Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction

Peter Backeman, Philipp Rümmer, Aleksandar Zeljić
Department of Information Technology, Uppsala University, Sweden

***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 [1], non-linear real arithmetic [9], 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 α over the set $\mathbb{Z}$ of integers. The syntax of terms and formulae is defined by the following grammar:

$$\phi ::= t = 0 \,\big|\, t \leq 0 \,\big|\, p(t,\ldots,t) \,\big|\, \phi \wedge \phi \,\big|\, \phi \vee \phi \,\big|\, \neg\phi \,\big|\, \forall x.\phi \,\big|\, \exists x.\phi$$
$$t ::= \alpha \,\big|\, c \,\big|\, x \,\big|\, \alpha t + \cdots + \alpha t$$

The symbol t denotes terms of linear arithmetic. Substitution of a term t for a variable x in ϕ is denoted by $[x/t]\phi$; 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 *true* (*false*) stands for the equality $0 = 0$ ($1 = 0$), and the formula $\phi \rightarrow \psi$ abbreviates $\neg\phi \vee \psi$. 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 Γ, Δ are sets of formulae, then $\Gamma \vdash \Delta$ is a *sequent*. A sequent is *valid* if the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. Positions in Δ that are underneath an even/odd number of negations are called positive/negative; and vice versa for Γ. 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 formulas 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 undecidable fragments, for instance for sequents $\vdash \phi$ with only existential quantifiers.

For quantifier-free input formulas, proof search can be implemented in depth-first style following the core concepts

$$\frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta}\ \vee\text{-LEFT} \qquad \frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \wedge\text{-RIGHT}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \wedge\text{-LEFT} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \vee\text{-RIGHT}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \neg\text{-LEFT} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\text{-RIGHT}$$

$$\frac{*}{\Gamma, \phi \vdash \phi, \Delta}\ \text{CLOSE}$$

$$\frac{\Gamma, [x/t]\phi, \forall x.\phi \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ \forall\text{-LEFT} \qquad \frac{\Gamma, [x/c]\phi \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta}\ \exists\text{-LEFT}$$

$$\frac{\Gamma \vdash [x/t]\phi, \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \exists\text{-RIGHT} \qquad \frac{\Gamma \vdash [x/c]\phi, \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \forall\text{-RIGHT}$$

Fig. 1. A selection of the basic calculus rules for propositional logic (upper box) and quantifier rules (lower box). In the rules $\exists$-LEFT and $\forall$-RIGHT, c is a constant that does not occur in the conclusion.

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 ϕ is a formula without uninterpreted predicates and without constants c, but possibly containing variables x. Formula ϕ 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 ψ that is equivalent to ϕ, 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 can introduce fresh constants, which are subsequently isolated and eliminated by the arithmetic rules. To find ψ, it is essentially enough to extract the constant-free formulas $\Gamma_i^v \subseteq \Gamma_i$, $\Delta_i^v \subseteq \Delta_i$ 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.

$$\frac{\Gamma, \lfloor\phi\rfloor_L \vdash \Delta \blacktriangleright I \quad \Gamma, \lfloor\psi\rfloor_L \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_L \vdash \Delta \blacktriangleright I \vee J} \vee\text{-LEFT}_L$$

$$\frac{\Gamma, \lfloor\phi\rfloor_R \vdash \Delta \blacktriangleright I \quad \Gamma, \lfloor\psi\rfloor_R \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_R \vdash \Delta \blacktriangleright I \wedge J} \vee\text{-LEFT}_R$$

$$\frac{\Gamma, \lfloor\phi\rfloor_D, \lfloor\psi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\phi \wedge \psi\rfloor_D \vdash \Delta \blacktriangleright I} \wedge\text{-LEFT}_D$$

$$\frac{\Gamma \vdash \lfloor\phi\rfloor_D, \Delta \blacktriangleright I}{\Gamma, \lfloor\neg\phi\rfloor_D \vdash \Delta \blacktriangleright I} \neg\text{-LEFT}_D$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright \mathit{false}} \text{CLOSE}_{LL}$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright \phi} \text{CLOSE}_{LR}$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_R \vdash \lfloor\phi\rfloor_L, \Delta \blacktriangleright \neg\phi} \text{CLOSE}_{RL}$$

$$\frac{\Gamma, \lfloor[x/t]\phi\rfloor_L, \lfloor\forall x.\phi\rfloor_L \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_L \vdash \Delta \blacktriangleright \forall_{Rt}\, I} \forall\text{-LEFT}_L$$

$$\frac{\Gamma, \lfloor[x/t]\phi\rfloor_R, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright \exists_{Lt}\, I} \forall\text{-LEFT}_R$$

$$\frac{\Gamma, \lfloor[x/c]\phi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\exists x.\phi\rfloor_D \vdash \Delta \blacktriangleright I} \exists\text{-LEFT}_D$$

Fig. 2. The upper box presents a selection of interpolating rules for propositional logic, while the lower box shows rules for quantifiers. Parameter D stands for either L or R. The quantifier $\forall_{Rt}$ denotes universal quantification over all constants occurring in t but not in $\Gamma_L \cup \Delta_L$; likewise, $\exists_{Lt}$ denotes existential quantification over all constants occurring in t but not in $\Gamma_R \cup \Delta_R$. In $\exists$-LEFT$_D$, c is a constant that does not occur in the conclusion.

C. Craig Interpolation in the Base Logic

Given formulas A and B such that $A \wedge B$ 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 $\lfloor\phi\rfloor_L$ (with L for "left") to indicate that ϕ is derived from A, and similarly formulas $\lfloor\phi\rfloor_R$ for ϕ derived from B. If Γ, Δ are finite sets of L/R-labelled formulas, and I is an unlabelled formula, then $\Gamma \vdash \Delta \blacktriangleright I$ is an *interpolating sequent*.

Semantics of interpolating sequents is defined using projections $\Gamma_L =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_L \in \Gamma\}$ and $\Gamma_R =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_R \in \Gamma\}$, which extract the L/R-parts of a set Γ of labelled formulae. A sequent $\Gamma \vdash \Delta \blacktriangleright I$ 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 $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$ is valid iff I is an interpolant of $A \wedge B$. Soundness of the calculus guarantees that the root of a closed interpolating proof is a valid interpolating sequent.

[1]http://www.philipp.ruemmer.org/princess.shtml

To solve an interpolation problem $A \wedge B$, 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 $\lfloor A\rfloor_L, \lfloor B\rfloor_R$, 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 $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$.

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., $\times(s,t,r) \Leftrightarrow s \cdot t = r$. 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 $\neg\times(s,t,r) \Leftrightarrow \exists x.(\times(s,t,x) \wedge x \neq r)$.

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 ϕ 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 $\{\times(s_i,t_i,r_i)\}_{i=1}^n$ 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

[2]The set of *all* linear equations implied by a set of $\times$-literals over integers is clearly not computable, by reduction of Hilbert's 10th problem.

equation $f = 0$ that logically follows from the literals, and can therefore be added to a proof goal:

$$\frac{\Gamma, \{\times(s_i, t_i, r_i)\}_{i=1}^n, \{e_j = 0\}_{j=1}^m, f = 0 \vdash \Delta}{\Gamma, \{\times(s_i, t_i, r_i)\}_{i=1}^n, \{e_j = 0\}_{j=1}^m \vdash \Delta} \times\text{-EQ}$$
$$\text{if } f \text{ is linear, has integer coefficients, and } f \in I$$

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 $\times$-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 $\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m$ 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 $\Pi = \{\times(x, x, c_1), \times(x, y, c_2), \times(y, y, c_3), \times(x+y, x+y, c_4)\}$:

$$\frac{\begin{array}{c}\vdots^{*}\\ \Pi, c_1 + 2c_2 + c_3 - c_4 = 0 \vdash c_4 = c_1 + 2c_2 + c_3\end{array}}{\Pi \vdash c_4 = c_1 + 2c_2 + c_3} \times\text{-EQ}$$

Here, the $\times$-EQ-step is motivated by the fact that the Gröbner basis derived from Π 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 $Prop_{\{\phi_1,\ldots,\phi_n\}}$ 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 $c \in C \cup X$:

$$\frac{\Gamma, \phi_1, \ldots, \phi_n,\ l \le c, c \le u \vdash \Delta}{\Gamma, \phi_1, \ldots, \phi_n \vdash \Delta} \times\text{-ICP}$$
$$\text{if } (\text{gfp}\, Prop_{\{\phi_1,\ldots,\phi_n\}})(c) = [l, u]$$

Example 2: From two inequalities $x \ge 5$ and $y \ge 5$, the rule $\times$-ICP can derive $(x + y)^2 \ge 100$:

$$\frac{\times(x + y, x + y, c_4), x \ge 5, y \ge 5, 100 \le c_4 \vdash}{\times(x + y, x + y, c_4), x \ge 5, y \ge 5 \vdash} \times\text{-EQ}$$

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

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 $s \cdot t \ge 0$ if the term $s \cdot t$ can be expressed linearly. For this, we again write $I = (\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$ for the ideal induced by equations and $\times$-literals:

$$\frac{\Gamma, s \le 0, t \le 0,\ -f \le 0 \vdash \Delta}{\Gamma, s \le 0, t \le 0 \vdash \Delta} \times\text{-CROSS}$$
$$\text{if } f \text{ is linear, has integer coefficients, and } s \cdot t - f \in I$$

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 $c \in C \cup X$:

$$\frac{\Gamma, c \le \alpha - 1 \vdash \Delta \qquad \Gamma, c \ge \alpha \vdash \Delta}{\Gamma \vdash \Delta} \times\text{-SPLIT}$$

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 $\times$-ICP: when no ranges can be tightened any further using $\times$-ICP, instead $\times$-SPLIT can be applied to split one of the intervals in half.

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

$$\frac{\Gamma \vdash \Delta}{\Gamma, \times(s, t, r) \vdash \Delta} \times\text{-ELIM}$$
$$\text{if } s \cdot t - r \in I$$

Note that $\times$-ELIM only eliminates non-linear $\times$-literals, whereas $\times$-EQ only introduces linear equations, 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 ϕ 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

$$\frac{\begin{array}{c}\vdots\\ \Gamma,\Gamma',\Gamma_1 \vdash \Delta_1,\Delta',\Delta\end{array} \quad \cdots \quad \begin{array}{c}\vdots\\ \Gamma,\Gamma',\Gamma_n \vdash \Delta_n,\Delta',\Delta\end{array}}{\begin{array}{c}\Gamma,\Gamma' \vdash \Delta',\Delta\\ \vdots\end{array}}\ R$$

in which Γ',Δ' are the formulas assumed by the rule application, Γ,Δ 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 (unquantified) theory axiom Ax corresponding to the rule application expresses that the conjunction of the premises has to imply the conclusion; the quantified theory axiom $QAx =_{\text{def}} \forall S.\, Ax$ in addition contains universal quantifiers for all constants $S \subseteq C$ occurring in Ax.

$$Ax \;=_{\text{def}}\; \bigwedge_{i=1}^{n} \left(\bigwedge \Gamma_i \rightarrow \bigvee \Delta_i\right) \;\rightarrow\; \left(\bigwedge \Gamma' \rightarrow \bigvee \Delta'\right)$$

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:

$$\frac{\dfrac{\begin{array}{c}\vdots^{*}\\ \Gamma,\Gamma',\bigwedge\Gamma' \rightarrow \bigvee\Delta' \vdash \Delta',\Delta\end{array} \qquad \begin{array}{c}\Gamma,\Gamma',\Gamma_1 \vdash \Delta_1,\Delta',\Delta \quad \cdots\\ \vdots\end{array}}{\Gamma,\Gamma',Ax \vdash \Delta',\Delta}}{\Gamma,\Gamma',\forall S.Ax \vdash \Delta',\Delta}\ \forall\text{-LEFT}*$$

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 $\lfloor QAx \rfloor_L$ or $\lfloor QAx \rfloor_R$ on demand.

The obvious downside of this approach is the possibility of quantifiers occurring in interpolants. The interpolating rules $\forall$-LEFT$_{L/R}$ (Fig. 2) have to introduce quantifiers $\forall_{Rt}/\exists_{Lt}$ 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 $bmod_a^b$. The predicate is only gradually reduced to Presburger arithmetic by applying the calculus rules introduced later in this section.

Formally, we introduce a set $P_{bv} = \{bmod_a^b \mid a,b \in \mathbb{Z},\ a < b\}$ of binary predicates. The semantics of $bmod_a^b$ is to relate any whole number $x \in \mathbb{Z}$ to its remainder modulo $b-a$ in the interval $\{a,\ldots,b-1\}$:

$$\begin{aligned} bmod_a^b(s,r) \;&\Leftrightarrow\; a \le r < b \;\wedge\; \exists z.\; r = s + (b-a)\cdot z\\ &\Leftrightarrow\; a \le r < b \;\wedge\; r \equiv s \pmod{b-a}\end{aligned}$$

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

$$ubmod_w \;=_{\text{def}}\; bmod_0^{2^w}, \quad sbmod_w \;=_{\text{def}}\; bmod_{-2^{w-1}}^{2^{w-1}}.$$

A. Translating Bit-Vector Constraints to the Core Language

For the rest of the section, we use the base logic augmented with $\times$ and $bmod_a^b$-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, $\mathsf{BVOP} = \{\mathsf{bvadd}_w,\ \mathsf{bvmul}_w,\ \mathsf{bvudiv}_w,\ \mathsf{bvneg}_w,\ \mathsf{ze}_{w+w'},\ \mathsf{bvule}_w,\ \mathsf{bvsle}_w\}$. All operations are sub-scripted with the bit-width of the operands; the zero-extend function $\mathsf{ze}_{w+w'}$ maps bit-vectors of width w to width $w+w'$. Semantics

[3] Non-linear integer arithmetic in general does not admit quantifier-free interpolants. For instance, $(x > 1 \wedge x = y^2) \wedge x = z^2 + 1$ is unsatisfiable, but no quantifier-free interpolants exist, regardless of whether divisibility predicates $\alpha \mid t$ are allowed or not.

$$\mathsf{bvadd}_w(s,t) = r \;\twoheadrightarrow\; ubmod_w(s+t,r) \qquad \mathsf{bvneg}_w(s) = r \;\twoheadrightarrow\; ubmod_w(-s,r)$$
$$\mathsf{bvmul}_w(s,t) = r \;\twoheadrightarrow\; \exists x.\ \big(\times(s,t,x) \wedge ubmod_w(x,r)\big) \qquad \mathsf{ze}_{w+w'}(s) = r \;\twoheadrightarrow\; s = r$$
$$\mathsf{bvsle}_w(s,t) \;\twoheadrightarrow\; \exists x,y.\ (sbmod_w(s,x) \wedge sbmod_w(t,y) \wedge x \le y) \qquad \mathsf{bvule}_w(s,t) \;\twoheadrightarrow\; s \le t$$
$$\neg\mathsf{bvsle}_w(s,t) \;\twoheadrightarrow\; \exists x,y.\ (sbmod_w(s,x) \wedge sbmod_w(t,y) \wedge x > y) \qquad \neg\mathsf{bvule}_w(s,t) \;\twoheadrightarrow\; s > t$$
$$\mathsf{bvudiv}_w(s,t) = r \;\twoheadrightarrow\; \big(t = 0 \wedge r = 2^w - 1\big) \vee \big(t \ge 1 \wedge \exists x.\ (\times(t,r,x) \wedge s - t < x \le s)\big)$$

Fig. 3. Rules translating bit-vector operations into the core language. The rules only apply in negative positions.

follows the FixedSizeBitVectors[4] theory of the SMT-LIB [30]. Other arithmetic operations, for instance bvsdiv_w or bvsmod_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 ϕ to core formulas ϕ_{core} has two parts: first, BVOP occurrences in a formula ϕ 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 ϕ 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 $f(\bar{s}) = t$ in negative positions. The translation from ϕ to ϕ' is then defined by the rewriting rules in Fig. 3. Since the rules for the predicate bvsle_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 bvadd_w, bvneg_w, $\mathsf{ze}_{w+w'}$, and bvule_w simply translate to the corresponding Presburger term, if necessary followed by remainder $ubmod_w$. Multiplication bvmul_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 bvsle_w simply maps the operands to a signed bit-vector domain $\{-2^{w-1}, \ldots, 2^{w-1} - 1\}$. The rule for unsigned division bvudiv_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 ϕ are interpreted as unbounded integer variables in ϕ_{core}, which therefore has to contain explicit assumptions about the ranges of bit-vector variables. We use the abbreviation $in_w(x) =_{\text{def}} (0 \le x < 2^w)$ and define

$$\phi_{core} \;=\; \Big(\bigwedge_{x \in S} in_{w_x}(x)\Big) \to \phi'$$

where $S \subseteq C \cup X$ is the set of free variables and constants occurring in ϕ, w_x is the bit-width of $x \in S$, and ϕ' is the result of applying rules from Fig. 3 to ϕ. Similar constraints are used to express quantification over bit-vectors, for instance $\exists x.\ (in_w(x) \wedge \ldots)$ and $\forall x.\ (in_w(x) \to \ldots)$.

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:

$$\mathsf{bvule}_{32}(b, \mathsf{bvudiv}_{32}(2^{32}-1, a)) \to \mathsf{bvule}_{64}(\mathsf{bvmul}_{64}(\mathsf{ze}_{32+32}(a), \mathsf{ze}_{32+32}(b)), 2^{32}-1)$$

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:

$$\big(\mathsf{bvudiv}_{32}(2^{32}-1, a) = c_1 \wedge \mathsf{bvmul}_{64}(c_3, c_4) = c_2 \wedge \mathsf{ze}_{32+32}(a) = c_3 \wedge \mathsf{ze}_{32+32}(b) = c_4 \wedge \mathsf{bvule}_{32}(b, c_1)\big) \to \mathsf{bvule}_{64}(c_2, 2^{32}-1)$$

The final formula ϕ_{core} is obtained by application of the rules in Fig. 3, and adding domain constraints:

$$\begin{aligned}&\big(in_{32}(a) \wedge in_{32}(b) \wedge in_{32}(c_1) \wedge in_{64}(c_2) \wedge in_{64}(c_3) \wedge in_{64}(c_4)\big) \wedge \\ &\left(\begin{array}{l}(a = 0 \wedge c_1 = 2^{32}-1) \vee \\ (a \ge 1 \wedge \exists x.(\times(a, c_1, x) \wedge 2^{32} - 1 - a < x \le 2^{32}-1))\end{array}\right) \wedge \\ &\exists z.\ (\times(c_3, c_4, z) \wedge ubmod_{64}(z, c_2)) \wedge a = c_3 \wedge b = c_4 \wedge b \le c_1\big) \\ &\to c_2 \le 2^{32} - 1\end{aligned}$$

B. Preprocessing and Simplification

An encoded formula ϕ_{core} tends to contain a lot of redundancy, in particular nested or unnecessary occurrences of the $bmod_a^b$ 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 $\Pi : \phi \twoheadrightarrow \phi'$ expresses that formula ϕ can be rewritten to ϕ', given the set Π of formulas as context. The structural rules in the upper half of Fig. 4 define how formulas are traversed, and how the context Π is extended to Π, Lit' when encountering further literals. We apply the structural rules modulo associativity and commutativity of $\wedge, \vee$, and prioritise LIT-$\wedge$-RW and LIT-$\vee$-RW over the other

[4]http://www.smtlib.org/theories-FixedSizeBitVectors.shtml

$$\frac{\Pi : \phi \twoheadrightarrow \phi' \qquad \Pi : \psi \twoheadrightarrow \psi'}{\Pi : \phi \circ \psi \twoheadrightarrow \phi' \circ \psi'} \circ\text{-RW}$$

$$\frac{\Pi : Lit \twoheadrightarrow Lit' \qquad \Pi, Lit' : \phi \twoheadrightarrow \phi'}{\Pi : Lit \wedge \phi \twoheadrightarrow Lit' \wedge \phi'} \text{LIT-}\wedge\text{-RW}$$

$$\frac{\Pi : Lit \twoheadrightarrow Lit' \qquad \Pi, \neg Lit' : \phi \twoheadrightarrow \phi'}{\Pi : Lit \vee \phi \twoheadrightarrow Lit' \vee \phi'} \text{LIT-}\vee\text{-RW}$$

$$\frac{\Pi : \phi \twoheadrightarrow \phi'}{\Pi : \neg\phi \twoheadrightarrow \neg\phi'} \neg\text{-RW} \qquad \frac{\Pi : \phi \twoheadrightarrow \phi'}{\Pi : Qx.\phi \twoheadrightarrow Qx.\phi'} Q\text{-RW}$$

$$\frac{\left\lfloor \frac{lbound(\Pi,s)-a}{b-a} \right\rfloor = k = \left\lfloor \frac{ubound(\Pi,s)-a}{b-a} \right\rfloor}{\Pi : bmod_a^b(s,r) \twoheadrightarrow s = r + k \cdot (b-a)} \text{BOUND-RW}$$

$$\frac{s + (b-a) \cdot t \prec s}{\Pi : bmod_a^b(s,r) \twoheadrightarrow bmod_a^b(s + (b-a) \cdot t, r)} \text{COEFF-RW}$$

$$\frac{bmod_{a'}^{b'}(s',r') \in \Pi, \quad (b-a) \mid k \cdot (b'-a'), \quad s + k \cdot (s'-r') \prec s}{\Pi : bmod_a^b(s,r) \twoheadrightarrow bmod_a^b(s + k \cdot (s'-r'), r)} \text{BMOD-RW}$$

Fig. 4. Simplification rules for bit-vector formulas. In $\circ$-RW, ϕ and ψ are not literals, and $\circ \in \{\wedge, \vee\}$. In LIT-$\wedge$-RW and LIT-$\vee$-RW, the formula Lit is a literal. In Q-RW, x must not occur in Π, and $Q \in \{\forall, \exists\}$. 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:

$$\frac{\Gamma, \phi' \vdash \Delta}{\Gamma, \phi \vdash \Delta} \text{RW-LEFT} \qquad \frac{\Gamma \vdash \phi', \Delta}{\Gamma \vdash \phi, \Delta} \text{RW-RIGHT}$$
$$\text{if } \Gamma \cup \{\neg\psi \mid \psi \in \Delta\} : \phi \twoheadrightarrow \phi'$$

The lower half of Fig. 4 shows three of the bit-vector-specific rules. Rule BOUND-RW defines elimination of $bmod_a^b$-predicates that do not require any case splits; the definition of the rule assumes functions $lbound(\Pi, s)$ and $ubound(\Pi, s)$ that derive lower and upper bounds of a term s, respectively, given the current context Π. The two functions can be implemented by collecting inequalities (and possibly type information available for predicates) in Π to obtain an over-approximation of the range of s.

Rule COEFF-RW reduces coefficients in $bmod_a^b(s,r)$ by adding a multiple of the modulus $b-a$ to s. The rule assumes a well-founded order $\prec$ on terms to prevent cycles during simplification. One way to define such an order is to choose a total well-founded order $\prec$ on the union $C \cup X$ of variables and constants, extend $\prec$ to expressions $\alpha \cdot x$ by sorting coefficients as $0 \prec 1 \prec -1 \prec 2 \prec \cdots$, and finally extend $\prec$ to arbitrary terms $\alpha_1 t_1 + \cdots + \alpha_n t_n$ as a multiset order [19].

The same order $\prec$ is used in BMOD-RW, defining how $bmod_a^b(s,r)$ can be rewritten in the context of a second literal $bmod_{a'}^{b'}(s',r')$. The rule is useful to optimise the translation of nested bit-vector operations. Assuming $bmod_{a'}^{b'}(s',r')$,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\ldots, a \geq 1, e < 2^{32}, b \leq c_1, d \geq 2^{32}, e - d - c_1 + b \geq 0 \vdash}\,(*)}{\ldots, \times(a,b,d), \times(a,c_1,e), a \geq 1, e < 2^{32}, b \leq c_1, d \geq 2^{32} \vdash}\,\times\text{-CROSS}}{\ldots, 0 \leq d, d \leq 2^{64} - 2^{33} + 1, d = c_2 \vdash}\,(b)}{\ldots, 0 \leq d, d \leq 2^{64} - 2^{33} + 1, ubmod_{64}(d, c_2) \vdash}\,\text{RW-LEFT}}{\ldots, in_{32}(a), in_{32}(b), \times(a,b,d), ubmod_{64}(d,c_2) \vdash}\,\times\text{-ICP}}{\vdash \phi_{core}}\,(a)$$

Fig. 5. Proof tree for Example 5, with the sequences (a) and (b) of rule applications not shown in detail.

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 $bmod_a^b(s,r)$ is not affected by adding $k \cdot (s' - r')$ to s.

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

Example 4: The expression $\mathsf{bvadd}_{32}(\mathsf{bvadd}_{32}(a,b),c)$ corresponds to $ubmod_{32}(a+b, r_1) \wedge ubmod_{32}(r_1 + c, r_2)$ in the core language. Using BMOD-RW, the formula can be rewritten to $ubmod_{32}(a+b, r_1) \wedge ubmod_{32}(a+b+c, r_2)$, provided that $a + b + c \prec r_1 + c$.

Example 5: We continue Ex. 3 and show that ϕ_{core} is valid, focusing on the $a \geq 1$ case of bvudiv_{32}. The proof (Fig. 5) consists of three core steps: 1) using $\times$-ICP, from the constraints $in_{32}(a)$, $in_{32}(b)$, $\times(a,b,d)$ the inequalities $0 \leq d$ and $d \leq 2^{64} - 2^{33} + 1$ can be derived; 2) therefore, using RW-LEFT and BOUND-RW, the literal $ubmod_{64}(d, c_2)$ can be rewritten to $d = c_2$, capturing the fact that 64-bit multiplication cannot overflow for unsigned 32-bit operands; 3) using $\times$-CROSS, from the inequalities $a \geq 1$ and $b \leq c_1$ and the products $\times(a,b,d)$, $\times(a,c_1,e)$ 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:

$$\frac{\left\{\Gamma, a \leq r < b, s = r + i \cdot (b-a) \vdash \Delta\right\}_{i=l}^{u}}{\Gamma, bmod_a^b(s,r) \vdash \Delta} \text{BMOD-SPLIT}$$

assuming $\left\lfloor \frac{lbound(\Pi,s)-a}{b-a} \right\rfloor = l$ and $\left\lfloor \frac{ubound(\Pi,s)-a}{b-a} \right\rfloor = u$ 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
COMPARISON OF ELDARICA CONFIGURATIONS AND CPACHECKER. FOR EACH FAMILY, THE TABLE SHOWS THE NUMBER OF SAFE/UNSAFE RESULTS, THE AVERAGE TIME, THE REQUIRED NUMBER OF CEGAR ITERATIONS, AND THE AVERAGE SIZE OF COMPUTED INTERPOLANTS FOR ELDARICA.

Categories	Total	ELDARICA math				ELDARICA ilp32				CPACHECKER -32		
		Solved	Time	Iter.	P. Size	Solved	Time	Iter.	P. Size	Solved	Time	Iter.
All	551	293	21.0	11.1	1.0	217	28.0	13.6	1.4	180	30.6	28.5
		101	73.4	31.8	1.0	117	49.7	21.7	1.2	168	48.6	3.9
HOLA	46	44	11.4	8.9	1.1	21	11.0	5.8	2.0	12	84.1	87.4
		0				4	6.0	0.0	1.3	4	11.4	0.0
llreve	21	16	13.1	16.1	1.1	8	17.4	27.3	1.6	7	26.5	75.7
		5	7.4	7.6	1.1	4	8.5	5.8	1.1	5	37.3	7.0
VeriMAP	155	132	5.8	2.3	1.0	100	5.9	3.6	1.1	87	12.2	18.5
		21	8.4	4.4	1.0	41	11.6	2.4	1.5	33	24.8	1.3
SVCOMP	329	101	46.1	22.9	1.0	88	58.1	25.7	1.3	74	44.0	26.3
		75	96.0	41.1	1.0	68	77.7	35.5	1.1	126	56.5	4.5

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

$$\frac{\Gamma, a \le r < b, s = r + (b-a)\cdot c \vdash \Delta}{\Gamma, bmod_a^b(s,r) \vdash \Delta} \text{ BMOD-CONST}$$

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 $bmod_a^b$ literal often allows subsequent simplifications that eliminate other $bmod_a^b$ occurrences.

Example 6: We consider one of the examples from [15], the interpolation problem $A \wedge B$ defined by

$$A = \neg \mathsf{bvule}_8(\mathsf{bvadd}_8(y_4, 1), y_3) \wedge y_2 = \mathsf{bvadd}_8(y_4, 1)$$
$$B = \mathsf{bvule}_8(\mathsf{bvadd}_8(y_2, 1), y_3) \wedge y_7 = 3 \wedge y_7 = \mathsf{bvadd}_8(y_2, 1)$$

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 $y_4' = \mathsf{bvadd}_8(y_4, 1)$ to $y_4' = y_4 + b1 - 2^8\sigma_1 \wedge 0 \le y_4' < 2^8 \wedge 0 \le \sigma_1 \le 1$. Additional variables tend to be hard for interpolation, and the LIA interpolant presented in [15] is the formula $I_{LIA} = -255 \le y_2 - y_3 + 256\lfloor -1\frac{y_2}{256}\rfloor$; the formula can be mapped back to a pure bit-vector formula if needed.

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

$$A_{\text{core}} = \psi_A \wedge ubmod_w(y_4 + 1, c_1) \wedge c_1 > y_3 \wedge y_2 = c_1$$

$$B_{\text{core}} = \psi_B \wedge ubmod_w(y_2 + 1, c_2) \wedge c_2 \le y_3 \wedge y_7 = 3 \wedge y_7 = c_2$$

where $\psi_A = in_8(y_2) \wedge in_8(y_3) \wedge in_8(y_4) \wedge in_8(c_1)$ and $\psi_B = in_8(y_2) \wedge in_8(y_3) \wedge in_8(y_7) \wedge in_8(c_2)$ 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 $A_{\text{core}}, B_{\text{core}}$:

$$\frac{\ldots, 0 \le c_2 < 256, y_2 + 1 = c_2 \vdash \quad \ldots, 0 \le c_2 < 256, y_2 + 1 = c_2 + 256 \vdash}{\ldots, ubmod_w(y_2 + 1, c_2) \vdash} \text{ BMOD-SPLIT}$$

Due to $y_7 = 3 \wedge y_7 = c_2$, the cases reduce to $y_2 = 2$ and $y_2 = 258$, and immediately contradict $A_{\text{core}}, B_{\text{core}}$.

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 $A \wedge B$, 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 ϕ 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 $A \wedge B$. The core step is to turn the

TABLE II
PERFORMANCE ON SMT-LIB BV AND QF_BV PROBLEMS. FOR EACH FAMILY, THE FIRST/SECOND ROW GIVES SAT/UNSAT PROBLEMS.

Category	PRINCESS		Z3		CVC4	
	Total	Time	Total	Time	Total	Time
Automizer	16	158.2	16	0.1	14	0.1
	127	215.1	137	0.0	137	0.3
keymaera	5	268.6	108	6.9	34	1.0
	3771	2.5	3923	0.3	3921	0.1
psyco	2	2.5	132	0.1	132	1.5
	3	141.6	62	0.2	62	0.5
tptp	15	2.3	17	0.0	17	0.0
	54	1.7	56	0.0	56	0.0
RND	2	40.8	40	6.9	25	40.7
	5	188.5	28	6.7	22	13.2
RNDPRE	2	7.4	20	19.0	22	26.9
	14	53.9	36	14.1	26	29.3
model	16	1.9	144	0.0	73	10.8
	0		0		0	
Heizmann	13	49.8	15	37.8	18	18.1
	27	155.5	17	50.7	108	8.3
ranking	0		34	4.4	32	1.5
	5	12.0	19	19.5	13	0.4
fixpoint	25	94.9	36	0.5	54	14.2
	26	85.0	73	0.6	75	2.3
QFBV	334	2.3	2701	11.6	2632	17.4
	164	16.4	1967	29.7	1919	19.3

application of BMOD-SPLIT into an explicit axiom; after slight simplifications, this axiom is:

$$Ax = \begin{array}{l}\big(ubmod_w(y_2+1,c_2) \wedge 3 \leq y_2 < 256 \wedge in_8(c_2)\big) \rightarrow \\ \big(y_2+1 = c_2 \vee y_2+1 = c_2+256\big)\end{array}$$

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 $c_1 > y_3, y_2 = c_1, c_2 \leq y_3, y_7 = 3, y_7 = c_2$ from which the bounds derive). The axiom also includes domain constraints like $in_8(c_2)$ 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:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots^{*} \\ \lfloor c_1 > y_3 \rfloor_L, \lfloor y_2 = c_1 \rfloor_L, \lfloor c_2 \leq y_3 \rfloor_R, \\ \lfloor y_7 = 3 \rfloor_R, \lfloor y_7 = c_2 \rfloor_R, \lfloor y_2 + 1 = c_2 \rfloor_R \end{array} \vdash \emptyset \blacktriangleright y_3 < y_2}{\mathcal{P}} \quad \dfrac{\cdots \quad \mathcal{P} \quad \cdots}{\lfloor A_{\text{core}} \rfloor_L, \lfloor B_{\text{core}} \rfloor_R, \lfloor Ax \rfloor_R \vdash \emptyset \blacktriangleright y_3 < y_2} \vee\text{-LEFT}_R}{\lfloor A_{\text{core}} \rfloor_L, \lfloor B_{\text{core}} \rfloor_R, \lfloor QAx \rfloor_R \vdash \emptyset \blacktriangleright y_3 < y_2} \forall\text{-LEFT}_R$$

We only show one of the cases, $\mathcal{P}$, resulting from splitting the axiom $\lfloor Ax \rfloor_R$ using the rules from Fig. 2. The final interpolant $y_3 < y_2$ records the information needed from A_{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) SAT Checking on BV and QF_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_BV, we include overall numbers for completeness (excluding the families ASP and Sage). However, the overwhelming majority of the QF_BV benchmarks contains bit-wise operations not fully supported by PRINCESS yet. QF_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

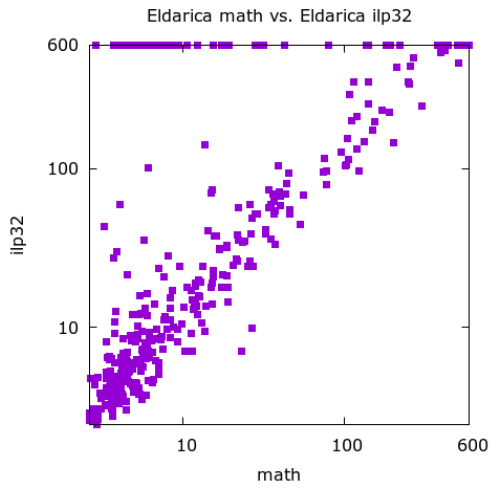

Fig. 6. Scatter plot comparing runtime of math and ilp32 semantics on the C benchmarks.

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 bvand, bvor, or 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

[1] K. L. McMillan, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, 2005.

[2] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant strength," in *VMCAI*, ser. LNCS. Springer, 2010.

[3] A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli, "Ground interpolation for the theory of equality," in *TACAS*, ser. LNCS. Springer, 2009.

[4] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "Beyond quantifier-free interpolation in extensions of Presburger arithmetic," in *VMCAI*, ser. LNCS. Springer, 2011, pp. 88–102.

[5] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 413–427.

[6] L. Kovács and A. Voronkov, "Interpolation and symbol elimination," in *CADE*, 2009, pp. 199–213.

[7] M. P. Bonacina and M. Johansson, "On interpolation in automated theorem proving," *J. Autom. Reasoning*, vol. 54, no. 1, pp. 69–97, 2015.

[8] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *SIGSOFT'06/FSE-14*. New York, NY, USA: ACM, 2006, pp. 105–116.

[9] L. Dai, B. Xia, and N. Zhan, "Generating non-linear interpolants by semidefinite programming," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 364–380.

[10] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "An interpolating sequent calculus for quantifier-free Presburger arithmetic," *Journal of Automated Reasoning*, vol. 47, pp. 341–367, 2011.

[11] A. Griggio, T. T. H. Le, and R. Sebastiani, "Efficient interpolant generation in satisfiability modulo linear integer arithmetic," *Logical Methods in Computer Science*, vol. 8, no. 3, 2010.

[12] R. Bruttomesso, S. Ghilardi, and S. Ranise, "Quantifier-free interpolation of a theory of arrays," *Logical Methods in Computer Science*, vol. 8, no. 2, 2012.

[13] N. Totla and T. Wies, "Complete instantiation-based interpolation," *J. Autom. Reasoning*, vol. 57, no. 1, pp. 37–65, 2016.

[14] J. Hoenicke and T. Schindler, "Efficient interpolation for the theory of arrays," *CoRR*, vol. abs/1804.07173, 2018.

[15] A. Griggio, "Effective word-level interpolation for software verification," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 28–36.

[16] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *TACAS*, ser. LNCS, vol. 7795, 2013.

[17] D. Kroening and G. Weissenbacher, "Lifting propositional interpolants to the word-level," in *FMCAD*. IEEE Computer Society, 2007, pp. 85–89.

[18] ——, "An interpolating decision procedure for transitive relations with uninterpreted functions," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 6405. Springer, 2009, pp. 150–168.

[19] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *LPAR*, ser. LNCS, vol. 5330. Springer, 2008, pp. 274–289.

[20] M. C. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd ed. Springer-Verlag, New York, 1996.

[21] J. Y. Halpern, "Presburger arithmetic with unary predicates is Π^1_1 complete," *Journal of Symbolic Logic*, vol. 56, 1991.

[22] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[23] A. Reynolds, T. King, and V. Kuncak, "Solving quantified linear arithmetic by counterexample-guided instantiation," *Formal Methods in System Design*, vol. 51, no. 3, pp. 500–532, 2017.

[24] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 250–268, September 1957.

[25] S. Lang, *Algebra (3. ed.)*. Addison-Wesley, 1993.

[26] B. Buchberger, "An algorithm for finding the basis elements in the residue class ring modulo a zero dimensional polynomial ideal," Ph.D. dissertation, 3 2006.

[27] P. Van Hentenryck, D. McAllester, and D. Kapur, "Solving polynomial systems using a branch and prune approach," *SIAM J. Numer. Anal.*, vol. 34, no. 2, pp. 797–827, Apr. 1997.

[28] J. Warren A. Hunt, R. B. Krug, and J. S. Moore, "Linear and nonlinear arithmetic in ACL2." in *Proceedings, Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference*, ser. LNCS, vol. 2860. Springer, 2003, pp. 319–333.

[29] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "SAT modulo linear arithmetic for solving polynomial constraints," *J. Autom. Reasoning*, vol. 48, no. 1, pp. 107–131, 2012.

[30] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[31] Y. Demyanova, P. Rümmer, and F. Zuleger, "Systematic predicate abstraction using variable roles," in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, ser. Lecture Notes in Computer Science, C. Barrett, M. Davies, and T. Kahsai, Eds., vol. 10227, 2017, pp. 265–281.

[32] J. Leroux, P. Rümmer, and P. Subotic, "Guiding craig interpolation with domain-specific abstractions," *Acta Inf.*, vol. 53, no. 4, pp. 387–424, 2016.

[33] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," *CoRR*, vol. abs/0902.0019, 2009.