Word Level Property Directed Reachability

Hari Govind V K
University of Waterloo, Waterloo
Canada
hgkv94@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

1 INTRODUCTION

Firmware implements key hardware functions, accelerates maintenance, and enables to deploy bugsfixes 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.
Figure 1: An example program in C-like syntax. The variables \( x \) and \( y \) are unsigned integers. \( \text{nd}() \) 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 \( \text{ELDARICA} \) [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: \((x = 1, y = 0), (x = -1, y = -4), \) 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 \( x + y = 1, x + y = -3, . . . \), to \( x + y \mod 2 = 1 \) (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 \( \Sigma \) contains an infinite number of constant symbols (zero-ary, uninterpreted functions) denoted with \( x, y, \) 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.

\[
\begin{align*}
term &::= \text{con} \mid \text{num} \mid \text{var} \mid \text{term} \times \text{term} \mid \text{term} \div \text{num} \mid \\
& \quad \text{term} + \text{term} \mid \text{term} - \text{term} \mid \\
& \quad \text{bvop}(\text{term}) \ldots \mid \text{bvop}(\text{term}, \ldots)
\end{align*}
\]

\(^1\)Adopted from https://github.com/sosy-lab/sv-benchmarks/blob/master/c/bitvector/jain_2-1.c.

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., \( \text{div} \) is the division operator (and \( e.g., 5 \text{ div } 3 = 1 \)). The grammar supports bit operators (\( \text{bvop} \)) such as bit shifts, bitwise and/or, extracts, etc.

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

**MBP.** Given a cube \( \varphi \) 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 \( \exists X_s \cdot \phi \). That is, MBP\( (X_s, \varphi, M) \) is a cube \( \varphi \) such that \( \varphi \Rightarrow \exists X_s \cdot \phi \wedge \text{conj}(P) \Rightarrow 0 \) and \( M = P \). An MBP is finite if the range of MBP is finite after fixing \( \varphi \) and \( X_s \). Every theory that admits quantifier elimination also admits a finite MBP. We skip mentioning \( X_s \) and write \( \text{MBP}(\varphi, M) \) when the constants that are eliminated are obvious from the context. The notion of MBP can be lifted to an arbitrary formula \( \psi \) (which is not necessarily a cube) by considering a model \( M \) of \( \psi \), conjoining literals of \( \psi \) 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 \( \text{const}(I) \subseteq \text{const}(A) \cap \text{const}(B) \). 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 \((X, \text{Init}(X), \text{Tr}(X, X'))\) where \( X \) is the set of constants that represent the state of the system (called state variables), \( \text{Init}(X) \) is a formula representing the set of initial states of the system, and \( \text{Tr}(X, X') \) 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 up to a depth \( d \), relative to a set of bad states \( \text{Bad} \), if there are no counterexamples of depth less than \( d \). That is, if there are no sequences of states \( s_0, s_1, s_2, \ldots, s_n \) such that \( n < d \) and \( s_0 \in \text{Init}, \text{Tr} \subseteq i < n \cdot \{s_i, s'_{i+1}\} \in \text{Tr} \text{ and } s_n \in \text{Bad} \). The transition system is safe if there are no counterexamples of any depth. A safe inductive invariant is a certificate for safety. A formula \( \text{Inv} \) is a safe inductive invariant if \( \text{Init} \Rightarrow \text{Inv}, \text{Inv} \Rightarrow \text{Tr} \Rightarrow \text{Inv}' \), and \( \text{Inv} \Rightarrow \neg \text{Bad} \).

3.2 Modular linear arithmetic

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

\(^2\)Signed comparison \( <, \leq \) is defined via unsigned in the next subsection.
Algorithm 1: Rew\((\psi, x, M, R)\): Model-based rewriting

In: Cube \(\psi\), constant \(x\), model \(M\) such that \(M \models \psi\), set of rules 
\(R = \{ (\text{prs}, \text{concl}) \}\)

Out: Formula \(\varphi\), such that \(\varphi \models \psi\), and either \(\varphi = \text{false}\) or \(M \models \varphi\)

1 if \(\text{contains}(\psi, x)\) or \(\text{pattern-match}(\psi, \text{term} < \text{num} \times x)\) or 
\(\text{pattern-match}(\psi, \text{num} \times x \leq \text{term})\) then 
    return \(\psi\)
2 for each \((\text{prs}, \text{concl}) \in R\) do 
3    if \(\text{pattern-match}(\psi, \text{concl})\) then 
4        res \(\leftarrow\) true 
5        \(\varphi \leftarrow\) true 
6    for each \(p \in \text{prs}\) do 
7        \(q \leftarrow\) rewrite \((p, \psi)\) 
8        \(q \leftarrow\) Rew \((q, x, M, R)\) 
9        if \(M \not\models q\) then 
10            \(\varphi \leftarrow\) \(\varphi \land q\) 
11        else 
12            res \(\leftarrow\) false 
13    if res then return \(\varphi\)
14 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:
  \(a \leq b \iff \text{if}((a < 2^{n-1} \land b < 2^{n-1}) \lor (2^{n-1} \leq a \land 2^{n-1} \leq b), \ a \leq b, (2^{n-1} - a \land b \leq 2^{n-1}))\),

• divisibility constraint:
  \((a \mid b) \iff ((a - 0) \lor ((a - 1) \div b < a \div b)\),

• remainder: \(a \mod b \equiv (a - (a \div b) \times b)\).

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 \((\text{prs}, \text{concl})\) consists of a set of premises \(\text{prs}\) (in above case, two) and a conclusion. Let \(X\) be a set of constants \(\text{const}(\text{concl}) \cup \bigcup_{p \in \text{prs}} \text{const}(p)\). We claim that for all the rules, it holds that \(\forall X \cdot \bigwedge_{p \in \text{prs}} P(X) \Rightarrow \text{concl}(X)\), 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 \(\text{add}_1, \ldots, \text{add}_5\) 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 \(t(x) + y \leq z\) or \(t(x) + y \geq z\), 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 \(\psi\), the result of the rewriter is a formula \(\varphi\), such that: (1) \(\varphi\) is satisfied by the model \(M\), i.e., \(M \models \varphi\), (2) \(\varphi\) is conjunctive, (3) each of its conjuncts either has the form \(\text{term} \leq \text{num} \times x\) or \(\text{num} \times x \leq \text{term}\), or is \(x\)-free, and (4) \(\varphi \models \psi\). Thus, if the given formula \(\psi\) meets (1), (2), and (3), the algorithm returns \(\psi\) (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 try 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 \text{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 not the focus of this paper and left for future work).

Example 4.1. Assume bit-width of 8. Let us normalize inequality \(\psi \equiv (c - x) + y \leq z\) 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., \((84 - 9) + 255 > y\), which 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:

\[
\begin{align*}
&x \geq c + y - z & x \leq c & x \leq c + y \\
&\text{[inv]} & \text{[inv]} & \text{[inv]}
\end{align*}
\]

\[
\begin{align*}
-x \leq z - y - c & -c \leq -x & -y - c \leq -x & -y \leq c - 1 \\
\text{[add]} & \text{[add]} & \text{[add]} & \text{[add]} \\
\end{align*}
\]

\[
\begin{align*}
&c - x \leq z - y & -y \leq c - x \\
&\text{[add]} & \text{[add]} \\
\end{align*}
\]

\[
(c - x) + y \leq z
\]

In order to create this graph, the algorithm begins with \(\psi\) (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 implication of \(\psi\). In particular, in order to construct an incoming edge to the root, we can either apply \([\text{add}_1]\), \([\text{add}_2]\), or \([\text{add}_3]\). To decide which one, we check if premises are satisfied by \(M\): for \([\text{add}_1]\), \(M(y \leq z) = 255 \leq 80 = \text{false}\), but for \([\text{add}_2]\), \(M(-y \leq c - x) = 1 \leq (84 - 9) = \text{true}\) and \(M(c - x \leq z - y) = 84 - 9 \leq 80 = \text{true}\). Then, the process continues onward, by applying rules to premises of \([\text{add}_2]\).

\(^3\)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.
When restricted to LIA. For example, we prefer integer division (see Sect. 6) will take care of them first, by substitution of the values $x, t$. BV operations behave as their arithmetic counterparts and adapt the idea of MBP. 5 DEFINING RBP rewriting presented in this section, we are interested in populating the normalization graph cannot be constructed with respect to the given condition. Note that if two or more rules are used to construct MBP. Note that we can normalize literals of form $a \leq A$ and $B < b$ by as follows:

- $a \leq A$ is equivalent to $a = 0 \lor a = 1 < A$, and
- $B < b$ is equivalent to $b \neq A \land B < b – 1$, where we use the model to pick one of the disjuncts in the first case.

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

- $\forall i \in \psi \land \bigwedge_i (\psi_j \land x \leq b_i) \| \psi$. Note that the model $M$ is not used. There always exists a value $x \mapsto 0$, making $\psi \equiv \bigwedge_i 0 \leq b_i$, and since each $b_i$ cannot be negative, $\psi$ is true.
- $\bigwedge_i M, \psi \land \bigwedge_i (\psi_i) \land (\bigwedge_i (\psi_j \land x \leq b_i)) \| \psi \land \bigwedge_j (a_i \land (\text{lcm} \div a_i \div \text{lcm}) \land (\text{b} \div \text{lcm} \div \beta_i)) \land \bigwedge_j (a_i \land (\text{lcm} \div a_i \div \text{lcm})) \land \bigwedge_j (b_j \div \text{lcm} \div \beta_j) \land \bigwedge_j (a_i \land (\text{lcm} \div a_i \div \text{lcm} \div \beta_i) \land (\text{b} \div \text{lcm} \div \beta_j))$, where the lcm is the least common multiple of $\{a_i \cup \beta_j\}$, and the $a_i$ and $\beta_i$ are coefficients corresponding to the maximum lower bound and the least upper bound w.r.t. $M$, respectively:

$\forall i \cdot (a_i \land (\text{lcm} \div a_i \div \text{lcm}) \land (\text{b} \div \text{lcm} \div \beta_i)) \land \bigwedge j (a_i \land (\text{lcm} \div a_i \div \text{lcm} \div \beta_i)) \land (\text{b} \div \text{lcm} \div \beta_j)$.

After the normalization graph is constructed, the normalized underapproximation $\psi' = x \geq c + y > 0 \land x \leq c \land x \leq c + y \land y \leq c = 1$ 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

In this section, we define our procedure, MBP, 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 $\varphi$ of the form $\psi \land f(x)$, where $\psi$ is $x$-free and $f(x)$ meets the grammar from Sect. 3.2. We assume that $\varphi$ is satisfiable, and that a model $M$ of $\varphi$ 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 presentation, we use arbitrary precision arithmetic to check for overflow conditions. The basic idea of MBP in the model $M$ to identify the cases when the BV operations behave as their arithmetic counterparts 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 (di\lor) 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 $f(x) \leq b$, 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 $a \leq A$ and $B < b$ as follows:

- $a \leq A$ is equivalent to $a = 0 \lor a = 1 < A$, and
- $B < b$ is equivalent to $b \neq A \land B < b – 1$, where we use the model to pick one of the disjuncts in the first case.

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

- $a \leq A$ is equivalent to $a = 0 \lor a = 1 < A$, and
- $B < b$ is equivalent to $b \neq A \land B < b – 1$, where we use the model to pick one of the disjuncts in the first case.

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

- $a \leq A$ is equivalent to $a = 0 \lor a = 1 < A$, and
- $B < b$ is equivalent to $b \neq A \land B < b – 1$, where we use the model to pick one of the disjuncts in the first case.

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

- $a \leq A$ is equivalent to $a = 0 \lor a = 1 < A$, and
- $B < b$ is equivalent to $b \neq A \land B < b – 1$, where we use the model to pick one of the disjuncts in the first case.
We thus create an MBP by substituting $1$ for $\neg P_0$ under side-condition $M \models g < (2^{n-1}) \div y$.

MBP is under-approximation $\mathcal{M} = \{x \mapsto 1\}$, thus we can construct the following MBP:

$$\mathcal{M} = \{x \mapsto 1\}$$

The idea is to under-approximate $\psi$ after checking that $M$ models the under-approximation. We need to check because $(f(x) \div y) \times \alpha < \delta$ might not imply $(f(x) \times \alpha < (d + 1) \times \delta - 1)$.

MBP is under-approximation $\mathcal{M} = \{x \mapsto 1\}$, and the subset $\mathcal{S}$ is applicable to $\psi$.

The MBP is the weakest under-approximation.

Algorithm 2: Lazy MBP computation for BV

In: $\psi(x, y)$, constant $x$, and $M$, such that $M \models \psi$

Out: $\mathcal{M}(y)$, such that $M \models \mathcal{M}$ and $\mathcal{M}(y) = \exists x \cdot \psi(x, y)$

1. Let $\psi = \exists x \cdot \psi(x, y)$

2. $\mathcal{M} = \text{MBP}(M, \psi)$

3. $S \leftarrow \{x_i \mapsto M(x_i)\}$

4. foreach $i$ do

5. if $\mathcal{M} \land \sigma \models \exists x \cdot \psi(x, y)$ then $S \leftarrow S \cup \{x_i \mapsto M(x_i)\}$

6. return $\mathcal{M} \land \sigma \models \exists x \cdot \psi(x, y)$

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

$$\exists x \cdot a \leq x \land x < b \land \text{extract}(x, 7, 7) = 0$$

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 $\mathcal{M} = \mathcal{MBP}(M, a \leq x \land x < b) = a < b$. Note that $\mathcal{M} \models \exists x \cdot \psi$ because, e.g., if $a$ is instantiated with 199, and $b$ with 200, then the only value of $x$ satisfying $a \leq x < b$ 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 $x < b$. Since $\text{extract}(x, 7, 7) = 0$ 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:

$$a < b \land 64 < b \Rightarrow \exists x \cdot \psi$$

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 $a < b \land a \leq 64$.

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 $(\phi, i)$, where $\phi$ is a set of states that can lead to either a bad state, or another POB, at depth $i + 1$. It is reachable if $\phi$ is reachable from $\text{Init}$ at depth $i$. Alg. 3 explains dropping the corresponding substitutions and checking mutual implications. This is expensive, and, thus, we do not do it.
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 \( \pi \) if there is a substitution \( \sigma \) 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 \( \pi \) and a set of formulas \( \Phi \), a cluster \( C_\Phi(\pi) \) is the set of all formulas in \( \Phi \) that matches \( \pi \) with numeric substitutions. We apply the guidance rules on clusters of lemmas.

In practice, given a set of lemmas, 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 \( \pi = \varphi \lor \bigvee_{i=1}^{N} \ell_i \) where \( \varphi \) is a ground formula and each literal \( \ell_i \) is either of the form \( v_i \lor t_i \) or of the form \( v_i \land t_i \) where \( \mathbb{S} = \{ \leq, \leq s, <, < s, = \} \). We also have the additional constraint that all the variables in \( \varphi \) are of the same bit-width.

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

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 \( C_\varphi(\pi') \). Let \( P_k \) be the ordered set of numeric values corresponding to cube \( \ell_k \) in \( C_\varphi(\pi') \). That is \( P_k = (n_1, \ldots, n_N) \) such that \( \ell_k = \land_{i=1}^{N} v_i = n_i \) and \( \pi_k = c_k \). 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 \( C_\varphi(\pi') \). Let \( r \) be a row in a kernel of \( A \). Then we know that \( r^T [v_1, \ldots, v_N, 1]^T = 0 \) is an equality implied by the cubes in \( C_\varphi(\pi') \). 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 \( v \), let \( S_1 \) be the set of values that \( v \) can be substituted into. That is \( S_1 = \{ n \mid \sigma[v_i] = n \land \pi \sigma \in C_\varphi(\pi') \} \). We add a divisibility constraint \( (a \mid (v_i - b)) \) if \( (a \mid (s - b)) \land a \neq 1 \) for all \( s \in S_1 \). If there are multiple such \( a \)'s, the largest one is chosen. Let \( M \) be a conjunction of all such divisibility constraints.

---

**Algorithm 3: Spacer algorithm as a set of guarded commands.** A command can be executed whenever its preconditions are met. We use the shorthand \( \Phi(\varphi) = \varphi' \lor (\varphi \land \text{Tr}) \).

We use \( [x \mapsto y] \) to mean replacing each occurrence of \( x \) in \( \varphi \) with \( y \).

**In:** A transition system \((X, \text{Init}, \text{Tr})\)

**Out:** (safe, \( l_n \)) or unsafe

1. \( Q \leftarrow 0 \) // pob queue

2. \( N \leftarrow 0 \) // maximum safe level

3. \( O_i \leftarrow \text{Init}, O_i \leftarrow \top \) for all \( i > 0 \) // inducive trace

4. \( \text{Tr} \leftarrow \text{Init} \) // reachable states

5. \( \text{Bad} \leftarrow \neg \varphi \) // bad states

6. forever do

   Candidate \( [\text{ssort}(O_N \land \text{Bad})] \)

   7. \( Q \leftarrow Q \cup (\text{Bad, } N) \)

   8. Predecessor \( [\{ (p, p + 1) \in Q, M \mid O_i \land \text{Tr} \land \varphi' \}] \)

   9. \( Q \leftarrow Q \cup (\text{Pre}(X, \text{Tr} \land \varphi'), M, i) \)

   10. Successor \( [\{ (p, p + 1) \in Q, M \mid \text{Tr}(U) \land \varphi' \}] \)

   11. \( U \leftarrow \text{Tr}(U \lor \text{Tr}(I(U), M)[X \mapsto X']) \)

   12. Conflict \( [\{ (p, p + 1) \in Q, \text{Tr}(O_i) \Rightarrow \neg \text{ssort}\}] \)

   13. \( O_i \leftarrow (O_i \land \text{Tr}) \) for all \( i > 0 \)

   14. \( O_i \leftarrow (O_i \land \text{Tr}) \) for all \( i > 0 \)

   15. \( \text{Unfold} \leftarrow \neg \text{Bad} \)

   16. return unsafe, (safe, \( O_i \))

   Unsafe \( [\text{ssort}(\text{Bad} \land \text{Un})] \)

   17. return unsafe
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:

- **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 competitive with Spacer. The results shown here are obtained by running Spacer and G SpacerBV 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 G SpacerBV.

### 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: VAT [7] and Eldarica [2]. G SpacerBV 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 G SpacerBV outperforms the baseline Spacer. However, there are some instances where the baseline Spacer outperforms us. Interestingly,

---

6We refer the reader to [34] for a complete description of designs.

with Counterexample-Guided Abstraction Refinement (CEGAR) [8], Abstraction for hardware [17] is also based on this idea, but it is more instances than 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


