Acceleration-based safety decision procedure for programs with arrays

Francesco Alberti[1], Silvio Ghilardi[2], Natasha Sharygina[1]

[1] University of Lugano, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy

Abstract

Reachability analysis of programs with arrays is a well-known challenging problem and many existing approaches are bound to incomplete solutions. In this paper we identify a class of programs with arrays for which safety is fully decidable. The completeness result we provide is built up from acceleration techniques for arrays and the decision procedure for the Array Property Fragment class.

1 Introduction

Reachability analysis plays a crucial role in program verification. It relies on algorithms for computing a fixed point of the transition relation representing the program body and checking if any execution leads to the violation of a given property. The main weakness of reachability analysis is that the fixed point computation, which is required for determining the unreachability of erroneous configurations, is intractable in general. This implies that most procedures dealing with the verification of safety properties of programs can be, at most, sound, but in general still incomplete or non-terminating. Completeness of the required satisfiability tests and termination of the whole procedure can be achieved only by constraining the class of systems under analysis. This paper addresses this challenge in the context of programs handling arrays, by identifying a class of them for which reachability analysis is complete and terminating.

More precisely, in this paper we consider programs handling arrays of unknown length. Recently it has been shown (see [4]) that accelerations of a wide class of ground transitions over arrays and counters can be defined in Presburger arithmetic enriched with free function symbols. Definability of accelerations of such relations does not come for free, though. The price to pay is admitting nested quantifiers in the accelerated transitions and, as a consequence, in the formulæ used to symbolically describe sets of states reachable by the program. The presence of such quantifiers has the drawback of keeping completeness of satisfiability tests still unachievable in the general case.

In this paper we pursue the line of research of [4] and focus on identifying a clear set of transition relations for which acceleration, despite requiring alternation of quantifiers to be defined, can be exploited to achieve the completeness of the reachability analysis, by using well-known decidability results on quantified fragments for theories of arrays (e.g., [10]).

The contribution of the paper is, therefore, the identification of a class of programs with arrays of unknown length for which the unbounded reachability analysis problem is decidable. We call this class basic-flat-programs. The method we use for showing these decidability results is similar to a classical method adopted in the model-checking literature for programs manipulating integer variables (see for instance [9, 12, 14]): we first assume flatness conditions on the control flow graph of the program and then we assume that transitions labeling cycles are "accelerable". Such transitions consists on assignments admitting definable acceleration in the theory of Presburger arithmetic enriched with free function symbols. Completeness and termination results on the reachability analysis of these programs are achieved by applying a preprocessing test in charge of reducing the original program to an "accelerated" program, where all the relations encoding loops are accelerated. Given an accelerated basic-flat-program, the reachability of an error configuration can be checked by testing the satisfiability of finitely many proof-obligations, which are formulæ admitting a decision procedure.

Related work This paper identifies a class of programs with arrays for which safety can be *fully decided*. This is in sharp contrast with other solutions for analysis of programs with arrays, which, while focusing (at most) on soundness of their results, sacrifice completeness and/or termination of their procedures for generality (e.g., [2, 6, 13, 15, 19]).

The closest solution to our contribution is presented in [4], although in [4] the focus is on the combination of acceleration and abstraction. Even in this case, the provided solution is sound but it is not guaranteed to terminate. The completeness result we provide in this paper goes over the borders of the one studied in [1,11] because this work considers arrays indexed by full Presburger arithmetic. The lack of a full decidability result distinguishes as well our work from other automata-based solutions [7] for the verification of programs with arrays.

Broadening the borders of related work, we acknowledge the work of Bozga et al. [8] on the definability of the accelerations of relations over integers in Presburger arithmetic. Recently, acceleration has been integrated in refinement procedures for the verification of integer programs to generate inductive loop invariants from interpolants [17]. Kroening et al. describe in [18] how to generate sound under-approximations of loops in C programs with bit-vector semantics with the goal of gaining a speed-up in finding deep counterexamples. Such technique does not involve fixed-point detection, therefore cannot be complete. Acceleration has been successfully adopted also in other application domains: besides reachability analysis of integer variables programs [9,12,14], acceleration plays e.g. an important role in the system UPPAAL for reducing the fragmentation problem [5].

Plan of the paper In Section 2 we fix some notation and in Section 3 we introduce a formal model for array programs. In Section 4 we discuss acceleration of a special class of transitions, called basic-assignments; we finally use definability of accelerated basic-assignments in the array property fragment of [10] to obtain our main decidability result. We conclude in Section 5.

2 Preliminaries

We work in Presburger arithmetic enriched with free function symbols and with definable function symbols (see below); when we speak about validity or satisfiability of a formula, we mean *satisfiability and validity in all structures having the standard structure of natural numbers as reduct*. Thus, satisfiability is decidable if we limit to quantifier-free formulæ (by adapting Nelson-Oppen combination results [20, 22]), but may become undecidable otherwise (because of the presence of free function symbols).

We use $x, y, z, \dots$ or $i, j, k, \dots$ for variables; $t, u, \dots$ for terms, $c, d, \dots$ for free constants, $a, b, \dots$ for free function symbols, $\phi, \psi, \dots$ for *quantifier-free* formulæ. Bold letters are used for tuples and $|\cdot|$ indicates tuples length; hence for instance $\mathbf{u}$ indicates a tuple of terms $u_1, \dots, u_m$, where $m = |\mathbf{u}|$. These tuples may contain repetitions. For variables, we use underlined letters $\underline{x}, \underline{y}, \dots, \underline{i}, \underline{j}, \dots$ to indicate tuples without repetitions. Vector notation can also be used for equalities: if $\mathbf{u} = u_1, \dots, u_n$ and $\mathbf{v} = v_1, \dots, v_n$, we may use $\mathbf{u} = \mathbf{v}$ to mean the formula $\bigwedge_{i=1}^{n} u_i = v_i$.

If we write $t(x_1, \dots, x_n), \mathbf{u}(x_1, \dots, x_n), \phi(x_1, \dots, x_n)$ (or $t(\underline{x}), \mathbf{u}(\underline{x}), \phi(\underline{x}), \dots$, in case $\underline{x} = x_1, \dots, x_n$), we mean that the term t, the tuple of terms $\mathbf{u}$, the quantifier-free formula ϕ contain variables only from the tuple $x_1, \dots, x_n$. Similarly, we may use $t(\mathbf{a}, \mathbf{c}, \underline{x}), \phi(\mathbf{a}, \mathbf{c}, \underline{x}), \dots$ to mean *both* that the term t or the quantifier-free formula ϕ have free variables included in $\underline{x}$ *and* that the free function symbols and constants occurring in them are among $\mathbf{a}, \mathbf{c}$. Notations like $t(\mathbf{u}/\underline{x}), \phi(\mathbf{u}/\underline{x}), \dots$ or $t(u_1/x_1, \dots, u_n/x_n), \phi(u_1/x_1, \dots, u_n/x_n), \dots$ - or occasionally just $t(\mathbf{u}), \phi(\mathbf{u}), \dots$ if confusion does not arise - are used for simultaneous substitutions within terms and formulæ. If a is a unary free function symbol and t a term, we may write $a[t]$ instead of $a(t)$.

By a *definable function symbol*, we mean the following. Take a quantifier-free formula $\phi(\underline{j}, y)$ such that $\forall \underline{j} \exists! y \phi(\underline{j}, y)$ is valid ($\exists! y$ stands for 'there is a unique y such that ...'). Then a definable function symbol F (defined by ϕ) is a fresh function symbol, matching the length of $\underline{j}$ as arity, which is constrained to be interpreted in such a way that the formula $\forall y. F(\underline{j}) = y \leftrightarrow \phi(\underline{j}, y)$ is true. The addition of definable function symbols does not affect decidability of quantifier-free formulæ and can be used for various purposes, like directly expressing case-defined functions, array updates, etc. For instance, if a is a unary free function symbol, the term $wr(a, i, x)$ (expressing the update of the array a at position i by over-writing x) is a definable function; formally, we have $\underline{j} := i, x, j$ and $\phi(\underline{j}, y)$ is given by $(j = i \wedge y = x) \vee (j \neq i \wedge y = a(j))$. This formula $\phi(\underline{j}, y)$ (and similar ones) is usually written as

$$y \ = \ (\texttt{if } j = i \texttt{ then } x \texttt{ else } a(j))$$

to improve readability. As further examples of definable function symbols one may consider (in Presburger arithmetic) integer quotient and remainder modulo a fixed number n.

3 Array program representation and their safety

To model a program we need to fix program variables and transition relations. Program variables are a tuple $\mathbf{v} := \mathbf{a}, \mathbf{c}$ (to be fixed from now on), where

- the tuple $\mathbf{a} = a_1, \dots, a_s$ contains free unary function symbols, i.e., the arrays manipulated by the program;
- the tuple $\mathbf{c} = c_1, \dots, c_t$ contains free constants, i.e., the integer data manipulated by the program;

Transition relations are formulæ of the kind $\tau(\mathbf{v}, \mathbf{v}')$ (here $\mathbf{v}'$ are renamed copies of the tuple $\mathbf{v}$ representing the next-state variables).

Sentences denoting sets of states reachable by a program can be:

- *ground* sentences, i.e., sentences of the kind $\phi(\mathbf{a}, \mathbf{c})$;
- Σ^0_1-*sentences*, i.e., sentences of the form $\exists \underline{i}.\ \phi(\underline{i}, \mathbf{a}, \mathbf{c})$;
- Σ^0_2-*sentences*, i.e., sentences of the form $\exists \underline{i} \forall \underline{j}.\ \phi(\underline{i}, \underline{j}, \mathbf{a}, \mathbf{c})$.

We remark that in our context satisfiability can be fully decided only for ground sentences and Σ^0_1-sentences (by Skolemization, as a consequence of the general combination results [20, 22]), while only subclasses of Σ^0_2-sentences enjoy a decision procedure. Of particular interest for the present paper is the *array property fragment* which is shown to be decidable in [10]. We reformulate below the related definition because it is used later; we say that a formula or a term is *arithmetical* iff it is built up from variables using only the symbols $=, <, +, -, 0, 1$ (in particular free function symbols do not occur in it).

Definition 3.1 *A formula is in the* array property fragment *[10] iff it is equivalent to a disjunction of formulae of the kind* $\exists \underline{i}\ (\psi_1(\underline{i}, \mathbf{a}) \wedge \dots \wedge \psi_n(\underline{i}, \mathbf{a}))$, *where each* ψ_i *is either a literal* *or a guard. In turn, a guard is a formula of the kind* $\forall \underline{j}\ (G \to \theta)$, *where (i)* G *is a conjunction of atoms of the kind* $j_1 \le j_2, j_1 \le t(\underline{i}), t(\underline{i}) \le j_1$ *for* $j_1, j_2 \in \underline{j}$ *and* $t(\underline{i})$ *arithmetical; (ii)* θ *is obtained from a quantifier-free arithmetical formula* $\alpha(\underline{i}, \underline{x})$ *by replacing the variables* $\underline{x}$ *by terms of the kind* $a[t(\underline{i})], a[j]$ *for* $j \in \underline{j}, t(\underline{i})$ *arithmetical and* $a \in \mathbf{a}$.

Transition formulæ can also be classified into three groups:

- *ground assignments*, i.e., transitions of the form

$$\phi_L(\mathbf{c}, \mathbf{a}) \ \wedge \ \mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, j) \ \wedge \ \mathbf{c}' = H(\mathbf{c}, \mathbf{a}) \tag{1}$$

- Σ^0_1-*assignments*, i.e., transitions of the form

$$\exists \underline{k}\ \ (\phi_L(\mathbf{c}, \mathbf{a}, \underline{k}) \ \wedge \ \mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, \underline{k}, j) \ \wedge \ \mathbf{c}' = H(\mathbf{c}, \mathbf{a}, \underline{k})) \tag{2}$$

- Σ^0_2-*assignments*, i.e., transitions of the form

$$\exists \underline{k} \begin{pmatrix} \phi_L(\mathbf{c}, \mathbf{a}, \underline{k}) \ \wedge \ \forall \underline{j}\ \psi_U(\mathbf{c}, \mathbf{a}, \underline{k}, \underline{j}) \ \wedge \\ \mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, \underline{k}, j) \wedge \ \mathbf{c}' = H(\mathbf{c}, \mathbf{a}, \underline{k}) \end{pmatrix} \tag{3}$$

where $G = G_1, \ldots, G_s$, $H = H_1, \ldots, H_t$ are tuples of definable functions (vectors of equations like $\mathbf{a}' = \lambda j.\ G(\mathbf{c}, \mathbf{a}, \underline{k}, j)$ can be replaced by the corresponding first order sentences $\forall j.\ \bigwedge_{h=1}^{s} a'_h(j) = G_h(\mathbf{c}, \mathbf{a}, \underline{k}, j)$).

3.1 Modeling programs

In this work, programs will be represented by their control-flow automaton.

Definition 3.2 *Giving a set of variables* $\mathbf{v}$ *like above, a* program *is a triple* $\mathcal{P} = (L, \Lambda, E)$, *where (i)* $L = \{l_1, \ldots, l_n\}$ *is a set of* program locations *among which we distinguish an initial location* l_{init} *and an error location* l_{error}*; (ii)* Λ *is a finite set of transition formulæ* $\{\tau_1(\mathbf{v}, \mathbf{v}'), \ldots, \tau_r(\mathbf{v}, \mathbf{v}')\}$ *and (iii)* $E \subseteq L \times \Lambda \times L$ *is a set of* actions.

We indicate by $src, \mathcal{L}, trg$ the three projection functions on E; that is, for $e = (l_i, \tau_j, l_k) \in E$, we have $src(e) = l_i$ (this is called the 'source' location of e), $\mathcal{L}(e) = \tau_j$ (this is called the 'label' of e) and $trg(e) = l_k$ (this is called the 'target' location of e).

A *program path* (in short, *path*) of $\mathcal{P} = (L, \Lambda, E)$ is a sequence $\rho \in E^n$, i.e., $\rho = e_1, e_2, \ldots, e_n$, such that for every $i < n$, we have that $trg(e_i) = src(e_{i+1})$. We denote with $|\rho|$ the length of the path.

Of particular interest for our work is the class of flat^0-programs, i.e., programs admitting only self-loops for which each location belongs to at most one loop.

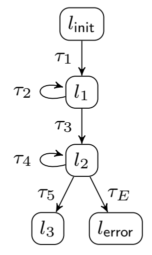
Definition 3.3 *A program* $\mathcal{P} = (L, \Lambda, E)$ *is a* flat^0-program *if for every path* $\rho = e_1, \ldots, e_n$ *of* $\mathcal{P}$ *it holds that for every* $j < k \le n$, *if* $src(e_j) = trg(e_k)$ *then* $e_j = e_{j+1} = \cdots = e_k$.

Example 3.1 *Consider the procedure* init *in Figure 1. For this procedure,* $\mathbf{a} = a$, $\mathbf{c} = i, v$. N *is a free constant.* Λ *is the set of formulæ (we omit identical updates):*

$$\begin{aligned}
&\tau_1 := i' = 0 \\
&\tau_2 := i < N \wedge a' = \lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ v\ \mathsf{else}\ a[j] \wedge i' = i + 1 \\
&\tau_3 := i \ge N \wedge i' = 0 \\
&\tau_4 := i < N \wedge i' = i + 1 \\
&\tau_5 := i \ge N \\
&\tau_E := i < N \wedge a[i] \ne v
\end{aligned}$$

```
    procedure init ( $a[N]$ , $v$ ) :
$l_1$    for ($i = 0$; $i < N$; $i = i + 1$)  $a[i] = v$;
$l_2$    for ($i = 0$; $i < N$; $i = i + 1$)  assert($a[i] = v$);
```

(a)



(b)

Figure 1: The init procedure (a) and its control-flow graph (b).

The procedure init *can be formalized as the control-flow graph depicted in Figure 1(b), where* $L = \{l_{\mathsf{init}}, l_1, l_2, l_3, l_{\mathsf{error}}\}$.

3.2 The reachability problem

Definition 3.4 *An* error path *in a program* $\mathcal{P} = (L, \Lambda, E)$ *is a path* $\rho = e_1, e_2, \ldots, e_n$ *with* $src(e_1) = l_{\mathsf{init}}$ *and* $trg(e_n) = l_{\mathsf{error}}$. *A path* ρ *is a* feasible path *if* $\bigwedge_{j=1}^{|\rho|} \mathcal{L}(e_j)^{(j)}$ *is satisfiable, where* $\mathcal{L}(e_j)^{(j)}$ *represents* $\tau_{i_j}(\mathbf{v}^{(j-1)}, \mathbf{v}^{(j)})$, *with* $\mathcal{L}(e_j) = \tau_{i_j}$.

The *(unbounded) reachability problem* for a program $\mathcal{P}$ is to detect if $\mathcal{P}$ admits a feasible error path. Proving the safety of $\mathcal{P}$, therefore, means solving the reachability problem for $\mathcal{P}$. This problem, given well known limiting results, is not decidable for any program $\mathcal{P}$. The consequence is that, in general, any reachability analysis is sound, but not complete, and its incompleteness manifests itself in (possible) divergence of the verification algorithm.

The problem, in our context, is generated by loops, which give rise to possibly infinite error paths to check. Focusing our attention only on flat^0-programs, the application of an acceleration procedure may help in limiting divergence since it would substitute each loop with its accelerated form. Leveraging the accelerated transitions, one can compute *in one shot* the set of reachable states after n unwindings of the corresponding loop, for any n. This prevents the divergence of the reachability analysis caused by considering always deeper unwindings.

4 Deciding the safety of basic-flat-programs

We recall the definition of an accelerated transition:

Definition 4.1 *The* composition $\tau_1 \circ \tau_2$ *of two transitions* $\tau_1(\mathbf{v}, \mathbf{v}')$ *and* $\tau_2(\mathbf{v}, \mathbf{v}')$ *is expressed by the formula* $\exists \mathbf{v}_1(\tau_1(\mathbf{v}, \mathbf{v}_1) \wedge \tau_2(\mathbf{v}_1, \mathbf{v}'))$ *The n-th composition of a transition* $\tau(\mathbf{v}, \mathbf{v}')$ *with itself is recursively defined by* $\tau^1 := \tau$ *and* $\tau^{n+1} := \tau \circ \tau^n$. *The* acceleration τ^+ *of* τ *is* $\bigvee_{n \geq 1} \tau^n$.

In general, acceleration requires a logic supporting infinite disjunctions, but it has been shown in [4] that infinite disjunctions (which anyhow are a formalism outside the realm of first-order logic) are not needed for expressing the acceleration of some ground assignments involving array variables: their accelerations can be expressed as Σ_2^0-assignments. However, the practical advantage of expressing in a first order language the transitive closure of relations via Σ_2^0-assignments is limited by the undecidability of satisfiability of Σ_2^0-formulæ. Thus, it will become essential to identify classes of ground assignments whose acceleration belongs to a decidable class of Σ_2^0-formulæ.

4.1 The class of basic-assignments

Next definition aims at identifying a class of transitions whose acceleration fits Definition 3.1.

Definition 4.2 *A* basic*-assignment is a ground assignment of the following kind (we split the scalar variables* $\mathbf{c}$ *as* $\mathbf{c} := d, \mathbf{d}$ *to highlight a 'counter'* d *among them):*

$$\phi_L^1(d,\mathbf{d}) \wedge \phi_L^2(\mathbf{d},\mathbf{a}[d]) \ \wedge\ d' = d+1 \ \wedge\ \mathbf{d}' = \mathbf{d} \ \wedge\ \mathbf{a}' = \lambda j.\mathsf{if}\ (j=d)\ \mathsf{then}\ \mathbf{t}(\mathbf{d},\mathbf{a}[d])\ \mathsf{else}\ \mathbf{a}[j] \quad (4)$$

where (i) the tuple of terms[1] $\mathbf{a}[d]$ *stands for* $a_1[d],\dots,a_s[d]$*; (ii)* $\mathbf{t}$ *is an* s*-tuple of terms; (iii)* $\phi_L^1(d,\mathbf{d}), \phi_L^2(\mathbf{d},\mathbf{a}[d])$ *and* $\mathbf{t}(\mathbf{d},\mathbf{a}[d])$ *are obtained from arithmetical* $\phi_L^1(x,\mathbf{x}), \phi_L^2(\mathbf{x},\mathbf{z})$ *and* $\mathbf{t}(\mathbf{x},\mathbf{z})$ *by replacing* $x,\mathbf{x},\mathbf{z}$ *with* $d,\mathbf{d},\mathbf{a}[d]$*, respectively.*

The next lemma (an easy special case of a result from [4]) gives the template for the acceleration of basic-assignments.

Lemma 4.1. *The acceleration of the* basic*-assignment* (4) *is equivalent to the formula* $\tau^+(\mathbf{v},\mathbf{v}')$ *given by*

$$\exists y > 0 \begin{pmatrix} \forall j\ (d \le j < d+y \to \phi_L^1(j,\mathbf{d}) \wedge \phi_L^2(\mathbf{d},\mathbf{a}[j]) \ \wedge\ d' = d+y \ \wedge\ \mathbf{d}' = \mathbf{d}\ \wedge \\ \mathbf{a}' = \lambda j.\ (\mathsf{if}\ (d \le j < d+y)\ \mathsf{then}\ \mathbf{t}(\mathbf{d},\mathbf{a}[j])\ \mathsf{else}\ \mathbf{a}[j]) \end{pmatrix} \quad (5)$$

Example 4.1 *Consider transition* τ_2 *from the formalization of our running example in Figure 1. The acceleration* τ_2^+ *of such formula is*

$$\exists y. \begin{pmatrix} y > 0 \wedge \forall z.(i \le z < i+y \to z < N) \ \wedge\ i' = i+y\ \wedge \\ a' = \lambda j.\,(\mathsf{if}\ \ (i \le j < i+y)\ \ \mathsf{then}\ v\ \mathsf{else}\ a[j]) \end{pmatrix}$$

The following lemma is crucial for our decidability result:

Lemma 4.2. *The formula* (5) *is equivalent to a formula in the array property fragment.*

Proof. We need to rewrite (5) a bit. Forgetting about the top existential quantifier, (5) can be rewritten as the conjunction of the following formulæ:

1. $y > 0 \ \wedge\ d' = d+y \ \wedge\ \mathbf{d}' = \mathbf{d}$;
2. $\forall j\ (d \le j \ \wedge\ j \le d+y-1 \to \phi_L^1(j,\mathbf{d}))$;
3. $\forall j\ (d \le j \ \wedge\ j \le d+y-1 \to \phi_L^2(\mathbf{d},\mathbf{a}[j]))$;
4. $\forall j\ (d \le j \ \wedge\ j \le d+y-1 \to \mathbf{a}'[j] = \mathbf{t}(\mathbf{d},\mathbf{a}[j]))$;
5. $\forall j\ (j \le d-1 \to \mathbf{a}'[j] = \mathbf{a}[j])$;
6. $\forall j\ (d+y \le j \to \mathbf{a}'[j] = \mathbf{a}[j])$

(notice that in 4-5-6 the equality $\mathbf{a}'[j] = \cdots$ is in fact a conjunction of s equations). The only conjunct that might give problems with respect to Definition 3.1 is the second one; however this is a formula in Presburger arithmetic, hence we can apply quantifier elimination to it and get an equivalent quantifier-free formula of the kind $\psi(d,\mathbf{d})$. This fits Definition 3.1 (but we have to introduce extra existentially quantified variables to get rid of the equivalence-modulo-n predicates that are introduced by quantifier elimination [21]). □

[1]Recall that in Section 3 we put $\mathbf{a} := a_1,\dots,a_s$.

4.2 The Decision Algorithm

We are now ready to identify a class of programs with arrays for which the unbounded reachability problem is decidable. We call these programs basic-flat-programs: basic-flat-*programs are* flat^0*-programs for which every non-loop edge is labeled with a ground or* Σ_1^0*-assignment and every loop edge is labeled with a* basic-*assignment.* In the following, let us modify the projection function $\mathcal{L}$ of a basic-flat-program $\mathcal{P} = (L, \Lambda, E)$ by denoting $\mathcal{L}^+(e) := \mathcal{L}(e)^+$ if $src(e) = trg(e)$ and $\mathcal{L}^+(e) := \mathcal{L}(e)$ otherwise, where $\mathcal{L}(e)^+$ denotes the acceleration of the transition labeling the edge e.

Theorem 4.1. *The reachability problem for* basic-flat-*programs is decidable.*

Proof. Let $\rho = e_1, \dots, e_n$ be an error path of a basic-flat-program $\mathcal{P}$; when testing its feasibility, according to Definition 3.3, we can limit ourselves to the case in which $e_1, \dots, e_n$ are all distinct, provided we replace the labels $\mathcal{L}(e_k)^{(k)}$ with $\mathcal{L}^+(e_k)^{(k)}$ in the formula $\bigwedge_{j=1}^{n} \mathcal{L}(e_j)^{(j)}$ from Definition 3.4. Thus $\mathcal{P}$ is unsafe iff, for some error path $e_1, \dots, e_n$ whose edges are all distinct, the formula

$$\mathcal{L}^+(e_1)^{(1)} \wedge \cdots \wedge \mathcal{L}^+(e_n)^{(n)} \tag{6}$$

is satisfiable. Since the paths to be checked are finitely many and since, by lemma 4.2, formulæ like (6) are equivalent to formulæ in the array property fragment (whose satisfiability is decidable [10]), the safety of $\mathcal{P}$ can be decided. $\square$

5 Conclusion and future work

We have proved that the unbounded reachability problem for a class of programs is decidable. One may wonder how significant is that class. In fact, the class is not that small: intuitively, it contains programs (without nested loops) implementing functions like searching, comparing, initializing, testing, etc. for arrays or strings. Our decision procedure relies on the decidability result for the array fragment considered in [10]; we presume that by our method one can rely on different decidable array fragments (e.g., [3,16]) and get decidability results for slightly different classes of programs.

Another question is related to the actual ability of available SMT-Solvers in discharging the quantified proof obligations (6) generated by our decision procedure. Preliminary experiments[2] show that the presence of nested quantifiers in such proof obligations constitute a non-trivial challenge for state-of-the-art SMT-Solvers.

References

[1] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012.

[2] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In *CAV*, volume 7358 of *LNCS*, pages 679–685. Springer, 2012.

[3] F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. Technical Report 2013/04, University of Lugano, October 2013. Available at `http://www.inf.usi.ch/research_publication.htm?id=77`.

[2]More information at `http://www.inf.usi.ch/phd/alberti/prj/booster`.

[4] F. Alberti, S. Ghilardi, and N. Sharygina. Definability of accelerated relations in a theory of arrays and its applications. In *FroCoS*, volume 8152 of *LNCS*, pages 23–39. Springer, 2013. Extended version available at `http://www.inf.usi.ch/research_publication.htm?id=71`.

[5] G. Behrmann, J. Bengtsson, A. David, K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL implementation secrets. In *FTRTFT*, volume 2469 of *LNCS*, pages 3–22. Springer, 2002.

[6] N. Bjørner, K.L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, volume 7935 of *LNCS*, pages 105–125. Springer, 2013.

[7] M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, volume 5643 of *LNCS*, pages 157–172. Springer, 2009.

[8] M. Bozga, R. Iosif, and F. Konecný. Fast acceleration of ultimately periodic relations. In *CAV*, volume 6174 of *LNCS*, pages 227–242. Springer, 2010.

[9] M. Bozga, R. Iosif, and Y. Lakhnech. Flat parametric counter automata. *Fundamenta Informaticae*, (91):275–303, 2009.

[10] A.R. Bradley, Z. Manna, and H.B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[11] A. Carioni, S. Ghilardi, and S. Ranise. Automated termination in model-checking modulo theories. *Int. J. Found. Comput. Sci.*, 24(2):211–232, 2013.

[12] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *CAV*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.

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

[14] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FSTTCS*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.

[15] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV*, volume 8044 of *LNCS*, pages 813–829. Springer, 2013.

[16] P. Habermehl, R. Iosif, and T. Vojnar. A logic of singly indexed arrays. In *LPAR*, volume 5330 of *LNCS*, pages 558–573. Springer, 2008.

[17] H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer. Accelerating interpolants. In *ATVA*, volume 7561 of *LNCS*, pages 187–202. Springer, 2012.

[18] D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *CAV*, volume 8044 of *LNCS*, pages 381–396, 2013.

[19] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *VMCAI*, volume 7737 of *LNCS*, pages 169–188. Springer, 2013.

[20] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transaction on Programming Languages and Systems*, 1(2):245–257, 1979.

[21] D.C. Oppen. A superexponential upper bound on the complexity of Presburger arithmetic. *J. Comput. System Sci.*, 16(3):323–332, 1978.

[22] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In *Proc. of FroCoS 1996*, pages 103–119. Kluwer, 1996.