This is the accepted version of a paper presented at *The eighteenth in a series of conferences on the theory and applications of formal methods in hardware and system verification (FMCAD 2018)*.

Citation for the original published paper:

Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-368030
Abstract—The inference of program invariants over machine arithmetic, commonly called bit-vector arithmetic, is an important problem in verification. Techniques that have been successful for unbounded arithmetic, in particular Craig interpolation, have turned out to be difficult to generalise to machine arithmetic: existing bit-vector interpolation approaches are based either on eager translation from bit-vectors to unbounded arithmetic, resulting in complicated constraints that are hard to solve and interpolate, or on bit-blasting to propositional logic, in the process losing all arithmetic structure. We present a new approach to bit-vector interpolation, as well as bit-vector quantifier elimination (QE), that works by lazy translation of bit-vector constraints to unbounded arithmetic. Laziness enables us to fully utilise the information available during proof search (implied by decisions and propagation) in the encoding, and this way produce constraints that can be handled relatively easily by existing interpolation and QE procedures for Presburger arithmetic. The lazy encoding is complemented with a set of native proof rules for bit-vector equations and non-linear (polynomial) constraints, this way minimising the number of cases a solver has to consider.

I. INTRODUCTION

Craig interpolation is a commonly used technique to infer invariants or contracts in verification. Over the last 15 years, efficient interpolation techniques have been developed for a variety of logics and theories, including propositional logic [1], [2], uninterpreted functions [1], [3], [4], first-order logic [5], [6], [7], algebraic data-types [8], linear real arithmetic [9], non-linear real arithmetic [10], Presburger arithmetic [10], [4], [11], and arrays [12], [13], [14].

A theory that has turned out notoriously difficult to handle in Craig interpolation is bounded machine arithmetic, commonly called bit-vector arithmetic. Decision procedures for bit-vectors are predominantly based on bit-blasting, in combination with sophisticated preprocessing and simplification methods, which implies that also extracted interpolants stay on the level of propositional logic and are difficult to map back to compact high-level bit-vector constraints. An alternative interpolation approach translates bit-vector constraints to unbounded integer arithmetic formulas [15], but is limited to linear constraints and tends to produce integer formulas that are hard to solve and interpolate, due to the necessary introduction of additional variables and large coefficients to model wrap-around semantics correctly.

In this paper, we introduce a new Craig interpolation method for bit-vector arithmetic, focusing on arithmetic bit-vector operations including addition, multiplication, and division. Like [15], we compute interpolants by reducing bit-vectors to unbounded integers; unlike in earlier approaches, we define a calculus that carries out this reduction lazily, and can therefore dynamically choose between multiple possible encodings of the bit-vector operations. This is done by initially representing bit-vector operations as uninterpreted predicates, which are expanded and replaced by Presburger arithmetic expressions on demand. The calculus also includes native rules for non-linear constraints and bit-vector equations, so that formulas can often be proven without having to resort to a full encoding as integer constraints. Our approach gives rise to both Craig interpolation and quantifier elimination (QE) methods for bit-vector constraints, with both procedures displaying competitive performance in our experiments.

- Reduction of bit-vectors to unbounded integers has the additional advantage that integer and bit-vector formulas can be combined efficiently, including the use of conversion functions between both theories, which are difficult to support using bit-blasting. This combination is of practical importance in software verification, since programs and specifications often mix machine arithmetic with arbitrary-precision numbers; tools might also want to switch between integer semantics (if it is known that no overflows can happen) and bit-vector semantics for each individual program instruction.

The contributions of the paper are: 1) a new calculus for non-linear integer arithmetic, which can eliminate quantifiers (in certain cases) and extract Craig interpolants (Section III); 2) a corresponding calculus for arithmetic bit-vector constraints (Section IV); 3) an experimental evaluation using SMT-LIB and model checking benchmarks (Section V).

A. Related Work

Most SMT solvers handle bit-vectors using bit-blasting and SAT solving, and usually cannot extract interpolants for bit-vector problems. The exception is MathSAT [16], which uses a layered approach [15] to compute interpolants: MathSAT first tries to compute interpolants by keeping bit-vector operations uninterpreted; then using a restricted form of quantifier elimination; then by eager encoding into linear integer arithmetic (LIA); and finally through bit-blasting. Our approach has some similarities to the LIA encoding, but can choose simpler encodings thanks to laziness, and also covers non-linear arithmetic constraints.

Other related work has focused on fragments of bit-vector logic. In [17], an algorithm is given for reconstructing bit-vector interpolants from bit-level interpolants, however restricted to the case of bit-vector equalities. An interpolation
procedure based on a set of tailor-made (but incomplete) rewriting rules for bit-vectors is given in [18].

II. PRELIMINARIES: THE BASE LOGIC

We formulate our approach on top of a simple logic of Presburger arithmetic constraints combined with uninterpreted predicates, introduced in [19] and extended in [4], [10] to support Craig interpolation. Let $x$ range over an infinite set $X$ of variables, $c$ over an infinite set $C$ of constants, $p$ over a set $P$ of uninterpreted predicates with fixed arity, and $\alpha$ over the set $\mathbb{Z}$ of integers. The syntax of terms and formulæ is defined by the following grammar:

$$\begin{align*}
\phi &::= t = 0 \mid t \leq 0 \mid p(t, \ldots, t) \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \forall x.\phi \mid \exists x.\phi
\end{align*}$$

t ::= \alpha \mid c \mid x \mid at + \cdots + at$

The symbol $t$ denotes terms of linear arithmetic. Substitution of a term $t$ for a variable $x$ in $\phi$ is denoted by $\phi[t/x]$; we assume that variable capture is avoided by renaming bound variables as necessary. For simplicity, we sometimes write $s = t$ as a shorthand of $s - t = 0$, inequality $s \leq t$ for $s - t \leq 0$, and $\forall c.\phi$ as a shorthand of $\forall x,c/x.\phi$ if $c$ is a constant. The abbreviation $\texttt{true}$ ($\texttt{false}$) stands for the equality $0 = 0$ ($1 = 0$), and the formula $\phi \rightarrow \psi$ abbreviates $\neg\phi \lor \psi$.

Semantic notions such as structures, models, satisfiability, and validity are defined as is common (e.g., [20]), but we assume that evaluation always happens over the universe $\mathbb{Z}$ of integers; bit-vectors will later be defined as a subset of the integers.

A. A Sequent Calculus for the Base Logic

For checking whether a formula in the base logic is satisfiable or valid, we work with the calculus presented in [19], a part of which is shown in Fig. 1. If $\Gamma, \Delta$ are sets of formulæ, then $\Gamma \vdash \Delta$ is a sequent. A sequent is valid if the formula $\Lambda \Delta \rightarrow \Delta$ is valid. Positions in $\Delta$ that are underneath an even/odd number of negations are called positive/negative; and vice versa for $\Gamma$. Proofs are trees growing upward, in which each node is labelled with a sequent, and each non-leaf node is related to the node(s) directly above it through an application of a calculus rule. A proof is closed if it is finite and all leaves are justified by an instance of a rule without premises. Soundness of the calculus implies that the root of a closed proof is a valid sequent.

In addition to propositional and quantifier rules in Fig. 1, the calculus in [19] also includes rules for equations and inequalities in Presburger arithmetic; the details of those rules are not relevant for this paper. The calculus is complete for quantifier-free formulæ in the base logic, i.e., for every valid quantifier-free sequent a closed proof can be found. It is well-known that the base logic including quantifiers does not admit complete calculi [21], but as discussed in [19] the calculus can be made complete (by adding slightly more sophisticated quantifier handling) for interesting undecided fragments, for instance for sequents $\vdash \phi$ with only existential quantifiers.

For quantifier-free input formulæ, proof search can be implemented in depth-first style following the core concepts of DPLL(T) [22]: rules with multiple premises correspond to decisions and explore the branches one by one; rules with a single premise represent propagation or rewriting; and logging of rule applications is used in order to implement conflict-driven learning and proof extraction. For experiments, we use the implementation of the calculus in PRINCESS.1

B. Quantifier Elimination in the Base Logic

The sequent calculus can eliminate quantifiers in Presburger arithmetic, i.e., in the base logic without uninterpreted predicates, since the arithmetic calculus rules are designed to systematically eliminate constants. To illustrate this use case, suppose $\phi$ is a formula without uninterpreted predicates and without constants $c$, but possibly containing variables $x$. Formula $\phi$ furthermore only contains $\forall/\exists$ under an even/odd number of negations, i.e., all quantifiers are effectively universal.

To compute a quantifier-free formula $\psi$ that is equivalent to $\phi$, we can construct a proof with root sequent $\vdash \phi$, and keep applying rules until no further applications are possible in any of the remaining open goals $\{\Gamma_i \vdash \Delta_i \mid i = 1, \ldots, n\}$. In this process, rules $\exists$-LEFT and $\forall$-RIGHT introduce fresh constants, which are subsequently isolated and eliminated by the arithmetic rules. To find $\psi$, it is essentially enough to extract the constant-free formulæ $\Gamma_i^v \subseteq \Gamma_i$, $\Delta_i^v \subseteq \Delta_i$ in the open goals, and construct $\psi = \bigwedge_{i=1}^n (\bigwedge \Gamma_i^v \rightarrow \bigvee \Delta_i^v)$.

The full calculus [19] is moreover able to eliminate arbitrarily nested quantifiers, and can be used similarly to prove validity of sequents with quantifiers. A recent independent evaluation [23] showed that the resulting proof procedure is competitive with state-of-the-art SMT solvers and theorem provers on a wide range of quantified integer problems.
C. Craig Interpolation in the Base Logic

Given formulas $A$ and $B$ such that $A \land B$ is unsatisfiable, Craig interpolation can determine a formula $I$ such that the implications $A \Rightarrow I$ and $B \Rightarrow \neg I$ hold, and non-logical symbols in $I$ occur in both $A$ and $B$ [24]. An interpolating version of our sequent calculus has been presented in [4], [10], and is summarised in Fig. 2. To keep track of the partitions $A$, $B$, the calculus operates on labelled formulas $[\phi]_L$ (with $L$ for “left”) to indicate that $\phi$ is derived from $A$, and similarly formulas $[\phi]_R$ for $\phi$ derived from $B$. If $\Gamma, \Delta$ are finite sets of linear equations implied by a set of linear literals, and $I$ is an unlabelled formula, then $\Gamma \vdash \Delta \triangleright I$ is an interpolating sequent.

Semantics of interpolating sequents is defined using projections $\Gamma_L = \{\phi \mid [\phi]_L \in \Gamma\}$ and $\Gamma_R = \{\phi \mid [\phi]_R \in \Gamma\}$, which extract the $L/R$-parts of a set $\Gamma$ of labelled formulae. A sequent $\Gamma \vdash \Delta \triangleright I$ is valid if 1) the sequent $\Gamma_L \vdash I$, $\Delta_L$ is valid, 2) the sequent $\Gamma_R, I \vdash \Delta_R$ is valid, and 3) the constants and uninterpreted predicates/functions in $I$ occur in both $\Gamma_L \cup \Delta_L$ and $\Gamma_R \cup \Delta_R$. As a special case, note that the sequent $[A]_L, [B]_R \vdash \emptyset \triangleright I$ is valid iff $I$ is an interpolant of $A \land B$. Soundness of the calculus guarantees that the root of a closed interpolating proof is a valid interpolating sequent.

To solve an interpolation problem $A \land B$, a prover typically first constructs a proof of $A, B \vdash \emptyset$ using the ordinary calculus from Section II-A. Once a closed proof has been found, it can be lifted to an interpolating proof: this is done by replacing the root formulas $A, B$ with $[A]_L, [B]_R$, respectively, and recursively assigning labels to all other formulas as defined by the rules from Fig. 2. Then, starting from the leaves, intermediate interpolants are computed and propagated back to the root, leading to an interpolating sequent $[A]_L, [B]_R \vdash \emptyset \triangleright I$.

III. SOLVING NON-LINEAR CONSTRAINTS

We extend the base logic in two steps: in this section, symbols and rules are added to solve non-linear diophantine problems; a second extension is then done in Section IV to handle arithmetic bit-vector constraints. Both constructions preserve the ability of the calculus to eliminate quantifiers (under certain assumptions) and derive Craig interpolants.

For non-linear constraints, we assume that the set $P$ of uninterpreted predicates contains a distinguish ternary predicate $\times$, with the intended semantics that the third argument represents the result of multiplying the first two arguments, i.e., $x(s, t, r) \iff s \cdot t = r$. The predicate $\times$ is clearly sufficient to express arbitrary polynomial constraints by introducing a $\times$-literal for each product in a formula, at the cost of introducing a linear number of additional constants or existentially quantified variables. We make the simplifying assumption that $\times$ only occurs in negative positions; that means, top-level occurrences will be on the left-hand side of sequents. Positive occurrences can be eliminated thanks to the equivalence $\neg x(s, t, r) \iff \exists x. (x(s, t, x) \land x \neq r)$.

A. Calculus Rules for Non-Linear Constraints

We now introduce different classes of calculus rules to reason about the $\times$-predicate. The rules are necessarily incomplete for proving that a sequent is valid, but they are complete for finding counterexamples: if $\phi$ is a satisfiable quantifier-free formula with $\times$ as the only uninterpreted predicate, then it is possible to construct a proof for $\phi \vdash \emptyset$ that has an open and unprovable goal in pure Presburger arithmetic (by systematically splitting variable domains, Section III-A4).

1) Deriving Implied Equalities with Gröbner Bases:
The first rule applies standard algebra methods to infer new equalities from multiplication literals. To avoid the computation of more and more complex terms in this process, we restrict the calculus to the inference of linear equations that can be derived through computation of a Gröbner basis. 2 Given a set $\{x(s_i, t_i, r_i)\}_{i=1}^n$ of $\times$-literals and a set $\{e_j = 0\}_{j=1}^m$ of linear equations, the generated ideal $I = (\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$ over rational numbers is the smallest set of rational polynomials that contains $\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m$, is closed under addition, and closed under multiplication with arbitrary rational polynomials [25]. Any $f \in I$ corresponds to an

\[1\text{http://www.phillipp.ruemmer.org/princess.shtml} \]
equation \( f = 0 \) that logically follows from the literals, and can therefore be added to a proof goal:

\[
\Gamma, \{ (s_i, t_i, r_i) \}_{i=1}^n, \{ e_j = 0 \}_{j=1}^m, f = 0 \vdash \Delta \quad \text{-}\times\text{-EQ}
\]

if \( f \) is linear, has integer coefficients, and \( f \in I \).

To see how this rule can be applied practically, note that the subset of linear polynomials in \( I \) forms a rational vector space, and therefore has a finite basis. It is enough to apply \( \times\text{-EQ} \) for terms \( f_1, \ldots, f_k \) corresponding to any such basis, since linear arithmetic reasoning (in the base logic) will then be able to derive all other linear polynomials in \( I \). To compute a basis \( f_1, \ldots, f_k \), we can transform \( \{ (s_i, t_i, r_i) \}_{i=1}^n \cup \{ e_j = 0 \}_{j=1}^m \) to a Gröbner basis using Buchberger’s algorithm [26], and then apply Gaussian elimination to find linear basis polynomials (or directly by choosing a suitable monomial order).

**Example 1:** Consider the square of a sum: \( (x + y)^2 = x^2 + 2xy + y^2 \). This can be proven in the following way. We begin by rewriting the equation to normal form, let \( \Pi = \{ (x, x, c_1), (x, y, c_2), (y, y, c_3), (x + y, x + y, c_4) \} \):

\[
\Pi, c_1 + 2c_2 + c_3 - c_4 = 0 \vdash c_4 = c_1 + 2c_2 + c_3 \quad \times\text{-EQ}
\]

Here, the \( \times\text{-EQ} \)-step is motivated by the fact that the Gröbner basis derived from \( \Pi \) contains the linear polynomial \( c_1 + 2c_2 + c_3 - c_4 \), from which the desired equation can be derived using linear reasoning.

2) **Interval Constraint Propagation (ICP):** Our main technique for inequality reasoning in the presence of \( \times \)-predicates is interval constraint propagation (ICP) [27], which computes greatest fixed-points over-approximating the ranges of constants or free variables. Due to lack of space we do not introduce ICP in full detail, but only assume that \( \text{Prop}_{\{\phi_1, \ldots, \phi_n\}} \) is a monotonic function describing the propagation of bounds information implied by equalities, inequalities, and \( \times \)-literals \( \phi_1, \ldots, \phi_n \), and \( \text{gfp Prop}_{\{\phi_1, \ldots, \phi_n\}} \) is its greatest fixed-point. The ICP rule adds resulting bounds for a constant or variable \( c \in C \cup X \):

\[
\Gamma, \text{Prop}_{\{\phi_1, \ldots, \phi_n\}}(c) = [l, u] \quad \text{\( \times\text{-ICP} \)}
\]

**Example 2:** From two inequalities \( x \geq 5 \) and \( y \geq 5 \), the rule \( \times\text{-ICP} \) can derive \( (x + y)^2 \geq 100 \):

\[
\times(\{x + y, x + y, c_4\}, x \geq 5, y \geq 5, 100 \leq c_4) \vdash \times(\{x + y, x + y, c_4\}, x \geq 5, y \geq 5) \quad \times\text{-EQ}
\]

The slightly different problem \( x + y \geq 10 \implies (x + y)^2 \geq 100 \) cannot be proven in the same way, since ICP will not be able to deduce bounds for \( x \) or \( y \) from \( x + y \geq 10 \).

3) **Cross-Multiplication of Inequalities:** While ICP is highly effective for approximating the range of constants, and quickly detecting inconsistencies, it is less useful for inferring relationships between multiple constants that follow from multiplication literals. We cover such inferences using a cross-multiplication rule that resembles procedures used in ACL2 [28]. The rule captures the fact that if \( s, t \) are both non-negative, then also the product \( s \cdot t \) is non-negative.

Like in Section III-A1, we prefer to avoid the introduction of new multiplication literals during proof search, and only add \( s \cdot t \geq 0 \) if the term \( s \cdot t \) can be expressed linearly. For this, we again write \( I = \{ (s_i, t_i, r_i) \}_{i=1}^n \cup \{ e_j = 0 \}_{j=1}^m \) for the ideal induced by equations and \( \times \)-literals:

\[
\Gamma, s \leq 0, t \leq 0, -f \leq 0 \vdash \Delta \quad \times\text{-CROSS}
\]

if \( f \) is linear, has integer coefficients, and \( s \cdot t - f \in I \).

The term \( f \) can practically be found by computing a Gröbner basis of \( I \), and reducing the product \( s \cdot t \) to check whether an equivalent linear term exists.

4) **Interval Splitting:** If everything else fails, as last resort it can become necessary to systematically split over the possible values of a variable or constant \( c \in C \cup X \):

\[
\Gamma, c \leq \alpha - 1 \vdash \Delta \quad \Gamma, c \geq \alpha \vdash \Delta \quad \times\text{-SPLIT}
\]

The \( \alpha \in \mathbb{Z} \) can in principle be chosen arbitrarily in the rule, but in practice a useful strategy is to make use of the range information derived for \( \times\text{-ICP} \); when no ranges can be tightened any further using \( \times\text{-ICP} \), instead \( \times\text{-SPLIT} \) can be applied to split one of the intervals in half.

5) **\( \times\text{-Elimination} \):** Finally, occurrences of \( \times \) can be eliminated whenever a formula is subsumed by other literals in a goal, again writing \( I = \{ (s_i, t_i, r_i) \}_{i=1}^n \cup \{ e_j = 0 \}_{j=1}^m \):

\[
\Gamma \vdash \Delta \quad \Gamma, \times(s, t, r) \vdash \Delta \quad \times\text{-ELIM}
\]

if \( s \cdot t - r \in I \).

Note that \( \times\text{-ELIM} \) only eliminates non-linear \( \times \)-literals, whereas \( \times\text{-EQ} \) only introduces linear \( \times \)-literals, so that the application of the two rules cannot induce cycles.

B. **Quantifier Elimination for Non-Linear Constraints**

Due to necessary incompleteness of calculi for Peano arithmetic, quantifiers can in general not be eliminated in the presence of the \( \times \) predicate, even when considering formulas that do not contain other uninterpreted predicates. By combining the QE approach in Section II-B with the rules for \( \times \) that we have introduced, it is nevertheless possible to reason about quantified non-linear constraints in many practical cases, and sometimes even get rid of quantifiers. This is possible because the rules in Section III-A are not only sound, but even equivalence transformations: in any application of the rules, the conjunction of the premises is equivalent to the conclusion.
Similarly as in [29], QE is always possible if sufficiently many constants or variables in a formula \( \phi \) range over bounded domains: if there is a set \( B \subseteq C \cup X \) of symbols with bounded domain such that in each literal \( \times (s, t, r) \) either \( s \) or \( t \) contain only symbols from \( B \). In this case, proof construction will terminate when applying the rule \( \times \)-SPLIT only to variables or constants with bounded domain. This guarantees that eventually every literal \( \times (s, t, r) \) can be turned into a linear equation using \( \times \)-EQ, and then be eliminated using \( \times \)-ELIM, only leaving proof goals with pure Presburger arithmetic constraints. The boundedness condition is naturally satisfied for bit-vector formulas.

C. Craig Interpolation for Non-Linear Constraints

To carry over the Craig interpolation approach from Section II-C to non-linear formulas, interpolating versions of the calculus rules for the \( \times \)-predicate are needed. For this, we follow the approach used in [4] (which in turn resembles the use of theory lemmas in SMT in general): when translating a proof to an interpolating proof, we replace applications of the \( \times \)-rules with instantiation of an equivalent theory axiom \( QAx \). Suppose a non-interpolating proof contains a rule application

\[
\frac{\Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta \ldots \Gamma, \Gamma', \Gamma_2 \vdash \Delta_1, \Delta', \Delta}{\Gamma, \Gamma' \vdash \Delta', \Delta}
\]


\[
\frac{\Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta \ldots \Gamma, \Gamma', \Gamma_2 \vdash \Delta_1, \Delta', \Delta}{\Gamma, \Gamma' \vdash \Delta', \Delta'}
\]

in which \( \Gamma', \Delta' \) are the formulas assumed by the rule application, \( \Gamma, \Delta \) are side formulas not required or affected by the application, and \( \Gamma_1, \Delta_1, \ldots, \Gamma_n, \Delta_n \) are newly introduced formulas in the individual branches.

The (quantified) theory axiom \( Ax \) corresponding to the rule application expresses that the conjunction of the premises has to imply the conclusion; the quantified theory axiom \( QAx =_{def} \forall S. Ax \) in addition contains universal quantifiers for all constants \( S \subseteq C \) occurring in \( Ax \).

\[ Ax =_{def} \bigwedge_{i=1}^{n} (\bigwedge \Gamma_i \rightarrow \bigvee \Delta_i) \rightarrow (\bigwedge \Gamma' \rightarrow \bigvee \Delta') \]

\( Ax \) and \( QAx \) are specific to the application of \( R \): the axioms for two distinct applications of \( R \) will in general be different formulas. \( QAx \) is defined in such a way that the effect of \( R \) can be simulated by introducing \( QAx \) in the antecedent, instantiating it with the right constants, and applying propositional rules:

\[
\frac{\Gamma, \Gamma', \Delta_1 \vdash \Delta_1, \Delta', \Delta \ldots \Gamma, \Gamma' \vdash \Delta', \Delta \ldots}{\Gamma, \Gamma', Ax \vdash \Delta', \Delta} \quad \vdash \text{-LEFT*}
\]

This construction leads to a proof using only the standard rules from Section II-A, which can be interpolated as discussed earlier. Since \( QAx \) is a valid formula not containing any constants, it can be introduced in a proof at any point, and labelled \( [QAx]_L \) or \( [QAx]_R \) on demand.

The obvious downside of this approach is the possibility of quantifiers occurring in interpolants. The interpolating rules \( \forall \text{-LEFT}^{L/R} \) (Fig. 2) have to introduce quantifiers \( \forall \mu_{L/R} \) for local symbols occurring in the substituted term \( t \): whether such quantifiers actually occur in the final interpolant depends on the applied \( \times \)-rules, and on the order of rule application. For instance, with \( \times \)-SPLIT it is always possible to choose the label of \( QAx \) so that no quantifiers are needed, whereas \( \times \)-EQ might mix symbols from left and right partitions in such a way that quantifiers become unavoidable. In our implementation we approach this issue pragmatically. We leave proof search unrestricted, and might thus sometimes get proofs that do not give rise to quantifier-free interpolants; when that happens, we afterwards apply QE to get rid of the quantifiers. QE is always possible for bit-vector constraints, see Section IV-D.\(^3\)

IV. Solving Bit-Vector Constraints

We now define the extension of the base logic to bit-vector constraints. The main idea of the extension is to represent bit-vectors of width \( w \) as integers in the interval \( \{0, \ldots, 2^w - 1\} \), and to translate bit-vector operations to the corresponding operation in Presburger arithmetic (or possible the \( \times \)-predicate for non-linear formulas), followed by an integer remainder operation to map the result back to the correct bit-vector domain. Since the remainder operation tends to be a bottleneck for interpolation, we keep the operation symbolic and initially consider it as an uninterpreted predicate \( bmod^b \). The predicate is only gradually reduced to Presburger arithmetic by applying the calculus rules introduced later in this section.

Formally, we introduce a set \( P_{bw} = \{ bmod^b_a | a, b \in \mathbb{Z}, a < b \} \) of binary predicates. The semantics of \( bmod^b \) is to relate any whole number \( x \in \mathbb{Z} \) to its remainder modulo \( b-a \) in the interval \( \{a, \ldots, b-1\} \):

\[
\begin{align*}
\text{bmod}^b(s, r) &\iff a \leq r < b \land \exists z. r = s + (b-a) \cdot z \\
&\iff a \leq r < b \land r \equiv s \pmod{b-a}
\end{align*}
\]

We also introduce short-hand notations for the casts to the unsigned and signed bit-vector domains:

\[ \text{ubmod}^w_w =_{def} bmod^w_0, \quad \text{sbmod}^w_w =_{def} bmod^w_{2^w-1} \]

A. Translating Bit-Vector Constraints to the Core Language

For the rest of the section, we use the base logic augmented with \( \times \) and \( bmod^b \)-predicates as the core language to which bit-vector constraints are translated. For presentation, the translation focuses on a subset of the arithmetic bit-vector operations, \( \text{BVOP} = \{ \text{bvadd}_w, \text{bvmul}_w, \text{bvddiv}_w, \text{bvneg}_w, \text{ze}_{w+w}, \text{bvule}_w, \text{bvslt}_w \} \). All operations are sub-scripted with the bit-width of the operands; the zero-extend function \( \text{ze}_{w+w} \) maps bit-vectors of width \( w \) to width \( w+w \).

\(^3\)Non-linear integer arithmetic in general does not admit quantifier-free interpolants. For instance, \( (x > 1) \land x = y^2) \land x = z^2 + 1 \) is unsatisfiable, but no quantifier-free interpolants exist, regardless of whether divisibility predicates \( a \mid t \) are allowed or not.
follows the FixedSizeBitVectors\(^4\) theory of the SMT-LIB [30]. Other arithmetic operations, for instance bsvdiv\(_w\) or bsvmod\(_w\), can be handled in the same way as shown here, though sometimes the number of cases to be considered is larger.

The translation from bit-vector constraints \(\phi\) to core formulas \(\phi\text{\textsubscript{core}}\) has two parts: first, BVOP occurrences in a formula \(\phi\) have to be replaced with equivalent expressions in the core language; second, since the core language only knows the sort of unbounded integers, type information has to be made explicit by adding domain constraints.

a) BVOP elimination: Like in Section III, we assume that the bit-vector formula \(\phi\) has already been brought into a flat form by introducing additional constants or quantified variables: the operations in BVOP must not occur nested, and functions only occur in equations of the form \(f(s) = t\) in negative positions. The translation from \(\phi\) to \(\phi'\) is then defined by the rewriting rules in Fig. 3. Since the rules for the predicate bsvle\(_w\) distinguish between positive and negative occurrences, we assume that rules are only applied to formulas in negation normal-form, and only in negative positions.

The rules for bv\(_{\text{add}}\), bvneg\(_w\), ze\(_w\), and bvu\(_w\) simply translate to the corresponding Presburger term, if necessary followed by remainder \(u\text{mod}_{\text{w}}\). Multiplication \(b\text{mul}_{\text{w}}\) is mapped similarly to the \(\times\)-predicate defined in Section III, adding an existential quantifier to store the intermediate product. Since rules are only applied in negative positions, the quantified variable can later be replaced with a Skolem constant. An optimised rule could be defined for the case that one of the factors is constant, avoiding the use of the \(\times\)-predicate. Translation of bsvle\(_w\) maps the operands to a signed bit-vector domain \([-2^{w-1}-1,2^{w-1}-1\). The rule for unsigned division bvu\(_w\) distinguishes the cases that the divisor \(t\) is zero or positive (as required by SMT-LIB), and maps the latter case to standard integer division.

b) Domain constraints: Bit-vector variables/constants \(x\) of width \(w\) occurring in \(\phi\) are interpreted as unbounded integer variables in \(\phi\text{\textsubscript{core}}\), which therefore has to contain explicit assumptions about the ranges of bit-vector variables. We use the abbreviation \(in_{\text{w}}(x) = \text{def} (0 \leq x < 2^w)\) and define

\[ \phi_{\text{core}} = \bigwedge_{x \in S} in_{\text{w}}(x) \rightarrow \phi' \]

where \(S \subseteq C \cup X\) is the set of free variables and constants occurring in \(\phi\), \(w_x\) is the bit-width of \(x \in S\), and \(\phi'\) is the result of applying rules from Fig. 3 to \(\phi\). Similar constraints

are used to express quantification over bit-vectors, for instance

\[ \exists x. (in_{\text{w}}(x) \wedge \ldots) \quad \text{and} \quad \forall x. (in_{\text{w}}(x) \rightarrow \ldots) \]

\[ \exists x. \varphi(x, y) \rightarrow \exists y. \varphi(x, y) \]

\[ \exists x. \varphi(x) \rightarrow \exists y. \varphi(y) \]

Example 3: We consider the SMT-LIB QF_BV problem challenge/multiplyOverflow.smt2, a bit-vector formula that is known to be hard for most SMT solvers since it contains both multiplication and division. In experiments, neither Z3 nor CVC4 could prove the formula within 10min.

In our notation, the problem amounts to showing validity of the following implication, with \(a, b\) ranging over bit-vectors of width 32:

\[ \text{bvule}_{32}(b, \text{bvdiv}_{32}(2^{32} - 1, a)) \rightarrow \text{bvule}_{64}(\text{bmul}_{64}(\text{ze}_{32+32}(a), \text{ze}_{32+32}(b)), 2^{32} - 1) \]

As a flat formula, with additional constants \(c_1\) of width 32 and \(c_2, c_3, c_4\) of width 64, the implication takes the form:

\[ \text{bvdiv}_{32}(2^{32} - 1, a) = c_1 \wedge \text{bmul}_{64}(c_3, c_4) = c_2 \wedge \text{ze}_{32+32}(a) = c_3 \wedge \text{ze}_{32+32}(b) = c_4 \wedge \text{bvule}_{32}(b, c_1) \rightarrow \text{bvule}_{64}(c_2, 2^{32} - 1) \]

The final formula \(\phi_{\text{core}}\) is obtained by application of the rules in Fig. 3, and adding domain constraints:

\[ \text{in}_{32}(a) \wedge \text{in}_{32}(b) \wedge \text{in}_{64}(c_2) \wedge \text{in}_{64}(c_3) \wedge \text{in}_{64}(c_4) \wedge \left((a = 0 \wedge c_1 = 2^{32} - 1) \lor \left(a \geq 1 \wedge \exists x. (x(a, c_1, x) = 2^{32} - 1 - a < x \leq 2^{32} - 1)\right)\right) \land \exists z. (x(c_3, c_4, z) \wedge \text{bmul}_{64}(z, c_2)) \land a = c_3 \land b = c_4 \land b \leq c_1 \rightarrow c_2 = 2^{32} - 1 \]

B. Preprocessing and Simplification

An encoded formula \(\phi_{\text{core}}\) tends to contain a lot of redundancy, in particular nested or unnecessary occurrences of the \(\text{bmod}\) predicates. As an important component of our calculus, and in line with the approach in other bit-vector solvers, we therefore apply simplification rules both during preprocessing and during the solving phase (“inprocessing”). The most important simplification rules are shown in Fig. 4. Our implementation in addition applies rules for Boolean and Presburger connectives.

The notation \(\Pi : \phi \rightarrow \phi'\) expresses that formula \(\phi\) can be rewritten to \(\phi'\), given the set \(\Pi\) of formulas as context. The structural rules in the upper half of Fig. 4 define how formulas are traversed, and how the context \(\Pi\) is extended to \(\Pi, Lit'\) when encountering further literals. We apply the structural rules modulo associativity and commutativity of \(\land, \lor\), and prioritise LIT-\(\land\)-RW and LIT-\(\lor\)-RW over the other

\[
\begin{align*}
\text{bvadd}_w(s, t) &= r \rightarrow u\text{mod}_w(s + t, r) \\
\text{bmul}_w(s, t) &= r \rightarrow \exists x. (x(s, t, x) \land u\text{mod}_w(x, r)) \\
\text{bsle}_w(s, t) &= r \rightarrow \exists x, y. (\text{bsh}_{s}(s, x) \land \text{sbmul}_{w}(t, y) \land x \leq y) \\
\text{bvu}_{w}(s, t) &= r \rightarrow s \leq t \\
\text{bvu}_{w}(s, t) &= r \rightarrow s > t \\
\text{bvu}_{w}(s, t) &= r \rightarrow (t = 0 \land r = 2^w - 1) \lor (t \geq 1 \land \exists x. (x(t, r, x) \land s - t < x \leq s)) \\
\end{align*}
\]

\(\text{Fig. 3. Rules translating bit-vector operations into the core language. The rules only apply in negative positions.}\)
Fig. 4. Simplification rules for bit-vector formulas. In \( \circ \)-RW, \( \phi \) and \( \psi \) are not literals, and \( \circ \in \{\land, \lor\} \). In LIT-\( \land \)-RW and LIT-\( \lor \)-RW, the formula LIT is a literal. In Q-RW, \( x \) must not occur in \( P \), and \( Q \in \{\lor, \land\} \). In COEFF-RW, all constants or variables in \( t \) also occur in \( s \).

rules. Simplification is iterated until a fixed-point is reached and no further rewriting is possible. The connection between rewriting rules and the sequent calculus is established by the following rules:

\[
\frac{\Pi : \phi \to \phi'}{\Pi : \psi \to \psi'} \quad \circ\text{-RW} \\
\frac{\Pi : \text{Lit} \to \text{Lit}}{\Pi : \text{Lit} \land \phi \to \text{Lit} \land \phi'} \quad \text{LIT}\land\text{-RW} \\
\frac{\Pi : \text{Lit} \to \text{Lit}'}{\Pi : \text{Lit} \lor \phi \to \text{Lit} \lor \phi'} \quad \text{LIT}\lor\text{-RW} \\
\frac{\Pi : \phi \to \phi'}{\Pi : \neg \phi \to \neg \phi'} \quad \neg\text{-RW} \\
\frac{\Pi : Q \cdot \phi \to \phi, \phi' \to \phi'}{\Pi : Q \cdot \phi \to \phi'} \quad \text{Q-RW}
\]

Fig. 4. Simplification rules for bit-vector formulas. In \( \circ \)-RW, \( \phi \) and \( \psi \) are not literals, and \( \circ \in \{\land, \lor\} \). In LIT-\( \land \)-RW and LIT-\( \lor \)-RW, the formula LIT is a literal. In Q-RW, \( x \) must not occur in \( P \), and \( Q \in \{\lor, \land\} \). In COEFF-RW, all constants or variables in \( t \) also occur in \( s \).

The lower half of Fig. 4 shows three of the bit-vector-

The value of \( s' - r' \) is known to be a multiple of \( b' - a' \), and therefore \( k \cdot (s' - r') \) is a multiple of \( b - a \) provided that \( b - a \) divides \( k \cdot (b' - a') \). This implies that the truth value of \( bmod_r^a(s, r) \) is not affected by adding \( k \cdot (s' - r') \) to \( s \).

Our implementation uses various further simplification rules, for instance to eliminate \( \times \) or \( \bmod \) whose result is never used; we skip those for lack of space.

Example 4: The expression \( \text{bvadd}_32(b, c) \) corresponds to \( \text{ubmod}_32(a + b, r_1) \land \text{ubmod}_32(r_1 + c, r_2) \) in the core language. Using BMOD-RW, the formula can be rewritten to \( \text{ubmod}_32(a + b, r_1) \land \text{ubmod}_32(a + b + c, r_2) \), provided that \( a + b + c < r_1 + c \).

Example 5: We continue Ex. 3 and show that \( \phi_{core} \) is valid, focusing on the \( a \geq 1 \) case of \( \text{bvdv}_32 \). The proof (Fig. 5) consists of three core steps: 1) using \( \times\text{-ICP} \), from the constraints \( in_32(a), in_32(b) \), \( (a, b, d) \) the inequalities \( 0 \leq d \) and \( d \leq 2^{32} - 2^{33} - 1 \) can be derived; 2) therefore, using RW-LEFT and BOUND-RW, the literal \( \text{ubmod}_{64}(d, c) \) can be rewritten to \( d = c_2 \), capturing the fact that 64-bit multiplication cannot overflow for unsigned 32-bit operands; 3) using \( \times\text{-CROSS} \), from the inequalities \( a \geq 1 \) and \( b \leq c_1 \) and the products \( (a, b, d) \), \( (a, c_1, e) \) we can derive \( e - d - c_1 + b \geq 0 \). The proof branch can then be closed using standard arithmetic reasoning. The implementation of our procedure can easily find the outlined proof automatically.

C. Splitting Rules for \( bmod_a^b \)

In general, formulas will of course also contain occurrences of \( bmod_a^b \) that cannot be eliminated just by simplification. We introduce two calculus rules for reasoning about such general literals \( bmod_a^b(s, r) \). The first rule makes the assumption that lower and upper bounds of \( s \) are available, and are reasonably tight, so that an explicit case analysis can be carried out; the rule generalises BOUND-RW to the situation in which the factors \( l, u \) do not coincide:

\[
\frac{\{ \Gamma, a \leq r < b, s = r + i \cdot (b - a) \vdash \Delta \}^u_{\forall i} \text{BMOD-SPLIT}}{\Gamma, bmod_a^b(s, r) \vdash \Delta}
\]

assuming \( \left\lfloor \text{bound}(II, s - a) \right\rfloor = l \) and \( \left\lceil \text{bound}(II, s - a) \right\rceil = u \) with \( \Pi = \Gamma \cup \{ \neg \psi \mid \psi \in \Delta \} \).

If the bounds \( l, u \) are too far apart, the number of cases created by BMOD-SPLIT would become unmanageable, and it
TABLE I

| Categories | Total | ELDARICA math | | | | ELDARICA ilp32 | | | | CPACHECKER - 32 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| All | 551 | 293 | 21.0 | 11.1 | 1.0 | 217 | 28.0 | 13.6 | 1.4 | 180 | 30.6 | 28.5 |
| HOLA | 46 | 101 | 73.4 | 31.8 | 1.0 | 117 | 49.7 | 21.7 | 1.2 | 168 | 48.6 | 3.9 |
| Iireve | 21 | 4 | 11.4 | 8.9 | 1.1 | 4 | 6.0 | 0.0 | 1.3 | 4 | 11.4 | 0.0 |
| VeriMAP | 155 | 132 | 5.8 | 2.3 | 1.0 | 8 | 17.4 | 27.3 | 1.6 | 100 | 5.9 | 3.6 | 1.1 |
| SVCOMP | 329 | 101 | 46.1 | 22.9 | 1.0 | 88 | 58.1 | 25.7 | 1.3 | 74 | 44.0 | 26.3 |

is better to choose a direct encoding of the remainder operation in Presburger arithmetic:

$$\Gamma, a \leq r < b, s = r + (b - a) \cdot c \vdash \Delta$$

$$\Gamma, bmod^R(s, r) \vdash \Delta$$

where c is assumed to be a fresh constant. Rule BMOD-CONST corresponds to the encoding chosen in [15].

In practice, it turns out to be advantageous to prioritise rule BMOD-SPLIT over BMOD-CONST, as long as the number of cases does not become too big. This is because each of the premises of BMOD-SPLIT tends to be significantly simpler to solve (and interpolate) than the conclusion; in addition, splitting one \(bmod^R\) literal often allows subsequent simplifications that eliminate other \(bmod^R\) occurrences.

**Example 6:** We consider one of the examples from [15], the interpolation problem \(A \land B\) defined by

\[
\begin{align*}
A &= \neg bvule_b (bvadd_b (y_4, 1), y_3) \land y_2 = bvadd_d (y_4, 1) \\
B &= bvule_b (bvadd_d (y_2, 1), y_3) \land y_7 = 3 \land y_7 = bvadd_d (y_2, 1)
\end{align*}
\]

where all variables range over unsigned 8-bit bit-vectors. An eager encoding into LIA would typically add variables to handle wrap-around semantics, e.g., mapping \(y'_4 = bvadd_d (y_4, 1)\) to \(y'_4 = y_4 + b1 - 2^8 \sigma_1 \land 0 \leq y'_4 < 2^8 \land 0 \leq \sigma_1 \leq 1\). Additional variables tend to be hard for interpolation, and the LIA interpolant presented in [15] is the formula \(I_{LIA} = -255 \leq y_2 - y_3 + 256[-1_{\frac{y_2}{256}}]\); the formula can be mapped back to a pure bit-vector formula if needed.

We outline how our calculus proves the unsatisfiability of \(A \land B\). Translation of the formulas to the core language gives:

\[
\begin{align*}
A_{core} &= \psi_A \land ubmod_w (y_2 + 1, c_2) \land \\
&\quad c_1 > y_3 \land y_2 = c_1 \\
B_{core} &= \psi_B \land ubmod_w (y_2 + 1, c_2) \land \\
&\quad c_2 \leq y_3 \land y_7 = 3 \land y_7 = c_2
\end{align*}
\]

where \(\psi_A = in_b (y_2) \land in_b (y_3) \land in_b (y_4) \land in_b (c_1)\) and \(\psi_B = in_d (y_2) \land in_d (y_3) \land in_d (y_7) \land in_d (c_2)\) are the domains. The main reasoning step is application of the rule BMOD-SPLIT to \(ubmod_w (y_2 + 1, c_2)\), using the bounds \(lbound (\Pi, y_2 + 1) = 4\) and \(ubound (\Pi, y_2 + 1) = 256\) that follow from \(A_{core}, B_{core}\):

\[
\begin{align*}
\ldots, 0 \leq c_2 < 256, y_2 + 1 = c_2 \vdash \\
\ldots, 0 \leq c_2 < 256, y_2 + 1 = c_2 + 256 \vdash \\
\ldots, ubmod_w (y_2 + 1, c_2) \vdash
\end{align*}
\]

Due to \(y_7 = 3 \land y_7 = c_2\), the cases reduce to \(y_2 = 2\) and \(y_2 = 258\), and immediately contradict \(A_{core}, B_{core}\).

**D. Quantifier Elimination and Craig Interpolation**

Since the bit-vector rules in this section are all equivalence transformations, QE for bit-vectors can be done exactly as described in Section III-B. As the ranges of all symbols are now bounded, it is guaranteed that any formula will eventually be reduced to Presburger arithmetic, so that we obtain complete QE for (arithmetic) bit-vector constraints.

Similarly, the interpolation approach from Section III-C carries over to bit-vectors, with theorem axioms being generated for each of the rules defined in this section. Since the translation of bit-vector formulas to the core language happens upfront, also interpolants are guaranteed to be in the core language, and can be mapped back to bit-vector formulas if necessary (e.g., as in [15]). Interpolants might contain quantifiers, in which case QE can be applied (as described in the first paragraph), so that we altogether obtain a complete procedure for quantifier-free interpolation of arithmetic bit-vector formulas. For interpolation problems from software verification, it happens rarely, however, that QE is needed.

In our implementation, we restrict the use of the simplification rules RW-LEFT and RW-RIGHT when computing proofs for the purpose of interpolation. Unrestricted use could quickly mix up the vocabularies of the individual partitions in an interpolation problem \(A \land B\), and thus increase the likelihood of quantifiers in interpolants. Instead we simplify \(A, B\) separately upfront using rules in Fig. 4, and apply RW-LEFT, RW-RIGHT only when the modified formula \(\phi\) is a literal.

**Example 7:** We continue Example 6, and show how our calculus finds the simpler interpolant \(I_{LIA} = y_3 < y_2\) for the interpolation problem \(A \land B\). The core step is to turn the
application of BMOD-SPLIT into an explicit axiom; after slight 
simplifications, this axiom is:

\[
Ax = (\text{umod}_{w}(y_2 + 1, c_2) \land 3 \leq y_2 < 256 \land \text{in8}(c_2)) \rightarrow 
(y_2 + 1 = c_2 \lor y_2 + 1 = c_2 + 256)
\]

The axiom mentions all assumptions made by the rule, including 
the bounds \( 3 \leq y_2 < 256 \) that determine the 
number of resulting cases (or, alternatively, the formulas 
\( c_1 > y_3, y_2 = c_1, c_2 \leq y_3, y_3 = c_3 \) from which the 
bounds derive). The axiom also includes domain constraints 
like \( \text{in8}(c_2) \) for occurring symbols, which later ensures that 
possible quantifiers in interpolants range over bounded 
domains. The quantified axiom is \( QAx = \forall y_2, c_2, Ax \), and can be 
used to construct an interpolating proof:

\[
\begin{align*}
|c_1 > y_3|_L, |y_2 = c_1|_L, |c_2 \leq y_3|_R, |y_3 = c_3|_R, |y_3 = c_2|_R, |y_2 + 1 = c_2 + 256|_R & \vdash \emptyset \triangleright y_3 < y_2 \\
\text{valid} & \triangleright y_3 < y_2 \quad \triangleright \text{valid}
\end{align*}
\]

We only show one of the cases, \( \mathcal{P} \), resulting from splitting 
the axiom \( |Ax|_R \) using the rules from Fig. 2. The final interpolant 
\( y_3 < y_2 \) records the information needed from \( A_{\text{core}} \) to derive 
a contradiction in the presence of \( y_2 + 1 = c_2 \); the branch is 
closed using standard arithmetic reasoning [10].

V. Experiments

We have implemented the procedures in the PRINCESS 
theorem prover. PRINCESS also partly supports operators like 
shift and bit-wise and/or. All experiments were done using 
PRINCESS version 2018.05.25 on an AMD Opteron 2220 SE 
machine, running 64-bit Linux and Java 1.8. Runtime was 
limited to 10min wall clock time, and heap space 2GB.

c) \textit{SAT Checking on BV and QF$_\text{BV}$ Problems:} Results 
on SMT-LIB benchmarks are given in Table II. We compare 
our implementation with Z3 4.8.0 and CVC4 1.6. Our procedure 
can solve a similar number of problems as Z3 and 
CVC4 on many of the BV families. Although our procedure 
is not specifically designed for QF$_\text{BV}$, we include overall 
numbers for completeness (excluding the families ASP and 
Sage). However, the overwhelming majority of the QF$_\text{BV}$ 
benchmarks contains bit-wise operations not fully supported 
by PRINCESS yet. QF$_\text{BV}$ families on which our procedure 
does well include Example 3 and the PSPACE family.

d) Verification of C Programs: Since it is difficult to 
compare interpolation procedures outside of an application, 
we present results of running the ELDARICA version 
2.0-alpha3 model checker\(^5\) on a benchmark set of 551 C programs, 
using the implementation of our calculus in PRINCESS as 
interpolation procedure (Table I). The benchmarks are the 
programs used in [31] for evaluating different predicate generation 
strategies. The programs use only arithmetic operations, no 
arrays or heap data structures. For this paper, we interpret 
the programs as operating either on the mathematical integers 
(math), or on signed 32-bit bit-vectors (ilp32) with wrap- 
around semantics. Both configurations were running a parallel 
portfolio of two interpolation strategies (ELDARICA option 
-abstractPO): straightforward interpolation to compute 
predicates, and the interpolation abstraction technique [32]. 
The experiments show that our interpolation approach for bit- 
vectors can solve almost as many programs as the existing 
interpolation methods for mathematical integers, with a 
similar number of CEGAR iterations, and with interpolants of 
comparable size. The scatter plot in Fig. 6 indeed shows very 
similar runtimes for the two configurations.

As comparison, we also ran CPACHECKER 1.7 [33] on the 
benchmarks, using options -predicateAnalysis -32 
and MATHSAT as solver; MATHSAT uses the interpolation 
method from [15]. As can be seen in Table I, our method is 
competitive with CPACHECKER on all considered families, 
in particular for the safe programs. We remark, however, 
that we are comparing different verification systems here. 
Although both ELDARICA and CPACHECKER apply CEGAR 
and interpolation, there are many factors affecting the results.

VI. Conclusions

We have presented a new calculus for Craig interpolation 
and quantifier elimination in bit-vector arithmetic. While the 
experimental results in model checking are already promising, 
we believe that there is still a lot of room for extension and

\(^5\)https://github.com/uuverifiers/eldarica
improvement of the approach. This includes more powerful propagation and simplification rules, and more sophisticated strategies to apply the splitting rules $\times$-SPLIT and BMOD-SPLIT. Future work also includes the extension of our calculus to bit-wise operations like $\text{bvand}$, $\text{bvor}$, or $\text{bvxor}$, for which we plan to add further uninterpreted predicates to our setting to preserve laziness as far as possible.

Acknowledgements: We thank the reviewers for helpful comments. This work was supported by the Swedish Research Council (VR) under grant 2014-5484, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

REFERENCES