Polyhedra to the rescue of array interpolants

Francesco Alberti, David Monniaux

Polyhedra to the rescue of array interpolants

Francesco Alberti*
University of Lugano

David Monniaux†
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France

December 10, 2014

Abstract

We propose a new approach to the automated verification of the correctness of programs handling arrays. An abstract interpreter supplies auxiliary numeric invariants to an interpolation-based refinement procedure suited to array programs. Experiments show that this combination approach, implemented in an enhanced version of the Booster software model-checker, performs better than the pure interpolation-based approach, at no additional cost.

1 Introduction

Our goal is to automatically prove the correctness of programs handling arrays; that is, to show that they always compute what they are supposed to, as opposed to merely testing them on a limited set of samples. This is a difficult challenge, if only because this problem is undecidable in general: our hope is thus to succeed *in practice*, on industrially-relevant cases.

In order to prove that a program handling arrays (especially those of dynamic length) is safe with respect to a list of assertions, i.e., cannot exhibit executions violating any of them, one has to generate *quantified* properties that are both (i) maintained inductively by the program and (ii) strong enough to entail the validity of the assertions of the program. As an example, consider the following procedure, from [16]:

```
void D10( ) {
  int i = 1; int j = 0;
```

*Supported by Swiss National Science Foundation under grant no. P1TIP2_152261.

†The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR" http://stator.imag.fr/.

```
  while( i < size ) {
    a[j] = b[i]; i = i+2; j = j+1;
  }
  i = 1; j = 0;
  while( i < size ) {
    assert(a[j] == b[2*j+1]); i = i+2; j = j+1;
  }
}
```

To prove the safety of D10 one has to come up with the following invariant[1]

$$\forall x, y. \begin{pmatrix} 0 \leq x \wedge y = 2x + 1 \wedge \\ 2j = i - 1 \wedge 2x + 1 \leq 2j \end{pmatrix} \rightarrow a[x] = b[y] \tag{1}$$

which holds at the head of the loop by induction on the number of loop iterations.

The approach presented here can infer invariants like (1) *completely automatically*, and can successfully verify many programs with arrays that are still out of reach for state-of-the-art software model-checkers (e.g., [17, 19, 14, 7, 11, 16]).

Inferring invariants such as (1) automatically is challenging. An approach aimed at producing such invariants has to address several sub-problems: (i) deducing general inductive properties of the program (like $2j = i - 1$), (ii) establishing the number of quantified variables required to express the program invariant (here, x and y), (iii) finding the relation between the quantified variables and the program variables (e.g., $2x + 1 \leq 2j$), (iv) and establishing useful properties of the array contents by using appropriate variables as indexes.

1.1 Contribution

The contribution of this paper is a new approach for the completely automatic inference of complex quantified invariants for programs operating over arrays. Our approach splits the generation of a quantified invariant in two sub-tasks: (i) generating property-independent inductive invariants (ii) refining them to property-dependent *safe* inductive invariants, i.e., strong enough to prove the safety of the given system. This idea arose from the observation that some atomic formulas of (1), such as $2j = i - 1$, are general inductive facts of the program, *not* depending on the assertions we want to prove; other facts, instead, *do* depend on the property, meaning that they are required specifically to prove the safety of this particular assertion (e.g. quantified invariants are needed because the assertion is a universal property over the array contents). In light of this, we can use an abstract interpreter, working with a suitable abstract domain, e.g., convex polyhedra [15, 8], to infer the "general purpose" inductive facts for the program,

[1] `http://rise4fun.com/Boogie/GG2`.

and then generate the remaining facts using refinement procedures based on Craig interpolation, tailored to the inference of quantified invariants for programs with arrays, e.g., [5].

Our approach combines the strengths of abstract interpretation and interpolation-based refinement. Indeed, interpolation is great at proving array properties, but often fails to establish necessary auxiliary arithmetic constraints, such as $2j = i - 1$, $y = 2x + 1$ and $2x + 1 \leq 2j$. For instance, instead of coming up with $2j = i - 1$, which would help establish the safety of the program after any number of iterations, they may successively come up with $i = 1 \wedge j = 0$, then $i = 3 \wedge j = 1$, then $i = 5 \wedge j = 2 \ldots$, refuting the existence of counterexample traces of increasing length. This causes the enumeration of all possible unwindings of program loops and, by consequence, the divergence of the model-checker (unless the loops have small maximal trip counts). In contrast, abstract interpretation easily establishes such inductive arithmetic facts, but lifting abstract interpretation to array properties is nontrivial [21, 22, 14, 20].

Architecture-wise, our solution pipelines an abstract interpreter into software model checker based on the Lazy Abstraction with Interpolants (LAWI) framework. We will prove (Section 3) that this combination is sound. In addition, in order to be successful, our approach also applies FLATTENING and TERMABSTRACTION phases for better generalization of the interpolants within the LAWI framework. FLATTENING is a preprocessing technique known in software model-checking frameworks for array programs (see, e.g., [5]), used to transform the declarative encoding of a program into a format particularly suitable for inferring quantified invariants. TERMABSTRACTION is a heuristic meant to nudge interpolating procedures towards the inference of "good" interpolants, with the overall effect of improving the model-checker performance. In our setting, FLATTENING plays a central role in connecting the abstract interpreter with the interpolation-based refinement procedure: we will show (Section 4) that interpolating procedures, once enhanced with the TERMABSTRACTION heuristic, can successfully leverage the lemmas inferred by the abstract interpreter thanks to the FLATTENING-based preprocessing.

We implemented our framework on the top of the BOOSTER software model-checker [2]. Experiments discussed in Section 5 show clearly that on the one hand, our solution allows verifying programs that were previously out of the capabilities of this tool and, on the other hand, it does not affect negatively the performances of the tool on those examples that could already be verified, improving the solving time for many of them.

1.2 Related Work

Combining different techniques for enabling the efficient analysis of complex inputs is a common practice in software verification (see, e.g., [1, 23, 2]). Our

work was inspired by [1], presenting an efficient combination of Abstract Interpretation with interpolation-based refinement procedures, limited to the quantifier-free case, though, while our approach works at a quantified level.

Our approach is completely automatic: this mainly differentiates it from semi-automatic approaches, where usually the predicates are suggested by the user, such as [18]. Our approach is not constrained to the inference of invariants of a given shape, in contrast with the approach of [25]. We can handle program with any control-flow structure, as opposed to [17], and we exploit SMT-based interpolating procedures for the refinement of inductive invariant: this differentiates our work from those exploiting first-order theorem provers [24]. In contrast to [19], our approach is not based on machine-learning.

Inference of array-programs properties can be performed as well by adopting an abstract domain segmenting the array, syntactically [20, 22] or semantically [14], and assigning to each segment an abstract value. Such approaches may be very efficient in terms of computation time and memory usage, but may produce false alarms due to the over-approximations involved: (i) the invariants are constrained by the abstract domain (e.g. if the domain is intervals, only interval properties may be expressed) and the segmentation in use; in contrast, our approach has more flexibility (our domain is a large class of first-order formulas); (ii) in addition to the over-approximation introduced by the abstraction, further over-approximation is generally needed to enforce termination of the analysis, through widening operators.

2 Preliminaries

We use lower-case Latin letters $x, i, c, d, e, \ldots$ for variables; for tuples of variables we use bold face letters like $\mathbf{x}, \mathbf{i}, \mathbf{c}, \mathbf{d}, \mathbf{e} \ldots$. $|\mathbf{x}|$ is the length of the tuple $\mathbf{x}$. Given a variable v, v' is a copy of v with a prime, $v^{(n)}$ is a copy of v with n primes, $\mathbf{v}' = \{v' \mid v \in \mathbf{v}\}$ and $\mathbf{v}^{(n)} = \{v^{(n)} \mid v \in \mathbf{v}\}$ for any tuple of variables $\mathbf{v}$.

With $E(\mathbf{x})$ we denote that the syntactic expression (term, formula, tuple of terms or of formulæ) E contains *at most* the free variables *taken from* the tuple $\mathbf{x}$. We use lower-case Greek letters $\phi, \varphi, \psi, \ldots$ for formulæ. $E(y/x)$ is the expression E where the free occurrences of x have been substituted by y. The notation $\phi(\mathbf{t})$ identifies a quantifier-free formula ϕ obtained from $\phi(\mathbf{x})$ by substituting the tuple of variables $\mathbf{x}$ with the tuple of terms $\mathbf{t}$.

As in the SMT-LIB standard [9], a theory $\mathcal{T}$ is a pair $(\Sigma, \mathcal{C})$, where Σ is a signature and $\mathcal{C}$ is a class of Σ-structures; the structures in $\mathcal{C}$ are called the models of $\mathcal{T}$. A Σ-formula α is $\mathcal{T}$-satisfiable if there exists a Σ-structure $\mathcal{M}$ in $\mathcal{C}$ such that α is true in $\mathcal{M}$ under a suitable assignment to the free variables

of α (in symbols, $\mathcal{M} \models \alpha$); it is $\mathcal{T}$-valid (in symbols, $\mathcal{T} \models \alpha$) if its negation is $\mathcal{T}$-unsatisfiable. ψ_1 $\mathcal{T}$-entails ψ_2 (in symbols, $\psi_1 \models_{\mathcal{T}} \psi_2$) iff $\psi_1 \rightarrow \psi_2$ is $\mathcal{T}$-valid. The satisfiability modulo the theory $\mathcal{T}$ ($SMT(\mathcal{T})$) problem amounts to establishing the $\mathcal{T}$-satisfiability of quantifier-free Σ-formulæ.

In this paper the theory of arrays plays a central role, given the desired applications of our approach. We will work mainly with arrays of integers. and denote with $A_{\mathbb{Z}}$ the theory of Presburger arithmetic with the addition of free function symbols (the free function symbols will represent the array variables of the analyzed programs).

3 Safe Transition Systems

We assume programs are represented as transition systems, i.e., triples $\mathcal{S}_{\mathcal{T}} = (\mathbf{v}, l_{\mathsf{init}}, T)$ where $\mathbf{v}$ is the tuple of variables handled by the program with the addition of a fresh variable pc taking values over the set of program locations $L = \{l_1, \ldots, l_n\}$, among which we distinguish an 'initial' location l_{init} and an 'error' location l_{error}. T is a set of relations $\{\tau_1(\mathbf{v}, \mathbf{v}'), \ldots, \tau_n(\mathbf{v}, \mathbf{v}')\}$ encoding the program body, and $\mathcal{T}$ is a first order theory fixing the semantics of programs operations encoded into the τ's. By abuse of notation,

$$T(\mathbf{v}, \mathbf{v}') := \bigvee_{\tau \in T} \tau(\mathbf{v}, \mathbf{v}')$$

For each $\tau_j \in T$ we identify two locations called the 'source' location $src(\tau_j)$ and the 'target' location $trg(\tau_j)$. For each τ_j it holds that $\tau_j \models pc = src(\tau_j)$ and $\tau_j \models pc' = trg(\tau_j)$. For all $\tau_j \in T, src(\tau_j) \neq l_{\mathsf{error}}$ and there exists at least one transition $\tau_i \in T$ such that $src(\tau_i) = l_{\mathsf{init}}$.

Example 3.1. The D10 procedure given in the introduction is represented by the transition system $\mathcal{S}_{A_{\mathbb{Z}}} = (\mathbf{v}, l_{\mathsf{init}}, T)$ with $\mathbf{v} = (pc, i, j, a, b)$. pc takes values over the set $L = \{l_{\mathsf{init}}, l_1, l_2, l_3, l_{\mathsf{error}}\}$. T comprises the following relations[2]:

$$\begin{aligned}
\tau_1 &:= pc = l_{\mathsf{init}} \wedge i' = 1 \wedge j' = 0 \wedge pc' = l_1 \\
\tau_2 &:= \begin{pmatrix} pc = l_1 \wedge i < size \wedge a[j] = b[2j+1] \wedge \\ a' = \mathsf{store}(a, j, b[i]) \wedge i' = i + 2 \wedge j' = j + 1 \end{pmatrix} \\
\tau_3 &:= pc = l_1 \wedge i \geq size \wedge i' = 1 \wedge j' = 0 \wedge pc' = l_2 \\
\tau_4 &:= pc = l_2 \wedge i < size \wedge a[j] = b[2j+1] \wedge i' = i + 2 \wedge j' = j + 1 \\
\tau_5 &:= pc = l_2 \wedge i \geq size \wedge pc' = l_3 \\
\tau_6 &:= pc = l_2 \wedge i < size \wedge a[j] \neq b[2j+1] \wedge pc' = l_{\mathsf{error}}
\end{aligned}$$

Our goal is to check if a given transition system $\mathcal{S}_{\mathcal{T}}$ is safe with respect to its error location l_{error}. Formally we have the following definition:

[2]For every τ_j, we assume $v' = v$ for any $v' \in \mathbf{v}'$ not appearing in τ_j.

Definition 3.1 (Safety)**.** *A transition system* $\mathcal{S}_\mathcal{T} = (\mathbf{v}, l_{\mathsf{init}}, T)$ *is* safe *iff the following formula*

$$pc^{(0)} = l_{\mathsf{init}} \ \wedge \ \bigwedge_{i=0}^{n} T(\mathbf{v}^{(i)}, \mathbf{v}^{(i+1)}) \ \wedge \ pc^{(n+1)} = l_{\mathsf{error}} \tag{2}$$

is $\mathcal{T}$*-unsatisfiable for any* $n \geq 1$*.*

It is well-known (see, e.g., [26]) that one can show (2) to be unsatisfiable by providing a safe *inductive invariant* for $\mathcal{S}_\mathcal{T}$:

Definition 3.2 (Invariants)**.** *A* safe inductive invariant *for* $\mathcal{S}$ *is a formula* $H(\mathbf{v})$ *such that*

$$\begin{aligned} &(i)\ \mathcal{T} \models \forall \mathbf{v}. pc = l_{\mathsf{init}} \rightarrow H(\mathbf{v}) \\ &(ii)\ \mathcal{T} \models \forall \mathbf{v}, \mathbf{v}'. H(\mathbf{v}) \wedge T(\mathbf{v}, \mathbf{v}') \rightarrow H(\mathbf{v}') \\ &(iii)\ \mathcal{T} \models \forall \mathbf{v}. H(\mathbf{v}) \rightarrow pc \neq l_{\mathsf{error}} \end{aligned} \tag{3}$$

If $H(\mathbf{v})$ *satisfies only* (i) *and* (ii)*, it is said to be an* inductive invariant *(but not safe).*

In our approach, as discussed in the next Section, an Abstract Interpreter infers inductive invariants (which, by themselves, do not prove the safety of the system), which are then "suggested" to a LAWI framework, in order to limit its divergence. This "suggestion" phase can be formalized with the notion of *constrained transition system.*

Definition 3.3 (Constrained transition system)**.** *Let* $\mathcal{S}_\mathcal{T} = (\mathbf{v}, l_{\mathsf{init}}, T)$ *be a transition system and* $K(\mathbf{v})$ *an invariant for* $\mathcal{S}_\mathcal{T}$*. The* constrained transition system $\mathcal{S}_\mathcal{T}^{K(\mathbf{v})}$ *is defined as* $(\mathbf{v}, l_{\mathsf{init}}, T')$ *where* $T' = \{K(\mathbf{v}) \wedge \tau_j(\mathbf{v}, \mathbf{v}') \mid \tau_j(\mathbf{v}, \mathbf{v}') \in T\}$*.*

The transformation is sound, from a safety point of view, as shown by the following result.

Theorem 3.1. $\mathcal{S}_\mathcal{T}$ *is safe iff* $\mathcal{S}_\mathcal{T}^{K(\mathbf{v})}$ *is safe.*

4 Generating better interpolants

The Lazy Abstraction with Interpolants (LAWI) framework resembles CounterExample-Guided Abstraction Refinement (CEGAR) [12]. Given $\mathcal{S}_\mathcal{T}$ it generates an abstract system $\hat{\mathcal{S}}_\mathcal{T}$ in such a way that the set of possible executions of $\mathcal{S}_\mathcal{T}$ is a sub-set of those of $\hat{\mathcal{S}}_\mathcal{T}$; the converse does not hold. Thus, any property that holds for the executions of $\hat{\mathcal{S}}_\mathcal{T}$ also holds for those of $\mathcal{S}_\mathcal{T}$. If there exists an execution $\hat{\pi}$ of $\hat{\mathcal{S}}_\mathcal{T}$ not satisfying the property, we cannot conclude that there

exists an execution of $\mathcal{S}_\mathcal{T}$ violating the property. In this case, the framework generates a formula ϕ_π of the kind

$$pc^{(0)} = l_{\mathsf{init}} \wedge \left[\bigwedge_{i=0}^{n} \tau_i(\mathbf{v}^{(i)}, \mathbf{v}^{(i+1)})\right] \wedge pc^{(n+1)} = l_{\mathsf{error}} \tag{4}$$

with $\tau_i \in T$, that is $\mathcal{T}$-satisfiable only if $\mathcal{S}_\mathcal{T}$ admits π as a counterexample. If ϕ_π is $\mathcal{T}$-unsatisfiable, a set of interpolants are computed to refine the abstraction level of $\hat{\mathcal{S}}_\mathcal{T}$ and exclude $\hat{\pi}$ from the set of its admitted counterexamples. Such interpolants are a sequence of formulæ $\{I_0(\mathbf{v}), \ldots, I_n(\mathbf{v})\}$ satisfying the following constraints [28]: (i) $I_0(\mathbf{v}) \equiv \top$, (ii) $I_n(\mathbf{v}) \equiv \bot$, (iii) for all $1 \leq i \leq n, I_{i-1}(\mathbf{v}) \wedge \tau_i(\mathbf{v}, \mathbf{v}') \models_\mathcal{T} I_i(\mathbf{v}')$ and (iv) all the free variables of I_i, for all $1 \leq i < n$ occur both in τ_i and τ_{i+1}. Interpolating procedures usually exploit the proof of unsatisfiability of the (4) in order to generate the sequence $\{I_0(\mathbf{v}), \ldots, I_n(\mathbf{v})\}$. By transforming a transition system $\mathcal{S}_\mathcal{T}$ into a constrained transition system $\mathcal{S}_\mathcal{T}^{K(\mathbf{v})}$, the (4) becomes

$$pc^{(0)} = l_{\mathsf{init}} \wedge \left[\bigwedge_{i=0}^{n} K(\mathbf{v}^{(i)}) \wedge \tau_i(\mathbf{v}^{(i)}, \mathbf{v}^{(i+1)})\right] \wedge pc^{(n+1)} = l_{\mathsf{error}} \tag{5}$$

The main issue is that interpolating procedures tend to ignore the presence of $K(\mathbf{v})$ in (5). This section describes how it is possible to "force" them to exploit this additional information to generate better interpolants. We will first briefly introduce the FLATTENING and TERMABSTRACTION techniques. We will subsequently show that the additional information contained in $K(\mathbf{v})$ can lead, by adopting such two techniques, to the generation of better interpolants.

4.1 Flattening and Term Abstraction

FLATTENING and TERMABSTRACTION are two techniques required in LAWI approaches dealing with programs with arrays, as shown in [5]. FLATTENING is based on the rewriting rule $\phi(a[t], ...) \rightsquigarrow \exists x(x = t \wedge \phi(a[x], ...))$, for a fresh x. Given a transition system $\mathcal{S}_\mathcal{T}$, FLATTENING returns an equivalent copy of $\mathcal{S}_\mathcal{T}$ where every array variable is indexed only by existentially quantified variables. This preprocessing operation gives the opportunity of computing quantified invariants by exploiting quantifier-free interpolants. The idea is that the procedure in charge of checking the $\mathcal{T}$-satisfiability of a formula like (4) or (5) pre-processes the counterexample formulæ by Skolemizing the existentially quantified variables introduced by FLATTENING and then instantiating the implicit universally quantified variables contained in the array store symbols, since $E(\ldots, \mathsf{store}(a, i, e), \ldots) \equiv E(\ldots, a', \ldots) \wedge a'[i] = e \wedge \forall x.(x \neq i \rightarrow a'[x] = a[x])$ for a fresh a', and atoms like $a = b$ for a, b array variables, because $a = b \equiv \forall x.a[x] = b[x]$, for a fresh x. Any Skolem constant contained in the interpolants will act as an "implicitly quantified" variable[3].

[3]The interested reader can refer to [5] for further information about this framework.

As discussed in the introduction, the performance of a LAWI-based framework strongly depends on the quality of the computed interpolants. TERMABSTRACTION addresses this problem. It takes in input the two inconsistent formulæ A, B for which one wants to compute an interpolant I and iteratively try to remove a list of terms $\lambda = \langle t_1, \ldots, t_n \rangle$ from the shared language of A and B. Terms in λ are guilty of keeping the interpolants too precise. TERMABSTRACTION iteratively checks whether $A(c_i/t_i) \wedge B(d_i/t_i)$ is inconsistent, for term t_i and fresh constants c_i, d_i. At the end, A and B will be free from some terms of λ and the interpolant will be likely free from such terms as well. The intuition behind TERMABSTRACTION is that if a formula remains unsatisfiable even though some terms are replaced with new unknowns, then the resulting interpolants will be less specific, since they will not exploit the particulars of these terms. Notably, BOOSTER (the software model-checker over which we implemented our new invariant generation technique) does not require any user interaction for generating suitable term abstraction lists, as explained in [2].

Example 4.1. LAWI frameworks usually carries out the verification of a program by representing precisely the control-flow structure and abstracting away the data-flow information (see, e.g., [28]). Let's consider the counterexample τ_1, τ_3, τ_6, which is the one raised by the execution of SAFARI [6] on the transition system discussed in the Example 3.1. This counterexample is represented by the conjunction of the formulæ[4]

$$\begin{array}{ll} (\tau_6) & i^{(1)} < size \wedge a^{(1)}[j] \neq b^{(1)}[2j+1] \\ (\tau_3) & i^{(2)} \geq size \wedge i^{(1)} = 1 \wedge j^{(1)} = 0 \wedge id(a, b, 2) \\ (\tau_1) & i^{(2)} = 1 \wedge j^{(2)} = 0 \end{array}$$

A state-of-the-art interpolating prover, like iZ3 [27], returns this set of interpolants[5] for such a formula:

$$I_0 := \top \qquad I_1 := i^{(1)} < size$$
$$I_2 := i^{(2)} \neq 1 \qquad I_3 := \bot$$

The transition system returned by FLATTENING is

$$\tau_1 := pc = l_{\mathsf{init}} \wedge i' = 1 \wedge j' = 0 \wedge pc' = l_1$$
$$\tau_2 := \exists x. \begin{pmatrix} pc = l_1 \wedge x = i \wedge i < size \wedge 2j + 1 = i \wedge i \geq 1 \wedge \\ a' = \mathsf{store}(a, j, b[x]) \wedge i' = x + 2 \wedge j' = j + 1 \end{pmatrix}$$
$$\tau_3 := pc = l_1 \wedge i \geq size \wedge 2j + 1 = i \wedge i \geq 1 \wedge i' = 1 \wedge j' = 0 \wedge pc' = l_2$$
$$\tau_4 := \exists x, y. \begin{pmatrix} pc = l_2 \wedge x = j \wedge y = 2x + 1 \wedge \\ a[x] = b[y] \wedge 2j + 1 = i \wedge i < size \wedge \\ i \geq 1 \wedge i' = i + 2 \wedge j' = j + 1 \end{pmatrix}$$

[4] For readability we omitted the *pc* variable etc. Also, $id(t_1, \ldots, t_n; k)$ for $t_1^{(k)} = t_1^{(k-1)} \wedge \ldots \wedge t_n^{(k)} = t_n^{(k-1)}$.

[5] `http://rise4fun.com/iZ3/ukd`

$$\tau_5 := pc = l_2 \wedge i \geq size \wedge 2j + 1 = i \wedge i \geq 1 \wedge pc' = l_3$$

$$\tau_6 := \exists x, y. \begin{pmatrix} pc = l_2 \wedge x = j \wedge y = 2x + 1 \wedge \\ i < size \wedge a[x] \neq b[y] \wedge 2j + 1 = i \wedge \\ i \geq 1 \wedge pc' = l_{\mathsf{error}} \end{pmatrix}$$

Unsurprisingly, the interpolants do not change if we exploit the counterexample generated from these transitions[6]. We obtain better interpolants by applying TermAbstraction with the term abstraction list $\langle i, j \rangle$. In this case, j can be abstracted away, giving the interpolants

$$I_0 := \top$$
$$I_1 := size > i^{(1)} \wedge z_1 = 2z_0 + 1 \wedge b^{(1)}[z_1] \neq a^{(1)}[z_0]$$
$$I_2 := z_1 = 2z_0 + 1 \wedge a^{(2)}[z_0] \neq b^{(2)}[z_1] \wedge i^{(2)} > 1$$
$$I_3 := \bot$$

These two interpolants are better than the one obtained without TermAbstraction, as they include some Skolem variables. Given the backward nature of the lawi framework we are exploiting, these Skolem variables will be the universally quantified variables of the final safe inductive invariant that we are aiming for.

The combination of Flattening and TermAbstraction inside a lawi framework allows to achieve very good results, but still fails on non-trivial examples such as the D10 procedure. Next Section shows how to overcome this limitation.

4.2 Constrained transition systems yielding better interpolants

In our framework we exploit an abstract interpreter for the generation of inductive invariants. *Abstract interpretation* is a general framework for the efficient generation of invariants (see, e.g., [13]). It restricts the search for the invariant to an *abstract domain*: e.g. a conjunction of *interval* constraints $L_x \leq x \leq U_x$, one for each variable x, where constants L_x and U_x are found by the abstract interpreter [13]; or, in the domain of (convex) *polyhedra*, arbitrary conjunctions of linear (in)equalities [15].

In this paper we will work with convex polyhedra. This is the abstract domain $\mathcal{P} = (\mathbb{P}, \sqsubseteq, \sqcup, \sqcap, \triangledown)$ where $\mathbb{P}$ is the set of linear inequalities over $\mathbf{v}$, $\sqsubseteq$ is a partial order over $\mathbb{P}$, $\sqcup$ and $\sqcap$ are respectively the join and the meet operators of the lattice $(\mathbb{P}, \sqsubseteq)$ and $\triangledown$ is a widening operator. We assume that our abstract interpreter computes a standard upward Kleene iteration sequence over $\mathcal{P}$ driven by program instructions over the scalars. Operations on arrays are treated as follows: array reads return undefined values, array

[6] `http://rise4fun.com/iZ3/plGl`

writes are ignored. Convergence to a fixpoint is guaranteed by the application of the widening operator $\triangledown$[7]. The abstract interpreter takes, therefore, as input a program $\mathcal{S}_\mathcal{T}$ and returns for each control location an *inductive invariant*, that is, an element of $\mathbb{P}$ closed by post-image computation with respect to the τ's of $\mathcal{S}_\mathcal{T}$, which therefore includes all reachable states at that location. After converting, for each program location l, the invariant into a first-order formula $C_l(\mathbf{v})$, one obtains an inductive invariant for $\mathcal{S}_\mathcal{T}$, satisfied by all reachable states:

$$K(\mathbf{v}) \;:=\; \bigwedge_{l \in L} pc = l \rightarrow C_l(\mathbf{v}) \tag{6}$$

Given an inductive invariant $K(\mathbf{v})$ for a transition system $\mathcal{S}_\mathcal{T}$, we can build a constrained system $\mathcal{S}_\mathcal{T}^{K(\mathbf{v})}$, in such a way to include the $K(\mathbf{v})$ in the counterexample formulæ, as stated in Section 3. Providing (5) instead of (2) to an interpolating theorem prover does not always ensure that the new interpolants will be more general, though. The reason is that the unsatisfiability cores of (2) and (5), from which the interpolants are computed, might be the same; this depends on the internal heuristics of the (SMT-)solver in use. We thus need to nudge the solver into using the invariants in its proof and thus in the interpolants: this can be done by exploiting Flattening and TermAbstraction.

The intuition is that the additional inductive invariants may help in abstracting away more terms in the term abstraction list: the Flattening procedure links the new inductive invariants to the transition relation allowing TermAbstraction to reveal different unsat-cores, leading to the computation of different interpolants. Lemmas inferred by an abstract interpreter are general facts about the program (since they are inductive facts), and unsat-cores containing them are more likely to generate general interpolants.

Example 4.2. We consider again the transition system given in the Example 3.1. An abstract interpreter working with the polyhedra abstract domain [15, 8] infers the inductive invariant $K(\mathbf{v}) := pc \neq l_{\mathsf{init}} \rightarrow (2j+1 = i \wedge i \geq 1)$. The counterexample τ_6, τ_3, τ_1 of the 'flat' version of $\mathcal{S}_{A_\mathbb{Z}}^{K(\mathbf{v})}$ is represented by the conjunction of the formulæ

$$\begin{aligned}
(\tau_6)\quad & \begin{bmatrix} 1 \leq i^{(1)} \wedge 2j^{(1)} + 1 = i^{(1)} \wedge z_1 = 2z_0 + 1 \wedge \\ j^{(1)} = z_0 \wedge i^{(1)} < size \wedge a^{(1)}[z_0] \neq b^{(1)}[z_1] \end{bmatrix} \\
(\tau_3)\quad & \begin{bmatrix} 1 \leq i^{(2)} \wedge 2j^{(2)} + 1 = i^{(2)} \wedge i^{(1)} = 1 \wedge \\ size \leq i^{(2)} \wedge j^{(1)} = 0 \wedge id(a[z_0], a[z_1], b[z_0], b[z_1]; 2) \end{bmatrix} \\
(\tau_1)\quad & i^{(2)} = 1 \wedge j^{(2)} = 0
\end{aligned}$$

[7] For more information on this abstract domain, the interested reader is referred to [15, 8].

This time, TermAbstraction can abstract away the occurrences of i and j when generating the interpolant "between" τ_6 and τ_3, and abstracts away i when computing the interpolant "between" τ_3 and τ_1:

$$
\begin{aligned}
&I_0 := \top \\
&I_1 := 0 \leq z_0 \wedge b[z_1] \neq a[z_0] \wedge z_1 = 2z_0 + 1 \wedge size > 2z_0 + 1 \\
&I_2 := 0 \leq z_0 \wedge b[z_1] \neq a[z_0] \wedge z_1 = 2z_0 + 1 \wedge 2j > 1 \\
&I_3 := \bot
\end{aligned}
$$

With the adoption of this combined framework, Booster can infer a safe inductive invariant for the D10 procedure.

5 Experimental evaluation

We implemented our technique on the top of the Booster model-checker, available from `http://www.inf.usi.ch/phd/alberti/prj/booster/`. Booster is a software model-checker targeting the analysis of array programs [2]. It integrates several orthogonal verification techniques, such as *bounded model checking* [10] (for fast bug finding only), *acceleration* [3, 4] and *lazy abstraction with interpolants* (lawi) [5]. It follows the standard architecture of a compiler: a parser yields an intermediate representation, which is subject to several optimizations before being fed to an engine for checking its safety.

The abstract interpreter we implemented in Booster relies on the Parma Polyhedra Library, offering an efficient implementation of the results presented in [8].

Flattening and TermAbstraction are already implemented in Booster, as described in [2]. Moreover, Booster adopts a portfolio approach for the lawi framework, i.e., it executes in parallel several instances of the fixpoint engine trying different settings among the most promising ones, including different term abstraction lists. This way, the user does not have to suggest any term abstraction list. For the experimental evaluation, we compared the old version of Booster with the new one, enhanced with the abstract interpreter.[8]

[8] Besides a lawi framework for arrays, Booster comprises an acceleration-based framework. Acceleration is a completely orthogonal technique with respect to abstraction, and its performance is not affected by the presence of the abstract interpreter. This means that acceleration contributes *equally* to both versions of the tool and it is not biasing the experimental evaluation.

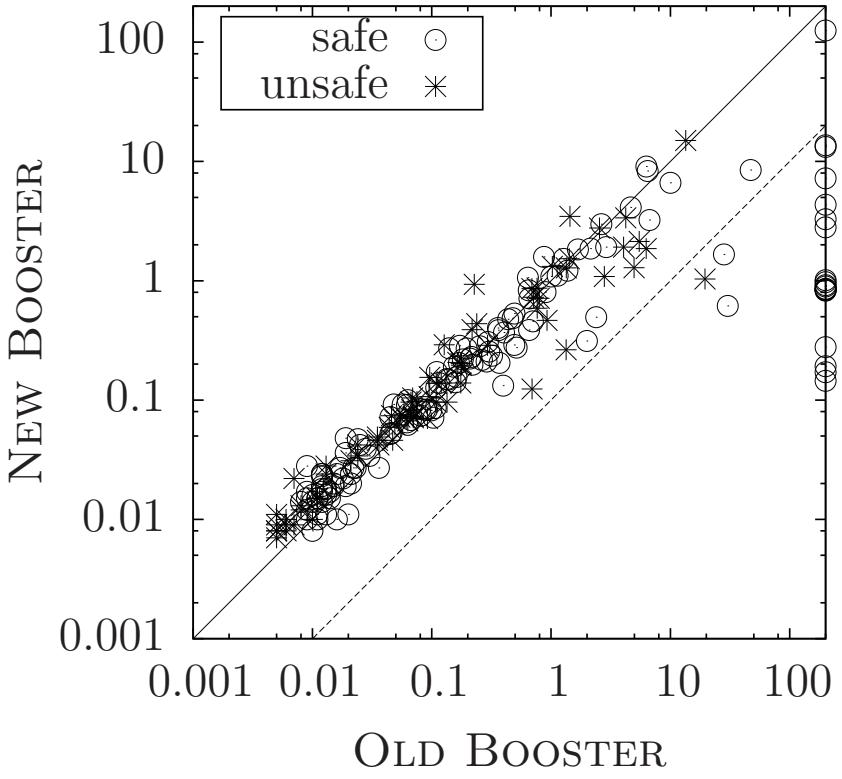

Figure 1: Comparison of the running time of the old and the new version of Booster. Time-out has been set to 200 seconds. A point below the diagonal represents an example where the new version beats the old version. Points on the right vertical axes represent benchmarks on which the old version times out.

Benchmark	Status	New	Old
set property (ground)	SAFE	4.333	T.O.
copy9 (ground)	SAFE	6.618	10.027
partial init (ground)	SAFE	0.615	30.232
bubble sort v1	SAFE	0.368	0.403
bubble sort v1 (bug, ground)	UNSAFE	1.085	2.798
bubble sort (ground)	SAFE	13.655	T.O.
selection sort	SAFE	13.252	T.O.
D10	SAFE	0.972	T.O.

Figure 2: Running time (in seconds) for significant examples. The "ground" flag indicates that the program have only quantifier-free assertions, while "bug" flags the examples edited in order to insert a bug.

We experimentally evaluated our new technique on a large set of programs handling arrays. These files comes from different heterogeneous sources (related works, Internet, SV-COMP, etc.). Figure 1 compares the running time of the new version of BOOSTER with the old version. Table 2 reports the statistics for some representative benchmarks. The results show clearly that the abstract interpreter does not decrease the performances on the examples that were already verifiable by BOOSTER in its previous version. In the set of the new examples verifiable by BOOSTER there are the entire benchmark suite of [16] and several sorting algorithms such as bubble sort (in two different versions) and selection sort. The negligible slowdown on very easy examples, i.e., those for which the verification takes a time between 0.01 and 0.1 seconds, is due to the execution of the abstract interpreter. In contrast, the analysis is sped up by more than an order of magnitude on bigger (and more significant) examples (represented by the points below the dashed line), and examples that resulted in time-out are now proved automatically. We point out that the verification starts *from the source-code* and is *fully automatic*.

6 Conclusions and Future work

In this paper we have shown the beneficial effects of integrating an abstract interpreter working on a numerical domain with an SMT-based refinement procedure. Despite its simplicity, this framework is able to achieve important results in the area of analysis of programs with arrays.

As a future work, it would be interesting to evaluate the benefits of abstract domain targeting the inference of *quantified* inductive invariants, e.g., [14, 21]. This is not straightforward: a quantified inductive invariant would add universal quantifiers in the guard of the transitions. This means that the LAWI framework should include some techniques to deal with these extra quantifiers, as it has been done in [4] to deal with accelerated transitions.

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. "Craig Interpretation". In: *SAS*. Vol. 7460. LNCS. Springer, 2012, pp. 300–316.

[2] F. Alberti, S. Ghilardi, and N. Sharygina. "Booster: an acceleration-based verification framework for array programs". In: *ATVA*. Vol. 8837. LNCS. Springer, 2014.

[3] F. Alberti, S. Ghilardi, and N. Sharygina. "Decision Procedures for Flat Array Properties". In: *TACAS*. Vol. 8413. LNCS. Springer, 2014, pp. 15–30.

[4] F. Alberti, S. Ghilardi, and N. Sharygina. "Definability of Accelerated Relations in a Theory of Arrays and Its Applications". In: *FroCoS*. Vol. 8152. LNCS. Springer, 2013, pp. 23–39.

[5] F. Alberti et al. "An extension of lazy abstraction with interpolation for programs with arrays". In: *FMSD* 45.1 (2014), pp. 63–109.

[6] F. Alberti et al. "SAFARI: SMT-Based Abstraction for Arrays with Interpolants". In: *CAV*. Vol. 7358. LNCS. Springer, 2012, pp. 679–685.

[7] E. De Angelis et al. "VeriMAP: A Tool for Verifying Programs through Transformations". In: *TACAS*. Vol. 8413. LNCS. Springer, 2014, pp. 568–574.

[8] R. Bagnara et al. "Precise widening operators for convex polyhedra". In: *Sci. Comput. Program.* 58.1-2 (2005), pp. 28–56.

[9] C. Barrett, A. Stump, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2010.

[10] A. Biere et al. "Symbolic Model Checking without BDDs". In: *TACAS*. Vol. 1579. LNCS. Springer, 1999, pp. 193–207.

[11] N. Bjørner, K.L. McMillan, and A. Rybalchenko. "On Solving Universally Quantified Horn Clauses". In: *SAS*. Vol. 7935. LNCS. Springer, 2013, pp. 105–125.

[12] E.M. Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *CAV*. Vol. 1855. LNCS. Springer, 2000, pp. 154–169.

[13] P. Cousot and R .Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *POPL*. ACM, 1977, pp. 238–252.

[14] P. Cousot, R. Cousot, and F. Logozzo. "A parametric segmentation functor for fully automatic and scalable array content analysis". In: *POPL*. ACM, 2011, pp. 105–118.

[15] P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *POPL*. ACM, 1978, pp. 84–96.

[16] I. Dillig, T. Dillig, and A. Aiken. "Fluid Updates: Beyond Strong vs. Weak Updates". In: *ESOP*. Vol. 6012. LNCS. Springer, 2010, pp. 246–266.

[17] I. Dragan and L. Kovács. "LINGVA: Generating and Proving Program Properties using Symbol Elimination". In: *PSI*. To appear. 2014.

[18] C. Flanagan and S. Qadeer. "Predicate abstraction for software verification". In: *POPL*. ACM, 2002, pp. 191–202.

[19] P. Garg et al. "ICE: A Robust Framework for Learning Invariants". In: *CAV*. Vol. 8559. LNCS. Springer, 2014, pp. 69–87.

[20] D. Gopan, T.W. Reps, and S. Sagiv. “A framework for numeric analysis of array operations”. In: *POPL*. ACM, 2005, pp. 338–350.

[21] S. Gulwani, B. McCloskey, and A. Tiwari. “Lifting abstract interpreters to quantified logical domains”. In: *POPL*. ACM, 2008, pp. 235–246.

[22] N. Halbwachs and M. Péron. “Discovering properties about arrays in simple programs”. In: *PLDI*. 2008, pp. 339–348.

[23] J. Henry, D. Monniaux, and M. Moy. “Succinct Representations for Abstract Interpretation”. In: *SAS*. Vol. 7460. LNCS. Springer, 2012, pp. 283–299.

[24] K. Hoder, L. Kovács, and A. Voronkov. “Invariant Generation in Vampire”. In: *TACAS*. Vol. 6605. LNCS. Springer, 2011, pp. 60–64.

[25] R. Jhala and K.L. McMillan. “Array Abstractions from Proofs”. In: *CAV*. Vol. 4590. LNCS. Springer, 2007, pp. 193–206.

[26] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[27] K.L. McMillan. “Interpolants from Z3 proofs.” In: *FMCAD*. FMCAD Inc., 2011, pp. 19–27.

[28] K.L. McMillan. “Lazy Abstraction with Interpolants”. In: *CAV*. Vol. 4144. LNCS. Springer, 2006, pp. 123–136.