Form Methods Syst Des (2014) 45:63–109
DOI 10.1007/s10703-014-0209-9

An extension of lazy abstraction with interpolation for programs with arrays

Francesco Alberti · Roberto Bruttomesso · Silvio Ghilardi · Silvio Ranise · Natasha Sharygina

Published online: 21 May 2014
© Springer Science+Business Media New York 2014

Abstract Lazy abstraction with interpolation-based refinement has been shown to be a powerful technique for verifying imperative programs. In presence of arrays, however, the method suffers from an intrinsic limitation, due to the fact that invariants needed for verification usually contain universally quantified variables, which are not present in program specifications. In this work we present an extension of the interpolation-based lazy abstraction framework 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 supports reasoning about arrays. The new approach has been implemented in a tool, called SAFARI, which has been validated on a wide range of benchmarks. We show by means of experiments that our approach can synthesize and prove universally quantified properties over arrays in a completely automatic fashion.

Keywords SMT · Model checking · Lazy abstraction · Array programs

This paper combines and extends materials previously published in [4,5].

F. Alberti (✉) · N. Sharygina
Faculty of Informatics, University of Lugano, via G. Buffi, 13, 6904 Lugano, Switzerland
e-mail: francesco.alberti@usi.ch

N. Sharygina
e-mail: natasha.sharygina@usi.ch

R. Bruttomesso
Atrenta Advanced R&D of Grenoble, Grenoble, France
e-mail: roberto@atrenta.com

S. Ghilardi
Department of Mathematics, Università degli Studi di Milano,
via C. Saldini, 50, 20133 Milan, Italy
e-mail: silvio.ghilardi@unimi.it

S. Ranise
Security and Trust Unit, Fondazione Bruno Kessler,
via Sommarive, 18, 38123 Trento, Italy
e-mail: ranise@fbk.eu

1 Introduction

Automated verification of software is a long standing scientific challenge that, in the last decades, received a lot of attention. The goal of software verification approaches is to automatically infer when programs exhibit undesired behaviors, that violate annotations in the code (e.g., invariants and post-conditions). If every execution of a program reaching annotations does not violate them (i.e. the program shows no undesired behavior), the program is said to be *safe*. Since the problem is undecidable [65], complete and fully automatic techniques cannot exist and the programmer must manually add annotations for the verification to be successful. Verification techniques that reduce this burden and increase the level of automation are thus highly desirable. In this respect, one of the most promising techniques is the use of Model Checking to automatically explore the state-space of a program, and checking it with respect to user-specified properties.

Model Checking has been shown quite successful in the analysis of large but finite state systems (e.g., hardware designs). For software, because of the presence of data structures ranging over infinite domains (e.g., integers) and dynamic memory handling, the challenge is to adapt Model Checking to handle infinite state spaces. In this respect, Abstraction [50] and its refinements, CounterExample Guided Abstraction Refinement (CEGAR) [26] and Lazy Abstraction [54], have been shown successful and are nowadays employed in many state-of-the-art software verification tools. Roughly, Abstraction consists of constructing an abstract program P^a from a given program P in such a way that the set of possible executions of P is a sub-set of those of P^a; the vice-versa does not hold. Thus, any safety property that holds for the executions of P^a also holds for those of P. If there exists an execution of P^a not satisfying the property, we cannot conclude that there exists an execution of P violating the property and P^a must be refined.

The idea underlying CEGAR is to iteratively refine abstractions by applying the following steps. After building an abstraction P^a of a program P, Model Checking is applied to P^a. If P^a is found to be safe, then also the safety of P is reported. Otherwise, it is checked if an execution of P^a violating the property is an execution of P: if this is the case, the program P is declared to be unsafe. If the execution of P^a corresponds to no execution of P, it is said to be a *spurious counter-example* and used to *refine* P^a to a new abstract program that does not admit the counter-example as one of its execution.

In many approaches to CEGAR (e.g., [11,13]), Abstraction is performed with respect to a given set S of predicates over the variables of the program P. An abstract state (of P^a) is created by invoking a theorem prover, usually a Satisfiability Modulo Theories (SMT) solver, that computes a Boolean combination of the predicates in S that over-approximates a concrete state (of P). Then, refinement extends S to S' by adding new predicates so that the Boolean combinations of the predicates in S' allow for a better (over-)approximation of the concrete states.

One of the most difficult problems in CEGAR is to identify, during the refinement phase, appropriate criteria to discover new predicates that provide better abstractions. In this respect, Lazy Abstraction is particularly interesting since it is capable of refining the abstraction by using different degrees of precision for different parts of the program. The idea is to use a control-flow graph (see Figs. 1, 2 for an example of a program and the associated control-flow graph) to keep track of how the program locations are traversed and of predicates to represent the data-flow and the program annotations. Lazy Abstraction is based on a CEGAR loop in which the control-flow graph is iteratively unwound and the data in the newly explored locations is over-approximated. When reaching a location in which a property is violated and the execution is a counter-example, the abstraction along the path is (locally) refined. Since

```
procedure Running( ) {
    i = 0;
    while ( i < L ) {
        if ( a[i] ≥ 0 ) b[i] = true;
        else b[i] = false;
        i = i + 1;
    }
    f = true; i = 0;
    while ( i < L ) {
        if ( a[i] ≥ 0 ∧ ¬b[i] ) f = false;
        if ( a[i] < 0 ∧ b[i] ) f = false;
        i = i + 1;
    }
    assert ( f );
}
```

Fig. 1 The procedure Running

in several approaches (see, e.g., [53,67]) data structures are formalized as theories—e.g., the theory of arrays [66]—SMT solvers are not only used to compute abstractions (as Boolean combinations of predicates) but also in the refinement phase to discover new predicates by computing (Craig) interpolants [31].

Given a pair (A, B) of inconsistent formulas, an interpolant is a formula I built over the common vocabulary of A and B, entailed by A and unsatisfiable when put in conjunction with B. For refinement, the interpolant I may contain additional predicates and can be used to eliminate the part B of the counter-example that does not correspond to any execution of the concrete program while leaving A untouched. In this sense, the abstract program is refined locally by eliminating only part of the abstraction (namely, B) that gives rise to a counter-example [67]. Interpolation capabilities are available in several SMT solvers; e.g., MathSAT [24], OpenSMT [21], and Z3 [33].

Verification tools based on CEGAR or Lazy Abstraction have been successfully applied to certain classes of programs, e.g., device drivers [11]. However, the annotations of such programs involve only simple properties about the data-flow with a limited interplay with the control-flow. When used to verify programs manipulating sophisticated data-structures—such as arrays, CEGAR and Lazy Abstraction show some limitations. One of the most important reason for the the limited success of Lazy Abstraction on programs manipulating arrays is the fact that program annotations often require (universal) quantification. To illustrate, consider the procedure Running in Fig. 1.

The first loop of the procedure initializes the array b according to the content of the array a such that, at the end of the loop, the following assertion holds:

$$\text{for every index } i \text{ in the range } 0 \ldots L,\ b[i] = \mathsf{true} \text{ iff } a[i] \geq 0. \quad (1)$$

The second loop of the procedure sets the Boolean flag f to false if a position in the array a contradicting assertion (1) is found. The program is clearly safe, i.e. after the second loop f is always true for any execution of the procedure, but in order to prove it, we need the quantified assertion (1).

The main contribution of this paper is a new verification approach that overcomes the above problems by redefining Lazy Abstraction with Interpolation-based refinement and makes it

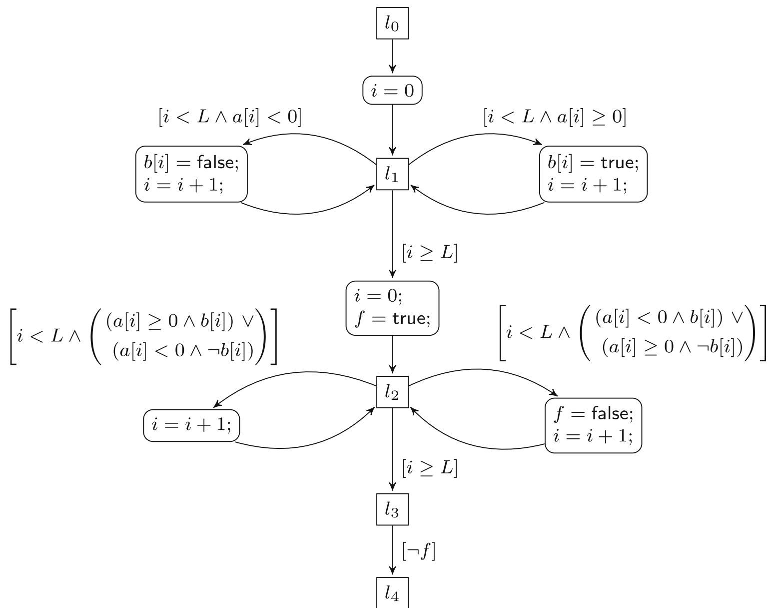

Fig. 2 The control-flow graph of Running

possible to reason about arrays of unknown length. Our technique is developed in the Model Checking Modulo Theory (MCMT) approach [46,47] in which verification is performed by a symbolic backward reachability procedure. Certain classes of formulas represent sets of backward reachable states and fix-point checks are reduced to logical problems that SMT solvers are able to tackle, once extended with suitable quantifier instantiation techniques. The MCMT approach has been successfully exploited for the verification of parameterized (distributed) systems (see, e.g., [6,7,46]) but it fails when applied to the verification of imperative programs because of the lack of suitable abstraction-refinement techniques. To overcome this problem, we extend the backward reachability procedure of MCMT with a carefully designed interpolation-based abstraction refinement technique capable of generating the quantified predicates required for the synthesis of the inductive invariants, needed to establish the safety of programs manipulating arrays. For this, we need to address the following technical challenges:

(i) Refinement must be able to deal with quantified formulas, i.e. it is necessary to discover new predicates possibly containing quantifiers. Indeed, this is a much more difficult task than finding predicates that are equivalent to quantifier-free formulas as it is the case in many Lazy Abstraction approaches focusing on scalar data structures (see, e.g., [53]). To understand the problem, consider the procedure Running in Fig. 1 and recall that (1) is the invariant required for proving its safety. Refinement should be able to generate it as a single predicate, because of the universally quantified variable i; definitely a non-trivial task.

(ii) Satisfiability of formulas representing (abstract) counter-examples must be decidable. This is key to be able to automatically detect when the abstract program requires to be refined. Unfortunately, the situation is complicated by the fact that interpolation-based refinement may introduce extra quantifiers in the new predicates because, as shown

in [60], the "standard" theory of arrays [66] does not admit quantifier-free interpolation. As a consequence, refinement needs to be carefully controlled since the introduction of quantifiers may give rise to formulas containing alternations of quantifiers. This easily leads to the undecidability of the satisfiability of the formulas representing sets of backward reachable states.

(iii) The implementation of interpolation-based refinement procedures is delicate because the "quality" of the generated interpolants may generate too many refinements, thereby degrading performances unacceptably, or even worse making the procedure diverging. This is so because a pair (A, B) of inconsistent formulas may admit several (even infinitely many) interpolants and choosing the one that is "the best" with respect to refinement is an undecidable problem.
To illustrate the problem, consider again the procedure Running in Fig. 1. An interpolation-based refinement procedure may generate the sequence $b[0] \leftrightarrow a[0] \geq 0$, $b[1] \leftrightarrow a[1] \geq 0$, …of infinitely many (quantifier-free) predicates. After each iteration of refinement, the conjunction of these predicates offers only an approximation of the quantified assertion (1) needed to prove the safety of Running and the Lazy Abstraction procedure diverges because of the infinite (increasingly precise) sequence of approximations.
Heuristics (see, e.g., [58]) to tune the generation of interpolants and avoid divergence are crucial for efficient implementations.

Our solution tackles the aforementioned challenges by exploiting the following ideas. We will work with *flattened formulas*, i.e., formulas where array variables are dereferenced only by existentially quantified variables. Thus, a formula of the kind $\phi(a[i], \ldots)$ (where i is a constant or more generally a term) is first rewritten as $\exists x\ (x = i \wedge \phi(a[x], \ldots))$. During consistency tests, the existentially quantified variable x is skolemized away, so that consistency tests are made with quantifier-free formulas. Interpolants search is performed at quantifier-free level and it is guided by the so-called *term abstraction* technique: the aim of term abstraction is to try, as much as possible, to get interpolants not containing certain undesired terms (the list of such terms can be either supplied by the user or synthesized by the tool according to some general heuristics). Now, if the interpolant abstracts away the constant i from $x = i \wedge \phi(a[x], \ldots)$, when de-Skolemization reintroduces the variable x, this x will be a genuine existentially quantified variable. In fact, the negation of the resulting formula will be part of the universally quantified invariant we are looking for (recall that backward search produces, when successful, existentially quantified formulas whose negations turn out to be invariants).

Besides presenting theoretical solutions to the first two challenges above, the paper describes also an efficient implementation in a model checker called SAFARI [1] — "SMT-Based Abstraction For Arrays with Interpolants". SAFARI is built on top of the OpenSMT SMT-Solver. To show the practical viability of our technique, we successfully evaluated SAFARI on several programs handling arrays taken from the recent literature. In summary, the contributions of this paper are:

- a framework for abstraction-refinement with quantified predicates;
- a quantifier-free interpolation algorithm for a relevant class of formulas with array variables;
- a heuristic to tune interpolation procedures and help convergence of abstraction-refinement procedures;

[1] Available at http://verify.inf.usi.ch/content/safari.

– a tool, SAFARI, designed for proving safety properties of programs with arrays of unbounded length.

Some of the material in this paper has already been published in preliminary form in [4,5]. This paper not only gives a comprehensive account of our approach to Lazy Abstraction with interpolation for programs manipulating arrays by presenting all the proofs of our results but it also describes in detail the architecture and the heuristics of SAFARI, along with a thorough experimental evaluation on challenging benchmarks.

Organization of the paper. To make the paper self-contained, in Sect. 2 we introduce some formal preliminaries and selected notions from [4]. Section 3 recalls basic notions about the class of transition systems manipulating arrays used in the paper and shows how sequential programs can be specified using this model. Section 4 introduces the new lazy abstraction framework. The main procedure underlying our approach, UNWIND is presented in Sect. 5, its soundness, completeness and termination are discussed in Sect. 6. Heuristics implemented in the tool SAFARI are presented in Sect. 7, followed by experiments that are presented in Sect. 8. Section 9 discusses related work. We conclude in Sect. 10.

2 Formal preliminaries

We assume the usual syntactic (e.g., signature, variable, term, atom, literal, and formula) and semantic (e.g., structure, sub-structure, assignment, truth, satisfiability, and validity) notions of many-sorted first-order logic with equality (see, e.g., [39]). The equality symbol $=$ is included in all signatures considered henceforth. 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 $\mathbf{a}, \mathbf{v}, \ldots$. Bold face letters are used for tuples of variables which are kept fixed for largest parts of the paper. With $E(\underline{x})$ we denote that the syntactic expression (term, formula, tuple of terms or of formulas) E contains at most the free variables in the tuple $\underline{x}$. If $\underline{t} = t_1, \ldots, t_n$ and $\underline{s} = s_1, \ldots, s_n$ are tuples of terms with the same length, we abbreviate $\bigwedge_{i=1}^{n}(t_i = s_i)$ with $\underline{t} = \underline{s}$.

According to [72], 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. Given a Σ-structure $\mathcal{M}$, we denote by $S^{\mathcal{M}}, f^{\mathcal{M}}, P^{\mathcal{M}}, \ldots$ the interpretation in $\mathcal{M}$ of the sort S, the function symbol f, the predicate symbol P, etc. If Σ_0 is a sub-signature of Σ, the structure $\mathcal{M}_{|\Sigma_0}$ results from $\mathcal{M}$ by forgetting about the interpretation of the sort, function, and predicate symbols that are not in Σ_0 and $\mathcal{M}_{|\Sigma_0}$ is called the *reduct* of $\mathcal{M}$ to Σ_0.

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, when φ is a sentence and no free variable assignment is needed, we write $\mathcal{M} \models \varphi$); it is T-valid (in symbols, $T \models \varphi$) if its negation is T-unsatisfiable. Two formulas φ_1 and φ_2 are T-equisatisfiable iff if there exist a model of T and a free variable assignment in which φ_1 holds, then there exist a model of T and a free variable assignment in which also φ_2 holds, and vice-versa; they 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 $\psi_1 \to \psi_2$ is T-valid. The satisfiability modulo the theory T ($SMT(T)$) problem amounts to establishing the T-satisfiability of quantifier-free Σ-formulas.

A theory T has *quantifier-free interpolation* iff there exists an algorithm that, given two quantifier free formulas 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 ψ occur in θ.

For the rest of the paper, two theories will be particular relevant. The former is the mono-sorted theory of an *enumerated data-type* $\{e_1, \ldots, e_n\}$ in which the interpretation of the sort is a set of cardinality n, the signature of the theory contains only n constant symbols that are interpreted as the n distinct elements in the interpretation of the sort. Indeed, the SMT problem for an enumerated data-type theory is decidable and every enumerated datatype theory has quantifier-free interpolation. As we will see, theories of enumerated-data types are useful to model the Boolean values (true and false) as well as the locations $l_0, \ldots, l_n$ of a program. The second theory is that of *integer difference logic* $\mathcal{IDL}$. The theory $\mathcal{IDL}$ is mono-sorted and its signature contains the constant symbol 0, the unary function symbols $succ$ and $pred$, and the binary predicate symbol $<$. The intended model of $\mathcal{IDL}$ (formed by the integers under the natural interpretation of $succ$, $pred$, and $<$) satisfies the following sentences:[2] the irreflexivity, transitivity and linearity of $<$ together with $\forall x.succ(pred(x)) = x$, $\forall x.pred(succ(x)) = x$, $\forall x, y.x < succ(y) \leftrightarrow (x < y \vee x = y)$, and $\forall x, y.pred(x) < y \leftrightarrow (x < y \vee x = y)$. The atoms of $\mathcal{IDL}$ are equivalent to formulas of the form $i \bowtie f^n(j)$ (for $n \in \mathbb{Z}$, $\bowtie \in \{=, <\}$) where i, j are variables or the constant 0, $f^0(j)$ is j, $f^k(j)$ abbreviates $succ(succ^{k-1}(j))$ when $k > 0$ or $pred(pred^{k+1}(j))$ when $k < 0$. Usually, $i \bowtie f^n(j)$ is written as $i - j \bowtie n$ or as $i \bowtie j + n$ from which the name of "integer difference logic." As shown in, e.g., [67], the $SMT(\mathcal{IDL})$ problem is decidable and $\mathcal{IDL}$ has quantifier-free interpolation. As we will see, this theory is useful to model the operations of incrementing and decrementing by a fixed amount (in many cases 1) counters in loops.

Given a theory $T = (\Sigma, \mathcal{C})$, a *$T$-partition* is a finite set $C_1(\underline{x}), \ldots, C_n(\underline{x})$ of quantifier-free formulas (with free variables contained in the tuple $\underline{x}$) such that $T \models \forall \underline{x} \bigvee_{i=1}^n C_i(\underline{x})$ and $T \models \bigwedge_{i \neq j} \forall \underline{x} \neg(C_i(\underline{x}) \wedge C_j(\underline{x}))$. The formulas $C_1, \ldots, C_k$ are called the *components* of the T-partition. A *case-definable extension* $T' = (\Sigma', \mathcal{C}')$ of a theory $T = (\Sigma, \mathcal{C})$ is obtained from T by applying (finitely many times) the following procedure:

(i) take a T-partition $C_1(\underline{x}), \ldots, C_n(\underline{x})$ together with Σ-terms $t_1(\underline{x}), \ldots, t_n(\underline{x})$;
(ii) let Σ' be $\Sigma \cup \{F\}$, where F is a "fresh" function symbol (i.e., $F \notin \Sigma$) whose arity matches the tuple $\underline{x}$;
(iii) take as $\mathcal{C}'$ the class of Σ'-structures $\mathcal{M}$ whose Σ-reduct is a model of T and such that $\mathcal{M} \models \bigwedge_{i=1}^n \forall \underline{x}\ (C_i(\underline{x}) \to F(\underline{x}) = t_i(\underline{x}))$.

Thus a case-definable extension T' of a theory T contains finitely many additional function symbols, called case-defined functions. By abuse of notation, we shall identify T with its case-definable extensions T'; it is not difficult to prove [46] that the decidability of the $SMT(T)$-problem implies the decidability of the $SMT(T')$-problem.

Definable extensions can be used, for instance, to define conditionals, i.e. $\mathsf{if\text{-}then\text{-}else}$'s. Suppose we are given terms $t_1(\underline{x}), t_2(\underline{x})$ and a 'condition' expressed as a quantifier-free formula $C(\underline{x})$. Using $C, \neg C$ as a partition and t_1, t_2 as terms, we can introduce a definable function symbol F whose meaning corresponds to $\mathsf{if}\ C\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$.

3 Array-based transition systems and their safety

We introduce *array-based transition systems* and show how it is possible to encode in this formalism procedures written in a high-level programming language. For a more extensive

[2] These sentences can be used to axiomatize the set of sentences true in the integers [39].

discussion about array-based transition systems—also for application domains different from imperative programs—the reader is pointed to [46].

Array-based systems are a particular class of guarded assignment systems whose state variables comprise arrays. They are represented symbolically using certain classes of formulas and are endowed with theories specifying the algebraic structures of the indexes and elements of arrays. Ingredients for the definition of an array-based transition systems are 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 $\mathtt{INDEX}$ and a sort of T_E is called $\mathtt{ELEM}_\ell$, where ℓ ranges over a given (finite) set. We assume one of the $\mathtt{ELEM}_\ell$ sorts represents the set $\{l_1, \ldots, l_n\}$ of locations of the program (in the sense that the interpretation of that sort is constrained to be the set $\{l_1, \ldots, l_n\}$ in every model of T_E).

We assume that the $SMT(T_I)$**- and** $SMT(T_E)$**-problems are decidable** and that T_I **and** T_E **have quantifier-free interpolation**.

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 $\mathtt{INDEX}$, $\mathtt{ELEM}_\ell$, and $\mathtt{ARRAY}_\ell$, its signature Σ contains all the symbols in the (disjoint) union $\Sigma_I \cup \Sigma_E \cup \{_[_]_\ell\}_\ell$ where $_[_]_\ell : \mathtt{ARRAY}_\ell \times \mathtt{INDEX} \to \mathtt{ELEM}_\ell$ are the usual dereference operations for arrays, 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 $\mathtt{ARRAY}_\ell$ are interpreted as the sets of all (total) functions from $\mathtt{INDEX}^{\mathcal{M}}$ to $\mathtt{ELEM}_\ell^{\mathcal{M}}$, 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 keep technicalities to a minimum, we adopt the following variant of the notion of an array-based system that can be easily reduced to that given in [46].

An **array-based system (for** T_I, T_E**) is a tuple** $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; \{\tau_h\}_h\rangle$, 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 $\mathtt{ARRAY}$;
- the tuple $\mathbf{c} = c_0, \ldots, c_t$ contains variables of sort $\mathtt{INDEX}$ (called, *counters*);
- the tuple $\mathbf{d} = d_0, \ldots, d_u$ contains variables of sort $\mathtt{ELEM}$ (called, *simple variables*).

All variables are sorted, e.g., for $\mathbf{a}$, this means that each $i = 0, \ldots, t$ is assigned some ℓ so that a_i is of type $\mathtt{ARRAY}_\ell$. The variable d_0 is called the program counter, is sometimes indicated by pc and its sort is the sort interpreted as the set of locations $\{l_1, \ldots, l_n\}$. Among the program locations, we shall distinguish an *initial* location l_{init} and an *error* location l_{error}.

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 symbol e range over variables of a sort $\mathtt{ELEM}$ in Σ_E while i, j, k, z range over variables of sort $\mathtt{INDEX}$.

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 $\mathtt{INDEX}$ (thus, $\mathbf{a}[\underline{i}]$ is an $s \times n$-tuple of terms). Expressions of the form $\phi(\underline{i}, \underline{e})$, $\psi(\underline{i}, \underline{e})$ (possibly sub/super-scripted) denote *quantifier-free* $(\Sigma_I \cup \Sigma_E)$-formulas *in which at most the variables in* $\underline{i} \cup \underline{e}$ *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* τ_h *in functional form* is a formula of the form

$$\exists \underline{k} \begin{pmatrix} \phi_L(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge \ \mathbf{a}' = \lambda j.\ G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \wedge\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge \ \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix} \tag{2}$$

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. As usual, $\mathbf{a}'$, $\mathbf{c}'$, $\mathbf{d}'$ are renamed copies of $\mathbf{a}$, $\mathbf{c}$, $\mathbf{d}$, 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 (2) 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 into pairs of 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.

An array-based system $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; \{\tau_h\}_h \rangle$ is *safe* iff the formulas

$$pc^{(n)} = l_{\mathsf{init}} \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \right) \wedge \cdots \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \right) \wedge pc^{(0)} = l_{\mathsf{error}} \tag{3}$$

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 $0, \ldots, n$). If there exists a value of n for which (3) is A_I^E-satisfiable, then this means that there exists an execution of $\mathcal{S}$ starting from the first location and ending in an error location.

3.1 From programs to array-based transition systems

It is possible to associate an array-based transition system to the body of a procedure written in an imperative language by means of standard syntactical transformations. We illustrate the process on the procedure in Fig. 1.

We assume the theory T_I to be the theory $\mathcal{IDL}$ of integer difference logic (introduced in Sect. 2) extended with a constant L. The sort `INDEX` is interpreted as the set $\mathbb{N}$ of the natural numbers. The theory T_E is composed of three mono-sorted theories: one is $\mathcal{IDL}$, another is the theory of the enumerated data-type of the Boolean values true and false, and the third one is the theory of the enumerated data-type of locations l_0, l_1, l_2, l_3, l_4.

The tuple $\mathbf{a}$ of array state variables contains the variables a and b, the tuple $\mathbf{c}$ of counters contain just i, and the tuple $\mathbf{d}$ of simple variables contains pc and f.

The following transitions $\tau_0, \ldots, \tau_9$ specify the instructions of the Running procedure.[3]

$$\begin{aligned}
\tau_0 &:= pc = l_0 \wedge i' = 0 \wedge pc' = l_1 \\
\tau_1 &:= pc = l_1 \wedge i < L \wedge a[i] \geq 0 \wedge i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{true}) \\
\tau_2 &:= pc = l_1 \wedge i < L \wedge a[i] < 0 \wedge i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{false}) \\
\tau_3 &:= pc = l_1 \wedge i \geq L \wedge pc' = l_2 \wedge i' = 0 \wedge f' = \mathsf{true} \\
\tau_4 &:= pc = l_2 \wedge i < L \wedge a[i] < 0 \wedge b[i] \wedge f' = \mathsf{false} \wedge i' = i + 1 \\
\tau_5 &:= pc = l_2 \wedge i < L \wedge a[i] \geq 0 \wedge \neg b[i] \wedge f' = \mathsf{false} \wedge i' = i + 1 \\
\tau_6 &:= pc = l_2 \wedge i < L \wedge a[i] \geq 0 \wedge b[i] \wedge i' = i + 1 \\
\tau_7 &:= pc = l_2 \wedge i < L \wedge a[i] < 0 \wedge \neg b[i] \wedge i' = i + 1 \\
\tau_8 &:= pc = l_2 \wedge i \geq L \wedge pc' = l_3 \\
\tau_9 &:= pc = l_3 \wedge f = \mathsf{false} \wedge pc' = l_4
\end{aligned}$$

[3] For simplicity, in this example we omit identical updates.

where $\mathsf{store}(b, i, e)$ abbreviates the expression $\lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ e\ \mathsf{else}\ b[j]$. Notice that transitions $\tau_1, \tau_2, \tau_4, \tau_5, \tau_6$ and τ_7 are not instances of formula (2) since terms of the form $\mathbf{a}[\mathbf{c}]$ are not allowed.

This is, however, without loss of generality. In fact, any formula of the form $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ can be rewritten to $\exists \underline{x}(\underline{x} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{x}] \cdots))$ by using (fresh) existentially quantified variables $\underline{x}$ of sort `INDEX`. So, the formula above can be re-written as follows:

$$\tau_1 := pc = l_1 \wedge i < L \wedge \exists x.(x = i \wedge a[x] \geq 0) \wedge i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{true})$$
$$\tau_2 := pc = l_1 \wedge i < L \wedge \exists x.(x = i \wedge a[x] < 0) \wedge i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{false})$$
$$\tau_4 := pc = l_2 \wedge i < L \wedge \exists x.(x = i \wedge a[x] < 0 \wedge b[x]) \wedge f' = \mathsf{false} \wedge i' = i + 1$$
$$\tau_5 := pc = l_2 \wedge i < L \wedge \exists x.(x = i \wedge a[x] \geq 0 \wedge \neg b[x]) \wedge f' = \mathsf{false} \wedge i' = i + 1$$
$$\tau_6 := pc = l_2 \wedge i < L \wedge \exists x.(x = i \wedge a[x] \geq 0 \wedge b[x]) \wedge i' = i + 1$$
$$\tau_7 := pc = l_2 \wedge i < L \wedge \exists x.(x = i \wedge a[x] < 0 \wedge \neg b[x]) \wedge i' = i + 1$$

Notice also that the use of λ-abstractions (recall that $\mathsf{store}(b, i, e)$ stands for $\lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ e\ \mathsf{else}\ b[j]$) in (2) does not go beyond first-order logic, since $\mathbf{a}' = \lambda j.\ G(j, \ldots)$ can be rewritten to the pure first-order formula $\forall j.\ \mathbf{a}'[j] = G(j, \ldots)$.

We are left to specify the initial l_{init} and error l_{error} locations. For the procedure Running in Fig. 1, we define $l_{\mathsf{init}} = l_0$ and $l_{\mathsf{error}} = l_4$. As a consequence, the A_I^E-satisfiability of a formula of the form (3) for some $n \geq 0$ implies that there exists an execution of Running in which it is possible to reach location 4 from location 1. Inspecting the program, it is clear that this happens iff a non-negative (negative) value in the array a is associated to false (true, respectively) in the array b. If this is the case, then property (1), i.e.

$$\forall x.(0 \leq x < L) \rightarrow (a[x] \geq 0 \leftrightarrow b[x] = \mathsf{true}),$$

would not be an invariant of the first loop of Running. The above invariant however is not annotated in the program itself and the challenge is that of designing a model-checking procedure that is able to automatically synthesize it: this will be done in Sect. 5 below.

4 Unwindings for the safety of array-based systems

In our framework, the verification of a safety property for an imperative program P can be reduced to check the reachability of the error location l_{error} by the array-based system associated to P. This amounts to establish if (3) is A_I^E-satisfiable for some $n \geq 0$. Assuming that the A_I^E-satisfiability of formulas of the form (3) is decidable, a possible way to solve the problem is to enumerate the instances of (3) for increasing values of n. When the error condition is reachable, the procedure terminates; otherwise, it diverges. A standard solution to avoid divergence is to compute the set of reachable states and check if a fix-point has been reached. The set of *forward* or *backward* reachable states is obtained by the repeated symbolic execution of transitions from the initial or the error location, respectively. For example, the symbolic execution of a transition τ from a set of states represented by a formula $K(\mathbf{v})$ amounts to the computation of the *pre-image* of $K(\mathbf{v})$ with respect to $\tau(\mathbf{v}, \mathbf{v}')$ as follows:

$$Pre(\tau, K) := \exists \mathbf{v}'.\ (\tau(\mathbf{v}, \mathbf{v}') \wedge K(\mathbf{v}')) . \tag{4}$$

By taking the disjunction of the pre-images of $pc = l_{\mathsf{error}}$ with respect to all transitions, it is possible to compute the set of states from which l_{error} is reachable by applying just one transition. The reachability of the error location can be established with an iterative pre-image

computation procedure, interleaved with checks for detecting fix-points or the presence of the initial location in the set of reachable states. Even when there is no sequence of transitions leading the system from the initial to the error location, it is possible to stop the procedure and conclude safety.

The problem with this procedure is that it is often impossible to compute fix-points for infinite state systems such as those associated to many programs. To alleviate this problem, an over-approximation of the set of reachable states is computed. This set has to be sufficiently coarse to permit the detection of a fix-point and sufficiently precise to show the safety of the analyzed system, if the case. In program verification it is a common practice to compute an over-approximation of the set of forward reachable states. In our case, given the backward reachability procedure, we consider the computation of an over-approximation of a backward reachable state-space. In this section, we show how it is possible to over-approximate the set of backward reachable states of an array-based system by using *labeled unwindings* [54].

4.1 Labeled unwindings for the safety of array-based systems

Preliminarily, we introduce some technical notions and notations. If ψ is a quantifier-free formula in which at most the index variables in $\underline{i}$ 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 (2) is the formula (2) itself without the existential prefix $\exists \underline{k}$; the *proper variables* of τ are those in $\underline{k}$. Below, we freely rename bounded variables in formulas of the form (2) without explicit mention.

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

- (i) for every $v \in V$, if $v = \varepsilon$, then $M_V(\varepsilon)$ is $pc = l_{\mathsf{error}}$; otherwise (i.e. $v \neq \varepsilon$), $M_V(v)$ 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;
- (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

$$M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}') \models_{A_I^E} M_V(v)(\mathbf{v}); \quad (5)$$

- (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 this 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*). A closer look at condition (5) allows us to show how the set of backward reachable states obtained by repeatedly computing pre-images (4) can be over-approximated by the the formulas attached to the vertices of a labeled unwinding. For this, we show 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 a transitionτ.

Lemma 1 *Let* $(u, w) \in E$ *be an arc in a labeled unwinding* (V, E, M_E, M_V)*; we have*

$$Pre(\tau, M_V(w)^{\exists}) \models_{A_I^E} M_V(v)^{\exists}$$

where τ *is the guarded assignment in functional form whose matrix is* $M_E(v, w)$.

Proof If we introduce existential quantifiers in both members of (5), we get

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

taking into consideration that the proper variables of τ are the only index variables occurring free in the matrix of τ and that such proper variables do not occur in $M_V(w)$, we can move inside index quantifiers and get

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

which is the claim because $M_E(v, w)(\mathbf{v}, \mathbf{v}')^{\exists}$ is $\tau(\mathbf{v}, \mathbf{v}')$. □

From this, it is clear that the disjunction of the existential index closure of the formulas labeling the vertices of an unwinding is an over-approximation of the set of backward reachable states. As discussed above, the over-approximation is useful only when it allows us to prove safety when this is the case, i.e. when the approximation is not too coarse. This is equivalent to say that the negation of the formula representing the over-approximated set of (backward) reachable states is an invariant of the system. We now characterize the conditions (see Definition 2 below) under which this is possible.

A set C of vertexes in a labeled unwinding (V, E, M_V, M_E) *covers* a vertex $v \in V$ iff

$$M_V(v)^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}. \tag{6}$$

Definition 2 The labeled unwinding (V, E, M_V, M_E) is *safe* iff for all $v \in V$ we have that if $M_V(v) \models pc = l_{\mathsf{init}}$, 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 [67] may have noticed that our notion of covering involves a set of vertexes rather than a single one as in [67]. Indeed, an efficient implementation of our notion is crucial for efficiency and is discussed in Sect. 7. 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 *If there exists a safe and complete labeled unwinding of* $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$*, then* $\mathcal{S}$ *is safe.*

Proof Let (V, E, M_V, M_E) be a safe and complete labeled unwinding of $\mathcal{S}$ with covering C. We show that $\bigvee_{w \in C} M_V(w)^{\exists}$, which is a disjunction of $\exists^I$-formulas having the variables in $\mathbf{v} = \mathbf{a}, \mathbf{c}, \mathbf{d}$ as free variables, overapproximates the set of the system states that can reach the error location. More formally, we show that for every n the formula

$$\left(\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \right) \wedge \cdots \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \right) \wedge pc^{(0)} = l_{\mathsf{error}}$$

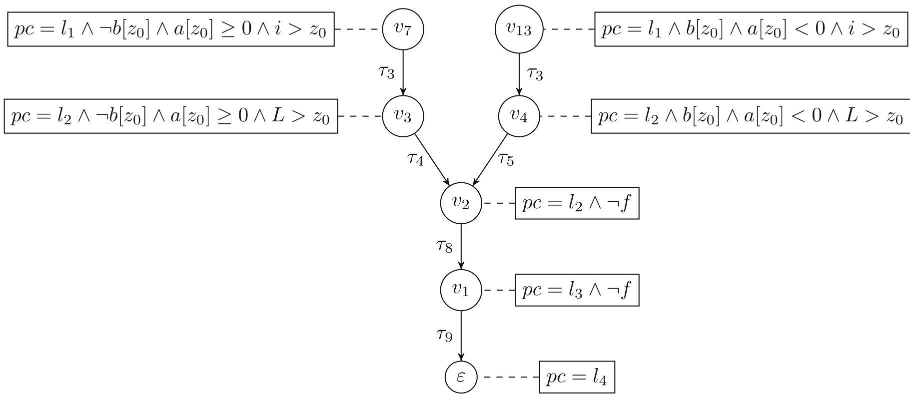

Fig. 3 Covering associated to a labeled unwinding proving the safety of the Running procedure. The variable z_0 has sort `INDEX` and is introduced during backward reachability

A_I^E-entails the formula $\bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)})$. This implies also that the formula (3) cannot be satisfiable, because (V, E, M_V, M_E) is safe. Indeed, if (3) is satisfiable and the claim holds, this means that $pc^{(n)} = l_{\mathsf{init}} \wedge \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)})$ is satisfiable, which can only be if some of the $M_V(w)$ is consistent and A_I^E-entails $pc = l_{\mathsf{init}}$, i.e. if (V, E, M_V, M_E) is unsafe.

The proof of the statement is by induction on n. The case $n = 0$ is trivial because $\varepsilon \in C$ is labeled $pc = l_{\mathsf{error}}$; so suppose $n > 0$. By induction hypothesis, we need to show that

$$\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n-1)}) \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists^I}(\mathbf{v}^{(n)})$$

i.e. that for each $\tau \in \{\tau_h\}_h$ and $v \in C$ we have

$$\tau(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge M_V(v)^{\exists}(\mathbf{v}^{(n-1)}) \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)}).$$

By the definition of a labeled unwinding, either there is a location mismatch and $\tau(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge M_V(v)^{\exists}(\mathbf{v}^{(n-1)})$ is inconsistent, or according to Definition 1(iii) there must be a vertex v' with an edge (v', v) labeled by the matrix of τ in the tree (V, E) (this is because coverings do not contain leaves, hence v is not a leaf). We can now derive our claim from the definition of a covering and the fact that $\tau(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge M_V(v)^{\exists}(\mathbf{v}^{(n-1)})$ A_I^E-entails the formula $M_V(v')^{\exists}(\mathbf{v}^{(n)})$ by Lemma 1. □

As a final remark, we point out that safe and complete labeled unwindings are *quantified* safety certificates for array-based systems. To see why, consider the covering C associated to a safe and complete labeled unwinding. Then, a safe inductive invariant for the array based transition system is represented by the formula

$$\bigwedge_{w \in C} \neg \left(M_V(w)^{\exists}(\mathbf{v}) \right). \tag{7}$$

Consider again the transition system representing the Running procedure. Our framework can generate a safe and complete labeled unwinding for such transition system. The covering associated to this labeled unwinding is depicted in Fig. 3, and represents the following

invariant:

$$
\begin{aligned}
&pc = l_1 \to (\forall z_0.\,((0 \le z_0 \wedge z_0 < i) \to (a[z_0] \ge 0 \leftrightarrow b[z_0])) \quad \wedge \\
&pc = l_2 \to (\forall z_0.\,((0 \le z_0 \wedge z_0 < L) \to (a[z_0] \ge 0 \leftrightarrow b[z_0])) \quad \wedge \\
&pc = l_2 \to f \wedge pc = l_3 \to f \wedge pc \ne l_4\,.
\end{aligned}
$$

4.2 On checking the safety and completeness of labeled unwindings

Theorem 1 states that the safety of an array-based system can be established by checking if there exists a labeled unwinding that is safe and complete. A procedure for searching such an unwinding will be described in the next section. For the moment, assume that a candidate labeled unwinding has been found and consider the problem of checking if it is safe and complete.

It is easy to see that the safety check can be reduced to the A_I^E-satisfiability of a quantifier-free formula. In fact, the formula $M_V(v)$ associated to a vertex v in a labeled unwinding is quantifier-free by Definition 1.(i).

According to Definition 2, testing safety amounts to checking unsatisfiability of quantifier-free formulas. Thus, we need to show that the $SMT(A_I^E)$ problem is decidable. Below we prove that it is indeed so provided that both the $SMT(T_I)$ and $SMT(T_E)$ problems are decidable; recall that this has been assumed in Sect. 3.

Lemma 2 *The* $SMT(A_I^E)$ *problem is decidable.*

Proof Let ψ be a conjunction of literals in the signature of A_I^E. We can assume that such literals are *dereference flat*, i.e. the only terms occurring as arguments of the read operations _[_] are variables. This is without loss of generality since $\phi(t/x)$ can be rewritten to the equisatisfiable formula $t = x \wedge \phi(x)$ with x fresh.

Let $\underline{i} = i_1, \ldots, i_n$, $\underline{a} = a_1, \ldots, a_s$, $\underline{e}$ be the index, array and element variables occurring in ψ, respectively. By making case-splits, we can assume that ψ contain either $i = j$ or $i \ne j$ for all distinct $i, j \in \underline{i}$; in addition, in case $i = j$ is a conjunct of ψ, we can freely assume that $a_k[i] = a_k[j]$ is in ψ for all $a_k \in \underline{a}$.

We can further separate the literals whose root predicate symbol has argument of sort `INDEX` from the literals whose root predicate has arguments of sort `ELEM`,

thus (from the way A_I^E is built) ψ can be rewritten as

$$\psi^I(\underline{i}) \wedge \psi^E(\underline{a}[\underline{i}], \underline{e}). \tag{8}$$

Let $\underline{d} = d_{11}, \ldots, d_{sn}$ be $s \times n$ fresh variables abstracting out the $\underline{a}[\underline{i}]$: we claim that ψ is A_I^E-satisfiable iff ψ^I is T_I-satisfiable and ψ^E is T_E-satisfiable. In fact, given models of ψ^I and ψ^E in the respective theories, it is easy to build a combined model for (8): thanks to the fact that ψ contains a complete partition of the variables in $\underline{i}$ and equalities have been propagated to ψ^E, it is sufficient to assign to $a_k \in \underline{a}$ any function whose value on the element assigned to i_l is d_{kl}. □

This lemma is an important building block for many other results in the paper. Key to its proof is a procedure based on reducing the A_I^E-satisfiability check of quantifier-free formulas to $SMT(T_I)$ and $SMT(T_E)$ problems by means of a (unidirectional) variant of the Nelson-Oppen combination schema [70] in which only disjunctions of equalities between terms of sort `INDEX` are exchanged, whereas those involving terms of sort `ELEM` are not. (For the sake of completeness, we mention that Lemma 2 can be seen as an application of a more general combination result stated in [10].) We point out that the procedure used in the proof

might need to be complemented by suitable heuristics to scale up and handle large formulas generated during backward reachability. In our implementation, instead of building from scratch the procedure in the proof of Lemma 2, we prefer to re-use available SMT-Solvers for checking the satisfiability of $SMT(A_I^E)$ problems. This is discussed in Sect. 7.

We now turn to the problem of checking the completeness of a labeled unwinding. 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, by refutation from (6), this may be reduced to repeatedly check the A_I^E-unsatisfiability of $\exists^{A,I}\forall^I$-sentences, i.e. formulas of the form

$$\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i}\, \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d}), \tag{9}$$

where $\underline{i}, \underline{j}, \mathbf{c}$ are of sort `INDEX`, $\mathbf{a}$ of sort `ARRAY`, and $\mathbf{d}$ of sort `ELEM` (recall the definition from Sect. 3). Unfortunately, the A_I^E-satisfiability of these sentences is (in general) undecidable [46]. The problem is the handling of the universally quantified variables of $\underline{j}$ that occur in (9) since all the other existentially quantified variables in $\mathbf{a}$, $\mathbf{c}$, $\mathbf{d}$, and $\underline{i}$ can be regarded as Skolem constants. To alleviate the problem, an idea is to design an incomplete instantiation procedure for the variables in $\underline{j}$ so as to obtain a conjunction of quantifier-free formulas whose A_I^E-satisfiability is decidable by Lemma 2. Our *default instantiation procedure* computes the set Σ of all possible substitutions mapping the variables in $\underline{j}$ into $\underline{i} \cup \mathbf{c}$. Our *default satisfiability procedure* uses the default instantiation procedure so as to check the A_I^E-unsatisfiability of the formula

$$\bigwedge_{\sigma \in \Sigma} \psi(\underline{i}, \underline{j}\sigma, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,. \tag{10}$$

It returns the A_I^E-unsatisfiability of (9) when (10) is so and returns "unknown" when (10) is A_I^E-satisfiable. In other words, the default satisfiability procedure is sound but incomplete for checking the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences. In Sect. 6, we show that the adoption of such a procedure allows us to use labeled unwindings as safety certificates. To clarify that the notion of completeness for labeled unwindings is relative to the incomplete algorithm used to check the completeness of coverings, we introduce the following notion.

Definition 3 The labeled unwinding (V, E, M_V, M_E) is *recognized to be complete* iff there exists a set of non-leaf vertexes C (called a 'recognized covering' or simply a 'covering' for the sake of simplicity) containing ε and such that for every $v \in C$ and $(v', v) \in E$, it happens that the relation (6) is verified to hold by using the default satisfiability procedure for A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences.

In Sect. 6, we will identify sufficient conditions under which the default instantiation procedure allows us to build a decision procedure for the A_I^E-satisfiability problem of $\exists^{A,I}\forall^I$-sentences. We will also see that the same conditions guarantee the termination of the procedure described in the next section that finds a safe and complete labeled unwinding.

In Sect. 7, we will describe heuristics to reduce the number of possible instances that must be considered by the default instantiation procedure so as to improve performance. The experiments described in Sect. 8 show the efficiency of the default satisfiability procedure described above.

5 Lazy abstraction with interpolation-based refinement for arrays

We now describe how to construct labeled unwindings and how this process is interleaved with the checks for safety and completeness described in Sect. 4.2. Similarly to [67], we design a possibly non-terminating procedure 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 $pc = l_{\mathsf{error}}$. UNWIND uses two sub-procedures: EXPAND builds the labeled unwinding and REFINE refines labeled unwindings by eliminating spurious unsafe traces via interpolants. 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: $\mathcal{S}$ is safe by Theorem 1.

As we will see below, a crucial advantage of our approach is that REFINE *needs to compute only quantifier-free interpolants (in a restricted form) to refine spurious unsafe traces, despite the fact that quantified formulas are used to represent sets of states and transitions*. Technically, this is possible because formulas describing potentially unsafe traces can be transformed to equisatisfiable quantifier-free formulas by a partial instantiation procedure (see Sect. 5.2 below for details).

In the following, we give a non-deterministic version of UNWIND: the two procedures EXPAND and UNWIND can be non-deterministically applied to a labeled unwinding to obtain a new one, whenever this is possible according to their applicability conditions (described below). The implementation strategies of UNWIND will be described in Sect. 7.

5.1 The two sub-procedures of UNWIND

Let (V, E, M_V, M_E) 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_V, M_E) is not recognized to be complete (recall Definition 3) and there exists a leaf vertex v whose location is such that $M_V(v) \not\models_{A_I^E} pc = l_{\mathsf{init}}$. From the applicability condition and Definition 1(i), we have that $M_V(v) \models_{A_I^E} pc = l$ for some $l \neq l_{\mathsf{init}}$.
The effects of applying this procedure are the following: for each transition $\tau \in \{\tau_h\}_h$ whose target is l, a new leaf w_τ, labeled by $pc = src(\tau)$, is added together with a new edge (w_τ, v), labeled by τ, to the current unwinding.

REFINE. The applicability condition is that (V, E, M_V, M_E) is not recognized to be complete (recall Definition 3) and there exists a vertex $v \in V$ whose location is l_{init} and it is such that $M_V(v)$ is A_I^E-satisfiable.
In the current labeled unwinding, 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; the set of these transitions is called a *counterexample*. If

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \qquad (11)$$

is A_I^E-satisfiable then the counterexample is said to be *feasible*, the procedure fails, and reports the unsafety of $\mathcal{S}$. Otherwise, the counterexample is said to be *infeasible* and the effect of applying the procedure is to strengthen the labels of the counterexample vertices by using the interpolants retrieved from the unsatisfiability of (11).

The mechanization of the applicability conditions for both sub-procedures have been discussed in Sect. 4.2. This means that enough details for the mechanization of EXPAND are already available. This is not the case for REFINE because it is unclear how to check the A_I^E-satisfiability of formulas of the form (11)—this is crucial to establish the feasibility or infeasibility of a counterexample—and we do not know how to compute interpolants and how to use them in order to "strengthen the labels in the counterexample."

The feasibility of counterexamples is discussed in Sect. 5.2, the computation of (quantifier-free) interpolants in Sect. 5.4, and their use in refining (infeasible) counterexamples in Sect. 5.3.

5.2 Checking the feasibility of counterexamples

We describe a decision procedure for checking the A_I^E-satisfiability of formulas of the form (11), thereby enabling to check the feasibility of counterexamples in REFINE. The idea underlying the procedure is to instantiate the variables bound by the λ-abstraction in the updates of the transitions occurring in (11) with finitely many constants and then check the resulting quantifier-free formula for A_I^E-satisfiability. The fact that only finitely many instances are sufficient is shown by the following observations.

By recalling (2), rewrite (11) to

$$\bigwedge_{k=1}^{m} \exists \underline{i}_k \left[\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right] \tag{12}$$

which, by Skolemizing existentially quantified variables, can be further rewritten to the equisatisfiable formula (here and in the following, by abuse of notation, we consider the variables in $\underline{i}_k$ as Skolem constants):

$$\bigwedge_{k=1}^{m} \left[\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right]. \tag{13}$$

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

$$\bigwedge_{k=1}^{m} \left[\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \bigwedge_{j \in \underline{i}_{k+1}, \dots, \underline{i}_m} \mathbf{a}^{(k)}[j] = G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right] \tag{14}$$

Lemma 3 *Formulas (13) and (14) are A_I^E-equisatisfiable.*

Proof Indeed, (13) A_I^E-entails (14). Vice-versa, suppose we are given an A_I^E-model $\mathcal{M}$ and a satisfying assignment $\mathtt{s}$ for (14), our goal is to produce a satisfying assignment $\tilde{\mathtt{s}}$ for (13)

based on the same A_I^E-model $\mathcal{M}$. For simplicity, let us call $\underline{i}_1, \dots, \underline{i}_m, \mathbf{v}^{(0)}, \dots, \mathbf{v}^{(m)}$ the elements from the support of $\mathcal{M}$ assigned by $\mathtt{s}$ to the variables $\underline{i}_1, \dots, \underline{i}_m, \mathbf{v}^{(0)}, \dots, \mathbf{v}^{(m)}$ occurring free in (13) and (14). The assignment $\tilde{\mathtt{s}}$ will change only the values assigned to $\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(m)}$ (notice that $\mathbf{v}^{(0)}$ is left unchanged). We define $\tilde{\mathtt{s}}(\mathbf{v}^k)$ for $k > 0$ inductively as follows:

$$\begin{aligned}\tilde{\mathtt{s}}(\mathbf{a}^{(k)}) &= \lambda j\, G_k(\underline{i}_k, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[j])\\ \tilde{\mathtt{s}}(\mathbf{c}^{(k)}) &= H_k(\underline{i}_k, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)})\\ \tilde{\mathtt{s}}(\mathbf{d}^{(k)}) &= K_k(\underline{i}_k, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)})\end{aligned}$$

To show that (13) holds under $\tilde{\mathtt{s}}$, a simple induction on k $(= 1, \dots, m)$ is sufficient to check that $\tilde{\mathtt{s}}(\mathbf{c}^{(k-1)}) = \mathbf{c}^{(k-1)}$, $\tilde{\mathtt{s}}(\mathbf{d}^{(k-1)}) = \mathbf{d}^{(k-1)}$ and $\tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[j] = \mathbf{a}^{(k-1)}[j]$ for all $j \in \underline{i}_k \cup \dots \cup \underline{i}_m$.

As a consequence of this, the formulas ϕ_k's still hold under $\tilde{\mathtt{s}}$ and the remaining conjuncts of (13) hold by construction. □

An easy corollary of Lemmas 3 and 2 is the following result.

Lemma 4 *The A_I^E-satisfiability of formulas of the form* (11) *is decidable.*

This means that we can check the feasibility of counterexamples under the assumption that the SMT problems of the theory T_I over indexes and the theory T_E over elements are decidable (recall that this has been assumed in Sect. 3). A by-product of this result is the decidability of the bounded model checking problem (formally defined below) for array-based systems.

Let $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; \{\tau_h\}_h \rangle$ and recall formula (3), i.e.

$$pc^{(n)} = l_{\mathsf{init}} \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \right) \wedge \dots \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \right) \wedge pc^{(0)} = l_{\mathsf{error}}\,.$$

When $n \geq 0$ is known, we say that the *bounded model checking problem* for $\mathcal{S}$ consists of checking the A_I^E-satisfiability of the formula above for the given value of n. We now show that Lemmas 3 and 2 also imply the decidability of this problem.

First of all, observe that, by applying standard distributive laws and renaming of variables (the variable $\mathbf{v}^{(k)}$ is renamed to $\mathbf{v}^{(n-k)}$, so $\mathbf{v}^{(n)}$ is renamed to $\mathbf{v}^{(0)}$, $\mathbf{v}^{(n-1)}$ to $\mathbf{v}^{(1)}$, ..., $\mathbf{v}^{(1)}$ to $\mathbf{v}^{(n-1)}$, and $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(n)}$), the formula above can be rewritten to a disjunction of formulas of the form

$$pc^{(0)} = l_{\mathsf{init}} \wedge \tau_{h_1}(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \dots \wedge \tau_{h_n}(\mathbf{v}^{(n-1)}, \mathbf{v}^{(n)}) \wedge pc^{(n)} = l_{\mathsf{error}}\,, \qquad (15)$$

where h_j ranges over the same set of indexes of the transitions in $\mathcal{S}$ and $j = 1, \dots, n$. Now, observe that $\tau_{h_1}(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \dots \wedge \tau_{h_n}(\mathbf{v}^{(n-1)}, \mathbf{v}^{(n)})$ has the same form of (11) and, by Lemma 3, it is A_I^E-equisatisfiable to a quantifier-free formula ϕ of the form (14).

The decidability of (3) is now obvious because every transition formula $\tau_h(\mathbf{v}, \mathbf{v}')$ entails $pc = src(\tau_h) \wedge pc' = trg(\tau_h)$ (recall the definition of a guarded assignment in functional form from Sect. 3) and, "modulo" A_I^E formulas of the form $l_1 = l_2$, are unsatisfiable when locations l_1 and l_2 are distinct. Thus (15) is either trivially unsatisfiable (in case of the locations are different) or equisatisfiable to ϕ.

Theorem 2 *The bounded model checking problem for array-based systems is decidable.*

5.3 Refining counterexamples with interpolants

Assume that Refine has detected that the infeasibility of the counterexample associated to the path $v_0 \xrightarrow{\tau_1} v_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_m} v_m = \varepsilon$ as shown in Sect. 5.2, i.e. by the checking the A_I^E-unsatisfiability of the formula $\tau_1 \wedge \cdots \wedge \tau_m$ of the form (11). At this point, Refine needs to refine the counterexample. Following [67], this is done by computing *path interpolants* that are conjoined to the labels of the vertices of the path under consideration to strengthen them. This is detailed in the following by assuming the availability of a procedure capable of computing interpolants for quantifier-free formulas (the description of such a procedure is postponed to Sect. 5.4).

Let us consider an A_I^E-unsatisfiable formula of the form (11). By Lemma 3, this formula is A_I^E-equisatisfiable to a quantifier-free formula of the form (14). This implies that also (14) is A_I^E-unsatisfiable.

Let us abbreviate the k-th conjunct in (14) as

$$\tilde{\tau}_k\left(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], tc^{(k-1)}, \mathbf{c}^{(k)}, \mathbf{d}^{(k-1)}, \mathbf{d}^{(k)}\right). \quad (16)$$

Thus, (14) can be written as $\tilde{\tau}_1 \wedge \cdots \wedge \tilde{\tau}_m$. Now, let

$$\psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_{k+1}], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}) \quad (17)$$

be one of the quantifier-free interpolants (for $k = 1, \ldots, m$)—computed by repeatedly invoking the available interpolation procedure on the A_I^E-unsatisfiable formula (14) from right-to-left. The ψ_k's are such that

$$\psi_0 \equiv \bot, \qquad \psi_m \equiv \top, \quad (18)$$

$$\psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], \mathbf{c}^{(k)}, \mathbf{d}^{(k)}) \wedge \tilde{\tau}_k \models_{A_I^E} \psi_{k-1}(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}). \quad (19)$$

Once these interpolants are computed, Refine updates the label of v_k, for $k = 0, \ldots, m-1$, in the path $v_0 \xrightarrow{\tau_1} v_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_m} v_m = \varepsilon$ as follows:

$$M_V(v_k) \equiv M_V(v_k) \wedge \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_k], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}). \quad (20)$$

Since the matrix of τ_k A_I^E-entails $\tilde{\tau}_k$, condition (5) of Definition 1.(*ii*) of labeled unwinding (see Sect. 4.1) is preserved and the vertex v_0 is now labeled by an A_I^E-unsatisfiable formula. Figure 4 illustrates how this works on an counterexample associated to the Running procedure.

5.4 An interpolation procedure for quantifier-free formulas

We now describe the interpolation procedure for quantifier-free formulas used to compute path-interpolants for refining infeasible counterexamples (as described in Sect. 5.3).

First of all, recall that we assumed that quantifier-free interpolants can be computed for both T_I and T_E in Sect. 3. Unfortunately, this is not sufficient to guarantee the possibility to compute quantifier-free interpolants for quantifier-free formulas in A_I^E. In fact, this theory can be seen as a combination of T_I and T_E with (uninterpreted) function symbols by considering arrays as function symbols and the dereference operation as function application. Negative results (such as [18,19]) are available in the literature showing that the addition of (uninterpreted) function symbols to theories allowing for the computation of quantifier-free

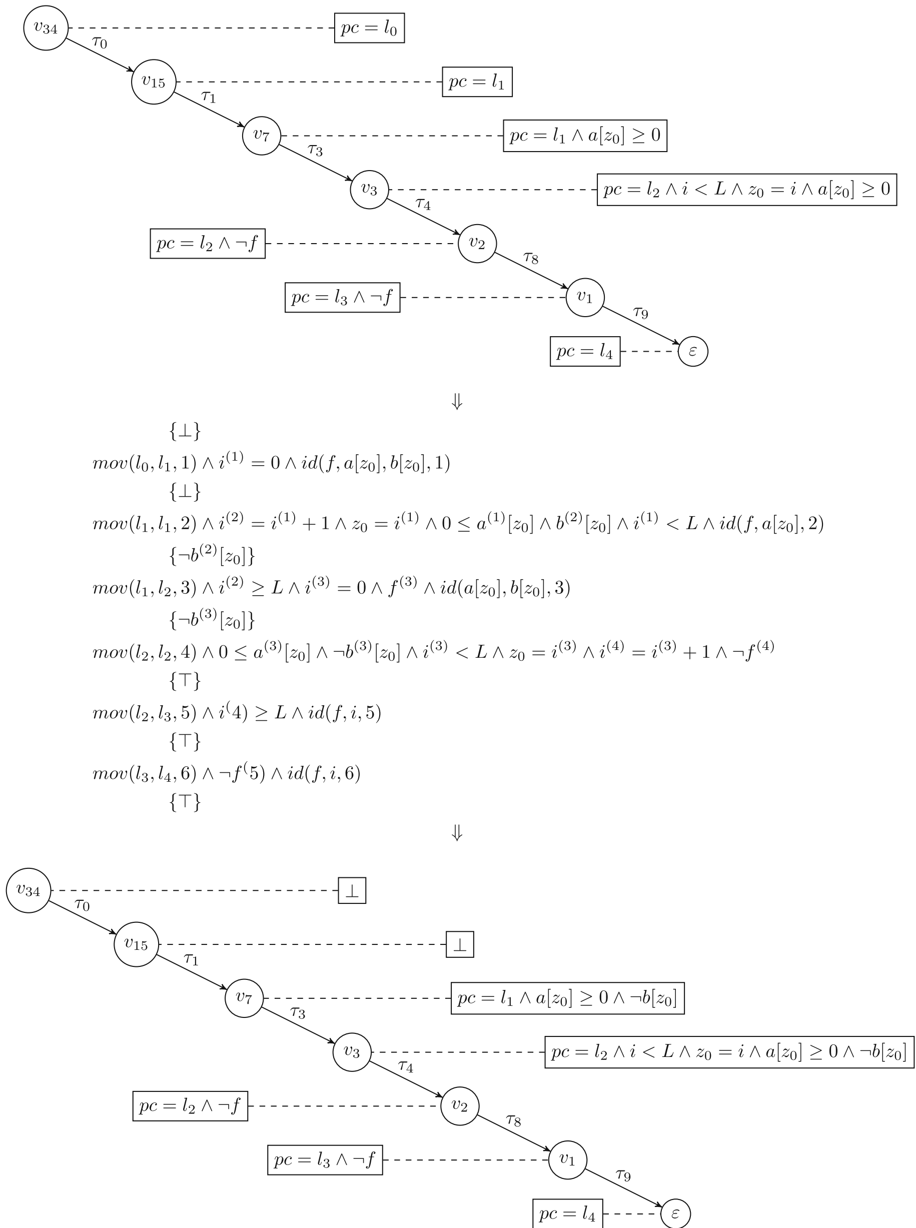

Fig. 4 Refinement of a spurious path. The *upper figure* represents a path in the labeled unwinding of the array-based system associated to the procedure Running in Fig. 1. The vertices along the path are labeled with predicates generated during previous refinements. The counterexample associated to the path is infeasible. The interpolants computed for this path are shown in the middle. For the sake of readability, $mov(l_i, l_j, k)$ stands for $pc^{(k-1)} = l_i \wedge pc^{(k)} = l_j$ and $id(t_1, \ldots, t_n; k)$ for $t_1^{(k)} = t_1^{(k-1)} \wedge \ldots \wedge t_n^{(k)} = t_n^{(k-1)}$. The Skolem variables introduced by REFINE are denoted by z_j for $j \geq 0$. The picture below shows the refined path

interpolants prevents the existence of quantifier-free interpolants in the extended theory. Fortunately, the A_I^E-unsatisfiable formulas of the form $\psi_1 \wedge \psi_2$ for which an interpolant must be computed when invoking the procedure REFINE are such that ψ_1 and ψ_2 satisfy certain conditions on their shape that guarantee the possibility to compute quantifier-free interpolants as stated in the following result.

Theorem 3 *Suppose that $\psi_1 \wedge \psi_2$ is an A_I^E-unsatisfiable quantifier-free formula such that all terms of sort* `INDEX` *occurring in ψ_2 under the scope of the dereference operation _[_] 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; and (iii) all free variables occurring in ψ_0 occur both in ψ_1 and ψ_2.*

Proof Let us call *critical* the index variables occurring both in ψ_1 and ψ_2 (by assumptions, the index variables occurring in ψ_2 under the scope of the dereference operator _[_] are critical). Without loss of generality, we may assume that ψ_1 and ψ_2 are conjunctions of dereference flat literals (see the proof of Lemma 2 for this notion) and that for all distinct variables i, j occurring in ψ_1, we have that ψ_1 contains either the literal $i = j$ or the literal $i \neq j$. These assumptions can be justified by standard considerations. For instance, once interpolants for $\psi_1' \wedge \psi_2$ and for $\psi_1'' \wedge \psi_2$ are known, one can combine them to an interpolant for $(\psi_1' \vee \psi_1'') \wedge \psi_2$ by taking disjunction.[4] We can also assume that, whenever ψ_1 contains $i = j$, then it contains also $a[i] = a[j]$ for every array variable occurring in ψ_1; finally, if i, j are critical variables and $i = j$ is a conjunct of ψ_1, then we assume that ψ_2 contains $a[i] = a[j]$ for every array variable a occurring in ψ_2. In fact, if adding $i = j \wedge a[i] = a[j]$ to ψ_2 one gets the interpolant ψ_0, it is possible to get the interpolant back from ψ_2 by taking $i = j \to \psi_0$.

Let now ψ_1 be of the kind

$$\psi_1(\underline{i}_1, \underline{i}_0, \underline{a}_1[\underline{i}_1], \underline{a}_1[\underline{i}_0], \underline{a}_0[\underline{i}_1], \underline{a}_0[\underline{i}_0], \underline{e}_1, \underline{e}_0)$$

and ψ_2 be of the kind

$$\psi_2(\underline{i}_0, \underline{i}_2, \underline{a}_2[\underline{i}_0], \underline{a}_0[\underline{i}_0], \underline{e}_2, \underline{e}_0),$$

where $\underline{a}_1, \underline{a}_0, \underline{a}_2$ are array variables, $\underline{e}_0, \underline{e}_1, \underline{e}_2$ are element variables, and $\underline{i}_0, \underline{i}_1, \underline{i}_2$ are index variables (the $\underline{i}_0$ are the critical ones - notice that terms $\underline{a}_0[\underline{i}_2], \underline{a}_2[\underline{i}_2]$ do not occur in ψ_2). We can further separate the literals whose root predicate symbol has argument of sort `INDEX` from the literals whose root predicate has arguments of sort `ELEM`, thus ψ_1 can be rewritten as

$$\psi_1^I(\underline{i}_1, \underline{i}_0) \wedge \psi_1^E(\underline{a}_1[\underline{i}_1], \underline{a}_1[\underline{i}_0], \underline{a}_0[\underline{i}_1], \underline{a}_0[\underline{i}_0], \underline{e}_1, \underline{e}_0)$$

whereas ψ_2 as

$$\psi_2^I(\underline{i}_0, \underline{i}_2) \wedge \psi_2^E(\underline{a}_2[\underline{i}_0], \underline{a}_0[\underline{i}_0], \underline{e}_2, \underline{e}_0)$$

for ψ_g^I and ψ_g^E conjunctions of literals whose root predicate symbols have argument of sort `INDEX` and `ELEM`, respectively, and $g = 1, 2$.

Now, since a complete partition on indexes $\underline{i}_0, \underline{i}_1$ is included in ψ_1[5] and relevant index equalities have been fully propagated through array variables, it is easy to see, by using the

[4] For a general framework covering all these transformations, the reader is pointed to [20].

[5] In practice, this might result in a large combinatorial blow-up. Practical optimizations for the scalability of this procedure are described in Sect. 7.4.

same argument as in the proof of Lemma 2, that the inconsistency of $\psi_1 \wedge \psi_2$ implies that either

$$\psi_1^I(\underline{i}_1, \underline{i}_0) \wedge \psi_2^I(\underline{i}_0, \underline{i}_2)$$

is T_I-unsatisfiable or

$$\psi_1^E(\underline{d}_1', \underline{d}_1'', \underline{d}_1''', \underline{d}_0, \underline{e}_1, \underline{e}_0) \wedge \psi_2^E(\underline{d}_2, \underline{d}_0, \underline{e}_2, \underline{e}_0)$$

is T_E-unsatisfiable, where we used fresh element variables $\underline{d}_0, \underline{d}_1', \underline{d}_1'', \underline{d}_1''', \underline{d}_2$ instead of the terms $\underline{a}_0[\underline{i}_0], \underline{a}_1[\underline{i}_1], \underline{a}_1[\underline{i}_0], \underline{a}_0[\underline{i}_1], \underline{a}_2[\underline{i}_0]$, respectively. Now it is clear that we can use the available quantifier-free interpolation algorithms for T_I and T_E in order to compute the interpolant ψ_0. □

6 Correctness and termination

Recall that Unwind consists of the exhaustive (non-deterministic) application of Expand and Refine. We now show that Unwind correctly establishes the safety of an array-based system when terminating.

Theorem 4 *Let* Unwind *be applied to an array-based system $\mathcal{S}$. If* Unwind *reports unsafety, then $\mathcal{S}$ is unsafe. If neither* Expand *nor* Refine *can be applied to a labeled unwinding P of $\mathcal{S}$, then P is safe and complete (and thus $\mathcal{S}$ is safe by Theorem 1).*

Proof The first part of the claim is obvious. For the second part, let us consider a labeled unwinding $P = (V, E, M_V, M_E)$ of $\mathcal{S}$ to which neither Expand nor Refine applies. We first show that P is complete. Notice that if leaves are all labeled by A_I^E-unsatisfiable formulas, non-leaf vertexes are a covering, and the system is complete. On the other hand, if there is a leaf labeled by an A_I^E-satisfiable formula, one of the two sub-procedures applies unless the current labeled unwinding is recognized to be complete—according to Definition 3 in Sect. 4.2—and hence complete *tout court*. Thus, the labeled unwinding must be complete when no sub-procedure is applicable.

Finally, if P is not safe, there is a consistent vertex v whose location is l_{init}. Now, since l_{init} is not a target location, v must be a leaf; for the same reason, v is not covered by non-leaf vertexes (the location of these vertexes is not l_{init}). Thus the labeled unwinding is not complete, hence it cannot be recognized as such, and Refine is applicable. □

This result implies the partial correctness of Unwind. In the rest of this section, we investigate total correctness.

6.1 Precisely recognizing complete labeled unwindings

The first step towards the total correctness of Unwind is to have a complete "default satisfiability procedure" for recognizing complete covers; recall Definition 3 in Sect. 4.2. The default satisfiability procedure uses the "default instantiation procedure" to reduce the problem of checking the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences to checking the A_I^E-satisfiability of quantifier-free formulas. Since a decision procedure for the latter is available (under the hypothesis that the $SMT(T_I)$ and the $SMT(T_E)$ problems are decidable as assumed in Sect. 3), we need to find conditions under which the default instantiation procedure is complete. To formally characterize this, we need to introduce the following notion.

A class $\mathcal{C}$ of structures is *closed under substructures* if for every structure $\mathcal{M} \in \mathcal{C}$, it happens that all the sub-structures of $\mathcal{M}$ are also in $\mathcal{C}$. Any theory whose class of models is specified as the class of models of a set of universal sentences, i.e. formulas containing no free variables obtained by prefixing a quantifier-free formula with a finite sequence of universal quantifiers, is closed under substructures by well-known results in model theory (see, e.g., [57]). For example, the theory of posets (i.e. of sets endowed with a reflexive, transitive and antisymmetric relation) can be axiomatized by a set of universal sentences and it is thus closed under substructures.

Theorem 5 ([46]) *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.*

Proof We claim that, under the hypotheses of the theorem, the A_I^E-satisfiability of (10), i.e.

$$\bigwedge_{\sigma\in\Sigma} \psi(\underline{i}, \underline{j}\sigma, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})$$

(where Σ denotes the set of all possible substitutions mapping the variables in $\underline{j}$ into $\underline{i} \cup \mathbf{c}$) implies the A_I^E-satisfiability of (9), i.e.

$$\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i}\, \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,.$$

This is sufficient to show the decidability of the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences since the A_I^E-satisfiability of (10) is decidable by Lemma 2 and the A_I^E-satisfiability of (9) implies the A_I^E-satisfiability of (10).

We consider a structure $\mathcal{M}$ which (together with an assignment to the free variables $\mathbf{a}$, $\mathbf{c}$, $\mathbf{d}$) is a model of (10) and we derive from this a structure $\mathcal{M}'$ as follows. First, the interpretation of the sort `INDEX` in $\mathcal{M}'$ is obtained by restricting that in $\mathcal{M}$ of the same sort `INDEX` (as well as of all symbols in Σ_I) to the subset containing only the elements assigned to the variables in $\underline{i}$, $\mathbf{c}$. The interpretation of the symbols of Σ_E in $\mathcal{M}'$ is identical to that of $\mathcal{M}$ and the functions assigned to the $\mathbf{a}$'s in $\mathcal{M}'$ are the same of those in $\mathcal{M}$ but restricted to their domains. Since $\mathcal{C}_I$ is closed under substructures, $\mathcal{M}'$ is still an A_I^E-model. It is easy to see that, since (10) is quantifier-free, the truth of (10) is inherited by $\mathcal{M}'$. Additionally, because of the restriction of the interpretation of the sort `INDEX`, (9) also holds in $\mathcal{M}'$. This concludes the proof of the claim above. □

6.2 Termination of Unwind

Now, that we have found conditions under which precise checks to recognize the completeness of labeled unwindings can be obtained, we focus on studying the termination of Unwind.

First of all, we notice that the termination of Unwind can be easily ensured when $\mathcal{S}$ *is unsafe* by adopting suitable strategies for the application of the sub-procedure Expand. For example, a breadth-first strategy used when expanding the labeled unwinding certainly guarantees termination (the design of other strategies is mostly an implementation issue, see for instance Sect. 8 or also [67]).

If $\mathcal{S}$ *is safe*, the termination of Unwind cannot be shown for arbitrary array-based systems since their safety problem is undecidable in general (see, e.g., [46]). In the following, we investigate sufficiently restrictive conditions under which Unwind is guaranteed to terminate.

In particular, we identify two sufficient conditions for this. 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 formulas 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 vertex.

The second condition for the termination of UNWIND concerns the theory T_E. To formalize this, we need to introduce some formal notions. An *existential* Σ*-sentence* is a formula containing no free variables that is obtained by prefixing a quantifier-free Σ-formula with a finite sequence of existential quantifiers. A structure $\mathcal{M}$ is *finitely generated* iff there exists a finite sub-set X of the support[6] of $\mathcal{M}$ such that the smallest substructure of $\mathcal{M}$ containing X is $\mathcal{M}$ itself. An embedding is an injective homomorphism that preserves and reflects relations and operations. A reflexive-transitive relation $\preceq$ on a set P is a *well-quasi-order* (wqo) iff given $p_0, p_1, \dots p_n, \dots$ from P, there are $n < m$ such that $p_n \preceq p_m$. A *wqo-theory* [22] 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 $\mathcal{M}_2$. As shown in [22], the following is a wqo-theory: it contains one sort, finitely many 0-ary and unary predicate symbols, a single binary predicate symbol $\leq$, and its class of models satisfies the following three (universal) sentences: $\forall x\,(x \leq x)$, $\forall x, y, z\,(x \leq y \wedge y \leq z \rightarrow x \leq z)$, and $\forall x, y\,(x \leq y \vee y \leq x)$, constraining $\leq$ to be interpreted as a total pre-order.

We also need the following technical result.

Lemma 5 *Let $T = (\Sigma, \mathcal{C})$ be a wqo-theory and $K_0, K_1, \dots, K_n, \dots$ be an infinite sequence of existential Σ-sentences such that $K_n \models_T K_{n+1}$ for all $n \geq 0$. Then, there exists $n > 0$ such that $K_n \models_T K_{n-1}$.*

Proof Suppose the statement does not hold. Then, for every n there exists a model $\mathcal{M}_n \in \mathcal{C}$ such that $\mathcal{M}_n \models K_n$ and $\mathcal{M}_n \not\models K_{n-1}$. Since $\mathcal{C}$ is closed under substructures and K_n is an existential sentence, we can take $\mathcal{M}_n$ to be finitely generated. Notice that truth of $\neg K_{n-1}$ is preserved by substructures because this is a universal formula (see, e.g., [57]). Since $K_m \models_T K_{n-1}$ for $m < n$, we have that $\mathcal{M}_n \not\models K_m$ for every $m < n$. Consider now the sequence $\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_n, \dots$ of finitely generated models in $\mathcal{C}$. By definition of a well-quasi-order, there must be $m < n$ such that $\mathcal{M}_m$ embeds in $\mathcal{M}_n$. Then, from $\mathcal{M}_m \models K_m$ and the fact that K_m is existential, it follows that $\mathcal{M}_n \models K_m$. Contradiction! □

We are now in the position to state and prove our result on the termination of UNWIND.

Theorem 6 *Let $\mathcal{S}$ be an array-based system for T_I, T_E. Suppose that T_I satisfies the hypotheses of Theorem 5 and that the theory obtained from $T_I \cup T_E$ by adding the symbols in $\mathbf{v}$, seen as free constants of appropriate sorts, is a wqo theory. Then,* UNWIND *terminates when applied to $\mathcal{S}$ with a fair strategy.*

Proof If we view the state variables $\mathbf{v} := \mathbf{a}, \mathbf{c}, \mathbf{d}$ of the array-based system $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$ as free (function or constants) symbols, the existential (index) closures of the formulas (and their disjunctions) labeling the vertexes in a labeled unwinding of $\mathcal{S}$ are $\exists^I$-formulas of the form $\exists \underline{i}\, \psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$. Thus these are existential formulas of the wqo theory mentioned in the statement of the theorem and Lemma 5 is applicable.

6 In a many-sorted context, the support of $\mathcal{M}$ is taken to be the disjoint union of the sets $S^{\mathcal{M}}$, varying S over the sorts of Σ.

If the fair strategy used to apply EXPAND and REFINE does not terminate, it generates a sequence of labeled unwindings $P_0, P_1, P_2, \ldots$ where $P_j = (V_j, E_j, M_V^j, M_E^j)$ is such that $V_j \subseteq V_{j+1}$ and $E_j \subseteq E_{j+1}$, written as $(V_j, E_j) \subseteq (V_{j+1}, E_{j+1})$, for $j \geq 0$. In other words, we have an increasing sequence of trees of the form $(V_0, E_0), (V_1, E_1), \ldots$ Consider now the union $(V, E) = (\bigcup_k V_k, \bigcup_k E_k)$ of all the trees in the sequence. Since vertices are not refined infinitely often, we can associate with any vertex $v \in V$ its (ultimate) label $M(v)$. Let K_n be the disjunction of the labels $M(v)$ where v is a vertex of (V, E) of depth at most n: by Lemma 5, we have that $K_n \models_{A_I^E} K_{n-1}$ for some $n > 0$. This means that for every vertex v in (V, E) of depth at most n, we have that $M(v) \models_{A_I^E} \bigvee_{w \in C} M(w)$ where C is the set of vertexes of (V, E) of depth at most $n - 1$ whose label is A_I^E-satisfiable.

Let now i be large enough so that every non-leaf vertex of depth at most n in (V, E)—together with its ultimate label—is in P_i: we show that UNWIND should have terminated after P_i has been produced. There are two cases to consider. First, C is a covering for all labeled unwinding P_j such that $P_i \subseteq P_j$ and would cause UNWIND to terminate. Second, C is not a covering because C contains a leaf w. However $M(w)$ is A_I^E-satisfiable by the definition of C and is the ultimate label of w. Now we have that $M(w) \models pc = l_{\mathsf{init}}$, otherwise our fair strategy would have added some vertices as sons of w, because locations $l \neq l_{\mathsf{init}}$ are target locations. This means that a refinement step applies to w. Since $M(w)$ is A_I^E-satisfiable and is the ultimate label of w, this means that such refinement step must have reported the unsafety of $\mathcal{S}$. □

The hypotheses of Theorem 6 are rather restrictive when it comes to the analysis of imperative programs. Fragments of arithmetic play a central role in this domain and their usage in modeling operations on array indexes prevents the applicability of Theorem 6. For an application of this result, let us consider, therefore, a different application domain, like that of broadcast protocols (see, e.g., [34]). These are systems composed of a finite but arbitrary number of (identical) processes that can communicate by rendez-vous (a process sends a message to another) or broadcast (a process sends a message to all the others). Any such system can be specified by an array-based system $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; \{\tau_h\}_h \rangle$ for T_I the (pure) theory of equality (used to represent process identifiers) and T_E an enumerated data-type theory (representing the finite set of locations of each (identical) process) where $\mathbf{v} = \mathbf{a}, \mathbf{c}, \mathbf{d}$ and $\mathbf{a}$ contains just one array (associating a process identifier to the actual location reached by the process) whereas both $\mathbf{c}$ and $\mathbf{d}$ are empty. As shown in [46], it is possible to represent rendez-vous and broadcast of messages as guarded assignments in functional form (2). In [22], it is shown that the theories T_I and T_E satisfy the hypotheses of Theorem 6. Thus, UNWIND behaves as a decision procedure for the safety problem of broadcast protocols. A similar result using forward reachability has been proved in [36].

It is also possible to show that UNWIND behaves as a decision procedure for the safety problem of lossy channel system systems (see, e.g., [1]): their representation as array-based systems can be found in [46] and the fact that the latter satisfy the hypotheses of Theorem 6 is shown in [22].

7 Implementation and heuristics

The framework presented in the previous sections has been implemented in a tool called SAFARI, *SMT-based Abstraction For Arrays with Interpolants*, available at http://verify.inf.usi.ch/content/safari. Below, we discuss the implementation strategies and heuristics devised for the efficient execution of the UNWIND procedure.

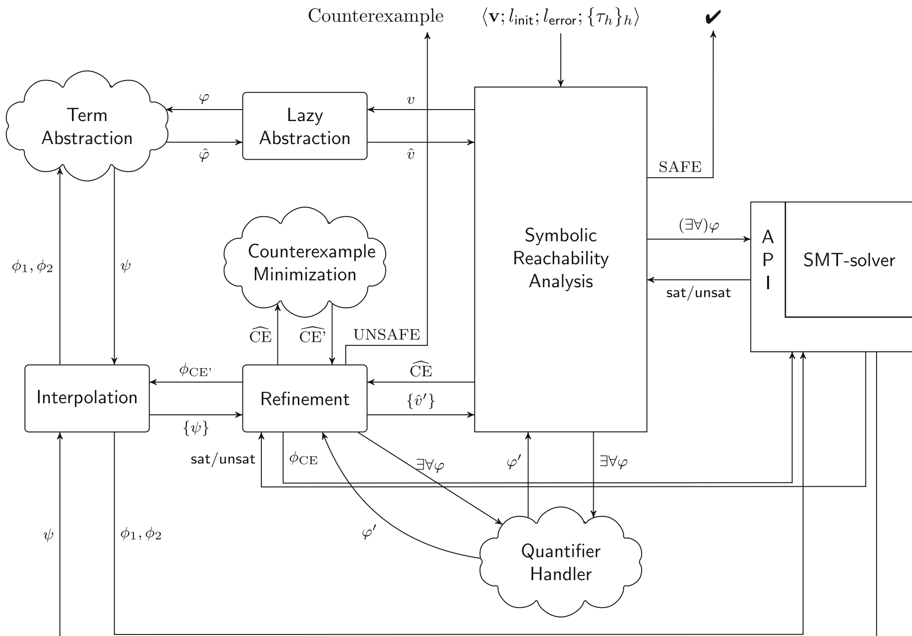

Fig. 5 The architecture of SAFARI

7.1 Implementation strategies and tool architecture

The architecture of the tool is depicted in Fig. 5. Modules drawn as square boxes represent usual modules of CEGAR-based model checkers with interpolation-based refinement. Those drawn as clouds constitutes the novel features of SAFARI.

Our tool maintains and modifies a labeled unwinding (V, E, M_V, M_E). We assume a total ordering $\preceq \subseteq V \times V$ respecting the ancestor relation. In our implementation, each vertex $v \in V$ is flagged as *free*, *covered* or *locked*. When created, all the vertices are free. A vertex v can become covered only if *i*) there exists a set of free vertices C such that (6) holds, i.e.

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

where $w \preceq v$ for all $w \in C$, and *ii*) all the vertices from v to ε are free. A vertex becomes locked when one of its ancestors gets covered.

The Symbolic Reachability Analysis module implements two procedures: EXPAND and REDUCE. The EXPAND procedure is in charge of expanding the labeled unwinding, as explained in Sect. 5. The practical implementation of this procedure, however, deviates from the high-level description provided in the previous sections by introducing some important optimizations. In our implementation EXPAND is applied only to free leaves. Every new leaf w generated by a vertex v is labeled with the preimage of $M_V(v)$ along the transition whose matrix is associated to $M_E(w, v)$. This allows to discover immediately trivial infeasible paths, i.e., those for which the preimage is A_I^E-unsatisfiable. The choice of the leaf to expand is also subject to several optimizations. As will be detailed later, the efficiency of the tool greatly depends on its ability to perform covering tests. Such tests are based on instantiation procedures whose complexity might badly affect the overall performance of

SAFARI. Also the exploration strategy (i.e. the selection of the leaves to expand) strongly affects the performance of the tool. We will describe the exploration strategy implemented in SAFARI later in Sect. 7.4, when heuristics and optimizations for efficient covering checks will be discussed. The other procedure implemented by this module, namely REDUCE, is in charge of limiting the growth of the labeled unwinding. It works by checking the vertices of the labeled unwinding with the goal of finding the covered or locked ones. REDUCE is eagerly applied before and after the EXPAND procedure. When applied before the expansion of the labeled unwinding, REDUCE checks if any vertex on the path from ε to the leaf selected for expansion is covered, starting from ε. Its application after the generation of the new leaves avoid their processing in case they are already covered. Indeed, only free newly generated vertices are passed to the Lazy Abstraction module. Given an abstracted leaf $\hat{v}$, it is checked if $M_V(\hat{v}) \wedge pc = l_{\mathsf{init}}$ is A_I^E-satisfiable. If so, the path from ε to $\hat{v}$, represented as $\widehat{\mathrm{CE}}$ in Fig. 5, is passed to the Refinement module. If all the leaves are flagged as covered or locked, the labeled unwinding is complete (recall Definition 3) and the set of free vertices is the covering associated to it. In this case, SAFARI reports that the system is *safe*.

The Lazy Abstraction module is in charge of abstracting labels of vertices in the unwinding. Remember that for every vertex v, $M_V(v)$ 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. This module returns a vertex $\hat{v}$ such that $M_V(\hat{v}) \models_{A_I^E} pc = l$ and $M_V(v) \models_{A_I^E} M_V(\hat{v})$.

The Refinement module implements the procedure described in Sect. 5.2. It takes as input a sequence of transitions representing a candidate counterexample, and it is in charge of generating a formula attesting its feasibility. If this module fails (i.e. the formula is unsatisfiable), then the Interpolation module comes into play, as in standard interpolation-based refinement procedures. In case the (external) SMT-Solver implements interpolating procedures, the Interpolation module can be bypassed by asking interpolants to the external tool. An abstract interface provides an API to separate the actual SMT-Solver used and the services which are requested by SAFARI. (The interface with external tools is based on the SMT-LIB v.2 standard [72].) Refining a path might result in uncovering some vertices. Refining a vertex in the covering set C triggers a procedure that checks if the covering relation (6) still holds or not, and modifies the labeled unwinding as a consequence of this fact: if a vertex v was covered by a refined vertex w, and this covering relation does not hold anymore, v is considered again as a free vertex, with any locked descendant.

7.2 Term abstraction

State-of-the-art interpolating procedures seldom allow the convergence of the model-checker on tricky examples. Divergence due to the inability of interpolation algorithms to come up with the "right" predicate has been already discussed in [58,59] in the context of verification of programs with scalar variables. Here, we propose a technique, called *Term Abstraction*, to tune interpolation algorithms in presence of array variables. The heuristic is implemented by the module Term Abstraction in the architecture of Fig. 5 and its goal is to compute (whenever possible) an interpolant where a certain set T of terms (called *undesired terms*), which are responsible for keeping interpolants too specific for the analyzed counterexample, do not occur. Ultimately, abstracting away undesired terms in T aims to avoid the divergence of the sequence of interpolants generated during unwinding calls. In particular, Term Abstraction is based on the preprocessing technique described in Sect. 3.1 that rewrite formulas of the form

$\psi(\cdots \mathbf{a}[\mathbf{c}]\cdots)$ to $\exists \underline{j}(\underline{j} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{j}]\cdots))$. More precisely, term abstraction works as follows.

Suppose we are given an unsatisfiable formula $\psi_1 \wedge \psi_2$ and the set $T = \{t_1, \ldots, t_n\}$ of undesired terms. We iteratively check if $\psi_1(c_i/t_i) \wedge \psi_2(d_i/t_i)$ is unsatisfiable, for c_i and d_i being fresh constants. If this is the case, we substitute ψ_j with $\psi_j(c_i/t_i)$ for $j = 1, 2$. Eventually, we are left with an unsatisfiable formula $\psi_1 \wedge \psi_2$, where some of the undesired terms in T might have been removed: the interpolant of ψ_1 and ψ_2, which can be computed with available interpolating procedures, is also likely not to contain the eliminated terms. SAFARI is capable of automatically computing a set of undesired terms from the input transition system by identifying loop iterators, variables representing the lengths of the arrays, or loop bounds. Alternatively, the user can suggest terms to be put in the set of undesired terms.

The experimental evaluation of SAFARI in Sect. 8 shows that Term Abstraction plays a crucial role in the success of SAFARI.

Example 2 Consider location l_2 in Fig. 2 corresponding to the end of the first loop in the Running procedure of Fig. 1. SAFARI has to generate the following invariant:

$$pc = l_2 \rightarrow \forall z_0.((0 \leq z_0 \wedge z_0 < L) \rightarrow (a[z_0] \geq 0 \leftrightarrow b[z_0])). \tag{21}$$

Key to generate this invariant is Term Abstraction. In the following, we explain how this is done. Consider the counterexample represented by the sequence of transitions $\tau_0, \tau_3, \tau_4, \tau_8, \tau_9$, generated by SAFARI during the verification of the Running procedure. To generate (21), we can consider the following two partitions:

$$B := \begin{pmatrix} mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], b[z_0], 1) \wedge \\ mov(l_1, l_2, 2) \wedge i^{(2)} = 0 \wedge i^{(1)} \geq L \wedge f^{(2)} \wedge id(a[z_0], b[z_0], 2) \wedge \end{pmatrix}$$

$$A := \begin{pmatrix} mov(l_2, l_2, 3) \wedge a^{(2)}[z_0] \geq 0 \wedge \neg b^{(2)}[z_0] \wedge i^{(2)} < L \wedge \\ z_0 = i^{(2)} \wedge i^{(3)} = i^{(2)} + 1 \wedge \neg f^{(3)} \wedge \\ mov(l_2, l_3, 4) \wedge i^{(3)} \geq L \wedge id(i, f, 4) \wedge \\ mov(l_3, l_4, 5) \wedge \neg f^{(4)} \wedge id(i, f, 5) \end{pmatrix}$$

An interpolant for these partitions is $I_1 := i^{(2)} < L$ since $A \models_{A_I^E} I_1$ and $I_1 \wedge B$ is A_I^E-unsatisfiable. Unfortunately, I_1 cannot be generalized to a quantified invariant as it contains no index variable.

Now, let $T = \{L, i\}$ be the set of undesired terms. The term abstraction procedure checks the unsatisfiability of $A(c/L) \wedge B(d/L)$ for the fresh constants c and d. The resulting formula is satisfiable, the procedure restores the original formulas A and B, and checks whether $A(c/i^{(2)}) \wedge B(d/i^{(2)})$ is unsatisfiable. In this case it succeeds and it is thus able to generalize over the variable i. The interpolant produced in this case is $I_2 := z_0 < L$. Beside being a correct interpolant for the two original partitions, since $A \models_{A_I^E} I_2$ and $I_2 \wedge B$ is A_I^E-unsatisfiable, I_2 can be generalized to a quantified property that constitutes one of the building blocks of (21).

7.3 Minimizing counterexamples

It is useful for Refinement to apply a *minimization* procedure to counterexamples with the goal of computing interpolants from a minimal (unsatisfiable) suffix of a trace containing the atom $pc^{(n)} = l_I$. We illustrate the advantages of this by considering the following situation.

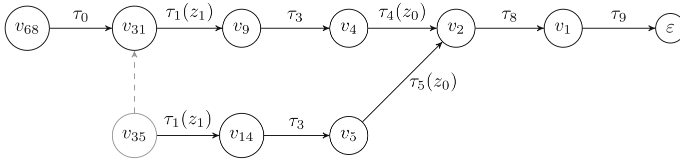

Fig. 6 Part of the labeled unwinding for the Running procedure. $M_V(v_{68}) \wedge pc = l_I$ is A_I^E-satisfiable and $M_V(v_{35})^{\exists} \models_{A_I^E} M_V(v_{31})^{\exists}$

Consider (part of) the labeled unwinding depicted in Fig. 6, generated by SAFARI while analyzing the Running procedure in Fig. 1. $M_V(v_{68}) \wedge pc = l_I$ is A_I^E-satisfiable, and v_{31} covers v_{35} since

$$M_V(v_{31}) := pc = l_1 \wedge i < L \wedge z_0 \neq z_1 \wedge a[z_1] \geq 0 \wedge z_1 = i$$

$$M_V(v_{35}) := \begin{pmatrix} pc = l_1 \wedge i < L \wedge z_0 \neq z_1 \wedge a[z_1] = 0 \wedge z_1 = i \wedge \\ b[z_0] \wedge z_0 = 0 \wedge L > 0 \wedge L \leq i + 1 \end{pmatrix}$$

The counterexample is represented by the following formula:

$$\begin{aligned}
&mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], a[z_1], b[z_0], b[z_1], 1) \wedge \\
&mov(l_1, l_1, 2) \wedge z_0 \neq z_1 \wedge i^{(2)} = i^{(1)} + 1 \wedge i^{(1)} > L \wedge z_1 = i^{(1)} \wedge a^{(1)}[z_1] \geq 0 \wedge id(f, a[z_0], b[z_0], 2) \wedge \\
&mov(l_1, l_2, 3) \wedge i^{(3)} = 0 \wedge L \leq i^{(2)} \wedge f^{(3)} \wedge id(a[z_0], a[z_1], 3) \wedge \\
&mov(l_2, l_2, 4) \wedge a^{(3)}[z_0] \geq 0 \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)} \wedge \\
&mov(l_2, l_3, 5) \wedge L \leq i^{(4)} \wedge id(i, f, 5) \wedge \\
&mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(i, f, 6)
\end{aligned}$$

The analysis of this counterexample can produce two different set of interpolants:

$$\begin{aligned}
&\quad \{\bot\} \\
&mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], a[z_1], b[z_0], b[z_1], 1) \wedge \\
&\quad \{i^{(1)} > z_0\} \\
&mov(l_1, l_1, 2) \wedge z_0 \neq z_1 \wedge i^{(2)} = i^{(1)} + 1 \wedge i^{(1)} > L \wedge z_1 = i^{(1)} \wedge a^{(1)}[z_1] \geq 0 \wedge id(f, a[z_0], b[z_0], 2) \wedge \\
&\quad \{z_0 < i^{(2)} \wedge z_0 \geq 0\} \\
&mov(l_1, l_2, 3) \wedge i^{(3)} = 0 \wedge L \leq i^{(2)} \wedge f^{(3)} \wedge id(a[z_0], a[z_1], 3) \wedge \\
&\quad \{z_0 < L \wedge i^{(3)} \leq z_0\} \\
&mov(l_2, l_2, 4) \wedge a^{(3)}[z_0] \geq 0 \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)} \wedge \\
&\quad \{\top\} \\
&mov(l_2, l_3, 5) \wedge L \leq i^{(4)} \wedge id(i, f, 5) \wedge \\
&\quad \{\top\} \\
&mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(i, f, 6) \\
&\quad \{\top\}
\end{aligned}$$

or

$$\{\perp\}$$
$$mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], a[z_1], b[z_0], b[z_1], 1) \wedge$$
$$\{\perp\}$$
$$mov(l_1, l_1, 2) \wedge z_0 \neq z_1 \wedge i^{(2)} = i^{(1)} + 1 \wedge i^{(1)} > L \wedge z_1 = i^{(1)} \wedge a^{(1)}[z_1] \geq 0 \wedge id(f, a[z_0], b[z_0], 2) \wedge$$
$$\{z_0 < i^{(2)} \wedge L \leq z_0 + 1\}$$
$$mov(l_1, l_2, 3) \wedge i^{(3)} = 0 \wedge L \leq i^{(2)} \wedge f^{(3)} \wedge id(a[z_0], a[z_1], 3) \wedge$$
$$\{z_0 = L - 1\}$$
$$mov(l_2, l_2, 4) \wedge a^{(3)}[z_0] \geq 0 \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)} \wedge$$
$$\{L \leq i^{(4)}\}$$
$$mov(l_2, l_3, 5) \wedge L \leq i^{(4)} \wedge id(i, f, 5) \wedge$$
$$\{\top\}$$
$$mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(i, f, 6)$$
$$\{\top\}$$

The analysis of the first counterexample allows for the refinement of vertices v_4, v_9, and v_{31}. The analysis of the second counterexample permits the deletion of vertex v_{31}, as the new label is unsatisfiable, and the refinement of vertices v_9, v_4, and v_2. Notice that the second case has the drawback of "uncovering" vertex v_{35}, that, before the refinement, was covered by v_{31} since

$$M_V(v_{35})^{\exists} \models_{A_I^E} M_V(v_{31})^{\exists}.$$

After the refinement such relation does not hold anymore and v_{31} can be explored again.

The goal of minimizing the counterexample is to save and preserve as much as possible the labeled unwinding. In fact, in the situation considered above, while the first set of interpolants refines only a small portion of the labeled unwinding, the second modifies a substantial part of the unwinding and destroys part of it. The flip side of this heuristic is that postponed inconsistencies in the data-flow might appear again in counterexamples generated by later calls of UNWIND, constituting the only unsat core of infeasible formula from which interpolants will be computed. In this case, the new set of interpolants would refine (and maybe destroy) the already specialized and well-refined peripheral parts of the labeled unwinding. In practice, our experience suggests that minimizing counterexamples pays off in most situations.

7.4 Instantiating universal quantifiers

The presence of quantified formulas can be problematic and requires particular attention in several phases of the analysis. Quantified formulas arise while checking covering tests and the feasibility of counterexamples. In particular, given the eager application of the REDUCE procedure, the vast majority of SAFARI execution time is spent for checking covering relations. As stated in Sect. 4.2, a vertex v is covered by a set of vertices C iff

$$M_V(v)^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists} \quad (22)$$

holds or, dually, if

$$M_V(v)^{\exists} \wedge \bigwedge_{w \in C} \neg \left(M_V(w)^{\exists}\right) \quad (23)$$

is A_I^E-unsatisfiable. Stack-handling procedures available in state-of-the-art SMT-Solvers allows to perform such a test in an incremental way, asserting few formulas representing the labels of the vertices in the set C at a time. As discussed in Sect. 4.2, (23) is a formula of the form

$$\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i}\, \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,, \tag{24}$$

where the $\underline{i}$ are the INDEX variables of the vertex v and $\underline{j}$ comes from the INDEX variables of vertices w. As said in Sect. 4.2, SAFARI deals with formulas of the form 24 by using an (incomplete) satisfiability procedure based on the instantiation of $\underline{j}$ over the set $\underline{i} \cup \mathbf{c}$ of variables. Considering all possible instances soon becomes infeasible as they are $|\underline{j}|^{|\underline{i} \cup \mathbf{c}|}$. Several heuristics are integrated in SAFARI to efficiently handle this instantiation process, part of which are inherited from the tool MCMT [47,48]. We discuss them in the rest of this section.

Exploration strategy. This heuristic addresses the problem of limiting the growth of the length of the tuple $\underline{j}$ of variables; recall that $\underline{j}$ represents, intuitively, the INDEX variables of the labels $M_V(w)$ in (23).

With standard exploration strategies, such as breadth- or depth-first search, the number of index variables labeling the leaves might grow very quickly. Notice that it is possible to predict the number e_k of (implicitly existentially quantified) index variables occurring in the formulas labeling the vertex v_k in a path of the form $\pi = v_0 \to \cdots \to v_m$ with $v_m = \varepsilon$ by simply counting the existentially quantified index variables in $\tau_{k+1} \wedge \cdots \wedge \tau_m$ from (11). In fact, the number of index variables that will occur in the formula labeling v_k after the update (20) is bounded by e_k, because it is derived from the interpolants computed along the path π above.

Heuristics [45,47] designed to reduce the number of index variables in preimages developed for the backward reachability procedure of MCMT can also be put to productive use in SAFARI. These heuristics affect the selection of leaves in the EXPAND procedure, promoting the expansion of leafs with a small number of index variables. SAFARI keeps an ordered list of leaves of the tree. The ordering of the leaves is firstly based on the number of INDEX variables, and secondly, if the number of INDEX variables is equal, on the $\preceq$ relation introduced in previous section. The effect of maintaining such a list is that EXPAND works always on a leaf with the smallest number of variables. Such a smart exploration strategy helps also during refinement, where quantified queries (expressing trace feasibility) are Skolemized and instantiated, thus producing equisatisfiable quantifier-free queries on which interpolation algorithms are executed.

Filtering instances. Adopting a smart exploration strategy helps in alleviating the burdens on the default quantifier instantiation procedure described in Sect. 4.2. Even if the problem of checking satisfiability of quantified formulas attracted a lot of interest recently (e.g., [32,43,44]), efficient solutions have been implemented only in few SMT-Solvers. We describe here another optimization devised for reducing the impact of our default instantiation procedure on the performances of SAFARI even more. This other optimization plays a significant role in the instantiation process, especially when checking covering of vertices, aims to reducing the instantiations performed for each covering test. Such optimization is based on the *filtering modulo enumerated data-type* [45] heuristics. They cut the number of instantiations of the universally quantified variables by exploiting cheap checks involving information cached in specific data-structures used to represent formulas.

Primitive differentiated form. SAFARI inherits from MCMT the feature of keeping all formulas labeling vertices of the unwinding in a primitive differentiated form. An $\exists^I$-formula

$\exists \underline{i}.\phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ is *primitive* iff it is a conjunction of literals and is *differentiated* iff it contains the negative literal $i_k \neq i_l$ for every $i_k, i_l \in \underline{i}$. Notably, this format avoids the computationally expensive enumeration of partitions in the interpolation algorithm described in Sect. 5. Primitive differentiated form helps also in reducing the number of possible instantiations while checking the unsatisfiability of formulas of the form (24).

8 Experiments

We have run SAFARI against safety problems that require reasoning on arrays of unknown length (the benchmarks are illustrated in Sect. 8.1). The goal of the experimental analysis is two-fold. First, we want to measure the impact of the heuristics *Term Abstraction* (TA) and *Counterexample Minimization* (CM) discussed in Sect. 7.2 and Sect. 7.3, respectively (our findings are reported in Sect. 8.2). Second, we want to conduct a comparison with state-of-the-art tools implementing alternative approaches to the verification of programs manipulating arrays (Sect. 8.3). In particular, we consider a state-of-the-art software model checker and a verifier based on abstract interpretation.

8.1 Benchmarks

Our problems are divided in two benchmark suites:

- SUITE 1 consists of 13 of the 28 problems (both safe and unsafe) considered in [35]. The programs in the problems perform simple manipulations on arrays; e.g., copying an array into another, concatenating two arrays, and swapping the content of two arrays. The safety properties are expressed by loops containing quantifier-free assertions (similarly to what is done in Fig. 1 for the procedure Running). Each problem in SUITE 1 is labeled by "Dn" where n is a natural number used to identify the problem in [35]. Since our tool is capable of natively supporting quantified assertions (such as (1) for the procedure Running), from each problem "Dn" we have derived a new (equivalent) problem identified with "QDn" by replacing the loop (or loops) encoding the safety property with the corresponding quantified property. There are no problems "QD06" and "QD17" since the quantified properties require the use of divisibility predicates in Linear Arithmetic or the introduction of an alternation of quantifiers. Both cases are beyond the expressiveness of the language currently taken in input by SAFARI. There are two reasons for the exclusion of 15 problems in [35]. First, some of the problems in [35] require interpolants over Linear Arithmetic while the actual implementation of SAFARI is only able to compute interpolants over $\mathcal{IDL}$. (This is not a conceptual but a technological limitation that will be overcome in future releases of SAFARI by incorporating interpolation capabilities for Linear Arithmetic.) Second, the remaining problems have been discarded because of the presence of C functions, such as `buffer_size`, that are not related to the kind of (quantified) array properties of interest to us in this work.
- SUITE 2 contains 25 programs taken from several sources, e.g., the benchmark suite of BOOGIE[7] and WHY3,[8] papers [9,56] on tools related to SAFARI, books on algorithms and data structures (such as [78]), standard C string functions library, and problems suggested by experts in the area. Each program generates both a safe and an unsafe problem; the

[7] http://research.microsoft.com/en-us/projects/boogie/.

[8] http://proval.lri.fr/.

latter obtained from the former by manually inserting a bug in the problem. The programs in SUITE 2 can be briefly described as follows:

- *binarySort* is an implementation of the “binary sort” algorithm in [78]. We check that, once the procedure terminates, the array is sorted.
- *bubbleSort* is an implementation of the “bubble sort” algorithm in [9]. We check that, once the procedure terminates, the array is sorted.
- *comp* implements the `strcmp` function in [56] for comparing the content of two arrays. This function returns true if the two input arrays are equal. We check that if the procedure returns true, the two input arrays are indeed equal.
- *compM* is a modified version of *comp* where the first equal segment of two arrays is copied in a third one. This function returns true if the two input arrays are equal. We check that if the procedure returns true, the two input arrays are indeed equals and also that the local copy of the array is equal to the input array.
- *copy* implements the `strcpy` function in [56] for copying the content of an array into another. The property we check is that, at the end of the procedure, the input array has been correctly copied in the returned one.
- *copyN* is a modified version of *copy* where the content of the input array is copied in N arrays (one at a time) before being copied in the last array. We check that, in the end, the N-th copied array is equal to the first one.
- *find* implements the linear search algorithm in [56]. Such function returns the smallest index of the array where the element of interest is stored. We check that if the procedure returns a value bigger than the size of the array, the array does not contain the given element to search for.
- *findTest* is an extended version of *find* with an extra loop that checks if the returned index is the smallest one storing the given element that has been searched for. If so the function returns true. We check that the function always returns such a value.
- *heapArr* - Benchmark where the heap (abstracted as an array) is modified only in some parts. Since the postcondition asserts facts on a bigger portion, the tool has to infer that for any position outside the modified ones, the heap remained untouched. (This example has been kindly suggested by K. Rustan M. Leino).
- *init* implements the procedure in [56] to initialize all the cells of an array to some value. We check that, at the end of the procedure, the array has been correctly initialized.
- *initTest* is an extended version of *init* with an extra loop checking that the array has been initialized. This function returns true if the extra loop does not find any error. We verify that the procedure always returns true.
- *maxInArr* and *minInArr* implement linear search procedures for largest and smallest, respectively, values in an array (taken from http://proval.lri.fr/). We check that the functions respectively correctly return the biggest or smallest value of the array.
- *nonDisj* is a procedure that takes in input an array a of integers and saves in a local array variable b all the position i where $a[i] > 0$, such that the property $a[b[j]] > 0$ is satisfied for all the element j such that $b[j]$ is smaller than than the size of a. We check that this property is satisfied by every position of b that has been initialized by the procedure.
- *partition* implements an algorithm to distribute the content of an array in two: one holding all non-negative values and the other all the negative values (taken from [56]). We check that the two target arrays contains only non-negative and positive values, respectively.
- *running* is the procedure in Fig. 1. We check that assertion (1) is never violated.
- *vararg* is the procedure in [56] searching for the first position of the input array storing the symbolic constant *NULL*, marking the point up to which the array has been initialized.

We check that the procedure returns the first position where the input array contains the value *NULL*.

To quantitatively characterize the problems in the two benchmark suites, we have identified the following three parameters: the numbers L and N of non-nested and nested, respectively, loops in the body of the program and the number Q of quantifiers in the safety property. The interest of these figures lies in the fact that SAFARI, like any tool based on a CEGAR-like strategy, suffers from

- the presence of several non-nested loops in the program. This is because each counter-example found by unwinding must go through the L loops. Thus, refinement should be able to generalize the invariants for all the L loops from the same (inconsistent) formula representing the (infeasible) counter-example. In this respect, the problems identified by "copy$N$," where N represents the number of loops in the program, in SUITE 2 are particularly relevant (notice that L = N).
- the "depth" N of nested loops.[9] The problem is that the infeasibility of a counter-example may derive from the interaction of variables that are updated in two or more nested loops. For example, in the case of two nested loops, the behavior of the inner loop is influenced by the operations performed in the outer loop. The interplay among the variables is indeed reflected in the counter-example found by unwinding and refinement must then be able to synthesize an invariant describing the possibly complex relationships among the elements stored in several array variables. In this respect, the problems *binarySort* and *bubbleSort* in SUITE 2 are particularly interesting because they contain two nested loops (N = 1).
- the presence of a number Q of quantifiers in the property to be verified. The crucial observation here is that unbounded arrays (i.e. of finite but unknown dimension) require the capability of identifying quantified predicates for synthesizing the invariants for discharging the safety property.
  So, the higher the number Q of quantified variables in the property, the higher the complexity of finding quantified predicates that imply the property. In this respect, the problems identified by "QDn" in SUITE 1 are particularly relevant (notice that Q = 1). In fact, comparing the performances of SAFARI on "Dn" and "QDn" will give an idea of the advantages and disadvantages of using properties expressed by quantified (Q $>$ 0) and quantifier-free (Q = 0) assertions, respectively.

8.2 Importance of the heuristics

We now show that the heuristics *Term Abstraction* (TA) and *Counterexample Minimization* (CM)—described in Sect. 7.2 and Sect. 7.3, respectively—are key to the scalability of SAFARI. To show this, we have run SAFARI on both benchmark suites with the heuristics turned on and off. All the experiments have been conducted on a computer equipped with an Intel(R) Core(TM)2 Quad CPU @ 3.00GHz and 12 GB of RAM running Linux Debian "jessie." The complete benchmark suites and the executable of SAFARI used for the evaluation are available at http://verify.inf.usi.ch/content/safari. The results are reported in Table 1 for SUITE 1 and Table 2 for SUITE 2.

In both tables, the column 'PB.' reports the identifier of the problem together with the tuple (L, N, Q) representing the number of loops, maximum level of nesting, and number

[9] N = 0 means that the program does not have nested loops, N = 1 identifies programs with at least one nested loop, etc.

Table 1 Experiments on SUITE 1: statistics for SAFARI with different heuristics turned on and off

PB. (L,N,Q)	STATUS	NOA	NOH	CM	TA	CMTA
TIMINGS [TIME OUT = 3600] (IN SECONDS)						
D01 (2,0,0)	Safe	x	–	–	0.36	0.38
D02 (2,0,0)	Safe	x	–	–	0.39	0.28
D03 (2,0,0)	Safe	x	–	–	0.37	0.52
D04 (2,0,0)	UnSafe	3.92	0.51	0.30	0.18	0.28
D06 (2,0,0)	UnSafe	x	–	–	2.68	0.78
D08 (2,0,0)	Safe	x	–	–	0.36	0.50
D09 (2,0,0)	Safe	x	–	–	0.50	0.40
D11 (2,0,0)	UnSafe	1.54	0.35	0.28	1.53	1.02
D13 (2,0,0)	UnSafe	0.45	0.42	0.34	0.33	0.45
D14† (4,0,0)	Safe	x	–	–	1.60	1.06
D15 (4,0,0)	UnSafe	2.62	1.60	1.33	1.46	1.56
D16† (5,0,0)	Safe	x	–	–	2.22	1.10
D17 (2,0,0)	Safe	x	0.72	0.80	x	0.68
D20 (2,0,0)	Safe	x	–	–	0.81	0.47
QD01 (1,0,1)	Safe	x	x	x	0.38	0.39
QD02 (1,0,1)	Safe	x	x	x	0.43	0.35
QD03 (1,0,1)	Safe	x	x	x	0.36	0.38
QD04 (1,0,1)	UnSafe	0.34	0	1.44	0.31	0.37
QD08 (1,0,1)	Safe	x	x	x	0.36	0.21
QD09 (1,0,1)	Safe	x	x	x	0.43	0.44
QD11 (1,0,1)	UnSafe	0.46	0	0.36	0.63	0.58
QD13 (2,0,2)	UnSafe	0.44	0	0.41	0.61	0.35
QD14† (3,0,1)	Safe	x	x	x	0.78	0.64
QD15 (3,0,1)	UnSafe	0.53	4	2.62	1.09	0.94
QD16† (4,0,1)	Safe	x	–	–	1.38	1.14
QD20 (1,0,1)	Safe	x	x	x	0.37	0.28
NUMBER OF REFINEMENTS [MAXIMUM = 150]						
D01 (2,0,0)	Safe	x	–	–	5	3
D02 (2,0,0)	Safe	x	–	–	5	3
D03 (2,0,0)	Safe	x	–	–	5	3
D04 (2,0,0)	UnSafe	0	0	0	0	0
D06 (2,0,0)	UnSafe	x	–	–	2	2
D08 (2,0,0)	Safe	x	–	–	5	3
D09 (2,0,0)	Safe	x	–	–	5	3
D11 (2,0,0)	UnSafe	0	0	0	0	0
D13 (2,0,0)	UnSafe	0	0	0	0	0
D14† (4,0,0)	Safe	x	–	–	8	8
D15 (4,0,0)	UnSafe	0	6	4	9	9
D16† (5,0,0)	Safe	x	–	–	18	14
D17 (2,0,0)	Safe	x	3	3	x	4
D20 (2,0,0)	Safe	x	–	–	5	3

Table 1 continued

PB. (L,N,Q)	STATUS	NOA	NOH	CM	TA	CMTA
QD01 (1,0,1)	Safe	x	x	x	2	2
QD02 (1,0,1)	Safe	x	x	x	2	2
QD03 (1,0,1)	Safe	x	x	x	2	2
QD04 (1,0,1)	UnSafe	0	0	0	0	0
QD08 (1,0,1)	Safe	x	x	x	2	2
QD09 (1,0,1)	Safe	x	x	x	2	2
QD11 (1,0,1)	UnSafe	0	0	0	0	0
QD13 (2,0,2)	UnSafe	0	0	0	0	0
QD14† (3,0,1)	Safe	x	x	x	6	6
QD15 (3,0,1)	UnSafe	0	4	3	7	7
QD16† (4,0,1)	Safe	x	–	–	12	12
QD20 (1,0,1)	Safe	x	x	x	2	2

'x' indicates that SAFARI was not able to converge in the given time out of 1 h. '–' indicates that SAFARI was not able to converge with less than 150 refinements. The examples labeled with † have been pre-processed with *loop fusion*, a compiler optimization technique which replaces multiple loops (iterating over the same range) with a single one when the instructions in the body of a loop do not interfere with those in the bodies of the others (see, e.g., [2])

of quantified variables in the assertions, respectively (see Sect. 8.1 for a description). Since SUITE 1 contains both safe and unsafe problems, the column 'STATUS' of Table 1 reports if the problem is safe or unsafe. Since SUITE 2 contains a safe and an unsafe version of the same problem, Table 2 groups the statistics of SAFARI for the safe and unsafe variants of the same problem. Both Tables 1 and 2 are organized in two sub-tables: the first for the timings (in seconds with a time out of 1 h) and the second for the number of refinements (with a maximum of 150) used by SAFARI. Each sub-table reports measures (time or number of refinements) for the following configurations of SAFARI: no use of abstraction (NOA), i.e. SAFARI performs backward reachability, use of abstraction with both heuristics switched off (NOH), use of abstraction with only Counter-example Minimization turned on (CM), use of abstraction with only Term Abstraction turned on (TA), use of abstraction with both heuristics turned on (CMTA).

The results reported in the tables show the importance of heuristics for the scalability of SAFARI. Heuristics play a crucial role in allowing SAFARI to converge on safe programs: without them, in fact, SAFARI is almost never able to converge as shown by looking at the columns NOH in both Tables 1 and 2. We also observe that the role of the two heuristics is quite different. In fact, Counter-example Minimization alone allows SAFARI to converge on few more examples than when the tool is executed without options (compare the columns NOH and CM in the tables). Instead, Term Abstraction alone enables SAFARI to converge on many more problems (compare the columns NOH and TA in the tables). The problems on which SAFARI fails to converge with Term Abstraction only turned on are successfully verified by using both heuristics (compare the columns TA and CMTA in the tables). We can explain the differences in the impact of the heuristics as follows.

Recall from Sect. 7.2 that Term Abstraction allows SAFARI to induce the interpolation procedure to return an interpolant that could be potentially more useful for refinement. In other words, Term Abstraction has an impact on *how* a counter-example is refined. Instead, Counterexample Minimization (recall Sect. 7.3) tries to find the smallest unsatisfiable suffix

Table 2 Experiments on SUITE 2: statistics for SAFARI with different heuristics turned on and off

PB. (L,N,Q)	SAFE				UNSAFE				
	NOH	CM	TA	CMTA	NOA	NOH	CM	TA	CMTA
TIMINGS [TIME OUT = 3600] (IN SECONDS)									
BinarySort (3,1,2)	–	0.93	4.20	2.81	3.95	27.22	–	8.26	6.53
BubbleSort (2,1,2)	–	–	1.20	0.97	1.04	14.73	13.84	8.89	8.26
Comp (1,0,1)	x	x	0.25	0.40	0.32	0.36	0.39	0.34	0.40
CompM (1,0,1)	x	x	0.67	0.53	0.38	0.49	0.58	0.36	0.48
Copy (1,0,1)	x	x	1.58	0.23	0.28	0.29	0.41	0.19	0.34
Copy2 (2,0,1)	–	–	x	0.61	0.33	0.44	0.49	0.33	0.45
Copy3 (3,0,1)	–	–	x	1.02	0.39	0.57	0.67	0.51	0.57
Copy4 (4,0,1)	–	–	x	1.77	0.45	0.89	0.88	0.64	0.78
Copy5 (5,0,1)	–	–	x	3.47	0.50	1.19	1.15	0.83	0.98
Copy6 (6,0,1)	–	–	x	6.73	0.57	1.56	1.52	1.20	1.22
Copy7 (7,0,1)	–	–	x	9.27	0.64	2.13	1.93	1.23	1.51
Copy8 (8,0,1)	–	–	x	15.89	0.67	2.81	2.48	1.40	1.76
Copy9 (9,0,1)	–	–	x	24.84	0.72	3.36	3.14	1.71	2.17
Copy10 (10,0,1)	–	–	x	36.45	0.80	4.84	3.92	2.59	2.57
Find (1,0,1)	x	x	0.42	0.60	0.28	0.23	0.34	0.21	0.36
FindTest (2,0,0)	x	–	1.33	1.22	0.41	0.63	1.36	0.59	0.85
HeapArr (1,0,0)	5.56	3.85	0.80	0.88	0.34	0.75	0.85	0.31	0.51
Init (1,0,1)	x	x	0.37	0.30	0.29	0.17	0.28	0.17	0.31
InitTest (2,0,0)	–	–	x	1.53	0.35	0.40	0.54	0.26	0.42
MaxInArr (1,0,1)	–	–	0.43	0.30	0.29	0.29	0.42	0.23	0.38
MinInArr (1,0,1)	–	–	0.43	0.46	0.29	0.30	0.42	0.23	0.39
NonDisj (1,0,2)	–	–	0.60	0.70	0.55	0.59	0.69	0.54	0.76
Partition (1,0,1)	x	x	0.48	0.53	2.24	1.81	1.86	0.38	0.61
Running (2,0,0)	x	x	0.92	0.87	0.28	0.44	0.47	0.29	0.46
Vararg (1,0,1)	x	x	0.44	0.46	0.19	0.27	0.30	0.21	0.35
NUMBER OF REFINEMENTS [MAXIMUM = 150]									
BinarySort (3,1,2)	–	7	21	21	0	61	–	6	6
BubbleSort (2,1,2)	–	–	5	5	0	39	39	14	14
Comp (1,0,1)	x	x	2	2	0	1	1	1	1
CompM (1,0,1)	x	x	4	4	0	3	3	2	2
Copy (1,0,1)	x	x	2	2	0	2	2	1	1
Copy2 (2,0,1)	–	–	x	6	0	2	2	2	2
Copy3 (3,0,1)	–	–	x	12	0	3	3	3	3
Copy4 (4,0,1)	–	–	x	20	0	4	4	4	4
Copy5 (5,0,1)	–	–	x	30	0	5	5	5	5
Copy6 (6,0,1)	–	–	x	42	0	6	6	6	6
Copy7 (7,0,1)	–	–	x	56	0	7	7	7	7
Copy8 (8,0,1)	–	–	x	72	0	8	8	8	8
Copy9 (9,0,1)	–	–	x	90	0	9	9	9	9
Copy10 (10,0,1)	–	–	x	110	0	10	10	10	10

Table 2 continued

PB. (L,N,Q)	SAFE				UNSAFE				
	NOH	CM	TA	CMTA	NOA	NOH	CM	TA	CMTA
Find (1,0,1)	x	x	3	4	0	1	1	1	1
FindTest (2,0,0)	x	–	14	19	0	6	13	8	8
HeapArr (1,0,0)	68	54	9	9	0	9	9	4	4
Init (1,0,1)	x	x	2	2	0	0	0	0	0
InitTest (2,0,0)	–	–	x	11	0	3	3	1	1
MaxInArr (1,0,1)	–	–	3	3	0	2	2	2	2
MinInArr (1,0,1)	–	–	3	3	0	2	2	2	2
NonDisj (1,0,2)	–	–	0	0	0	4	4	5	5
Partition (1,0,1)	x	x	1	1	0	7	7	2	2
Running (2,0,0)	x	x	6	10	0	2	2	3	3
Vararg (1,0,1)	x	x	4	4	0	1	1	2	2

'x' indicates that SAFARI was not able to converge in the given time out of 1 h. '–' indicates that SAFARI was not able to converge in less than 150 refinements. We do not report the column NOA for safe problems since SAFARI always diverges on them when abstraction is disabled

of the counter-example in order to prune the search space as much as possible. In other words, Counterexample Minimization addresses the problem of finding *where* to refine a counter-example. So, Term Abstraction alone is sufficient when the counter-examples to be refined are not long and it is thus crucial how refinement is performed. When counter-examples become longer, it is also important where to refine them, not only how. On such problems, it is only the combination of the two heuristics that is winning.

We conclude by observing that in case of unsafe problems, the overhead of using abstractions with the heuristics turned on is small (compare the columns NOA and CMTA in the tables for unsafe problems).

8.3 Comparison with other tools

We now compare SAFARI with other state-of-the-art program verification tools. Our goal is to comparatively evaluate the effectiveness of alternative techniques with respect to those in SAFARI on the verification of programs manipulating unbounded arrays. Among the available alternatives, we have selected three well-known tools: CBMC [27], a tool based on Bounded Model Checking (BMC), CPACHECKER [15], a tool based on lazy abstraction and interpolation-based refinement, and CLOUSOT [40], a recent tool based on Abstract Interpretation.[10] The comparison with CBMC and CPACHECKER shows the advantages of the capability of SAFARI to reason about unbounded arrays over the other approaches which requires to fix their dimension. Indeed, neither CBMC nor CPACHECKER support the analysis of programs with unbounded arrays. For our comparison, we consider the problem *bubbleSort*. From Table 2, we know that SAFARI solves this safety problem in 0.97 s when run with both heuristics turned on. For CBMC and CPACHECKER we consider increasing sizes of the array *a* with

[10] Notice that CBMC and CPACHECKER won the first and second place, respectively, of the overall category in the 3rd International Competition on Software Verification (SV-COMP'14), http://sv-comp.sosy-lab.org/2014/results/index.php.

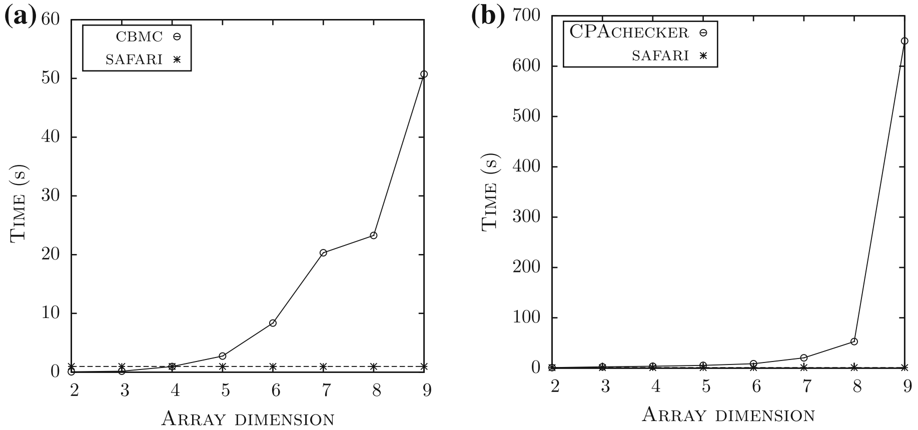

Fig. 7 Running times for SAFARI and CBMC and CPACHECKER on the *bubbleSort* algorithm. SAFARI execution time is 0.97 s

N ranging from 2 to 9. Notice that, since all these tools do not support quantified assertions, we need to express it by using two nested loops as follows:

```
for ( int x = 0 ; x < N ; x++ ) {
  for ( int y = x+1 ; y < N ; y++ ) {
    assert( a[x] <= a[y] );
  }
}
```

Comparing SAFARI with CLOUSOT highlights instead the need of using increasingly precise abstractions obtained by refinement as done by SAFARI with respect to the adoption of coarser abstractions—due to the application of widening and join operators—for scalability as done by CLOUSOT.

SAFARI*vs* CBMC. BMC [16] is a verification approach based on unrolling, a bounded number κ of times, the control-flow graph of a program. The feasibility of executions (of length up to κ included) leading from an initial to an error state is reduced to the satisfiability of a Boolean formula, that can be checked with available SAT solvers. The method is, in general, incomplete since it can prove the presence or absence of bugs for executions with bounded length. In some situations, however, it is possible to establish a value for κ which is sufficient to consider to guarantee the safety of executions of the program of arbitrary length. For example, consider the problem *bubbleSort*: given the size N of the array a to be sorted, it is sufficient to take $\kappa = N + 1$ to establish the safety of the program with respect to the following (quantified) post-condition:

$$\forall x, y.((0 \leq x < y < N) \rightarrow a[x] \leq a[y])$$

since the number of iterations of the loop in the program is a function of the size N of the array a.

Figure 7a reports running times for CBMC on the *bubbleSort* algorithm for increasing values of the size N of the array a.[11] The plot shows the execution time in function of the size of the array. As expected, SAFARI outperforms CBMC especially for larger values of N.

[11] We run CBMC v4.3 with the option `-unwind` N`+1`.

SAFARI*vs* CPACHECKER. CPACHECKER is a tool implementing a lazy abstraction and interpolation-based refinement framework. The key difference of CPACHECKER with respect to other tools is the availability of several different techniques which can be combined together. For our experiments, we run CPACHECKER v1.2 with the option `-predicateAnalysis-PredAbsRefiner-ABEl-UF`.[12]

Figure 7b reports running times for CPACHECKER on the *bubbleSort* algorithm for increasing values of the size N of the array a. This time as well the execution time of CPACHECKER rapidly grows with respect to the size of the array, while SAFARI execution time does not depend on the size of the array.

SAFARI*vs* CLOUSOT. Abstract Interpretation [28] is a static analysis technique based on sound approximations of the semantics of programs, obtained by using monotonic functions over ordered sets. In general, tools based on Abstract Interpretation trade efficiency for precision (see Sect. 9 for a more detailed discussion about abstract interpretation). Instead, SAFARI tries to gain more and more precision by using suitable refinements. We believe this is crucial for successfully handling (quantified) assertions about arrays of unbounded size. To check this intuition, we compare SAFARI with CLOUSOT, a recent tool based on Abstract Interpretation, under active development at Microsoft Research, which is capable of handling programs manipulating arrays (as discussed in [29]). The goal of the comparison is to evaluate the success rate of CLOUSOT on the problems in SUITE 2 rather than its efficiency. For this, we have run the on-line version of the tool available at http://rise4fun.com.[13] Our findings are the following. On the safe versions of the 25 programs in SUITE 2, CLOUSOT is able to verify only 4 programs (namely, *find*, *init*, *partition*, and *vararg*) while on the unsafe versions is able to identify the bug for 2 programs only (namely, *partition* and *vararg*). This confirms our intuition that the trade-off between precision and efficiency in CLOUSOT is not satisfactory when (quantified) assertions about array programs are to be verified.

8.4 Discussion

We can summarize the findings of the experimental analysis as follows.

The success of SAFARI is determined by a careful tuning of precision in the refinement phase of the CEGAR loop on which the tool is based. In particular, Term Abstraction is capable of inducing the interpolation procedure to provide the “right” interpolants, i.e. formulas that give rise to a more precise but not too precise abstraction of the program so as to permit SAFARI to converge. When counter-examples are longer, the use of Counter-example Minimization in conjunction with Term Abstraction becomes crucial to drive the refinement procedure towards a good and successful refinement of the abstract model.

The capability of specifying quantified assertions and reasoning about arrays of unbounded length allows SAFARI to consider compact annotations and verify programs regardless of the number of cells in an array. This makes the results of the verification more useful since safety holds for arrays of finite but arbitrary size and, at the same time, may improve performance by using compact (symbolic) representations of the set of (backward) reachable states during unwinding.

To conclude, we believe that SAFARI should be part of the toolkit of software verifiers since, on selected pieces of code, it complements available techniques (e.g., those based on

[12] We would like to thank Dirk Beyer and its group for their support in running CPACHECKER.

[13] We were not able to retrieve the version of CLOUSOT invoked by the web interface. We assume it to be the last available version, i.e. 1.5.60502.11.

Abstract Interpretation), when these fail because of the use of too coarse abstractions that trade precision for scalability.

9 Related work

A long list of sound and efficient techniques for the analysis of programs handling data-structures is available in the literature. Below, we discuss the relevant works classified according to the main technique they use as follows: predicate abstraction with counterexample guided abstraction refinement procedures, abstract interpretation, theorem proving-based, shape analysis and template-based solutions.

9.1 Predicate abstraction

Since the seminal paper [50], *Predicate abstraction* has become a very popular technique in software verification. It allows an abstraction of the concrete semantics of the program to an abstract semantics where reachable states of the programs are grouped according to the predicates they satisfy.

In presence of unbounded data-structure, like the programs we target in this paper, predicate abstraction has to work with quantified predicates. One of the first approaches for software verification based on predicate abstraction and able to handle quantified predicate is in [41]. This solution exploits *ghost* variables, i.e., Skolem constants which are never modified by the program. Ghost variables, once the procedure terminates, are not assigned to a precise value and hence can be universally quantified. The *index predicate* solution [63] fixes the number of “index variables”, i.e., universally quantified variables, in order to exploit standard predicate abstraction algorithms. For such two solutions predicates are generally suggested by the user. The work in [62] proposes a refinement technique based on the weakest precondition, in charge of generating new intermediate annotations. The main limitation of the aforementioned approaches is their inability of generating quantified predicates. These approaches would be inefficient, therefore, on programs without quantified post-conditions or assertions like those considered in part of our experimental analysis. The generation of quantified predicates has been addressed also by Jhala and McMillan in [59], as an extension of their previous work [58]. The interpolation procedure is driven by new axioms with the goal of generating quantified predicates, called *range predicates*, representing properties for ranges of cells in the arrays. While such predicates are restricted to a particular shape, this is not the case of our technique. The algorithm implemented in the ACSAR model checker [76] adopts a backward reachability procedure in which new predicates are generated by simulating the “pre” operator on spurious counterexamples. This constitutes the main difference with respect to our approach, which performs refinement by means of interpolants. Invariants and predicates can also be generated by analyzing the postcondition with some patterns, like *variable aging* or *constant relaxation* [42]. This approach can generate invariants for many interesting problems, like sorting algorithms. On the other hand, it cannot handle programs which require quantified invariants but do not have quantified assertions in their specifications.

Arrays can also represent a contiguous, fixed-size, portion of memory. For this class of programs, blasting every cell of the array as a single, uncorrelated variable results in inefficient procedures, as pointed out by in [8,9], which present an abstraction-refinement procedure for linear programs with fixed-size arrays.

9.2 Abstract interpretation

The approach described in this paper aims at developing a sound analysis procedure at the price of non-termination. Our solution does not suffer from the loss of precision deriving from the use of approximation techniques and, upon termination, returns either an invariant, which is both safe and inductive, or a real counterexample. *Abstract Interpretation* (AI) approaches target efficiency, i.e., they aim to generate inductive (but not necessarily safe) facts at compile-time. The application of widening operators, required to ensure the convergence of the analysis, may cause loss of precision, though, with the result that inferred inductive properties might be too weak to prove the absence of paths violating a given property.

AI solutions rely on the availability of some *abstract domains* for inferring invariants. An abstract domain can be thought of as a (fragment of a) theory [51] identifying a class of formulas over which the concrete semantics of the input program is abstracted. Since the seminal paper [28], several domains (such as interval arithmetic [28], octagons [69], octahedra [25], and convex-polyhedra [30]) have been studied in order to reason about different properties of programs.

AI analysis for arrays can be performed by associating one abstract value to each cell of the array or by *smashing* array variables, i.e., using one abstract value representing all the possible values of the array [17]. The first approach is precise but extremely inefficient while the second, on the contrary, is much more efficient at the price of (greatly) degrading precision. Other approaches segment either syntactically [49,52] or semantically [29] an array and assign to each segment an abstract value.

The long-term project Code Contracts[14] carried on at Microsoft Research has obtained very good results and its value in both the academic and industrial scenarios should not be neglected. The project supports static verification of programs with several analysis tools, many of which are based on AI techniques; such as Clousot, discussed in Sect. 8.3.

It is worth to notice that abstract interpretation and CEGAR-based approaches are not mutually exclusive. They have been successfully combined, for example, in recent work [3].

9.3 Theorem proving

Inference of quantified array properties is the goal of the techniques in [56,61,68]. The generation of quantified predicates relies on the use of saturation-based theorem proving (i.e. resolution extended with inferences to reason about equalities) combined with interpolation [56,68] or the solution of recurrence relations [61].

Invariants produced by these approaches may be more expressive than those found by our technique; for instance, they may contain alternations of quantifiers. Indeed, considering a larger class of properties makes the problem of avoiding divergence even more acute than in our setting. The situation is further complicated by the fact that saturation-based theorem provers need to be instructed with axioms for handling arithmetic and this may, in practice, further contribute to the non-termination of the inference process (theoretically, satisfiability of arbitrary first-order formulas is semi-decidable). Instead, our approach relies on SMT-Solvers to take care of the arithmetic operations arising from the analysis of programs. This, combined with the heuristic of Term Abstraction (see Sect. 7.2), greatly helps to avoid divergence in practice as shown by the experiments in Sect. 8.

[14] http://research.microsoft.com/projects/contracts.

9.4 Shape analysis and separation logic

Heap manipulating programs are the target of *shape analysis* and *separation logic* approaches. Their goal is to infer a conservative characterization of the structure of the heap at each point of the program (see, e.g., [55,73]). Objects allocated on the heap are represented by a *heap graph*, where vertices are object allocated on the heap and edges are pointers accessing the objects [23]. Abstraction of these graphs can be done by using a three-value logic [75] or extending predicate abstraction to work with heap predicates [71].

While the goal of these techniques is to provide efficient and, at the same time, expressive analysis for pointers and unbounded data structures, our goal is to discover invariants for unbounded array elements. This is the target, for example, of the tool PREDATOR [37,38]. While PREDATOR was successfully used to prove memory safety of programs operating on unbounded linked lists [12], it is not yet able to prove that the array returned by a sorting algorithm is sorted. Additionally, the abstraction algorithms implemented in PREDATOR cannot handle arrays of unbounded size. However, as pointed out in [35], the two techniques are orthogonal and their integration is likely to benefit both of them.

9.5 Template-based approaches

Template based approaches (e.g., [14,77] to cite a few) may infer properties which are more expressive than the properties inferred by SAFARI, but are limited to those matching a given pattern. On the contrary our solution does not require in general user intervention in specifying templates for invariant: the only interaction of the user with the tool is by suggesting an appropriate term abstraction list whenever the tool seems to diverge. Recently, [64] presents a constraint-based invariant generation technique suited for the synthesis of quantified array invariants. This approach is SMT-based and uses non-linear constraints. It can synthesize invariants containing just one quantified variable and does not apply to nested loops. Our approach, instead, is not limited to invariants containing one quantified variables and can be applied to programs with nested loops, as witnessed by the experiments in Sect. 8.

10 Conclusion

We have described a new abstraction-based framework for the verification of programs handling arrays of unknown length. Our framework follows the "Lazy Abstraction with Interpolant" approach, where refinement is performed by computing interpolants from unsatisfiable formulas encoding spurious counterexamples.

Our technique is based on a backward reachability procedure for array-based transition systems [46] interleaved with a CEGAR procedure. Distinguishing features of our technique are the generation of quantified predicates, obtained via a preprocessing of the transition relation, followed by a refinement phase using quantifier-free interpolants. We have also identified a fragment of the theory of arrays enjoying quantifier-free interpolation, and studied hypothesis for the termination of the backward (CEGAR-based) reachability analysis.

The paper has presented implementation details and heuristics necessary for a successful experimentation. In particular, the heuristic of Term Abstraction addresses the problem of tuning interpolation by pre-processing input formulas. Since Term Abstraction does not interfere with the internals of interpolation algorithms, it can be potentially adopted in any verification tool handling problems for which there is a risk of divergence. In this respect,

we observe that Term Abstraction has been generalized and successfully applied to the verification of integer programs in [74].

Acknowledgments The authors would like to thank the anonymous reviewers for their comments and criticisms that helped to improve the quality of the paper. 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). The third author would like to acknowledge the support of the PRIN 2010-2011 project "Logical Methods for Information Management" funded by the Italian Ministry of Education, University and Research (MIUR).

References

1. Abdulla PA, Jonsson B (1996) Verifying programs with unreliable channels. Inf Comput 127(2):91–101
2. Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers: principles, techniques, and tools, 2nd edn. Pearson-Addison Wesley.
3. Albarghouthi A, Gurfinkel A, Chechik M (2012) Craig interpretation. In: Miné A, Schmidt D (eds) SAS. Springer, Lecture Notes in Computer Science, pp 300–316
4. Alberti F, Bruttomesso R, Ghilardi S, Ranise S, Sharygina N (2012) Lazy abstraction with interpolants for arrays. In: Bjørner N, Voronkov A (eds) LPAR, Lecture Notes in Computer Science, vol 7180, pp 46–61. Springer.
5. Alberti F, Bruttomesso R, Ghilardi S, Ranise S, Sharygina N (2012) SAFARI: SMT-based abstraction for arrays with interpolants. In: Madhusudan P, Seshia SA (eds) CAV., Lecture Notes in Computer Science, vol 7358, Springer, Berlin, pp 679–685
6. Alberti F, Ghilardi S, Pagani E, Ranise S, Rossi GP (2010). Automated support for the design and validation of fault tolerant parameterized systems: a case study. ECEASST, p 35.
7. Alberti F, Ghilardi S, Pagani E, Ranise S, Rossi GP (2012) Universal guards, relativization of quantifiers, and failure models in Model Checking Modulo theories. JSAT 8(1/2):29–61
8. Armando A, Benerecetti M, Carotenuto D, Mantovani J, Spica P (2007) The Eureka tool for software model checking. In Stirewalt REK, Egyed A, Fischer B (eds), ASE. ACM, pp 541–542.
9. Armando A, Benerecetti M, Mantovani J (2007). Abstraction refinement of linear programs with arrays. In: Grumberg O, Huth M (eds) TACAS, Lecture Notes in Computer Science, vol 4424. Springer, pp 373–388.
10. Franz Baader, Silvio Ghilardi (2007) Connecting many-sorted theories. J Symb Logic 72:535–583
11. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: Launchbury and Mitchell (eds) Conference record of POPL 2002: The 29th SIGPLAN-SIGACT symposium on principles of programming languages, Portland, OR, USA, January 16–18, 2002. ACM, pp 1–3.
12. Beyer D (2013) Second competition on Software Verification–(Summary of SV-COMP 2013). In Piterman N, Smolka SA (eds) Proceedings of the 19th international conference on tools and algorithms for the construction and analysis of systems, TACAS 2013, held as part of the European joint conferences on theory and practice of software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Lecture Notes in Computer Science, vol 7795. Springer, pp 594–609
13. Beyer D, Henzinger TA, Jhala R, Majumdar R (2007) The software model checker blast. STTT 9(5–6):505–525
14. Beyer D, Henzinger TA, Jhala R, Majumdar R, Rybalchenko A (2007) Invariant synthesis for combined theories. In Cook B, Podelski A (eds) VMCAI, Lecture Notes in Computer Science, vol 4349. Springer, pp 378–394.
15. Beyer D, Erkan Keremoglu M (2011) CPAchecker: a tool for configurable software verification. In: Gopalakrishnan G, Qadeer S (eds) Proceedings of the 23rd international conference on computer aided verification, CAV 2011, Snowbird, UT, USA, July 14–20, 2011. Lecture Notes in Computer Science, vol 6806. Springer pp 184–190.
16. Biere A, Cimatti AA, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Cleaveland R (ed) TACAS, Lecture Notes in Computer Science, vol 1579. Springer, pp 193–207.
17. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2002) Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen TÆ, Schmidt DA, Sudborough IH (eds) The essence of computation, Lecture Notes in Computer Science, vol 2566. Springer, pp 85–108.

18. Brillout A, Kroening, D, Rümmer P, Wahl T (2010) An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: Giesl H (ed) Proceedings of the 5th international joint conference on automated reasoning, IJCAR 2010, Edinburgh, UK, July 16–19, 2010. Lecture Notes in Computer Science, vol 6173. Springer, pp 384–399.
19. Bruttomesso R, Ghilardi S, Ranise S (2012) From strong amalgamability to modularity of quantifier-free interpolation. In: IJCAR, Lecture Notes in Computer Science. Springer, pp 118–133.
20. Bruttomesso R, Ghilardi S, Ranise S (2012) Quantifier-free interpolation of a theory of arrays. Logical Methods in Computer Science 8(2)
21. Bruttomesso R, Pek E, Sharygina N, Tsitovich A (2010) The OpenSMT solver. In: Esparza J, Majumdar R (eds) TACAS, Lecture Notes in Computer Science, vol 6015. Springer, pp 150–153.
22. Carioni A, Ghilardi S, Ranise S (2011) Automated termination in model checking Modulo theories. In: Delzanno G, Potapov I (eds) RP, Lecture Notes in Computer Science, vol 6945. Springer, pp 110–124.
23. Chase DR, Wegman MN, Zadeck FK (1990) Analysis of pointers and structures. In: Fischer BN (ed) PLDI. ACM, pp 296–310.
24. Cimatti A, Griggio A, Schaafsma BJ, Sebastiani R (2013) The MathSAT5 SMT solver. In: Piterman N, Smolka SA (eds) Proceedings of the 19th international conference on tools and algorithms for the construction and analysis of systems, TACAS 2013, held as part of the European joint conferences on theory and practice of software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Lecture Notes in Computer Science, vol 7795. Springer, pp 93–107.
25. Robert Clarisó, Jordi Cortadella (2007) The octahedron abstract domain. Sci Comput Program 64(1):115–139
26. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Allen Emerson E, Prasad Sistla A (eds) CAV, Lecture Notes in Computer Science, vol 1855. Springer, pp 154–169.
27. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In Jensen K, Podelski A (eds) TACAS, Lecture Notes in Computer Science, vol 2988. Springer, pp 168–176.
28. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham RM, Harrison MA, Sethi R (eds) POPL. ACM, pp 238–252
29. Cousot P, Cousot R, Logozzo F (2011) A parametric segmentation functor for fully automatic and scalable array content analysis. In Ball T, Sagiv M (eds) POPL. ACM, pp 105–118.
30. Cousot P, Halbwachs N (1978) Automatic discovery of linear restraints among variables of a program. In: Aho Alfred V, Zilles Stephen N, Szymanski Thomas G (eds) POPL. ACM Press, pp 84–96.
31. Craig W (1957) Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J Symb Log 22(3):269–285
32. Mendonça de Moura L, Bjørner N (2007) Efficient e-matching for SMT solvers. In Pfenning F (ed) CADE, Lecture Notes in Computer Science, vol 4603. Springer, pp 183–198.
33. Mendonça de Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J (eds) Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems, TACAS 2008, held as part of the joint European conferences on theory and practice of software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, Lecture Notes in Computer Science, vol 4963. Springer, pp 337–340.
34. Delzanno G, Esparza J, Podelski A (1999) Constraint-based analysis of broadcast protocols. Proceedings of CSL, LNCS 1683:50–66
35. Dillig I, Dillig T, Alex Aiken T (2010) Fluid updates: beyond strong vs. weak updates. In Gordon AD (ed), ESOP, Lecture Notes in Computer Science, vol 6012. Springer, pp 246–266.
36. Dimitrova R, Podelski A (2008) Is lazy abstraction a decision procedure for broadcast protocols? In: Logozzo F, Peled D, Zuck LD (eds) VMCAI, Lecture Notes in Computer Science, vol. 4905. Springer, pp 98–111.
37. Dudka K, Peringer P, Vojnar T (2011) Predator: a practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan G, Qadeer S (eds) Proceedings of the 23rd international conference on computer aided verification, CAV 2011, Snowbird, UT, USA, July 14–20, 2011. Lecture Notes in Computer Science, vol 6806. Springer, pp 372–378.
38. Dudka K, Peringer P, Vojnar T (2013) Byte-precise verification of low-level list manipulation. In: Logozzo F, Fähndrich M (eds) SAS, Lecture Notes in Computer Science, vol 7935. Springer, pp 215–237.
39. Enderton HB (2001) A Mathematical introduction to logic. Elsevier Science.
40. Fähndrich M, Logozzo F (2010) Static contract checking with abstract interpretation. In Beckert B, Marché C (eds) FoVeOOS, Lecture Notes in Computer Science, vol 6528. Springer, pp 10–30.

41. Flanagan C, Qadeer S (2002) Predicate abstraction for software verification. In: Launchbury J, Mitchell JC (eds) Conference record of POPL 2002: the 29th SIGPLAN-SIGACT symposium on principles of programming languages, Portland, OR, USA, January 16–18, 2002. ACM, pp 191–202.
42. Furia C.A., Meyer B. (2010). Inferring loop invariants using postconditions. In A. Blass, N. Dershowitz, and W. Reisig (eds), Fields of Logic and Computation, volume 6300 of Lecture Notes in Computer Science, pages 277–300. Springer.
43. Ge Y, Barrett CW, Tinelli C (2009) Solving quantified verification conditions using Satisfiability Modulo Theories. Ann. Math. Artif. Intell. 55(1–2):101–122
44. Ge Y, Mendonça de Moura L (2009) Complete instantiation for quantified formulas in Satisfiabiliby Modulo Theories. In Bouajjani A, Maler O (eds) CAV, Lecture Notes in Computer Science, vol 5643. Springer, pp 306–320.
45. Ghilardi S, Ranise S (2009) Model checking Modulo theory at work: the integration of Yices in MCMT. In: AFM.
46. Ghilardi S, Ranise S (2010) Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Logical Methods in Computer Science 6(4)
47. Ghilardi S, Ranise S (2010) Mcmt: a model checker modulo theories. In Giesl J, Hähnle R (eds) Proceedings of the 5th international joint conference on automated reasoning, IJCAR 2010, Edinburgh, UK, July 16–19, 2010. Lecture Notes in Computer Science, vol 6173. Springer, pp 22–29.
48. Ghilardi S, Ranise S, Valsecchi T (2009) Light-weight SMT-based model checking. Electron Notes Theor Comput Sci 250(2):85–102
49. Gopan D, Reps TW, Sagiv S (2005) A framework for numeric analysis of array operations. In: Palsberg J, Abadi M (eds) POPL. ACM, pp 338–350.
50. Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In Grumberg O (ed) CAV, Lecture Notes in Computer Science, vol 1254. Springer, pp 72–83.
51. Gulwani S, Tiwari A (2006) Combining abstract interpreters. In: Schwartzbach MI, Ball T (eds) PLDI. ACM, pp 376–386.
52. Halbwachs N, Péron M (2008) Discovering properties about arrays in simple programs. In Gupta R, Amarasinghe SP (eds) PLDI. ACM, pp 339–348.
53. Henzinger TA, Jhala R, Majumdar R, McMillan KL (2004) Abstractions from proofs. In: Jones ND, Leroy X (eds) POPL. ACM, pp 232–244.
54. Henzinger TA, Jhala R, Majumdar R, Sutre G (2002) Lazy abstraction. In: Launchbury J, Mitchell JC (eds) Conference record of POPL 2002: the 29th SIGPLAN-SIGACT symposium on principles of programming languages, Portland, OR, USA, January 16–18, 2002. ACM, pp 58–70.
55. Hind M (2001) Pointer analysis: haven't we solved this problem yet? In: Field J, Snelting G (eds) PASTE. ACM, pp 54–61.
56. Hoder K, Kovács L, Voronkov A (2010) Interpolation and symbol elimination in Vampire. In: Giesl H (ed) Proceedings of the 5th international joint conference on automated reasoning, IJCAR 2010, Edinburgh, UK, July 16–19, 2010. Lecture Notes in Computer Science, vol 6173. Springer, pp 188–195.
57. Hodges W (1993) Model theory, volume 42 of encyclopedia of mathematics and its applications. Cambridge University Press, Cambridge.
58. Jhala R, McMillan KL (2006) A practical and complete approach to predicate refinement. In: Hermanns H, Palsberg J (eds) TACAS, Lecture Notes in Computer Science, vol 3920. Springer, pp 459–473.
59. Jhala R, McMillan KL (2007) Array abstractions from proofs. In Damm W, Hermanns H (eds) CAV, Lecture Notes in Computer Science, vol 4590. Springer, pp 193–206.
60. Kapur D, Majumdar R, Zarba CG (2006) Interpolation for data structures. In: Young M, Devanbu PT (eds) SIGSOFT FSE. ACM, pp 105–116.
61. Kovács L, Voronkov A (2009) Finding loop invariants for programs over arrays using a theorem prover. In Chechik M, Wirsing M (eds) FASE, Lecture Notes in Computer Science, vol 5503. Springer, pp 470–485.
62. Lahiri SK, Bryant RE (2004) Constructing quantified invariants via predicate abstraction. In Steffen B, Levi G (eds) VMCAI, Lecture Notes in Computer Science, vol 2937. Springer, pp 267–281.
63. Lahiri SK, Bryant RE (2004) Indexed predicate discovery for unbounded system verification. In Alur R, Peled D (eds) CAV, Lecture Notes in Computer Science, vol. 3114. Springer, pp 135–147.
64. Larraz D, Rodríguez-Carbonell E, Rubio A (2013) SMT-based array invariant generation. In: Giacobazzi R, Berdine J, Mastroeni I (eds) VMCAI, Lecture Notes in Computer Science, vol 7737. Springer, pp 169–188.
65. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems–specification. Springer, Berlin
66. McCarthy J (1962) Towards a mathematical science of computation. In: IFIP Congress, pp 21–28.

67. McMillan KL (2006) Lazy abstraction with interpolants. In: Ball T, Jones RB (eds) Proceedings of the 18th international conference on computer aided verification, CAV 2006, Seattle, WA, USA, August 17–20, 2006, Lecture Notes in Computer Science, vol 4144. Springer, pp 123–136.
68. McMillan KL (2008) Quantified invariant generation using an interpolating saturation prover. In Ramakrishnan CR, Rehof J (eds) Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems, TACAS 2008, held as part of the joint European conferences on theory and practice of software, ETAPS 2008, Budapest, Hungary, March–April 6, 2008, Lecture Notes in omputer Science, vol 4963. Springer, pp 413–427.
69. Antoine Miné (2006) The octagon abstract domain. Higher-Order Symb Comput 19(1):31–100
70. Nelson G, Oppen DC (1979) Simplification by Cooperating Decision Procedures. ACM Trans Program Lang Syst 1(2):245–257
71. Podelski A, Wies T (2005) Boolean heaps. In Hankin C, Siveroni I (eds) SAS, Lecture Notes in Computer Science, vol 3672. Springer, pp 268–283.
72. Ranise S, Tinelli C (2006). The satisfiability Modulo theories library (SMT-LIB). http://www.smt-lib.orgwww.SMT-LIB.org
73. Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: LICS. IEEE Computer Society, pp 55–74.
74. Rümmer P, Subotić P (2013) Exploring interpolants. In: Jobstmann B, Ray S (eds) FMCAD. FMCAD Inc., pp 69–76.
75. Sagiv S, Reps TW, Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In: Appel AW, Aiken A (eds) POPL. ACM, pp 105–118 (1999).
76. Seghir MN, Podelski A, Wies T (2009) Abstraction refinement for quantified array assertions. In: Palsberg J, Su Z (eds) SAS, Lecture Notes in Computer Science, vol 5673. Springer, pp 3–18.
77. Srivastava S, Gulwani S (2009) Program verification using templates over predicate abstraction. In: Hind M, Diwan A (eds) PLDI. ACM, pp 223–234.
78. Wirth N (1978) Algorithms + data structures = programs. Prentice-Hall Series in Automatic Computation, Pearson Education