Effective Word-Level Interpolation for Software Verification

Alberto Griggio [*]
Embedded Systems Unit – FBK-IRST – Trento, Italy.

Abstract—**We present an interpolation procedure for the theory of fixed-size bit-vectors, which allows to apply effective interpolation-based techniques for software verification without giving up the ability of handling precisely the word-level operations of typical programming languages. Our algorithm is based on advanced SMT techniques, and, although general, is optimized to exploit the structure of typical interpolation problems arising in software verification. We have implemented a prototype version of it within the MATHSAT SMT solver, and we have integrated it into a software verification framework based on standard predicate abstraction. Our experimental results show that our new technique allows our prototype to significantly outperform other systems on programs requiring bit-precise modeling of word-level operations.**

I. Introduction and Related Work

Since the seminal paper of McMillan [1], (Craig) interpolation has been recognized to be a substantial tool for formal verification. In particular, one of its most successful applications is in the context of software verification based on counterexample-guided abstraction-refinement (CEGAR), where interpolants of quantifier-free formulas in suitable theories are computed for automatically refining abstractions in order to rule out spurious counterexamples [2], [3].

Most programming languages use a fixed amount of bits for representing values of primitive data types, such as integers. However, most interpolation-based software verification tools represent primitive types using mathematical integers or rational numbers, encoding program operations into e.g. a combination of linear arithmetic and uninterpreted functions. This results in loss of precision, which might not only lead to the generation of false alarms, in which correct programs are classified as incorrect, but also, and worse, to failures in detecting bugs. As a simple example, the code fragment `if (x > 0 && y > 0) { assert(x + y > 0); }` is wrongly classified as safe if variables are modeled using unbounded integers.

One of the main reasons for not using a more accurate modeling of program operations is the lack of effective interpolation procedures for the theory of bit-vectors (BV), that allow bit-precise representation of operations while retaining the advantages of reasoning at the word-level structure of problems. Although a significant amount of work has been done on interpolation procedures for several important theories (including theories of equality, linear arithmetic, data structures) [4], [5], [6], [7], [8], [9], [10], [11], interpolation for bit-vectors has received very little attention so far. To the best of our knowledge, the only complete interpolation algorithm for BV is based on naïvely mapping a bit-level propositional interpolant into BV (by replacing propositional variables with bit-extraction terms and Boolean connectives with bit-wise operations), which however completely destroys the word-level structure of the original problem, thus defeating all the benefits of reasoning at a level of abstraction higher than that of single bits. The first partial solution for this problem was proposed in [12], where an algorithm is given for constructing a word-level interpolant from a bit-level proof of unsatisfiability. The approach, however, was limited to equality logic only. A different direction is explored in [13], where a rewrite-based procedure for a fragment of BV is presented. The procedure is incomplete in general, but the authors show that their specialised rewrite rules are often enough for successfully verifying programs, in particular in the domain of device drivers.

In this paper, we present a novel, complete interpolation procedure for BV which tries to retain as much as possible the word-level structure of the input problem. Our approach is based on lazy/DPLL(T) SMT techniques for interpolation [7], and generates interpolants from DPLL(T)-proofs of unsatisfiability by combining a layered hierarchy of different interpolation procedures for conjunctions of BV-constraints, of increasing power and complexity, with a standard Boolean interpolation algorithm. Although general, the BV-interpolation layers are optimized for the kind of formulas arising in software verification, exploiting "definitional" equalities and interpolation procedures for linear integer arithmetic, and falling back to a bit-level algorithm when none of the specialised techniques can be applied.

We have implemented a prototype version of our procedure within the MATHSAT [14] SMT solver, and we have integrated it into a software verification framework based on standard predicate abstraction and interpolation-based refinement. Our prototype significantly outperforms other systems on programs requiring a bit-precise modeling of word-level operations, which could not be verified when using linear arithmetic and uninterpreted functions instead of BV.

Paper Outline. We introduce some background concepts in Sec. II. In Sec. III we describe a simple bit-level interpolation algorithm for BV. Our new procedure is described and

[*] Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

discussed in Sec. IV, and experimentally evaluated in Sec. V. We conclude in Sec. VI with directions for future work.

II. Background

A. Terminology and Notation

We work in the setting of standard first-order logic. We denote formulas with φ, ψ, A, B, I, variables with x, y, z, terms with s, t, predicates with p, q, possibly adding sub- and/or superscripts. As usual in the SAT and SMT community, we call 0-arity predicates *Boolean variables*. In the following, we only deal with quantifier-free formulas, in which all variables are implicitly existentially quantified. We use the standard definitions of theory, model, satisfiability, validity. If ψ is a logical consequence of φ in a theory T, we write $\varphi \models_T \psi$. If φ is unsatisfiable in T, we write $\varphi \models_T \perp$. A (fixed-width) bit-vector, or word, is a list of bits of fixed size n. We denote bit-vector terms of size n with $t_{[n]}$. We use the definition of the theory of bit-vectors, BV, given e.g. in [15]. BV-operators include sub-word selection $t_{[n]}[i:j]$ (where $0 \leq j \leq i \leq n$), concatenation $t_{1[n]} :: t_{2[m]}$, arithmetic operations (addition $+$, subtraction $-$, multiplication $\cdot$, signed and unsigned division $/_s$ and $/_u$, and signed and unsigned remainder $\%_s$ and $\%_u$), shifts ($<<$ and $>>$), and bitwise operations (bitwise *not* $\sim$, bitwise *and* $\&$, *or* $|$ and *xor* ^). BV-predicates include equality $=$ and signed and unsigned inequalities $\leq_s$ and $\leq_u$.

B. Interpolation for Software Verification

Given an ordered pair of formulas (A, B) in a theory T, a (Craig) interpolant is a formula I that satisfies the following constraints: (i) $A \models_T I$; (ii) $B \wedge I \models_T \perp$; and (iii) all the uninterpreted (in T) symbols occurring in I occur in both A and B (i.e., they are *AB-common*). Interpolants have important applications in software verification, and in particular in software model checking based on counterexample-guided abstraction-refinement (CEGAR) [16]. When using predicate abstraction, predicates for automatic abstraction refinement can be extracted from interpolants generated from formulas representing (sets of) spurious counterexamples (i.e. program paths leading to an error location which are feasible in the abstract space but infeasible in the concrete program) [2]. Similarly, interpolants from spurious counterexamples can also be used for directly representing and refining program abstractions without the need of computing predicate abstractions [3]. Both techniques proved to be quite effective, and are now implemented within many model checking tools (e.g. [17], [13], [18], [19], [20]).

C. Bit-Vectors in SMT

For many important theories T, the currently most-popular approach for checking the satisfiability of a formula φ in T, SMT(T), is the so-called "lazy" or "DPLL(T)" approach [21], in which a DPLL-based SAT solver is used for enumerating truth-assignments for the propositional skeleton of φ, which are then checked for T-satisfiability by a decision procedure for conjunctions of constraints in T (T-solver).

However, bit-vectors are an exception to this trend. Although several different algorithms have been proposed in recent years (see e.g. [15] for a survey), most current state-of-the-art SMT(BV)-solvers (e.g. [22], [23], [24], [25]) are based on (i) preprocessing the input BV-formula by applying several word-level simplification techniques followed by (ii) eagerly encoding the result of such preprocessing into a purely-propositional formula ("bit-blasting"), which is then given to an efficient SAT solver. In a nutshell, bit-blasting consists of encoding each bit-vector $t_{[n]}$ using n Boolean variables $p^t_0, \ldots, p^t_{n-1}$ representing its bits, and then translating each BV operation into an equivalent Boolean circuit, possibly introducing fresh auxiliary Boolean variables.

III. Simple Interpolation for BV

From the purely theoretical point of view, computing interpolants in the theory of bit-vectors is an easy problem. It is solved by a conceptually-simple algorithm, based on bit-blasting, which exploits the availability of off-the-shelf (and efficient) algorithms for interpolation for propositional logic (e.g. [1]).

Given a pair of BV-formulas A and B, an interpolant I for (A, B) can be generated from a propositional interpolant as follows.

- First, A and B are converted via bit-blasting into two purely-Boolean formulas A^p and B^p. For interpolation, it is important that A and B are bit-blasted using disjoint sets of auxiliary variables (see Sec. II-C).
- A^p and B^p are then converted to CNF[1] and given to an interpolating SAT solver, which checks the satisfiability of $A^p \wedge B^p$ and computes a propositional interpolant I^p for (A^p, B^p).
- By construction, I^p contains only variables that occur in both A^p and B^p, and so it can not contain auxiliary Boolean variables introduced by bit-blasting or CNF conversion. Each variable p^t_j in I^p then corresponds to a single bit j of a bit-vector term $t_{[n]}$ that occurs in both A and B. An interpolant I for (A, B) can therefore be obtained from I^p by replacing each variable p^t_j with the bit-extraction $t_{[n]}[j:j]$, and each Boolean connective with its corresponding bitwise operator (i.e. $\neg$ with $\sim$, $\wedge$ with $\&$ and $\vee$ with $|$).

This procedure is simple both to define and to implement. It has, however, an obvious and major drawback: it *completely destroys the word-level structure of the problem*, since it only generates interpolants as Boolean combinations of individual bits. Clearly, this completely defeats the benefits of reasoning at a higher level of abstraction, for instance by making it very difficult to apply effective word-level simplification techniques which are crucial for the efficiency of current state-of-the-art SMT solvers for BV [22], [26], [25], or to extract useful high-level information which can be effectively exploited in software verification, like "good" word-level predicates for abstraction refinement.

[1]CNF conversion might introduce more auxiliary "label" variables. As before, it is important that the sets of label variables for A^p and B^p are disjoint.

IV. A Layered Approach to BV Interpolation

The above reasons make the simple bit-level interpolation procedure of the previous section not very appealing in practice. Rather, we would like to obtain an interpolation procedure that retains as much as possible the word-level structure of formulas. At the same time, we would also like to keep the performance benefits of bit-blasting, which is still the dominant technique for SMT(BV), adopted by the most efficient state-of-the art solvers ([22], [26], [23], [24], [25]).

In the rest of the section, we present our solution to this problem. Its main idea is that of reducing the problem of interpolant generation for BV-formulas with an arbitrary Boolean structure to that of computing interpolants for *conjunctions of BV-constraints*, which can then be interpolated using a *layered approach*, by applying a hierarchy of different techniques which try to retain as much as possible the word-level structure of the input problem.

This reduction to dealing only with conjunctions of constraints is standard in interpolation for SMT when using the lazy/DPLL(T) approach, in which interpolants can be extracted from proofs of unsatisfiability consisting of a Boolean skeleton, to which a propositional interpolation algorithm is applied, and a set of T-inconsistent conjunctions of constraints, corresponding to negations of the T-lemmas occurring in the proof, which are handled by T-specific interpolation procedures [4], [7]. However, SMT solvers based on bit-blasting typically follow the eager approach, for which such reduction is harder to achieve, and to the best of our knowledge has been done only for equality logic [12]. Here, we exploit the combination of bit-blasting and DPLL(T), which allows us to generate proofs of unsatisfiability which can be easily partitioned into Boolean and T-specific parts while still retaining as much as possible the performance advantage of SAT encodings for solving BV-formulas. After having generated such proofs, we then tackle interpolation for the conjunctions of BV-constraints corresponding to the negated BV-lemmas in the proof using a layered hierarchy of four different techniques of increasing power and complexity.

A. *Lazy bit-blasting in DPLL(T)*

Lazy bit-blasting is a simple technique for integrating a decision procedure for BV based on SAT encoding within an SMT solver based on the DPLL(T) approach. It is the default strategy used by the MathSAT SMT solver [14], a state-of-the-art solver for BV.[2]

The main idea of lazy bit-blasting is that of using two (DPLL-based) independent SAT solvers, $\text{DPLL}_{\text{Bool}}$ and DPLL_{BV}, organized in a hierarchy. $\text{DPLL}_{\text{Bool}}$ corresponds to the "DPLL" part of the standard DPLL(T) approach, whereas DPLL_{BV} takes the role of the T-solver. More precisely, when solving a BV-formula φ, $\text{DPLL}_{\text{Bool}}$ is used to reason on the Boolean skeleton of (the CNF conversion of) φ, like in the usual DPLL(T) approach, whereas DPLL_{BV} is used for checking the consistency of truth-assignments of BV-atoms enumerated by $\text{DPLL}_{\text{Bool}}$. This is done by exploiting the capability of modern SAT solvers of *reasoning under assumptions* [27], [28]. DPLL_{BV} is initialized by adding to it, for each BV-atom a occurring in φ, the clauses resulting from the bit-blasting of the formula $(l_a \leftrightarrow a)$, where l_a is a fresh Boolean variable, which we call the *label* for a. Notice that this means that the set of clauses in DPLL_{BV} is always satisfiable. When DPLL_{BV} is asked to check the consistency of a set of BV-literals $L_1, \ldots, L_n$ generated by $\text{DPLL}_{\text{Bool}}$, the corresponding labels $l_1, \ldots, l_n$ are added as temporary assumptions to DPLL_{BV} (if L_i is a negative literal, $\neg l_i$ is added as assumption instead of l_i). If the resulting formula becomes unsatisfiable, then it is possible to compute the (typically small) subset of assumptions $l_j, \ldots, l_k$ (some of which possibly negated) which is responsible for the inconsistency (see e.g. [28]). From this set, a BV-conflict set $L_j, \ldots, L_k$ is computed, whose negation $\neg L_j \vee \ldots \vee \neg L_k$ is a BV-lemma that is given back to $\text{DPLL}_{\text{Bool}}$ as usual in DPLL(T).

B. *BV Interpolation via EUF layering*

A good "side-effect" of using lazy bit-blasting is that it enables the use of *layering* of theory solvers. In particular, since BV-constraints are not bit-blasted at the main DPLL level, truth assignments can be checked using a solver for equality and uninterpreted functions (EUF) before invoking DPLL_{BV}. In this way, "cheap" conflicts that are due to the violations of equality axioms can be handled efficiently, without resorting to the potentially-expensive SAT checks in DPLL_{BV}. In such cases, interpolants can be computed by efficient existing algorithms for EUF, starting from the proofs of unsatisfiability generated by the EUF solver [4], [9]. This is therefore the first layer of our procedure.

Example 1: Consider the BV-interpolation problem:

$$\begin{aligned} A &\overset{\text{def}}{=} (x_{1[32]} = 3_{[32]}) \wedge (x_{3[32]} = x_{1[32]} \cdot x_{2[32]}) \\ B &\overset{\text{def}}{=} (x_{4[32]} = x_{2[32]}) \wedge (x_{5[32]} = 3_{[32]} \cdot x_{4[32]}) \wedge \\ &\quad \neg(x_{3[32]} = x_{5[32]}). \end{aligned}$$

In order to detect the unsatisfiability of $A \wedge B$, it is not necessary to take the precise semantics of the BV multiplication operation $\cdot$. In fact, a solver for EUF, which treats $\cdot$ as an uninterpreted function, is enough to construct a proof of unsatisfiability for $A \wedge B$. From such proof, the BV-interpolant $I \overset{\text{def}}{=} (x_{3[32]} = 3_{[32]} \cdot x_{2[32]})$ can be computed using an efficient algorithm for EUF interpolation. ◇

C. *BV Interpolation via Equality Substitution*

Equalities can still be exploited even when EUF is not enough for detecting unsatisfiability. As an example, consider an interpolation problem for an inconsistent pair (A, B) of formulas in which A is of the form $(x = e) \wedge \varphi$, x does not occur in e and x is the only non-common symbol between A and B. Then, it is easy to see that the formula obtained by replacing x with e everywhere in φ (denoted $\varphi[x \mapsto e]$) is an interpolant

[2]The latest version MathSAT was the winner of the 2011 SMT competition on the "BV+uninterpreted functions" (QF_UFBV) and the "incremental BV" (QF_BV application) categories, and performed better than the winner of 2010 on the "plain BV" (QF_BV) category (see http://smtcomp.org/2011/).

for (A, B). Similarly, if it is B to be of the form $(x = e) \wedge \psi$, then $\neg\psi[x \mapsto e]$ is also an interpolant for (A, B). These two examples are just special cases of the well-known general algorithm for computing interpolants via quantifier elimination (for theories for which this is possible): roughly speaking, given an inconsistent pair (A, B) of formulas, an interpolant can be computed by performing the existential elimination of all the non-common variables either from A or from $\neg B$.[3] In general, however, existential quantification for BV can be quite expensive, it may require bit-blasting (and a consequent loss of word-level structure), and it may cause a blow-up in the size of the formula.

The idea of our second technique is that of detecting situations in which interpolation via existential elimination amounts to *performing substitutions using equalities*.

More in detail, given an inconsistent conjunction of BV-constraints partitioned into A and B, we remove from A a positive equality $(x = e)$ in which x is a variable that does not occur in e and x is not AB-common. We then replace x with e in the rest of the constraints of A, and repeat the process until either all the non-AB-common variables have been eliminated, or a fixpoint is reached. If the result A' of this procedure contains no non-AB-common variable, then we can return A' as an interpolant. Otherwise, we try eliminating non-common variables from B, obtaining B', and if this operation succeeds, we return $\neg B'$ as an interpolant. Notice that this procedure requires no satisfiability checks, and is therefore very cheap.

Example 2: Consider the BV-interpolation problem:

$$A \overset{\text{def}}{=} (0_{[32]} \leq_s (0_{[24]} :: x_{1[8]}) - 1_{[32]}) \wedge (x_{2[8]} = x_{1[8]})$$
$$B \overset{\text{def}}{=} (x_{3[8]} = (-(0_{[24]} :: x_{2[8]}))[7:0]) \wedge (x_{3[8]} = 0_{[8]}).$$

The unsatisfiability of $A \wedge B$ cannot be determined with the EUF layer alone. However, using equality substitution, we can easily compute an interpolant for (A, B). The only non-AB-common symbol in A is the variable $x_{1[8]}$, which can be eliminated by exploiting the equality $(x_{2[8]} = x_{1[8]})$, thus generating the BV-interpolant $I \overset{\text{def}}{=} (0_{[32]} \leq_s (0_{[24]} :: x_{2[8]}) - 1_{[32]})$. $\diamond$

D. BV Interpolation via LIA Encoding

In the third layer of our procedure, we try to reduce the problem of generating interpolants for BV to the computation of interpolants in linear arithmetic over the integers (LIA).

In principle, the idea is similar to the reduction to propositional logic described in Sec. III: given an unsatisfiable conjunction of BV-constraints partitioned into A and B, the algorithm consists of: (i) generating two LIA-formulas A_{LIA} and B_{LIA} using the encoding described in [29], (ii) building an interpolant I_{LIA} for $(A_{\text{LIA}}, B_{\text{LIA}})$ using any off-the-shelf efficient interpolation algorithm for LIA (e.g. [10], [30], [11]), and finally (iii) "translating back" I_{LIA} in order to obtain a BV-interpolant I for (A, B). In practice, however, reduction to LIA presents several difficulties that do not occur in the case of reduction to propositional logic:

– First, from the theoretical point of view the problem of obtaining a BV-interpolant I from a LIA-interpolant I_{LIA} is non-trivial. In particular, in the translation of arithmetic operations and predicates from LIA to BV, issues like overflow or signed/unsigned semantics should be properly taken into account. (We shall give examples of some of the problems that may arise later in this section, after having given some details of the encoding of BV into LIA.)

– Moreover, from the practical point of view, encoding of BV constraints into LIA might result in *very challenging* SMT(LIA)-formulas, which might be out of reach of current state-of-the-art SMT(LIA)-solvers, even for BV-problems that current SMT(BV)-solvers can easily handle. This might happen especially when encoding BV-operations that require a "mixed LIA/bit-blasting" approach, like e.g. multiplication of two variables [29].

We address the above two issues by taking an *optimistic approach*. First, in the encoding of BV constraints into LIA, we abstract away all the operations that require a mixed LIA/bit-blasting approach, in order to reduce the likelihood of generating difficult SMT(LIA)-formulas, by simply encoding them with integer variables. Further, we set bounds (dependent on the size of the input problem) to the resources available (time and memory) for solving and constructing the proof of unsatisfiability of the SMT(LIA)-formula $A_{\text{LIA}} \wedge B_{\text{LIA}}$ resulting from the encoding of the BV-interpolation problem (A, B). If the formula $A_{\text{LIA}} \wedge B_{\text{LIA}}$ turns out to be satisfiable (because of the abstraction) or its unsatisfiability can not be determined within the resource bounds, we resort to the last layer of our procedure, described in Sec. IV-E. Otherwise, we compute an interpolant I_{LIA} for $(A_{\text{LIA}}, B_{\text{LIA}})$, and we translate it back to a BV-formula I using an *optimistic naïve approach* that essentially disregards overflow and signed/unsigned issues. We then check whether I is actually an interpolant for (A, B), by testing whether the BV-formula $(A \wedge \neg I) \vee (B \wedge I)$ is unsatisfiable,[4] again using bounded resources. If the check succeeds, then we return I as an interpolant. Otherwise, we resort to the last layer of our procedure.

In the rest of this section, we provide some details of the encoding and the construction of a BV-interpolant from a LIA-interpolant, giving also examples of why this might fail.

Encoding of BV constraints into LIA. We use the LIA encoding of BV constraints described, e.g., in [29]. Each BV term $t_{[n]}$ of n bits is encoded as a LIA variable x_t, together with the constraint

$$(0 \leq x_t) \wedge (x_t \leq 2^n - 1). \qquad (1)$$

(In what follows, we shall denote (1) as $x_t \in [0, 2^n)$.) Each BV-operation/predicate is encoded as a Boolean combination of LIA constraints, possibly introducing some auxiliary LIA

[3]It can be observed that this algorithm always generates the strongest or the weakest interpolant (wrt. logical implication) for (A, B), the former when starting from A, the latter when starting from $\neg B$.

[4]As described later in this section, I contains only AB-common symbols by construction.

Selection: $t_{[i-j+1]} \overset{\text{def}}{=} t_{1[n]}[i:j]$ becomes $(x_t = m) \wedge (x_{t_1} = 2^{i+1}h + 2^j m + l) \wedge l \in [0, 2^i) \wedge m \in [0, 2^{i-j+1}) \wedge h \in [0, 2^{n-i-1})$, where h, m, l are fresh.
Concatenation: $t_{[n+m]} \overset{\text{def}}{=} t_{1[n]} :: t_{2[m]}$ becomes $(x_t = 2^m x_{t_1} + x_{t_2})$.
Addition: $t_{[n]} \overset{\text{def}}{=} t_{1[n]} + t_{2[n]}$ becomes $(x_t = x_{t_1} + x_{t_2} - 2^n\sigma) \wedge (0 \leq \sigma) \wedge (\sigma \leq 1)$, where σ is fresh.
Multiplication by constant: $t_{[n]} \overset{\text{def}}{=} t_{1[n]} \cdot k$ becomes $(x_t = k \cdot x_{t_1} - 2^n\sigma) \wedge (0 \leq \sigma) \wedge (\sigma \leq k)$, where σ is fresh.
Left-shift by constant: $t_{[n]} \overset{\text{def}}{=} t_{1[n]} << k$ becomes $(x_t = 2^k x_{t_1[n-k-1:0]})$, where $x_{t_1[n-k-1:0]}$ is the LIA variable for the selection $t_1[n-k-1:0]$

Equality: $(t_{1[n]} = t_{2[n]})$ becomes $(x_{t_1} = x_{t_2})$.
Unsigned $\leq$: $(t_{1[n]} \leq_u t_{2[n]})$ becomes $(x_{t_1} \leq x_{t_2})$.
Signed $\leq$: $(t_{1[n]} \leq_s t_{2[n]})$ becomes $ITE(((x_{t_1} \leq 2^{n-1}-1) \wedge (x_{t_2} \leq 2^{n-1}-1)) \vee ((x_{t_1} \geq 2^{n-1}) \wedge (x_{t_2} \geq 2^{n-1})), (x_{t_1} \leq x_{t_2}), (x_{t_1} \geq 2^{n-1}))$, where $ITE(c,t,e)$ is a shorthand for $(c \rightarrow t) \wedge (\neg c \rightarrow e)$.

Fig. 1. LIA encoding of some BV operations and predicates.

and Boolean variables. Some examples are shown in Fig. 1.[5] As already mentioned before, for performance reasons we abstract BV-terms t requiring a mixed LIA/bit-blasting encoding [29] (i.e. non-linear multiplication/division/remainder, bit-wise operations between two non-constant terms, and shift by a non-constant term) by simply using the corresponding LIA-variable x_t, without additional constraints (other than (1)). As in the bit-blasting approach (see Sec. III), we assume that when generating an interpolant for a pair of formulas (A, B), A and B are encoded using disjoint sets of auxiliary variables.

Constructing a BV-interpolant from a LIA-interpolant. Current interpolation algorithms for LIA allow to produce interpolants in an extension of LIA with either divisibility predicates [10] or with the floor function $\lfloor \cdot \rfloor$ [11].Once a LIA-interpolant I_{LIA} (in either of the two extended signatures) for the encoded pair of formulas $(A_{\text{LIA}}, B_{\text{LIA}})$ has been generated, we translate it back to a *candidate BV-interpolant* I for the original pair (A, B), by replacing LIA-variables with the corresponding BV-variables, LIA-numbers with their BV-encoding, and LIA-operations with the corresponding BV-operations. More formally, we proceed as follows.

1) Let β be a (partial) mapping from LIA-terms/predicates to BV-terms/predicates. β is initialized by setting, for every LIA-variable x_t occurring in I_{LIA}, $\beta(x_t)$ to the corresponding BV-term t. Notice that, similarly to the case of interpolation via bit-blasting, I_{LIA} contains no auxiliary variables, since A and B were encoded using disjoint sets of such variables.
2) Integer constants k occurring in I_{LIA} are mapped to BV constants using a 2's complement representation. The size of the target BV-constant $\beta(k)$ is determined by examining the context in which k occurs. In particular, if k is the argument of a binary LIA-term or predicate $t \bowtie k$, we encode k with a bit-vector of the same width n as $\beta(t)$, [6] simply truncating if k does not fit in n bits.
3) LIA-additions and multiplications are mapped to BV-additions and multiplications respectively, without considering potential overflow issues. (If the bit-width of $\beta(t_1)$ and $\beta(t_2)$ are different, we extend the shortest of the operands by padding it with zeros).
4) For floor terms $\lfloor \frac{t}{k} \rfloor$, where k is a positive constant,[7] $\beta(\lfloor \frac{t}{k} \rfloor)$ is set to $\beta(t) /_u \beta(k)$.
5) LIA-equalities $(t_1 = t_2)$ and inequalities $(t_1 \leq t_2)$ are mapped to BV-equalities $(\beta(t_1) = \beta(t_2))$ and unsigned inequalities $(\beta(t_1) \leq_u \beta(t_2))$ respectively.
6) For divisibility predicates $k|t$, where k is a positive constant, $\beta(k|t)$ is set to the BV-equality $(\beta(t) \%_u \beta(k) = 0_n)$, where n is the bit-width of $\beta(t)$.
7) We construct I from I_{LIA} by replacing each LIA-atom a occurring in I_{LIA} with $\beta(a)$.

Example 3: Consider the BV-interpolation problem:

$$A \overset{\text{def}}{=} (y_{1[8]} = y_{5[4]} :: y_{5[4]}) \wedge (y_{1[8]} = y_{2[8]}) \wedge (y_{5[4]} = 1_{[4]})$$
$$B \overset{\text{def}}{=} \neg(y_{4[8]} + 1_{[8]} \leq_u y_{2[8]}) \wedge (y_{4[8]} = 1_{[8]}).$$

Encoding A and B into LIA (see Fig. 1) results in the following:

$$\begin{aligned} A_{\text{LIA}} \overset{\text{def}}{=} &(x_{y_2} = 16x_{y_5} + x_{y_5}) \wedge (x_{y_1} = x_{y_2}) \wedge (x_{y_5} = 1) \wedge \\ &(x_{y_1} \in [0, 2^8)) \wedge (x_{y_2} \in [0, 2^8)) \wedge (x_{y_5} \in [0, 2^4)) \\ B_{\text{LIA}} \overset{\text{def}}{=} &\neg(x_{y_4+1} \leq x_{y_2}) \wedge (x_{y_4+1} = x_{y_4} + 1 - 2^8\sigma) \wedge \\ &(x_{y_4} = 1) \wedge \\ &(x_{y_4+1} \in [0, 2^8)) \wedge (x_{y_4} \in [0, 2^8)) \wedge (0 \leq \sigma \leq 1) \end{aligned}$$

$A_{\text{LIA}} \wedge B_{\text{LIA}}$ is LIA-inconsistent, and an interpolant for $(A_{\text{LIA}}, B_{\text{LIA}})$ is $I_{\text{LIA}} \overset{\text{def}}{=} (17 \leq x_{y_2})$. Using β, we obtain the formula $I \overset{\text{def}}{=} (17_{[8]} \leq_u y_{2[8]})$, which is a BV-interpolant for (A, B). $\diamond$

Notice that, since we encoded A and B using disjoint sets of auxiliary variables, a formula I generated via β from a LIA-interpolant I_{LIA} is guaranteed to fulfill the third condition of the definition of interpolant. However, I is *not guaranteed* to be an interpolant for (A, B), as shown by the following examples.

Example 4: Consider the BV-interpolation problem:

$$\begin{aligned} A \overset{\text{def}}{=} &(y_{2[8]} = 81_{[8]}) \wedge (y_{3[8]} = 0_{[8]}) \wedge (y_{4[8]} = y_{2[8]}) \\ B \overset{\text{def}}{=} &(y_{13[16]} = 0_{[8]} :: y_{4[8]}) \wedge \\ &(255_{[16]} \leq_u y_{13[16]} + (0_{[8]} :: y_{3[8]})) \end{aligned}$$

[5] Several optimizations are possible, but they are not discussed here. We refer to [29] for the full details.

[6] Notice that we can always assume w.l.o.g. that I_{LIA} is normalized such that all integer constants occur as argument of binary terms/predicates in which the other argument is not a constant.

[7] Notice that the interpolation procedure of [11] always produces floor terms of this form.

and its LIA-encoding:

$$\begin{aligned}A_{\text{LIA}} &\stackrel{\text{def}}{=} (x_{y_2}=81)\wedge(x_{y_3}=0)\wedge(x_{y_4}=x_{y_2})\wedge\\ &\quad (x_{y_2}\in[0,2^8))\wedge(x_{y_3}\in[0,2^8))\wedge(x_{y_4}\in[0,2^8))\\ B_{\text{LIA}} &\stackrel{\text{def}}{=} (x_{y_{13}}=2^8\cdot 0+x_{y_4})\wedge(255\le x_{y_{13+(0::y_3)}})\wedge\\ &\quad (x_{y_{13+(0::y_3)}}=x_{y_{13}}+2^8\cdot 0+x_{y_3}-2^{16}\sigma)\wedge\\ &\quad (x_{y_{13}}\in[0,2^{16}))\wedge(x_{y_{13+(0::y_3)}}\in[0,2^{16}))\wedge\\ &\quad (0\le\sigma\le 1).\end{aligned}$$

A LIA-interpolant for $(A_{\text{LIA}}, B_{\text{LIA}})$ is $I_{\text{LIA}} \stackrel{\text{def}}{=} (x_{y_3}+x_{y_4}\le 81)$. However, the formula $I \stackrel{\text{def}}{=} \beta(I_{\text{LIA}}) \stackrel{\text{def}}{=} (y_{3[8]}+y_{4[8]}\le_u 81_{[8]})$ *is not* an interpolant for (A,B), because $I\wedge B\not\models_{\text{BV}}\bot$. The problem is that in BV, addition might overflow. In fact, if we make sure this does not happen in I, then we obtain a correct interpolant I' for (A,B):

$$I' \stackrel{\text{def}}{=} ((0_{[1]}::y_{3[8]})+(0_{[1]}::y_{4[8]})\le_u 81_{[9]}).$$

$\diamond$

The above example shows that, in the translation from I_{LIA} to I, overflows are an issue. However, they are not the only problem that might arise.

Example 5: Consider the BV-interpolation problem:

$$\begin{aligned}A &\stackrel{\text{def}}{=} \neg(y_{4[8]}+1_{[8]}\le_u y_{3[8]})\wedge(y_{2[8]}=y_{4[8]}+1_{[8]})\\ B &\stackrel{\text{def}}{=} (y_{2[8]}+1_{[8]}\le_u y_{3[8]})\wedge(y_{7[8]}=3_{[8]})\wedge\\ &\quad (y_{7[8]}=y_{2[8]}+1_{[8]})\end{aligned}$$

and its LIA-encoding:

$$\begin{aligned}A_{\text{LIA}} &\stackrel{\text{def}}{=} \neg(x_{y_4+1}\le x_{y_3})\wedge(x_{y_2}=x_{y_4+1})\wedge\\ &\quad (x_{y_4+1}=x_{y_4}+1-2^8\sigma_1)\wedge\\ &\quad (x_{y_2}\in[0,2^8))\wedge(x_{y_3}\in[0,2^8))\wedge(x_{y_4}\in[0,2^8))\wedge\\ &\quad (x_{y_4+1}\in[0,2^8))\wedge(0\le\sigma_1\le 1)\\ B_{\text{LIA}} &\stackrel{\text{def}}{=} (x_{y_2+1}=x_{y_3})\wedge(x_{y_7}=3)\wedge(x_{y_7}=x_{y_2+1})\wedge\\ &\quad (x_{y_2+1}=x_{y_2}+1-2^8\sigma_2)\wedge\\ &\quad (x_{y_7}\in[0,2^8))\wedge(x_{y_2+1}\in[0,2^8))\wedge(0\le\sigma_2\le 1).\end{aligned}$$

A LIA-interpolant for $(A_{\text{LIA}}, B_{\text{LIA}})$, computed with the algorithm of [11], is $I_{\text{LIA}} \stackrel{\text{def}}{=} (-255\le x_{y_2}-x_{y_3}+256\lfloor -1\frac{x_{y_2}}{256}\rfloor)$. Applying β to it, we obtain $I \stackrel{\text{def}}{=} \beta(I_{\text{LIA}}) \stackrel{\text{def}}{=} (1_{[8]}\le_u y_{2[8]}-y_{3[8]}+0_{[8]}\cdot(255_{[8]}\cdot y_{2[8]}/_u 0_{[8]})$, because of the truncations that occur when converting the LIA-constants -255 and 256 into 8-bit BV-constants. I is not an interpolant for (A,B), since both $A\not\models_{\text{BV}} I$ and $B\wedge I\not\models_{\text{BV}}\bot$. However, in this case avoiding overflows (e.g. by using 16-bit words) is not enough: the formula $I' \stackrel{\text{def}}{=} (65281_{[16]}\le_u (0_{[8]}::y_{2[8]})-(0_{[8]}::y_{3[8]})+256_{[16]}\cdot(65535_{[16]}\cdot(0_{[8]}::y_{2[8]})/_u 256_{[16]}))$ is still not an interpolant for (A,B), since $A\not\models_{\text{BV}} I'$. In this case, we could fix the problem by using a signed inequality predicate instead of the unsigned one in I': the formula

$$\begin{aligned}I'' \stackrel{\text{def}}{=} (&65281_{[16]}\le_s (0_{[8]}::y_{2[8]})-(0_{[8]}::y_{3[8]})+\\ &256_{[16]}\cdot(65535_{[16]}\cdot(0_{[8]}::y_{2[8]})/_u 256_{[16]}))\end{aligned}$$

is a correct BV-interpolant for (A,B). $\diamond$

Using signed inequality, however, does not always work.

Example 6: Consider again the interpolation problem of Example 4 and the interpolant $I' \stackrel{\text{def}}{=} ((0_{[1]}::y_{3[8]})+(0_{[1]}::y_{4[8]})\le_u 81_{[9]})$ for (A,B). If we replace $\le_u$ with $\le_s$ in I', the resulting formula is not an interpolant for (A,B) anymore. $\diamond$

As mentioned before, currently we address the potential failures in the translation from I_{LIA} to I by explicitly checking whether I is a correct interpolant for (A,B), which amounts to checking whether the BV-formula $(A\wedge\neg I)\vee(B\wedge I)$ is unsatisfiable. If the test fails, we simply discard I and resort to the last layer of our procedure. The investigation of more effective ways of extracting BV-interpolants from LIA-interpolants is part of ongoing and future work.

E. When Everything Else Fails

When none of the above techniques can be successfully applied to the current conjunction of BV-constraints, we resort to the bit-level interpolation procedure described in Sec. III. This makes our algorithm trivially complete. Clearly however, the effectiveness of our procedure crucially depends on how often this last layer is needed in practice. We discuss this topic in the following section.

F. Discussion

In the worst case, our procedure does not behave much differently from the simple bit-level algorithm of Sec. III. Furthermore, our algorithm is also typically more expensive to apply, since it might result in several extra calls to an SMT solver (for both LIA and BV) for each of the (negations of the) theory lemmas occurring in the DPLL(T)-proof of unsatisfiability, whereas the bit-level algorithm requires only one call to an eager proof-producing SMT(BV)-solver. In fact, it is not too difficult to craft some SMT(BV)-formulas for which our technique will always need to resort to the bit-level interpolation layer. On the other hand, for formulas for which this last layer is not needed, our procedure has the clear advantage of producing interpolants which preserve the word-level structure of BV-constraints, rather than flattening everything down to the bit level. Generally, this advantage will show up in all the cases in which the bit-level layer is needed only for a small fraction of the (negations of the) theory lemmas occurring in the proof of unsatisfiability.

We argue that for interpolation problems arising in software verification, the good cases are much more likely to occur than the bad cases, for the following reasons.

First, as already observed by other authors (e.g. [31], [13]), in important domains in which interpolation-based software verification has been successfully applied (e.g. device drivers), programs typically do not contain complex arithmetic expressions. In such cases, our experiments have shown that the LIA-based interpolation procedure of Sec. IV-D typically produces correct BV-interpolants in practice.

The second reason is that in software verification, interpolation is applied to formulas representing unrollings of the

TABLE I
PERFORMANCE RESULTS ON C PROGRAMS REQUIRING BIT-PRECISION

	KRATOS					SATABS	WOLVERINE
Program	**BV-1**	**BV-2**	**BV-3**	**BV-4**	**BV-5**		
byte_add_1.c	31.00	T.O.	M.O.	57.30	31.54	T.O.	T.O.
byte_add_2.c	47.98	T.O.	M.O.	72.17	44.42	T.O.	T.O.
num_conversion_1.c	1.85	3.20	3.67	2.67	1.13	23.78	2.16
num_conversion_2.c	48.04	776.53	72.12	763.16	47.73	T.O.	T.O.
gcd_1.c	1.75	20.45	20.56	1.05	1.27	FAIL	515.31
gcd_2.c	29.21	M.O.	M.O.	39.21	28.21	339.86	185.56
gcd_3.c	70.05	T.O.	M.O.	209.34	70.59	T.O.	290.03
gcd_4.c	3.58	M.O.	T.O.	T.O.	4.25	T.O.	1.26
interleave_bits.c	45.90	T.O.	T.O.	T.O.	49.01	836.78	T.O.
modulus.c	4.87	34.00	M.O.	3.30	4.15	T.O.	M.O.
parity.c	387.56	M.O.	M.O.	T.O.	391.84	T.O.	T.O.
soft_float_1.c.cil.c	48.02	T.O.	T.O.	T.O.	T.O.	T.O.	136.88
soft_float_2.c.cil.c	61.34	T.O.	T.O.	70.02	T.O.	1101.54	177.63
soft_float_3.c.cil.c	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
soft_float_4.c.cil.c	51.67	T.O.	M.O.	247.31	49.88	T.O.	T.O.
soft_float_5.c.cil.c	61.70	T.O.	T.O.	78.54	T.O.	T.O.	193.76
s3_clnt_1.BV.c.cil.c	41.06	50.82	T.O.	48.77	42.32	FAIL	T.O.
s3_clnt_2.BV.c.cil.c	20.96	9.92	116.03	8.59	22.01	T.O.	T.O.
s3_clnt_3.BV.c.cil.c	7.66	T.O.	93.77	T.O.	6.68	T.O.	T.O.
s3_srvr_1.BV.c.cil.c	11.59	35.91	240.77	34.74	11.63	160.74	T.O.
s3_srvr_2.BV.c.cil.c	150.64	62.22	116.54	61.26	152.10	342.11	T.O.
s3_srvr_3.BV.c.cil.c	48.35	124.32	43.63	125.19	48.36	405.48	T.O.
jain_1.c	0.34	0.39	0.30	0.12	0.36	FAIL	T.O.
jain_2.c	0.43	0.48	0.35	0.21	0.44	FAIL	T.O.
jain_4.c	0.55	0.60	0.40	0.33	0.54	FAIL	T.O.
jain_5.c	T.O.	T.O.	T.O.	T.O.	T.O.	FAIL	T.O.
jain_6.c	0.18	0.12	0.09	0.15	0.16	FAIL	T.O.
jain_7.c	0.29	0.23	0.15	0.26	0.27	FAIL	T.O.
TOTAL (solved / time)	**26 / 1176.57**	**14 / 1119.19**	**13 / 708.38**	**21 / 1823.69**	**23 / 1008.89**	**7 / 3210.29**	**8 / 1500.43**

Execution times are in seconds. T.O. indicates timeouts (using a cutoff value of 1200 seconds), M.O. memory outs (3GBytes), FAIL other kinds of errors (e.g. failure in computing interpolants or in refining the abstraction). All the programs are safe.

control-flow graph of programs, represented using a Static Single Assignment (SSA) form. Such formulas make heavy use of "definitional" equalities, i.e. equalities of the form $(x = t)$ in which x does not occur in t. For example, all equalities representing an assignment statement in SSA form are of this kind. Such equalities are exactly the kind of constraints that are exploited by our substitution-based technique of Sec. IV-C.

Finally, as regards performance, we remark that the BV-interpolation layers are invoked only on the (negations of the) BV-lemmas occurring in the final DPLL(T)-proof of unsatisfiability for the input problem $A \wedge B$. At solving time, only the lazy bit-blasting procedure of Sec. IV-A is used, whose efficiency is typically comparable to that of eager encodings into SAT. In general, only a fraction of all the BV-lemmas discovered during search occur in the final proof of unsatisfiability. Moreover, such lemmas typically involve only a subset of the constraints occurring in the formula. In fact, although not guaranteed to be minimal, they contain very often almost no redundant constraints, and are thus usually much easier to solve than the whole input problem.

V. EXPERIMENTAL EVALUATION

We have implemented the procedure described in the previous section within the MATHSAT SMT solver, and we have integrated it within KRATOS [17], a software model checker implementing a CEGAR-based lazy predicate abstraction algorithm with interpolation-based refinement in the style of [2] (but using the "large-block" encoding introduced in [32] for better exploiting the underlying SMT solver). In this section, we experimentally evaluate the effectiveness and efficiency of our technique in the context of software model checking.

All the experiments have been run on a Linux machine with a 2.2GHz Intel Xeon CPU, using a memory limit of 3GB. All the data and executables needed for reproducing the experiments are available at http://es.fbk.eu/people/griggio/papers/fmcad11_bv_interpolation.tar.bz2.

A. Effectiveness

In order to evaluate the effectiveness of our technique, we have collected a set of C programs whose verification requires the use of a bit-precise modeling of operations. These programs can not be proved safe by KRATOS in its default configuration, since by default it models program variables using rational numbers, and program operations using linear rational arithmetic (LRA) and uninterpreted functions. In particular, we use the following benchmark sets:

– ***byte_add*** **and** ***num_conversion*** implement arithmetic operations using shifts and bit-wise operations;
– ***gcd*** check simple assertions on Euclid's algorithm for computing the greatest common divisor;
– ***interleave_bits*****,** ***modulus*** **and** ***parity*** check the correctness of some "bit twiddling hacks" described at http://www-graphics.stanford.edu/~seander/bithacks.html;

– ***soft_float*** check simple assertions on the software floating-point implementation used in the Picosat [33] SAT solver;
– ***s3_clnt*** **and** ***s3_srvr*** are modified versions of some SSH programs used in several papers on software model checking, in which some bit-wise and non-linear operations have been introduced;
– ***jain*** are the simple programs used in [8].

We compare KRATOS using BV-interpolation against the only other two software model checkers supporting BV (to the best of our knowledge): SATABS, which implements CEGAR-based predicate abstraction but uses weakest preconditions for refinement [34], and WOLVERINE [35], which implements the interpolation-based lazy abstraction of [3], using an incomplete rewrite-based procedure for BV-interpolation [13]. [8] For KRATOS, we use not only the configuration in which all the layers described in Sec. IV are active (called "BV-1"), but also configurations in which some of the layers have been disabled or rearranged: "BV-2" does not use equality substitution, "BV-3" uses only the bit-level algorithm, "BV-4" tries LIA encoding before equality substitution, and "BV-5" does not use LIA encoding. The results of our experiments are reported in Table I. They show that not only KRATOS outperforms the other systems, but also that all the layers of our procedure contribute to the performance of KRATOS, since the default configuration "BV-1" is the clear winner. In particular, our full procedure can solve twice as many instances as the naïve configuration "BV-3" which uses only the bit-level algorithm of Sec. IV-E. Using "BV-1", the final bit-level layer is needed only for four of the programs, and always for less than 1% of the BV-interpolation problems, and in many cases the equality substitution layer alone is enough.

B. Efficiency

In order to evaluate the efficiency of our technique, we compare KRATOS-BV with the default version of KRATOS using linear rational arithmetic and uninterpreted functions, on programs that can be verified without the need of a bit-precise modeling of program variables and operations. We use common benchmarks for software model checking with predicate abstraction, used e.g. in [18]. The results are reported in Table II.[9] They show that, when the additional precision given by using bit-vectors instead of rationals is not needed, our procedure introduces very little overhead: KRATOS-BV-1 can solve only one instance less than KRATOS-LRA, but in fact there are cases in which KRATOS-BV-1 is one order of magnitude faster than KRATOS-LRA.[10] It is somewhat surprising to observe that, for these programs, even the naïve configuration "BV-3" which uses only the bit-level interpolation layer is not dramatically inferior to KRATOS-LRA.

[8]We used the latest versions of SATABS and WOLVERINE, i.e. version 2.6 and 0.5 respectively. For SATABS, we used Cadence SMV as underlying model checker.

[9]We omit the results for "BV-4" and "BV-5", as they are very similar to those for "BV-2" and "BV-1" respectively.

[10]Such differences are due to the fact that the two versions of KRATOS in general discover different sets of predicates, which lead to the exploration of different abstract search spaces.

TABLE II
PERFORMANCE RESULTS ON C PROGRAMS NOT REQUIRING BIT-PRECISION

	KRATOS configuration			
Program	**LRA**	**BV-1**	**BV-2**	**BV-3**
cdaudio_simpl1.cil.c	37.03	61.79	53.05	59.47
diskperf_simpl1.cil.c	40.14	89.25	52.63	64.55
floppy_simpl3.cil.c	18.37	41.06	28.61	33.55
floppy_simpl4.cil.c	36.75	91.73	47.44	58.97
kbfiltr_simpl1.cil.c	1.37	1.66	1.38	1.90
kbfiltr_simpl2.cil.c	1.68	2.70	2.43	2.94
s3_clnt_1.cil.c	5.59	5.20	65.34	10.62
s3_clnt_2.cil.c	4.71	5.33	20.72	7.33
s3_clnt_3.cil.c	8.52	4.87	14.72	4.86
s3_clnt_4.cil.c	3.20	6.04	29.41	T.O.
s3_srvr_1.cil.c	69.35	7.97	166.88	14.71
s3_srvr_2.cil.c	65.95	224.63	313.30	16.08
s3_srvr_3.cil.c	35.54	8.52	97.51	12.24
s3_srvr_4.cil.c	99.67	185.83	312.21	T.O.
s3_srvr_6.cil.c	90.48	25.60	24.71	163.21
s3_srvr_7.cil.c	218.26	15.10	17.28	40.26
s3_srvr_8.cil.c	72.94	13.83	170.53	23.27
s3_srvr_9.cil.c	5.43	22.92	24.50	53.44
s3_srvr_10.cil.c	9.82	14.68	14.14	255.90
s3_srvr_11.cil.c	36.47	15.49	19.03	156.01
s3_srvr_12.cil.c	19.56	60.78	47.94	328.75
s3_srvr_13.cil.c	289.77	T.O.	82.42	T.O.
s3_srvr_14.cil.c	18.16	22.50	61.08	99.72
s3_srvr_15.cil.c	24.55	27.77	27.28	20.00
s3_srvr_16.cil.c	57.93	12.13	39.05	46.38
TOTAL (solved / time)	**25 / 1271.24**	**24 / 967.38**	**25 / 1733.59**	**22 / 1474.16**

Execution times are in seconds. T.O. indicates timeouts (cutoff time of 600 seconds). All the programs are safe.

VI. CONCLUSIONS AND FUTURE WORK

We have presented an interpolation procedure for bit-vectors based on lazy SMT and on a layer of different techniques optimized for exploiting the structure of typical interpolation problems arising in software verification. We have integrated it in KRATOS, a CEGAR-based software model checker with interpolation-based refinement, and our experiments have shown that the new procedure makes it possible to verify programs requiring a bit-precise modeling of operations which could not be verified by KRATOS before.

For future work, we plan to explore several directions. First, we want to investigate in more depth the problem of computing BV-interpolants by reduction to LIA, in particular to identify smarter ways of generating a correct interpolant in BV from an interpolant for the LIA-encoding of the problem. Second, we plan to experiment with the integration of other layers in our procedure, e.g. based on the application of rewriting rules in the style of [13]. Finally, we would also like to investigate the problem of constructing a word-level interpolant from a bit-level proof of unsatisfiability, by exploring the possibility of extending the work of [12] for equality logic to more general cases.

REFERENCES

[1] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV'03*, ser. LNCS, W. A. Hunt Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[2] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proc. of POPL'04*, N. D. Jones and X. Leroy, Eds. ACM, 2004, pp. 232–244.

[3] K. L. McMillan, "Lazy Abstraction with Interpolants," in *Proc. of CAV'06*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 123–136.

[4] ——, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005.

[5] V. Sofronie-Stokkermans, "Interpolation in Local Theory Extensions," *Logical Methods in Computer Science (Special issue dedicated to IJCAR 2006)*, vol. 4, no. 4, p. Paper 1, 2008.

[6] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *Proc. of FSE'05*, M. Young and P. T. Devanbu, Eds. ACM, 2006, pp. 105–116.

[7] A. Cimatti, A. Griggio, and R. Sebastiani, "Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories," *ACM Trans. Comput. Log.*, vol. 12, no. 1, p. 7, 2010.

[8] H. Jain, E. M. Clarke, and O. Grumberg, "Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations," in *Proc. of CAV'08*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 254–267.

[9] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli, "Ground Interpolation for the Theory of Equality," in *Proc. of TACAS'09*, ser. LNCS, S. Kowalewski and A. Philippou, Eds., vol. 5505. Springer, 2009, pp. 413–427.

[10] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic," in *Proc. IJCAR'10*, ser. LNCS, J. Giesl and R. Hähnle, Eds., vol. 6173. Springer, 2010, pp. 384–399.

[11] A. Griggio, T. T. H. Le, and R. Sebastiani, "Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic," in *Proc. of TACAS'11*, ser. LNCS, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 143–157.

[12] D. Kroening and G. Weissenbacher, "Lifting Propositional Interpolants to the Word-Level," in *Proc. of FMCAD'07*. IEEE Computer Society, 2007, pp. 85–89.

[13] ——, "An Interpolating Decision Procedure for Transitive Relations with Uninterpreted Functions," in *Proc. of HVC'09*, ser. LNCS, K. S. Namjoshi, A. Zeller, and A. Ziv, Eds., vol. 6405. Springer, 2009, pp. 150–168.

[14] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *Proc. of CAV'08*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 299–303.

[15] A. Franzén, "Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT," Ph.D. dissertation, DISI - University of Trento, 2010.

[16] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, pp. 21:1–21:54, October 2009.

[17] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos – a Software Model Checker for SystemC," in *Proc. of CAV'11*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 310–316, to appear.

[18] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate Abstraction with Adjustable-Block Encoding," in *Proc. of FMCAD'10*, R. Bloem and N. Sharygina, Eds., 2010, pp. 189–187.

[19] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim, "SLAB: A Certifying Model Checker for Infinite-State Concurrent Systems," in *Proc. of TACAS'10*, ser. LNCS. Springer, 2010.

[20] N. Caniart, "Merit: An Interpolating Model-Checker," in *Proc. of CAV'10*, ser. LNCS, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 162–166.

[21] R. Sebastiani, "Lazy Satisfiability Modulo Theories," *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, vol. 3, no. 3-4, pp. 141–224, 2007.

[22] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Proc. of TACAS'09*, ser. LNCS, S. Kowalewski and A. Philippou, Eds., vol. 5505. Springer, 2009, pp. 174–177.

[23] S. Jha, R. Limaye, and S. A. Seshia, "Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic," in *Proc. of CAV'09*, ser. LNCS, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 668–674.

[24] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. of TACAS'08*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[25] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Proc. of CAV'07*, ser. LNCS, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.

[26] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev, "Applying SMT in Symbolic Execution of Microcode," in *Proc. of FMCAD'10*, R. Bloem and N. Sharygina, Eds., 2010, pp. 121–128.

[27] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003.

[28] R. J. A. Achá, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Practical algorithms for unsatisfiability proof and core generation in SAT solvers," *AI Commun.*, vol. 23, no. 2-3, pp. 145–157, 2010.

[29] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani, "Encoding RTL Constructs for MathSAT: a Preliminary Report," *Electr. Notes Theor. Comput. Sci.*, vol. 144, no. 2, pp. 3–14, 2006.

[30] D. Kroening, J. Leroux, and P. Rümmer, "Interpolating Quantifier-Free Presburger Arithmetic," in *Proc. of LPAR'10*, ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Springer, 2010, pp. 489–503.

[31] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang, "Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement," in *Proc. of CAV'04*, ser. LNCS, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 457–461.

[32] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Proc. of FMCAD'09*, 2009, pp. 25–32.

[33] A. Biere, "PicoSAT Essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[34] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate Abstraction of ANSI-C Programs Using SAT," *Formal Methods in System Design*, vol. 25, no. 2-3, pp. 105–127, 2004.

[35] D. Kroening and G. Weissenbacher, "Interpolation-Based Software Verification with Wolverine," in *Proc. of CAV'11*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 573–578.