Finding Loop Invariants for Programs over Arrays Using a Theorem Prover ⋆

Laura Kovács[1] and Andrei Voronkov[2]

[1] EPFL
[2] University of Manchester

Abstract. We present a new method for automatic generation of loop invariants for programs containing arrays. Unlike all previously known methods, our method allows one to generate first-order invariants containing alternations of quantifiers. The method is based on the automatic analysis of the so-called *update predicates* of loops. An update predicate for an array A expresses updates made to A. We observe that many properties of update predicates can be extracted automatically from the loop description and loop properties obtained by other methods such as a simple analysis of counters occurring in the loop, recurrence solving and quantifier elimination over loop variables. We run the theorem prover Vampire on some examples and show that non-trivial loop invariants can be generated.

1 Introduction

Invariants with quantifiers are important for verification and static analysis of programs over arrays due to the unbounded nature of array structures. Such invariants can express relationships among array elements and properties involving arrays and scalar variables of the loop, and thus significantly ease the verification task. Automated discovery of array invariants therefore became a challenging topic, see e.g. [9, 20, 10, 12, 3, 11, 22, 13]. Approaches presented in these papers combine inductive reasoning with predicate abstraction, constraint solving and interpolation-based techniques and normally require user guidance in providing necessary templates, assertions or predicates.

In this paper we present a framework for automatically inferring array invariants without any user guidance and without using a priori defined boolean templates or predicates. Moreover, unlike all previously known methods, our method allows one to generate loop invariants containing *quantifier alternations*.

The method is based on the following idea.

1. Given a loop over array and scalar variables, we first try to extract from it various information that can be expressed by first-order formulas. This can be information

⋆ This research was partly done in the frame of the Transnational Access Programme at RISC, Johannes Kepler University Linz, supported by the European Commission Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIEnce (contract No 026133). The first author was supported by the Swiss NSF.

about scalar variables occurring in the loops, such as precise values of these variables in terms of the loop counter, monotonicity properties of these variables considered as functions of the loop counter and polynomial relations among these variables. For extracting this information we deploy techniques from symbolic computation, such as recurrence solving and quantifier elimination, as presented in [18, 14], to perform inductive reasoning over scalar variables.
2. Using the derived loop properties, we then automatically discover first-order properties of the so-called *update predicates* for array variables used in the loop and monotonicity properties for scalar variables. The update predicates describe the positions at which arrays are updated, iterations at which the updates occur and the update values. The first-order information extracted from the loop description can use auxiliary symbols, such as symbols denoting update predicates or loop counters.
3. After having collected the first-order information, we run a saturation theorem prover to eliminate the auxiliary symbols and obtain loop invariants expressed as first-order formulas. When the invariants obtained in this way contain skolem functions, we de-skolemise them into formulas with quantifier alternations.

The main features of the technique presented here are the following.

1. We require no user guidance such as a postcondition or a collection of predicates from which an invariant can be built: all we have is a loop description.
2. We are able to generate automatically complex invariants involving quantifier alternations.

All experiments described in this paper were carried out using two systems: `Aligator` — the package for invariant generation described in [18, 14], and the first-order theorem prover `Vampire` [25].

$a := 0; b := 0; c := 0;$
while $(a \leq k)$ **do**
  if $A[a] \geq 0$
    then $B[b] := A[a]; b := b + 1;$
    else $C[c] := A[a]; c := c + 1;$
  $a := a + 1;$
end do

Fig. 1. Array partitioning [3].

This paper is organised as follows. Section 2 motivates our work with an example. Section 3 presents our program model together with some basic principles of saturation theorem proving. The notion of update predicates is introduced in Section 4 together with properties involving such predicates. Section 5 describes how properties of update predicates and scalar variables are extracted from the loop description. Section 6 presents our method of invariant generation and Section 7 discusses some experiments with the theorem prover `Vampire`. Section 8 focuses on related work. Section 9 concludes the paper with some ideas for future work.

2 Example

In this section we give an example illustrating what kind of loop invariant we would like to generate.

We will use the program of Figure 1 as our running example throughout the paper. The program fills an array B with the non-negative values of a source array A, and an array C with the negative values of A. It is not hard to derive that after n iterations of this loop (assuming $n \leq k$) the value of a is equal to the value of the loop counter n. For example, this property can be derived by the methods of [6, 23] or by the recurrence solving part of `Aligator` [14, 18]. Moreover, `Aligator` is able to find the linear invariant relation $a = b + c$.

Using light-weight analysis, it is also not hard to see that the values of the variables b and c may not decrease during the loop execution, therefore $c \geq 0$ and $b \geq 0$ are loop invariants. This property can also be extracted by `Aligator` using more complex reasoning involving quantifier elimination techniques [14]. However, such a light-weight analysis would not give us much information about arrays A, B, C and their relationships, apart from the fact that the value of A does not change since A is not updated. For example, one may want to derive the following properties of the loop (n denotes the loop counter).

1. Each of $B[0], \ldots, B[b-1]$ is non-negative and equal to one of $A[0], \ldots, A[n-1]$.
2. Each of $C[0], \ldots, C[c-1]$ is negative and equal to one of $A[0], \ldots, A[n-1]$.
3. Each non-negative value in $A[0], \ldots, A[n-1]$ is equal to one of $B[0], \ldots, B[b-1]$.
4. Each negative value in $A[0], \ldots, A[n-1]$ is equal to one of $C[0], \ldots, C[c-1]$.
5. For every $p \geq b$, the value of $B[p]$ is equal to its initial value.
6. For every $p \geq c$, the value of $C[p]$ is equal to its initial value.

These properties in fact describe much of the intended function of the loop and can be used to verify properties of programs manipulating arrays in which this loop is embedded. However, the first four of these invariants cannot be obtained by other methods of invariant generation since, when formulated in first-order logic, they require quantifier alternations.

In this paper we introduce a new method that can be used to derive such loop properties automatically using a first-order theorem prover. For example, all of the invariants given above were automatically generated by the theorem prover Vampire.

3 Preliminaries

In this section, we describe our program model and give a brief introduction into saturation theorem proving.

Array and scalar variables. We assume that programs contain *array variables*, denoted by capital-case letters $A, B, C, \ldots$, and *scalar variables*, denoted by lower-case letters $a, b, c, \ldots$. All notations may have indices. The lower-case letter n will be reserved for *the loop counter*.

Program $\mathcal{P}$. Consider a program $\mathcal{P}$ consisting of a single loop whose body contains assignments, sequencing and conditionals. In the sequel we assume that $\mathcal{P}$ is fixed and give all definitions relative to it. Denote by Var the set of all variables occurring in $\mathcal{P}$, and by Arr the set of all array variables occurring in it.

Expressions. We will use a *language $Expr$ of expressions*. We assume that $Expr$ contains constants (including all integer constants), variables in $Var \cup Arr$, logical variables, some interpreted function symbols, including the standard arithmetical function symbols $+, -, \cdot$, and interpreted predicates symbols, including the standard arithmetical predicate symbols $\geq, \leq$. We assume that expressions are well-typed with respect to a set of sorts and ι is a sort of integers. *Types* are defined as follows: every sort is a type and types can be built from other types using type constructors $\times$ and $\rightarrow$. We assume that each scalar variable has a sort and each array variable has a type $\iota \rightarrow \tau$, where τ is a sort. If A is an array variable and e an expression, we will write $A[e]$ instead of $A(e)$ to mean the element of A at the position e.

Semantics of Expressions. We assume that every sort has an associated non-empty domain and that the domain associated with ι is the set of integers. Furthermore we assume that interpreted function and predicate symbols of the language are interpreted by functions and relations of appropriate sorts. For example, we assume that $\geq$ is interpreted as the standard inequality on integers.

The semantics of the language $Expr$ is defined using the notion of *state*. A state maps each scalar variable of a sort τ into a value in the domain associated with τ, and each array variable A of a type $\iota \rightarrow \tau$ into a function from integers to the domain associated with τ. Note that (for the sake of simplicity) we do not consider arrays as partial functions and do not analyse array bounds. Given a state σ, we can define the value of any expression in this state in the standard way, see e.g. [21].

Semantics of programs. We can define the semantics of programs with assignment, sequencing and conditionals in the standard way, see e.g. [21]. A program of this kind can be considered as a mapping from states to states. A *computation* of a program is a sequence of states.

Extended expressions $v^{(i)}$. Remember that we are dealing with a program $\mathcal{P}$ consisting of a single loop. Suppose that a computation of $\mathcal{P}$ starts at some *initial state* σ_0. If we ignore the loop condition, then after i iterations of the loop the computation will reach a state σ_i. Let us now extend the notion of expression to capture the state σ_i of program execution obtained after i iterations of the main loop. To this end, we first *fix a program $\mathcal{P}$ and some initial state σ_0* so that the definition is parametrised by this initial state and the program. Let σ_i be the state obtained after i iterations of the computation of $\mathcal{P}$ starting at σ_0.

For every integer expression i and loop variable v of a type τ, we define a new expression $v^{(i)}$ of the type τ. The value of this expression is defined to be the value of v at the state σ_i. We say that a formula φ, possibly using extended expressions $v^{(i)}$, is *valid for $\mathcal{P}$*, if this formula is true for every computation of $\mathcal{P}$, that is, for all computations starting at an arbitrary initial state.

Example 1. Consider the loop $\mathcal{P}$ whose body consists of a single assignment $c := c+2$, where c is a scalar variable. Then the formula $(\forall i)(i \geq 0 \implies c^{(i)} = c^{(0)} + 2 \cdot i)$ is valid for $\mathcal{P}$.

Note that $v^{(0)}$ is the value of v in the initial state. We will use expressions $v^{(i)}$ only when we reason about programs or assert their properties. We *will not use these expressions in programs.*

Relativised expressions $i :: e$ and formulas $i :: F$. Given an expression e or a formula F, we would like to "relativise" it to an iteration i. The relativised expression and formula will be denoted by $i :: e$ and $i :: F$, respectively. These expressions are only defined when e and F are non-extended expressions, that is, expressions containing no occurrences of subexpressions of the form $v^{(j)}$ for some v and j.

Definition 1. *For every expression e, formula F having no occurrences of extended expressions $v^{(j)}$ for any v and j, and every integer expression i, let us define an expression $i :: e$ and a formula $i :: F$ by induction as follows. In the definition below e with indices stands for expressions and F with indices stands for formulas.*

1. *If v is a loop (scalar or array) variable, then $i :: v \stackrel{\text{def}}{=} v^{(i)}$.*
2. $i :: (e_1[e_2]) \stackrel{\text{def}}{=} (i :: e_1)[i :: e_2]$.
3. *If e is a constant or a variable (but not an array or a scalar variable) then $i :: e \stackrel{\text{def}}{=} e$.*
4. *If f is an interpreted function, then $i :: (f(e_1, \ldots, e_n)) \stackrel{\text{def}}{=} f(i :: e_1, \ldots, i :: e_n)$.*
5. *If P is a predicate symbol, then $i :: (P(e_1, \ldots, e_n)) \stackrel{\text{def}}{=} P(i :: e_1, \ldots, i :: e_n)$.*
6. $i :: (F_1 \wedge \ldots \wedge F_n) \stackrel{\text{def}}{=} i :: F_1 \wedge \ldots \wedge i :: F_n$ *and similar for other connectives instead of $\wedge$.*
7. *Let y be a variable not occurring in i. Then $i :: ((\forall y)F) \stackrel{\text{def}}{=} (\forall y)(i :: F)$ and similar for $\exists$ instead of $\forall$.*
8. $i :: ((\forall i)F) \stackrel{\text{def}}{=} (\forall i)(F)$ *and similar for $\exists$ instead of $\forall$.*

For example, if F is the formula $(\forall j)(a = 0 \implies A[b] = c + j)$, where A is an array variable and a, b, c are scalar variables, then $i :: F$ is the formula $(\forall j)(a^{(i)} = 0 \implies A^{(i)}[b^{(i)}] = c^{(i)} + j)$.

Loop body and guarded assignments. For simplicity of presentation we assume that the loop body of $\mathcal{P}$ is represented by an equivalent collection of *guarded assignments*. Let us now define guarded assignments and their semantics. We call a *guarded assignment* an expression

$$G \to \alpha_1; \ldots; \alpha_m, \tag{1}$$

where each of the α_j's is an assignment either of the form $v := e$ or of the form $A[e_1] := e_2$, and G is a formula, called the *guard* of this guarded assignment. We assume that each guarded assignment of the form (1) satisfies the following conditions.

1. The left-hand sides of all assignments are syntactically different;
2. If some of the assignments α_j has a form $A[e_1] := e_2$, and some α_k for $k \neq j$ has the form $A[e_3] := e_4$, then in every state satisfying G the expressions e_1 and e_3 have different values.

Furthermore, for every collection of guarded assignments whose guards are $G_1, \ldots, G_p$ we assume that

1. for all $j, k \in \{1, \ldots, p\}$, if $j \neq k$ then the formula $G_j \wedge G_k$ is unsatisfiable (that is, the guards are mutually exclusive);

2. the formula $G_1 \vee \ldots \vee G_p$ is true in all states (that is, for every state at least one of the guards is true in this state).

Let us now define the semantics of collections of guarded assignments satisfying these properties and also briefly discuss how any program can be translated into an equivalent collection of guarded assignments.

Consider a guarded assignment $G \to e_1 \ := \ e'_1; \ldots; e_m \ := \ e'_m$. The sequence of assignments in a guarded assignment has the semantics of a *simultaneous assignment*

$$(e_1, \ldots, e_m) \ := \ (e'_1, \ldots, e'_m).$$

For example, the guarded assignment $\mathtt{true} \to x \ := \ 0; y \ := \ x$ changes any state in which $x = 1$ to a state in which $y = 1$ but not $y = 0$.

One can automatically transform any loop body into an equivalent finite set of guarded assignments [7, 21]. In general, such a transformation may result in a set of guarded assignments of size exponential in the size of the loop body, but one can also avoid exponential size by using a slightly different notion of guarded assignment. To satisfy the condition on the left-hand side of guarded assignments one can add extra equalities and inequalities in the guards. For example, the loop body consisting of the sequence of assignments $A[a] \ := \ 0; A[b] \ := \ 1$ can be transformed into the system consisting of two guarded assignments:

$$\begin{array}{l} a \neq b \to A[a] \ := \ 0; A[b] \ := \ 1 \\ a = b \to A[b] \ := \ 1. \end{array}$$

Let us consider an example.

Example 2 (Partition). Consider the partition program of Figure 1. Then the loop body of this program has the following representation in the guarded assignment form:

$$A[a] \geq 0 \to B[b] \ := A[a]; \ b \ := b + 1; \ a \ := a + 1 \tag{2}$$

$$\neg A[a] \geq 0 \to C[c] \ := A[a]; \ c \ := c + 1; \ a \ := a + 1. \tag{3}$$

General setting. Given a loop $\mathcal{P}$ we would like to generate invariants of this loop, that is, find formulas that are true after n iterations of the loop, where n is an arbitrary non-negative integer. These formulas will express the values of loop variables after n iterations in terms of their initial values, i.e. values of loop variables after 0 iterations. To find these formulas, we will write some general properties of loop variables at an arbitrary iteration between 0 and n, using formulas with extended expressions $v^{(i)}$. In the sequel we assume that n *is an arbitrary but fixed non-negative integer*. We will also use a constant with the same name n in formulas to denote the number n. When we discuss iteration steps, we are only interested in iterations between 0 and $n - 1$. To this end, we introduce a *predicate* $iter$ denoting such iterations. To improve readability, we will normally write $e \in iter$ instead of $iter(e)$, where e is an expression. The predicate $iter$ has the following definition:

$$(\forall i)(i \in iter \iff 0 \leq i \wedge i < n). \tag{4}$$

Saturation theorem proving. In our approach to invariant generation, we rely on a saturation prover to infer automatically first-order formulas with equality as quantified invariants from a set of first-order loop properties extracted from loops. We shortly describe the basic ideas of saturation theorem proving, and refer to [24] for more details.

First-order theorem provers using saturation algorithms employ a *superposition calculus*, see e.g. [24]. This calculus works with *clauses* (disjunctions of atomic formulas and their negations) and consists of inference rules that allow one to derive new clauses from existing clauses. To prove a formula F, saturation-based provers convert $\neg F$ to a set of clauses and try to derive the empty clause from this set. If the empty clause is derived, then $\neg F$ is unsatisfiable and so F is a theorem. In saturation-based provers the newly derived clauses are normally consequences of the initial clauses. We use this property to derive invariants instead of establishing unsatisfiability: starting with the set of initial clauses, we derive new clauses from it using a superposition calculus and special kinds of reduction orderings and check if some of the newly derived clauses can be used as invariants.

4 Update predicates

To make a saturation-based theorem prover find loop invariants we have to extract some properties of the loop and give them to the prover as initial formulas. Our technique for doing this is based on the analysis of updates to arrays. To analyse updates we introduce so-called *update predicates* and some axioms about these predicates. There are also other formulas we extract automatically from the loop description, they are described in the next section.

For each array variable V that is updated in the program we introduce two predicates:

1. $upd_V(i, p)$: at the loop iteration i the array V is updated at the position p;
2. $upd_V(i, p, v)$: at the loop iteration i the array V is updated at the position p by the value v.

The definition of these update predicates can be extracted automatically from the collection of guarded assignments associated with the loop. For example, guarded assignments (2) and (3) result in the following update predicates for B:

$$upd_B(i, p) \iff i \in iter \wedge p = b^{(i)} \wedge A^{(i)}[a^{(i)}] \geq 0; \tag{5}$$

$$upd_B(i, p, v) \iff i \in iter \wedge p = b^{(i)} \wedge A^{(i)}[a^{(i)}] \geq 0 \wedge v = A[a^{(i)}]. \tag{6}$$

We introduce these update predicates to express the following key properties of array updates:

1. if an array V is never updated at an index p then the final value of $V[p]$ is constant;
2. if an array V is updated at an index p at an iteration i and not updated at any further iteration, then $V[p]$ receives its final value at the iteration i.

These two properties do not depend on the loop. For the array B they are formally expressed as follows.

$$(\forall i)\neg upd_B(i,p) \implies B^{(n)}[p] = B^{(0)}[p]; \tag{7}$$

$$upd_B(i,p,v) \wedge (\forall j > i)\neg upd_B(j,p) \implies B^{(n)}[p] = v. \tag{8}$$

We will refer to these two properties as the *stability property* and the *last update property* for B, respectively.

5 Extracting Loop Properties

In this section we will describe some properties that can be automatically extracted from the loop. Given the loop body, we add all these properties as additional axioms to the theorem prover `Vampire` to help it generate loop invariants.

Constant array. If we have an array A that is never updated in the loop, we can add an axiom $(\forall i)(A^{(i)} = A^{(0)})$. A simpler approach (and the one we adopt here) is to treat such an array A as a constant and simply use A instead of $A^{(i)}$. In our example, A is such an array so we will simply write $A[p]$ instead of $A^{(i)}[p]$.

Monotonicity properties. Let us call a scalar variable v *increasing* if it has the property $(\forall i \in iter)(v^{(i+1)} \geq v^{(i)})$ for all possible computations of the loop. Likewise, a variable is called *decreasing* if it has the property $(\forall i \in iter)(v^{(i+1)} \leq v^{(i)})$ for all possible computations of the loop. A *monotonic variable* is a variable that is either increasing or decreasing.

The monotonicity properties can be discovered either by program analysis tools or by some light-weight analysis. For example, if all assignments to a variable v in the loop have the form $v = v + c$ where c is a non-negative integer constant, then v is obviously increasing. In our example, the variables a, b and c can be identified as increasing using such light-weight analysis.

We can introduce a more fine-grained classification of monotonic variables. A variable v is called *strictly increasing* if it has the property $(\forall i \in iter)(v^{(i+1)} > v^{(i)})$. *Strictly decreasing variables* are defined similarly. In our example the variable a is strictly increasing.

Let us call an increasing integer variable v *dense* if it has the property

$$(\forall i \in iter)(v^{(i+1)} = v^{(i)} \vee v^{(i+1)} = v^{(i)} + 1)$$

for all possible computations of the loop, and similarly for decreasing variables. In our example, the variables a, b, c are all dense.

Let us now formulate properties that we extract from loops automatically for various kinds of monotonic variable. We will only formulate them for increasing variables, leaving the case of decreasing variables to the reader.

1. If a variable v is strictly increasing and dense, then we add the following property:

   $$(\forall i)(v^{(i)} = v^{(0)} + i).$$

   Note that we do not restrict i in this formula to range over iterations only, as well as we did so in formulas (7) and (8): one can prove that our approach is still sound if we use these more general formulas.

2. If a variable v is strictly increasing but not dense, then we add the following property:

$$(\forall j)(\forall k)(k > j \implies v^{(k)} > v^{(j)}).$$

3. If a variable v is increasing but not strictly increasing, then we add the following property:

$$(\forall j)(\forall k)(k \geq j \implies v^{(k)} \geq v^{(j)}).$$

4. If a variable v is increasing and dense but not strictly increasing, then we add the following property:

$$(\forall j)(\forall k)(k \geq j \implies v^{(j)} + k \geq v^{(k)} + j).$$

Note that, under the monotonicity and density assumptions stated above, the above formula follows from the property $(\forall j)(\forall k)(v^{(k)} \leq v^{(j)} + k - j)$.

In our example the following properties of the monotonic variables a, b, c will be added:

$$\begin{array}{l}
(\forall i)(a^{(i)} = a^{(0)} + i). \\
(\forall j)(\forall k)(k \geq j \implies b^{(k)} \geq b^{(j)}). \\
(\forall j)(\forall k)(k \geq j \implies c^{(k)} \geq c^{(j)}). \\
(\forall j)(\forall k)(k \geq j \implies b^{(j)} + k \geq b^{(k)} + j). \\
(\forall j)(\forall k)(k \geq j \implies c^{(j)} + k \geq c^{(k)} + j).
\end{array} \tag{9}$$

To describe the other properties extracted from loops we will assume that the loop has the following presentation by guarded assignments:

$$\begin{array}{rcl}
G_1 & \to & \alpha_1, \\
\cdots & & \\
G_m & \to & \alpha_m.
\end{array} \tag{10}$$

Update properties of monotonic variables. Suppose that x is a monotonic variable. Intuitively, an update property for this variable expresses that, if the variable changes its value, then there exists a program point at which conditions for this change have been enabled. As before, we will only formulate these properties for increasing variables.

Suppose that x is increasing. Further, assume that $U \subseteq \{1, \ldots, m\}$ is the set of guarded assignments that may update the value of x, that is, $u \in U$ if and only if α_u contains an assignment to x. Then, if x is dense, we add the following property:

$$(\forall v)(v \geq x^{(0)} \wedge x^{(n)} > v \implies (\exists i \in iter)(\bigvee_{u \in U} (i :: G_u) \wedge x^{(i)} = v).$$

If x is not dense, then the property is slightly more complex:

$$(\forall v)(v \geq x^{(0)} \wedge x^{(n)} > v \implies (\exists i \in iter)(\bigvee_{u \in U} (i :: G_u) \wedge v \geq x^{(i)} \wedge x^{(i+1)} > v).$$

For our example the following two axioms will be added:

$$(\forall v)(v \geq b^{(0)} \wedge b^{(n)} > v \implies (\exists i \in iter)(b^{(i)} = v \wedge A[a^{(i)}] \geq 0)); \quad (11)$$
$$(\forall v)(v \geq c^{(0)} \wedge c^{(n)} > v \implies (\exists i \in iter)(c^{(i)} = v \wedge \neg A[a^{(i)}] \geq 0)).$$

Translation of guarded assignments. Suppose that $G \rightarrow e_1 \ := \ e'_1; \ldots; e_k \ := \ e'_k$ is a guarded assignment in the loop representation and $v_1, \ldots, v_l$ are all scalar variables of the loop not belonging to $\{e_1, \ldots, e_k\}$. Define the *translation* $t(e_j)$ *at iteration* i of a left-hand side of an assignment as follows: for a scalar variable x, we have $t(x) \stackrel{\text{def}}{=} x^{(i+1)}$, and for an array variable X and expression e we have $t(X[e]) \stackrel{\text{def}}{=} X^{(i+1)}[e^{(i)}]$. Then we add the following axiom:

$$(\forall i \in iter)(i :: G \implies \bigwedge_{j=1,\ldots,k} t(e_j) = (i :: e'_j) \wedge \bigwedge_{j=1,\ldots,l} v_j^{(i+1)} = v_j^{(i)}).$$

For our running example, we add the following two formulas:

$$\begin{aligned}(\forall i \in iter)(\ A[a^{(i)}] \geq 0 \implies\ & B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge \\ & b^{(i+1)} = b^{(i)} + 1 \wedge \\ & c^{(i+1)} = c^{(i)}\ ); \\ (\forall i \in iter)(\ \neg A[a^{(i)}] \geq 0 \implies\ & C^{(i+1)}[c^{(i)}] = A[a^{(i)}] \wedge \\ & c^{(i+1)} = c^{(i)} + 1 \wedge \\ & b^{(i+1)} = b^{(i)}\ ).\end{aligned} \quad (12)$$

6 Invariant Generation

Our method of invariant generation works as follows.

1. Given a loop, create its representation by a collection of guarded assignments.
2. Generate loop invariants over scalars using `Aligator`. Note, that any other static analysis tool, e.g. [6, 23], can be also used.
3. Extract, using the techniques of Sections 4 and 5, first-order properties of the loop in the logic using expressions $v^{(i)}$. Note that these first-order properties use auxiliary function and predicate symbols that cannot occur in the invariants.
4. Eliminate auxiliary function and predicate symbols by running the saturation theorem prover `Vampire` on the collection of first-order properties of the loop obtained in steps 2 and 3, and finding consequences not using these symbols.

The rest of this section discusses how one can eliminate auxiliary symbols and generate invariants using `Vampire`.

Modern resolution theorem provers [25–27] lack several features essential for implementing our procedure for invariant generation. These are

1. reasoning with linear integer arithmetic;

2. procedures for eliminating symbols.

The first problem is very hard (see, e.g. [17] for some results on combining first-order superposition provers and arithmetic). However, one can provide a sound but incomplete axiomatisation of linear integer arithmetic that is sufficient for proving many essential properties of integers. In our experiments we used the following very simple axiomatisation of the arithmetical relations $>$ and $\geq$, and the successor function s (in our examples we substituted $s(e)$ instead of expressions $e + 1$):

$$
\begin{array}{l}
x \geq y \iff x > y \vee x = y; \\
x > y \implies x \neq y; \\
x \geq y \wedge y \geq z \implies x \geq z; \\
s(x) > x; \\
x \geq s(y) \iff x > y.
\end{array}
$$

To solve the second problem (changing a theorem prover to handle symbol elimination) we used the following idea. For every array and scalar variable v that occurs on the left-hand side of an assignment we introduce two new symbols v_0 and v' together with the following axioms: $v^{(0)} = v_0$ and $v^{(n)} = v'$. We call these new symbols *target symbols*. Let us call a clause *useful* if it satisfies the following conditions (by a symbol below we mean a signature symbol, that is, a non-variable).

1. Every symbol in this clause is either a target symbol, or an interpreted symbol or a skolem function introduced by `Vampire`. We call such symbols *usable* and all other symbols *useless*.
2. The clause contains at least one target symbol or a skolem function.

We are interested in deriving only useful clauses. Indeed, all other clauses either contain symbols, such as update predicates, that cannot occur in invariants and so should be eliminated, or represent valid arithmetical properties and so are irrelevant to the loop.

To this end, we make `Vampire` use a reduction ordering that makes all useless symbols large in precedence and having a large weight in the Knuth-Bendix ordering used by `Vampire`[3]. We also make `Vampire` output all generated useful clauses.

If we derive a useful clause containing no skolem functions, then this clause denotes an invariant of the loop for all initial states satisfying the condition $v = v_0$ for all loop variables, after replacing all variables v' by v. For example, from the properties presented in Sections 4 and 5, `Vampire` derived the following useful clause:

$$\neg x \geq b' \vee B'[x] = B_0[x],$$

which denotes the invariant $\neg x \geq b \vee B[x] = B_0[x]$ and can also be written as

$$(\forall x)(x \geq b \implies B[x] = B_0[x]).$$

If a clause with skolem functions is derived, we can de-skolemise this clause by introducing existential quantifiers. For example, `Vampire` derived the clause

[3] Essentially, resolution theorem provers prefer to apply inferences with atoms containing large and heavy symbols and thus eventually remove these atoms from clauses.

$$\neg b' > x \vee \neg x \geq 0 \vee A[\$i(x)] = B'[x], \tag{13}$$

where $\$i$ is a skolem function. This clause can be de-skolemised into a *quantified invariant*

$$(\forall x)(b > x \wedge x \geq 0 \implies (\exists y)A[y] = B[x]). \tag{14}$$

However, there are reasons to use clauses with skolem functions directly rather then de-skolemise them. Consider, for example, the following formula derived by `Vampire` for our running example.

$$\neg b' > x \vee \neg x \geq 0 \vee A[\$i(x)] \geq 0. \tag{15}$$

It can be de-skolemised into the invariant

$$(\forall x)(b > x \wedge x \geq 0 \implies (\exists y)A[y] \geq 0). \tag{16}$$

The problem is that (13) and (15) imply the following invariant:

$$(\forall x)(b > x \wedge x \geq 0 \implies (\exists y)(A[y] = B[x] \wedge A[y] \geq 0)),$$

which is not implied by their de-skolemised forms (14) and (16).

7 Experiments with `Vampire`

We made experiments with invariant generation for our running example and also for an example of [22] where an array is filled with 0's at positions from 0 to $n-1$. In [22] it took 0.01 seconds to generate an invariant for proving the assertion that all elements of the array are zeros. `Vampire` *derived* this property as a loop invariant in less than 0.01 seconds: more precisely, it derived that all array elements up to the loop counter n are zeros.

For our running example, among all generated invariants we were interested in finding out how fast `Vampire` can derive the following two properties:

1. Array B does not change at positions greater than or equal to the final value of b, that is

   $$\forall p(p \geq b' \implies B'[p] = B_0[p]).$$

   The corresponding clause was generated in 0.73 seconds.
2. Every value in $\{B[0], \ldots, B[b-1]\}$ is a non-negative value in $\{A[0], \ldots, A[a-1]\}$:

   $$\forall p(b' > p \wedge p \geq 0 \implies B'[p] \geq 0 \wedge \exists k(a' > k \wedge k \geq 0 \wedge A[k] = B'[p]).$$

   There are four clauses the conjunction of which imply this formula and which were derived by `Vampire`, one of them is (13). The derivation was found in about 53 seconds.

8 Related Work

Recently, the problem of automatically generating quantified invariant properties for loops with arrays received a considerable attention [9, 4, 20, 16, 2, 12, 11]. Based on the abstract interpretation framework [5], the approaches described in [4, 9, 11, 12, 2] use a set of a priori defined atomic predicates over program variables, from which universally quantified array properties are then inferred. Paper [9] iteratively approximates the strongest boolean combination of a given set of suitable predicates for the loop, until a *fixpoint*, i.e. an invariant, is reached. The approach is based on predicate abstraction with skolem constants for the quantified variables, and implements heuristics for guessing some of the appropriate predicates used further for invariant generation. Iterative computation of invariant predicates is also used in [20]. In [11] a priori fixed templates describing candidate invariant properties are used to generate quantified invariants by *under-approximation* algorithms of logical boolean operators for building abstract interpreters over quantified abstract domains. However, these approaches require a given set of predicates from which invariants can be built; some of them also require user guidance.

Using the combined theory of linear arithmetic and uninterpreted function, [2] presents a *constraint-based* invariant synthesis. The method relies on user-given invariant templates over program variables. Constraints on the unknown parameters of the template invariants are generated based on the inductiveness property of an invariant assertion. Solutions to these constraints are substituted for parameters in the template to derive (universally quantified) invariants. Using counterexample guided abstraction, the method is further extended in [3] to the generation of *path invariants*. A counterexample guided abstraction refinement method is presented also in [16], where *range predicates* are used to characterize properties of array segments between specified bounds. Array invariants are then inferred from the predefined range predicates by interpolation-based techniques. The appropriate range predicates are however supplied manually.

A fundamental difference of our approach compared to these works is that we do not require user-defined templates or a fixed collection of predicates. Our invariants can be arbitrary assertions inferred by a theorem prover from assertions over variables obtained by recurrence solving and quantifier elimination methods and by a light-weight analysis of monotonic variables of loops. The advantage of using general recurrence solving methods together with quantifier elimination is also confirmed by comparing our framework to [19] where loop invariants are inferred by providing predefined solutions for a special subclass of recurrences over scalar variables. Moreover, unlike our approach, the above mentioned methods do not infer automatically polynomial/linear relations among scalar variables as invariants.

Based on the abstract interpretation framework, [10, 13] infer universally quantified array invariants. Their approach requires *no user guidance*. The key idea is to partition values used as array indexes into symbolic intervals and use abstract interpretation. Paper [10] infer invariants of a special form essentially involving a single array index, such as $(\forall i \leq n)(A[i] > 0)$. Paper [13] goes further and, using more sophisticated analysis, derives invariants that may involve several arrays in which indexes are obtained from each other by using a "shift" by an expression. An example of such an invariant is $(\forall i \leq n)(A[i] = B[i + e])$, where e is an expression in which i does not occur. These

papers do not derive properties with quantifier alternations but [13] treats nested loops. It seems that we can benefit from integrating the approach of [10, 13] into ours, both by deriving properties of a single loop iteration and by using their invariants as additional formulas in a theorem prover.

In [22], based on an earlier work [15], a saturation theorem prover together with elimination of symbols is used for generating interpolants and proving loop properties over arrays. Although our approach has much in common with that of [22], there are essential differences. First, we do not require to have a loop property for generating invariants so our approach can be useful for generating properties of loops embedded into large programs. Second, we support richer arithmetic reasoning by using symbolic computation methods. Third, we are able to generate invariants also containing quantifier alternations. Finally, [15, 22] require some form of guidance by providing a growing sequence of sets of atoms from which invariants can be built and the efficiency of their analysis may crucially depend on the choice of such a sequence.

9 Conclusion

We showed how quantified loop invariants of programs over arrays can be automatically inferred using a first order theorem prover, reducing the burden of annotating loops with complete invariants. For doing so, we deploy symbolic computation methods to generate numeric invariants of the scalar loop variables and then use update predicates of the loop. Using this information quantified array invariants, including those with alternating quantifiers, are derived with the help of a saturation prover. In particular, our method does not require the user to give a post-condition, a predefined collection of predicates or any other form of human guidance and avoids inductive reasoning. Our initial experimental results on some benchmark examples demonstrate the potential of our method. Modifications of theorem provers are required to carry out large-scale experiments with our method.

Our work was partially inspired by an analysis of loops with arrays occurring in very large programs performed by Thibaud Hottelier, Andrey Rybalchenko and the first author (personal communication): it turned out that many uses of arrays involve either array initialisation, or array copying, either to another array or to itself, or simple iterations over array elements. In other words, typical loops for programs with arrays are not much more complex than the loop of our running example. This made us believe that analysis of counters and other monotonic variables in such loops may provide enough information to generate complex invariants.

Future work. To make our technique widely applicable one needs to extend first-order theorem provers by symbol elimination and generating various classes of clause sets with eliminated symbols: for example, minimal sets so that clauses in this set do not imply each other. Minimality is, obviously, undecidable, so we can instead use some light-weight removal of clauses implied by other clauses.

[22] formulates some results related to symbol elimination in resolution theorem proving. In general, it is interesting to develop a theory for symbol elimination and consequence finding, which is not well-understood in presence of equality.

It is possible that similar techniques can be successfully applied to programs with pointers. To this end one should find out which properties of loops should be extracted

automatically to derive interesting invariants for such programs. Another interesting extension would be programs with nested loops: we believe many of them can be handled using the same techniques.

It is also interesting to see how our method can be used for proving loop properties rather than generating them. To this end one can try to embed it into systems for proving program properties, such as [8, 1, 14].

We also believe that more complex kinds of loop analysis followed by theorem proving would be able to discover non-trivial invariants of logically much more complex loops, such as implementations of quick-sort and union-find algorithms.

We did not treat nested loops or multi-dimensional arrays due to a lack of space, though they can be treated in a similar way, by using two loop counters and presenting arrays as functions of more than one argument and modifying update predicates and their automatically generated properties. One needs extensive experiments to understand the efficiency of the method for these extensions too. We are going to make such experiments after modifying Vampire.

Acknowledgments. We thank Andrey Rybalchenko for motivating discussions, Thibaud Hottelier whose work provided parts of the infrastructure used by us, Rustan Leino whose numerous remarks and careful proofreading helped us to improve the paper considerably, and Sumit Gulwani and Shaz Qadeer for discussions on our method and related work.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ Programming System: An Overview. In *Proc. of CASSIS*, volume 3362 of *LNCS*, 2004.
2. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In *Proc. of VMCAI*, volume 4349 of *LNCS*, 2007.
3. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *Proc. of PLDI*, 2007.
4. P. Cousot. Verification by Abstract Interpretation. In *Proc. of Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, volume 2772, pages 243–268. Springer, 2003.
5. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL*, pages 238–252, 1977.
6. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. of POPL*, pages 84–96, 1978.
7. E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. of PLDI*, 2002.
9. C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proc. of POPL*, pages 191–202, 2002.
10. D. Gopan, T. W. Reps, and M. Sagiv. A Framework for Numeric Analysis of Array Operations . In *POPL*, pages 338–350, 2005.
11. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *Proc. of POPL*, pages 235–246, 2008.

12. S. Gulwani and A. Tiwari. An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 379–392, 2007.
13. N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.
14. T. A. Henzinger, T. Hottelier, and L. Kovacs. Valigator: A Verification Tool with Bound and Invariant Generation. In *Proc. of LPAR*, pages 333–342, 2008.
15. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Proc. of TACAS*, pages 459–473, 2006.
16. R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 193–206, 2007.
17. K. Korovin and A. Voronkov. Integrating Linear Arithmetic into Superposition Calculus. In *Proc. of CSL*, volume 4646 of *LNCS*, pages 223–237, 2007.
18. L. Kovacs. Reasoning Algebraically About P-Solvable Loops. In *TACAS*, volume 4963 of *LNCS*, pages 249–264, 2008.
19. D. Kroening and G. Weissenbacher. Counterexamples with Loops for Predicate Abstraction. In *Proc. of CAV*, volume 4144 of *LNCS*, pages 152–165, 2006.
20. S. K. Lahiri and R. E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In *Proc. of CAV*, volume 3114 of *LNCS*, pages 135–147, 2004.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
22. K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.
23. A. Miné. The Octagon Abstract Domain. In *Proc. of WCRE*, pages 310–319, 2001.
24. R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 7, pages 371–443. Elsevier Science, Amsterdam, 2001.
25. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
26. S. Schulz. System Description: E 0.81. In *Proc. of IJCAR*, volume 3097 of *LNAI*, pages 223–228, 2004.
27. C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System Description: SpassVersion 3.0. In *Proc. of CADE*, volume 4603 of *LNAI*, pages 514–520, 2007.