Automatic Verification of Integer Array Programs*

Marius Bozga[1], Peter Habermehl[2], Radu Iosif[1], Filip Konečný[1,3], and Tomáš Vojnar[3]

[1] VERIMAG, CNRS, 2 av. de Vignate, 38610 Gières, France, {bozga,iosif}@imag.fr
[2] LIAFA, University Paris 7, Case 7014, 75205 Paris 13, France, haberm@liafa.jussieu.fr
[3] FIT BUT, Božetěchova 2, 61266, Brno, Czech Republic, {ikonecny,vojnar}@fit.vutbr.cz

Abstract. We provide a verification technique for a class of programs working on *integer arrays* of finite, but not a priori bounded length. We use the logic of integer arrays **SIL** [13] to specify pre- and post-conditions of programs and their parts. Effects of non-looping parts of code are computed syntactically on the level of **SIL**. Loop pre-conditions derived during the computation in **SIL** are converted into counter automata (CA). Loops are automatically translated—purely on the syntactical level—to transducers. Pre-condition CA and transducers are composed, and the composition over-approximated by flat automata with difference bound constraints, which are next converted back into **SIL** formulae, thus inferring post-conditions of the loops. Finally, validity of post-conditions specified by the user in **SIL** may be checked as entailment is decidable for **SIL**.

1 Introduction

Arrays are an important data structure in all common programming languages. Automatic verification of programs using arrays is a difficult task since they are of a finite, but often not a priori fixed length, and, moreover, their contents may be unbounded too. Nevertheless, various approaches for automatic verification of programs with arrays have recently been proposed.

In this paper, we consider programs over integer arrays with assignments, conditional statements, and *non-nested* while loops. Our verification technique is based on a combination of the logic of integer arrays **SIL** [13], used for expressing pre-/post-conditions of programs and their parts, and *counter automata* (CA) and *transducers*, into which we translate both **SIL** formulae and program loops in order to be able to compute the effect of loops and to be able to check entailment.

SIL (Single Index Logic) allows one to describe properties over arrays of integers and scalar variables. **SIL** uses difference bound constraints to compare array elements situated within a window of a constant size. For instance, the formula $(\forall i.0 \leq i \leq n_1 - 1 \rightarrow b[i] \geq 0) \ \wedge \ (\forall i.0 \leq i \leq n_2 - 1 \rightarrow c[i] < 0)$ describes a post-condition of a program partitioning an array a into an array b containing its positive elements and an array c containing its negative elements. **SIL** formulae are interpreted over program *states* assigning integers to scalar variables and finite sequences of integers to array variables. As already proved in [13], the set of models of an $\exists^*\forall^*$-**SIL** formula corresponds naturally to the set of traces of a *flat* CA with loops labelled by difference bound constraints. This entails decidability of the satisfiability problem for $\exists^*\forall^*$-**SIL**.

In this paper we take a novel perspective on the connection between $\exists^*\forall^*$-**SIL** and CA, allowing to benefit from the advantages of both formalisms. Indeed, the logic is useful to express human-readable pre-/post-conditions of programs and their parts, and

* This work was supported by the French project RNTL AVERILES, the Czech Science Foundation (projects 102/07/0322, 102/09/H042), the Barrande project MEB 020840, and the Czech Ministry of Education by the project MSM 0021630528.

to compute the post-image of (non-looping) program statements symbolically. On the other hand, automata are suitable for expressing the effects of program loops.

In particular, given an $\exists^*\forall^*$-**SIL** formula, we can easily compute the strongest post-condition of an assignment or a conditional statement in the same fragment of the logic. Upon reaching a program loop, we then translate the $\exists^*\forall^*$-**SIL** formula φ describing the set of states at the beginning of the loop into a CA A_φ encoding its set of models. Next, to characterise the effect of a loop L, we translate it—purely syntactically—into a *transducer* T_L, i.e., a CA describing the input/output relation on scalars and array elements implemented by L. The post-condition of L is then obtained by composing T_L with A_φ. The result of the composition is a CA $B_{\varphi,L}$ representing the *exact* set of states after *any number* of iterations of L. Finally, we translate $B_{\varphi,L}$ back into $\exists^*\forall^*$-**SIL**, obtaining a post-condition of L w.r.t. φ. However, due to the fact that counter automata are more expressive than $\exists^*\forall^*$-**SIL**, this final step involves a (refinable) *abstraction*. We first generate a *flat* CA that over-approximates the set of traces of $B_{\varphi,L}$, and then translate the flat CA back into $\exists^*\forall^*$-**SIL**.

Our approach thus generates automatically a *human-readable post-condition* for each program loop, giving the end-user some insight of what the program is doing. Moreover, as these post-conditions are expressed in a decidable logic, they can be used to check entailment of user-specified post-conditions given in the same logic.

We validate our approach by successfully and fully algorithmically verifying several array-manipulating programs, like splitting of an array into positive and negative elements, rotating an array, inserting into a sorted array, etc. Some of the steps were done manually as we have not yet implemented all of the techniques—a full implementation that will allow us to do more examples is underway.

Due to space reasons, we skip below some details of the techniques and their proofs, which are deferred to [4].

Related Work. The area of automated verification of programs with arrays and/or synthesising loop invariants for such programs has recently received a lot of attention. For instance, [8, 18, 1, 2, 16, 12] build on templates of universally quantified loop invariants and/or atomic predicates provided by the user. The form of the sought invariants is then based on these templates. Inferring the invariants is tackled by various approaches, such as predicate abstraction using predicates with Skolem constants [8], constraint-based invariant synthesis [1, 2], or predicate abstraction combined with interpolation-based refinement [16].

In [20], an interpolating saturation prover is used for deriving invariants from finite unfoldings of loops. In the very recent work of [17], loop invariants are synthesised by first deriving scalar invariants, combining them with various predefined first-order array axioms, and finally using a saturation prover for generating the loop invariants on arrays. This approach can generate invariants containing quantifier alternation. A disadvantage is that, unlike our approach, the method does not take into account loop preconditions, which are sometimes necessary to find reasonable invariants. Also, the method does not generate invariants in a decidable logical fragment, in general.

Another approach, based on abstract interpretation, was used in [11]. Here, arrays are suitably partitioned, and summary properties of the array segments are tracked. The partitioning is based on heuristics related to tracking the position of index variables. These heuristics, however, sometimes fail, and human guidance is needed. The approach was recently improved in [15] by using better partitioning heuristics and relational abstract domains to keep track of the relations of the particular array slices.

Recently, several works have proposed decidable logics capable of expressing complex properties of arrays [6, 21, 9, 3, 10]. In general, these logics lack the capability of universally relating two successive elements of arrays, which is allowed in our previous work [14, 13]. Moreover, the logics of [6, 21, 9, 3, 10] do not give direct means of automatically dealing with program loops, and hence, verifying programs with arrays. In this work, we provide a fully algorithmic verification technique that uses the decidable logic of [13]. Unlike many other works, we do not synthesise loop invariants, but directly post-conditions of loops with respect to given preconditions, using a two-way automata-logic connection that we establish.

2 Preliminaries

For a set A, we denote by A^* the set of finite sequences of elements from A. For such a sequence $\sigma \in A^*$, we denote by $|\sigma|$ its length, and by σ_i the element at position i, for $0 \leq i < |\sigma|$. We denote by $\mathbb{N}$ the set of natural numbers, and by $\mathbb{Z}$ the set of integers. For a function $f : A \to B$ and a set $S \subseteq A$, we denote by $f\downarrow_S$ the restriction of f to S. This notation is naturally lifted to sets, pairs or sequences of functions.

Given a formula φ, we denote by $FV(\varphi)$ the set of its free variables. If we denote a formula as $\varphi(x_1,...,x_n)$, we assume $FV(\varphi) \subseteq \{x_1,...,x_n\}$. For $\varphi(x_1,\ldots,x_n)$, we denote by $\varphi[t_1/x_1,\ldots,t_n/x_n]$ the formula which is obtained from φ by replacing each free occurrence of $x_1,\ldots,x_n$ by the terms $t_1,\ldots,t_n$, respectively. Moreover, we denote by $\varphi[t/x_1,\ldots,x_n]$ the formula that arises from φ when all free occurrences of all the variables $x_1,\ldots,x_n$ are replaced by the same term t. Given a formula φ and a valuation ν of its free variables, we write $\nu \models \varphi$ if by replacing each free variable x of φ with $\nu(x)$ we obtain a valid formula. By $\models \varphi$ we denote the fact that φ is valid.

A *difference bound constraint* (DBC) is a conjunction of inequalities of the forms (1) $x - y \leq c$, (2) $x \leq c$, or (3) $x \geq c$, where $c \in \mathbb{Z}$ is a constant. We denote by $\top$ (true) the empty DBC.

A *counter automaton* (CA) is a tuple $A = \langle X, Q, I, \to, F \rangle$, where: X is a finite set of counters ranging over $\mathbb{Z}$, Q is a finite set of control states, $I \subseteq Q$ is a set of initial states, $\to$ is a transition relation given by a set of rules $q \xrightarrow{\varphi(X,X')} q'$ where φ is an arithmetic formula relating current values of counters X to their future values $X' = \{x' \mid x \in X\}$, and $F \subseteq Q$ is a set of final states.

A *configuration* of a CA A is a pair $\langle q, \nu \rangle$ where $q \in Q$ is a control state, and $\nu : X \to \mathbb{Z}$ is a valuation of the counters in X. For a configuration $c = \langle q, \nu \rangle$, we designate by $val(c) = \nu$ the valuation of the counters in c. A configuration $\langle q', \nu' \rangle$ is an *immediate successor* of $\langle q, \nu \rangle$ if and only if A has a transition rule $q \xrightarrow{\varphi(X,X')} q'$ such that $\nu \cup \nu' \models \varphi$. Given two control states $q, q' \in Q$, a run of A from q to q' is a finite sequence of configurations $c_1 c_2 \ldots c_n$ with $c_1 = \langle q, \nu \rangle$, $c_n = \langle q', \nu' \rangle$ for some valuations $\nu, \nu' : X \to \mathbb{Z}$, and c_{i+1} is an immediate successor of c_i, for all $1 \leq i < n$. Let $\mathcal{R}(A)$ denote the set of runs of A from some initial state $q_0 \in I$ to some final state $q_f \in F$, and $Tr(A) = \{val(c_1)val(c_2)\ldots val(c_n) \mid c_1 c_2 \ldots c_n \in \mathcal{R}(A)\}$ be its set of *traces*.

For two counter automata $A_i = \langle X_i, Q_i, I_i, \rightarrow_i, F_i \rangle,\ i = 1,2$ we define the *product automaton* as $A_1 \otimes A_2 = \langle X_1 \cup X_2, Q_1 \times Q_2, I_1 \times I_2, \rightarrow, F_1 \times F_2 \rangle$, where $\langle q_1, q_2 \rangle \xrightarrow{\varphi} \langle q_1', q_2' \rangle$ if and only if $q_1 \xrightarrow{\varphi_1}_1 q_1'$, $q_2 \xrightarrow{\varphi_2}_2 q_2'$ and $\models \varphi \leftrightarrow \varphi_1 \wedge \varphi_2$. We have that, for all sequences $\sigma \in Tr(A_1 \otimes A_2)$, $\sigma\downarrow_{X_1} \in Tr(A_1)$ and $\sigma\downarrow_{X_2} \in Tr(A_2)$, and vice versa.

3 Counter Automata as Recognisers of States and Transitions

In the rest of this section, let $\mathbf{a} = \{a_1, a_2, \ldots, a_k\}$ be a set of *array variables*, and $\mathbf{b} = \{b_1, b_2, \ldots, b_m\}$ be a set of *scalar variables*. A *state* $\langle \alpha, \iota \rangle$ is a pair of valuations $\alpha : \mathbf{a} \rightarrow \mathbb{Z}^*$, and $\iota : \mathbf{b} \rightarrow \mathbb{Z}$. For simplicity, we assume that $|\alpha(a_1)| = |\alpha(a_2)| = \ldots = |\alpha(a_k)| > 0$, and denote by $|\alpha|$ the size of the arrays in the state.

In the following, let X be a set of counters that is partitioned into *value counters* $\mathbf{x} = \{x_1, x_2, \ldots, x_k\}$, *index counters* $\mathbf{i} = \{i_1, i_2, \ldots, i_k\}$, *parameters* $\mathbf{p} = \{p_1, p_2, \ldots, p_m\}$, and *working counters* $\mathbf{w}$. Notice that $\mathbf{a}$ is in a 1:1 correspondence with both $\mathbf{x}$ and $\mathbf{i}$, and that $\mathbf{b}$ is in a 1:1 correspondence with $\mathbf{p}$.

Definition 1. *Let $\langle \alpha, \iota \rangle$ be a state. A sequence $\sigma \in (X \rightarrow \mathbb{Z})^*$ is said to be* consistent *with $\langle \alpha, \iota \rangle$, denoted $\sigma \vdash \langle \alpha, \iota \rangle$ if and only if, for all $1 \leq p \leq k$, and all $1 \leq r \leq m$:*

1. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, we have $0 \leq \sigma_q(i_p) \leq |\alpha|$,*
2. *for all $q, r \in \mathbb{N}$ with $0 \leq q < r < |\sigma|$, we have $\sigma_q(i_p) \leq \sigma_r(i_p)$,*
3. *for all $s \in \mathbb{N}$ with $0 \leq s \leq |\alpha|$, there exists $0 \leq q < |\sigma|$ such that $\sigma_q(i_p) = s$,*
4. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, if $\sigma_q(i_p) = s < |\alpha|$, then $\sigma_q(x_p) = \alpha(a_p)_s$,*
5. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, we have $\sigma_q(p_r) = \iota(b_r)$.*

Intuitively, a run of a CA represents the contents of a single array by traversing all of its entries in one move from the left to the right. The contents of multiple arrays is represented by arbitrarily interleaving the traversals of the different arrays. From this point of view, for a run to correspond to some state (i.e., to be *consistent* with it), it must be the case that each index counter either keeps its value or grows at each step of the run (point 2 of Def. 1) while visiting each entry within the array (points 1 and 3 of Def. 1).[4] The value of a certain entry of an array a_p is coded by the value that the array counter x_p has when the index counter i_p contains the position of the given entry (point 4 of Def. 1). Finally, values of scalar variables are encoded by values of the appropriate parameter counters which stay constant within a run (point 5 of Def. 1).

A CA is said to be *state consistent* if and only if for every trace $\sigma \in Tr(A)$, there exists a (unique) state $\langle \alpha, \iota \rangle$ such that $\sigma \vdash \langle \alpha, \iota \rangle$. We denote $\Sigma(A) = \{\langle \alpha, \iota \rangle \mid \exists\, \sigma \in Tr(A)\ .\ \sigma \vdash \langle \alpha, \iota \rangle\}$ the set of states recognised by a CA.

A consequence of Definition 1 is that, in between two adjacent positions of a trace, in a state-consistent CA, the index counters never increase by more than one. Consequently, each transition whose relation is non-deterministic w.r.t. an index counter can be split into two transitions: an *idle* (no change) and a *tick* (increment by one). In the

[4] In fact, each index counter reaches the value $|\alpha|$ which is by one more than what is needed to traverse an array with entries $0, \ldots, |\alpha| - 1$. The reason is technical, related to the composition with transducers representing program loops (which produce array entries with a delay of one step and hence need the extra index value to produce the last array entry) as will become clear later. Note that the entry at position $|\alpha|$ is left unconstrained.

following, we will silently assume that each transition of a state-consistent CA is either idle or tick w.r.t. a given index counter.

For any set $U = \{u_1, ..., u_n\}$, let us denote $U^i = \{u_1^i, ..., u_n^i\}$ and $U^o = \{u_1^o, ..., u_n^o\}$. If $s = \langle \alpha, \iota \rangle$ and $t = \langle \beta, \kappa \rangle$ are two states such that $|\alpha| = |\beta|$, the pair $\langle s, t \rangle$ is referred to as a *transition*. A CA $T = \langle X, Q, I, \rightarrow, F \rangle$ is said to be a *transducer* iff its set of counters X is partitioned into: *input counters* $\mathbf{x}^i$ and *output counters* $\mathbf{x}^o$, where $\mathbf{x} = \{x_1, x_2, \ldots, x_k\}$, *index counters* $\mathbf{i} = \{i_1, i_2, \ldots, i_k\}$, *input parameters* $\mathbf{p}^i$ and *output parameters* $\mathbf{p}^o$, where $\mathbf{p} = \{p_1, p_2, \ldots, p_m\}$, and *working counters* $\mathbf{w}$.

Definition 2. *A sequence $\sigma \in (X \rightarrow \mathbb{Z})^*$ is said to be* consistent *with a transition $\langle s, t \rangle$, where $s = \langle \alpha, \iota \rangle$ and $t = \langle \beta, \kappa \rangle$, denoted $\sigma \vdash \langle s, t \rangle$ if and only if, for all $1 \leq p \leq k$ and all $1 \leq r \leq m$:*

1. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, we have $0 \leq \sigma_q(i_p) \leq |\alpha|$,*
2. *for all $q, r \in \mathbb{N}$ with $0 \leq q < r < |\sigma|$, we have $\sigma_q(i_p) \leq \sigma_r(i_p)$,*
3. *for all $s \in \mathbb{N}$ with $0 \leq s \leq |\alpha|$, there exists $0 \leq q < |\sigma|$ such that $\sigma_q(i_p) = s$,*
4. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, if $\sigma_q(i_p) = s < |\alpha|$, then $\sigma_q(x_p^i) = \alpha(a_p)_s$,*
5. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, if $\sigma_q(i_p) = s > 0$, then $\sigma_q(x_p^o) = \beta(a_p)_{s-1}$,*
6. *for all $q \in \mathbb{N}$ with $0 \leq q < |\sigma|$, we have $\sigma_q(p_r^i) = \iota(b_r)$ and $\sigma(p_r^o) = \kappa(b_r)$.*

The intuition behind the way the transducers represent transitions of programs with arrays is very similar to the way we use counter automata to represent states of such programs—the transducers just have input as well as output counters whose values in runs describe the corresponding input and output states. Note that the definition of transducers is such that the output values occur with a delay of exactly one step w.r.t. the corresponding input (cf. point 5 in Def. 2).[5]

A transducer T is said to be *transition consistent* iff for every trace $\sigma \in Tr(T)$ there exists a transition $\langle s, t \rangle$ such that $\sigma \vdash \langle s, t \rangle$. We denote $\Theta(T) = \{\langle s, t \rangle \mid \exists\, \sigma \in Tr(T) \,.\, \sigma \vdash \langle s, t \rangle\}$ the set of transitions recognised by a transducer.

Dependencies between Index Counters. Counter automata and transducers can represent one array in more than one way, which poses problems when composing them. For instance, the array $a = \{0 \mapsto 4, 1 \mapsto 3, 2 \mapsto 2\}$ may be encoded, e.g., by the runs $(0,4), (0,4), (1,3), (2,2), (2,2)$ and $(0,4), (1,3), (1,3), (2,2)$, where the first elements of the pairs are the values taken by the index counters, and the second elements are the values taken by the value counters corresponding to a. To obtain a sufficient criterion that guarantees that a CA and a transducer can be composed, meaning that they share a common representation of arrays, we introduce a notion of *dependence*. Intuitively, we call two or more index counters *dependent* if they increase at the same moments in all possible runs of a CA or transducer.

For the rest of this section, let $X \subset \mathbf{i}$ be a fixed set of index counters. A *dependency* δ is a conjunction of equalities between elements belonging X. For a sequence of valuations $\sigma \in (X \rightarrow \mathbb{Z})^*$, we denote $\sigma \models \delta$ if and only if $\sigma_l \models \delta$, for all $0 \leq l < |\sigma|$.

For a dependency δ, we denote $[\![\delta]\!] = \{\sigma \in (X \rightarrow \mathbb{Z})^* \mid$ there exists a state s such that $\sigma \vdash s$ and $\sigma \models \delta\}$, i.e., the set of all sequences that correspond to an array and that

[5] The intuition is that it takes the transducer one step to compute the output value, once it reads the input. It is possible to define a completely synchronous transducer, we, however, prefer this definition for technical reasons related to the translation of program loops into transducers.

satisfy δ. A dependency δ_1 is said to be *stronger* than another dependency δ_2, denoted $\delta_1 \rightarrow \delta_2$, if and only if the first order logic entailment between δ_1 and δ_2 is valid. Note that $\delta_1 \rightarrow \delta_2$ if and only if $[\![\delta_1]\!] \subseteq [\![\delta_2]\!]$. If $\delta_1 \rightarrow \delta_2$ and $\delta_2 \rightarrow \delta_1$, we write $\delta_1 \leftrightarrow \delta_2$. For a state consistent counter automaton (transition consistent transducer) A, we denote by $\Delta(A)$ the strongest dependency δ such that $Tr(A) \subseteq [\![\delta]\!]$.

Definition 3. *A CA $A = \langle \mathbf{x}, Q, I, \rightarrow, F \rangle$, where $\mathbf{x} \subseteq X$, is said to be* state-complete *if and only if for all states $s \in \Sigma(A)$, and each sequence $\sigma \in (X \rightarrow \mathbb{Z})^*$, such that $\sigma \vdash s$ and $\sigma \models \Delta(A)$, we have $\sigma \in Tr(A)$.*

Intuitively, an automaton A is state-complete if it represents any state $s \in \Sigma(A)$ in all possible ways w.r.t. the strongest dependency relation on its index counters.

Composing Counter Automata with Transducers. For a counter automaton A and a transducer T, $\Sigma(A)$ represents a set of states, whereas $\Theta(T)$ is a transition relation. A natural question is whether the post-image of $\Sigma(A)$ via the relation $\Theta(T)$ can be represented by a CA, and whether this automaton can be effectively built from A and T.

Theorem 1. *If A is a state-consistent and state-complete counter automaton with value counters $\mathbf{x} = \{x_1, ..., x_k\}$, index counters $\mathbf{i} = \{i_1, ..., i_k\}$, and parameters $\mathbf{p} = \{p_1, ..., p_m\}$, and T is a transducer with input (output) counters $\mathbf{x}^i$ ($\mathbf{x}^o$), index counters $\mathbf{i}$, and input (output) parameters $\mathbf{p}^i$ ($\mathbf{p}^o$) such that $\Delta(T)[\mathbf{x}/\mathbf{x}^i] \rightarrow \Delta(A)$, then one can build a state-consistent counter automaton B, such that $\Sigma(B) = \{t \mid \exists\, s \in \Sigma(A) \,.\, \langle s, t \rangle \in \Theta(T)\}$, and, moreover $\Delta(B) \rightarrow \Delta(T)[\mathbf{x}/\mathbf{x}^i]$.*

4 Singly Indexed Logic

We consider three types of variables. The *scalar variables* $b, b_1, b_2, ... \in BVar$ appear in the bounds that define the intervals in which some array property is required to hold and within constraints on non-array data variables. The *index variables* $i, i_1, i_2, ... \in IVar$ and *array variables* $a, a_1, a_2, ... \in AVar$ are used in array terms. The sets $BVar$, $IVar$, and $AVar$ are assumed to be pairwise disjoint.

$n, m, p \ldots$	$\in \mathbb{Z}$	integer constants	$i, j, i_1, i_2, \ldots$	$\in IVar$	index variables
$b, b_1, b_2, \ldots$	$\in BVar$	scalar variables	$a, a_1, a_2, \ldots$	$\in AVar$	array variables
ϕ		Presburger constraints	$\sim$	$\in \{\leq, \geq\}$	

B	$:= n \mid b+n$	array-bound terms
G	$:= \top \mid B \leq i \leq B \mid G \wedge G \mid G \vee G$	guard expressions
V	$:= a[i+n] \sim B \mid a_1[i+n] - a_2[i+m] \sim p \mid i - a[i+n] \sim m \mid V \wedge V$	value expressions
F	$:= \forall i \,.\, G \rightarrow V \mid \phi(B_1, B_2, \ldots, B_n) \mid \neg F \mid F \,\wedge\, F$	formulae

Fig. 1. Syntax of the Single Index Logic

Fig. 1 shows the syntax of the Single Index Logic **SIL**. We use the symbol $\top$ to denote the boolean value *true*. In the following, we will write $i < f$ instead of $i \leq f - 1$, $i = f$ instead of $f \leq i \leq f$, $\varphi_1 \vee \varphi_2$ instead of $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, and $\forall i \,.\, \upsilon(i)$ instead of $\forall i \,.\, \top \rightarrow \upsilon(i)$. If $B_1(b_1), ..., B_n(b_n)$ are bound terms with free variables $b_1, ..., b_n \in BVar$, respectively, we write any Presburger formula φ on terms $a_1[B_1], ..., a_n[B_n]$ as a shorthand for $(\bigwedge_{k=1}^{n} \forall j \;.\; j = B_k \rightarrow a_k[j] = b'_k) \wedge \varphi[b'_1/a_1[B_1],...,b'_n/a_n[B_n]]$, where $b'_1,...,b'_n$ are fresh scalar variables.

The semantics of a formula φ is defined in terms of the forcing relation $\langle \alpha, \iota \rangle \models \varphi$ between states and formulae. In particular, $\langle \alpha, \iota \rangle \models \forall i \;.\; \gamma(i, \mathbf{b}) \rightarrow \upsilon(i, \mathbf{a}, \mathbf{b})$ if and only if, for all values n in the set $\bigcap\{[-m, |\alpha| - m - 1] \mid a[i+m] \text{ occurs in } \upsilon\}$, if $\iota \models \gamma[n/i]$, then also $\iota \cup \alpha \models \upsilon[n/i]$. Intuitively, the value expression γ should hold only for those indices that do not generate out of bounds array references.

We denote $[\![\varphi]\!] = \{\langle \alpha, \iota \rangle \mid \langle \alpha, \iota \rangle \models \varphi\}$. The *satisfiability problem* asks, for a given formula φ, whether $[\![\varphi]\!] \stackrel{?}{=} \emptyset$. We say that an automaton A and a **SIL** formula φ *correspond* if and only if $\Sigma(A) = [\![\varphi]\!]$.

The $\exists^*\forall^*$ fragment of **SIL** is the set of SIL formulae which, when written in prenex normal form, have the quantifier prefix of the form $\exists i_1 \ldots \exists i_n \forall j_1 \ldots \forall j_m$. As shown in [13] (for a slightly more complex syntax), the $\exists^*\forall^*$ fragment of **SIL** is equivalent to the set of existentially quantified boolean combinations of (1) Presburger constraints on scalar variables $\mathbf{b}$, and (2) array properties of the form $\forall i \;.\; \gamma(i, \mathbf{b}) \rightarrow \upsilon(i, \mathbf{b}, \mathbf{a})$.

Theorem 2 ([13]). *The satisfiability problem is decidable for the $\exists^*\forall^*$ fragment of* **SIL**.

Below, we establish a two-way connection between $\exists^*\forall^*$-**SIL** and counter automata. Namely, we show how loop pre-conditions written in $\exists^*\forall^*$-**SIL** can be translated to CA in a way suitable for their further composition with transducers representing program loops (for this reason the translation differs from [13]). Then, we show how $\exists^*\forall^*$-**SIL** formulae can be derived from the CA that we obtain as the product of loop transducers and pre-condition CA.

4.1 From $\exists^*\forall^*$-SIL to Counter Automata

Given a pre-condition φ expressed in $\exists^*\forall^*$-**SIL**, we build a corresponding counter automaton A, i.e., $\Sigma(A) = [\![\varphi]\!]$. Without loosing generality, we will assume that the pre-condition is satisfiable (which can be effectively checked due to Theorem 2).

For the rest of this section, let us fix a set of array variables $\mathbf{a} = \{a_1, a_2, \ldots, a_k\}$ and a set of scalar variables $\mathbf{b} = \{b_1, b_2, \ldots, b_m\}$. As shown in [13], each $\exists^*\forall^*$-**SIL** formula can be equivalently written as a boolean combination of two kinds of formulae:

(i) array properties of the form $\forall i \;.\; f \leq i \leq g \rightarrow \upsilon$, where f and g are bound terms, and υ is either: (1) $a_p[i] \sim B$, (2) $i - a_p[i] \sim n$, or (3) $a_p[i] - a_q[i+1] \sim n$, where $\sim \in \{\leq, \geq\}$, $1 \leq p, q \leq k$, $n \in \mathbb{Z}$, and B is a bound term.
(ii) Presburger constraints on scalar variables $\mathbf{b}$.

Let us now fix a (normalised) pre-condition formula $\varphi(\mathbf{a}, \mathbf{b})$ of $\exists^*\forall^*$-**SIL**. By pushing negation inwards (using DeMorgan's laws) and eliminating it from Presburger constraints on scalar variables, we obtain a boolean combination of formulae of the forms (i) or (ii) above, where *only array properties may occur negated*.

W.l.o.g., we consider only pre-condition formulae without disjunctions.[6] For such formulae φ, we build CA A_φ with index counters $\mathbf{i} = \{i_1, i_2, ..., i_k\}$, value counters $\mathbf{x} = \{x_1, x_2, ..., x_k\}$, and parameters $\mathbf{p} = \{p_1, p_2, ..., p_m\}$, corresponding to the scalars $\mathbf{b}$.

For a term or formula f, we denote by $\overline{f}$ the term or formula obtained from f by replacing each b_q by p_q, $1 \leq q \leq m$, respectively. The construction of A_φ is defined recursively on the structure of φ:

[6] Given a formula containing disjunctions, we put it in DNF and check each disjunct separately.

– If $\varphi = \psi_1 \wedge \psi_2$, then $A_\varphi = A_{\psi_1} \otimes A_{\psi_2}$.
– If φ is a Presburger constraint on $\mathbf{b}$, then $A_\varphi = \langle X, Q, \{q_i\}, \rightarrow, \{q_f\}\rangle$ where:
  • $X = \{p_q \mid b_q \in FV(\varphi) \cap BVar,\ 1 \leq q \leq m\}$,
  • $Q = \{q_i, q_f\}$,
  • $q_i \xrightarrow{\overline{\varphi} \wedge \bigwedge_{x \in X} x'=x} q_f$ and $q_f \xrightarrow{\bigwedge_{x \in X} x'=x} q_f$.
– For φ being $\forall i \,.\, f \leq i \leq g \rightarrow \upsilon$, A_φ and $A_{\neg\varphi}$ have states $Q = \{q_i, q_1, q_2, q_3, q_f\}$, with q_i and q_f being the initial and final states, respectively. Intuitively, the automaton waits in q_1 increasing its index counters until the lower bound f is reached, then moves to q_2 and checks the value constraint υ until the upper bound g is reached. Finally, the control moves to q_3 and the automaton scans the rest of the array until the end. In each state, the automaton can also non-deterministically choose to idle, which is needed to ensure state-completeness when making a product of such CA. For υ of type (1) and (2), the automaton has one index (i_p) and value (x_p) counters, while for υ of type (3), there are two dependent index (i_p, i_q) and value (x_p, x_q) counters. The full definitions of A_φ and $A_{\neg\varphi}$ are given in [4], for space reasons.

We aim now at computing the strongest dependency $\Delta(A_\varphi)$ between the index counters of A_φ, and, moreover, at showing that A_φ is state-complete (cf. Definition 3). Since A_φ is defined inductively, on the structure of φ, $\Delta(A_\varphi)$ can also be computed inductively. Let $\delta(\varphi)$ be the formula defined as follows:
– $\delta(\varphi) = \top$ if φ is a Presburger constraint on $\mathbf{b}$,
– for $\varphi \equiv \forall i \,.\, f \leq i \leq g \rightarrow \upsilon$, $\delta(\varphi) \stackrel{\Delta}{=} \delta(\neg\varphi) \stackrel{\Delta}{=} \begin{cases} \top & \text{if } \upsilon \text{ is } a_p[i] \sim B \text{ or } i - a_p[i] \sim n, \\ i_p = i_q & \text{if } \upsilon \text{ is } a_p[i] - a_q[i+1] \sim n, \end{cases}$
– $\delta(\varphi_1 \wedge \varphi_2) = \delta(\varphi_1) \wedge \delta(\varphi_2)$.

Theorem 3. *Given a satisfiable $\exists^*\forall^*$-**SIL** formula φ, the following hold for the CA A_φ defined above: (1) A_φ is state consistent, (2) A_φ is state complete, (3) A_φ and φ correspond, and (4) $\delta(A_\varphi) \leftrightarrow \Delta(A_\varphi)$.*

4.2 From Counter Automata to $\exists^*\forall^*$-SIL

The purpose of this section is to establish a dual connection, from counter automata to the $\exists^*\forall^*$ fragment of **SIL**. Since obviously, counter automata are much more expressive than $\exists^*\forall^*$-**SIL**, our first concern is to abstract a given state-consistent CA A by a set of *restricted* CA $\mathcal{A}_1^K, \mathcal{A}_2^K, \ldots, \mathcal{A}_n^K$, such that $\Sigma(A) \subseteq \bigcap_{i=1}^n \Sigma(\mathcal{A}_i^K)$, and for each $\mathcal{A}_i^K$, $1 \leq i \leq n$, to generate an $\exists^*\forall^*$-**SIL** formula φ_i that corresponds to it. As a result, we obtain a formula $\varphi_A = \bigwedge_{i=1}^n \varphi_i$ such that $\Sigma(A) \subseteq [\![\varphi_A]\!]$.

Let $\rho(X, X')$ be a relation on a given set of integer variables X, and $I(X)$ be a predicate defining a subset of $\mathbb{Z}^k$. We denote by $\rho(I) = \{X' \mid \exists X \in I \,.\, \langle X, X'\rangle \in R\}$ the image of I via R, and we let $\rho \wedge I = \{\langle X, X'\rangle \in \rho \mid X \in I\}$. By ρ^n, we denote the n-times relational composition $\rho \circ \rho \circ \ldots \circ \rho$, $\rho^* = \bigvee_{n \geq 0} \rho^n$ is the reflexive and transitive closure of ρ, and $\top$ is the entire domain $\mathbb{Z}^k$. If ρ is a difference bound constraint, then ρ^n is also a difference bound constraint, for a fixed constant $n > 0$, and ρ^* is a Presburger definable relation [7, 5] (but not necessarily a difference bound constraint).

Let $\mathcal{D}(\rho)$ denote the strongest (in the logical sense) difference bound relation D s.t. $\rho \subseteq D$. If ρ is Presburger definable, $\mathcal{D}(\rho)$ can be effectively computed[7], and, moreover,

[7] $\mathcal{D}(\rho)$ can be computed by finding the unique minimal assignment $\nu : \{z_{ij} \mid 1 \leq i, j \leq k\} \rightarrow \mathbb{Z}$ that satisfies the Presburger formula $\phi(\mathbf{z}) \ :\ \forall X \forall X' \,.\, \rho(X, X') \rightarrow \bigwedge_{x_i, x_j \in X \cup X'} x_i - x_j \leq z_{ij}$.

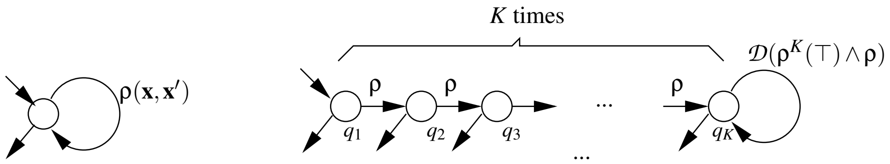

Fig. 2. K-abstraction of a relation

if ρ is a finite union of n difference bound relations, this takes $O(n \times 4k^2)$ time[8], where k is the number of free variables in ρ.

We now define the restricted class of CA, called *flat counter automata with difference bound constraints* (FCADBC) into which we abstract the given CA. A *control path* in a CA A is a finite sequence $q_1q_2...q_n$ of control states such that, for all $1 \leq i < n$, there exists a transition rule $q_i \xrightarrow{\varphi_i} q_{i+1}$. A *cycle* is a control path starting and ending in the same control state. An *elementary cycle* is a cycle in which each state appears only once, except for the first one, which appears both at the beginning and at the end. A CA is said to be *flat* (FCA) iff each control state belongs to at most one elementary cycle. An FCA such that every relation labelling a transition occurring in an elementary cycle is a DBC, and the other relations are Presburger constraints, is called an FCADBC.

With these notations, we define the *K-unfolding* of a one-state self-loop CA $A_\rho = \langle X, \{q\}, \{q\}, q \xrightarrow{\rho} q, \{q\}\rangle$ as the FCADBC $A^K_\rho = \langle X, Q^K_\rho, \{q_1\}, \rightarrow^K_\rho, Q^K_\rho\rangle$, where $Q^K_\rho = \{q_1, q_2, ..., q_K\}$ and $\rightarrow^K_\rho$ is defined such that $q_i \xrightarrow{\rho} q_{i+1}$, $1 \leq i < K$, and $q_K \xrightarrow{\rho^K(\top) \wedge \rho} q_K$. The *K-abstraction* of A_ρ, denoted $\mathcal{A}^K_\rho$ (cf. Fig. 2), is obtained from A^K_ρ by replacing the transition rule $q_K \xrightarrow{\rho^K(\top) \wedge \rho} q_K$ with the difference bound rule $q_K \xrightarrow{\mathcal{D}(\rho^K(\top) \wedge \rho)} q_K$. Intuitively, the information gathered by unfolding the *concrete* relation K times prior to the abstraction on the loop $q_K \to q_K$, allows to tighten the abstraction, according to the K parameter. Notice that the $\mathcal{A}^K_\rho$ abstraction of a relation ρ is an FCADBC with exactly one initial state, one self-loop, and all states final. The following lemma proves that the abstraction is sound, and that it can be refined, by increasing K.

Lemma 1. *Given a relation $\rho(X, X')$ on $X = \{x_1, ..., x_k\}$, the following facts hold: (1) $Tr(A_\rho) = Tr(A^K_\rho) \subseteq Tr(\mathcal{A}^K_\rho)$, for all $K > 0$, and (2) $Tr(\mathcal{A}^{K_2}_\rho) \subseteq Tr(\mathcal{A}^{K_1}_\rho)$ if $K_1 \leq K_2$.*

For the rest of this section, assume a set of arrays $\mathbf{a} = \{a_1, a_2, \ldots, a_k\}$ and a set of scalars $\mathbf{b} = \{b_1, b_2, \ldots, b_m\}$. At this point, we can describe an abstraction for counter automata that yields from an arbitrary state-consistent CA A, a set of state-consistent FCADBC $\mathcal{A}^K_1, \mathcal{A}^K_2, ..., \mathcal{A}^K_n$, whose intersection of sets of recognised states is a superset of the original one, i.e., $\Sigma(A) \subseteq \bigcap_{i=1}^n \Sigma(\mathcal{A}^K_i)$. Let A be a state-consistent CA with counters X partitioned into value counters $\mathbf{x} = \{x_1, ..., x_k\}$, index counters $\mathbf{i} = \{i_1, ..., i_k\}$, parameters $\mathbf{p} = \{p_1, ..., p_m\}$ and working counters $\mathbf{w}$. We assume that the only actions on an index counter $i \in \mathbf{i}$ are *tick* ($i' = i + 1$) and *idle* ($i' = i$), which is sufficient for the CA that we generate from **SIL** or loops.

The main idea behind the abstraction method is to keep the idle relations separate from ticks. Notice that, by combining (i.e., taking the union of) idle and tick transitions,

[8] If $\rho = \rho_1 \vee \rho_2 \vee \ldots \vee \rho_n$, and each ρ_i is represented by a $(2k)^2$-matrix M_i, $\mathcal{D}(\rho)$ is given by the pointwise maximum among all matrices M_i, $1 \leq i \leq n$.

we obtain non-deterministic relations (w.r.t. index counters) that may break the state-consistency requirement imposed on the abstract counter automata. Hence, the first step is to eliminate the idle transitions.

Let δ be an over-approximation of the dependency $\Delta(A)$, i.e., $\Delta(A) \rightarrow \delta$. In particular, if A was obtained as in Theorem 1, by composing a pre-condition automaton with a transducer T, and if we dispose of an over-approximation δ of $\Delta(T)$, i.e., $\Delta(T) \rightarrow \delta$, we have that $\Delta(A) \rightarrow \delta$, cf. Theorem 1—any over-approximation of the transducer's dependency is an over-approximation of the dependency for the post-image CA.

The dependency δ induces an equivalence relation on index counters: for all $i, j \in \mathbf{i}$, $i \simeq_\delta j$ iff $\delta \rightarrow i = j$. This relation partitions $\mathbf{i}$ into n equivalence classes $[i_{s_1}], [i_{s_2}], ..., [i_{s_n}]$, where $1 \leq s_1, s_2, ..., s_n \leq k$. Let us consider n identical copies of A: $A_1, A_2, ..., A_n$. Each copy A_j will be abstracted *w.r.t. the corresponding* $\simeq_\delta$*-equivalence class* $[i_{s_j}]$ into $\mathcal{A}_j^K$ obtained as in Fig. 2. Thus we obtain $\Sigma(A) \subseteq \bigcap_{j=1}^n \Sigma(\mathcal{A}_j^K)$, by Lemma 1.

We describe now the abstraction of the A_j copy of A into $\mathcal{A}_j^K$. W.l.o.g., we assume that the control flow graph of A_j consists of one strongly connected component (SCC)—otherwise we separately replace each (non-trivial) SCC by a flat CA obtained as described below. Out of the set of relations $\mathcal{R}_{A_j}$ that label transitions of A_j, let $\upsilon_1^j, ..., \upsilon_p^j$ be the set of *idle* relations w.r.t. $[i_{s_j}]$, i.e., $\upsilon_t^j \rightarrow \bigwedge_{i \in [i_{s_j}]} i' = i,\ 1 \leq t \leq p$, and $\theta_1^j, ..., \theta_q^j$ be the set of *tick* relations w.r.t. $[i_{s_j}]$, i.e., $\theta_t^j \rightarrow \bigwedge_{i \in [i_{s_j}]} i' = i + 1,\ 1 \leq t \leq q$. Note that since we consider index counters belonging to the same $\simeq_\delta$-equivalence class, they either all idle or all tick, hence $\{\upsilon_1^j, \ldots, \upsilon_p^j\}$ and $\{\theta_1^j, \ldots, \theta_q^j\}$ form a partition of $\mathcal{R}_{A_j}$.

Let $\Upsilon_j = \mathcal{D}(\bigvee_{t=1}^p \upsilon_t^j)$ be the best difference bound relation that approximates the idle part of A_j, and Υ_j^* be its reflexive and transitive closure[9]. Let $\Theta_j = \bigvee_{t=1}^q \mathcal{D}(\Upsilon_j^*) \circ \theta_t^j$, and let A_{Θ_j} be the one-state self-loop automaton whose transition is labelled by Θ_j, and $\mathcal{A}_j^K$ be the K-abstraction of A_{Θ_j} (cf. Fig. 2). It is to be noticed that the abstraction replaces a state-consistent FCA with a single SCC by a set of state-consistent FCADBC *with one self-loop*. The soundness of the abstraction is proved in the following:

Lemma 2. *Given a state-consistent CA A with index counters* $\mathbf{i}$ *and a dependency* δ *s.t.* $\Delta(A) \rightarrow \delta$*, let* $[i_{s_1}], [i_{s_2}], \ldots, [i_{s_n}]$ *be the partition of* $\mathbf{i}$ *into* $\simeq_\delta$*-equivalence classes. Then each* $\mathcal{A}_i^K,\ 1 \leq i \leq n$ *is state-consistent, and* $\Sigma(A) \subseteq \bigcap_{i=1}^n \Sigma(\mathcal{A}_i^K)$*, for any* $K \geq 0$.

The next step is to build, for each FCADBC $\mathcal{A}_i^K,\ 1 \leq i \leq n$, an $\exists^*\forall^*$-**SIL** formula φ_i such that $\Sigma(\mathcal{A}_i^K) = [\![\varphi_i]\!]$, for all $1 \leq i \leq n$, and, finally, let $\varphi_A = \bigwedge_{i=1}^n \varphi_i$ be the needed formula. The generation of the formulae builds on that we are dealing with CA of the form depicted in the right of Fig. 2.[10]

For a relation $\varphi(X, X')$, $X = \mathbf{x} \cup \mathbf{p}$, let $\mathcal{T}_i(\varphi)$ be the **SIL** formula obtained by replacing (1) each unprimed value counter $x_s \in FV(\varphi) \cap \mathbf{x}$ by $a_s[i]$, (2) each primed value

[9] Since Υ_j is a difference bound relation, by [7, 5], we have that Υ_j^* is Presburger definable.

[10] In case we start from a CA with more SCCs, we get a CA with a DAG-shaped control flow interconnecting components of the form depicted in Fig. 2 after the abstraction. Such a CA may be converted to **SIL** by describing each component by a formula as above, parameterised by its beginning and final index values, and then connecting such formulae by conjunctions within particular control branches and taking a disjunction of the formulae derived for the particular branches. Due to lack of space, we give this construction in detail in [4] only.

counter $x'_s \in FV(\varphi) \cap \mathbf{x}'$ by $a_s[i+1]$, and (3) each parameter $p_r \in FV(\varphi) \cap \mathbf{p}$ by b_r, for $1 \le s \le k$, $1 \le r \le m$.

For the rest, fix an automaton $\mathcal{A}_j^K$ of the form from Fig. 2 for some $1 \le j \le n$, and let $q_p \xrightarrow{\rho} q_{p+1}$, $1 \le p < K$, be its sequential part, and $q_K \xrightarrow{\lambda} q_K$ its self-loop. Let $[i_{s_j}] = \{i_{t_1}, i_{t_2}, ..., i_{t_q}\}$ be the set of relevant index counters for $\mathcal{A}_j^K$, and let $\mathbf{x}_r = \mathbf{x} \setminus \{x_{t_1}, ..., x_{t_q}\}$ be the set of redundant value counters. With these notations, the desired formula is defined as $\varphi_j = (\bigvee_{l=1}^{K-1} \tau(l)) \ \vee \ (\exists b \ . \ b \ge 0 \wedge \tau(K) \wedge \omega(b))$, where:

$$\tau(l)\text{: } \bigwedge_{s=0}^{l-1} \mathcal{T}_s(\exists \mathbf{i}, \mathbf{x}_r, \mathbf{x}'_r, \mathbf{w}.\ \rho) \qquad \omega(b)\text{: } (\forall i \ . \ K \le j < K+b \rightarrow \mathcal{T}_i(\exists \mathbf{i}, \mathbf{x}_r, \mathbf{x}'_r, \mathbf{w}.\ \lambda)) \ \wedge$$
$$\mathcal{T}_0(\exists \mathbf{i}, \mathbf{x}, \mathbf{x}', \mathbf{w}.\ \lambda^b[K/i_{t_1}, ..., i_{t_q}][K+b-1/i'_{t_1}, ..., i'_{t_q}])$$

Here, $b \in BVar$ is a fresh scalar denoting the number of times the self-loop $q_K \xrightarrow{\lambda} q_K$ is iterated. λ^b denotes the formula defining the b-times composition of λ with itself.[11]

Intuitively, $\tau(l)$ describes arrays corresponding to runs of $\mathcal{A}_j^K$ from q_1 to q_l, for some $1 \le l \le K$, without iterating the self-loop $q_K \xrightarrow{\lambda} q_K$, while $\omega(b)$ describes the arrays corresponding to runs going through the self-loop b times. The second conjunct of $\omega(b)$ uses the closed form of the b-th iteration of λ, denoted λ^b, in order to capture the possible relations between b and the scalar variables $\mathbf{b}$ corresponding to the parameters $\mathbf{p}$ in λ, created by iterating the self-loop.

Theorem 4. *Given a state-consistent CA A with index counters* $\mathbf{i}$ *and given a dependency* δ *such that* $\Delta(A) \rightarrow \delta$*, we have* $\Sigma(A) \subseteq [\![\varphi_A]\!]$*, where:*

- $\varphi_A = \bigwedge_{i=1}^{n} \varphi_i$*, where* φ_i *is the formula corresponding to* $\mathcal{A}_i^K$*, for all* $1 \le i \le n$*, and*
- $\mathcal{A}_1^K, \mathcal{A}_2^K, \ldots, \mathcal{A}_n^K$ *are the K-abstractions corresponding to the equivalence classes induced by* δ *on* $\mathbf{i}$.

5 Array Manipulating Programs

$b \in BVar$, $a \in AVar$, $i \in IVar$, $n \in \mathbb{Z}$, $c \in \mathbb{N}$

$$\begin{array}{lcl} ASGN & ::= & LHS = RHS \\ LHS & ::= & b \mid a[i+c] \\ TRM & ::= & LHS \mid i \\ RHS & ::= & TRM \mid \text{-}TRM \mid TRM\text{+}n \\ CND & ::= & CND \ \texttt{\&\&}\ CND \mid RHS \le RHS \end{array}$$

Fig. 3. Assignments and conditions

We consider programs consisting of assignments, conditional statements, and non-nested while loops in the form shown in Fig. 4, working over arrays *AVar* and scalar variables *BVar* (for a formal syntax, see [4]). In a loop, we assume a 1:1 correspondence between the set of arrays *AVar* and the set of indices *IVar*. In other words, each array is associated exactly one index variable. Each index $i \in IVar$ is initialised at the beginning of a loop using an expression of the form $b+n$ where $b \in BVar$ and $n \in \mathbb{Z}$. The indices are local to the loop. The body $S_1^l; ...; S_{n_l}^l$; of each loop branch consists of zero or more assignments followed by a single index increment statement `incr(`I`)`, $I \subseteq IVar$. The syntax of the assignments and boolean expressions used in conditional statements is shown in Fig. 3. We consider a simple syntax to make the presentation of the proposed techniques easier: various more complex features can be handled by straightforwardly extending the techniques described below.

A *state of a program* is a pair $\langle l, s \rangle$ where l is a line of the program and s is a state $\langle \alpha, \iota \rangle$ defined as in Section 3. The semantics of program statements is the usual one

[11] Since λ is difference bound relation, λ^b can be defined by a Presburger formula [7, 5].

(e.g., [19]). For simplicity of the further constructions, we assume that no *out-of-bound array references* occur in the programs—such situations are considered in [4].

Considering the program statements given in Fig. 3, we have developed a strongest post-condition calculus for the $\exists^*\forall^*$-**SIL** fragment. This calculus captures the semantics of the assignments and conditionals, and is used to deal with the sequential parts of the program (the blocks of statements outside the loops). It is also shown that $\exists^*\forall^*$-**SIL** is closed for strongest post-conditions. Full details are given in [4].

5.1 From Loops to Counter Automata

Given a loop L starting at control line l, such that l' is the control line immediately following L, we denote by $\Theta_L = \{\langle s,t\rangle \mid$ there is a run of L from $\langle l,s\rangle$ to $\langle l',t\rangle\}$ the transition relation induced by L.[12] We define the *loop dependency* δ_L as the conjunction of equalities $i_p = i_q,\ i_p, i_q \in IVar$, where (1) $e_p \equiv e_q$ where e_1 and e_2 are the expressions initialising i_p and i_q and (2) for each branch of L finished by an index increment statement `incr`(I), $i_p \in I \iff i_q \in I$. The equivalence relation $\simeq_{\delta_L}$ on index counters is defined as before: $i_p \simeq_{\delta_L} i_q$ iff $\models \delta_L \rightarrow i_p = i_q$.

$$
\begin{array}{l}
while_{a_1:i_1=e_1,\ldots,a_k:i_k=e_k}\ (C)\\
\quad if\ (C_1)\ S^1_1;\ldots;S^1_{n_1};\\
\quad else\ if\ (C_2)\ S^2_1;\ldots;S^2_{n_2};\\
\quad \ldots\\
\quad else\ if\ (C_{h-1})\ S^{h-1}_1;\ldots;S^{h-1}_{n_{h-1}};\\
\quad else\ S^h_1;\ldots;S^h_{n_h};
\end{array}
$$

Fig. 4. A while loop

Assume that we are given a loop L as in Fig. 4 with $AVar = \{a_1,\ldots,a_k\}$, $IVar = \{i_1,\ldots,i_k\}$, and $BVar = \{b_1,\ldots,b_m\}$ being the sets of array, index, and scalar variables, respectively. Let $I_1, I_2, \ldots, I_n \subseteq IVar$ be the partition of $IVar$ into equivalence classes, induced by $\simeq_{\delta_L}$. For E being a condition, assignment, index increment, or an entire loop, we define $d_E : AVar \rightarrow \mathbb{N} \cup \{\perp\}$ as $d_E(a) = max\{c \mid a[i+c]$ occurs in $E\}$ provided a is used in E, and $d_E(a) = \perp$ otherwise. The transducer $T_L = \langle X, Q, \{q_0\}, \rightarrow, \{q_{fin}\}\rangle$, corresponding to the program loop L, is defined below:

- $X = \{x^i_r, x^o_r, i_r \mid 1 \le r \le k\} \cup \{w^i_{r,l} \mid 1 \le r \le k, 1 \le l \le d_L(a_r)\} \cup \{w^o_{r,l} \mid 1 \le r \le k, 0 \le l \le d_L(a_r)\} \cup \{p^i_r, p^o_r, w_r \mid 1 \le r \le m\} \cup \{w_N\}$ where $x^{i/o}_r$, $1 \le r \le k$, are input/output array counters, $p^{i/o}_r$, $1 \le r \le m$, are parameters storing input/output scalar values, and w_r, $1 \le r \le m$, are working counters used for the manipulation of arrays and scalars (w_N stores the common length of arrays).
- $Q = \{q_0, q_{pre}, q_{loop}, q_{suf}, q_{fin}\} \cup \{q^l_r \mid 1 \le l \le h, 0 \le r < n_l\}$.
- The transition rules of T_L are the following. We assume an implicit constraint $x' = x$ for each counter $x \in X$ such that x' does not appear explicitly:
  - $q_0 \xrightarrow{\varphi} q_{pre}$, $\varphi = \bigwedge_{1\le r\le m}(w_r = p^i_r) \wedge w_N > 0 \wedge \bigwedge_{1\le r\le k}(i_r = 0 \wedge x^i_r = w^o_{r,0}) \wedge \bigwedge_{\substack{1\le r\le k\\ 1\le l\le d_L(a_r)}}(w^i_{r,l} = w^o_{r,l})$ (the counters are initialised).
  - For each $\simeq_{\delta_L}$-equivalence class I_j, $1 \le j \le n$, $q_{pre} \xrightarrow{\varphi} q_{pre}$ with $\varphi = \bigwedge_{1\le r\le k}(i_r < \xi(e_r)) \wedge \xi(incr(I))$ (T_L copies the initial parts of the arrays untouched by L).
  - $q_{pre} \xrightarrow{\varphi} q_{loop}$, $\varphi = \bigwedge_{1\le r\le k} i_r = \xi(e_r)$ (T_L starts simulating L).
  - For each $1 \le l \le h$, $q_{loop} \xrightarrow{\varphi} q^l_0$, $\varphi = \xi(C) \wedge \bigwedge_{1\le r<l}(\neg\xi(C_r)) \wedge \xi(C_l)$ where $C_h = \top$ (T_L chooses the loop branch to be simulated).

[12] Note that we ignore non-terminating runs of the loop in case there are some—our concern is not to check termination of the loop, but correctness of terminating runs of the loop.

- For each $1 \leq l \leq h$, $1 \leq r \leq n_l$, $q^l_{r-1} \xrightarrow{\xi(S^l_r)} q$ where $q = q^l_r$ if $r < n_l$, and $q = q_{loop}$ otherwise (the automaton simulates one branch of the loop).
- $q_{loop} \xrightarrow{\varphi} q_{suf}$, $\varphi = \neg\xi(C) \wedge \bigwedge_{1 \leq r \leq m}(w_r = p^o_r)$ (T_L finished the simulation of the actual execution of L).
- For each $\simeq_{\delta_L}$-equivalence class I_j, $1 \leq j \leq n$, and $i_r \in I_j$, $q_{suf} \xrightarrow{\varphi} q_{suf}$, $\varphi = i_r < w_N \wedge \xi(incr(I_j))$ (copy the array suffixes untouched by the loop).
- $q_{suf} \xrightarrow{\varphi} q_{fin}$, $\varphi = \bigwedge_{1 \leq r \leq k} i_r = w_N$ (all arrays are entirely processed).

The syntactical transformation ξ of assignments and conditions preserves the structure of these expressions, but replaces each b_r by the counter w_r and each $a_r[i_r + c]$ by $w^o_{r,c}$ for $b_r \in BVar$, $a_r \in AVar$, $i_r \in IVar$, and $c \in \mathbb{N}$. On the left-hand sides of the assignments, future values of the counters are used (cf. [4]). For increment statements we define, for all $i_r \in IVar$:

- $\xi(incr(i_r))$: $x^{i\prime}_r = w^i_{r,1} \wedge \bigwedge_{1<l \leq d_L(a_r)} w^{i\prime}_{r,l-1} = w^i_{r,l} \wedge x^{o\prime}_r = w^o_{r,0} \wedge \bigwedge_{0<l \leq d_L(a_r)} w^{o\prime}_{r,l-1} = w^o_{r,l} \wedge w^{i\prime}_{r,d_L(a_r)} = w^{o\prime}_{r,d_L(a_r)} \wedge i'_r = i_r + 1$, if $d_L(a_r) > 0$,
- $\xi(incr(i_r))$: $x^{i\prime}_r = w^{o\prime}_{r,0} \wedge x^{o\prime}_r = w^o_{r,0} \wedge i'_r = i_r + 1$, if $d_L(a_r) = 0$.

For the increment of a set of indices, we extend this definition pointwise.

The main idea of the construction is the following. T_L preserves the exact sequences of operations done on arrays and scalars in L, but performs them on suitably chosen counters instead, exploiting the fact that the program always accesses the arrays through a bounded window only, which is moving from the left to right. The contents of this window is stored in the working counters. The values stored in these counters are shifted among the counters at each increment step. In particular, the initial value of an array cell $a_r[l]$ is stored in $w^o_{r,d_L(a_r)}$ for $d_L(a_r) > 0$ (the case of $d_L(a_r) = 0$ is just a bit simpler). This value can then be accessed and/or modified via $w^o_{r,q}$ where $q \in \{d_L(a_r), ..., 0\}$ in the iterations $l - d_L(a_r), ..., l$, respectively, due to copying $w^o_{r,q}$ into $w^o_{r,q-1}$ whenever simulating $incr(i_r)$ for $q > 0$. At the same time, the initial value of $a_r[l]$ is stored in $w^i_{r,d_L(a_r)}$, which is then copied into $w^i_{r,q}$ for $q \in \{d_L(a_r) - 1, ..., 1\}$ and finally into x^i_r, which happens exactly when i_r reaches the value l. Within the simulation of the next $incr(i_r)$ statement, the final value of $a_r[l]$ appears in x^o_r, which is exactly in accordance with how a transducer expresses a change in a certain cell of an array (cf. Def. 2).

Note also that the value of the index counters i_r is correctly initialised via evaluating the appropriate initialising expressions e_r, it is increased at the same positions of the runs in both the loop L and the transducer T_L, and it is tested within the same conditions. Moreover, the construction takes care of appropriately processing the array cells which are accessed less than the maximum possible number of times (i.e., less than $\delta_L(a_r) + 1$-times) by (1) "copying" from the input x^i_r counters to the output x^o_r counters the values of all the array cells skipped at the beginning of the array by the loop, (2) by appropriately setting the initial values of all the working array counters before simulating the first iteration of the loop, and (3) by finishing the pass through the entire array even when the simulated loop does not pass it entirely.

The scalar variables are handled in a correct way too: Their input value is recorded in the p^i_r counters, this value is initially copied into the working counters w_r which are modified throughout the run of the transducer by the same operations as the appropriate program variables, and, at the end, the transducer checks whether the p_r^o counters contain the right output value of these variables.

Finally, as for what concerns the dependencies, note that all the arrays whose indices are dependent in the loop (meaning that these indices are advanced in exactly the same loop branches and are initialised in the same way) are processed at the same time in the initial and final steps of the transducers (when the transducer is in the control states q_{pre} or q_{suf}). Within the control paths leading from q_{loop} to q_{loop}, indices of such arrays are advanced at the same time as these paths directly correspond to the branches of the loop. Hence, the working counters of these arrays have always the same value, which is, however, not necessarily the case for the other arrays.

It is thus easy to see that we can formulate the correctness of the translation as captured by the following Theorem.

Theorem 5. *Given a program loop L, the following hold: (1) T_L is a transition-consistent transducer, (2) $\Theta(L) = \Theta(T_L)$, and (3) $\Delta(T_L) \rightarrow \delta_L$.*

The last point of Theorem 5 ensures that δ_L is a safe over-approximation of the dependency between the index counters of T_L. This over-approximation is used in Theorem 1 to check whether the post-image of a pre-condition automaton A can be effectively computed, by checking $\delta_T \rightarrow \Delta(A)$. In order to meet requirements of Theorem 1, one can extend T_L in a straightforward way to copy from the input to the output all the arrays and integer variables which appear in the program but not in L.

6 Examples

In order to validate our approach, we have performed proof-of-concept experiments with several programs handling integer arrays. Table 1 reports the size of the derived post-image automata (i.e., the CA representing the set of states after the main program loop) in numbers of *control states* and *counters*. The automata were slightly optimised using simple, lightweight static techniques (eliminating useless counters, compacting sequences of idling transitions with the first tick transition, eliminating clearly infeasible transitions). The result sizes give a hint on the simplicity and compactness of the obtained automata. As our prototype implementation is not completed to date, we have performed several steps of the translation into counter automata and back manually. The details of the experiments are given in [4].

Table 1. Examples

program	control states	counters
`init`	4	8
`partition`	4	24
`insert`	7	19
`rotate`	4	15

The `init` example is the classical initialisation of an array with zeros. The `partition` example copies the positive elements of an array a into another array b, and the negative ones into c. The `insert` example inserts an element on its corresponding position in a sorted array. The `rotate` example takes an array and rotates it by one position to the left. For all examples from Table 1, a human-readable post-condition describing the expected effect of the program has been inferred by our method.

7 Conclusion

In this paper, we have developed a new method for the verification of programs with integer arrays based on a novel combination of logic and counter automata. We use a logic of integer arrays to express pre- and post-conditions of programs and their parts, and counter automata and transducers to represent the effect of loops and to decide entailments. We have successfully validated our method on a set of experiments. A full implementation of our technique, which will allow us to do more experiments, is currently under way. In the future, we are, e.g., planning to investigate possibilities of using more static analyses to further shrink the size of the generated automata, optimisations to be used when computing transitive closures needed within the translation from CA to **SIL**, adjusted for the typical scenarios that happen in our setting, etc.

References

1. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In *In Proc. VMCAI'07*, *LNCS* 4349. Springer, 2007.
2. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *Proc. of PLDI'07, ACM SIGPLAN*, 2007.
3. A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting Systems with Data: A Framework for Reasoning about Systems with Unbounded Structures over Infinite Data Domains. In *Proc. FCT'07*, *LNCS* 4639, 2007.
4. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic Verification of Integer Array Programs. Technical Report TR-2009-2, Verimag, Grenoble, France, 2009.
5. M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, *LNCS* 4052. Springer, 2006.
6. A.R. Bradley, Z. Manna, and H.B. Sipma. What's Decidable About Arrays? In *Proc. of VMCAI'06*, *LNCS* 3855. Springer, 2006.
7. H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, *LNCS* 1427. Springer, 1998.
8. C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proc. of POPL'02*. ACM, 2002.
9. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision Procedures for Extensions of the Theory of Arrays. *Annals of Mathematics and Artificial Intelligence*, 50, 2007.
10. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model Checking of Array-based Systems. In *Proc. of IJCAR'08*, *LNCS* 5195. Springer, 2008.
11. D. Gopan, T.W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Array Operations. In *POPL'05*. ACM, 2005.
12. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *POPL'08*. ACM, 2008.
13. P. Habermehl, R. Iosif, and T. Vojnar. A Logic of Singly Indexed Arrays. In *Proc. of LPAR'08*, *LNAI* 5330. Springer, 2008.
14. P. Habermehl, R. Iosif, and T. Vojnar. What Else is Decidable about Integer Arrays? In *Proc. of FoSSaCS'08*, *LNCS* 4962. Springer, 2008.
15. N. Halbwachs and M. Péron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI'08*. ACM, 2008.
16. R. Jhala and K. McMillan. Array Abstractions from Proofs. In *CAV'07*, *LNCS* 4590. Springer, 2007.
17. L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE'09*, LNCS. Springer, 2009.
18. S.K. Lahiri and R.E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In *CAV'04*, *LNCS* 3114. Springer, 2004.
19. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
20. K. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS'08*, *LNCS* 4963. Springer, 2008.
21. A. Stump, C.W. Barrett, D.L. Dill, and J.R. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proc. of LICS'01*. IEEE Computer Society, 2001.