qx & \implies & \mathit{\Omega}^*(-p, -x) & \vee\, p \geq_{\mathsf{u}} q & \vee\, x = 0 & \vee\, p = 0 \\
px + s \leq_{\mathsf{u}} q & \implies & \mathit{\Omega}^*(p, x) & \vee\, \mathit{\Omega}^+(px, s) & \vee\, pr \leq_{\mathsf{u}} q & \vee\, x <_{\mathsf{u}} r \\
p \leq_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, x) & \vee\, pq \leq_{\mathsf{u}} r & & \\
p \leq_{\mathsf{u}} x \wedge qx <_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, x) & \vee\, pq <_{\mathsf{u}} r & & \\
p <_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, x) & \vee\, pq <_{\mathsf{u}} r & \vee\, q = 0 & \\
p <_{\mathsf{u}} x \wedge qx \leq_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, x) & \vee\, pq <_{\mathsf{u}} r & \vee\, r = 0 & \\
p \leq_{\mathsf{u}} qx \wedge x \leq_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, r) & \vee\, p \leq_{\mathsf{u}} qr & & \\
p <_{\mathsf{u}} qx \wedge x \leq_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, r) & \vee\, p <_{\mathsf{u}} qr & & \\
p \leq_{\mathsf{u}} qx \wedge x <_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, r) & \vee\, p <_{\mathsf{u}} qr & \vee\, p = 0 & \\
p \leq_{\mathsf{u}} qx \wedge x <_{\mathsf{u}} r & \implies & \mathit{\Omega}^*(q, r) & \vee\, p <_{\mathsf{u}} qr & \vee\, q = 0 &
\end{array}
$$

Note that these rules do not require $x \notin p, q, r, s$, so they can be applied even when the degree of x is larger than 1.

Lemma 10 ($x \coloneqq p \mid q$). *Bit-wise* or *is handled similarly as bit-wise* and.

$$
\begin{array}{lll}
\top & \implies & x \geq_{\mathsf{u}} p \\
p = 0 & \implies & x = q \\
p = -1 & \implies & x = -1 \\
p = q & \implies & x = p \\
p[i] & \implies & x[i] \quad \textit{for each } 0 \leq i < w \\
x[i] & \implies & p[i] \vee q[i] \ \textit{for each } 0 \leq i < w
\end{array}
$$

Lemma 11 ($x \coloneqq p \ll q$). *For shift operations, we split on the second argument.*

$$
\begin{array}{lll}
q \geq_{\mathsf{u}} w & \implies & x = 0 \\
q = 0 & \implies & x = p \\
q = i & \implies & x = 2^i p
\end{array}
$$

for all constants i such that $0 < i < w$.

Lemma 12 **($x \coloneqq p \gg q$).** *Logical right-shift is analogous.*

$$\begin{array}{lll} q \geq_{\mathsf{u}} w & \Longrightarrow & x = 0 \\ q = 0 & \Longrightarrow & x = p \\ q = i & \Longrightarrow & 2^i x \leq_{\mathsf{u}} p \leq_{\mathsf{u}} 2^i x + 2^i - 1 \land x <_{\mathsf{u}} 2^{w-i} \end{array}$$

for all constants i such that $0 < i < w$.

Lemma 13 **($x \coloneqq p \gg_{\mathsf{a}} q$).** *The arithmetic right-shift must take into account the sign bit $p[w-1]$.*

$$\begin{array}{lll} p[w-1] \land q \geq_{\mathsf{u}} w & \Longrightarrow & x = -1 \\ \neg p[w-1] \land q \geq_{\mathsf{u}} w & \Longrightarrow & x = 0 \\ q \geq_{\mathsf{u}} w & \Longrightarrow & x + 1 \leq_{\mathsf{u}} 1 \\ q = 0 & \Longrightarrow & x = p \\ q = i & \Longrightarrow & 2^i x \leq_{\mathsf{u}} p \leq_{\mathsf{u}} 2^i x + 2^i - 1 \\ p[w-1] \land q = i & \Longrightarrow & x \geq_{\mathsf{u}} 2^w - 2^{w-i-1} \\ \neg p[w-1] \land q = i & \Longrightarrow & x <_{\mathsf{u}} 2^{w-i-1} \end{array}$$

for all constants i such that $0 < i < w$.

PolySAT performs partial bit-blasting for multiplication overflow predicates. It is based on partitioning the conditions for overflow by using the sum of most significant bits into three cases. To describe these, first let us define the shorthand $\mathrm{msb}(p)$ for the one-based index of the most significant bit of p. For example, $\mathrm{msb}(1) = 1, \mathrm{msb}(2) = 2$. It can be defined indirectly using the equivalence $\mathrm{msb}(p) \geq i \iff p \geq_{\mathsf{u}} 2^{i-1}$ for $1 \leq i \leq w$. The cases are

$$\begin{array}{lll} \mathrm{msb}(p) + \mathrm{msb}(q) \geq w + 2 & \Longrightarrow & \Omega^*(p, q) \\ \mathrm{msb}(p) + \mathrm{msb}(q) \leq w & \Longrightarrow & \neg\Omega^*(p, q) \\ \mathrm{msb}(p) + \mathrm{msb}(q) = w + 1 & \Longrightarrow & \big(\Omega^*(p, q) \iff (0p) \cdot (0q) \geq_{\mathsf{u}} 2^w\big), \end{array}$$

where $0p$ and $0q$ stand for the zero-extension by a single bit of p and q, respectively. In other words, when the most significant bits add up to w, multiplication overflow affects exactly one additional bit, so it suffices to extend p and q by a single bit to determine overflow.