Noname manuscript No.
(will be inserted by the editor)

Interpolating Bit-Vector Formulas using Uninterpreted Predicates and Presburger Arithmetic

Peter Backeman · Philipp Rümmer · Aleksandar Zeljić

the date of receipt and acceptance should be inserted later

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. We also incorporate a method for handling concatenations and extractions of bit-vector efficiently.

Keywords bit-vectors, interpolation, quantifier elimination, Presburger arithmetic

Acknowledgements This research has been supported by the Swedish Research Council (VR) under the grants 2014-5484 and 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and by a grant from Microsoft. We also wish to thank the reviewers for their extensive and helpful feedback.

Peter Backeman
Mälardalen University, Västerås, Sweden

Philipp Rümmer
Department of Information Technology, Uppsala University, Sweden

Aleksandar Zeljić
Stanford University, Stanford, USA

1 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 [8,9], linear real arithmetic [1], non-linear real arithmetic [10], Presburger arithmetic [11, 4,12], and arrays [13,14,15].

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 [16], 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 article, we introduce a new Craig interpolation method for bit-vector arithmetic, initially focusing on arithmetic bit-vector operations including addition, multiplication, and division. Like [16], 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.

This is an extended version of a paper presented at FMCAD 2018 [17]. Compared to the conference version, this article considers an extended fragment of bit-vector logic, including also concatenation and extraction operations on bit-vectors, as well as bit-wise operators like bvor or bvnot. We show that the representation of concatenation and extraction using uninterpreted predicates is sufficient to obtain an interpolation procedure for the quantifier-free structural fragment of bit-vector logic, i.e., bit-vector constraints with only concatenation, extraction, and positive equations [18,19]. Bit-wise operations are handled via a direct translation to Presburger arithmetic akin to bit-blasting.

The contributions of the article are: 1. a new calculus for non-linear integer arithmetic, which can eliminate quantifiers (in certain cases) and extract Craig interpolants (Section 3); 2. a corresponding calculus for arithmetic bit-vector constraints (Section 4); 3. the extension of the calculus to handle concatenation, extraction, and bit-wise operations (Section 5); 4. an experimental evaluation using SMT-LIB and model checking benchmarks (Section 6).

1.1 Example 1: Interpolating Arithmetic Bit-Vector Operations

We start by considering one of the examples from [16], the interpolation problem $A \wedge B$ defined by

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

where all variables range over unsigned 8-bit bit-vectors. The function bvadd_8 represents addition of two bit-vectors, while the predicate bvule_8 is the unsigned $\leq$ comparison. An interpolant for $A \wedge B$ is a formula I such that the implications $A \Rightarrow I$ and $B \Rightarrow \neg I$ hold, and such that only variables common to A and B occur in I.

An eager encoding into Presburger arithmetic (linear integer arithmetic, LIA) would typically add variables to handle wrap-around semantics, e.g., mapping $y'_4 = \mathsf{bvadd}_8(y_4, 1)$ to $y'_4 = y_4 + 1 - 2^8\sigma_1 \wedge 0 \leq y'_4 < 2^8 \wedge 0 \leq \sigma_1 \leq 1$. This yields a formula in Presburger arithmetic that exactly models the bit-vector semantics, and can be solved and interpolated using existing methods implemented in SMT solvers. Interpolants can be mapped back to a pure bit-vector formula if needed. However, additional variables and large coefficients tend to be hard both for solving and interpolation; the LIA interpolant presented in [16] for $A \wedge B$ is the somewhat complicated formula $I_{LIA} = -255 \leq y_2 - y_3 + 256\lfloor -1\frac{y_2}{256}\rfloor$.

Our approach translates bit-vector formulas to our *core language* — an extension of Presburger arithmetic with constructs to express bit-vector domains, wrap-around semantics and operations that can be simplified in different ways, such as *bvmul*. For example, domain predicate $in_w(x)$ expresses that variable x belongs to the value range of a bit-vector of width w. Similarly, predicate $ubmod_w(x, y)$ expresses the unsigned wrap-around semantics without explicitly encoding it. Translating A and B to the core language yields:

$$\begin{aligned} A_{\text{core}} = &\quad \psi_A \wedge ubmod_8(y_4 + 1, c_1) \wedge c_1 > y_3 \wedge y_2 = c_1 \\ B_{\text{core}} = &\quad \psi_B \wedge ubmod_8(y_2 + 1, c_2) \wedge c_2 \leq y_3 \wedge y_7 = 3 \wedge y_7 = c_2 \end{aligned}$$

where $\psi_A = in_8(y_2) \wedge in_8(y_3) \wedge in_8(y_4) \wedge in_8(c_1)$ and $\psi_B = in_8(y_2) \wedge in_8(y_3) \wedge in_8(y_7) \wedge in_8(c_2)$ capture the domain constraints. The core language enables a layered calculus that encodes predicates on a case by case basis, preferring simpler encodings whenever possible. In our example, rule BMOD-SPLIT splits the $ubmod_8(y_2 + 1, c_2)$ into the only two relevant cases based on the bounds of y_2 implied by $A_{\text{core}}, B_{\text{core}}$:

$$\frac{\ldots, 0 \leq c_2 < 256, y_2 + 1 = c_2 \vdash \quad \ldots, 0 \leq c_2 < 256, y_2 + 1 = c_2 + 256 \vdash}{\ldots, ubmod_8(y_2 + 1, c_2) \vdash} \text{ BMOD-SPLIT}$$

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

When variable bounds are tight enough and there are only a few cases, case splits are more efficient than σ variables. However, that is not always the case and our calculus lazily decides how to handle each occurrence. Simpler proofs also lead to simpler and more compact interpolants; using our lazy approach, the final interpolant in the example is $I_{LAZY} = y_3 < y_2$, which is simple and avoids the division operator in I_{LIA}. We will revisit this example in Section 4.4 and explain in greater detail how this interpolant is obtained.

1.2 Example 2: Interpolating Structural Bit-Vector Operations

We continue with a (reduced) example taken from [19], a formula of equalities between (slices of) bit-vectors of length 8:

$$x[5:0] = 22 \wedge y[7:2] = 6 \wedge x = y$$

where $x[u:l]$ is the *extraction* of the slice of bits from uth down to lth (inclusive). In the previous example the bit-vector formula was translated to integer arithmetic, however this can sometimes be inefficient when dealing with *structural* bit-vector operations, e.g., extractions and concatenations. A direct translation to integer arithmetic has a hard time to isolate the conflict, since integer operations cannot capture extractions in a natural way.

Instead, it is possible to split the bit-vectors into segments

$x[5:0] = 22$	$y[7:2] = 6$	$x = y$
	$y[7:6] = 0$	$x[7:6] = y[7:6]$
$x[5:2] = 5$	$y[5:2] = 6$	$x[5:2] = y[5:2]$
$x[1:0] = 2$		$x[1:0] = y[1:0]$

Given this *decomposition* of the bit-vectors, it is easy see the conflict $x[5:2] = 5 \neq 6 = y[5:2]$ without a translation to integers. Interpolants can in this setting be extracted by referring to individual slices of bit-vectors, with the help of the extraction operator. In Section 5 we show how bit-vectors can be decomposed in this manner using an interpolating calculus.

1.3 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 [20], which uses a layered approach [16] 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.

A similarly layered approach, proposed in [21], can be used to compute function summaries in bounded model checking. When bounded model checking is able to

prove (bounded) safety of a program, Craig interpolation can subsequently be used to extract function summaries; such summaries can later be useful to speed up other verification tasks. To handle bit-vector constraints in this context, [21] successively applies more and more precise over-approximations of bit-vectors: using uninterpreted functions, linear real arithmetic, and finally using precise bit-blasting. Interpolants are computed in the coarsest theory that was able to prove safety of a verification task.

Other related work has focused on interpolation for fragments of bit-vector logic. In [22], 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 [23].

Looking more generally at model checking for finite-state systems formulated over the theory of bit-vectors (often called word-level model checking), lazy approaches to handle complex bit-vector operations have been proposed. In [24], an approximation method for model checking RTL designs is defined that instantiates complex bit-vector operations lazily. Initially, such operations are over-approximated by leaving the results unconstrained; when spurious counterexamples occur, the approximation is refined by adding additional constraints, or ultimately by precisely instantiating the operator. Such approaches are independent of the underlying finite-state model checking algorithm, and do not necessarily involve Craig interpolation, however.

The core logic of bit-vectors (formulas with only concatenation, extraction, and positive equations) was identified in [18] to be solvable in polynomial time.[1] Our work is inspired by the decomposition-based decision procedure for this fragment developed in [19], where the authors present an algorithm together with a data-structure designed for solving formulas over the core logic of bit-vectors efficiently. To the best of our knowledge, Craig interpolation for the structural fragment has not been considered previously.

2 Preliminaries: The Base Logic

We formulate our approach on top of a simple logic of Presburger arithmetic constraints combined with uninterpreted predicates, introduced in [25] and extended in [4,11] 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 predicate symbols with fixed arity, and α over the set $\mathbb{Z}$ of integers. The syntax of terms and formulas 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

[1] To avoid confusing with our own "core" fragment introduction in Section 4, we call the logic from [18] the "structural fragment" in this article.

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

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

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

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

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

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

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

$s = t$ as a shorthand of $s - t = 0$, inequalities $s \leq t$ and $t \geq s$ 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 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., [26]), 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.

2.1 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 [25], a part of which is shown in Figure 1. If Γ, Δ are finite sets of formulas, 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 labeled 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 Figure 1, the calculus in [25] 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 [27], but as discussed in [25] the calculus can be made complete (by adding slightly more sophisticated quantifier handling) for interesting undecidable fragments, for instance for sequents $\vdash \phi$ in which ϕ contains $\exists/\forall$ only under an even/odd number of negations.

For quantifier-free input formulas, proof search can be implemented in depth-first style following the core concepts of DPLL(T) [28]: 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.[2]

2.2 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 ($P = \emptyset$) 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 [25] is moreover able to eliminate arbitrarily nested quantifiers, and can be used similarly to prove validity of sequents with quantifiers. A recent independent evaluation [29] 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.

2.3 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 [30]. An interpolating version of our sequent calculus has been presented in [4,11], and is summarised in Figure 2. To keep track of the partitions A, B, the calculus operates on labeled 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-labeled formulas, and I is an unlabeled formula, then $\Gamma \vdash \Delta \blacktriangleright I$ is an *interpolating sequent.*

Semantics of interpolating sequents is defined using the following 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 labeled formulas. 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. constants and predicates 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.

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 2.1. Once a closed

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

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

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

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

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

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

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

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

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.

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 Figure 2. Then, starting from the leaves, intermediate interpolants are computed and propagated back to the root, leading to an interpolating sequent $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$.

3 Solving Non-Linear Constraints

We extend the base logic in three steps: in this section, symbols and rules are added to solve *non-linear diophantine problems;* a second extension is then done in Section 4 to handle *arithmetic bit-vector constraints;* and, finally, additional symbols to express *structural bit-vector constraints* are introduced in Section 5. All 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 predicates contains a distinguished 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 $\neg\times(s,t,r) \Leftrightarrow \exists x.(\times(s,t,x) \wedge x \neq r)$.

3.1 Calculus Rules for Non-Linear Constraints

We now introduce 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 predicate symbol, 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 3.1.4). The rule classes are:

- *Deriving Implied Equalities with Gröbner Bases*: if implied linear equalities can be found using Buchberger's algorithm these can be added to the proof goal.
- *Interval Constraint Propagation*: if new bounds for constants can be derived from existing bounds these can be added to the proof goal.
- *Cross-Multiplication of Inequalities*:, if two terms are known to be non-negative, then the non-negativity of their product can be added to the proof goal.
- *Interval Splitting*: as a last resort, the proof branch can be split by dividing the possible values for a constant or variable in half.
- $\times$*-Elimination*: if a occurrence of $\times$ is implied by other literals, it can be eliminated from the proof goal.

3.1.1 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.[3] 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 $I = \mathit{Ideal}(\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$ 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 [31]. Any $f \in I$ corresponds to an equation $f = 0$ that logically follows from the literals, and can therefore be added to a proof goal:

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

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 $\times$-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 [32], and then apply Gaussian elimination to find linear basis polynomials (or directly by choosing a suitable monomial order).

[3] 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.

Example 1 Consider the formula for the square of a sum: $(x+y)^2 = x^2 + 2xy + y^2$. We can show its validity by rewriting it to normal form and constructing a proof. 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 (using calculus rules not presented in this paper, see Section 2.1).

3.1.2 Interval Constraint Propagation (ICP)

Our main technique for inequality reasoning in the presence of $\times$-predicates is interval constraint propagation (ICP) [33]. ICP is a fixed-point computation on the lattice $\mathbb{I}^S$ of functions mapping constants and variables $S = C \cup X$ to intervals $\mathbb{I}$, and can efficiently approximate the value ranges of symbols. We define the lattice $\mathbb{I}$ of intervals and the lattice $\mathbb{I}^S$ of interval assignments as follows; $S \to \mathbb{I}$ represents the set of (total) functions from $S = C \cup X$ to $\mathbb{I}$, and $\bot$ is the distinguished bottom element of $\mathbb{I}^S$:

$$\begin{aligned}\mathbb{I} &= \{[x,y] \mid x,y \in \mathbb{Z}, x \le y\} \ \cup\ \{(-\infty,\infty)\} \ \cup \\ &\quad\ \{(-\infty,y) \mid y \in \mathbb{Z}\} \ \cup\ \{(x,\infty) \mid x \in \mathbb{Z}\} \\ \mathbb{I}^S &= (S \to \mathbb{I}) \cup \{\bot\}\end{aligned}$$

We denote the (point-wise) join and meet on $\mathbb{I}^S$ with $\sqcup, \sqcap$, respectively.

To define the fixed-point computation, we then introduce abstraction and concretisation functions that connect the lattice $\mathbb{I}^S$ with the powerset lattice $\mathcal{P}(S \to \mathbb{Z})$ of value assignments. The abstraction of a set $V \in \mathcal{P}(S \to \mathbb{Z})$ of value assignments is the least element $\alpha(V)$ of $\mathbb{I}^S$ such that the interval $\alpha(V)(c)$ assigned to a symbol $c \in S$ contains all values of c in V (or $\alpha(V) = \bot$ if V is empty). The abstraction function $\alpha : \mathcal{P}(S \to \mathbb{Z}) \to \mathbb{I}^S$ is formally defined as follows:

$$\alpha(V) = \bigsqcup_{\beta \in V} \{c \mapsto [\beta(c), \beta(c)] \mid c \in S\} \qquad \text{for } V \in \mathcal{P}(S \to \mathbb{Z}).$$

The concretisation $\gamma(I)$ of some interval assignment $I \in \mathbb{I}^S$ is the set V of all value assignments that stay within the intervals specified by I. More formally, $\gamma : \mathbb{I}^S \to \mathcal{P}(S \to \mathbb{Z})$ is defined by:

$$\gamma(I) = \begin{cases}\emptyset & \text{if } I = \bot \\ \{\beta : S \to \mathbb{Z} \mid \beta(c) \in I(c) \text{ for all } c \in S\} & \text{otherwise}\end{cases} \qquad \text{for } I \in \mathbb{I}^S.$$

The result of ICP can then be defined as the greatest fixed-point of a monotonic propagation function $\mathit{Prop} : \mathbb{I}^S \to \mathbb{I}^S$ on the lattice $\mathbb{I}^S$. Propagation can be

defined separately for each formula occurring in a sequent; in particular, propagation $Prop_\phi : \mathbb{I}^S \to \mathbb{I}^S$ for a multiplication literal $\phi = \times(s,t,r)$ is defined as:

$$Prop_{\times(s,t,r)}(I) \;=\; \alpha(\{\beta \in \gamma(I) \mid \beta \models s \cdot t = r\})$$

This means, propagation eliminates values from the intervals that are inconsistent with $\times(s,t,r)$. Propagation for equalities $t = 0$ and inequalities $t \leq 0$ is defined similarly; in practice, also any monotonic over-approximation of $Prop_\phi$ can be used instead of $Prop_\phi$, at the cost of more over-approximate results in the end.

Given a set $\{\phi_1, \ldots, \phi_n\}$ of formulas, the overall propagation function $Prop = Prop_{\{\phi_1,\ldots,\phi_n\}}$ is the meet of the individual propagators:

$$Prop_{\{\phi_1,\ldots,\phi_n\}}(I) \;=\; \prod_{i=1}^{n} Prop_{\phi_i}(I)$$

The ICP rule assumes that a greatest fixed-point gfp $Prop_{\{\phi_1,\ldots,\phi_n\}}$ for equality, inequality, and multiplication literals $\phi_1, \ldots, \phi_n$ in a sequent has been computed, and adds resulting bounds for a constant c:

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

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

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

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

3.1.3 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 [34]. 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 3.1.1, we prefer to avoid the introduction of new multiplication literals during proof search. By disallowing non-linear terms, we avoid the introduction of more and more complex terms and thus only add $s \cdot t \geq 0$ if the term $s \cdot t$ can be expressed linearly. For this, we again write $I = Ideal(\{s_i \cdot t_i - r_i\}_{i=1}^n \cup \{e_j\}_{j=1}^m)$ for the ideal induced by equations and $\times$-literals:

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

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.

3.1.4 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 $c \in C \cup X$:

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

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.

3.1.5 $\times$-Elimination

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

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

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.

3.2 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 uninterpreted predicates. By combining the QE approach in Section 2.2 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 3.1 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 [35], 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.

3.3 Craig Interpolation for Non-Linear Constraints

To carry over the Craig interpolation approach from Section 2.3 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

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta \end{array} \quad \cdots \quad \begin{array}{c}\vdots\\ \Gamma, \Gamma', \Gamma_n \vdash \Delta_n, \Delta', \Delta \end{array}}{\begin{array}{c}\Gamma, \Gamma' \vdash \Delta', \Delta\\ \vdots \end{array}} R \tag{1}$$

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 it can simulate the effect of R (as in (1)). This is done by introducing QAx in the antecedent of a sequent, applying the rule $\forall$-LEFT to instantiate the axiom with the constants S and obtain Ax, and then applying propositional rules. The propositional rules $\vee$-LEFT and $\neg$-LEFT are used to eliminate implications $\rightarrow$ (which are short-hand for $\neg, \vee$), and the rule $\wedge$-RIGHT to eliminate the conjunction $\bigwedge_{i=1}^{n}$:

$$\dfrac{\dfrac{\begin{array}{c}*\\ \vdots\\ \Gamma, \Gamma', \bigwedge \Gamma' \rightarrow \bigvee \Delta' \vdash \Delta', \Delta\end{array} \qquad \begin{array}{c}\Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta \quad \cdots \quad \Gamma, \Gamma', \Gamma_n \vdash \Delta_n, \Delta', \Delta\\ \vdots\end{array}}{\Gamma, \Gamma', Ax \vdash \Delta', \Delta}\ \vee\text{-LEFT}, \neg\text{-LEFT}, \wedge\text{-RIGHT}*}{\Gamma, \Gamma', \forall S.Ax \vdash \Delta', \Delta}\ \forall\text{-LEFT}*$$

This construction leads to a proof using only the standard rules from Section 2.1, 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 $\forall\text{-LEFT}_{L/R}$ (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 4.4.[4]

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

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

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

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

4.1 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, $\mathsf{BVOP_a} = \{\mathsf{bvadd}_w, \mathsf{bvmul}_w, \mathsf{bvudiv}_w, \mathsf{bvneg}_w, \mathsf{ze}_{w+w'}, \mathsf{bvule}_w, \mathsf{bvsle}_w\}$. An extension to bit-vector concatenation, extraction, and bit-wise functions is presented in Section 5. 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 follows the FixedSizeBitVectors[5] theory of the SMT-LIB [36]. 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.

[4] 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.

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

$$
\begin{array}{rcl}
\mathsf{bvadd}_w(s,t) = r & \twoheadrightarrow & \mathit{ubmod}_w(s+t,r) \\
\mathsf{bvneg}_w(s) = r & \twoheadrightarrow & \mathit{ubmod}_w(-s,r) \\
\mathsf{bvmul}_w(s,t) = r & \twoheadrightarrow & \exists x.\ \big(\times(s,t,x) \wedge \mathit{ubmod}_w(x,r)\big) \\
\mathsf{ze}_{w+w'}(s) = r & \twoheadrightarrow & s = r \\
\mathsf{bvsle}_w(s,t) & \twoheadrightarrow & \exists x,y.\ (\mathit{sbmod}_w(s,x) \wedge \mathit{sbmod}_w(t,y) \wedge x \leq y) \\
\neg\mathsf{bvsle}_w(s,t) & \twoheadrightarrow & \exists x,y.\ (\mathit{sbmod}_w(s,x) \wedge \mathit{sbmod}_w(t,y) \wedge x > y) \\
\mathsf{bvule}_w(s,t) & \twoheadrightarrow & s \leq t \\
\neg\mathsf{bvule}_w(s,t) & \twoheadrightarrow & s > t \\
\mathsf{bvudiv}_w(s,t) = r & \twoheadrightarrow & \big(t = 0 \wedge r = 2^w - 1\big) \vee \\
 & & \big(t \geq 1 \wedge \exists x.\,(\times(t,r,x) \wedge s - t < x \leq s)\big)
\end{array}
$$

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

The translation from bit-vector constraints ϕ to core formulas ϕ_{core} has two parts: first, $\mathsf{BVOP_a}$ 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.

$\mathsf{BVOP_a}$ *Elimination.* Like in Section 3, 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 $\mathsf{BVOP_a}$ 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 Figure 3. Since the rules for the predicates bvsle_w and bvule_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 3, 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 maps the operands to a signed bit-vector domain $\{-2^{w-1}, \ldots, 2^{w-1}-1\}$, in which then the arithmetic inequality predicates $\leq, >$ can be used. 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.

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 $\mathit{in}_w(x) =_{\text{def}} (0 \leq x < 2^w)$ and define

$$\phi_{core} \;=\; \Big(\bigwedge_{x \in S} \mathit{in}_{w_x}(x)\Big) \rightarrow \phi'$$

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 Figure 3 to ϕ. Similar constraints are used to express quantification over bit-vectors, for instance $\exists x.\ (in_w(x) \wedge \ldots)$ and $\forall x.\ (in_w(x) \rightarrow \ldots)$.

Example 3 Consider `challenge/multiplyOverflow.smt2`, a problem from SMT-LIB QF_BV containing 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:

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

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:

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

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

$$\begin{pmatrix}in_{32}(a) \wedge in_{32}(b) \wedge in_{32}(c_1) \wedge in_{64}(c_2) \wedge in_{64}(c_3) \wedge in_{64}(c_4) \wedge \\ \begin{pmatrix}(a = 0 \wedge c_1 = 2^{32}-1) \vee \\ (a \geq 1 \wedge \exists x.(\times(a, c_1, x) \wedge 2^{32} - 1 - a < x \leq 2^{32}-1))\end{pmatrix} \wedge \\ \exists z.\ (\times(c_3, c_4, z) \wedge \mathit{ubmod}_{64}(z, c_2)) \wedge a = c_3 \wedge b = c_4 \wedge b \leq c_1\end{pmatrix} \rightarrow c_2 \leq 2^{32}-1$$

4.2 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 Figure 4. Our implementation in addition applies rules for Boolean and Presburger connectives, for instance to inline equations $x = t$ or to propagate inequalities, not shown here.

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 Figure 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 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:

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

$$\text{if } \Gamma \cup \{\neg\psi \mid \psi \in \Delta\} : \phi \twoheadrightarrow \phi'$$

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

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

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

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

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

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

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

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.

The lower half of Figure 4 shows three of the bit-vector-specific rules. The BOUND-RW rule 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 [25].

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 $bmod_{a'}^{b'}(s',r')$, 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 Consider $\mathsf{bvadd}_{32}(\mathsf{bvadd}_{32}(a,b),c)$, which corresponds to the expression $ubmod_{32}(a+b,r_1) \wedge ubmod_{32}(r_1+c,r_2)$ 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 $a+b+c \prec r_1+c$.

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

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

Example 5 We continue Example 3 and show that ϕ_{core} is valid, focusing on the $a \geq 1$ case of bvudiv_{32}. The proof (Figure 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 $b \leq c_1$ we derive $(a-1)(c_1 - b) = ac_1 - ab - c_1 + b \geq 0$. Using the products $\times(a,b,d)$ and $\times(a,c_1,e)$, we can express it linearly as $e - d - c_1 + b \geq 0$. The proof branch can then be closed using standard arithmetic reasoning. The implementation of our procedure can easily find the outlined proof automatically.

4.3 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:

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

assuming the bounds $\left\lfloor \frac{lbound(\Pi,s)-a}{b-a} \right\rfloor = l$ 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 is better to choose a direct encoding of the remainder operation in Presburger arithmetic:

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

where c is assumed to be a fresh constant. Rule BMOD-CONST corresponds to the encoding chosen in [16].

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. We investigate experimentally in Section 6.1 how many applications of the rules BMOD-SPLIT and BMOD-CONST are needed to prove formulas satisfiable or unsatisfiable, and show that the numbers are surprisingly low, in particular in the unsatisfiable case.

4.4 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 3.2. 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 3.3 carries over to bit-vectors, with theory 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 [16]). 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.

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 Figure 4, and apply RW-LEFT, RW-RIGHT only when the modified formula ϕ is a literal.

Example 6 We recall the example from Section 1.1, 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 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:

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

We only show one of the cases, $\mathcal{P}$, resulting from splitting the axiom $\lfloor Ax \rfloor_R$ using the rules from Figure 2. The final interpolant $I_{LAZY} = y_3 < y_2$ 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 [11].

5 Interpolation in the Presence of Extract and Concat

Two bit-vector operations which are more tricky to translate to integer arithmetic are the extraction of bits from a larger bit-vector, and the concatenation of two bit-vectors. This is formalised using the function $\mathsf{bvextract}_{[u,l]}$, which cuts out a slice of $u - l + 1$ consecutive bits, and the function $\mathsf{bvconcat}_{v+w}$ forming a bit-vector of length $v + w$. We call the fragment of bit-vector logic containing only these operations, and positive equalities, the *structural fragment* of bit-vector theory.

Example 7 Consider the following bit-vectors:

$$\begin{array}{lll} c_1 = [0,0] = 0 & c_2 = [1,1] = 3 & c_3 = [0,1] = 1 \\ c_4 = [0,0,1,1] = 3 & c_5 = [1,1,0,0] = 12 & \end{array}$$

where a bit-sequence $[b_n, \ldots b_0]$ is represented by the number $\sum_{i=0}^{n} b_i 2^i$. The following equations hold between those bit-vectors:

$$\begin{array}{ll} \mathsf{bvconcat}_{2+2}(c_1, c_2) = c_4 & \mathsf{bvconcat}_{2+2}(c_2, c_1) = c_5 \\ \mathsf{bvextract}_{[3,2]}(c_4) = c_1 & \mathsf{bvextract}_{[1,0]}(c_4) = c_2 \\ \mathsf{bvextract}_{[2,1]}(c_4) = c_3\ . & \end{array}$$

While it is possible to translate extractions and concatenations to integer arithmetic, it is not always efficient. We show here that it can be more efficient to keep extractions abstract and only convert at need. This approach can also help to compute simpler interpolants.

Example 8 We consider the example from [19], a formula over the structural fragment, which we divide into two parts

$$\begin{array}{ll} A = & (x[5:0] = z \wedge z[5:2] = 11) \\ B = & (y[7:2] = 6 \wedge x = y) \end{array}$$

	7	6	5	4	3	2	1	0
$x_{[8]}$	*	*	1	0	1	1	*	*
$y_{[8]}$	0	0	0	1	1	0	*	*

	15	14	13	12	11	10	9	...
$x_{[16]}$	*	*	1	0	1	1	*	...
$y_{[16]}$	0	0	0	1	1	0	*	...

Fig. 6 Illustration of Example 8. The conflict is due to the contradicting assignments to slices of x and y. The upper table shows the situation with bit-vectors of size 8, the lower one with size 16.

where the bit-vectors x, y are of width 8, and z is of width 6. The unsatisfiability of $A \wedge B$ could be proved, as before, by translating the bit-vector constraints to our core language:

$$A_{\text{core}} = \quad in_8(x) \wedge in_6(z) \wedge ubmod_6(x, z) \wedge \exists c_1.\ (in_2(c_1) \wedge z = 11 \cdot 2^2 + c_1)$$
$$B_{\text{core}} = \quad in_8(x) \wedge in_8(y) \wedge \exists c_2.\ (in_2(c_2) \wedge y = 6 \cdot 2^2 + c_2) \wedge x = y$$

Using the interpolation procedure presented in Section 4, we can compute the following interpolant:

$$I = \quad \exists c.\ (x = 44 + 64c \vee x = 45 + 64c \vee x = 46 + 64c \vee x = 46 + 64c)$$

The conjunction is unsatisfiable due to the conflicting assignment of a small slice in x and y, as illustrated in Figure 6. However, when translating to integer arithmetic the overall structure is lost, and the interpolant contains a lot of redundancy. The situation gets worse with increasing bit-widths. Consider a formula where the width of x, y are doubled, while the widths of the extractions are kept constant:

$$A' = \quad (x[13:8] = z \wedge z[5:2] = 11)$$
$$B' = \quad (y[15:10] = 6 \wedge x = y)$$

The same procedure now yields an interpolant of exponentially greater size:

$$I' = \quad \exists c.\ (x = 4097 + 8192c \vee x = 4098 + 8192c \vee \cdots \vee x = 5120 + 8192c)$$

We will explain in the next sections how more succinct interpolants can be computed by eliminating the extraction operation only lazily.

5.1 The Structural Fragment

In [19], a polynomial fragment of the bit-vector theory is identified, consisting of formulas that only contain extractions, concatenations, and positive equalities. The satisfiability of formulas in the structural fragment is decidable in polynomial time by a congruence closure procedure over decomposed bit-vectors [18].

Definition 1 The *structural fragment* consists of bit-vector formulas of the form $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ where each ϕ_i is an equation constructed using bit-vector variables, concrete bit-vectors, and the operators $\mathsf{bvextract}_{[u,l]}$ and $\mathsf{bvconcat}_{v+w}$.

$$
\begin{aligned}
\mathsf{bvextract}_{[u,l]}(s) = r \;&\twoheadrightarrow\; extr_l^u(s,r)\\
\mathsf{bvconcat}_{w+v}(s,t) = r \;&\twoheadrightarrow\; in_{w+v}(r) \wedge extr_v^{w+v-1}(r,s) \wedge extr_0^{v-1}(r,t)\\
\mathsf{bvnot}_w(s) = r \;&\twoheadrightarrow\; in_w(r) \wedge \bigwedge_{i=0}^{w-1} \exists x.\ (extr_i^i(s,x) \wedge extr_i^i(r,1-x)\\
\mathsf{bvand}_w(s,t) = r \;&\twoheadrightarrow\; in_w(r) \wedge \bigwedge_{i=0}^{w-1} \exists s_b,t_b,r_b.\ \begin{pmatrix} extr_i^i(s,s_b) \wedge extr_i^i(t,t_b) \wedge extr_i^i(r,r_b)\\ \wedge\ r_b \leq s_b \wedge r_b \leq t_b \wedge r_b \geq s_b + t_b - 1 \end{pmatrix}\\
\mathsf{bvor}_w(s,t) = r \;&\twoheadrightarrow\; in_w(r) \wedge \bigwedge_{i=0}^{w-1} \exists s_b,t_b,r_b.\ \begin{pmatrix} extr_i^i(s,s_b) \wedge extr_i^i(t,t_b) \wedge extr_i^i(r,r_b)\\ \wedge\ r_b \geq s_b \wedge r_b \geq t_b \wedge r_b \leq s_b + t_b \end{pmatrix}\\
\mathsf{bvxor}_w(s,t) = r \;&\twoheadrightarrow\; in_w(r) \wedge \bigwedge_{i=0}^{w-1} \exists s_b,t_b,r_b.\ \begin{pmatrix} extr_i^i(s,s_b) \wedge extr_i^i(t,t_b) \wedge extr_i^i(r,r_b)\\ \wedge\ ubmod_l(s_b + t_b, r_b) \end{pmatrix}
\end{aligned}
$$

Fig. 7 Rules translating structural and bit-wise operations into the extended core language. As before, the rules are only applied in negative positions. Note that u, l, v, w are integer constants.

Note that a formula containing $\mathsf{bvconcat}_{v+w}$ can be translated into an equisatisfiable formula that only uses $\mathsf{bvextract}_{[u,l]}$. We illustrate the translation with an example, and introduce the formal rule in Figure 7. Consider the formula $\phi[\mathsf{bvconcat}_{v+w}(s,t)]$ containing the concatenation of two bit-vectors. The concatenation can be eliminated by introducing an existentially quantified variable to represent the result of the concatenation; the relationship with the arguments is established using extraction terms:

$$\exists x.\ (in_{v+w}(x) \wedge \mathsf{bvextract}_{[v+w-1,w]}(x) = s \wedge \mathsf{bvextract}_{[w-1,0]}(x) = t \wedge \phi[x])$$

To define a formal calculus for the structural fragment, we introduce an *extended* core language by adding a further family $P_{ex} = \{extr_l^u \mid u, l \in \mathbb{N}, u \geq l\}$ of predicates representing extraction from bit-vectors. Semantically, $extr_l^u(s,t)$ relates s and t if t is the result of extracting the bits u through l from s. This is formally expressed as the existence of two bit-vectors x, y such that t can be made equal to s by prepending x and appending y:

$$extr_l^u(s,t) \;\Leftrightarrow\; in_{u-l+1}(t) \wedge \exists x, y.\ (in_l(y) \wedge s = x \cdot 2^{u+1} + t \cdot 2^l + y)$$

Note that the argument s is not bounded, which implies that the definition contains a bound for the lower slice y, but not for x.

The rewriting rules in Figure 7 define how extraction and concatenation are translated to the P_{ex} predicates, following the same schema as in Section 4. As a side-effect of adding $\mathsf{bvextract}_{[u,l]}$ and $\mathsf{bvconcat}_{w+v}$, and moving beyond the structural fragment, the calculus can also reason about the bit-wise operators $\mathsf{BVOP}_{\mathsf{bv}} = \{\mathsf{bvnot}, \mathsf{bvand}, \mathsf{bvor}, \mathsf{bvxor}\}$, by extracting the individual bits of the operands and encoding the Boolean semantics using inequalities. The rewriting rules for such an encoding are given Figure 7 as well, and are also used in our implementation.

5.2 Bit-Vector Decomposition

As demonstrated in Section 1.2, unsatisfiability of formulas can sometimes be proven by just focusing on the right slice of bit-vectors; the challenge lies in how

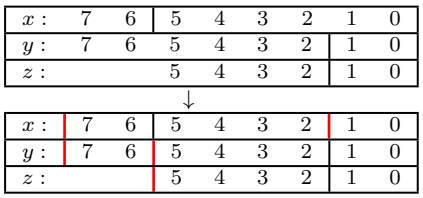

Fig. 8 Cut point propagation of bit-vectors x, y and z. The top table contains each bit-vector and the corresponding cut points. Since all bit-vectors are related by some constraints, all cut points (within the width of the bit-vector) are propagated, e.g., the cut point for y between bit 2 and 1 is propagated to x.

to decompose the bit-vectors to find the conflicts. Intuitively, there is no need to split apart bits which are never constrained individually. We follow the procedure described in [19], and use the notion of *cut points* for this reason. Cut points of a bit-vector variable determine the slices that need to be considered, and the points at which the bit-vectors might have to be decomposed, and are determine by the boundaries of extraction operations.

More formally, given a formula ϕ in the structural fragment over set $S = C \cup X$ of constants and variables, a cut point configuration is a function $\mathcal{C} : S \to \mathcal{P}(\mathbb{N})$ satisfying the following properties:

- for each $extr_l^u(s,t)$ literal, it is the case that:
  - $\{l, u+1\} \subseteq \mathcal{C}(s)$, and
  - $\{i - l \mid i \in \mathcal{C}(s) \text{ with } l \leq i \leq u+1\} = \{i \in \mathcal{C}(t) \mid i \leq u - l + 1\}$.
- for each equality $s = t$ it is the case that $\mathcal{C}(s) = \mathcal{C}(t)$.

The set of cut points for all bit-vectors can be obtained by a fix-point computation, which begins with all cut points from extractions, and then propagates using the equalities until all conditions hold.

Example 9 Translated to our extended core language, the constraints from Example 8 are:

$$\begin{aligned} A'_{\text{core}} = &\quad in_8(x) \wedge in_6(z) \wedge extr_0^5(x,z) \wedge extr_2^5(z,11) \\ B'_{\text{core}} = &\quad in_8(x) \wedge in_8(y) \wedge extr_2^7(y,6) \wedge x = y \end{aligned}$$

The extraction literals induce immediate cut points for x, y and z, respectively: $\{6,0\}, \{8,2\}$, and $\{6,2\}$. Since x and z are related by the literal $extr_0^5(x,z)$, the cut point 2 needs to be added to the set for x as well; similarly, due to the equation $x = y$, also the cut points $\{6,0\}$ have to be added for y, and 8 for x. The alignment of the bit-vectors is illustrated in Figure 8, and the complete sets of cut points after propagation are $\{8,6,2,0\}$, $\{8,6,2,0\}$, and $\{6,2,0\}$. If the bit-vectors x, y, z are decomposed according to their cut points, then simple reasoning reveals the inconsistency between $extr_2^5(x,5)$, $extr_2^5(y,6)$, and $x = y$.

5.3 An Interpolating Calculus for Extractions

A decomposition according to the cut points yields a complete and polynomial procedure for the structural fragment [19]. We formalise the splitting of bit-vector

$$\frac{\Gamma,\ \begin{array}{l}\lfloor \exists x_1.\,(extr_i^u(s,x_1) \ \wedge\ extr_{i-l}^{u-l}(r,x_1) \ \wedge\ in_{u-i+1}(x_1))\rfloor_D\\ \lfloor \exists x_2.\,(extr_l^{i-1}(s,x_2) \wedge extr_0^{i-1-l}(r,x_2) \wedge in_{i-1-l}(x_2))\rfloor_D\end{array} \ \vdash\ \Delta \blacktriangleright I}{\Gamma, \lfloor extr_l^u(s,r)\rfloor_D \ \vdash\ \Delta \blacktriangleright I}\ \textsc{extr-split}_D$$

$$\frac{\Gamma, \lfloor \exists x.\,(in_l(x) \wedge ubmod_{u+1}(s, r2^l + x))\rfloor_D \ \vdash\ \Delta \blacktriangleright I}{\Gamma, \lfloor extr_l^u(s,r)\rfloor_D \ \vdash\ \Delta \blacktriangleright I}\ \textsc{extr-arith}_D$$

$$\frac{\Gamma \ \vdash\ s_1 = s_2, \Delta \qquad \Gamma, r_1 = r_2 \ \vdash\ \Delta}{\Gamma, extr_l^u(s_1,r_1), extr_l^u(s_2,r_2) \ \vdash\ \Delta}\ \textsc{extr-cc}$$

$$\frac{\Gamma \ \vdash\ \lfloor s_1 = s_2\rfloor_L, \Delta \blacktriangleright I \qquad \Gamma, \lfloor r_1 = r_2\rfloor_L \ \vdash\ \Delta \blacktriangleright J}{\Gamma, \lfloor extr_l^u(s_1,r_1)\rfloor_L, \lfloor extr_l^u(s_2,r_2)\rfloor_L \ \vdash\ \Delta \blacktriangleright I \vee J}\ \textsc{extr-cc}_{LL}$$

$$\frac{\Gamma \ \vdash\ \lfloor s_1 = s_2\rfloor_R, \Delta \blacktriangleright I \qquad \Gamma, \lfloor r_1 = r_2\rfloor_R \ \vdash\ \Delta \blacktriangleright J}{\Gamma, \lfloor extr_l^u(s_1,r_1)\rfloor_R, \lfloor extr_l^u(s_2,r_2)\rfloor_R \ \vdash\ \Delta \blacktriangleright I \wedge J}\ \textsc{extr-cc}_{RR}$$

$$\frac{\Gamma \ \vdash\ \lfloor s_1 = s_2\rfloor_R, \Delta \blacktriangleright I \qquad \Gamma, \lfloor r_1 = r_2\rfloor_R \ \vdash\ \Delta \blacktriangleright J}{\Gamma, \lfloor extr_l^u(s_1,r_1)\rfloor_L, \lfloor extr_l^u(s_2,r_2)\rfloor_R \ \vdash\ \Delta \blacktriangleright \exists_{Ls_1r_1}.(extr_l^u(s_1,r_1) \wedge I \wedge J)}\ \textsc{extr-cc}_{LR}$$

$$\frac{\Gamma \ \vdash\ \lfloor s_1 = s_2\rfloor_L, \Delta \blacktriangleright I \qquad \Gamma, \lfloor r_1 = r_2\rfloor_L \ \vdash\ \Delta \blacktriangleright J}{\Gamma, \lfloor extr_l^u(s_1,r_1)\rfloor_R, \lfloor extr_l^u(s_2,r_2)\rfloor_L \ \vdash\ \Delta \blacktriangleright \forall_{Rs_1r_1}.(\neg extr_l^u(s_1,r_1) \vee I \vee J)}\ \textsc{extr-cc}_{RL}$$

$$\frac{\Gamma, \lfloor \phi\rfloor_L \ \vdash\ \Delta \blacktriangleright I \qquad \Gamma \ \vdash\ \lfloor \phi\rfloor_R, \Delta \blacktriangleright J}{\Gamma \ \vdash\ \Delta \blacktriangleright (I \vee \neg\phi) \wedge J}\ \textsc{cut}_{LR}$$

$$\frac{\Gamma, \lfloor \phi\rfloor_R \ \vdash\ \Delta \blacktriangleright I \qquad \Gamma \ \vdash\ \lfloor \phi\rfloor_L, \Delta \blacktriangleright J}{\Gamma \ \vdash\ \Delta \blacktriangleright (I \wedge \phi) \vee J}\ \textsc{cut}_{RL}$$

Fig. 9 Rules for handling extraction operations in bit-vector formulas. In EXTR-SPLIT and EXTR-ARITH, $D \in \{L, R\}$. In EXTR-SPLIT, $l < i \leq u$. In EXTR-CC$_{LR}$, $\exists_{Ls_1r_1}$ denotes existential quantification over all constants occurring in s_1 or t_1 but not in $\Gamma_R \cup \Delta_R \cup \{s_2, r_2\}$. In EXTR-CC$_{RL}$, $\forall_{Rs_1r_1}$ denotes universal quantification over all constants occurring in s_1 or t_1 but not in $\Gamma_L \cup \Delta_L \cup \{s_2, r_2\}$. The rules CUT$_{LR}$ and CUT$_{RL}$ are only allowed for formulas ϕ in which all constants are common to both $\Gamma_L \cup \Delta_L$ and $\Gamma_R \cup \Delta_R$.

extractions with an interpolating calculus rule EXTR-SPLIT in Figure 9. Intuitively, we cut the bit-vector s at point i and introduce two existential variables x_1, x_2 corresponding to the two slices. By constraining the corresponding slices of r according to the decomposition, we ensure the original equality between the extraction of s and r. The rule can be generalised to split into several slices at once, allowing for shorter proofs.

After extracts have been split at the cut points, we rely on congruence closure to take care of $extr_l^u$ literals, in a similar way as the procedure in [18]. Congruence closure is in our setting expressed by an axiom schema for functional consistency of the $extr_l^u$ predicates:

$$\forall s_1, s_2, r_1, r_2.\ (extr_l^u(s_1,r_1) \wedge extr_l^u(s_2,r_2) \wedge s_1 = s_2 \rightarrow r_1 = r_2) \tag{2}$$

The axiom states that two *extr*-literals will yield the same result if the first arguments coincide, and if the same bits are extracted. Similar axioms for predicate consistency and functional consistency are used in [4]. To simplify presentation, we model the instantiation of (2) using the calculus rule EXTR-CC in Figure 9; the

figure also gives four interpolating versions of the rule, for the four possible combinations of L/R-labels. The LR/RL rules differ in the label used in their premises. The rules can be derived by instantiating (2) using either $\forall$-LEFT$_L$ or $\forall$-LEFT$_R$, in a similar way as in Section 3.3. In practice, and in our implementation, the calculus rules are used as in classical SMT-style congruence closure: they are triggered when the first arguments of a pair of *extr*-literals have become equal.

Extraction operations can also be translated to arithmetic, which is needed to evaluate extraction from concrete numbers, and when constructing proofs for formulas that are not in the structural fragment (i.e., that combine extraction with other bit-vector operations). The rule EXTR-ARITH encodes an operation $extr_l^u(s,r)$ using a modulo constraint to eliminate bits above position u, and division to strip away bits below position l. We express the division by existentially quantifying the remainder. A more direct rule to perform evaluation is introduced in Section 5.4.

Example 10 We continue Example 9, and show how an interpolating proof can be constructed for the conjunction $A'_{\text{core}} \wedge B'_{\text{core}}$. The root sequent of the proof is $\lfloor A'_{\text{core}} \rfloor_L, \lfloor B'_{\text{core}} \rfloor_R \vdash \emptyset$. In the proof tree shown in Figure 10, we first split the given formulas using Boolean rules. Then, the literal $extr_0^5(x,z)$ can be decomposed using our EXTR-SPLIT rule at the cut point 2, and congruence closure is applied to the literals $extr_2^5(z,c_1)$ and $extr_2^5(z,11)$. We similarly decompose the literal $extr_2^7(y,6)$ at cut point 6, and then use EXTR-ARITH to reduce $extr_0^3(6,c_2)$ to $c_2 = 6$. Finally, we need a second application of congruence closure, EXTR-CC$_{LR}$, to relate the extractions from x and y.

The resulting interpolant is the formula $\exists c.(extr_2^5(x,c) \wedge c = 11)$. The quantifier in the formula stems from the application of EXTR-CC$_{LR}$, which transfers the local symbol c_1 from L to R, and thus makes it shared. A quantifier is needed to eliminate the symbol again from the interpolant. However, as can be seen the quantifier naturally disappears when translating the interpolant back to functional notation, which yields the formula $\mathsf{bvextract}_{[5,2]}(x) = 11$ in the structural fragment.

It is quite easy to see that our calculus is sound and complete for formulas in the structural fragment. The calculus does not guarantee, however, that interpolants computed for formulas in the structural fragment are again in the structural fragment, or that interpolants are quantifier-free. This leads to the question whether the structural fragment is actually closed under interpolation, i.e., whether every unsatisfiability conjunction has an interpolant that is again in the fragment. The answer to this question is positive, and it turns out that our calculus can also be used to compute such interpolants, if the right strategy is used to construct proofs.

Theorem 1 *The structural fragment is closed under interpolation.*

Proof We give a simple proof that follows from the fact that our calculus can model bit-blasting of bit-vector formulas, and builds on work on EUF interpolation in [3,4]. The existence of more compact interpolants, avoiding the need to blast to individual bits, can also be shown, but this is slightly more involved.

Suppose $A \wedge B$ is an unsatisfiable conjunction in the structural fragment, and $A_{\text{core}} \wedge B_{\text{core}}$ the translation to the (extended) core language. Further assume that $x_1, \ldots, x_m$ are the shared bit-vector constants of A, B, and that $w_1, \ldots, w_m$ are their bit-widths, respectively. This means that we are searching for an interpolant in the structural fragment that may only contain the constants $x_1, \ldots, x_m$. To

$$\dfrac{\overset{*}{\vdots}}{\lfloor c_1 = c_2 \rfloor_R, \lfloor c_1 = 11 \rfloor_L, \lfloor c_2 = 6 \rfloor_R, \ldots \ \vdash \ \blacktriangleright c_1 = 11}$$

$$\mathcal{B}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\lfloor x = y \rfloor_R \ \vdash \ \lfloor x = y \rfloor_R \ \blacktriangleright \mathit{true}}\ \text{CLOSE}_{RR} \qquad \mathcal{B}}{\lfloor c_2 = 6 \rfloor_R, \lfloor \mathit{extr}_2^5(x, c_1) \rfloor_L, \lfloor \mathit{extr}_2^5(y, c_2) \rfloor_R, \ldots \ \vdash \ \blacktriangleright \exists c.(\mathit{extr}_2^5(x, c) \land c = 11)}\ \text{EXTR-CC}_{LR}}{\lfloor \mathit{extr}_2^5(y, c_2) \rfloor_R, \lfloor \mathit{extr}_0^3(6, c_2) \rfloor_R, \lfloor \mathit{in}_4(c_2) \rfloor_R, \ldots \ \vdash \ \blacktriangleright I}\ \text{EXTR-ARITH}_R}{\lfloor c_1 = 11 \rfloor_L, \lfloor \mathit{extr}_2^7(y, 6) \rfloor_R \ldots \ \vdash \ \blacktriangleright I}\ \text{EXTR-SPLIT}_R, \ldots}{\mathcal{A}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\cdots \ \vdash \ \lfloor z = z \rfloor_L \ \blacktriangleright \mathit{false}} \qquad \mathcal{A}}{\lfloor \mathit{extr}_2^5(x, c_1) \rfloor_L, \lfloor \mathit{extr}_2^5(z, c_1) \rfloor_L, \lfloor \mathit{in}_4(c_1) \rfloor_L, \lfloor \mathit{extr}_2^5(z, 11) \rfloor_L, \ldots \ \vdash \ \blacktriangleright I}\ \text{EXTR-CC}_{LL}}{\lfloor \exists c.\, (\mathit{extr}_2^5(x, c) \land \mathit{extr}_2^5(z, c) \land \mathit{in}_4(c)) \rfloor_L, \lfloor \mathit{extr}_2^5(z, 11) \rfloor_L, \ldots \ \vdash \ \blacktriangleright I}\ \exists\text{-LEFT}_L, \land\text{-LEFT}_L^*}{\lfloor \mathit{extr}_0^5(x, z) \rfloor_L, \lfloor \mathit{extr}_2^5(z, 11) \rfloor_L, \lfloor \mathit{extr}_2^7(y, 6) \rfloor_R, \lfloor x = y \rfloor_R, \ldots \ \vdash \ \blacktriangleright I}\ \text{EXTR-SPLIT}_L}{\lfloor A'_{\text{core}} \rfloor_L, \lfloor B'_{\text{core}} \rfloor_R \ \vdash \ \blacktriangleright I}\ \land\text{-LEFT}_L^*, \land\text{-LEFT}_R^*$$

with $I = \exists c.(c = 11 \land \mathit{extr}_2^5(x, c))$, or in the structural fragment $\mathsf{bvextract}_{[5,2]}(x) = 11$.

Fig. 10 Proof tree for the formula $A'_{\text{core}} \land B'_{\text{core}}$ in Example 10.

model bit-blasting, we augment $A_{\text{core}}, B_{\text{core}}$ by adding fresh names $\{b_i^j, c_i^j\}_{i,j}$ for the individual bits of $x_1, \ldots, x_m$:

$$A'_{\text{core}} = \quad A_{\text{core}} \land \bigwedge_{\substack{i \in \{1,\ldots,m\} \\ j \in \{0,\ldots,w_i - 1\}}} \mathit{in}_1(b_i^j) \land \mathit{extr}_j^j(x_i, b_i^j)$$

$$B'_{\text{core}} = \quad B_{\text{core}} \land \bigwedge_{\substack{i \in \{1,\ldots,m\} \\ j \in \{0,\ldots,w_i - 1\}}} \mathit{in}_1(c_i^j) \land \mathit{extr}_j^j(x_i, c_i^j)$$

This means that b_i^j / c_i^j is the name of the jth bit of x_i in the L/R partition. Note that a formula I_{core} is an interpolant of $A_{\text{core}} \land B_{\text{core}}$ iff it is an interpolant of $A'_{\text{core}} \land B'_{\text{core}}$, because no shared symbols are added, and we can therefore work with the latter conjunction.

Without loss of generality, we further assume that even every local constant in A'_{core} and B'_{core} occurs as first argument of some extr_l^u-literal, and that every bit in such constants is extracted by some of the literals. This assumption can be ensured by adding further literals to the formulas, without changing the set of possible interpolants.

We then construct a proof for the root sequent $\lfloor A'_{\text{core}} \rfloor_L, \lfloor B'_{\text{core}} \rfloor_R \ \vdash \ \emptyset$ in our interpolating calculus by systematically applying the calculus rules. Rules are applied likewise to the L- and the R-formulas, so that we only mention the L-versions at this point for sake of brevity:

- $\land$-LEFT$_L$ to split conjunctions, and $\exists$-LEFT$_L$ to eliminate quantifiers;
- EXTR-SPLIT$_L$ to split every occurring *extr* predicate down to the level of individual bits;

- EXTR-CC$_{LL}$ whenever two extracts $\lfloor extr^j_j(s,r)\rfloor_L, \lfloor extr^j_j(s,r')\rfloor_L$ for the same bit occur, followed by arithmetic rules to close the left premise; this generates an equation $\lfloor r = r'\rfloor_L$;
- EXTR-CC$_{LL}$ whenever two extracts $\lfloor extr^j_j(s,r)\rfloor_L, \lfloor extr^j_j(t,r')\rfloor_L$ in combination with an equation $\lfloor s = t\rfloor_L$ occur, followed by an application of CLOSE$_{LL}$ to close the left premise; again, this generates an equation $\lfloor r = r'\rfloor_L$;
- EXTR-ARITH$_L$ whenever an extract $\lfloor extr^j_j(\alpha,r)\rfloor_L$ from a concrete number $\alpha \in \mathbb{Z}$ occurs, followed by arithmetic rules to simplify the generated formula to an equation $\lfloor r = 0\rfloor_L$ or $\lfloor r = 1\rfloor_L$;
- EXTR-ARITH$_L$ whenever an extract $\lfloor extr^0_0(s,r)\rfloor_L$ in combination with the domain constraint $in_1(s)$ occurs, i.e., when the single bit of a bit-vector of width 1 is extracted, followed by arithmetic rules to simplify the generated formula to an equation $\lfloor r = s\rfloor_L$.

After those rule applications, we can focus on the obtained bit-level equations in the proof goal: on equations $\lfloor s = t\rfloor_D$ or $\lfloor s = \alpha\rfloor_D$ in which s,t have width 1 (i.e., $in_1(s)$ and $in_1(t)$) and $\alpha \in \{0,1\}$, with $D \in \{L,R\}$. By construction, the set of L-labelled bit-level equations is equi-satisfiable to the original formula A'_{core}, and the R-labelled equations are equi-satisfiable to B'_{core}, so that we can continue constructing a bit-level proof using the equations.

Since the conjunction $A'_{\text{core}} \wedge B'_{\text{core}}$ is by assumption unsatisfiable, there are three possible cases: (i) the equations labelled with L are by themselves unsatisfiable, the interpolant is *false*, and the proof can be closed by applying arithmetic rules; (ii) symmetrically, the equations labelled with R are unsatisfiable, and the interpolant is *true*; or (iii) the equations are unsatisfiable only in combination.

In case (iii), there has to be a chain of equations, alternating between L- and R-equations, that witnesses unsatisfiability. There are several symmetric cases, of which we only consider one:

$$\underbrace{0 = \cdots = b^{j_1}_{i_1}}_{L-\text{equations}} \doteq \underbrace{c^{j_1}_{i_1} = \cdots = c^{j_2}_{i_2}}_{R-\text{equations}} \doteq \underbrace{b^{j_2}_{i_2} = \cdots = b^{j_3}_{i_3}}_{L-\text{equations}} \doteq \cdots \doteq \underbrace{c^{j_k}_{i_k} = \cdots = 1}_{R-\text{equations}} \tag{3}$$

In the other cases, the chain can start in R and end in L, or start and end in the same partition. The dotted equations $b^j_i \doteq c^j_i$ are implied by the literal $\lfloor extr^j_j(x_i, b^j_i)\rfloor_L$, $\lfloor extr^j_j(x_i, c^j_i)\rfloor_R$, but do not exist explicitly in the proof goal.

From (3), it is easy to read off an interpolant for $A \wedge B$ in the structural fragment by summarising the L-chains:

$$I = \quad \mathsf{bvextract}_{[j_1,j_1]}(x_{i_1}) = 0 \wedge \bigwedge_{l=2}^{k-1} \mathsf{bvextract}_{[j_l,j_l]}(x_{i_l}) = \mathsf{bvextract}_{[j_{l+1},j_{l+1}]}(x_{i_{l+1}})$$

Clearly, I follows from A, and $I \wedge B$ is unsatisfiable due to (3).

To construct an interpolant mechanically in our calculus, there are several strategies. The simplest one is to apply the interpolating cut-rule CUT$_{RL}$ to each the formulas $extr^{j_1}_{j_1}(x_{i_1}, 0)$ and $\{\, \exists z.\, (extr^{j_l}_{j_l}(x_{i_l}, z) \wedge extr^{j_{l+1}}_{j_{l+1}}(x_{i_{l+1}}, z))\,\}^{k-1}_{l=2}$, which yields an interpolant I_{core} in the core language that corresponds to the structural interpolant I shown above. □

5.4 A Rewriting Rule for Constant Extraction

Given an extraction $extr_l^u(s,r)$ and bounds on s, it is in some cases possible to determine value of extracted bits. For example, the longest prefix on which the lower and upper bound agree is guaranteed to be present in any consistent value of s. Therefore, extractions that overlap with that prefix yield some bit values of the extraction without knowing the exact value of s. We allow rewriting if the extraction operator falls entirely within the common prefix:

$$\frac{\begin{array}{l}(\mathit{lbound}(\Pi,s)\ \mathsf{xor}\ \mathit{ubound}(\Pi,s)) < 2^l \wedge \\ c = (\mathit{lbound}(\Pi,s)\ \mathit{rem}\ 2^u)\ \mathit{div}\ 2^l \wedge 0 \leq c < 2\end{array}}{\Pi : \mathit{extr}_l^u(s,r) \twoheadrightarrow r = c}\ \textsc{extr-const}$$

where *rem* and *div* are the integer remainder and division, respectively. The rule EXTR-CONST allows in particular evaluation of extractions from constant bit-vectors.

5.5 Splitting of Disequalities

As shown above, proofs can be closed by finding contradicting assignments to (a slice of) a bit-vector. In general, formulas can also contain bit-vector disequalities, i.e., negative equalities between bit-vectors. As an optimisation, disequalities can be split using the notion of cut points as well. Given a formula with a disequality $s \neq t$, we extend the notion of cut point configurations (Section 5.2) by also propagating between s and t. For a cut point $i \in \mathcal{C}(s) = \mathcal{C}(t)$, we can then replace the disequality with a disjunction of two disequalities, as expressed by the following rule:

$$\frac{\begin{array}{r}\Gamma, \mathit{in}_w(s), \mathit{in}_w(t), \mathit{in}_{w-i}(c), \mathit{in}_{w-i}(d), \mathit{extr}_i^{w-1}(s,c), \mathit{extr}_i^{w-1}(t,d)\ \vdash\ c = d, \Delta \\ \Gamma, \mathit{in}_w(s), \mathit{in}_w(t), \mathit{in}_i(c), \mathit{in}_i(d), \mathit{extr}_0^{i-1}(s,c), \mathit{extr}_0^{i-1}(t,d)\ \vdash\ c = d, \Delta\end{array}}{\Gamma, \mathit{in}_w(s), \mathit{in}_w(t)\ \vdash\ s = t, \Delta}\ \neq\textsc{-split}$$

The constants c, d must be fresh and not occur in the conclusion in this rule.

6 Experiments

To evaluate the effectiveness of the approach, the procedures described in this article have been implemented in the Princess theorem prover[6] [25]. The implementation of the full SMT-LIB theory of bit-vectors in Princess is still an ongoing effort, and at this point includes fairly refined versions of the calculi for non-linear arithmetic (Section 3) and for arithmetic bit-vector operators (Section 4). The implementation of the calculus for the structural fragment (Section 5) has been added more recently, and still lacks many optimisations that could be applied. Support for bit-wise operations (like bvand) is also quite naïve at the moment, and simply bit-blasts each bit-wise operation separately by introducing $\mathsf{bvextract}_{[i,i]}$ terms for the individual bits, as shown in Figure 7. A more refined encoding would choose, for each sub-expression, whether the arithmetic encoding or bit-blasting should

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

be applied, but this refinement is left for future work. The implementation also supports the SMT-LIB shift operators, which are handled by splitting over the possible values of the second argument. The SMT-LIB rotation operators are not supported yet; those operators are over-approximated as uninterpreted functions, which means that it might be possible to prove problems involving the operators unsatisfiable, but not satisfiable.

All experiments were done 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 to 2GB. We used Princess version 2019-10-02 for all experiments. Where runtimes are reported, we use wall clock time.

We evaluate the performance of our approach in three different ways:

- Section 6.1: performance of satisfiability queries on quantifier-free bit-vector formulas (SMT-LIB QF_BV), in comparison to the state-of-the-art solvers Z3 4.8.0 [37] and CVC4 1.6 [38].
- Section 6.2: performance of satisfiability queries on bit-vector formulas with quantifiers (SMT-LIB BV), again with comparison to Z3 and CVC4.
- Section 6.3: applicability of the interpolation procedure for software model checking, using the integration in the Horn solver Eldarica. We compare to the software model checker CPAchecker 1.7 [39], which internally uses MathSAT 5 [20] and the interpolation method from [16].

6.1 Satisfiability Queries on Quantifier-Free Formulas

While our procedure is not specifically designed for just checking satisfiability of formulas, it is nevertheless interesting to evaluate how the approach performs on problems from the QF_BV category of the SMT-LIB. Results for this category are given in Table 1, and show that our implementation can overall solve a decent number of benchmarks, but is not competitive with Z3 and CVC4 on most of the benchmark families. As a general trend, and unsurprisingly, it can be observed that our lazy arithmetic encoding works relatively well for problems that use arithmetic bit-vector operators, but does not pay off for problems that are mostly Boolean, or problems involving bit-wise operators.

Our implementation can solve the "challenge/multiplyOverflow.smt2" problem discussed in Example 3, and it performs particularly well on the "brummayer-biere4" and "pspace" families, which contain benchmarks with large bit-widths, with variables with up to 30 000 bits. This is to be expected, since our arithmetic encoding is essentially agnostic about the bounds of bit-vector variables, so that the complexity of a problem hardly changes when adding more bits.

The family "bruttomesso/core" consists of SMT-LIB problems in the structural fragment from Section 5 (with additional Boolean structure). Compared to the implementation described in the FMCAD 2018 paper [17], the performance on those problems has improved significantly as a result of adding the procedure described in Section 5. In 2018, only 8 benchmarks from the "core" family could be solved, compared to 142 in the new version. This is still lower than the results for Z3 and CVC4, which is likely due to more efficient Boolean reasoning.

We also investigated how often the rules $\times$-split, bmod-split, and bmod-const for splitting and eliminating predicates were applied on the benchmarks. We first determined how many of the problems required the application of the rules at all:

Table 1 Performance on SMT-LIB QF_BV Problems. For each row, the first/second value gives sat/unsat problems. Experiments were done with Princess 2019-10-02, default settings.

Family	Princess		Z3		CVC4	
	Solved	Time (s)	Solved	Time (s)	Solved	Time (s)
2.-BuchwaldFried			0/1	-/292	0/1	-/0.7
2.-Hansen-Check	1/2	0.9/0.7	1/2	0.0/0.0	1/2	0.0/0.0
asp	1/3	9.8/14.5	202/68	86.5/67.6	104/22	130.5/82.4
RWS			16/0	16.5/-	16/0	34.0/-
VS3			2/0	217/-		
bench_ab	284/0	2.2/-	285/0	0.0/-	285/0	0.0/-
bmc-bv	10/8	5.3/45.0	15/15	2.2/13.6	15/12	7.5/35.8
bmc-bv-svcomp14	0/17	-/65.5	8/56	1.5/6.0	8/54	67.1/47.2
brummayerbiere	0/6	-/40.6	0/40	-/30.7	0/37	-/37.6
brummayerbiere2	0/1	-/181	4/25	46.1/90.4	6/43	61.3/54.6
brummayerbiere3	1/0	574/-	5/37	112/97.5	6/12	0.7/85.9
brummayerbiere4	9/0	32.4/-	10/0	0.0/-		
bruttomesso						
.../simple_processor	0/1	-/38.1	0/64	-/53.1	0/64	-/1.2
.../core	0/142	-/56.3	0/672	-/1.2	0/672	-/16.1
.../lfsr			0/225	-/69.9	0/240	-/16.8
calypto	1/2	276/4.7	4/7	0.5/8.6	4/10	13.2/4.3
challenge	0/1	-/1.4				
check2	3/2	1.2/0.9	3/3	0.0/0.0	3/3	0.0/0.0
crafted	2/18	1.0/2.6	2/19	0.0/0.0	2/19	0.0/0.0
dwp_formulas	136/175	12.3/6.4	154/178	0.0/0.0	154/178	0.0/0.0
ecc			0/8	-/0.1	0/8	-/1.9
fft			5/4	40.8/212	1/0	5.7/-
float	0/1	-/270	59/53	116/89.6	38/28	75.6/131
galois			0/1	-/0.2	0/1	-/0.4
gulwani-pldi08			6/0	14.8/-	6/0	35.2/-
log-slicing			0/51	-/287	0/17	-/352
mcm			36/16	122/232	14/0	40.4/-
pspace	21/42	1.4/1.3	0/13	-/366	21/42	0.0/0.0
rubik			3/4	40.1/21.5	0/2	-/73.1
sage	6601/14658	8.2/23.6	8077/18530	0.3/0.1	8077/18530	0.3/1.7
spear						
.../openldap_v2.3.35			3/0	0.1/-	5/0	138/-
.../samba_v3.0.24	34/0	46.6/-	1373/13	8.8/2.4	1343/13	19.9/16
.../zebra_v0.95a	9/0	16.3/-	9/0	1.9/-	9/0	4.0/-
.../xinetd_v2.3.14	0/2	-/2.4	0/2	-/1.4	0/2	-/1.6
.../cvs_v1.11.22	0/5	-/5.3	24/5	4.9/7.4	24/5	14.4/4.5
.../wget_v1.10.2	5/0	8.1/-	38/4	60.7/3.7	36/4	74.8/9.1
.../inn_v2.4.3	163/0	14.2/-	219/0	13.7/-	204/0	29.1/-
stp			1/0	22.4/-	1/0	288/-
stp_samples	48/35	114/27.9	151/273	0.0/0.0	151/273	0.3/0.5
tacas07	2/0	6.9/-	3/0	1.0/-	3/2	22.1/186
uclid/catchconv	0/16	-/35.5	262/152	2.2/0.9	262/152	7.79/9.6
uclid/tcas	0/2	-/1.4	0/2	-/0.0	0/2	-/0.1
uclid_c._smtcomp09			0/6	-/218	0/6	-/383
uum			0/2	-/5.6	0/1	-/2.2
wienand-cav2008						
.../Booth			0/2	-/12.0	0/2	-/16.6
.../Distrib	0/6	-/2.9	0/6	-/0.0	0/6	-/0.0
.../Commute	0/3	-/4.0	0/6	-/0.0	0/6	-/0.0
Total						
SAT	7331	9.14	10980	4.69	10799	5.53
UNSAT	15148	23.7	20565	3.19	20471	3.42

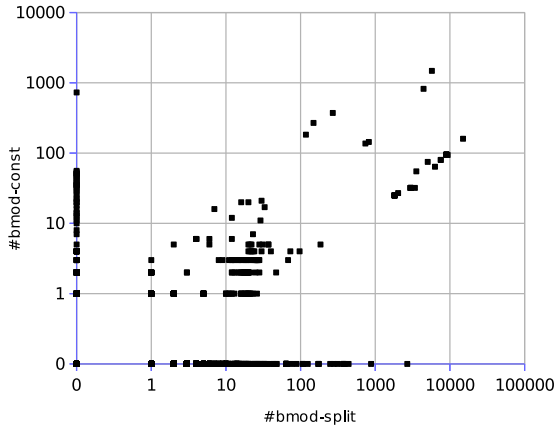

Fig. 11 Scatter plot comparing the number of applications of the rules BMOD-SPLIT and BMOD-CONST on QF_BV benchmarks.

	Total solved	BMOD-SPLIT	BMOD-CONST	×-SPLIT
SAT	7331	2699	268	334
UNSAT	15148	647	130	44

The statistics show that a large number of the benchmarks can indeed be solved without those rules. This is in particular the case for unsatisfiable problems, for which it is apparently largely sufficient to work with the simplification rules from Figure 4, in combination with the rules for Presburger arithmetic, (non-splitting) multiplication, and extraction. In Figure 11 we compare the required number of applications of BMOD-SPLIT and BMOD-CONST for the individual benchmarks; the scatter plot shows that often a small number of rule applications is sufficient.

The runtimes reported in Table 1 for PRINCESS are somewhat higher than those of Z3 and CVC4, which can partly be explained by the fact that PRINCESS is entirely implemented in Scala, and runs on a JVM. This results in repeated overhead for starting up the JVM and for just-in-time compilation. In actual applications, for instance software model checking as discussed in Section 6.3, normally many queries are handled without restarts in between, and the amortised overhead is smaller.

6.2 Satisfiability Queries on Formulas with Quantifiers

We evaluate the effectiveness of our quantifier elimination approach on problems from the BV category of SMT-LIB. In order to check whether a quantified bit-vector formula is satisfiable, QE often does not have to be run to completion, instead the elimination approach from Section 2.2 can be stopped as soon as a statement about satisfiability of the resulting formula can be made. This incremental approach to solving quantified formulas has been implemented in PRINCESS for Presburger arithmetic, and in combination with our lazy encoding for bit-vectors also directly applies to quantified bit-vector formulas.

Results on the SMT-LIB BV benchmarks are given in Table 2. Our procedure can solve a similar number of problems as Z3 and CVC4 on many of the BV

Table 2 Performance on SMT-LIB BV problems. For each family, the first/second row gives sat/unsat problems. Several of the families contains benchmarks with unknown status; for those families only the total number of benchmarks is specified. Experiments were done with Princess 2019-10-02 and option `-portfolio=bv`.

Category	Problems	Princess		Z3		CVC4	
		Solved	Time (s)	Solved	Time (s)	Solved	Time (s)
Automizer	16	16	5.7	16	0.1	14	0.1
	137	137	7.5	137	0.0	137	0.3
keymaera	4035	3	5.6	108	6.9	34	1.0
		3754	1.2	3923	0.3	3921	0.1
psyco	132	4	24.0	132	0.1	132	1.5
	62	3	90.8	62	0.2	62	0.5
tptp	17	16	1.2	17	0.0	17	0.0
	56	54	1.3	56	0.0	56	0.0
RND	100	5	5.2	40	6.9	24	39.5
		2	2.3	28	6.7	22	13.2
RNDPRE	130	13	30.6	20	19.0	22	26.9
		15	47.4	36	14.1	26	29.3
model	144	138	8.3	144	0.0	73	10.8
	0	0		0		0	
Heizmann	131	17	32.9	15	37.8	18	18.1
		45	37.7	17	50.7	108	8.3
ranking	60	9	8.4	34	4.4	32	1.5
		4	10.2	19	19.5	13	0.4
fixpoint	131	34	16.0	36	0.5	54	14.2
		30	55.8	73	0.6	75	2.3
Total	5151	255	11.6	562	3.9	420	8.8
		4044	2.5	4351	0.7	4420	0.6

families, although the total number of problems solved is still lower than for Z3 and CVC4. Like for QF_BV, the results confirm that the encoding of bit-vectors into arithmetic is more effective for problems that are arithmetic in nature (e.g., the families "Automizer," "model," and "Heizmann"), than for more combinatorial problems (e.g., "psyco"). In general, quantified bit-vector problems tend to be smaller and harder than quantifier-free problems, which leads to a situation where it is essential to have the right heuristics and optimisations in place; in this respect our implementation is clearly still lagging behind Z3 and CVC4.

In the experiments, we used a simple portfolio mode enabled by the option `-portfolio=bv`. This mode is inspired by the observation that a closed bit-vector formula ϕ (i.e., a formula without free variables or uninterpreted predicates) can be shown to be satisfiable also be proving that the negation $\neg\phi$ is unsatisfiable, and vice versa. Experiments showed that often one of ϕ or $\neg\phi$ is significantly simpler to solve than the other, but that it is difficult to predict the easier one; in the portfolio mode the prover therefore simultaneously tries to solve ϕ and $\neg\phi$.

6.3 Interpolation and Verification of C Programs

The main purpose of our procedure is the computation of Craig interpolants for bit-vector formulas. Unfortunately, comparing and evaluating interpolation procedures is relatively tricky, since the properties that can be measured easily (e.g., the size, shape, or strength of interpolants, or the time required to extract interpolants) are ultimately only of limited importance in applications. The decisive property that

Table 3 Comparison of Eldarica configurations *math* and *ilp32*. For each category, the table shows the number of safe/unsafe results, and for the solved cases the average time, the required number of CEGAR iterations, and the average size of computed interpolants. For *, after removing an outlier the number is 1.1, and ** becomes 1.3.

Category	Programs	Eldarica *math*				Eldarica *ilp32*			
		Solved	Time (s)	Iter.	I. Size	Solved	Time (s)	Iter.	I. Size
HOLA	46	43	5.3	4.1	1.1	21	7.1	4.8	2.0
		0				5	17.3	41.0	1.3
llreve	21	16	9.8	13.0	1.3	8	13.4	14.8	2.0
		5	7.6	6.8	1.0	5	120.6	8.6	427.5*
VeriMAP	155	133	4.4	2.4	1.0	100	3.8	1.5	1.2
		21	6.7	4.2	1.0	39	14.2	7.4	1.4
SVCOMP	329	107	53.7	22.3	1.0	89	65.7	21.6	1.4
		78	93.5	41.0	1.0	70	66.7	30.7	1.2
Total	551	299	22.5	10.3	1.1	218	29.8	10.5	1.5
		104	71.8	31.9	1.0	119	49.7	21.2	23.7**

Table 4 Comparison of Eldarica configuration *ilp32* and CPAchecker. For each category, the table shows the number of safe/unsafe results, and for the solved cases the average time, and the required average number of CEGAR iterations.

Category	Programs	Eldarica *ilp32*			CPAchecker -32		
		Solved	Time (s)	Iter.	Solved	Time (s)	Iter.
HOLA	46	21	7.1	4.8	12	80.8	20.1
		5	17.3	41.0	4	8.5	7.0
llreve	21	8	13.4	14.8	7	19.5	14.7
		5	120.6	8.6	5	33.8	7.0
VeriMAP	155	100	3.8	1.5	87	9.4	3.3
		39	14.2	7.4	33	20.9	6.1
SVCOMP	329	89	65.7	21.6	74	37.5	12.1
		70	66.7	30.7	126	49.0	9.0
Total	551	218	29.8	10.5	180	26.1	8.5
		119	49.7	21.2	168	42.1	8.3

makes interpolants useful is the ability to generalise, which is hard to measure syntactically. Adding to this, there is no standard set of interpolation benchmarks that could be used, and the interpolation queries that occur in model checking can differ from run to run. In model checking, moreover interpolation queries are interdependent: the results of earlier queries in a run will affect the later interpolation queries being generated.

We therefore decided to evaluate in an application-oriented way, by integrating our interpolation procedure into a model checker and measuring its ability to verify safety properties of C programs with machine integer semantics. As model checker we use Eldarica version 2.0.2[7] [40], a Horn clause-based model checker that uses Cartesian predicate abstraction and the CEGAR algorithm. Eldarica was already previously tightly integrated with Princess, and was in the scope of this work extended to also handle Horn clauses over bit-vectors. Since Eldarica internally uses the Princess data-structures to store Horn clauses, we could implement the translation from bit-vectors to our core language (Section 4.1) as a preprocessing step that is applied to all Horn clauses upfront. This means that the actual model checking engine operates purely on expressions in the core language,

[7] `https://github.com/uuverifiers/eldarica`

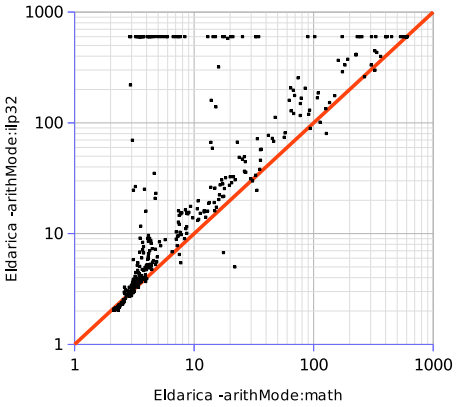

Fig. 12 Comparison of the ELDARICA runtime in seconds for *math* and *ilp32* semantics.

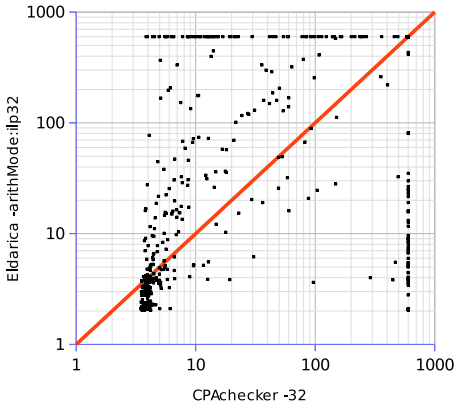

Fig. 13 ELDARICA vs. CPACHECKER runtime in seconds for *ilp32* semantics.

and all interpolation queries and implication checks stay within the core language; the need to translate back and forth between bit-vector formulas and core language is eliminated. As an obvious downside of this approach, however, it is no longer easily possible to replace the interpolation procedure with other solvers.

Benchmarks. For the experiments, we used the built-in C parser of ELDARICA, and work with the benchmark set of 551 C programs already used in [41] for evaluating different predicate generation strategies. The programs stem from a variety of sources, including the SV-COMP 2016 categories "Integers and Control Flow" and "Loops," and were selecting by taking all programs that do not include arrays or heap data structures (i.e., only arithmetic operations). The verification task consisted in showing that safety assertions included in the programs can never fail. For the experiment, we interpret the programs as operating either on the unbounded mathematical integers (*math*, ELDARICA option `-arithMode:math`), or on signed 32-bit bit-vectors (*ilp32*, ELDARICA option `-arithMode:ilp32`) with wrap-around semantics. ELDARICA was otherwise run with default settings, which means that it also applies the interpolation abstraction technique from [42].

Comparison math vs. ilp32. The results for *math* and *ilp32* semantics are given in Table 3 and Figure 12. It has to be pointed out that the status of the programs depends on the chosen semantics: for instance, the 46 HOLA programs [43] are all known to be safe in mathematical semantics, but several of the programs turn out to be unsafe in bit-vector semantics due to the possibility of overflow. ELDARICA can consistently verify safety of more programs in *math* than in *ilp32*, but it can disprove safety in more of the *ilp32* cases. The total number of solved cases is higher in *math* than in *ilp32*, but *ilp32* is quite close (403 vs. 337); given the higher complexity of the bit-vector semantics, this is an encouraging result. The scatter plot in Figure 12 shows that the runtimes for the two semantics are strongly correlated, and while *ilp32* is on average slower than *math* the difference is relatively small.

Table 3 also shows that the number of CEGAR iterations is comparable for *math* and *ilp32*, while the size of interpolants (measured as the average number of

sub-formulas of interpolants) is bigger for *ilp32* than for *math*, but usually by less than a factor of 2. The exception is the category "llreve/unsafe," where drastically bigger interpolants are computed for *ilp32* than for *math*. Inspecting this case, we found that there was a single benchmark in "llreve" that was solved after 579 seconds with interpolants of size 2133; when removing this outlier, the average interpolant size for "llreve/unsafe" is only 1.1.

Comparison with CPAchecker. As comparison, we also ran the model checker CPAchecker 1.7 [39], using options `-predicateAnalysis -32` and MathSAT 5 [20] as solver. MathSAT 5 uses the interpolation method from [16]. The results are given in Table 4 and Figure 13. Our method is competitive with CPAchecker on all considered categories: Eldarica with *ilp32* can consistently prove more programs safe, whereas CPAchecker can show more programs unsafe, with a lower number of CEGAR iterations. We suspect that the use of large-block encoding [44] in CPAchecker is responsible for this phenomenon, and indeed makes CPAchecker very effective for bug finding. The runtimes of the systems are on average close, but the scatter plot in Figure 13 shows no clear trend.

Altogether, we remark that we are comparing different verification systems here: although both Eldarica and CPAchecker apply CEGAR and interpolation, there are many factors affecting the results. What the experiments do show, however, is that the interpolation method proposed in this paper can be used to create a software model checker that is competitive with the state of the art.

7 Conclusions

We have presented a new calculus for Craig interpolation and quantifier elimination in bit-vector arithmetic. Furthermore, we have shown how to efficiently integrate reasoning over the structural fragment. While the experimental results in model checking are already promising, we believe that there is still a lot of room for extension and 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 more efficient use of bounds, and a strategy to employ bit-blasting directly to whole sub-expressions when deemed more efficient.

References

1. K. L. McMillan, An interpolating theorem prover, Theor. Comput. Sci. 345 (1) (2005).
2. V. D'Silva, M. Purandare, G. Weissenbacher, D. Kroening, Interpolant strength, in: VMCAI, LNCS, Springer, 2010.
3. A. Fuchs, A. Goel, J. Grundy, S. Krstić, C. Tinelli, Ground interpolation for the theory of equality, in: TACAS, LNCS, Springer, 2009.
4. A. Brillout, D. Kroening, P. Rümmer, T. Wahl, Beyond quantifier-free interpolation in extensions of Presburger arithmetic, in: VMCAI, LNCS, Springer, 2011, pp. 88–102.
5. K. L. McMillan, Quantified invariant generation using an interpolating saturation prover, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Vol. 4963 of Lecture Notes in Computer Science, Springer, 2008, pp. 413–427.
6. L. Kovács, A. Voronkov, Interpolation and symbol elimination, in: CADE, 2009, pp. 199–213.

7. M. P. Bonacina, M. Johansson, On interpolation in automated theorem proving, J. Autom. Reasoning 54 (1) (2015) 69–97. doi:10.1007/s10817-014-9314-0.
8. D. Kapur, R. Majumdar, C. G. Zarba, Interpolation for data structures, in: SIGSOFT'06/FSE-14, ACM, New York, NY, USA, 2006, pp. 105–116. doi:http://doi.acm.org/10.1145/1181775.1181789.
9. H. Hojjat, P. Rümmer, Deciding and interpolating algebraic data types by reduction, in: T. Jebelean, V. Negru, D. Petcu, D. Zaharie, T. Ida, S. M. Watt (Eds.), 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017, IEEE Computer Society, 2017, pp. 145–152. doi:10.1109/SYNASC.2017.00033.
10. L. Dai, B. Xia, N. Zhan, Generating non-linear interpolants by semidefinite programming, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Vol. 8044 of Lecture Notes in Computer Science, Springer, 2013, pp. 364–380. doi:10.1007/978-3-642-39799-8_25.
11. A. Brillout, D. Kroening, P. Rümmer, T. Wahl, An interpolating sequent calculus for quantifier-free Presburger arithmetic, Journal of Automated Reasoning 47 (2011) 341–367.
12. A. Griggio, T. T. H. Le, R. Sebastiani, Efficient interpolant generation in satisfiability modulo linear integer arithmetic, Logical Methods in Computer Science 8 (3) (2010). doi:10.2168/LMCS-8(3:3)2012.
13. R. Bruttomesso, S. Ghilardi, S. Ranise, Quantifier-free interpolation of a theory of arrays, Logical Methods in Computer Science 8 (2) (2012). doi:10.2168/LMCS-8(2:4)2012.
14. N. Totla, T. Wies, Complete instantiation-based interpolation, J. Autom. Reasoning 57 (1) (2016) 37–65. doi:10.1007/s10817-016-9371-7.
15. J. Hoenicke, T. Schindler, Efficient interpolation for the theory of arrays, CoRR abs/1804.07173 (2018). arXiv:1804.07173.
16. A. Griggio, Effective word-level interpolation for software verification, in: P. Bjesse, A. Slobodová (Eds.), International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, FMCAD Inc., 2011, pp. 28–36.
17. P. Backeman, P. Rümmer, A. Zeljic, Bit-vector interpolation and quantifier elimination by lazy reduction, in: Bjørner and Gurfinkel [45], pp. 1–10. doi:10.23919/FMCAD.2018.8603023.
18. D. Cyrluk, O. Möller, H. Rueß, An efficient decision procedure for the theory of fixed-sized bit-vectors, in: O. Grumberg (Ed.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 60–71.
19. R. Bruttomesso, N. Sharygina, A scalable decision procedure for fixed-width bit-vectors, in: Proceedings of the 2009 International Conference on Computer-Aided Design, ICCAD '09, ACM, New York, NY, USA, 2009, pp. 13–20. doi:10.1145/1687399.1687403.
URL http://doi.acm.org.ezproxy.its.uu.se/10.1145/1687399.1687403
20. A. Cimatti, A. Griggio, B. J. Schaafsma, R. Sebastiani, The MathSAT5 SMT solver, in: TACAS, Vol. 7795 of LNCS, 2013.
21. S. Asadi, M. Blicha, G. Fedyukovich, A. E. J. Hyvärinen, K. Even-Mendoza, N. Sharygina, H. Chockler, Function summarization modulo theories, in: G. Barthe, G. Sutcliffe, M. Veanes (Eds.), LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018, Vol. 57 of EPiC Series in Computing, EasyChair, 2018, pp. 56–75.
22. D. Kroening, G. Weissenbacher, Lifting propositional interpolants to the word-level, in: FMCAD, IEEE Computer Society, 2007, pp. 85–89.
23. D. Kroening, G. Weissenbacher, An interpolating decision procedure for transitive relations with uninterpreted functions, in: Haifa Verification Conference, Vol. 6405 of Lecture Notes in Computer Science, Springer, 2009, pp. 150–168.
24. Y. Ho, P. Chauhan, P. Roy, A. Mishchenko, R. K. Brayton, Efficient uninterpreted function abstraction and refinement for word-level model checking, in: R. Piskac, M. Talupur (Eds.), 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016, IEEE, 2016, pp. 65–72. doi:10.1109/FMCAD.2016.7886662.
25. P. Rümmer, A constraint sequent calculus for first-order logic with linear integer arithmetic, in: LPAR, Vol. 5330 of LNCS, Springer, 2008, pp. 274–289.
26. M. C. Fitting, First-Order Logic and Automated Theorem Proving, 2nd Edition, Springer-Verlag, New York, 1996.

27. J. Y. Halpern, Presburger arithmetic with unary predicates is Π_1^1 complete, Journal of Symbolic Logic 56 (1991).
28. R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T), Journal of the ACM 53 (6) (2006) 937–977.
29. A. Reynolds, T. King, V. Kuncak, Solving quantified linear arithmetic by counterexample-guided instantiation, Formal Methods in System Design 51 (3) (2017) 500–532.
30. W. Craig, Linear reasoning. A new form of the Herbrand-Gentzen theorem, The Journal of Symbolic Logic 22 (3) (1957) 250–268.
31. S. Lang, Algebra (3. ed.), Addison-Wesley, 1993.
32. B. Buchberger, An algorithm for finding the basis elements in the residue class ring modulo a zero dimensional polynomial ideal, Ph.D. thesis (3 2006).
33. P. Van Hentenryck, D. McAllester, D. Kapur, Solving polynomial systems using a branch and prune approach, SIAM J. Numer. Anal. 34 (2) (1997) 797–827.
34. J. Warren A. Hunt, R. B. Krug, 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, Vol. 2860 of LNCS, Springer, 2003, pp. 319–333.
35. C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio, SAT modulo linear arithmetic for solving polynomial constraints, J. Autom. Reasoning 48 (1) (2012) 107–131.
36. C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Tech. rep., Department of Computer Science, The University of Iowa, available at `www.SMT-LIB.org` (2017).
37. L. M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: TACAS, Vol. 4963 of LNCS, Springer, 2008, pp. 337–340.
38. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli, CVC4, in: CAV, Vol. 6806 of LNCS, Springer, 2011, pp. 171–177. `doi:10.1007/978-3-642-22110-1_14`.
39. D. Beyer, M. E. Keremoglu, CPAchecker: A tool for configurable software verification, CoRR abs/0902.0019 (2009). `arXiv:0902.0019`.
40. H. Hojjat, P. Rümmer, The ELDARICA Horn solver, in: Bjørner and Gurfinkel [45], pp. 1–7. `doi:10.23919/FMCAD.2018.8603013`.
41. Y. Demyanova, P. Rümmer, F. Zuleger, Systematic predicate abstraction using variable roles, in: C. Barrett, M. Davies, T. Kahsai (Eds.), NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings, Vol. 10227 of Lecture Notes in Computer Science, 2017, pp. 265–281. `doi:10.1007/978-3-319-57288-8_18`.
42. J. Leroux, P. Rümmer, P. Subotic, Guiding Craig interpolation with domain-specific abstractions, Acta Inf. 53 (4) (2016) 387–424. `doi:10.1007/s00236-015-0236-z`.
43. I. Dillig, T. Dillig, B. Li, K. L. McMillan, Inductive invariant generation via abductive inference, in: A. L. Hosking, P. T. Eugster, C. V. Lopes (Eds.), Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA, ACM, 2013, pp. 443–456. `doi:10.1145/2509136.2509511`.
44. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, R. Sebastiani, Software model checking via large-block encoding, in: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA, IEEE, 2009, pp. 25–32. `doi:10.1109/FMCAD.2009.5351147`.
45. N. Bjørner, A. Gurfinkel (Eds.), 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE, 2018.