Lazy Abstraction with Interpolants for Arrays

Francesco Alberti[1], Roberto Bruttomesso[2], Silvio Ghilardi[2], Silvio Ranise[3], and Natasha Sharygina[1]

[1] Università della Svizzera Italiana, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] FBK-Irst, Trento, Italy

Abstract. Lazy abstraction with interpolants has been shown to be a powerful technique for verifying imperative programs. In presence of arrays, however, the method shows an intrinsic limitation, due to the fact that successful invariants usually contain universally quantified variables, which are not present in the program specification. In this work we present an extension of the interpolation-based lazy abstraction in which arrays of unknown length can be handled in a natural manner. In particular, we exploit the Model Checking Modulo Theories framework, to derive a backward reachability version of lazy abstraction that embeds array reasoning. The approach is generic, in that it is valid for both parameterized systems and imperative programs. We show by means of experiments that our approach can synthesize and prove universally quantified properties over arrays in a completely automatic fashion.

1 Introduction

The automatic verification of software is a long standing scientific challenge. A promising line of research is that in which Model Checking techniques are employed to automatically traverse the state-space of a program, and check it with respect to a user-specified property. Since the problem is undecidable, complete and fully automatic techniques cannot exist and the programmer must provide additional annotations describing, for instance, loop invariants. It is well-known that the task of providing such annotations is far from trivial. In order to significantly alleviate the annotation burden, it is crucial to employ abstraction techniques. For example, Predicate Abstraction [13], the CEGAR approach [4], or Lazy Abstraction [16] have been shown successful and are nowadays employed in many state-of-the-art software verification tools. In particular, Lazy Abstraction is capable of tuning the abstraction by using different degrees of precision for different parts of the program by keeping track of both the control-flow graph, which describes *how* the program locations are traversed, and the data-flow, which describes *what* holds at a program location. The control-flow is represented explicitly, while the data-flow is symbolically encoded with quantifier-free first-order formulæ and it is subjected to abstraction. The procedure is therefore based on a CEGAR loop in which the control-flow graph is iteratively unwinded, and the data in the newly explored locations is overapproximated. When reaching an error location, if the path is spurious—i.e. the quantifier-free formula

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 46–61, 2012.
© Springer-Verlag Berlin Heidelberg 2012

representing the manipulations of the data along the path is unsatisfiable, the abstraction along the path is refined. In state-of-the-art methods, this is done by means of interpolants [15,21]. The procedure terminates when a non-spurious path is found, or when reaching an inductive invariant.

When arrays come into the picture the situation is complicated by at least two problems. First, the need of handling quantified formulæ (as opposed to just quantifier-free) to take care of meaningful array properties; e.g., a typical post-condition of a sorting algorithm is the following universally quantified formula:

$$\forall i, j.\ (0 \leq i < j \leq a.length) \Rightarrow a[i] \leq a[j],$$

expressing the fact that the array a is sorted, where $a.length$ represents the *symbolic* size of a. Second, the difficulty of computing quantifier-free interpolants. In [18], it is shown that quantifiers must occur in interpolants of quantifier-free formulæ for the "standard" theory of arrays.

This paper contributes a new verification approach that addresses the above problems. It redefines the lazy abstraction method based on interpolation (which is known to be one of the most effective approaches in program verification) and makes it possible to reason about arrays of unknown length. For that, it exploits the framework behind the Model Checking Modulo Theory approach (MCMT) [10, 11]. The reasoning about arrays and the corresponding quantified formula is performed by means of the symbolic backward reachability algorithm now extended with the interpolation-based abstraction refinement techniques. This combination is able to generate the quantified predicates required for the synthesis of the quantified inductive invariants needed to establish the validity of the program assertions. Notably, our abstraction-based approach can be applied to enhance the verification of array-based systems (a wide class of infinite-state systems currently handled by MCMT). We implemented the new approach and verified various common-use programs over arrays.

The paper is organized as follows. Section 2 recalls basic notions about array-based systems as used in MCMT and demonstrates how sequential programs can be specified using this model. Section 3 introduces the new lazy abstraction approach and discusses its completeness and termination. Experiments are presented in Section 4. We conclude in Section 5. Proofs of claims made within the paper are worked out in the online available extended version [1].

Related Work. The work described in this paper can be considered as part of the broad line of research in model-checking for infinite state systems that makes use of abstraction-refinement techniques to cope with the infinite search space [4, 13, 16]. A challenging task in this setting is to find the right predicates to ensure convergence; these predicates may be extracted, e.g., from the proof of unsatisfiability of an infeasible abstract path [15, 21]. When arrays come into the picture the situation is complicated by the need of using quantifiers to express meaningful properties (such as "sortedness"). Earlier work in predicate abstraction approached this issue by using *Skolem constants* [8], *indexed predicates* [24], or *range predicates* [17]. Approaches based on templates [25] may

synthesize more expressive formulæ, but they require manual specification of templates and predicates. To the best of our knowledge, the closest related work to ours is that of [23], where a backward reachability procedure for universally quantified assertions over arrays is described. As in our approach the procedure visits backward the set of unsafe states to find an intersection with the initial ones, by performing the coarsest possible abstraction first. However in [23] the computed abstraction is then refined with predicates obtained by simulating the "pre" operator on a spurious trace, and by performing classical predicate abstraction (requiring injection of, in the worst case, exponentially many bound constraints between indexes), whereas in our approach we achieve refinement by means of interpolants.

Proving properties over arrays has also been extensively studied in the context of abstract domains other than predicate abstraction. The approaches of [5, 6, 12, 14], for example, follow a line of research in which arrays are divided into *segments*, based on the access and write operations in the programs. Several techniques are employed to avoid the combinatorial explosion, e.g., by means of the introduction of a suitable widening operator. These approaches have been shown to be useful even at an industrial-scale level [5] to automatically infer a wide range of properties. The goal of our approach is to automatically verify that the program satisfy expressive properties.

A further promising direction of research relies on saturation-based theorem-provers [19, 20] to generate invariants over arrays. These approaches may in principle produce more expressive invariants than ours, but they require, on the other hand, to instruct the prover with axioms for handling arithmetic. In our setting, instead, we use SMT techniques to take care of the necessary arithmetic operations.

Backward reachability of array-based systems, implemented in the tool MCMT, has been successfully used for checking the safety of several classes of distributed algorithms [10]. The work in this paper shows that MCMT combined with Lazy Abstraction can also be used for verifying expressive properties of sequential programs manipulating arrays. We also characterize when our method behaves as a decision procedure for establishing the safety of classes of array-based systems that cover existing results (e.g., [7]).

2 Background Notions on MCMT

We assume the usual syntactic and semantic notions of many-sorted first-order logic with equality. We use lower-case latin letters $x, a, i, e, \ldots$ for free variables; for tuples of free variables we use underlined letters $\underline{x}, \underline{a}, \underline{i}, \underline{e}, \ldots$ or bold face letters like $\mathbf{a}, \mathbf{v}, \ldots$. With $E(\underline{x})$ 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 $\underline{x}$. According to [22], a theory 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 T. A Σ-formula φ is T-satisfiable if there exists a Σ-structure $\mathcal{M}$ in $\mathcal{C}$ such that φ is true in $\mathcal{M}$ under a suitable assignment to the free variables

of φ (in symbols, $\mathcal{M} \models \varphi$); it is T-valid (in symbols, $T \models \varphi$) if its negation is T-unsatisfiable. Two formulæ φ_1 and φ_2 are T-equivalent if $\varphi_1 \leftrightarrow \varphi_2$ is T-valid; ψ_1 T-entails ψ_2 (in symbols, $\psi_1 \models_T \psi_2$) iff $\psi_1 \rightarrow \psi_2$ is T-valid. The satisfiability modulo the theory T ($SMT(T)$) problem amounts to establishing the T-satisfiability of quantifier-free Σ-formulæ. A theory T has *quantifier-free interpolation* iff there exists an algorithm that, given two quantifier free formulæ ϕ, ψ such that $\phi \wedge \psi$ is T-unsatisfiable, returns a formula θ such that: (*i*) $\phi \models_T \theta$; (*ii*) $\theta \wedge \psi$ is T-unsatisfiable; (*iii*) only the free variables common to ϕ and in ψ occur in θ.

Array-Based Transition Systems and their Safety. We briefly recall some of the notions underlying the framework of MCMT; for an extensive discussion, the reader is pointed to [10]. Array-based systems are a particular class of guarded assignment systems whose state variables comprise arrays. They are represented symbolically using certain classes of formulæ and are endowed with theories specifying the algebraic structures of the indexes and elements of arrays. Roughly, the input language of MCMT for specifying array-based systems can be seen as a parameterized extension of the one used by UCLID (`http://www.cs.cmu.edu/~uclid`). Formally, it is a sub-set of multi-sorted first-order logic extended with theories. In particular, we assume a (mono-sorted) theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes of arrays and a multi-sorted theory $T_E = (\Sigma_E, \mathcal{C}_E)$ for the elements of the arrays. The unique sort of T_I is called `INDEX` and a sort of T_E is called $\texttt{ELEM}_\ell$, where ℓ ranges over a given (finite) set. We also assume that the $SMT(T_I)$**- and** $SMT(T_E)$**-problems are decidable** and that T_I **and** T_E **have quantifier-free interpolation** (further hypotheses will be discussed below in connection to specific model-checking features).

The theory $A_I^E = (\Sigma, \mathcal{C})$, specifying the algebraic structures of the array state variables manipulated by an array-based system is obtained by "composing" T_I and T_E as follows. The sort symbols of A_I^E are `INDEX`, $\texttt{ELEM}_\ell$, and $\texttt{ARRAY}_\ell$, its signature Σ contains all the symbols in the (disjoint) union $\Sigma_I \cup \Sigma_E \cup \{\_[\_]_\ell\}_\ell$ where $\_[\_]_\ell : \texttt{ARRAY}_\ell \times \texttt{INDEX} \rightarrow \texttt{ELEM}_\ell$ are the usual "read" operations of an array on a given cell, and a structure $\mathcal{M}$ is in the class $\mathcal{C}$ of the models of A_I^E when (*i*) the restrictions of $\mathcal{M}$ to Σ_I, Σ_E are models of T_I, T_E, respectively, (*ii*) the sorts $\texttt{ARRAY}_\ell$ are interpreted as (total) functions from $\texttt{INDEX}^{\mathcal{M}}$ to $\texttt{ELEM}_\ell^{\mathcal{M}}$, and (*iii*) the operations $\_[\_]_\ell$ are interpreted as function applications. In the following, the subscript ℓ will be omitted to simplify notation.

In this paper, to simplify technicalities, we adopt the following variant of the notion of an array-based system [10]. An **array-based system (for** T_I, T_E**)** is a pair $\mathcal{S} = \langle \mathbf{v}, \{\tau_h\}_h \rangle$, where $\mathbf{v} = \mathbf{a}, \mathbf{c}, \mathbf{d}$ is the tuple of *system variables* and is such that

- the tuple $\mathbf{a} = a_0, \ldots, a_s$ contains variables of sort `ARRAY`;
- the tuple $\mathbf{c} = c_0, \ldots, c_t$ contains variables of sort `INDEX` (called, *counters*);
- the tuple $\mathbf{d} = d_0, \ldots, d_u$ contains variables of sort `ELEM` (called, *simple variables*).

All variables are sorted, e.g., for $\mathbf{a}$, this means that to each $i = 0, \ldots, t$ is assigned some ℓ so that a_i is of type $\texttt{ARRAY}_\ell$. The variable d_0 ranges over a finite set $\{l_0, ..., l_n\}$ of *program locations* and is usually denoted with pc (short for *program counter*) instead of d_0. Among the program locations, we shall distinguish an *initial* location l_I and an *error* location l_E. It is assumed that the *initial state* of the array-based system $\mathcal{S}$ is represented by the formula $I(\mathbf{v}) := (pc = l_I)$ and the *error state* by the formula $U(\mathbf{v}) := (pc = l_E)$.

It is still possible to specify distributed algorithms considered in [10] using the notion of array-based systems introduced above. In fact, although using a program counter may not make sense for such systems, one can nevertheless use a trivial program counter with three locations only: the initial location l_I, the error location l_E, and a "standard" location l_S for the body of the distributed algorithm. Thus, the lazy abstraction technique that we are going to describe below can also be applied, without modifications, to the verification of distributed systems.

The τ_h's are **guarded assignments in functional form**. To precisely specify what this means, we need to introduce the following conventions and definitions. The symbols e range over variables of a sort $\texttt{ELEM}$ in Σ_E while i, j, k, z range over variables of sort $\texttt{INDEX}$. Notation $\mathbf{a}[\underline{i}]$ abbreviates $a_1[i_1], \ldots, a_s[i_1], \ldots, a_s[i_n]$ for a tuple $\underline{i} \equiv i_1, \ldots, i_n$ of variables of sort $\texttt{INDEX}$. Expressions of the form $\phi(\underline{i}, \underline{e}), \psi(\underline{i}, \underline{e})$ (possibly sub/super-scripted) denote *quantifier-free* $(\Sigma_I \cup \Sigma_E)$*-formulæ in which at most the variables* $\underline{i} \cup \underline{e}$ *may occur*. Furthermore, $\phi(\underline{i}, \underline{t}/\underline{e})$ (or simply $\phi(\underline{i}, \underline{t})$) abbreviates the substitution of the Σ-terms $\underline{t}$ for the variables $\underline{e}$. Thus, for instance, $\phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ denotes the formula obtained by replacing $\underline{e}, \underline{j}, \underline{e}'$ with $\mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d}$ respectively in the quantifier-free formula $\phi(\underline{i}, \underline{e}, \underline{j}, \underline{e}')$. A formula $\forall \underline{i}.\ \phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ is a $\forall^I$*-formula*, one of the form $\exists \underline{i}.\ \phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ is an $\exists^I$*-formula*, and a sentence $\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i} \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})$ is an $\exists^{A,I} \forall^I$*-sentence*. A *guarded assignment in functional form* is a formula of the form

$$\exists \underline{k} \left(\begin{array}{l} \phi_L(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge\ \mathbf{a}' = \lambda j.\ G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, a[j]) \ \wedge \\ \qquad \wedge\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge\ \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d})) \end{array} \right) \tag{1}$$

where $G = G_0, \ldots, G_s$, $H = H_0, \ldots, H_t$, $K = K_0, \ldots, K_u$ are tuples of case-defined functions (roughly, these can be thought of as nested if-then-else expressions, see [10] for a precise definition). As usual, $\mathbf{a}', \mathbf{c}', \mathbf{d}'$ are renamed copies of the $\mathbf{a}, \mathbf{c}, \mathbf{d}$, denoting the values of the state variables immediately after the execution of the guarded assignment. We assume that the guard ϕ_L of a guarded assignment in functional form (1) always contains a conjunct of the form $pc = l$ and that the update function K_0 is of the form $pc = l'$. In this way, we have mappings from guarded assignments and locations: if the guarded assignment is named τ, the locations l and l' are called the *source* and the *target* locations of τ and are denoted by $src(\tau)$ and $trg(\tau)$, respectively.

The array-based system
$\mathcal{S} = \langle \mathbf{v}, \{\tau_h\}_h \rangle$ is *safe* iff the formulæ

$$I(\mathbf{v}^{(n)}) \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \right) \wedge \cdots \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \right) \wedge U(\mathbf{v}^{(0)}) \tag{2}$$

```
   function find ( int a[ ] , int n ) {
1    c = 0;
2    while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3    if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4      ERROR;
   }
```

Fig. 1. Pseudo-code for the function `find`

are A_I^E-unsatisfiable for $n \geq 0$, where $\mathbf{v}^{(0)}, \ldots, \mathbf{v}^{(n)}$ are renamed copies of $\mathbf{v}$ (at time stamps $0, ..., n$). (Recall that, by assumption, $I(\mathbf{v}) := (pc = l_I)$ and $U(\mathbf{v}) := (pc = l_E)$.) If there exists a value of n for which (2) is A_I^E-satisfiable, then this means that there exists an execution of $\mathcal{S}$ starting in an initial state and ending in an error state.

Notice that, although terms of the form $\mathbf{a}[\mathbf{c}]$ *are not allowed* in formula (1), this is without loss of generality. In fact, any formula $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ containing such terms can be rewritten to $\exists \underline{j}(\underline{j} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{j}] \cdots))$ by using (fresh) existentially quantified variables $\underline{j}$ of sort `INDEX`. (Below, for the sake of brevity and only when discussing examples, we will write $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ in place of $\exists \underline{j}(\underline{j} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{j}] \cdots))$.) Interestingly, this syntactic restriction inherited from the specification language underlying MCMT inspired us an heuristic to abstract away counters dereferencing arrays and replace them with universally quantified variables of sort `INDEX` so as to synthesize universally quantified candidate invariants. Such heuristics, called *term abstraction*, will be described in Section 4.

Example 1. We illustrate how to encode the function `find` in Fig. 1 as an array-based system. The theory T_I is linear integer arithmetic (but notice that integer difference logic suffices), enriched with a constant `a.length`; the theory T_E has one sort constrained to be linear integer arithmetic enriched with a constant `n` (again, a very small fragment suffices) and one sort constrained to be the enumerated datatype theory of the set of locations $\{1, 2, 3, 4\}$ (where $l_I = 1$ and $l_E = 4$). The tuple $\mathbf{a}$ of array state variables contains only `a`, `c` is the unique counter, and `pc` is the only simple variable. The following five transitions specify the instructions of `find` (for simplicity, we omit mentioning identical updates):

$$\tau_1 \equiv \mathtt{pc} = 1 \wedge \mathtt{pc}' = 2 \wedge \mathtt{c}' = 0$$
$$\tau_2 \equiv \mathtt{pc} = 2 \wedge \mathtt{c} < \mathtt{a.length} \wedge \mathtt{a[c]} \neq \mathtt{n} \wedge \mathtt{c}' = \mathtt{c} + 1$$
$$\tau_3 \equiv \mathtt{pc} = 2 \wedge \mathtt{c} \geq \mathtt{a.length} \wedge \mathtt{pc}' = 3$$
$$\tau_4 \equiv \mathtt{pc} = 2 \wedge \mathtt{a[c]} = \mathtt{n} \wedge \mathtt{pc}' = 3$$
$$\tau_5 \equiv \mathtt{pc} = 3 \wedge \mathtt{c} \geq \mathtt{a.length} \wedge \exists x.\ (x \geq 0 \wedge x < \mathtt{a.length} \wedge \mathtt{a}[x] = \mathtt{n}) \wedge \mathtt{pc}' = 4.$$

The error location is unreachable iff $\forall x.\ (x \geq 0 \wedge x < \mathtt{a.length}) \Rightarrow \mathtt{a}[x] \neq \mathtt{n}$ holds when exiting `find`. ⊣

3 Unwinding Array-Based Systems

We adapt some of the notions in [21] so that they can be easily integrated in the framework of MCMT. If only simple variables are considered, our approach closely resembles that in [21]. The main difference is that our technique uses backward instead of forward reachability to explore the set of reachable states.

If ψ is a quantifier-free formula in which at most the index variables $\underline{i}$ occur, we denote by $\psi^{\exists}$ its existential (index) closure, namely the formula $\exists \underline{i}\, \psi$. The *matrix* of a guarded assignment in functional form $\tau(\mathbf{v}, \mathbf{v}')$ of the form (1) is the formula (1) itself without the existential prefix $\exists \underline{k}$; the *proper variables* of τ are the $\underline{k}$. Below, we shall feel free to apply bounded variables renamings to formulæ of the form (1) without explicit mention.

Definition 1. *A* labeled unwinding *of* $\mathcal{S} = \langle \mathbf{v}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$ *is a quadruple* (V, E, M_E, M_V)*, where* (V, E) *is a finite rooted tree (let* ε *be the root) and* M_E, M_V *are labeling functions for edges and vertices, respectively, such that:*

- (*i*) *for every* $v \in V$*, if* $v \neq \varepsilon$*, then* $M_V(v)$ *is a quantifier-free formula of the kind* $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ *such that* $M_V(v) \models_{A_I^E} pc = l$ *for some location* l*; otherwise* $M_V(\varepsilon)$ *is* $pc = l_E$*;*
- (*ii*) *for every* $(v, w) \in E$*,* $M_E(v, w)$ *is the matrix of some* $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$*; the proper variables of* τ *do not occur in* $M_V(w)$*; moreover, we have that* $M_V(w) \models_{A_I^E} pc = trg(\tau)$*, that* $M_V(v) \models_{A_I^E} pc = src(\tau)$*, and that*

$$M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}') \models_{A_I^E} M_V(v)(\mathbf{v}); \tag{3}$$

- (*iii*) *for each* $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$ *and every non-leaf vertex* $w \in V$ *such that* $M_V(w) \models_{A_I^E} pc = trg(\tau)$*, there exist* $v \in V$ *and* $(v, w) \in E$ *such that* $M_E(v, w)$ *is the matrix of* τ*.*

The intuition underlying the definition is that a vertex v in a labeled unwinding corresponds to a program location (*i*) and an edge (v, w) to the execution of a transition, whose source and target locations match with those of v and w, respectively (*ii* and *iii*). It is interesting to more closely analyze condition (3). To this end, we recall the definition of *pre-image* of a formula $K(\mathbf{v})$ with respect to a transition $\tau(\mathbf{v}, \mathbf{v}')$, which is one of the key ingredients of backward reachability (see, e.g., [10]): $Pre(\tau, K) := \exists \mathbf{v}'.\ (\tau(\mathbf{v}, \mathbf{v}') \wedge K(\mathbf{v}'))$. It is not difficult to see that condition (3) is equivalent to $\exists \mathbf{v}'.\ (M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}')) \models_{A_I^E} M_V(v)(\mathbf{v})$ which, in turn, implies $Pre(\tau, M_V(w)^{\exists}) \models_{A_I^E} M_V(v)^{\exists}$, if $M_E(v, w)$ is the matrix of τ. It is now clear that $M_V(v)^{\exists}$, i.e. the set of states associated to vertex v, *overapproximates* the set of states in the pre-image of $M_V(w)^{\exists}$ with respect to τ. Thus, the disjunction of the (existential index closure of the) formulæ labeling the nodes of an unwinding is an over-approximation of the set of backward reachable states and its negation (under suitable completeness conditions, see Definition 2 below) is an invariant of the system. A set C of vertexes in a labeled unwinding (V, E, M_E, M_V) *covers* a vertex $v \in V$ iff

$$M_V(v)^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}. \tag{4}$$

Definition 2. *The labeled unwinding* (V, E, M_E, M_V) *is* safe *iff for all* $v \in V$ *we have that if* $M_V(v) \models pc = l_I$*, then* $M_V(v)$ *is* A_I^E*-unsatisfiable. It is* complete *iff there exists a* covering*, i.e., a set of non-leaf vertexes* C *containing* ε *and such that for every* $v \in C$ *and* $(v', v) \in E$*, it happens that* C *covers* v'*.*

The reader familiar with [21] may have noticed that our notion of covering involves a set of vertexes rather than a single one as in [21]. Indeed, an efficient implementation of our notion is delicate and is discussed in Section 4. Here, we focus on abstract definitions which allow us to prove that safe and complete labeled unwindings can be seen as safety certificates for array-based systems.

Theorem 1. *An array-based system is safe if there exists a safe and complete labeled unwinding for it.*

3.1 Lazy Abstraction with Interpolants in MCMT

We are left with the problem of computing labeled unwindings and checking for their safety and completeness. Similarly to [21], we design a possibly non-terminating procedure, called Unwind, that, given an array-based system $\mathcal{S}$, computes a sequence of (increasingly larger) labeled unwindings. The initial labeled unwinding of $\mathcal{S}$ is the tree containing just the root labeled by $pc = l_E$. Unwind uses two sub-procedures, called Expand and Refine, which can be non-deterministically applied to a labeled unwinding to obtain a new one, if possible. When Refine is applicable but fails, $\mathcal{S}$ is unsafe. If none of the two procedures applies, then the current labeled unwinding is safe and complete; thus $\mathcal{S}$ is safe by Theorem 1.

The core of our procedure is the sub-procedure Refine that performs refinement of labelings in presence of spurious unsafety traces. The distinguishing feature of our method is that, *despite the fact that we use quantified formulæ to represent sets of states and transitions, for refinement we need only quantifier-free interpolation* (even in a restricted form). Technically, this is made possible because the formulæ describing potentially unsafe traces are equisatisfiable with quantifier-free formulæ obtained by a restricted form of instantiation (see below for the technical details). We now describe the two sub-procedures.

Let (V, E, M_E, M_V) be the current labeled unwinding of $\mathcal{S}$. From now on, we assume that *the initial location is not a target location, the error location is not a source location*, and that initial and error locations are *the only locations that are not both a source and a target location.*

Expand. The applicability condition is that (V, E, M_E, M_V) is not complete and that there exists a leaf vertex v whose location is such that $M_V(v) \not\models_{A_I^E} pc = l_I$. By Definition 1($i$), we must have $M_V(v) \models_{A_I^E} pc = l$ for some $l \neq l_I$. For each transition $\tau \in \{\tau_h\}_h$ whose target is l, add a new leaf w_τ, label it by $pc = src(\tau)$, add the edge (w_τ, v) to the current tree, and label it by τ. ⊣

Refine. The applicability condition is that (V, E, M_E, M_V) is not complete and there exists a vertex $v \in V$ whose location is l_I and it is such that $M_V(v)$ is

A_I^E-satisfiable. Consider the path $v = v_0 \to v_1 \to \cdots \to v_m = \varepsilon$ from v to the root and let $\tau_1, \ldots, \tau_m$ be the transitions labeling the edges from left to right. If

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \tag{5}$$

is A_I^E-satisfiable (notice that this is decidable, see [1] for details), then fail and report the unsafety of $\mathcal{S}$. Otherwise, update the formulæ labeling $v_0, \ldots, v_m$ by using interpolants as follows. By recalling (1), rewrite (5) as

$$\bigwedge_{k=1}^{m} \exists \underline{i}_k \left(\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \ \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right) \tag{6}$$

which, by Skolemizing existentially quantified variables, is transformed to the equi-satisfiable formula (below, by abuse of notation, we consider the symbols in $\underline{i}_k$ as Skolem constants):

$$\bigwedge_{k=1}^{m} \left(\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \ \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right) \tag{7}$$

Now, observe that $\mathbf{a}^{(k)} = \lambda j\, G_k(\ldots)$ is equivalent to $\forall j.\ \mathbf{a}^{(k)}[j] = G_k(\ldots j \ldots)$ and *instantiate the variable j with the Skolem constants in $\underline{i}_{k+1}, ..., \underline{i}_m$* to derive

$$\bigwedge_{k=1}^{m} \left(\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \bigwedge_{j \in \underline{i}_{k+1}, \ldots, \underline{i}_m} \mathbf{a}^{(k)}[j] = G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \ \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right). \tag{8}$$

Formula (8) is A_I^E-equisatisfiable to (7), see [1] for a proof. Now, (5) was supposed to be A_I^E-unsatisfiable, hence so are (6), (7) and finally (8). Let us abbreviate the k-th conjunct in the big conjunction (8) as

$$\tilde{\tau}_k(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], \mathbf{c}^{(k-1)}, \mathbf{c}^{(k)}, \mathbf{d}^{(k-1)}, \mathbf{d}^{(k)}) \ , \tag{9}$$

so that (8) is written as $\tilde{\tau}_1 \wedge \cdots \wedge \tilde{\tau}_m$. Finally, let

$$\psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_{k+1}], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}) \tag{10}$$

be the (quantifier-free interpolants) computed in (8) from right-to-left such that

$$\psi_0 \equiv \bot, \qquad \psi_m \equiv \top, \tag{11}$$

$$\begin{array}{r} \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], \mathbf{c}^{(k)}, \mathbf{d}^{(k)}) \wedge \tilde{\tau}_k \models_{A_I^E} \\ \psi_{k-1}(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}), \end{array} \tag{12}$$

and update the label of v_k as follows:

$$M_V(v_k) \;\equiv\; M_V(v_k) \wedge \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_k], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}). \tag{13}$$

Notice that, since the matrix of τ_k entails $\tilde{\tau}_k$, the condition (3) is preserved and the vertex $v = v_0$ is labeled by an A_I^E-unsatisfiable formula. ⊣

Both Expand and Refine prescribe to establish if the current unwinding is complete. According to Definition 2, this requires to guess a sub-set C of the set of vertexes in the unwinding and check if C covers v', for every $v \in C$ and $(v', v) \in E$. In turn, this may be reduced to repeatedly check the A_I^E-unsatisfiability of an $\exists^{A,I}\forall^I$-sentence (recall the definition in Section 2), reasoning by refutation from (4). These satisfiability checks are decidable under suitable conditions [10], which will be briefly recalled in Section 3.2 when discussing the completeness of our technique. However, even when the conditions for decidability are not satisfied, it is still possible to use sound but incomplete algorithms which preserves the soundness of Unwind. Concerning Refine, notice that it is possible to predict the numbers e_k of (implicitly existentially quantified) index variables occurring in the formulæ labeling the vertex v_k of a path of the form $v_0 \to \cdots \to v_m = \varepsilon$ by simply counting the existentially quantified index variables in $\tau_{k+1} \wedge \cdots \wedge \tau_m$ from (5). The number of index variables that will occur in the formula labeling v_k after the update (13) is bounded by e_k, because it is derived from the interpolants computed along the path considered above. On the other hand, the number of index variables labeling the leaves may grow very quickly, thereby posing a crucial problem for implementation (e.g., when instantiating universally quantified variables in covering tests). Fortunately, heuristics [9, 11] designed to reduce the number of index variables in pre-images developed for the backward reachability procedure of mcmt can also be put to productive use in the main loop of Unwind.

The third observation on Refine concerns the computation of the interpolants. An easy way to derive ψ_{k-1} from ψ_k would be to use the pre-image of ψ_{k-1} with respect to the transition labeling the edge connecting the vertexes whose label is to be updated by ψ_{k-1} and ψ_k. However, for Unwind to be truly an *abstraction*-based procedure, we need to compute interpolants which do not necessarily reduce to the precise preimage. This can be done by combining the available interpolation algorithms for T_I and T_E. Unrestricted combination is not always possible in general (there are negative results in the literature showing e.g. that the addition of free function symbols can destroy quantifier-free interpolation [2]); however, because of the form (9) of the above formulæ $\tilde{\tau}_k$, it follows that whenever Unwind needs to compute an interpolant for an unsatisfiable quantifier-free formula $\psi_1 \wedge \psi_2$, the formulæ ψ_1, ψ_2 satisfy the hypotheses of the following positive result:

Theorem 2. *Suppose that $\psi_1 \wedge \psi_2$ is an A_I^E-unsatisfiable quantifier-free formula such that all variables of sort* `INDEX` *occurring in ψ_2 under the scope of the read operator $\_[\_]$ occur also in ψ_1. Then, there exists a quantifier-free formula ψ_0 such that: (i) $\psi_2 \models_{A_I^E} \psi_0$; (ii) $\psi_0 \wedge \psi_1$ is A_I^E-unsatisfiable; (iii) all free variables occurring in ψ_0 occur both in ψ_1 and ψ_2.*

The soundness of UNWIND is guaranteed by the following result.

Theorem 3. *If neither* EXPAND *nor* REFINE *can be applied to a labeled unwinding $P = (V, E, M_E, M_V)$, then P is safe and complete.*

Example 2. We briefly discuss how UNWIND is applied to the array-based system of Example 1. Fig. 2 (without boxed literals) reports an unwinding that, starting from the error location ($M_V(\epsilon) \models \mathtt{pc} = 4$) reaches the initial location ($M_V(v_{25}) \models \mathtt{pc} = 1$). An infeasible trace depicted in Fig. 2. The counterexample associated to the trace is the following (for the sake of conciseness, we list only the variables changing their values):

$$
\begin{aligned}
&\mathtt{pc}^{(4)} = 1 \wedge \\
&\mathtt{pc}^{(4)} = 1 \wedge \mathtt{pc}^{(3)} = 2 \wedge \mathtt{c}^{(3)} = 0 \wedge \\
&\mathtt{pc}^{(3)} = 2 \wedge \mathtt{pc}^{(2)} = 2 \wedge \mathtt{a.length} > \mathtt{c}^{(3)} \wedge \mathtt{c}^{(2)} = \mathtt{c}^{(3)} + 1 \wedge i_1 = \mathtt{c}^{(3)} \wedge \mathtt{a}^{(3)}[i_1] \neq \mathtt{n} \wedge \\
&\mathtt{pc}^{(2)} = 2 \wedge \mathtt{pc}^{(1)} = 3 \wedge \mathtt{a.length} \leq \mathtt{c}^{(2)} \wedge \\
&\mathtt{pc}^{(1)} = 3 \wedge \mathtt{pc}^{(0)} = 4 \wedge \mathtt{a.length} \leq \mathtt{c}^{(1)} \wedge \mathtt{a}^{(1)}[i_0] = \mathtt{n} \wedge \mathtt{a.length} > i_0 \wedge \\
&\mathtt{pc}^{(0)} = 4
\end{aligned}
$$

The counterexample is unsatisfiable and it is thus infeasible in the concrete system. A set of interpolants computed from the trace above contains

$$
\begin{aligned}
&\psi_0 \equiv \bot, \quad \psi_1 \equiv \bot, \quad \psi_2 \equiv i_0 \leq \mathtt{c} \wedge i_0 \geq 0, \quad \psi_3 \equiv i_0 \leq \mathtt{c} \wedge i_0 \geq 0, \\
&\psi_4 \equiv i_0 \leq \mathtt{a.length} \wedge i_0 \geq 0, \quad \psi_5 \equiv i_0 \geq 0, \text{ and } \psi_6 \equiv \top.
\end{aligned}
$$

According to (13), the refinement of the infeasible trace is done by adding each interpolant to the corresponding vertex in the unwinding (see the boxed literals in Fig. 2). UNWIND is then able to generate the invariant

$$(\mathtt{pc} = 3 \wedge \mathtt{c} > 0 \wedge \mathtt{a.length} \geq 1) \Rightarrow \forall i.\, ((i < \mathtt{c} \wedge i \leq \mathtt{a.length}) \Rightarrow \mathtt{a}[i] \neq \mathtt{n})$$

as the negation of the label of v_1, which states that if the the loop is executed at least once (antecedent of the main implication), then at every position i (up to $\mathtt{c}$) of the array is stored a value distinct from $\mathtt{n}$. Notice that the predicates $\mathtt{c} > 0$, $\mathtt{a.length} \geq 1$ and $i < \mathtt{c}$ (where i is an universally quantified variable) are new and have been generated by the interpolation algorithm. ⊣

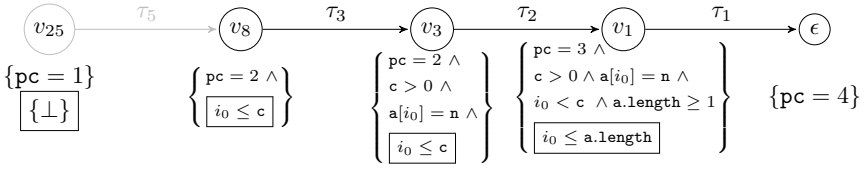

Fig. 2. Counterexample for Example 2. Boxed labels were added by refinement.

3.2 Completeness and Termination

The completeness of UNWIND depends on the decidability of checking whether a labeled unwinding is complete according to Definition 2. We have already argued (see first observation after the description of REFINE in Section 3.1) that this can be reduced to the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences.

Theorem 4 ([10]). *If there are no function symbols in the signature Σ_I of T_I and the class $\mathcal{C}_I$ of models of T_I is closed under substructures, then the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences is decidable.*

In [10], the proof of this result[1] is constructive by showing a procedure which first instantiates the universally quantified index variables with the existentially quantified index variables (considered as Skolem constants) of the sentence in all possible ways and then invokes a combination (*à la* Nelson-Oppen) of the available decision procedures for the $SMT(T_I)$- and $SMT(T_E)$-problems (recall the assumptions in Section 2). The procedure is still sound but incomplete when the assumptions on T_I in Theorem 4 do not hold. For efficiency, heuristics [9] have been designed to reduce the number of possible instantiations.

Conditions for the termination of UNWIND are much more restrictive. First, a fair strategy must be used to apply EXPAND and REFINE. Formally, a strategy is *fair* if it does not indefinitely delay the application of one of the two procedures and does not apply REFINE infinitely many times to the label of the same vertex. Notice that the latter holds if there are no infinitely many non-equivalent formulæ of the form $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ for a given $\underline{i}$ or, alternatively, if a refinement based on the computation of interpolants through the precise preimage is eventually applied when repeatedly refining a node. The second condition (since adopting a fair strategy alone is not sufficient) for termination concerns also the theory T_E. To formally state such conditions, we need to adapt some notions from [3, 10]. A *wqo-theory* is a theory $T = (\Sigma, \mathcal{C})$ such that $\mathcal{C}$ is closed under substructures and finitely generated models of T are a well-quasi-order with respect to the relation $\preceq$ that holds between $\mathcal{M}_1$ and $\mathcal{M}_2$ whenever $\mathcal{M}_1$ embeds into $\mathcal{M}_2$.

Theorem 5. *Let $\mathcal{S} = \langle \mathbf{v}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$ be an array-based system for T_I, T_E. Suppose that T_I satisfies the hypotheses of Theorem 4 and that the theory obtained from $T_I \cup T_E$ by adding it the symbols $\mathbf{v}$ (seen as free constants of appropriate sorts) is a wqo theory. Then,* UNWIND *terminates when applied to $\mathcal{S}$ with a fair strategy.*

As a consequence, UNWIND behaves as a decision procedure for those classes of array-based systems satisfying the conditions of Theorem 5. This is the case, for example, of broadcast protocols and lossy channels systems (see [3, 10] for details). A similar result for broadcast protocols is given in [7] within forward reachability.

[1] Although in this paper, we have also counters and simple variables in the definition of array-based systems, that were not considered in [10], this does not interfere with the correctness of the algorithm in [10] which can be easily extended to cope with them.

4 Implementation and Experiments

We have implemented UNWIND on top of a re-engineered version of MCMT[2] ; the following two heuristics are the key ingredients for its practical applicability.

Term Abstraction. While experimenting with our prototype, we realized that available interpolating procedures seldom permit to refine the abstraction in the "right" way because some terms are not eliminated. This dramatically decreases performances or, even worse, prevents to find the inductive invariant, when it exists. To alleviate this problem, the goal of the *term abstraction* heuristic is to compute (if possible) an interpolant where a certain term t does not occur. As briefly explained in Section 2, the term t is usually some counter which should be eliminated to synthesize (candidate) invariants involving universally quantified index variables, even when the problem specification mentions no quantifiers. Term abstraction proceeds as follows. Given an A_I^E-unsatisfiable formula of the form $\psi_1 \wedge \psi_2$, if $\psi_1(c_1/t) \wedge \psi_2(c_2/t)$ is A_I^E-unsatisfiable, for c_1 and c_2 fresh constants, then term abstraction returns the interpolant of $\psi_1(c_1/t) \wedge \psi_2(c_2/t)$, computed by running the available interpolation procedure. Otherwise, term abstraction returns the interpolant of the original formula $\psi_1 \wedge \psi_2$. Our prototype tool automatically extracts from the problem specification a list of "relevant" terms, called *term abstraction list*, which contains candidates for the term abstraction heuristic (alternatively, the user can provide such a list).

Covering Strategy. We implemented an additional procedure REDUCE which is to be interleaved with EXPAND so as to reduce as much as possible the invocations to the latter. REDUCE checks, given a vertex v of the unwinding and a set $\tilde{V}$ of nodes such that every $v_i \in \tilde{V}$ is not covered, whether $M_V(v) \models_{A_I^E} \bigvee_{v_i \in \tilde{V}} M_V(v_i)$, i.e., it checks if v is covered by a disjunction of vertexes that share the same value of the program counter. In addition we allow leaves to be covered by younger vertices (while in [21] a vertex can be covered only by one older vertex). REDUCE agrees with the notion of covering introduced in Definition 2. Moreover, before expanding a leaf v, we check if there is at least one v's ancestor u such that u is covered by its ancestors. If so, a descendant of u can neither be expanded nor cover other vertices as long as u is covered.

Table 1 reports the results of our experiments (run on an Intel i7 @2.66 GHz, equipped with 4GB of RAM and running OSX 10.7). Our benchmark set includes simple programs over arrays, e.g., initialization of all elements to 0, copy of one array into another, etc. They have been taken from other papers (e.g., [17, 19, 23]), or from standard textbooks on algorithms. All benchmarks diverge if no abstraction is provided.

The last two benchmarks in Table 1 are trickier to verify, as they feature two loops. In "init and test", there are two loops in sequence and the safety condition consists in reaching an error location. Although the property does not contain quantifiers, the inductive invariant of the program does need quantifiers. The

[2] The executable of the prototype tool and all the input files can be downloaded at `http://www.oprover.org/mcmt_abstraction.html`.

Table 1. Table reports: total verification time, number of nodes of the final unwinding, number of calls to the SMT-solver, number of CEGAR iterations, final safety result.

Benchmark	Description	Time (s)	Nodes	SMT-calls	Iter.	Result
find (v1)	Find an element	0.3	5	192	3	SAFE
find (v2)	(as above, alternative encoding)	0.07	5	48	1	SAFE
initialization	Initialize all elements to 0	0.1	5	96	1	SAFE
max in array	Find max element	0.9	72	1192	8	SAFE
partition	Partition an array	0.08	20	62	0	SAFE
strcmp	Compare arrays	0.4	14	329	4	SAFE
strcpy	Copy arrays	0.03	3	15	0	SAFE
vararg	Search for end of arguments	0.03	5	17	0	SAFE
integers	Numerical property	0.02	5	19	0	SAFE
init and test	Init. to 0 and tests	0.3	27	375	3	SAFE
binary sort	Sorting with binary search	0.3	48	457	2	SAFE

methodology applied by our tool to introduce extra quantifers is the following: first, recall from Section 2 that sentences like $\psi(..\mathbf{a}[\mathbf{c}]..)$ are written as $\exists \underline{i}(\underline{i} = \mathbf{c} \wedge \psi(..\mathbf{a}[\underline{i}]..))$, then term abstraction can get rid of (some of) the $\mathbf{c}$ thus letting (some of) the $\underline{i}$ be genuine new quantifiers. As for nested loops, "binary sort" is an encoding of the sorting algorithm based on binary search.

To test the flexibility of our approach, we run the prototype on some randomly generated problems taken from those shipped with the distribution of the ARMC model-checker (`http://www.mpi-sws.org/~rybal/armc/`). They consists of safety properties of numerical programs without arrays. Our tool can solve 22 out of 28 benchmarks with abstraction, but only 9 without using it. For those benchmarks that could be solved even without abstraction, the overhead of abstraction is generally negligible.

5 Conclusion

We have described UNWIND, a verification procedure for safety properties based on the combination of the backward reachability of MCMT and lazy-abstraction with interpolants. Lazy-abstraction is enabled to handle (unbounded) arrays while MCMT is now capable to cope with sequential programs in a uniform way by using abstraction and refinement. Our experiments show that the improved version of MCMT is able to prove safety properties in no time for common-use programs over arrays. As future work, we plan to tune the abstraction and refinement mechanisms to other classes of systems, such as distributed algorithms.

Acknowledgements. The work of the first author was supported by the Hasler Foundation under project 09047 and that of the fourth author was partially supported by the "SIAM" project founded by Provincia Autonoma di Trento in the context of the "team 2009 - Incoming" COFUND action of the European Commission (FP7).

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy Abstraction with Interpolants for Arrays. extended version, http://homes.dsi.unimi.it/~ghilardi/allegati/ABGRS_LPAR.pdf
2. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
3. Carioni, A., Ghilardi, S., Ranise, S.: Automated Termination in Model Checking Modulo Theories. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 110–124. Springer, Heidelberg (2011)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
5. Cousot, P., Cousot, R., Logozzo, F.: A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In: POPL (2011)
6. Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
7. Dimitrova, R., Podelski, A.: Is Lazy Abstraction a Decision Procedure for Broadcast Protocols? In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 98–111. Springer, Heidelberg (2008)
8. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)
9. Ghilardi, S., Ranise, S.: Model Checking Modulo Theory at work: the integration of Yices in MCMT. In: AFM (2009)
10. Ghilardi, S., Ranise, S.: Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. LMCS 6(4) (2010)
11. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)
12. Gopan, D., Reps, T., Sagiv, M.: A Framework for Numeric Analysis of Array Operations. In: POPL 2005, pp. 338–350 (2005)
13. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
14. Halbwachs, N., Mathias, P.: Discovering Properties about Arrays in Simple Programs. In: PLDI 2008, pp. 339–348 (2008)
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: POPL, pp. 232–244 (2004)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: POPL, pp. 58–70 (2002)
17. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
18. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for Data Structures. In: SIGSOFT 2006/FSE-14, pp. 105–116 (2006)
19. Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 199–213. Springer, Heidelberg (2009)
20. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)

21. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
22. Ranise, S., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2006), http://www.SMT-LIB.org
23. Seghir, M.N., Podelski, A., Wies, T.: Abstraction Refinement for Quantified Array Assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
24. Lahiri, S., Bryant, R.: Predicate Abstraction with Indexed Predicates. TOCL 9(1) (2007)
25. Srivastava, S., Gulwani, S.: Program Verification using Templates over Predicate Abstraction. In: PLDI (2009)