Word Level Property Directed Reachability

Hari Govind V K
University of Waterloo, Waterloo
Canada
hgvk94@gmail.com

Grigory Fedyukovich
Florida State University, Tallahassee
USA
grigory@cs.fsu.edu

Arie Gurfinkel
University of Waterloo, Waterloo
Canada
arie.gurfinkel@uwaterloo.ca

ABSTRACT

Verification approaches based on constraint solvers are successfully applied in firmware and other low-level code that interfaces with hardware. While for proving safety of gate-level sequential circuits, it often suffices to bit-blast and reduce to SAT-based IC3 or *Property Directed Reachability* (IC3/PDR), for handling machine-level instructions that perform arithmetic and data manipulation operations, word-level reasoning should be conducted. However, because of poor support for interpolation and quantifier elimination in the theory of bit-vectors (BV), previous attempts to lift IC3/PDR to word level required integrating it into an external abstraction-refinement loop. Aiming to reach more scalable bit-precise verification, we propose to bring useful insights from PDR-based verification algorithms used in software. In particular, instead of using bit-blasting to eliminate quantifiers from BV-formulas, we present a less expensive method for iterative approximate quantifier elimination in BV. It naturally supports all bit-operators and can be optimized further by applying rules inspired by modular linear arithmetic. Finally, we leverage recent techniques on learning inductive invariants based on explicit global guidance, thus allowing the approach to bypass interpolation. Our implementation on top of Spacer, a PDR-based verifier shows that such a word-level PDR is promising and can be more effective than state-of-the-art.

KEYWORDS

SMT, QF_BV, quantifier elimination, automated verification, inductive invariants

ACM Reference Format:
Hari Govind V K, Grigory Fedyukovich, and Arie Gurfinkel. 2020. Word Level Property Directed Reachability. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415708

1 INTRODUCTION

Firmware implements key hardware functions, accelerates maintenance, and enables to deploy bugfixes after the system is released. Such low-level software often performs complex arithmetic operators, such as adders, multipliers, and variable shifters. Its safety verification is challenging, especially because the tools are expected to deal with bit-precise and word-level models expressed in a rich theory of quantifier-free bit-vectors (QF_BV, or BV in short).

Automated verification makes extensive use of decision procedures for different theories in first-order logic. Whether a safety invariant or a counterexample is going to be found in software programs or in hardware designs, their first-order logic encodings are processed by solvers for satisfiability (SAT) [4, 6, 10, 13, 30] or satisfiability modulo theories (SMT) [17, 19, 27–29, 34]. Existing verification frameworks are designed such that plugging a theory solver does not require changing the verification workflow. Thus, to meet the scalability criteria, theory solvers should have efficient solving heuristics, rigorous support for quantified reasoning, and interpolation. However, unfortunately, BV is a known obstacle for verification approaches – due to the lack of proper support for quantified reasoning and interpolation, solvers either bit-blast the verification conditions and use SAT, or implement additional abstraction-refinement loops [7, 17, 25].

In this paper, we present a novel instantiation of the well known *Property Directed Reachability* (IC3/PDR) [4, 10] paradigm that relies on quantifier elimination *at word level* and does not require an additional abstraction-refinement loop. It incrementally strengthens a given safety property until it either becomes inductive, or a counterexample is found. Specifically, we built on top of the Spacer algorithm [23] which maintains over-approximations and under-approximations of sets of reachable states. The former is used to block spurious counterexamples, and the latter to create predecessors to bad states, called proof obligations and block them.

In hardware, proof obligations are computed by SAT and generalized by ternary simulation. This does not work for infinite theories, like linear integer arithmetic (LIA), which prompted researchers to invent *Model-Based Projection* (MBP) [3] that under-approximates existential quantifiers. While for any finite theory, like QF_BV, it is possible to use a model to generate a predecessor, this is extremely ineffective [31]. In particular, a problem that is trivial over LIA, becomes very difficult over QF_BV without MBP. For BV, defining an MBP is difficult mainly because the language of bit-operators (such as shifts, signed comparison, bitwise and/or) is way richer than the one of a lightweight theory. Finally, even if restricted to only an arithmetic fragment, the LIA-rules do not work because of the presence of overflow. We therefore propose first to create a modular-arithmetic MBP following a set of predetermined *rewriting rules* and then to iteratively repair it via a novel *lazy MBP algorithm*. The combination of both allows us to eliminate all BV operators.

Once proof obligations are proven to be unreachable, model checking algorithms usually employ only local interpolation techniques to generalize proofs of unreachability. However, there are no interpolation strategies for BV, that would be always effective. Therefore, we instantiate the global guidance rules recently proposed in [24] and obtain an effective alternative to interpolation.

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415708

```
uint32_t x = 1; y = 1;
while (1) {
  x = x + 2 * nd();
  y = y + 2 * nd();
  assert(x + y != 1);
}
```

Figure 1: An example program in C-like syntax. The variables x and y are unsigned integers. nd() is an external function that returns a non-deterministically chosen value.

We have implemented the technique, called GSpacerbv, on top of Spacer [23], a solver for Constrained Horn clauses (CHC). Spacer is part of the Z3 SMT solver [9] and is used to solve verification problems of arbitrary structure (i.e., not only transition systems, but also software with function calls, nested loops, etc). We compared GSpacerbv against state-of-the-art solver Eldarica [18] (the only other CHC solver that supports BV) and experimented on a range of benchmarks. On hardware benchmarks, GSpacerbv outperforms Eldarica and is competitive with Spacer. On software verification tasks, GSpacerbv exhibits a competitive performance w.r.t. Spacer and is able to prove more instances safe.

2 MOTIVATING EXAMPLE

Consider verifying an assertion in an example program[1] in Fig. 1. The program is safe because the sum $x+y$ is always even. To discover this inductive invariant, IC3/PDR algorithms like Spacer (see Sec. 7), computes and blocks predecessors to bad states, i.e., states from which the assertion is violated. In our example, the immediate bad states are $x + y = 1$, and their possible predecessors are: $(x = 1, y = 0)$, $(x = -1, y = -4)$, $(x + y = 1)$, and $(x + y = -3)$, where $(-\cdot)$ is the additive inverse (recall that all numbers in the example are unsigned). Clearly, the last two predecessors are more useful than the first two. Our first contribution is an algorithm that uses BV arithmetic to generate such good predecessors (see Sec. 5). Our second contribution is a *global guidance* technique that allows model checking algorithms to generalize from predecessors of the form $x + y = 1, x + y = -3, \ldots$, to $x + y \texttt{ mod } 2 = 1$ (see Sec. 7.1).

3 BACKGROUND AND NOTATION

3.1 Preliminaries

Logic. We work on sorted First Order Logic modulo theory of Fixed-Size Bit-Vectors (BV). Our signature Σ contains an infinite number of *constant* symbols (zero-ary, uninterpreted functions) denoted with $x, y, \ldots$ and *numerals* (zero-ary, interpreted functions) $1, 2, \ldots$. A *term* is a constant, numeral, variable, or a function applied to terms. An *atom* is a predicate applied to terms, a *literal* is either an atom or the negation of an atom, and a *cube* is a conjunction of literals. A *formula* is a Boolean combination of literals, compiled w.r.t. the following grammar.

$$
\begin{aligned}
\mathit{term} ::=& \mathit{con} \mid \mathit{num} \mid \mathit{var} \mid \mathit{term} \times \mathit{term} \mid \mathit{term} \texttt{ div } \mathit{num} \mid \\
& \mathit{term} + \mathit{term} \mid \mathit{term} - \mathit{term} \mid \\
& \mathit{bvop}(\mathit{term}) \mid \ldots \mid \mathit{bvop}(\mathit{term}, \mathit{term}, \ldots)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fla} ::=& \mathit{term} < \mathit{term} \mid \mathit{term} \leq \mathit{term} \mid \mathit{term} = \mathit{term} \mid \mathit{term} \neq \mathit{term} \mid \\
& \neg \mathit{fla} \mid \mathit{fla} \land \mathit{fla} \mid \mathit{fla} \lor \mathit{fla} \mid \mathit{fla} \Rightarrow \mathit{fla}
\end{aligned}
$$

The grammar allows multiplying and dividing terms, as well as addition and subtraction. Formulas are built using equalities, (unsigned) inequalities[2], and Boolean connectives. The predicates and functions have the usual meaning, e.g., `div` is the integer division (and e.g., $5 \texttt{ div } 3 = 1$). The grammar supports bit operators (*bvop*) such as bit shifts, bitwise and/or, extracts, etc.

We sometimes treat a formula as the set of all its satisfying assignments. We use $\mathit{const}(f)$ to mean the set of all uninterpreted constants in f and write $f(X)$ to emphasize that $\mathit{const}(f) \subseteq X$. Unless otherwise stated, all our formulas are *closed*.

MBP. Given a cube φ with constants X, a subset of its constants $X_s \subseteq X$, and a model $M \models \varphi$, Model Based Projection (MBP) computes a model preserving, closed, under-approximation of the quantified formula $\exists X_s \cdot \varphi$. That is, $\mathit{MBP}(X_s, \varphi, M)$ is a cube $\mathcal{P}$ such that $\mathcal{P} \Rightarrow \exists X_s \cdot \varphi$, $X_s \cap \mathit{const}(\mathcal{P}) = \emptyset$, and $M \models \mathcal{P}$. An MBP is finite if the range of MBP is finite after fixing φ and X_s. Every theory that admits quantifier elimination also admits a finite MBP. We skip mentioning X_s and write $\mathit{MBP}(\varphi, M)$ when the constants that are eliminated are obvious from the context. The notion of MBP can be lifted to an arbitrary formula ψ (which is not necessarily a cube) by considering a model M of ψ, conjoining literals of ψ that are evaluated to true by M and using the notion of MBP for the resulting cube.

Interpolation. Given an unsatisfiable formula $A \land B$ an interpolant is a formula I such that $A \Rightarrow I$, $I \land B \Rightarrow \bot$ and $\mathit{const}(I) \subseteq \mathit{const}(A) \cap \mathit{const}(B)$. Intuitively, interpolants summarize the reason for unsatisfiability of $A \land B$. Various theories have theory specific interpolation strategies which are quite effective in practice. However, there is a lack of good interpolation strategies for BV. Currently, UNSAT cores are used as interpolants in BV.

Safety. A transition system is a three tuple $\langle X, \mathit{Init}(X), \mathit{Tr}(X, X')\rangle$ where X is the set of constants that represent the state of the system (called state variables), $\mathit{Init}(X)$ is a formula representing the set of initial states of the system, and $\mathit{Tr}(X, X')$ is the transition relation. When writing a state formula, we skip the state variables when they are obvious from the context. We use primed state variables and formulas to represent state variables and formulas in the post-state. A transition system T is *safe upto a depth d*, relative to a set of bad states *Bad*, if there are no counterexamples of depth less than d. That is, there are no sequence of states $s_0, s_1, s_2, \ldots, s_n$ such that $n \leq d$ and $s_0 \in \mathit{Init}$, $\forall 0 \leq i < n \cdot \{s_i, s'_{i+1}\} \in \mathit{Tr}$ and $s_n \in \mathit{Bad}$. The transition system is safe if there are no counterexamples of any depth. A *safe inductive invariant* is a certificate for safety. A formula Inv is a safe inductive invariant if $\mathit{Init} \Rightarrow \mathit{Inv}$, $\mathit{Inv} \land \mathit{Tr} \Rightarrow \mathit{Inv}'$, and $\mathit{Inv} \Rightarrow \neg \mathit{Bad}$. Throughout the paper, we use invariants to mean safe inductive invariants.

3.2 Modular linear arithmetic

While reasoning in terms of bitvectors, it is often convenient to represent them as integers. Let *num* from the grammar in the previous subsection belong to a set for integer numbers over a fixed bitwidth n, $\mathbb{Z}_{2^n-1} = \{0, 1, \ldots, 2^n - 1\}$.

[1] Adopted from https://github.com/sosy-lab/sv-benchmarks/blob/master/c/bitvector/jain_2-1.c.

[2] Signed comparison $<_s$, $\leq_s$ is defined via unsigned in the next subsection.

Algorithm 1: Rew(ψ, x, M, R): Model-based rewriting

In: Cube ψ, constant x, model M such that $M \models \psi$, set of rules $R = \{\langle prs, concl\rangle_i\}$
Out: Formula φ, such that $\varphi \Rightarrow \psi$, and either $\varphi = false$ or $M \models \varphi$

```
1  if ¬contains(ψ, x) or pattern-match(ψ, term < num × x) or
     pattern-match(ψ, num × x ≤ term) then
2      return ψ
3  for each ⟨prs, concl⟩ ∈ R do
4      if pattern-match(ψ, concl) then
5          res ← true
6          φ ← true
7          for each p ∈ prs do
8              q ← rewrite (p, ψ)
9              q ← Rew (q, x, M, R)
10             if M ⊨ q then
11                 φ ← φ ∧ q
12             else
13                 res ← false
14         if res then return φ
15 return false
```

Note that several essential predicates and functions are not explicit in the grammar, but can be expressed using grammar's predicates/functions. We sometimes use them in the text, assuming the following transformations:

- *signed inequalities*:

$$a \le_s b \stackrel{\text{def}}{=} ite((a < 2^{n-1} \wedge b < 2^{n-1}) \vee (2^{n-1} \le a \wedge 2^{n-1} \le b),$$
$$a \le b, (2^{n-1} \le a \wedge b \le 2^{n-1})),$$

- *divisibility constraint*:

$$(a \mid b) \stackrel{\text{def}}{=} (a = 0) \vee ((a-1) \ \mathtt{div} \ b) < a \ \mathtt{div} \ b),$$

- *remainder*: $a \ \mathtt{mod} \ b \stackrel{\text{def}}{=} a - (a \ \mathtt{div} \ b) * b$.

In the next two sections we present our novel technique to producing an MBP for modular arithmetic, a part of the signature of BV, and in Sect. 6 we show how it can be used to produce an MBP even while dealing with non-linear BV formulas.

4 NORMALIZATION RULES FOR MODULAR ARITHMETIC

In order to produce an MBP for modular arithmetic, we have to rewrite the formula in a normal form. Similar to the Linear Integer Arithmetic (LIA) case, we wish to move all terms containing a variable to be eliminated to the left side of a formula, and everything else to the right side. However, it turns out to be difficult in general because of the overflow. Our core insight is to use the model M to identify the cases when the BV operations behave as their arithmetic counter-parts. In this section, we present the rules that drive the process of normalization.

Fig. 2 gives our rewrite rules for BV arithmetic. Each rule $\langle prs, concl\rangle$ consists of a set of premises prs (in above case, two) and a conclusion. Let X be a set of constants $const(concl) \cup \bigcup_{p \in prs} const(p)$. We claim that for all the rules, it holds that $\forall X \cdot \bigwedge_{p \in prs} p(X) \Rightarrow concl(X)$, i.e., the conjunction of premises always implies the conclusion[3]. The premises of a rule need to be explicitly checked and conjoined to yield a normalized formula. Finally and most importantly, when applying some rules, we have to check that the rewritten premises are satisfied by the given model. This way, whenever the rewriting is done, the result is guaranteed to be satisfied by the given model.

Intuitively, rules $add_1, \ldots, add_7$ enable moving an x-free operand of + from the left to right side of an inequality. The direction of inequality and possible overflow make them different. We prove that these rules are complete i.e., given a formula $t(x) + y \le z$ or $t(x) + y \ge z$, we would always be able to rewrite it to a form with $t(x)$ as the sole term on one side of the inequality.

Alg. 1 gives pseudocode of a recursive rewriter that is parametrized by a set of rules. Given an input formula ψ, the result of the rewriter is a formula φ, such that: (1) φ is satisfied by the model M, i.e., $M \models \varphi$, (2) φ is conjunctive, (3) each of its conjuncts either has the form $term < num \times x$ or $num \times x \le term$, or is x-free, and (4) $\varphi \Rightarrow \psi$. Thus, if the given formula ψ meets (1), (2), and (3), the algorithm returns ψ (line 2). Otherwise, it recurses and tries to apply any rule (by rewriting[4] and checking the consistency of all premises with M) and proceeds to trying the next rule if previous ones are not applicable. If it is impossible to get a normalized under-approximation after all rewriting attempts, the algorithm returns *false*. Note also that the algorithm allows specifying custom rules and adjusting the order of them by the user – if several rules are applicable and result in different normalizations leveraging these differences might lead to useful heuristics (which are mot the focus of this paper and left for future work).

Example 4.1. Assume bit-width of 8. Let us normalize inequality $\psi \equiv (c - x) + y \le z$ w.r.t. variable x and a model $M \equiv \{x \mapsto 9, y \mapsto 255, z \mapsto 80, c \mapsto 84\}$. It is easy to see that the left side of the inequality overflows (i.e., $(84-9)+255$ is greater than the maximal number that can be represented with bit-width 8). Alg. 1 produces the following graph from applications of rules and premises/conclusions.

$$\dfrac{\dfrac{\dfrac{x \ge c+y-z}{-x \le z-y-c}\,[inv] \qquad \dfrac{x \le c}{-c \le -x}\,[inv]}{c-x \le z-y}\,[add_2] \qquad \dfrac{\dfrac{x \le c+y}{-y-c \le -x}\,[inv] \qquad -y \le c-1}{-y \le c-x}\,[add_4]}{(c-x)+y \le z}\,[add_2]$$

In order to create this graph, the algorithm begins with ψ (which will appear as the root). Each incoming edge represents a successfully applied rule and connects the conclusion to its premise (specified in the corresponding rule), and the formulas on the leaves, when conjoined (1) are satisfied by M and (2) constitute an implicant of ψ. In particular, in order to construct an incoming edge to the root, we can either apply $[add_1]$, $[add_2]$, or $[add_3]$. To decide which one, we check if premises are satisfied by M: for $[add_1]$, $M(y \le z) = 255 \le 80 = false$, but for $[add_2]$, $M(-y \le c - x) = 1 \le (84-9) = true$ and $M(c - x \le z - y) = 84 - 9 \le 80 = true$. Then, the process continues onward, by applying rules to premises of $[add_2]$.

[3] We proved the validity of all these rules using the Z3 SMT solver for all bit-widths up to 128. Our automated script took around fifteen minutes. Note that such proving needs to be done for any new rules that the user wants to add. All our rules can be found at https://hgvk94.github.io/gspacerbv/.

[4] We use pattern-match and rewrite in pseudocode in the usual sense and implement them by unification and substitution, respectively.

$$\frac{t(x) \le z - y \quad y \le z}{t(x) + y \le z}\,[add_1] \qquad \frac{t(x) \le z - y \quad -y \le t(x)}{t(x) + y \le z}\,[add_2] \qquad \frac{-y \le t(x) \quad y \le z \quad y \ne 0}{t(x) + y \le z}\,[add_3] \qquad \frac{t(x) \ge z - y \quad z \le y - 1}{t(x) + y \ge z}\,[add_4]$$

$$\frac{t(x) \ge z - y \quad t(x) \le -y - 1 \quad y \ne 0}{t(x) + y \ge z}\,[add_5] \qquad \frac{y = 0 \quad z \le t(x)}{t(x) + y \ge z}\,[add_6] \qquad \frac{y \ne 0 \quad z \le y - 1 \quad x \le -y - 1}{t(x) + y \ge z}\,[add_7]$$

$$\frac{y \le t_2(x) - t_1(x) \quad t_1(x) \le t_2(x)}{t_1(x) + y \le t_2(x)}\,[bothx_1] \qquad \frac{y \le t_2(x) - t_1(x) \quad -t_1(x) \le y}{t_1(x) + y \le t_2(x)}\,[bothx_2] \qquad \frac{-t_1(x) \le y \quad t_1(x) \le t_2(x) \quad t_1(x) \ne 0}{t_1(x) + y \le t_2(x)}\,[bothx_3]$$

$$\frac{a \le b \; b \le a}{a = b}\,[eq] \qquad \frac{a < b}{a \ne b} \quad \frac{a > b}{a \ne b}\,[neq] \qquad \frac{b \le a - 1 \quad 1 \le a}{\neg(a \le b)}\,[nule] \qquad \frac{-y \le t(x)}{-t(x) \le y} \quad \frac{t(x) \le -y}{y \le -t(x)}\,[inv] \qquad \frac{x \le \frac{2^n k_2}{k_1}}{k_1 x \le k_2 x}\,[bothx_4]$$

Figure 2: Rewrite rules for BV arithmetic. Terms $t_1(x)$, $t_2(x)$, and $t(x)$ contain constant x. Terms y and z do not contain x. Terms a and b may or may not contain x. Rules add_1 to add_7 rewrite unsigned inequalities so that $t(x)$ is the sole term on one side of the inequality. Rules $bothx_1$ to $bothx_4$ rewrite inequalities that contain x on both sides. Rules inv remove the negation of the x term.

After the normalization graph is constructed, the normalized under-approximation $\psi = x \ge c + y - z \wedge x \le c \wedge x \le c + y \wedge -y \le c - 1$ can be used to construct MBP. Note that if two or more rules are applicable, our algorithm picks the smallest with respect to a some predetermined order. □

The success in the proposed normalization directly affects our MBP algorithm for modular arithmetic (see Sect. 5). The literals that cannot be normalized – either because they syntactically do not fit the arithmetic fragment of our grammar, or because the normalization graph cannot be constructed with respect to the given model – are left in the cube untouched. Our lazy MBP algorithm (see Sect. 6) will take care of them first, by substitution of the values from the model, and 2) iterative filtering unnecessary literals. But since the lazy MBP is in general more expensive than the syntactic rewriting presented in this section, we are interested in populating our rewriting system with more diverse rules.

5 DEFINING MBP$_{\mathbb{Z}}$

In this section, we define our procedure, MBP$_{\mathbb{Z}}$, for Model-Based Projection over an arithmetic signature of BV. The procedure described here is lifted to the full signature of BV in Sect. 6. The input to the procedure is a formula φ of the form $\psi \wedge f(x)$, where ψ is x-free and $f(x)$ meets the grammar from Sect. 3.2. We assume that φ is satisfiable, and that a model M of φ is available. The constant being eliminated is x. To eliminate multiple constants, the procedure is applied sequentially.

In the following, we assume that all numeric operators are over a fixed bit-width n. For simplicity of the presentation, we use arbitrary precision arithmetic to check for overflow conditions. The basic idea of MBP$_{\mathbb{Z}}$ is to use the model M to identify the cases when the BV operations behave as their arithmetic counter-parts and adapt the rules from the MBP for Linear Integer Arithmetic (LIA) (e.g., [3]). However, to our knowledge, the rules that we use are new, even when restricted to LIA. For example, we prefer integer division (`div`) to divisibility constraints.

Before applying the rewrite rules for MBP, we ensure that the normalization process from Sect. 4 has succeeded. On top of that, we rely on additional normalization rules that extract a cube, where x can occur only in literals of the form $a < f(x)$ or $f(x) \le b$, where $f(x)$ is a term containing only one constant x, and a, b are x-free.

Note that we can normalize literals of form $a \le A$ and $B < b$ as follows:

- $a \le A$ is equivalent to $a = 0 \vee a - 1 < A$, and
- $B < b$ is equivalent to $b \ne 0 \wedge B \le b - 1$,

where we use the model to pick one of the disjuncts in the first case.

In the rest of the section, we present MBP$_{\mathbb{Z}}$ as a series of rewrite rules, based on the form of $f(x)$, with side-conditions, and describe them as they are being presented:

- $\text{MBP}_{\mathbb{Z}}(M, \psi \wedge \bigwedge_i (\beta_j \times x \le b_i)) \stackrel{\text{def}}{=} \psi$. Note that the model M is not used. There always exists a value $x \mapsto 0$, making $\varphi \equiv \bigwedge_i 0 \le b_i$, and since each b_i cannot be negative, φ is true.
- $\text{MBP}_{\mathbb{Z}}\left(M, \psi \wedge \left(\bigwedge_i a_i < \alpha_i \times x\right) \wedge \left(\bigwedge_j \beta_j \times x \le b_j\right)\right) \stackrel{\text{def}}{=} \psi \wedge$
  $(a_L \times (\text{LCM}\ \mathtt{div}\ \alpha_L)\ \mathtt{div}\ \text{LCM}) < (b_U \times (\text{LCM}\ \mathtt{div}\ \beta_U)\ \mathtt{div}\ \text{LCM}) \wedge$
  $\bigwedge_i a_i \le (2^n{-}1)\ \mathtt{div}\ (\text{LCM}\ \mathtt{div}\ \alpha_i) \wedge$
  $\bigwedge_j b_j \le (2^n{-}1)\ \mathtt{div}\ (\text{LCM}\ \mathtt{div}\ \beta_j) \wedge$
  $\bigwedge_{i \ne L} (a_i \times (\text{LCM}\ \mathtt{div}\ \alpha_i) \le a_L \times (\text{LCM}\ \mathtt{div}\ \alpha_L) \wedge$
  $\bigwedge_{j \ne U} (b_U \times (\text{LCM}\ \mathtt{div}\ \beta_U) \le b_j \times (\text{LCM}\ \mathtt{div}\ \beta_j)$
  where the LCM is the least common multiple of $\{\alpha_i\} \cup \{\beta_j\}$, and the α_L and β_U are coefficients corresponding to the greatest lower bound and the least upper bound w.r.t. M, respectively:
  – $\forall i \cdot M(a_i) \times (\text{LCM}\ \mathtt{div}\ \alpha_i) \le M(a_L) \times (\text{LCM}\ \mathtt{div}\ \alpha_L)$, and
  – $\forall j \cdot M(b_j) \times (\text{LCM}\ \mathtt{div}\ \beta_j) \ge M(b_U) \times (\text{LCM}\ \mathtt{div}\ \beta_U)$;
  and, additionally, the following side-conditions are true:
  – $\text{LCM} \in \mathbb{Z}_{2^n-1}$,
  – $M(x) \times \text{LCM} \in \mathbb{Z}_{2^n-1}$, where $M(x)$ is the value of x in M,
  – for each i, $M \models a_i \le (2^n{-}1)\ \mathtt{div}\ (\text{LCM}\ \mathtt{div}\ \alpha_i)$, and
  – for each j: $M \models b_j \le (2^n{-}1)\ \mathtt{div}\ (\text{LCM}\ \mathtt{div}\ \beta_j)$.
- $\text{MBP}_{\mathbb{Z}}(M, \varphi \wedge (f(x)\ \mathtt{div}\ \delta \le d)) \stackrel{\text{def}}{=}$
  $\text{MBP}_{\mathbb{Z}}(M, \varphi \wedge (f(x) \le (d+1) \times \delta - 1) \wedge (d < (2^n{-}1)\ \mathtt{div}\ \delta))$
  under side-condition $M \models d < (2^n{-}1)\ \mathtt{div}\ \delta$.

- $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi \wedge (g < f(x) \texttt{ div } \gamma)) \stackrel{\text{def}}{=}$
  $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi \wedge (g+1) \times \gamma - 1 < f(x) \wedge (g < (2^n{-}1) \texttt{ div } \gamma))$
  under side-condition $M \models g < (2^n{-}1) \texttt{ div } \gamma$.
- $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi \wedge ((f(x) \texttt{ div } \delta) \times \alpha \leq d)) \stackrel{\text{def}}{=}$
  $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi \wedge (f(x) \times \alpha \leq (d+1) \times \delta - 1) \wedge (d < (2^n{-}1) \texttt{ div } \delta))$
  under side-condition $M \models (f(x) \times \alpha \leq (d+1) \times \delta - 1) \wedge (d < (2^n{-}1) \texttt{ div } \delta)$. The idea is to under-approximate ψ after checking that M models the under-approximation. We need to check because $((f(x) \texttt{ div } \delta) \times \alpha \leq d)$ might not imply $(f(x) \times \alpha \leq (d+1) \times \delta - 1)$.
- $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi \wedge (g < (f(x) \texttt{ div } \gamma) \times \beta)) \stackrel{\text{def}}{=}$
  $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi \wedge (g+1) \times \gamma - 1 < f(x) \times \beta \wedge (g < (2^n{-}1) \texttt{ div } \gamma))$
  under the following side-condition: $M \models (g+1) \times \gamma - 1 < f(x) \times \beta \wedge (g < (2^n{-}1) \texttt{ div } \gamma$.
- $\mathrm{MBP}_{\mathbb{Z}}(M, \varphi) \stackrel{\text{def}}{=} \varphi[x \mapsto M(x)]$, otherwise.

The last step above ensures that $\mathrm{MBP}_{\mathbb{Z}}$ is complete – in the worst case, x is projected by its value in the model. The soundness follows from soundness of the individual rewrite rules. While the rules themselves are non-trivial, we have checked their validity for many common bit-widths using Z3.

We conclude this section with two examples, where we assume that the bit-width n is set to 8, i.e., $\mathbb{Z}_{2^n-1} = \{0, 1, \ldots, 255\}$.

Example 5.1. Given a formula $a < 4 \times x \wedge 6 \times x \leq b$ and a model $M = \{a \mapsto 10, b \mapsto 100, x \mapsto 5\}$. The LCM = 12, and it does not overflow. Furthermore, LCM $\times M(x) = 12 \times 5 = 60$ does not overflow either. We then check if $M \models a \leq 255 \texttt{ div } (12 \texttt{ div } 4)$, i.e., $10 \leq 85$, and if $M \models b \leq 255 \texttt{ div } (12 \texttt{ div } 6)$, i.e., $100 \leq 127$. All side-conditions hold, thus we can construct the following MBP:

$$\mathrm{MBP}_{\mathbb{Z}}(x, \varphi) = ((3a \texttt{ div } 12) < (2b \texttt{ div } 12)) \wedge a \leq 85 \wedge b \leq 127$$

Example 5.2. Given a formula $a < 99 \times x \wedge 100 \times x \leq b$ and a model $M = \{a \mapsto 0, b \mapsto 200, x \mapsto 1\}$. The LCM = 9900 overflows. We thus, create an MBP by substituting 1 for x: $a < 99 \wedge 100 \leq b$.

6 LAZY MBP FOR BV

Alg. 2 gives our procedure to compute MBP for BV formulas (i.e., mixing both arithmetic and bit-manipulating operators). It first splits the input formula into two parts: $\wedge \varphi_j$ involving linear arithmetic only, and $\wedge \pi_i$ involving everything else (including non-linear arithmetic and bit operations). For the former, it applies the algorithm from the previous section (thus, getting P). For the latter (called *residual* part in the rest of the paper), it creates the projection by substitution (thus, getting S).

Note that it is not true in general that the conjunction of two projections (P and S) for the two parts of the formula ψ yield a projection of $\exists x \cdot \psi$. Thus, it might be necessary to strengthen the MBP with additional substitutions, i.e., for literals in $\wedge \varphi_j$. Afterwards, the algorithm proceeds in the reverse direction by weakening the resulting conjunction. The algorithm enumerates substitutions in S, tries removing them and checks if the result is an under-approximation. At the end of the loop, the under-approximation is *maximal* in a sense that no substitutions can be dropped[5].

Algorithm 2: Lazy MBP computation for BV

In: formula $\psi(x, y)$, constant x, and M, such that $M \models \psi$
Out: $\mathcal{P}(y)$, such that $M \models \mathcal{P}$ and $\mathcal{P}(y) \Rightarrow \exists x \cdot \psi(x, y)$

```
let ψ = ⋀_i π_i ∧ ⋀_j φ_j such that MBP_ℤ is applicable to ⋀_i π_i
𝒫 ← MBP_ℤ(M, ⋀_j φ_j)
S ← {π_i[x ↦ M(x)]}
if 𝒫 ∧ ⋀_{σ∈S} σ ⇏ ∃x · ψ(x, y) then S ← S ∪ {φ_j[x ↦ M(x)]}
for each i do
    if 𝒫 ∧ ⋀_{σ∈S∖{S_i}} σ ⇒ ∃x · ψ(x, y) then S ← S ∖ {S_i}
return 𝒫 ∧ ⋀_{σ∈S} σ
```

Example 6.1. Assume bit-width 8 and constant x that needs to be eliminated from the following formula.

$$\psi \equiv a \leq x \wedge x < b \wedge \texttt{extract}(x, 7, 7) = 0$$

Here, the arithmetic part of the formula imposes an upper and a lower bound on x, and the residual part requires that the most significant bit of x is set to zero. We are also given a model M such that $M(x) = 64$.

Alg. 2 first creates $\mathcal{P} = \mathrm{MBP}_{\mathbb{Z}}(M, a \leq x \wedge x < b) = a < b$. Note that $\mathcal{P} \not\Rightarrow \exists x \cdot \psi$ because, e.g., if a is instantiated with 199, and b with 200, then the only value of x satisfying $a \leq x \wedge x < b$ is 199, the most significant bit of which is 1. Alg. 2 then proceeds to strengthening of the MBP by taking into account the residual constraint and literals $a \leq x$ and $x < b$ with $M(x) = 64$ substituted for x. Since $\texttt{extract}(x, 7, 7) = 0$ has the only appearance of a single variable x, the substitution is trivially true, and thus ignored. Finally, Alg. 2 iteratively weakens the MBP by posing and solving a sequence (in our case, two) of quantified formulas:

$$a < b \wedge 64 < b \Rightarrow \exists x \cdot \psi \qquad \text{does not hold,}$$
$$a < b \wedge a \leq 64 \Rightarrow \exists x \cdot \psi \qquad \text{holds.}$$

Intuitively, Alg. 2 found a yet another upper bound of x and instantiated it with the value of x from the model. The final MBP is $a < b \wedge a \leq 64$. □

7 SPACER

Spacer is a state of the art solver for Constrained Horn Clauses (CHCs) based on the IC3/PDR paradigm. In this section, we describe the internals of Spacer including the use of MBP to compute predecessors and global guidance rules. Then, we present our second main contribution: that of extending the global guidance rules to BV (Sec. 7.1). We simplify the presentation by focusing on how Spacer establishes safety of transition systems. We stress that Spacer as well as our main ideas and implementation work on the more general setting of satisfiability of CHCs.

Given a transition system and a set of bad states, Spacer iteratively establishes safety at larger and larger depths until it either finds a counterexample or an invariant. To establish safety at a particular depth, Spacer recursively computes and blocks *proof obligations* (POB). A POB is a tuple $\langle \varphi, i \rangle$, where φ is a set of states that can lead to either a bad state, or another POB, at depth $i+1$. It is reachable if φ is reachable from *Init* at depth i. Alg. 3 explains

[5]Note that this does not mean that the under-approximation is the weakest possible. Finding the weakest under-approximation requires enumeriation all subsets of S, dropping the corresponding substitutions and checking mutual implications. This is expensive, and, thus, we do not do it.

Algorithm 3: SPACER algorithm as a set of guarded commands. A command can be executed whenever its preconditions are met. We use the shorthand $\mathcal{F}(\varphi) = \mathcal{U}' \vee (\varphi \wedge Tr)$. We use $\varphi[x \mapsto y]$ to mean replacing each occurrence of x in φ with y.

In: A transition system $\langle X, Init, Tr \rangle$
In: A set of bad states Bad
Out: $\langle$SAFE, $Inv\rangle$ or UNSAFE

$Q \leftarrow \emptyset$ `// pob queue`
$N \leftarrow 0$ `// maximum safe level`
$\mathcal{O}_0 \leftarrow Init, \mathcal{O}_i \leftarrow \top$ **for all** $i > 0$ `// inductive trace`
$\mathcal{U} \leftarrow Init$ `// reachable states`
$Bad \leftarrow \neg\varphi$ `// bad states`
forever do
  `Candidate` $[\![\ \textsc{isSat}(\mathcal{O}_N \wedge Bad)\ ]\!]$
    $Q \leftarrow Q \cup \langle Bad, N \rangle$
  `Predecessor` $[\![\ \langle \varphi, i+1 \rangle \in Q, M \models \mathcal{O}_i \wedge Tr \wedge \varphi'\ ]\!]$
    $Q \leftarrow Q \cup \langle \mathcal{P}(X', Tr \wedge \varphi', M), i \rangle$
  `Successor` $[\![\ \langle \varphi, i+1 \rangle \in Q, M \models \mathcal{F}(\mathcal{U}) \wedge \varphi'\ ]\!]$
    $\mathcal{U} \leftarrow \mathcal{U} \vee \mathcal{P}(X, \mathcal{F}(\mathcal{U}), M)[X' \mapsto X]$
  `Conflict` $[\![\ \langle \varphi, i+1 \rangle \in Q, \mathcal{F}(\mathcal{O}_i) \Rightarrow \neg\varphi'\ ]\!]$
    $\mathcal{O}_j \leftarrow (\mathcal{O}_j \wedge \textsc{itp}(\mathcal{F}(\mathcal{O}_i), \varphi')[X' \mapsto X])$ **for all** $j \leq i+1$
  `Propagate` $[\![\ \ell \in \mathcal{O}_i, \mathcal{O}_i \wedge Tr \Rightarrow \ell'\ ]\!]$
    $\mathcal{O}_{i+1} \leftarrow (\mathcal{O}_{i+1} \wedge \ell)$
  `Subsume` $[\![\ \mathcal{L} \subseteq \mathcal{O}_i, \forall \ell \in \mathcal{L} \cdot \psi \Rightarrow \ell, k \geq i, \mathcal{F}(\mathcal{O}_k) \Rightarrow \psi'\ ]\!]$
    $\mathcal{O}_j \leftarrow (\mathcal{O}_j \wedge \psi)$ **for all** $j \leq k+1$
  `Conjecture` $[\![\ \mathcal{L} \subseteq \mathcal{O}_i, \langle \varphi, k \rangle \in Q, \varphi \equiv \alpha \wedge \beta,$
      $\forall \ell \in \mathcal{L} \cdot \ell \Rightarrow \neg\beta \wedge \textsc{isSat}(\ell \wedge \alpha), \mathcal{U} \Rightarrow \neg\alpha\ ]\!]$
    $Q \leftarrow Q \cup \langle \alpha, k+1 \rangle$, where $k = max\{j \mid \mathcal{O}_j \Rightarrow \neg\alpha\}$
  `Unfold` $[\![\ \mathcal{O}_N \Rightarrow \neg Bad\ ]\!]$
    $N \leftarrow N + 1$
  `Safe` $[\![\ \mathcal{O}_{i+1} \Rightarrow \mathcal{O}_i$ **for some** $i < N\ ]\!]$
    return $\langle$SAFE, $\mathcal{O}_i\rangle$
  `Unsafe` $[\![\ \textsc{isSat}\ (Bad \wedge \mathcal{U})\ ]\!]$
    return UNSAFE

SPACER as a set of guarded commands. The state of SPACER consists of the following. A queue of proof obligations (Q). The depth upto which the system is SAFE (N). An inductive trace ($\mathcal{O}$) and a set of *reachable* states ($\mathcal{U}$). The inductive trace is a sequence of *frames* $\mathcal{O}_0, \mathcal{O}_1, \ldots \mathcal{O}_k$. Each frame is a conjunction of clauses called *lemmas*. The intuition is that frame $\mathcal{O}_i$ over-approximates the set of states reachable in i steps from *Init*. This is achieved by keeping the trace monotone ($\forall i \cdot \mathcal{O}_i \Rightarrow \mathcal{O}_{i+1}$) and inductive ($\forall i \cdot \mathcal{O}_i \wedge Tr \Rightarrow \mathcal{O}'_{i+1}$).

The `Candidate` rule adds the POB $\langle Bad, N\rangle$ to the queue of POBS. When a POB is shown to be reachable from the set of reachable facts, `Successor` rule extends the set of reachable facts. When a POB cannot be blocked by a frame, the `Predecessor` rule computes a predecessor POB using MBP. If a POB is blocked by a frame $\mathcal{O}_i$, the `Conflict` rule uses interpolation to strengthen frame $i+1$ (as well as all other frames before it). If a lemma is true at a higher frame, the `Propagate` rules pushes it forward. Once *Bad* has been shown to be unreachable, the `Unfold` rule increases the depth N. The `Safe` rule returns an inductive frame if it exists and the `Unsafe` rule returns UNSAFE if the reachable states intersect with *Bad*.

Global Guidance for SPACER. [24] introduced three global guidance rules for SPACER. These rules were designed to allow SPACER to detect and correct itself when it diverges. In this work, we use two of them: `Subsume` and `Conjecture`. Both of them operate on a subset $\mathcal{L}$ of the lemmas in the trace. The `Subsume` rule summarizes $\mathcal{L}$ by generating a single lemma ψ that is stronger than all the lemmas in $\mathcal{L}$. ψ is then added to the trace to replace all the lemmas in $\mathcal{L}$. Intuitively, this rule guides SPACER towards a succinct proof. The `Conjecture` rule gets the solver unstuck from a bad place in the search space. This rule is applied when SPACER learns multiple lemmas to block the same part of a POB. The rule creates an abstract POB and adds it to the queue, thereby allowing SPACER to focus on a different part of the search space.

7.1 Global Guidance for BV

In this section, we explain an instantiation of the global guidance rules for the theory of BV. In Sec. 7.2, we explain how we select the subset of lemmas in which to apply the rules. In Sec. 7.3, we explain an instantiation of the `Subsume` rule and in Sec. 7.4, we explain an instantiation of the `Conjecture` rule.

7.2 Clustering

We select subsets of lemmas based on syntactic pattern matching. A *pattern* is a formula with free variables. A formula f *matches* a pattern π if there is a substitution σ from free variables to terms such that $\pi\sigma = f$. A substitution is called *numeric* if its maps free variables to BV numerals. Given a pattern π and a set of formulas Φ, a *cluster* $C_\Phi(\pi)$, is the set of all formulas in Φ that matches π with numeric substitutions. We apply the guidance rules on clusters of lemmas.

In practice, given a set of lemma, we use the concept of anti-unification to generate patterns and hence identify clusters. Note that all definitions are syntactic. Therefore, a formula $x + 1$ can match a pattern $x + v$ but the formula $1 + x$ cannot. To avoid missing lemmas when picking subsets of lemmas, we use the normalization scheme for BV inside Z3 to normalize all lemmas before computing clusters of lemmas.

7.3 Subsume

We apply the subsume rule to a cluster $C_\Phi(\pi)$ if $\pi = \varphi \vee \bigvee_{i=1}^{N} \ell_i$ where φ is a ground formula and each literal ℓ_i is either of the form $v_i + t_1 \bowtie t_2$ or of the form $v_i \bowtie t_3$ where $\bowtie \in \{\leq, \leq_s, <, <_s, =\}$. We also have the additional constraint that all the variables in π are of the same bit-width.

This computation works in the cube space $C_{\neg\Phi}(\pi')$ where $\pi' = \neg\pi$. We compute an over-approximation Q of $(\bigwedge C_{\neg\Phi}(\pi'))$. Therefore, $\neg Q \Rightarrow (\bigvee C_\Phi(\pi))$ subsumes all the lemmas in $C_\Phi(\pi)$.

To compute such a cube Q, we compute a set of linear equalities that hold between the free variables $v_1, \ldots v_N$ that are implied by the cubes in $C_{\neg\Phi}(\pi')$. Let P_k be the ordered set of numeric values corresponding to cube c_k in $C_{\neg\Phi}(\pi')$. That is $P_k = (n_1, \ldots, n_N)$ such that $\sigma_k = \bigwedge_{i=1}^{N} v_i = n_i$ and $\pi\sigma_k = c_k$. Let Z_k be the integer relaxation of P_k. Let A be the matrix whose rows are $[Z_k, 1]$. The dimension of A is $K \times (N+1)$, where K is the number of cubes in $C_{\neg\Phi}(\pi')$. Let r be a row in a kernel of A. Then we know that $r[v_1, \ldots, v_N, 1]^T = 0$ is an equality implied by the cubes in $C_{\neg\Phi}(\pi')$. Thus, we get one equality per row of the kernel. Let E be the conjunction of all these equalities.

The cluster of cubes might also imply divisibility constraints between the variables in the pattern. For variable v_i, let S_i be the set of values that v_i can be substituted into. That is $S_i = \{n \mid \sigma[v_i] = n \wedge \pi'\sigma \in C_{\neg\Phi}(\pi')\}$. We add a divisibility constraint $(a \mid (v_i - b))$ if $(a \mid (s - b)) \wedge a \neq 1$ for all $s \in S_i$. If there are multiple such a's, the largest one is chosen. Let M be a conjunction of all such divisibility constraints.

Table 1: Verification results (time in seconds; TO = 1200s). Spacer† is the latest public version of Spacer.

Benchmark		GSpacerbv	Spacer	Spacer†	Eldarica
RC	1	0.70	0.57	0.06	35.43
SP	1	0.26	0.26	0.07	TO
	2	0.48	0.61	0.08	294.49
GBA	1	0.68	0.67	0.21	TO
	2	0.67	0.70	0.23	TO
	3	0.75	0.67	0.25	TO
AES	1	17.59	2.08	20.3	TO
	2	57.89	48.00	12.35	TO
	3	TO	TO	TO	TO
	4	TO	TO	TO	TO
	5	0.45	0.43	0.23	TO
	6	21.13	10.41	2.8	TO
	7	2.85	2.57	0.89	TO
PP	1	7.23	7.30	TO	TO
	2	4.15	4.34	TO	TO
	3	7.06	7.14	TO	TO
	4	2.75	2.81	TO	TO
	5	21.35	20.95	TO	TO
	6	6.04	6.12	TO	TO
	7	6.10	6.33	TO	TO
	8	12.92	13.32	TO	TO
	9	23.43	12.10	TO	TO

Let $C = \pi' \wedge E \wedge M$. To get an over-approximation, we eliminate the variables in the pattern π': $Q = \exists v_1, \ldots v_N \cdot C$. In our implementation, we use MBP to under-approximate quantifier elimination. In order to achieve good MBP results, whenever possible, we choose a model that is not satisfied by any cube in $C_{\neg\Phi}(\pi')$ but by C. Since MBP produces under-approximations of quantifier elimination, we drop literals from the result of MBP until it over-approximates $\exists v_1, \ldots, v_N \cdot C$.

7.4 Conjecture

The `Conjecture` rule is applied when a single proof obligation is blocked by many lemmas. Syntactically, this happens when the negation of the proof obligation and the lemma shares some common literals.

The `Conjecture` rule is applied to a cluster $C_\Phi(\pi)$ with a pattern $\pi = (\varphi_1 \vee \varphi_2)$, where φ_1 is ground, and a proof obligation $p = c_1 \wedge c_2 \wedge c_3$ under the following conditions: (1) one lemma in the cluster subsumes all others. The following syntactic restriction achieves this: $\varphi_2 = v \bowtie t$ with $\bowtie \in \{\leq, \leq_s, <, <_s\}$; (2) all the lemmas block one part of the proof obligation. This happens when there exists a substitution σ such that $\pi\sigma \in C_\Phi(\pi)$, $\varphi_2\sigma = \neg c_3$ and $\varphi_1 \Rightarrow c_2$; (3) all the lemmas do not block the other part of the proof obligation: for each $\ell \in C_\Phi(\pi)$, $\ell \wedge c_1 \wedge c_2$ is satisfiable; (4) The other part of the proof obligation is not reachable: $\mathcal{U} \not\Rightarrow c_1 \wedge c_2$.

8 EVALUATION

We have implemented both global guidance and BV MBP on top of the Spacer CHC engine [23] of the Z3 SMT solver [9]. We call the new tool GSpacerbv. We have compared GSpacerbv against the baseline Spacer, i.e., with our MBP and global guidance disabled, and to Eldarica [18], which, to the best of our knowledge, is the only other CHC solver that supports BV. The rest of the section elaborates on two of our cases studies: applying word-level PDR to verify hardware and software, respectively.

8.1 Hardware Verification

We considered benchmarks from the Grain suite [34] on discovering adequate environment abstractions for instruction-based (ILA) equivalence checking. Grain adopts an abstraction-refinement strategy to refine the environment by blocking spurious counterexamples that are found during equivalence checking, i.e., by solving a sequence of safety-verification tasks (encoded as CHC systems).

The suit has 22 CHC instances that belong to five top-level equivalence checking tasks, i.e., for five hardware designs[6]:

- *Redundant Counters* (RC) that uses two 4-bit counters, one of which is for redundancy – stored as 1's-complement (represented by one CHC system);
- *Simple Pipeline* (SP) with the back-end of a simple pipelined processor which has three stages and four 8-bit wide registers (two CHC systems);
- *Gaussian Blur Accelerator* (GBA) that uses the multiplication-accumulation units for convolution of an image with a Gaussian kernel (three CHC systems);
- *AES Block Encryption Accelerator* (AES) with a "load-compute-store" loop that works block-by-block (seven CHC systems);
- *PicoRV32 Processor* (PP) that implements the RISC-V RV32IMC instruction set and has some pipelining features (nine CHC systems).

Table 1 outlines an experimental comparison of GSpacerbv, different versions of Spacer, and Eldarica. Additionally, the comparison to Grain itself and ABC [17] can be derived from [34]. While Grain can solve all the instances, it should be parametrized by tailor-made templates for invariants. Eldarica, while fully automated, cannot solve most of these instances. Spacer and GSpacerbv, in contrast, are fully automated and can solve 20 (out of 22) instances. Spacer† is the latest public version of Spacer in the Z3 repository[7]. It contains an orthogonal bug that prevents it from converging on many of these instances. However, GSpacerbv could solve these instances irrespective of the bug. Additionally, we present a version of Spacer in which we fixed the bug (the Spacer column). Clearly, GSpacerbv is competitive with Spacer. The results shown here are obtained by running Spacer and GSpacerbv with the default random seed. We experimented with different random seeds and saw that the choice of random seed affect the results in favor of GSpacerbv.

8.2 Software Verification

Our technique targets verification problems of arbitrary structure (i.e., not only transition systems, but also software with function calls, nested loops, etc.). Therefore, we have also considered 631 open-source benchmarks from software verifiers: vmt [7] and Eldarica [2]. GSpacerbv has two configurations: with/without the global guidance, which, in combination, can solve more instances (and faster) than the baseline Spacer: 404 vs 398. Eldarica solved only 225 instances. 192 instances were not solved by any tool.

In the interest of saving space, we present the comparison in two scatterplots in Fig. 3. It is clear that in most of the cases GSpacerbv outperforms the baseline Spacer. However, there are some instances where the baseline Spacer outperforms us. Interestingly,

[6]We refer the reader to [34] for a complete description of designs.
[7]https://github.com/Z3Prover/z3/releases/tag/z3-4.8.8.

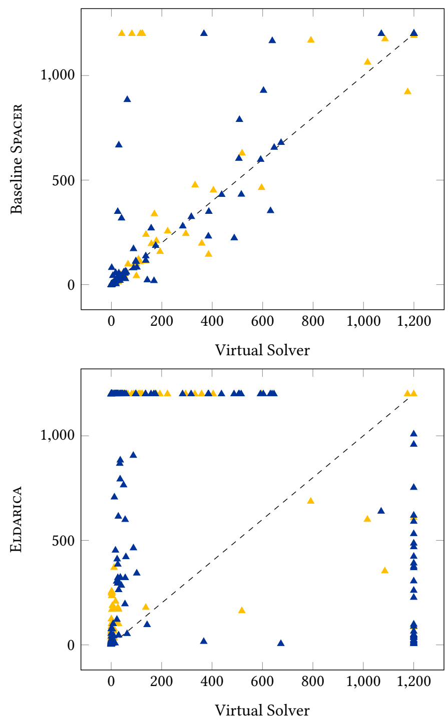

Figure 3: Scatterplots comparing against competitors (sec×sec): triangles above the diagonal represent runtimes for benchmarks on which the best configuration of GSpacerBV outperformed the competitor; blue triangles = safe instances, orange = unsafe; timeouts (1200 sec) are placed on the boundaries.

most of these instances are UNSAFE. More importantly, there are no SAFE instances (which are in general much harder) that Spacer can solve, but GSpacerBV cannot solve. GSpacerBV can solve 2 SAFE instances that the baseline Spacer cannot solve. While solving 179 more instances than Eldarica, GSpacerBV is unable to solve 34 instances that Eldarica solved (possibly, because their recent quantifier elimination and interpolation techniques [2] which in principle can be integrated to GSpacerBV). This lets us conclude that our technique is promising in practice, and we are looking forward to its optimizations and extensions.

9 RELATED WORK

Prior approaches to word-level unbounded model checking (inductive invariant synthesis) are accomplished mainly via an integration with Counterexample-Guided Abstraction Refinement (CEGAR) [8], attempting to reduce the use of bit-precise reasoning. In particular, [25] for a hardware design, prunes most of the state space using uninterpreted predicates and delegates the "residual" control space to IC3/PDR. This idea further evolved to an approach for model checking C programs [5] and Verilog designs [14]. A Word-Level Abstraction for hardware [17] is also based on this idea, but it is more optimized by the re-use of PDR traces and tailor-made refinement strategies. Our approach is substantially different in the way that there is no need to use the abstraction-refinement loop explicitly. This affords us a significant degree of flexibility in guiding the PDR algorithm as it explores the state space.

To bypass the need to extend the decision procedures with the support for quantifier elimination and interpolation, [7] proposed to integrate IC3 with predicate abstraction. This way, IC3 operates only at the Boolean level, and theory reasoning is conducted only by the underlying SMT solver. Another approach to get bit-precise invariants, proposed in [16], suggests to unsoundly translate machine integers by unbounded integers, use the LIA-verification tool and finally validate the resulting invariants on the original problem. Recently, in [34], it was proposed to to use user-given templates and Syntax-Guided Synthesis (SyGuS) [1] to guess-and-check bit-precise invariants. Our approach, in contrast, enjoys the native support for quantifier elimination and does not rely on predetermined predicates or templates, as well as on any external tools.

The closest approach to ours is on *Property Directed Reachability for QF_BV* [31] which was further extended to mixed type atomic reasoning units [32]. They proposed a version of the Subsume rule for QF_BV but did not use global guidance. Hybrid simulation and mixed types of atomic reasoning units are used for inductive and counterexample generalization. However, their method works only on the arithmetic level, and ours supports the all bitvector operators, thanks to state-of-the-art decision procedures.

An earlier approach to Word Level Predicate Abstraction [20] proposes to use weakest preconditions of Verilog statements to obtain new predicates during abstraction refinement. The weakest-precondition computation is in general expensive and uses some kind of quantifier elimination too. While using MBP, we do not guarantee the result is the weakest, but in practice it is often *adequate* and much less computationally expensive.

Some related research was done in the domain of solving BV-formulas. In particular [26] proposed to eliminate quantifiers using symbolic inverses of bit-vector operators, which are pre-computed using SyGuS. We can naturally plug them to our model-based rewriting system. Effective word-level interpolation approaches were proposed in [2, 15, 21, 33]. Tricks include computing interpolants by treating BV operations uninterpreted; using a restricted form of quantifier elimination; translating to (non)-linear integer arithmetic and back, and finally bit-blasting. These techniques can in principle be integrated to our approach too and accelerate convergence.

Aside of verification, MBP has applications in functional synthesis [11] to discover function implementations from declarative specifications, SyGuS-based CHC solving [12], and Validity-Guided Synthesis of Reactive Systems [22]. We believe, in the future, our new MBP algorithm for BV will strengthen those applications.

10 CONCLUSION

We have presented a new approach to word-level verification based on IC3/PDR that does not require an integration with an external abstraction-refinement loop. Instead of using bit-blasting to eliminate quantifiers from BV-formulas, we proposed a less expensive method for iterative approximate quantifier elimination in BV that supports all bit-operators and can be optimized further by applying rules inspired by modular arithmetic. It uses recent techniques for learning inductive invariants based on explicit global guidance, thus bypassing interpolation. Our implementation on top of the Spacer tool confirms that a word-level PDR is more effective than state-of-the-art on a range of hardware and software benchmarks.

ACKNOWLEDGMENTS

This research was funded by a grant from Waterloo-Huawei Joint Innovation Lab.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *FMCAD*. IEEE, 1–17.

[2] Peter Backeman, Philipp Rümmer, and Aleksandar Zeljic. 2018. Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction. In *FMCAD*. IEEE, 50–59.

[3] Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. In *LPAR (short papers) (EPiC Series in Computing, Vol. 35)*. EasyChair, 15–27.

[4] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *VMCAI (LNCS, Vol. 6538)*. Springer, 70–87.

[5] Denis Bueno and Karem A. Sakallah. 2019. EUFORIA: Complete Software Model Checking with Uninterpreted Functions. In *VMCAI (LNCS, Vol. 11388)*. Springer, 363–385.

[6] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. 2011. Incremental formal verification of hardware. In *FMCAD*, Per Bjesse and Anna Slobodová (Eds.). FMCAD Inc., 135–143.

[7] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2014. IC3 Modulo Theories via Implicit Predicate Abstraction. In *TACAS (LNCS, Vol. 8413)*. Springer, 46–61.

[8] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (LNCS, Vol. 1855)*. Springer, 154–169.

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.

[10] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *FMCAD*. IEEE, 125–134.

[11] Grigory Fedyukovich, Arie Gurfinkel, and Aarti Gupta. 2019. Lazy but Effective Functional Synthesis. In *VMCAI (LNCS, Vol. 11388)*. Springer, 92–113.

[12] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2018. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*. IEEE, 170–178.

[13] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. 2004. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *ICCAD*. IEEE Computer Society / ACM, 510–517.

[14] Aman Goel and Karem A. Sakallah. 2019. Model Checking of Verilog RTL Using IC3 with Syntax-Guided Abstraction. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. 166–185. https://doi.org/10.1007/978-3-030-20652-9_11

[15] Alberto Griggio. 2011. Effective word-level interpolation for software verification. In *FMCAD*. FMCAD Inc., 28–36.

[16] Arie Gurfinkel, Anton Belov, and João Marques-Silva. 2014. Synthesizing Safe Bit-Precise Invariants. In *TACAS (LNCS, Vol. 8413)*. Springer, 93–108.

[17] Yen-Sheng Ho, Alan Mishchenko, and Robert Brayton. 2017. Property Directed Reachability with Word-Level Abstraction. In *FMCAD*. ACM, 132–139.

[18] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *FMCAD*. IEEE, 158–164.

[19] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. 2016. Verilog2SMV: A tool for word-level verification. In *DATE*. IEEE, 1156–1159.

[20] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. 2005. Word level predicate abstraction and refinement for verifying RTL verilog. In *DAC*, William H. Joyner Jr., Grant Martin, and Andrew B. Kahng (Eds.). ACM, 445–450.

[21] Ajith K. John and Supratik Chakraborty. 2016. A layered algorithm for quantifier elimination from linear modular constraints. *Formal Methods Syst. Des.* 49, 3 (2016), 272–323.

[22] Andreas Katis, Grigory Fedyukovich, Huajun Guo, Andrew Gacek, John Backes, Arie Gurfinkel, and Michael W. Whalen. 2018. Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts. In *TACAS, Part II (LNCS, Vol. 10806)*. Springer, 176–193.

[23] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV (LNCS, Vol. 8559)*. 17–34.

[24] Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. 2020. Global Guidance for Local Generalization in Model Checking. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 101–125. https://doi.org/10.1007/978-3-030-53291-8_7

[25] Suho Lee and Karem A. Sakallah. 2014. Unbounded Scalable Verification Based on Approximate Property-Directed Reachability and Datapath Abstraction. In *CAV (Lecture Notes in Computer Science, Vol. 8559)*. Springer, 849–865.

[26] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. 2018. Solving Quantified Bit-Vectors Using Invertibility Conditions. In *CAV, Part II (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 236–255.

[27] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. 2018. Btor2 , BtorMC and Boolector 3.0. In *CAV, Part I (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 587–595.

[28] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *FCCM*. IEEE, 1–4.

[29] Pramod Subramanyan, Sharad Malik, Hareesh Khattri, Abhranil Maiti, and Jason M. Fung. 2016. Verifying information flow properties of firmware using symbolic execution. In *DATE*, Luca Fanucci and Jürgen Teich (Eds.). IEEE, 337–342.

[30] Yakir Vizel and Orna Grumberg. 2009. Interpolation-sequence based model checking. In *FMCAD*. IEEE, 1–8.

[31] Tobias Welp and Andreas Kuehlmann. 2013. QF BV model checking with property directed reachability. In *DATE*. EDA Consortium San Jose, CA, USA / ACM DL, 791–796.

[32] Tobias Welp and Andreas Kuehlmann. 2014. Property Directed Reachability for QF_BV with mixed type atomic reasoning units. In *ASP-DAC*. IEEE, 738–743.

[33] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. 2016. Deciding Bit-Vector Formulas with mcSAT. In *SAT (Lecture Notes in Computer Science, Vol. 9710)*. Springer, 249–266.

[34] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing Environment Invariants for Modular Hardware Verification. In *VMCAI (Lecture Notes in Computer Science, Vol. 11990)*. Springer, 202–225.