SMT-Based Array Invariant Generation*

Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio

Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. This paper presents a constraint-based method for generating universally quantified loop invariants over array and scalar variables. Constraints are solved by means of an SMT solver, thus leveraging recent advances in SMT solving for the theory of non-linear arithmetic. The method has been implemented in a prototype program analyzer, and a wide sample of examples illustrating its power is shown.

Keywords: Program correctness, Invariant generation, SMT.

1 Introduction

Discovering loop invariants is an essential task for verifying the correctness of software. In particular, for programs manipulating arrays, usually one has to take into account invariant relationships among values stored in arrays and scalar variables. However, due to the unbounded nature of arrays, invariant generation for these programs is a challenging problem. In this paper we present a method for generating universally quantified loop invariants over array and scalar variables.

Namely, programs are assumed to consist of unnested loops and linear assignments, conditions and array accesses. Let $\overline{a} = (A_1, \ldots, A_m)$ be the array variables. Given an integer $k > 0$, our method generates invariants of the form:

$$\forall \alpha \ : \ 0 \leq \alpha \leq C(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0\,,$$

where $\mathcal{C}, \mathcal{E}_{ij}, \mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables $\overline{v}$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$ for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, k\}$. This family of properties is quite general and allows handling a wide variety of programs.

Our method builds upon the so-called *constraint-based invariant generation* approach for discovering linear invariants [1], i.e., invariants expressed as linear inequalities over scalar variables. By means of Farkas' Lemma, the problem of the existence of an inductive invariant for a loop is transformed into a satisfiability problem in propositional logic over non-linear arithmetic. Despite the potential of the approach, its application has been limited so far due to the lack of good solvers for the obtained non-linear constraints.

However, recently significant progress has been made in SMT modulo the theory of non-linear arithmetic. In particular, the Barcelogic SMT solver has shown to be very effective on finding solutions in quantifier-free non-linear integer arithmetic [2]. These advances motivated us to revisit the constraint-based approach for linear invariants and extend it to programs with arrays.

* Partially supported by Spanish MEC/MICINN under grant TIN 2010-68093-C02-01.

2 Preliminaries

2.1 Transition Systems

Henceforth we will model programs by means of *transition systems*. A transition system $\mathcal{P} = \langle \overline{u}, \mathcal{L}, \ell_0, \mathcal{T} \rangle$ consists of a tuple of *variables* $\overline{u}$, a set of *locations* $\mathcal{L}$, an *initial location* ℓ_0 and a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is a triple $\langle \ell, \ell', \rho_\tau \rangle$, where ℓ, $\ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and ρ_τ is the *transition relation*: a first-order Boolean formula over the program variables $\overline{u}$ and their primed versions $\overline{u}'$, which represent the values of the variables after the transition. In general, to every formula P (or expression E) over the program variables $\overline{u}$ we associate a formula P' (or expression E') which is the result of replacing every variable u_i in P (or E) by its corresponding primed version u_i'.

In this paper we will consider *scalar* variables, which take integer values, and *array* variables. We will denote scalar variables by $\overline{v}$ and array variables by $\overline{a}$. The *size* of an array $A \in \overline{a}$ is denoted by $|A|$ and the *domain* of its indices is $\{0 \ldots |A| - 1\}$ (i.e., indices start at 0, as in C-like languages). We assume that arrays can only be indexed by expressions built over scalar variables. Hence, by means of the read/write semantics of arrays, we can describe transition relations as array equalities (possibly guarded by conjunctions of equalities and disequalities between scalar expressions) and quantified information of the form $\forall \alpha : 0 \leq \alpha \leq |A| - 1 \wedge P(\alpha) : A'[\alpha] = A[\alpha]$, where P does not depend on array variables. For example, Fig. 1 shows a program together with its transition system. A *path* π between two locations is associated to a transition

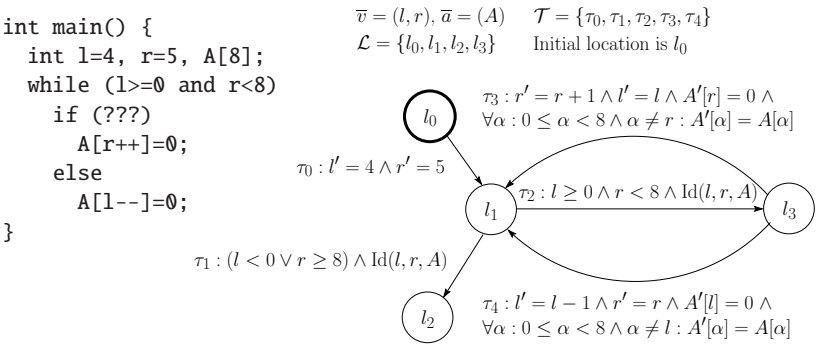

Fig. 1. Program and its transition system. Predicate $\mathrm{Id}(u_1, \ldots, u_k)$ is short for $u_1 = u_1' \wedge \cdots \wedge u_k = u_k'$, i.e., indicates those variables that remain identical after a transition.

relation ρ_π which is obtained by composition of the corresponding transitions relations. For instance, in the transition system in Fig. 1, the transition relations of the paths $\pi_0 = (l_0, \tau_0, l_1)$, $\pi_1 = (l_1, \tau_2, l_3, \tau_3, l_1)$ and $\pi_2 = (l_1, \tau_2, l_3, \tau_4, l_1)$ are:

$$\rho_{\pi_0} : l' = 4 \ \wedge \ r' = 5$$
$$\rho_{\pi_1} : l \geq 0 \ \wedge \ r < 8 \ \wedge \ r' = r + 1 \ \wedge \ l' = l \ \wedge \ A'[r] = 0 \ \wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq r \ : \ A'[\alpha] = A[\alpha]$$
$$\rho_{\pi_2} : l \geq 0 \ \wedge \ r < 8 \ \wedge \ l' = l - 1 \ \wedge \ r' = r \ \wedge \ A'[l] = 0 \ \wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \ \wedge \ \alpha \neq l \ : \ A'[\alpha] = A[\alpha] \, .$$

A path is *cyclic* if it contains a cycle. A set of locations $\mathcal{S}$ is a *cutset* if every cyclic path contains a location in $\mathcal{S}$. Locations in a cutset are *cutpoints*. In our example, paths π_1 and π_2 are cyclic, $\{l_1\}$ is a cutset and thus l_1 is a cutpoint.

Let $\mathcal{P}$ be a transition system with initial location ℓ_0, and $\mathcal{S}$ a cutset of $\mathcal{P}$. We call the *control-flow-graph of $\mathcal{P}$ induced by $\mathcal{S}$* the graph whose nodes are $\mathcal{N} = \{\ell_0\} \cup \mathcal{S}$, and such that for every path π_{ij} in the transition system connecting two locations ℓ_i and ℓ_j of $\mathcal{N}$ there exists a directed edge $\langle \ell_i, \ell_j, \pi_{ij} \rangle$. Note that therefore, every edge of the graph has an associated path in the transition system.

For a given strongly connected component (SCC) s of the control-flow-graph, its *initiation* paths are those paths in the transition system that label an edge from a location out of s to a location in s, and its *consecution* paths are those labeling an edge connecting only locations in s. For instance, the control-flow graph resulting from taking the cutset $\{l_1\}$ in our example has two nodes, l_0 and l_1, with one edge from l_0 to l_1 (π_0), and two self-edges at l_1 (π_1 and π_2). Thus, the SCC consisting of l_1 has one initiation path (π_0), and two consecution paths (π_1 and π_2).

2.2 Constraint-Based Invariant Generation

Here we review the *constraint-based invariant generation* approach [1]. Let us assume that we have selected all cutpoints, obtained all the SCCs and identified all respective initiation and consecution paths. The following well-known theorem establishes sufficient conditions for a set of properties to be invariant at the cutpoints:

Theorem 1. *Let $l_1^C, \ldots, l_p^C$ be a cutset of a SCC s. Let $P_1, \ldots, P_p$ be properties over the program variables $\overline{u}$ such that the following implications hold:*

i) for all initiation paths π^I from some l to some l_i^C: $\quad \forall \overline{u}, \overline{u}' \ \rho_{\pi^I} \quad \Rightarrow P_i'$
ii) for all consecution paths π^C from some l_j^C to some l_i^C: $\forall \overline{u}, \overline{u}' \ \rho_{\pi^C} \wedge P_j \Rightarrow P_i'$

Then $P_1, \ldots, P_p$ are invariant at $l_1^C, \ldots, l_p^C$. We say $P_1, \ldots, P_p$ are inductive invariants.

The idea of the constraint-based method is to consider a template for candidate invariant properties, e.g., linear inequalities in the scalar variables. These templates involve both program variables as well as parameters whose values are initially unknown and have to be determined so as to ensure invariance. To this end, the implications in Theorem 1 are expressed by means of *constraints* (hence the name of the approach) on the unknowns. If implications are encoded soundly, any solution to the constraints yields invariant properties for the cutpoints. In particular, if linear inequalities are taken as target invariants as in [1], implications can be transformed into arithmetic constraints over the unknowns by means of the following result from polyhedral geometry:

Theorem 2 (Farkas' Lemma [3]). *Consider a system S of linear inequalities $a_{i1}x_1 + \cdots + a_{in}x_n + b_i \leq 0$ ($i \in \{1, \ldots, m\}$) over real-valued variables $x_1, \ldots, x_n$. When S is satisfiable, it entails a linear inequality $c_1x_1 + \cdots + c_nx_n + d \leq 0$ iff there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_m$, such that $c_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \ldots, c_n = \sum_{i=1}^{m} \lambda_i a_{in}$, $d = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$. Further, S is unsatisfiable iff the inequality $1 \leq 0$ can thus be derived.*

Therefore, Farkas' Lemma allows one to transform an $\exists\forall$ problem into an $\exists$ problem. If all a_{ij} and b_i are known values, the resulting satisfiability problem is an SMT problem over linear arithmetic. Otherwise, an SMT problem over non-linear arithmetic is obtained. Moreover, if one is interested in linear invariants with integer coefficients, as some unknowns are integer (the invariant coefficients) and some are real (the multipliers $\lambda_0, \lambda_1, \ldots, \lambda_m$), an SMT problem in mixed arithmetic is obtained. However, as Farkas' Lemma applies to reals, one may lose some inductive invariants, namely those that only hold using the fact that the program variables are integers.

3 Array Invariants

In this section we present a constraint-based technique for generating array invariants for loop programs without nesting. Moreover, programs are assumed to contain linear expressions in assignments, `if` and `while` conditions, as well as in array accesses.

The idea of the method is, similarly as in [1], to express the conditions of Theorem 1 as algebraic constraints on the parameters of a prefixed invariant template. In order to provide the reader with intuition on how this is achieved, let us consider again the example in Fig. 1. In this program, an array A is filled with zeros from the middle outwards, moving alternatively to the left and to the right. Let us show that property $P \equiv \forall\alpha : 0 \leq \alpha < r - l - 1 : A[\alpha + l + 1] = 0$ is an inductive invariant for this program.

First of all, let us prove that initiation paths (namely, π_0) entail the property. In particular, we have to prove that $l' = 4 \wedge r' = 5 \rightarrow P'$.[1] This is trivial, since $l' = 4$ and $r' = 5$ imply that $r' - l' - 1$ is 0, i.e., the domain of the universally quantified variable α in P' is empty.

In general, our invariant generation method is aimed at universally quantified formulas, and we ensure that initiation paths imply the invariants by forcing that the domains of the universally quantified variables are empty.

Secondly, let us prove that consecution paths (i.e., π_1 and π_2) preserve the property. For example, for π_1 we have to prove that

$$P \wedge l \geq 0 \wedge r < 8 \wedge r' = r + 1 \wedge l' = l \wedge A'[r] = 0$$
$$\wedge \forall\alpha : 0 \leq \alpha < 8 \wedge \alpha \neq r : A'[\alpha] = A[\alpha] \rightarrow P' .$$

Now notice that the expression $r' - l' - 1$, which determines the domain of α in P', also has the property that $r' - l' - 1 = (r+1) - l - 1 = (r - l - 1) + 1$. This means that, after π_1, the domain of α has exactly one new element, $\alpha = r - l - 1$. First, let us see that, after the path, property $A'[\alpha + l' + 1] = A'[\alpha + l + 1] = 0$ holds for the other values of α, i.e., $\alpha \in \{0, \ldots, r - l - 2\}$. Indeed this is the case: since $\forall\alpha : 0 \leq \alpha < 8 \wedge \alpha \neq r : A'[\alpha] = A[\alpha]$,

[1] From now on, program variables and their primed versions are universally quantified.

all positions of A' except for the r-th remain the same. But $A'[r] = A'[(r-l-1)+l'+1]$ precisely corresponds to $\alpha = r-l-1$. Hence from P we have that $A'[\alpha+l'+1] = 0$ for all $\alpha \in \{0,\ldots,r-l-2\}$. Now we only need to prove $A'[\alpha+l'+1] = 0$ for $\alpha = r-l-1$, which follows from the premise $A'[r] = 0$. In conclusion, P' holds.

In general, our invariant generation method will require that, after each consecution path, at most one new element is added to the domain of our universally quantified invariant, and that the contents of the arrays involved in the invariant are not changed after the path.

Back to the example, as regards π_2 we have to prove that

$$P \ \wedge \ l \geq 0 \ \wedge \ r < 8 \ \wedge \ l' = l-1 \ \wedge \ r' = r \ \wedge \ A'[l] = 0$$
$$\wedge \ \forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq l \ : \ A'[\alpha] = A[\alpha] \ \rightarrow \ P' \, .$$

Again, the expression $r'-l'-1$ also satisfies that $r'-l'-1 = r-(l-1)-1 = (r-l-1)+1$. Hence the domain of α has exactly one new element. But unlike in the previous case, l changes its value. To prove P' from P, it is convenient to rewrite P so that array accesses are expressed in terms of $A[\alpha+l'+1]$. By making a shift, P is equivalent to $\forall \alpha \ : \ 1 \leq \alpha < r'-l'-1 \ : \ A[\alpha+l'+1] = 0$. Again, since $\forall \alpha \ : \ 0 \leq \alpha < 8 \wedge \alpha \neq l \ : \ A'[\alpha] = A[\alpha]$, all positions of A' except for the l-th remain the same. But $A'[l] = A'[l'+1]$ precisely corresponds to $\alpha = 0$. Therefore $A'[\alpha+l'+1] = 0$ for all $\alpha \in \{1,\ldots,r'-l'-2\}$. Further, as $A'[l] = 0$, we have that $A'[\alpha+l'+1] = 0$ for $\alpha = 0$. Thus P' holds.

Apart from proving that P is invariant, we may also want to check that the array accesses that occur in it are correct. As regards initiation paths, since the domain of α after π_0 is empty, there is nothing to check. Regarding consecution paths, for example for π_1 we have to see that

$$l \geq 0 \wedge r < 8 \wedge r' = r+1 \wedge l' = l \rightarrow \forall \alpha : 0 \leq \alpha < r'-l'-1 : \alpha+l'+1 \geq 0 \wedge \alpha+l'+1 < 8,$$

where for the sake of simplicity we have ignored the array variable. Now, given that array accesses are linear functions in α, it is sufficient to check correctness for $\alpha = 0$ and $\alpha = r'-l'-2$, i.e., that the above premises entail $l'+1 \geq 0 \ \wedge \ l'+1 < 8 \ \wedge \ r'-1 \geq 0 \wedge r'-1 < 8$. Let us assume that we have already looked for linear inequality invariants over scalar variables (e.g., with the techniques in [1, 4]), and have found that $l \leq r-1$ is a loop invariant. Adding this invariant to the transition relation suffices to prove the above implication. A similar argument applies for π_2.

In general, our invariant generation method guarantees that the array accesses occurring in the synthesized invariants are correct. As in the example, this is achieved by ensuring that the accesses of the extreme values of universally quantified variables are correct. Since this often requires arithmetic properties of the scalar variables of the program, in practice it is convenient that, prior to the application of our array invariant generation techniques, a linear relationship analysis for the scalar variables has already been carried out.

3.1 Invariant Generation for Programs with Arrays

Let $\overline{a} = (A_1,\ldots,A_m)$ be the tuple of array variables. Given a positive integer $k > 0$, our method generates invariants of the form

$$\forall \alpha \ : \ 0 \leq \alpha \leq C(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \leq \ 0 \, ,$$

where C, $\mathcal{E}_{ij}$ and $\mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables $\overline{v} = (v_1, \ldots, v_n)$ and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$, for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k\}$.

This template covers a quite general family of properties. See Sect. 5 for a sample of diverse programs for which we can successfully produce useful invariants and which cannot be handled by already existing techniques.

The invariant generation process at the cutpoint of the unnested loop under consideration is split into three steps, in order to make the approach computationally feasible:

1. Expressions C are generated such that the domain $\{0 \ldots C-1\}$ is empty after every initiation path reaching the cutpoint, and C does not change or is increased by one after every consecution path. This guarantees that any property universally quantified with this domain holds after all initiation paths and the domain includes at most one more element after every consecution path. We avoid the synthesis of different expressions that under the known information define the same domain. In the running example, we generate $C(l, r) = r - l - 1$.
2. For every expression C obtained in the previous step and for every array A_i, linear expressions $d_i\alpha + \mathcal{E}_i$ over the scalar variables are generated such that: (*i*) $A_i[d_i\alpha + \mathcal{E}_i]$ is a correct access for all α in $\{0 \ldots C-1\}$; (*ii*) none of the already considered positions in the quantified property is changed after any execution of the consecution paths; and (*iii*), after every consecution path, either $\mathcal{E}_i$ does not change or its value is $\mathcal{E}_i - d_i$. Namely, if C does not change, then $\mathcal{E}'_i = \mathcal{E}_i$ ensures that the invariant is preserved. Otherwise, the invariant has to be extended for a new value of α. If $\mathcal{E}_i$ does not change, from the previous condition for all $\alpha \in \{0, \ldots, C-1\}$ we have $A'_i[d_i\alpha + \mathcal{E}'_i] = A_i[d_i\alpha + \mathcal{E}_i]$. So we will try to extend the invariant with $\alpha = C$. Otherwise, if $\mathcal{E}'_i = \mathcal{E}_i - d_i$, then for all $\alpha \in \{1, \ldots, C\}$ we have $A'_i[d_i\alpha + \mathcal{E}'_i] = A_i[d_i(\alpha - 1) + \mathcal{E}_i]$. So we will try to extend the invariant with $\alpha = 0$.
   In the running example, we generate $d_{11} = 1$ and $\mathcal{E}_{11} = l + 1$.
3. For the selected C we choose k expressions $\mathcal{E}_{ij}$ for every array A_i among the generated $\mathcal{E}_i$, such that for each consecution path either all selected $\mathcal{E}_{ij}$ remain the same after the path, or all have as new value $\mathcal{E}_{ij} - d_{ij}$ after the path. Then, in order to generate invariant properties we just need to find integer coefficients a_{ij} and b_α and an expression $\mathcal{B}$ such that, depending on the case, either the property is fulfilled when $\alpha = C$ at the end of all consecution paths that increase C or it is fulfilled when $\alpha = 0$ at the end of all consecution paths that increase C. Further, $\mathcal{B}$ and b_α have to fulfill that the quantified property is maintained for $\alpha \in \{0 \ldots C-1\}$, assuming that the contents of the already accessed positions are not modified.
   For instance, in the running example for $k = 1$ we generate $a_{11} = 1$, $\mathcal{B} = b_\alpha = 0$, corresponding to the invariant $\forall \alpha : 0 \leq \alpha < r - l - 1 : A[\alpha + l + 1] \leq 0$; and $a_{11} = -1$, $\mathcal{B} = b_\alpha = 0$, corresponding to the invariant $\forall \alpha : 0 \leq \alpha < r - l - 1 : -A[\alpha + l + 1] \leq 0$.

Next we formalize all these conditions, which ensure that every solution to the last phase provides an invariant, and show how to encode them as SMT problems.

While for scalar linear templates the conditions of Theorem 1 can be directly transformed into constraints over the parameters [1], this is no longer the case for our template of array invariants. To this end we particularize Theorem 1 in a form that is suitable

for the constraint-based invariant generation method. The proof of this specialized theorem, given in detail below, mimics the proof of invariance of the running example given at the beginning of this section.

Let $\pi^I_1 \dots \pi^I_p$ be the initiation paths to our cutpoint and $\pi^C_1 \dots \pi^C_q$ the consecution paths going back to the cutpoint.

Theorem 3. *Let C, $\mathcal{B}$ and $\mathcal{E}_{ij}$ be linear polynomials with integer coefficients over the scalar variables, and a_{ij}, d_{ij}, $b_\alpha \in \mathbb{Z}$, for $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$. If*

1. *Every initiation path π^I_r with transition relation $\rho_{\pi^I_r}$ satisfies $\rho_{\pi^I_r} \Rightarrow C' = 0$.*
2. *For all consecution paths π^C_s with transition relation $\rho_{\pi^C_s}$, we have $\rho_{\pi^C_s} \Rightarrow (C' = C \vee C' = C + 1)$.*
3. *For all consecution paths π^C_s, all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$, we have $\rho_{\pi^C_s} \wedge C' > 0 \Rightarrow 0 \le \mathcal{E}'_{ij} \le |A_i| - 1 \wedge 0 \le d_{ij}(C' - 1) + \mathcal{E}'_{ij} \le |A_i| - 1$.*
4. *For all consecution paths π^C_s we have either*
   (a) $\rho_{\pi^C_s} \wedge C' > 0 \Rightarrow \mathcal{E}'_{ij} = \mathcal{E}_{ij}$ for all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$, or
   (b) $\rho_{\pi^C_s} \Rightarrow C' = C + 1 \wedge \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$ for all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$.
5. *For all consecution paths π^C_s, we have $\rho_{\pi^C_s} \Rightarrow \forall \alpha : 0 \le \alpha \le C - 1 : A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] = A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ for all $i \in \{1 \dots m\}$ and $j \in \{1 \dots k\}$.*
6. *For all consecution paths π^C_s, we have*
   – *$\rho_{\pi^C_s} \wedge C' = C + 1 \Rightarrow \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}C + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha C \le 0$, if case 4a applies.*
   – *$\rho_{\pi^C_s} \Rightarrow \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[\mathcal{E}'_{ij}] + \mathcal{B}' \le 0$, if case 4b applies.*
7. *For all consecution paths π^C_s, we have*
   – *$\rho_{\pi^C_s} \wedge 0 \le \alpha \le C - 1 \wedge x + \mathcal{B} + b_\alpha\alpha \le 0 \Rightarrow x + \mathcal{B}' + b_\alpha\alpha \le 0$ for some fresh universally quantified variable x, if case 4a applies.*
   – *$\rho_{\pi^C_s} \wedge 0 \le \alpha \le C - 1 \wedge x + \mathcal{B} + b_\alpha\alpha \le 0 \Rightarrow x + \mathcal{B}' + b_\alpha(\alpha + 1) \le 0$ for some fresh universally quantified variable x, if case 4b applies.*

Then $\forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha\alpha \le 0$ is invariant.

Proof. Following Theorem 1, we show that the property holds after each initiation path, and that it is maintained after each consecution path.

The first condition easily holds by applying 1, since we have that $\rho_{\pi^I_r} \Rightarrow C' = 0$ for every initiation path π^I_r, which implies $\forall \alpha : 0 \le \alpha \le C' - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$, since the domain of the quantifier is empty.

For the consecution conditions we have to show that for all consecution paths π^C_s, we have $\rho_{\pi^C_s} \wedge \forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha\alpha \le 0$ implies $\forall \alpha : 0 \le \alpha \le C' - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$.

By condition 2, we have $\rho_{\pi^C_s} \Rightarrow (C' = C \vee C' = C + 1)$, and by condition 4 either $\rho_{\pi^C_s} \wedge C' > 0 \Rightarrow \mathcal{E}'_{ij} = \mathcal{E}_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$, or $\rho_{\pi^C_s} \Rightarrow C' = C + 1 \wedge \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$ for all $i \in \{1 \dots m\}$, $j \in \{1 \dots k\}$. We distinguish three cases:

1. $C' = C$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij}$. Then we have to ensure $\forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$. By condition 5, we can replace A'_i by A_i in the given domain, and hence we have to show that $\forall \alpha : 0 \le \alpha \le C - 1 : \Sigma^m_{i=1}\Sigma^k_{j=1} a_{ij}A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha\alpha \le 0$. Then, since the array part coincides with

the one of the assumption, we can replace it in both places by some fresh variable x. Now it suffices to show that, assuming $x + \mathcal{B} + b_\alpha \alpha \leq 0$, we have $x + \mathcal{B}' + b_\alpha \alpha \leq 0$ for all value of x, which follows from the premises and condition 7.

2. $C' = C + 1$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. By conditions 1 and 2 we have $0 \leq C$, and hence $C = C' - 1$ belongs to the domain $\{0 \ldots C\}$ and $C' > 0$. Then, by condition 3, we have that $0 \leq d_{ij} C + \mathcal{E}_{ij} \leq |A_i| - 1 = |A'_i| - 1$ for all i and j. Therefore, we can extract the case $\alpha = C$ from the quantifier obtaining $\forall \alpha : 0 \leq \alpha \leq C - 1 : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ and $\Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij} C + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha C \leq 0$. The first part holds as before by the premises and conditions 5 and 7, and the second part holds by the premises and condition 6.
3. $C' = C + 1$ and all $\mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$. Then we have to ensure $\forall \alpha : 0 \leq \alpha \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij} - d_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. Since, by conditions 1 and 2, we have $0 \leq C$, we have that C belongs to the domain $\{0 \ldots C\}$. By condition 3, we have $0 \leq \mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij} \leq |A'_i| - 1$. Therefore, we can extract the case $\alpha = 0$ from the quantifier obtaining $\forall \alpha : 1 \leq \alpha \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij} - d_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha \alpha \leq 0$ and $\Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}_{ij} - d_{ij}] + \mathcal{B}' \leq 0$. For the first one, replacing α by $\alpha + 1$ we have $\forall \alpha : 1 \leq \alpha + 1 \leq C : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}(\alpha + 1) + \mathcal{E}_{ij} - d_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$, or equivalently $\forall \alpha : 0 \leq \alpha \leq C - 1 : \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \Sigma_{u=1}^{n} \mathcal{B}' + b_\alpha(\alpha + 1) \leq 0$, which holds by applying conditions 5 and 7 as before. The second part holds again by the premises and condition 6, using the fact that $\mathcal{E}'_{ij} = \mathcal{E}_{ij} - d_{ij}$. □

As we have described, our invariant generation method consists of three phases. The first phase looks for expressions C satisfying conditions 1 and 2. The second one provides, for every generated C and for every array A_i, expressions $\mathcal{E}_i$ with their corresponding integers d_i that fulfill conditions 3, 4 and 5. Note that, to satisfy condition 4, we need to record for each expression and path whether we have $\mathcal{E}'_i = \mathcal{E}_i$ or $\mathcal{E}'_i = \mathcal{E}_i - d_i$, so as to ensure that all expressions $\mathcal{E}_{ij}$ have the same behavior. Finally, in the third phase we have to find coefficients a_{ij} and b_α and an expression $\mathcal{B}$ fulfilling conditions 6 and 7.

Solutions to all three phases are obtained by encoding the conditions of Theorem 3 into SMT problems in non-linear arithmetic thanks to Farkas' Lemma. Note that, because of array updates, transition relations may not be conjunctions of literals (i.e., atomic predicates or negations of atomic predicates). As in practice the guarded array information is useless until the last phase, in the first two phases we use the unconditional part of a transition relation ρ, i.e., the part of ρ that is a conjunction of literals, denoted by $U(\rho)$.

3.2 Encoding Phase 1

Let C be $c_1 v_1 + \ldots + c_n v_n + c_{n+1}$, where $\overline{v}$ are the scalar variables and $\overline{c}$ are the integer unknowns. Then conditions 1 and 2 can be expressed as:

$$\exists \overline{c} \forall \overline{v}, \overline{v'} \ \bigwedge_{r=1}^{p} (U(\rho_{\pi_r^I}) \Rightarrow C' = 0) \wedge \ \bigwedge_{s=1}^{q} (U(\rho_{\pi_s^C}) \Rightarrow C' = C \ \vee \ C' = C + 1) \, .$$

We cannot apply Farkas' Lemma directly due to the disjunction in the conclusion of the second condition. To solve this, we move one of the two literals into the premise

and negate it. As the literal becomes a disequality, it can be split into a disjunction of inequalities. Finally, thanks to the distributive law, Farkas' Lemma can be applied and an existentially quantified SMT problem in non-linear arithmetic is obtained. We also encode the condition that each newly generated C must be different from all previously generated expressions at the cutpoint, considering all already known scalar invariants.

3.3 Encoding Phase 2

Here, for each C obtained in the previous phase and for each array A_i, we generate expressions $\mathcal{E}_i$ and integers d_i that satisfy conditions 3 and 5, and also condition 4 as a single expression and not combined with the other expressions.

The encoding of condition 3 is direct using Farkas' Lemma. Now let us sketch the encoding of condition 4. Let $\mathcal{E}_i$ be $e_1 v_1 + \ldots + e_n v_n + e_{n+1}$, where $\overline{e}$ are integer unknowns. Then, as $\mathcal{E}_i$ is considered in isolation, we need

$$\exists \overline{e}, d_i \; \forall \overline{v}, \overline{v'} \;\; \bigwedge_{s=1}^{q} \rho_{\pi_s^C} \Rightarrow ((C' = C + 1 \;\wedge\; \mathcal{E}'_i = \mathcal{E}_i - d_i) \;\vee\; C' \leq 0 \;\vee\; \mathcal{E}'_i = \mathcal{E}_i)\,.$$

To apply Farkas' Lemma, we use a similar transformation as for condition 2. In addition, it is imposed that the newly generated expressions are different from the previous ones.

Regarding condition 5, the encoding is rather different. In this case, for every consecution path π_s^C, array A_i and expression $G \Rightarrow A'_i[W] = M$ in $\rho_{\pi_s^C}$, we ensure that

$$\forall \alpha \; (\rho_{\pi_s^C} \;\wedge\; 0 \leq \alpha \leq C - 1 \;\wedge\; G \Rightarrow \; (W \neq d_i \alpha + \mathcal{E}_i \;\vee\; M = A_i[W]))\,.$$

To avoid generating useless expressions, we add in the encoding a condition stating that if $\mathcal{E}'_i = \mathcal{E}_i$ then for every consecution path where C is incremented, there is at least an access $A_i[W]$ in the path such that $W = d_i(C' - 1) + \mathcal{E}'_i$. Otherwise, i.e., if $\mathcal{E}'_i = \mathcal{E}_i - d_i$, then for every consecution path where C is incremented, there is at least an access $A_i[W]$ in the path such that $W = \mathcal{E}'_i$.

3.4 Encoding Phase 3

Condition 7 is straightforward. Regarding condition 6, the encoding does not need non-linear arithmetic, but requires to handle arrays:

$$\begin{array}{l}
\exists \overline{a}, \overline{b}, b_\alpha \; \forall \overline{v}, \overline{v'}, \overline{A}, \overline{A'} \\
\bigwedge_{s=1}^{q} \; (\rho_{\pi_s^C} \quad\quad\quad\quad\quad\quad \Rightarrow \;\; \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[\mathcal{E}'_{ij}] \; + \mathcal{B}' \;\leq\; 0) \quad\quad \wedge \\
\qquad\; (\rho_{\pi_s^C} \;\wedge\; C' = C + 1 \;\; \Rightarrow \;\; \Sigma_{i=1}^{m} \Sigma_{j=1}^{k} a_{ij} A'_i[C + \mathcal{E}'_{ij}] \; + \mathcal{B}' + b_\alpha C \;\leq\; 0)\,.
\end{array}$$

Here, the use of the guarded array information is crucial. However, since we want to apply Farkas' Lemma, array accesses have to be replaced by new universally quantified integer variables. In order to avoid losing too much information, we add the array read semantics after the replacement; i.e., if $A[i]$ and $A[j]$ have been respectively replaced by fresh variables z_i and z_j, then $i = j \Rightarrow z_i = z_j$ is added.

4 Extensions

4.1 Relaxations on Domains

Let us consider the following program:

```
int A[2*N], min, max, i;
if (A[0] < A[1]) { min = A[0]; max = A[1]; }
else             { min = A[1]; max = A[0]; }
for (i = 2; i < 2*N; i += 2) {
  int tmpmin, tmpmax;
  if (A[i] < A[i+1]) { tmpmin = A[ i ]; tmpmax = A[i+1]; }
  else               { tmpmin = A[i+1]; tmpmax = A[ i ]; }
  if (max < tmpmax) max = tmpmax;
  if (min > tmpmin) min = tmpmin;  }
```

It computes the minimum and the maximum of an even-length array simultaneously, using a number of comparisons which is 1.5 times its length. To prove correctness, the invariants $\forall \alpha : 0 \leq \alpha \leq i - 1 : v[\alpha] \geq min$ and $\forall \alpha : 0 \leq \alpha \leq i - 1 : v[\alpha] \leq max$ are required. To discover them, two extensions of Theorem 3 are required:

- The domain of the universally quantified variable α cannot be forced to be initially empty. In this example, when the loop is entered, both invariants already hold for $\alpha = 0, 1$. This can be handled by applying our invariant generation method as described in Sect. 3.1, and for each computed invariant trying to extend the property for decreasing values of $\alpha = -1, -2$, etc. as much as possible. Finally, a shift of α is performed so that the domain of α begins at 0 and the invariant can be presented in the form of Sect. 3.1.
- The domain of the universally quantified variable α cannot be forced to increase at most by one at each loop iteration. For instance, in this example at each iteration the invariants hold for two new positions of the array. Thus, for a fixed parameter Δ, Condition 2 in Theorem 3 must be replaced by $\rho_{\pi_s^C} \Rightarrow (C' = C \vee C' = C + 1 \vee \cdots \vee C' = C + \Delta)$. In this example, taking $\Delta = 2$ is required. Further, conditions 4b, 6 and 7 must also be extended accordingly in the natural way.

4.2 Sorted Arrays

The program below implements binary search: given a non-decreasingly sorted array A and a value x, it determines whether there is a position in A containing x:

```
assume(N > 0);
int A[N], l = 0, u = N-1;
while (l <= u) {
  int m = (l+u)/2;
  if      (A[m] < x) l = m+1;
  else if (A[m] > x) u = m-1;
  else break;  }
```

To prove that, on exiting due to $l > u$, the property $\forall\alpha : 0 \leq \alpha \leq N-1 : A[i] \neq x$ holds, one can use that $\forall\alpha : 0 \leq \alpha \leq l-1 : A[\alpha] < x$ and $\forall\alpha : u+1 \leq \alpha \leq N-1 : A[\alpha] > x$ are invariant. To synthesize them, the fact that A is sorted must be taken into account. The following theorem results from incorporating the property of sortedness into Theorem 3:

Theorem 4. *Let C, $\mathcal{B}$ and $\mathcal{E}_{ij}$ be linear polynomials with integer coefficients over the scalar variables, and a_{ij}, d_{ij}, $b_\alpha \in \mathbb{Z}$, for $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$. If*

1. *For all $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$ we have $b_\alpha \geq 0$, and $d_{ij} > 0 \Rightarrow a_{ij} \geq 0$, and $d_{ij} < 0 \Rightarrow a_{ij} \leq 0$.*
2. *Each initiation path π_r^I with transition relation $\rho_{\pi_r^I}$ fulfills $\rho_{\pi_r^I} \Rightarrow C' = 0$.*
3. *Each initiation path π_r^I with transition relation $\rho_{\pi_r^I}$ fulfills $\rho_{\pi_r^I} \Rightarrow \forall\beta : 0 < \beta \leq |A_i'| - 1 : A_i'[\beta - 1] \leq A_i'[\beta]$ for all $i \in \{1 \ldots m\}$.*
4. *Each consecution path π_s^C with transition relation $\rho_{\pi_s^C}$ fulfills $\rho_{\pi_s^C} \Rightarrow C' \geq C$.*
5. *For all consecution paths π_s^C all $i \in \{1 \ldots m\}$ and $j \in \{1 \ldots k\}$ we have $\rho_{\pi_s^C} \wedge C' > 0 \Rightarrow 0 \leq \mathcal{E}'_{ij} \leq |A_i| - 1 \wedge 0 \leq d_{ij}(C'-1) + \mathcal{E}'_{ij} \leq |A_i| - 1$.*
6. *For all consecution paths π_s^C we have one of the following:*
   (a) *$\rho_{\pi_s^C} \wedge C' > 0 \wedge a_{ij} > 0 \Rightarrow \mathcal{E}'_{ij} \leq \mathcal{E}_{ij}$ and $\rho_{\pi_s^C} \wedge C' > 0 \wedge a_{ij} < 0 \Rightarrow \mathcal{E}'_{ij} \geq \mathcal{E}_{ij}$ for all $i \in \{1 \ldots m\}$, $j \in \{1 \ldots k\}$;*
   (b) *$\rho_{\pi_s^C} \Rightarrow C' > C$ and $\rho_{\pi_s^C} \wedge a_{ij} > 0 \Rightarrow \mathcal{E}'_{ij} \leq \mathcal{E}_{ij} - (C'-C)d_{ij}$ and $\rho_{\pi_s^C} \wedge a_{ij} < 0 \Rightarrow \mathcal{E}'_{ij} \geq \mathcal{E}_{ij} - (C'-C)d_{ij}$ for all $i \in \{1 \ldots m\}$, $j \in \{1 \ldots k\}$.*
7. *For all consecution paths π_s^C, we have $\rho_{\pi_s^C} \Rightarrow \forall\beta : 0 \leq \beta \leq |A_i| - 1 : A_i'[\beta] = A_i[\beta]$ for all $i \in \{1 \ldots m\}$.*
8. *For all consecution paths π_s^C, we have*
   - *$\rho_{\pi_s^C} \wedge C' > C \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}(C'-1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(C'-1) \leq 0$, if case 6a applies.*
   - *$\rho_{\pi_s^C} \Rightarrow \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}(C'-C-1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(C'-C-1) \leq 0$, if case 6b applies.*
9. *For all consecution paths π_s^C, we have*
   - *$\rho_{\pi_s^C} \wedge 0 \leq \alpha \leq C-1 \wedge x + \mathcal{B} + b_\alpha\alpha \leq 0 \Rightarrow x + \mathcal{B}' + b_\alpha\alpha \leq 0$ for some fresh universally quantified variable x, if case 6a applies.*
   - *$\rho_{\pi_s^C} \wedge 0 \leq \alpha \leq C-1 \wedge x + \mathcal{B} + b_\alpha\alpha \leq 0 \Rightarrow x + \mathcal{B}' + b_\alpha(\alpha + C' - C) \leq 0$ for some fresh universally quantified variable x, if case 6b applies.*

Then $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha\alpha \leq 0$ is invariant.

Proof. First of all, let us remark that arrays are always sorted in non-decreasing order, and that their contents are never changed. This follows by induction from conditions 3 and 7. Moreover, it can also be seen from conditions 2 and 4 that $C \geq 0$ is an invariant property.

Now, we will show that the property in the statement of the theorem holds after every initiation path reaching our cutpoint and that it is maintained after every consecution path going back to the cutpoint.

The first condition easily holds applying 2, since we have that $\rho_{\pi_r^I} \Rightarrow C' = 0$ for every initiation path π_r^I, which implies $\forall\alpha : 0 \leq \alpha \leq C'-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i'[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha\alpha \leq 0$, since the domain of the quantifier is empty.

For the consecution conditions we have to show that for all consecution paths π_s^C, we have $\rho_{\pi_s^C} \wedge \forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ implies $\forall\alpha : 0 \leq \alpha \leq C'-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$.

By condition 4, we have that $\rho_{\pi_s^C} \Rightarrow C' \geq C$. We distinguish three cases:

1. $C' = C$ and case 6a holds. If $C' = 0$ there is nothing to prove. Otherwise $C' > 0$, and by hypothesis we have that $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$. Together with $\rho_{\pi_s^C}$, this implies $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ by instantiating appropriately x in condition 9.
   Now, let us show that for all $i \in \{1 \ldots m\}$, for all $j \in \{1 \ldots k\}$ and for all $\alpha \in \{0 \ldots C-1\}$ we have $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$. Let us consider three subcases:
   - $a_{ij} > 0$. Then $\mathcal{E}'_{ij} \leq \mathcal{E}_{ij}$ by condition 6. Hence for all $\alpha \in \{0 \ldots C-1\}$ we have $d_{ij}\alpha + \mathcal{E}'_{ij} \leq d_{ij}\alpha + \mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$.
   - $a_{ij} < 0$. Then $\mathcal{E}'_{ij} \geq \mathcal{E}_{ij}$ by condition 6. Hence for all $\alpha \in \{0 \ldots C-1\}$ we have $d_{ij}\alpha + \mathcal{E}'_{ij} \geq d_{ij}\alpha + \mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \geq A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order (note that, by condition 5, we have that $0 \leq d_{ij}\alpha + \mathcal{E}'_{ij} \leq |A_i| - 1 = |A'_i| - 1$, so array accesses are within bounds). Therefore $a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}]$.
   - $a_{ij} = 0$. Then the inequality trivially holds.

   Altogether we have that $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$. Now our goal easily follows, taking into account that $C' = C$ and that by condition 7 we can replace A_i by A'_i.
2. $C' > C$ and case 6a holds. Then $C' > 0$, and following the same argument as in the previous case we get that $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$, where A_i has been replaced by A'_i by virtue of condition 7.
   It only remains to prove that $\forall\alpha : C \leq \alpha \leq C'-1 : \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq 0$ (note that, by condition 5, we have that $0 \leq \mathcal{E}'_{ij} \leq |A'_i| - 1$ and $0 \leq d_{ij}(C'-1) + \mathcal{E}'_{ij} \leq |A'_i| - 1$, so again array accesses are within bounds). To this end, let us consider $\alpha \in \{C \ldots C'-1\}$ and let us show that $a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq a_{ij} A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}]$ for all $i \in \{1 \ldots m\}$ and for all $j \in \{1 \ldots k\}$. We distinguish three cases:
   - $d_{ij} > 0$. Then $\alpha \leq C'-1$ implies $d_{ij}\alpha \leq d_{ij}(C'-1)$, and hence $d_{ij}\alpha + \mathcal{E}'_{ij} \leq d_{ij}(C'-1) + \mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \leq A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \geq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} < 0$. Then $\alpha \leq C'-1$ implies $d_{ij}\alpha \geq d_{ij}(C'-1)$, and hence $d_{ij}\alpha + \mathcal{E}'_{ij} \geq d_{ij}(C'-1) + \mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] \geq A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \leq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} = 0$. The goal trivially holds.

   Thus $\Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' \leq \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}] + \mathcal{B}'$. Now, by condition 1 we have $b_\alpha \geq 0$, hence $\alpha \leq C'-1$ implies $b_\alpha \alpha \leq b_\alpha(C'-1)$. Therefore $\Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}\alpha + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha \alpha \leq \Sigma_{i=1}^m \Sigma_{j=1}^k a_{ij} A'_i[d_{ij}(C'-1) + \mathcal{E}'_{ij}] + \mathcal{B}' + b_\alpha(C'-1) \leq 0$ by condition 8.

3. $C' > C$ and case 6b holds (notice that $C' = C$ and case 6b together are not possible). By hypothesis we have $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}\alpha+\mathcal{E}_{ij}]+\mathcal{B}+b_\alpha\alpha \leq 0$. Together with $\rho_{\pi_s^c}$, this implies $\forall\alpha : 0 \leq \alpha \leq C-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}\alpha+\mathcal{E}_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ by instantiating appropriately x in condition 9. By shifting the universally quantified variable the previous formula can be rewritten as $\forall\alpha : C'-C \leq \alpha \leq C'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]+\mathcal{B}'+b_\alpha(\alpha-(C'-C)) \leq 0$.
Now, let us show that for all $i \in \{1\ldots m\}$, for all $j \in \{1\ldots k\}$ and for all $\alpha \in \{C'-C\ldots C'-1\}$ we have $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$. Let us consider three subcases:
   - $a_{ij} > 0$. Then $\mathcal{E}'_{ij} \leq \mathcal{E}_{ij}-(C'-C)d_{ij}$ by condition 6. Hence for all $\alpha \in \{C'-C\ldots C'-1\}$ we have $d_{ij}\alpha+\mathcal{E}'_{ij} \leq d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$.
   - $a_{ij} < 0$. Then $\mathcal{E}'_{ij} \geq \mathcal{E}_{ij}-(C'-C)d_{ij}$ by condition 6. Hence for all $\alpha \in \{C'-C\ldots C'-1\}$ we have $d_{ij}\alpha+\mathcal{E}'_{ij} \geq d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}$. This implies $A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \geq A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$ as A_i is sorted in non-decreasing order. Therefore $a_{ij}A_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A_i[d_{ij}(\alpha-(C'-C))+\mathcal{E}_{ij}]$.
   - $a_{ij} = 0$. Then the inequality trivially holds.

   Altogether we have that $\forall\alpha : C'-C \leq \alpha \leq C'-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$, where A_i has been replaced by A'_i by virtue of condition 7.
   It only remains to prove that $\forall\alpha : 0 \leq \alpha \leq C'-C-1 : \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq 0$ (note that, by condition 5, we have that $0 \leq \mathcal{E}'_{ij} \leq |A'_i|-1$ and $0 \leq d_{ij}(C'-1)+\mathcal{E}'_{ij} \leq |A'_i|-1$, so again array accesses are within bounds). To this end, let us consider $\alpha \in \{0\ldots C'-C-1\}$ and let us show that $a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq a_{ij}A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]$ for all $i \in \{1\ldots m\}$ and for all $j \in \{1\ldots k\}$. We distinguish three cases:
   - $d_{ij} > 0$. Then $\alpha \leq C'-C-1$ implies $d_{ij}\alpha \leq d_{ij}(C'-C-1)$, and hence $d_{ij}\alpha+\mathcal{E}'_{ij} \leq d_{ij}(C'-C-1)+\mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \leq A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \geq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} < 0$. Then $\alpha \leq C'-C-1$ implies $d_{ij}\alpha \geq d_{ij}(C'-C-1)$, and hence $d_{ij}\alpha+\mathcal{E}'_{ij} \geq d_{ij}(C'-C-1)+\mathcal{E}'_{ij}$. As A'_i is sorted in non-decreasing order, we have $A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}] \geq A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]$. Finally, by condition 1 it must be $a_{ij} \leq 0$, and multiplying at both sides the last inequality the goal is obtained.
   - $d_{ij} = 0$. The goal trivially holds.

   Thus $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}' \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}(C'-C-1)+\mathcal{E}'_{ij}]+\mathcal{B}'$. Now, by condition 1 we have $b_\alpha \geq 0$, hence $\alpha \leq C'-C-1$ implies $b_\alpha\alpha \geq b_\alpha(C'-C-1)$. Therefore $\Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}\alpha+\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha\alpha \leq \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij}A'_i[d_{ij}(C'-C-1)\mathcal{E}'_{ij}]+\mathcal{B}'+b_\alpha(C'-C-1) \leq 0$ by condition 8.

□

By means of the previous theorem, (an equivalent version of) the desired invariants can be discovered. However, to the best of our knowledge, results on the synthesis of invariants for programs with sorted arrays are not reported in the literature. See Sect. 5 for other examples that can be handled by means of this extension.

5 Experimental Evaluation

The method presented in Sects. 3 and 4 has been implemented in the tool CppInv[2]. For solving the generated constraints, we use the Barcelogic SMT solver [5]. As discussed in Sect. 2.2, after applying Farkas' Lemma an SMT problem for mixed non-linear arithmetic is obtained. For this theory, Barcelogic has proved to be very effective in finding solutions [2]; e.g., it won the division of quantifier-free non-linear integer arithmetic (QF_NIA) in the 2009 edition of the SMT-COMP competition (`www.smtcomp.org/2009`), and since then no other competing solver in this division has solved as many problems.

In addition to the examples already shown in this paper, CppInv automatically generates array invariants for a number of different programs. The following table shows some of them, together with the corresponding loop invariants:

Heap property:

```
const int N;
assume(N >= 0);
int A[2*N], i;
for (i = 0; 2*i+2 < 2*N; ++i)
  if (A[i]>A[2*i+1] or A[i]>A[2*i+2])
    break;
```

Loop invariants:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] \leq A[2\alpha+2]$
$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] \leq A[2\alpha+1]$

Partial initialization [6]:

```
const int N;
assume(N >= 0);
int A[N], B[N], C[N], i, j;
for (i = 0, j = 0; i < N; ++i)
  if (A[i] == B[i])
    C[j++] = i;
```

Loop invariant:

$\forall \alpha : 0 \leq \alpha \leq j-1 : C[\alpha] \leq \alpha + i - j$
$\forall \alpha : 0 \leq \alpha \leq j-1 : C[\alpha] \geq \alpha$

Array palindrome:

```
const int N;
assume(N >= 0);
int A[N], i;
for (i = 0; i < N/2; ++i)
  if (A[i] != A[N-i-1]) break;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = A[N-\alpha-1]$

Array initialization [6]:

```
const int N;
assume(N >= 0);
int A[N], i;
for (i = 0; i < N; ++i)
  A[i] = 2*i+3;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 2\alpha + 3$

Array insertion:

```
const int N;
int A[N], i, x, j;
assume(0 <= i and i < N);
for (x = A[i], j = i-1;
     j >=0 and A[j] > x; --j)
  A[j+1] = A[j];
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-j-2 : A[i-\alpha] \geq x+1$

Sequential initialization [7]:

```
const int N;
assume(N > 0);
int A[N], i;
for (i = 1, A[0] = 7; i < N; ++i)
  A[i] = A[i-1] + 1;
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-2 : \ A[\alpha+1] = A[\alpha] + 1$

[2] The tool, together with a sample of example programs it can analyze, can be downloaded at `www.lsi.upc.edu/~albert/cppinv-bin.tar.gz`.

Array copy [7]:

```
const int N;
assume(N >= 0);
int A[N], B[N], i;
for (i = 0; i < N; ++i)
  A[i] = B[i];
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = B[\alpha]$

First not null [7]:

```
const int N;
assume(N >= 0);
int A[N], s, i;
for (i = 0, s = N; i < N; ++i)
  if (s == N and A[i] != 0) {
    s=i;
    break;
  }
```

Loop invariant:

$\forall \alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 0$

Array partition [8]:

```
const int N;
assume(N >= 0);
int A[N], B[N], C[N], a, b, c;
for (a=0, b=0, c=0; a < N; ++a)
  if (A[a] >= 0) B[b++]=A[a];
  else           C[c++]=A[a];
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq b-1 : B[\alpha] \geq 0$
$\forall \alpha : 0 \leq \alpha \leq c-1 : C[\alpha] < 0$

Array maximum [7]:

```
const int N;
assume(N > 0);
int A[N], i, max;
for (i = 1, max = A[0]; i < N; ++i)
  if (max < A[i])
    max = A[i];
```

Loop invariant:

$\forall \alpha : 0 \leq \alpha \leq i-1 : A[\alpha] \leq max$

First occurrence:

```
const int N;
assume(N > 0);
int A[N], x = getX(), l, u;
// A is sorted in ascending order
for (l = 0, u = N; l < u; ) {
  int m = (l+u)/2;
  if (A[m] < x) l = m+1;
  else u = m;   }
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq l-1 : A[\alpha] < x$
$\forall \alpha : 0 \leq \alpha \leq N-1-u : A[N-1-\alpha] \geq x$

Sum of pairs:

```
const int N;
assume(N > 0);
int A[N], x = getX(), l = 0, u = N-1;
// A is sorted in ascending order
while (l < u)
  if      (A[l] + A[u] < x) l = l+1;
  else if (A[l] + A[u] > x) u = u-1;
  else break;
```

Loop invariants:

$\forall \alpha : 0 \leq \alpha \leq l-1 : A[\alpha] + A[u] < x$
$\forall \alpha : 0 \leq \alpha \leq N-u-2 : A[N-1-\alpha] + A[l] > x$

As a final experiment, we have run CppInv over a collection of programs written by students. It consists of 38 solutions to the problem of finding the first occurrence of an element in a sorted array of size N in $O(\log N)$ time. These solutions have been taken from the online learning environment for computer programming Jutge.org (see `www.jutge.org`), which is currently being used in several programming courses in the Universitat Politècnica de Catalunya. The benchmark suite corresponds to all submitted iterative programs that have been accepted, i.e., such that for all input tests the output matches the expected one. These programs can be considered more realistic code than the examples above (**First occurrence** program), since most often they are not the most elegant solution but a working one with many more conditional statements than necessary. For example, here is an instance of such a program:

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);
  int e = 0, d = N - 1, m, pos;
  bool found = false, exit  = false;
  while (e <= d and not exit) {
    m = (e+d)/2;
    if      (x > A[m]) {
      if (not found) e = m+1;
      else exit = true;
    }
    else if (x < A[m]) {
      if (not found) d = m-1;
      else exit = true;
    }
    else {
      found = true; pos = m; d = m-1;
    } }
  if (found) {
    while (x  == A[pos-1]) --pos;
    return pos; }
  return -1; }
```

This particular example is interesting because, with the aid of our tool, we realized that it does not work in $O(\log N)$ time as required, and is thus a false positive. Namely, our tool produces the following invariants for the first loop:

$$\forall \alpha : \quad 0 \leq \alpha \leq e-1 \ : A[\alpha] < x\,,$$
$$\forall \alpha : d+1 \leq \alpha \leq N-1 : A[\alpha] \geq x\,.$$

By manual inspection one can see that $found \rightarrow (A[pos] = x \wedge d = pos - 1)$ and $exit \rightarrow found$ are also invariant. Therefore, if on exit of the loop the property $e \leq d$ holds, then *exit* and *found* are true and, with all this information, it is unknown whether the contents of the array between e and $pos - 1$ are equal to x. Since this segment can be arbitrarily long, the second loop may take $O(N)$ time to find the first occurrence of x. This reasoning allowed us to cook an input for which indeed the program behaves linearly. On the other hand, by means of the generated invariants it can be seen that the problem is that the loop may be exited too early, and that by replacing in the first loop the body of the first conditional by `e = m+1` and the second one by `d = m-1`, the program becomes correct and meets the complexity requirements.

In general, for all programs in the benchmark suite our tool was able to find automatically both standard invariants. The time consumed was very different depending on how involved the code was. Anyway, the main problem as regards efficiency is that in its current form our prototype exhaustively generates first all scalar invariants and then, using all of them, generates all array invariants. Further work is needed to heuristically guide the search of scalar invariants, so that only useful information is inferred.

We also applied our tool to some of the submissions rejected by Jutge.org. In some cases the generated invariants helped us to fix the program. E.g., for the following code:

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);
  int i = 0, j = N-1;
  while (i <= j) {
    if (x == A[i]) return  i;
    if (x < A[i])  return -1;
    int m = (i+j)/2;
    if (x < A[m]) j = m-1;
    else          i = m+1; }
  return -1; }
```

In this case, the generated invariants are:

$$\forall \alpha : \quad 0 \le \alpha \le i-1 \ : A[\alpha] \le x\,,$$
$$\forall \alpha : j+1 \le \alpha \le N-1 : A[\alpha] > x\,.$$

One may notice that the first invariant should have a strict inequality, and that this problem may be due to a wrong condition in the last conditional. Indeed, by replacing the condition $x < A[m]$ by $x \le A[m]$, we obtain a set of invariants that allow proving the correctness of the program.

6 Related Work

There is a remarkable amount of work in the literature aimed at the synthesis of quantified invariants for programs with arrays. Some of the techniques fall into the framework of *abstract interpretation* [9]. In [6], the index domain of arrays is partitioned into several symbolic intervals I, and then each subarray $A[I]$ is associated to a summary auxiliary variable A_I. Although assignments to individual array elements can thus be handled precisely, in order to discover relations among the contents at different indices, hints must be manually provided. This shortcoming is overcome in [7], where additionally relational abstract properties of summary variables and shift variables are introduced to discover invariants of the form $\forall \alpha : \alpha \in I : \psi(A_1[\alpha + k_1], ..., A_m[\alpha + k_m], \overline{v})$, where $k_1, \ldots, k_m \in \mathbb{Z}$ and $\overline{v}$ are scalar variables. In comparison with our techniques, the previous approaches force all array accesses to be of the form $\alpha + k$. As a consequence, programs like **Array palindrome** or **Heap property** (see Sect. 5) cannot be handled. Moreover, the universally quantified variable is not allowed to appear outside array accesses. For this reason, our analysis can be more precise, e.g., in the **Array initialization** and the **Partial initialization** [6] examples. Another technique based on abstract interpretation is presented in [10]. While their approach can discover more general properties than ours, it requires that the user provides templates to guide the analysis. Yet another abstract interpretation-based method is given in [11], where *fluid updates* are defined on a symbolic heap abstraction.

Predicate abstraction techniques [12] can also be seen as instances of abstract interpretation. Here, a set of predefined predicates is considered, typically provided manually by the user or computed heuristically from the program code and the assertions to be proved. Then one generates an invariant built only over those predicates. This track of research was initiated in [13], where by introducing Skolem constants, universally-quantified loop invariants for arrays can be discovered. In [14], it is shown how the strongest universally quantified inductive invariant over the given predicates can be generated. Further works integrate predicate abstraction into the CEGAR loop [15, 16], apply algorithmic learning [17] or discover invariants with complex pre-fixed Boolean structure [18]. Unlike most of these predicate abstraction-based techniques, our approach does not require programs to be annotated with assertions, thus allowing one to analyze code embedded into large programs, or with predicates, which sometimes require ingenuity from the user. To alleviate the need of supplying predicates, in [19] *parametric predicate abstraction* was introduced. However, the properties considered

there express relations between all elements of two data collections, while our approach is able to express pointwise relations.

Another group of techniques is based on *first-order theorem proving*. In [20, 21], the authors generate invariants with alternations of quantifiers for loop programs without nesting. First, one describes the loop dynamics by means of first-order formulas, possibly using additional symbols denoting array updates or loop counters. Then a saturation theorem prover eliminates auxiliary symbols and reports the consequences without these symbols, which are the invariants. One of the problems of the method is the limited capability of arithmetic reasoning of the theorem prover (as opposed to SMT solvers, where arithmetic reasoning is hard-wired in the theory solvers). In [22] a related approach is presented, where invariants are generated by examining candidates supplied by an interpolating theorem prover. In addition to suffering from similar arithmetic reasoning problems as [20], the approach also requires program assertions.

Other methods use *computational algebra*, e.g., [23]. One of the limitations of [23] is that array variables are required to be either write-only or read-only. Hence, unlike our method, programs such as **Sequential initialization** [7] and **Array insertion** (see Sect. 5) cannot be handled.

Finally, the technique that has been presented in this paper belongs to the *constraint-based* methods. In this sense it is related to that in [24]. There, the authors present a constraint-based algorithm for the synthesis of invariants expressed in the combined theory of linear integer arithmetic (LI) and uninterpreted function symbols (UIF). By means of the reduction of the array property fragment to LI+UIF, it is claimed that the techniques can be extended for the generation of universally quantified invariants for arrays. However, the language of our invariants is outside the array property fragment, since we can generate properties where indices do not necessarily occur in array accesses (e.g., see the **Array initialization** or the **Partial initialization** examples in Sect. 5). Finally, the technique in [24] is applied in [8] to generating path invariants in the context of the CEGAR loop. As the framework in [8] is independent of any concrete invariant generation technique, we believe that our method could be used as an alternative in a portfolio approach to path invariant-based program analysis.

7 Conclusions and Future Work

In short, the contributions of this paper are:

- *a new constraint-based method for the generation of universally quantified invariants of array programs.* Unlike other techniques, it does not require extra predicates nor assertions. It does not need the user to provide a template either, but it can take advantage of hints by partially instantiating the global template considered here.
- *extensions of the approach for sorted arrays.* To our knowledge, results on the synthesis of invariants for programs with sorted arrays are not reported in the literature.
- *an implementation of the presented techniques that is freely available.* The constraint solving engine of our prototype depends on SMT. Hence, our techniques will directly benefit from any further advances in SMT solving.

For future work, we plan to extend our approach to a broader class of programs. As a first step we plan to relax Theorem 3, so that, e.g., overwriting on positions in which the invariant already holds is allowed. We would also like to handle nested loops, so that for instance sorting algorithms can be analyzed. Another line of work is the extension of the family of properties that our approach can discover as invariants. E.g., a possibility could be considering disjunctive properties, or allowing quantifier alternation. The former allows analyzing algorithms such as sentinel search, while the latter is necessary to express that the output of a sorting algorithm is a permutation of the input.

Moreover, the invariants that our method generates depend on the coefficients and expressions obtained in each of its three phases, which in turn depend on the previous linear relationship analysis of scalar variables. We leave for future research to study how to make the approach resilient to changes in the outcome of the different phases.

Finally, as pointed out in Sect. 5, the efficiency of CppInv can be improved. In particular, further work is needed to heuristically guide the search of scalar invariants, so that only useful information is inferred.

Acknowledgments. We would like to thank Jutge.org team for providing us with programs written by students. We are also grateful to the anonymous referees of a previous version of this paper for their helpful comments.

References

1. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear Invariant Generation Using Non-linear Constraint Solving. In Jr., W.A.H., Somenzi, F., eds.: CAV. Volume 2725 of Lecture Notes in Computer Science., Springer (2003) 420–432
2. Borralleras, C., Lucas, S., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: SAT Modulo Linear Arithmetic for Solving Polynomial Constraints. J. Autom. Reasoning **48**(1) (2012) 107–131
3. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (June 1998)
4. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL. (1978) 84–96
5. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The Barcelogic SMT Solver. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 294–298
6. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In Palsberg, J., Abadi, M., eds.: POPL, ACM (2005) 338–350
7. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In Gupta, R., Amarasinghe, S.P., eds.: PLDI, ACM (2008) 339–348
8. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In Ferrante, J., McKinley, K.S., eds.: PLDI, ACM (2007) 300–309
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
10. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In Necula, G.C., Wadler, P., eds.: POPL, ACM (2008) 235–246
11. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Proceedings of the 19th European conference on Programming Languages and Systems. ESOP'10, Berlin, Heidelberg, Springer-Verlag (2010) 246–266

12. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: CAV. Volume 1254 of Lecture Notes in Computer Science., Springer (1997) 72–83
13. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL. (2002) 191–202
14. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In Steffen, B., Levi, G., eds.: VMCAI. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 267–281
15. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007) 193–206
16. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-based Abstraction For Arrays with Interpolants. In: 24th International Conference on Computer Aided Verification (CAV), Berkeley, California, USA, Springer, Springer (2012)
17. Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In Ueda, K., ed.: APLAS. Volume 6461 of Lecture Notes in Computer Science., Springer (2010) 328–343
18. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In Hind, M., Diwan, A., eds.: PLDI, ACM (2009) 223–234
19. Cousot, P.: Verification by abstract interpretation. In Dershowitz, N., ed.: Verification: Theory and Practice. Volume 2772 of Lecture Notes in Computer Science., Springer (2003) 243–268
20. Kovács, L., Voronkov, A.: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In Chechik, M., Wirsing, M., eds.: FASE. Volume 5503 of Lecture Notes in Computer Science., Springer (2009) 470–485
21. Hoder, K., Kovács, L., Voronkov, A.: Case studies on invariant generation using a saturation theorem prover. In Batyrshin, I., Sidorov, G., eds.: Advances in Artificial Intelligence. Volume 7094 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2011) 1–15
22. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In Ramakrishnan, C.R., Rehof, J., eds.: TACAS. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 413–427
23. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for arrays (tool paper). In Fermüller, C.G., Voronkov, A., eds.: LPAR (Yogyakarta). Volume 6397 of Lecture Notes in Computer Science., Springer (2010) 348–356
24. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In Cook, B., Podelski, A., eds.: VMCAI. Volume 4349 of Lecture Notes in Computer Science., Springer (2007) 378–394