Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Automatic Verification of Integer Array Programs

Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, Tomáš Vojnar

Report n^o TR-2009-2

February 17, 2009

Automatic Verification of Integer Array Programs

Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, Tomáš Vojnar

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

February 17, 2009

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** [12] 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**.

Keywords: programs with integer arrays, formal verification, flat counter automata, logic of integer arrays

Reviewers: –

Notes: Version: 1.

How to cite this report:

@techreport {BHIKV09-TR,
title = {Automatic Verification of Integer Array Programs},
authors = {Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, Tomáš Vojnar},
institution = {Verimag Technical Report},
number = {TR-2009-2},
year = {2009},
note = {Version: 1}
}

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** [12], 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 [12], the set of models of an $\exists^*\forall^*$-**SIL** formula corresponds naturally to the set of traces of a *flat* CA with loops labeled 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 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 postcondition 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 characterize 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 $\exists^*\forall^*$-**SIL**, they can be used to check entailment of user-specified post-conditions given in $\forall^*$-**SIL**, which is possible due to the decidability of the satisfiability problem for $\exists^*\forall^*$-**SIL**.

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.

Related Work. The area of automated verification of programs with arrays and/or synthesizing loop invariants for such programs has recently received a lot of attention. For instance, [7, 17, 1, 2, 15, 11] 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 [7], constraint-based invariant synthesis [1, 2], or predicate abstraction combined with interpolation-based refinement [15].

In [19], an interpolating saturation prover is used for deriving invariants from finite unfoldings of loops. In the very recent work of [16], 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 using 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 [10]. 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 [14] 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 [5, 20, 8, 3, 9]. In general, these logics lack the capability of universally relating two successive elements of arrays, which is allowed in our previous work [13, 12]. Moreover, the logics of [5, 20, 8, 3, 9] 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 [12]. Unlike many other works, we do not synthesize 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 $\varphi(x)$, we denote by $\varphi[t/x_1,\ldots,x_n]$ the formula in which each free occurrence of $x_1,\ldots,x_n$ is replaced by a term t. Given a formula φ, we denote by $\models \varphi$ the fact that φ is logically valid, i.e., it holds in every structure corresponding to its signature.

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. It is well-known that the negation of a DBC is equivalent to a finite disjunction of *pairwise disjoint* DBCs since, e.g., $\neg(x-y \leq c) \iff y-x \leq -c-1$ and $\neg(x \leq c) \iff x \geq c+1$. In particular, the negation of $\top$ is the empty disjunction, denoted as $\perp$ (false).

A *counter automaton* (CA) is a tuple $A = \langle X, Q, I, \rightarrow, F\rangle$, 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, $\rightarrow$ 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 $\nu : X \rightarrow \mathbb{Z}$ 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 $\models \varphi[\nu(X)/X][\nu'(X')/X']$. Given two control states $q,q' \in Q$, a run of A from q to q' is a finite sequence of configurations $c_1c_2\ldots c_n$ with $c_1 = \langle q,\nu\rangle$, $c_n = \langle q',\nu'\rangle$ for some valuations $\nu,\nu' : X \rightarrow \mathbb{Z}$, 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 $Tr(A) = \{val(c_1)val(c_2)\ldots val(c_n) \mid c_1c_2\ldots c_n \in \mathcal{R}(A)\}$ 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 viceversa.

Lemma 1 *Given $A_i = \langle X_i, Q_i, I_i, \rightarrow_i, F_i\rangle$, $i = 1,2$, for all sequences $\sigma \in (X_1 \cup X_2 \rightarrow \mathbb{Z})^*$:*

$$\sigma \in Tr(A_1 \otimes A_2) \text{ if and only if } \sigma\downarrow_{X_1} \in Tr(A_1) \text{ and } \sigma\downarrow_{X_2} \in Tr(A_2)$$

Consequently, if $X_1 = X_2$ we have $Tr(A_1 \otimes A_2) = Tr(A_1) \cap Tr(A_2)$.

Proof: The states of $A_1 \otimes A_2$ are pairs of $Q_1 \times Q_2$, and the initial (final) states of $A_1 \otimes A_2$ are $I_1 \times I_2$ ($F_1 \times F_2$). There is a transition $\langle q_1,r_1\rangle \xrightarrow{\varphi} \langle q_2,r_2\rangle$ in $A_1 \otimes A_2$ if and only if there exist transitions $q_1 \xrightarrow{\varphi_1} q_2$ in A_1 and $r_1 \xrightarrow{\varphi_2} r_2$ in A_2, and $\varphi = \varphi_1 \wedge \varphi_2$. The equivalence condition is proved by induction on the length of the trace σ. $\square$

3 Counter Automata as Recognizers 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 $\alpha : \mathbf{a} \to \mathbb{Z}^*$, and $\iota : \mathbf{b} \to \mathbb{Z}$. 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 $\mathbf{a}$ and $\mathbf{b}$ are in 1:1 correspondence with $\mathbf{x}$, $\mathbf{i}$, and $\mathbf{p}$, respectively.

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

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

We call two sequences $\sigma, \rho \in (X \to \mathbb{Z})^*$ *equivalent*, denoted $\sigma \equiv \rho$, iff they agree on the value, index, and parameter counters, i.e., $\sigma\downarrow_{\mathbf{x}\cup\mathbf{i}\cup\mathbf{p}} = \rho\downarrow_{\mathbf{x}\cup\mathbf{i}\cup\mathbf{p}}$.

Proposition 1 *Let $\langle \alpha, \iota \rangle$ and $\langle \beta, \kappa \rangle$ be two states, and $\sigma \in (X \to \mathbb{Z})^*$ be a sequence that is consistent with both $\langle \alpha, \iota \rangle$ and $\langle \beta, \kappa \rangle$. Then $\alpha = \beta$ and $\iota = \kappa$.*

Proof: First we prove that $|\alpha| = |\beta|$. Suppose, by contradiction, that $|\alpha| < |\beta|$ for some $1 \le p \le k$. Then there exists $1 \le p \le k$ and $1 \le q < |\sigma|$ such that $|\alpha| < \sigma_q(i_p) \le |\beta|$, by the third point of

[1] 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.

Definition 1. Since $\sigma \vdash \alpha$, by the first point of Definition 1, $\sigma_q(i_p) \leq |\alpha|$: a contradiction. Then, $|\alpha| \geq |\beta|$, and symmetrically, we can prove that $|\alpha| \leq |\beta|$. Hence $|\alpha| = |\beta|$.

Next, we prove that, for all $1 \leq p \leq k$ and for all $0 \leq s < |\alpha|$, we have $\alpha(a_p)[s] = \beta(a_p)[s]$. By the third and fourth points of Definition 1, we have that, for all $1 \leq p \leq k$ and for all $0 \leq s < |\alpha|$, there exists $0 \leq q < |\sigma|$ such that $\alpha(a_p)[s] = \sigma_q(x_p) = \beta(a_p)[s]$.

Finally, by the fifth point of Definition 1, we have that, for all $1 \leq r \leq m$, $\iota(b_r) = \sigma_q(b_r) = \kappa(b_r)$ for all $1 \leq q < |\sigma|$. $\square$

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 recognized 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 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.

Proposition 2 *Let A be a state-consistent CA with index counters $\mathbf{i} = \{i_1, i_2, \ldots, i_k\}$, value counters $\mathbf{x} = \{x_1, x_2, \ldots, x_k\}$, and $\sigma \in Tr(A)$ be a trace. Then we have $0 \leq \sigma_{q+1}(i_p) - \sigma_q(i_p) \leq 1$, for all $1 \leq p \leq k$ and all $0 \leq q < |\sigma|$.*

Proof: By the second point of Definition 1, $\sigma_q(i_p) \leq \sigma_{q+1}(i_p)$. Now, suppose that $\sigma_{q+1}(i_p) > \sigma_q(i_p) + 1$, for some $1 \leq p \leq k$. Since A is state-consistent, there exists a state $\langle \alpha, \iota \rangle$ such that $\sigma \vdash \langle \alpha, \iota \rangle$. By the first point of Definition 1, $0 \leq \sigma_{q+1}(i_p) \leq |\alpha|$, hence $0 \leq \sigma_{q+1}(i_p) - 1 < |\alpha|$, and by the third point of Definition 1, there exists a position $0 \leq r < |\sigma|$ such that $\sigma_r(i_p) = \sigma_{q+1}(i_p) - 1$. Then either $0 \leq r < q$ or $q + 1 < r$. Both cases are in contradiction with the second point of Definition 1. $\square$

In the following, let $\mathbf{a}_l = \{a^l_1, a^l_2, \ldots, a^l_{k_l}\}$, $l = 1,2$, be two sets of array variables (not necessarily disjoint), and let $\mathbf{x}_l = \{x^l_1, x^l_2, \ldots, x^l_{k_l}\}$, $\mathbf{i}_l = \{i^l_1, i^l_2, \ldots, i^l_{k_l}\}$, $l = 1,2$, be the corresponding sets of value and index counters. Also, let $\mathbf{b}_l = \{b^l_1, b^l_2, \ldots, b^l_{m_l}\}$, $l = 1,2$, be two sets of scalar variables and let $\mathbf{p}_l = \{p^l_1, p^l_2, \ldots, p^l_{m_l}\}$, $l = 1,2$, be the corresponding sets of parameters.

Lemma 2 *Let A_l be two state consistent counter automata with value, index, parameter, and working counters $\mathbf{x}_l$, $\mathbf{i}_l$, $\mathbf{p}_l$, $\mathbf{y}_l$, $l = 1,2$, respectively. Then $A_1 \otimes A_2$ is state consistent, and, moreover, for all states $\langle \alpha, \iota \rangle$ where $\alpha : \mathbf{a}_1 \cup \mathbf{a}_2 \to \mathbb{Z}^*$ and $\iota : \mathbf{b}_1 \cup \mathbf{b}_2 \to \mathbb{Z}$, we have:*

$$\langle \alpha, \iota \rangle \in \Sigma(A_1 \otimes A_2) \Rightarrow \langle \alpha\!\downarrow_{\mathbf{a}_1}, \iota\!\downarrow_{\mathbf{b}_1} \rangle \in \Sigma(A_1) \text{ and } \langle \alpha\!\downarrow_{\mathbf{a}_2}, \iota\!\downarrow_{\mathbf{b}_2} \rangle \in \Sigma(A_2)$$

Consequently, if $\mathbf{a}_1 = \mathbf{a}_2$ and $\mathbf{b}_1 = \mathbf{b}_2$, we have $\Sigma(A_1 \otimes A_2) \subseteq \Sigma(A_1) \cap \Sigma(A_2)$.

Proof: We denote $X_l = \mathbf{x}_l \cup \mathbf{i}_l \cup \mathbf{p}_l \cup \mathbf{y}_l$, $l = 1,2$.

(1) To prove that $A_1 \otimes A_2$ is state consistent, let $\sigma \in Tr(A_1 \otimes A_2)$ be a trace. By Lemma 1, we have $\sigma\!\downarrow_{X_l} \in Tr(A_l)$, $l = 1,2$. Since A_1 and A_2 are state consistent, there exist (unique) states

$\langle \alpha_l, \iota_l \rangle$, where $\alpha_l : \mathbf{a}_l \to \mathbb{Z}^*$ and $\iota_l : \mathbf{b}_l \to \mathbb{Z}$, such that $\sigma\downarrow_{X_l} \vdash \langle \alpha_l, \iota_l \rangle$, $l = 1,2$. By Proposition 1, we have that $\langle \alpha_1 \downarrow_{\mathbf{a}_1 \cap \mathbf{a}_2}, \iota_1 \downarrow_{\mathbf{b}_1 \cap \mathbf{b}_2} \rangle = \langle \alpha_2 \downarrow_{\mathbf{a}_1 \cap \mathbf{a}_2}, \iota_2 \downarrow_{\mathbf{b}_1 \cap \mathbf{b}_2} \rangle$. Therefore, we can build from $\langle \alpha_l, \iota_l \rangle$, $l = 1,2$, a state $\langle \alpha, \iota \rangle$ such that $\sigma \vdash \langle \alpha, \iota \rangle$.

(2) To prove the second point, let $\sigma \in Tr(A_1 \otimes A_2)$ such that $\sigma \vdash \langle \alpha, \iota \rangle$. As before, there exist states $\langle \alpha_l, \iota_l \rangle$, where $\alpha_l : \mathbf{a}_l \to \mathbb{Z}^*$ and $\iota_l : \mathbf{b}_l \to \mathbb{Z}$, such that $\sigma\downarrow_{X_l} \vdash \langle \alpha_l, \iota_l \rangle$, $l = 1,2$. But since $\sigma \vdash \langle \alpha, \iota \rangle$, we have $\sigma\downarrow_{X_l} \vdash \langle \alpha\downarrow_{\mathbf{a}_l}, \iota\downarrow_{\mathbf{i}_l} \rangle$, $l = 1,2$. Thus $\langle \alpha\downarrow_{\mathbf{a}_l}, \iota\downarrow_{\mathbf{i}_l} \rangle \in \Sigma(A_l)$, $l = 1,2$. $\square$

For any set $U = \{u_1, \ldots, u_n\}$, let us denote $U^i = \{u_1^i, \ldots, u_n^i\}$ and $U^o = \{u_1^o, \ldots, u_n^o\}$. If $s = \langle \alpha, \iota \rangle$ and $t = \langle \beta, \kappa \rangle$ are two states such that $|\alpha| = |\beta|$ for all $1 \le p \le k$, the pair $\langle s, t \rangle$ is referred to as a *transition*. A CA $T = \langle X, Q, I, \to, F \rangle$ 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 $\sigma \in (X \to \mathbb{Z})^*$ 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 $1 \le p \le k$ and all $1 \le r \le m$:*

1. *for all $q \in \mathbb{N}$ with $0 \le q < |\sigma|$, we have $0 \le \sigma_q(i_p) \le |\alpha|$,*
2. *for all $q, r \in \mathbb{N}$ with $0 \le q < r < |\sigma|$, we have $\sigma_q(i_p) \le \sigma_r(i_p)$,*
3. *for all $s \in \mathbb{N}$ with $0 \le s \le |\alpha|$, there exists $0 \le q < |\sigma|$ such that $\sigma_q(i_p) = s$,*
4. *for all $q \in \mathbb{N}$ with $0 \le q < |\sigma|$, 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 $0 \le q < |\sigma|$, 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 $0 \le q < |\sigma|$, 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).[2]

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 recognized by a transducer.

[2]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.

3.1 Dependencies between Index Counters

Let X be a fixed set of counters, for the rest of this section. A *dependency* δ is a conjunction of equalities between elements belonging to (a subset of) X. For a valuation $\nu : X \to \mathbb{Z}$, we write $\nu \models \delta$ if and only if the relation obtained from δ by replacing each index counter i occurring in δ by $\nu(i)$, is logically valid. For a sequence $\sigma \in (X \to \mathbb{Z})^*$, we denote $\sigma \models \delta$ if and only if $\sigma_l \models \delta$, for all $0 \leq l < |\sigma|$.

Proposition 3 *Given an arbitrary sequence $\sigma \in (X \to \mathbb{Z})^*$, where $\mathbf{i} \subseteq X$ is a set of index counters, and a dependency δ on $\mathbf{i}$, we have $\sigma \models \delta$ if and only if $\sigma\downarrow_{\mathbf{i}} \models \delta$.*

Proof: From the definition of a dependency, δ involves only variables from $\mathbf{i}$. □

For a dependency δ, we denote $[\![\delta]\!] = \{\sigma \in (X \to \mathbb{Z})^* \mid$ 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 satisfy δ. A dependency δ_1 is said to be *stronger* than another dependency δ_2, denoted $\delta_1 \to \delta_2$, if and only if the first order logic entailment between δ_1 and δ_2 is valid. Note that $\delta_1 \to \delta_2$ if and only if $[\![\delta_1]\!] \subseteq [\![\delta_2]\!]$. If $\delta_1 \to \delta_2$ and $\delta_2 \to \delta_1$, 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 $A = \langle \mathbf{x}, Q, I, \to, F\rangle$, 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 $\sigma \in (X \to \mathbb{Z})^*$, 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. The next lemma is needed for technical reasons.

Lemma 3 *Let $A_l = \langle \mathbf{x}_l, Q_l, I_l, \to_l, F_l\rangle$, where $\mathbf{x}_l \subseteq X$, $l = 1,2$ be two state consistent and complete counter automata with the corresponding sets of arrays and scalars $\mathbf{a}_l$ and $\mathbf{b}_l$. If $\Sigma(A_1)\downarrow_{\mathbf{a}',\mathbf{b}'} \cap \Sigma(A_2)\downarrow_{\mathbf{a}',\mathbf{b}'} \neq \emptyset$, for $\mathbf{a}' = \mathbf{a}_1 \cap \mathbf{a}_2$ and $\mathbf{b}' = \mathbf{b}_1 \cap \mathbf{b}_2$, then the following hold:*

1. *$\Delta(A_1 \otimes A_2) \leftrightarrow \Delta(A_1) \wedge \Delta(A_2)$, and*
2. *$A_1 \otimes A_2$ is state complete.*

Proof: (1) Let $\mathbf{i}_1 \subseteq \mathbf{x}_1$ be the set of index counters of A_1, and $\mathbf{i}_2 \subseteq \mathbf{x}_2$ be the set of index counters of A_2.

"$\to$" According to the definition, we have

$$\begin{array}{rcl}
\Delta(A_1 \otimes A_2) & = & \bigwedge\{\delta \mid \text{for all } \sigma \in Tr(A_1 \otimes A_2) \text{ such that } \sigma \vdash s \text{ for some } s,\ \sigma \models \delta\} \\
& = & \bigwedge\{\delta \mid \text{for all traces } \sigma \in (X \to \mathbb{Z})^* \text{ such that } \sigma\downarrow_{\mathbf{x}_1} \in Tr(A_1), \\
& & \qquad \sigma\downarrow_{\mathbf{x}_2} \in Tr(A_2), \text{ and } \sigma \vdash s \text{ for some } s,\ \sigma \models \delta\} \\
& \to & \bigwedge\{\delta \mid \text{for all } \sigma \in Tr(A_1) \text{ such that } \sigma \vdash s \text{ for some } s,\ \sigma \models \delta\} \\
& = & \Delta(A_1)
\end{array}$$

Note that we have

$$\bigwedge\{\delta \mid \text{for all traces } \sigma \in (X \to \mathbb{Z})^* \text{ such that } \sigma\downarrow_{\mathbf{x}_1} \in Tr(A_1),$$
$$\sigma\downarrow_{\mathbf{x}_2} \in Tr(A_2), \text{ and } \sigma \vdash s \text{ for some } s, \sigma \models \delta\}$$
$$\to \bigwedge\{\delta \mid \text{for all } \sigma \in Tr(A_1) \text{ such that } \sigma \vdash s \text{ for some } s, \sigma \models \delta\}$$

because

$$\{\delta \mid \text{for all traces } \sigma \in (X \to \mathbb{Z})^* \text{ such that } \sigma\downarrow_{\mathbf{x}_1} \in Tr(A_1),$$
$$\sigma\downarrow_{\mathbf{x}_2} \in Tr(A_2), \text{ and } \sigma \vdash s \text{ for some } s, \sigma \models \delta\}$$
$$\supseteq \{\delta \mid \text{for all } \sigma \in Tr(A_1) \text{ such that } \sigma \vdash s \text{ for some } s, \sigma \models \delta\}$$

Symmetrically, we obtain $\Delta(A_1 \otimes A_2) \to \Delta(A_2)$. Then $\Delta(A_1 \otimes A_2) \to \Delta(A_1) \wedge \Delta(A_2)$ follows.

"$\leftarrow$" Let $\sigma \in (X \to \mathbb{Z})^*$ be a trace such that $\sigma \models \Delta(A_1) \wedge \Delta(A_2)$ and $s = \langle \alpha, \iota \rangle$ be an arbitrary state such that $\sigma \vdash s$. By Proposition 3, $\sigma\downarrow_{\mathbf{i}_1 \cup \mathbf{i}_2} \models \Delta(A_1) \wedge \Delta(A_2)$. Since $\Sigma(A_1)\downarrow_{\mathbf{a}',\mathbf{b}'} \cap \Sigma(A_2)\downarrow_{\mathbf{a}',\mathbf{b}'} \neq \emptyset$ for $\mathbf{a}' = \mathbf{a}_1 \cap \mathbf{a}_2$ and $\mathbf{b}' = \mathbf{b}_1 \cap \mathbf{b}_2$, there exists a state $t = \langle \beta, \kappa \rangle$ over arrays $\mathbf{a}_1 \cup \mathbf{a}_2$ and scalars $\mathbf{b}_1 \cup \mathbf{b}_2$ such that $t\downarrow_{\mathbf{a}_1,\mathbf{b}_1} \in \Sigma(A_1)$ and $t\downarrow_{\mathbf{a}_2,\mathbf{b}_2} \in \Sigma(A_2)$. Since $\sigma \vdash s$, it is possible to build a trace σ' such that $\sigma\downarrow_{\mathbf{i}_1 \cup \mathbf{i}_2} = \sigma'\downarrow_{\mathbf{i}_1 \cup \mathbf{i}_2}$ and $\sigma' \vdash t$: just replace the values $\alpha(a_p)[s]$ of the value counters $a_p \in \mathbf{a}$ by the values $\beta(a_p)[s]$ at each position in σ where the value of the index counter i_p is s and replace the value $\iota(b_q)$ of the parameter counters $b_q \in \mathbf{b}$ by the values $\kappa(b_q)$ at each position in σ.

We have thus $\sigma'\downarrow_{\mathbf{i}_1 \cup \mathbf{i}_2} \models \Delta(A_1) \wedge \Delta(A_2)$. By Proposition 3, we have $\sigma' \models \Delta(A_1) \wedge \Delta(A_2)$. Since A_1 is state-complete, $\sigma' \models \Delta(A_1)$ and $\sigma'\downarrow_{\mathbf{x}_1} \vdash t\downarrow_{\mathbf{a}_1,\mathbf{b}_1} \in \Sigma(A_1)$, we obtain $\sigma'\downarrow_{\mathbf{x}_1} \in Tr(A_1)$, by Definition 3. Symmetrically, we have $\sigma'\downarrow_{\mathbf{x}_2} \in Tr(A_2)$. Hence $\sigma' \in Tr(A_1 \otimes A_2)$ by Lemma 1. By Lemma 2, we have that $A_1 \otimes A_2$ is state consistent, and furthermore by definition $Tr(A_1 \otimes A_2) \subseteq [\![\Delta(A_1 \otimes A_2)]\!]$. Hence $\sigma' \models \Delta(A_1 \otimes A_2)$, therefore $\sigma'\downarrow_{\mathbf{i}_1 \cup \mathbf{i}_2} = \sigma\downarrow_{\mathbf{i}_1 \cup \mathbf{i}_2} \models \Delta(A_1 \otimes A_2)$, and by Proposition 3, $\sigma \models \Delta(A_1 \otimes A_2)$.

(2) Let $s \in \Sigma(A_1 \otimes A_2)$ be a state and $\sigma \in (X \to \mathbb{Z})^*$ be a trace such that $\sigma \vdash s$ and $\sigma \models \Delta(A_1 \otimes A_2)$. By Lemma 2, we have $s\downarrow_{\mathbf{a}_1,\mathbf{b}_1} \in \Sigma(A_1)$, $s\downarrow_{\mathbf{a}_2,\mathbf{b}_2} \in \Sigma(A_2)$ and by the previous point, $\sigma \models \Delta(A_1) \wedge \Delta(A_2)$. Since A_1 and A_2 are state-complete, we have $\sigma\downarrow_{\mathbf{x}_1} \in Tr(A_1)$ and $\sigma\downarrow_{\mathbf{x}_2} \in Tr(A_2)$ and hence $\sigma \in Tr(A_1 \otimes A_2)$ by Lemma 1. $\square$

3.2 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.

Definition 4 *Given a counter automaton A with index counters $\mathbf{i} = \{i_1, \ldots, i_k\}$, value counters $\mathbf{x} = \{x_1, \ldots, x_k\}$, and parameters $\mathbf{p} = \{p_1, \ldots, p_m\}$ and a transducer T with index counters $\mathbf{i}$, input/output counters $\mathbf{x}^i/\mathbf{x}^o$, and input/output parameters $\mathbf{p}^i/\mathbf{p}^o$, we say that A and T are* compatible *iff, for all $s \in \Sigma(A)$ such that $\langle s, t \rangle \in \Theta(T)$ for some state t, there exist traces $\sigma \in Tr(A)$ and $\rho \in Tr(T)$ such that $\sigma \vdash s$ and $\sigma \equiv \rho[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i]$.*

Informally, the definition above says that a counter automaton A and a transducer T are compatible if and only if they can agree on the representation of each state α from the intersection between $\Sigma(A)$ and the pre-image of $\Theta(T)$. This guarantees that the composition of the two will not "miss" any states.

The above definition gives a sufficient condition under which the post-image of $\Sigma(A)$ under $\Theta(T)$ can be represented by an effectively computable CA. Notice that, in general, even if the post image can be represented by a CA, it is not always the case that this CA can be computed from the description of A and T.

Lemma 4 *Given a state consistent counter automaton A with index counters $\mathbf{i} = \{i_1, \ldots, i_k\}$, value counters $\mathbf{x} = \{x_1, \ldots, x_k\}$, parameters $\mathbf{p} = \{p_1, \ldots, p_m\}$, and working counters $\mathbf{z}$ and a transition consistent transducer T with index counters $\mathbf{i}$, input/output counters $\mathbf{x}^i/\mathbf{x}^o$, input/output parameters $\mathbf{p}^i/\mathbf{p}^o$, and working counters $\mathbf{u}$, $\mathbf{z} \cap \mathbf{u} = \emptyset$, if A is compatible with T, one can construct a state consistent counter automaton B such that:*

$$\Sigma(B) = \{t \mid \exists\, s \in \Sigma(A) \,.\, \langle s, t\rangle \in \Theta(T)\}$$

Proof: We build a counter automaton B with index counters $\mathbf{i}$, value counters $\mathbf{y} = \{y_1, y_2, \ldots, y_k\}$, parameters $\mathbf{p}^o$, and working counters $\mathbf{x}^i \cup \mathbf{x}^o \cup \mathbf{p}^i \cup \mathbf{z} \cup \mathbf{u}$. By $\mathbf{x}^{io}$ and $\mathbf{p}^{io}$, we denote the sets $\mathbf{x}^i \cup \mathbf{x}^o$ and $\mathbf{p}^i \cup \mathbf{p}^o$, respectively.

First, let A' be the transducer with input counters $\mathbf{x}^i$, output counters $\mathbf{x}^o$, input parameters $\mathbf{p}^i$, output parameters $\mathbf{p}^o$, and working counters $\mathbf{z} \cup \mathbf{u}$, and with a transition rule $q \xrightarrow{\varphi[\mathbf{x}^i/\mathbf{x}][\mathbf{p}^i/\mathbf{p}]} q'$ for each rule $q \xrightarrow{\varphi} q'$ of A. Obviously, for each trace $\sigma \in Tr(A')$, we have $(\sigma\downarrow_{\mathbf{x}^i \cup \mathbf{i} \cup \mathbf{p}^i \cup \mathbf{z}})[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i] \in Tr(A)$.

Second, let T' be the transducer with input counters $\mathbf{x}^i$, output counters $\mathbf{x}^o$, and working counters $\mathbf{z} \cup \mathbf{u}$, and with the same set of transition rules as T. Finally, let $B' = A' \otimes T'$ and B be the counter automaton with index counters $\mathbf{i}$, value counters $\mathbf{y}$, where $\mathbf{y} \cap (\mathbf{x}^{io} \cup \mathbf{p}^{io} \cup \mathbf{i} \cup \mathbf{z} \cup \mathbf{u}) = \emptyset$, parameters $\mathbf{p}^o$, working counters $\mathbf{x}^{io} \cup \mathbf{p}^i \cup \mathbf{z} \cup \mathbf{u}$, and transition rules

$$q \xrightarrow{\varphi \,\wedge\, \bigwedge_{l=1}^{k} i_l' > i_l \rightarrow y_l = (x_l^o)' \,\wedge\, i_l' = i_l \rightarrow y_l' = y_l} q'$$

for each transition rule $q \xrightarrow{\varphi} q'$ of B'.

Let us prove now that B is state consistent. Let $\sigma \in Tr(B)$. By the definition of B, $\sigma' = \sigma\downarrow_{\mathbf{x}^{io} \cup \mathbf{p}^{io} \cup \mathbf{i} \cup \mathbf{z} \cup \mathbf{u}} \in Tr(B') = Tr(A') \cap Tr(T')$. Since A is state consistent, so is A'. Since T is transition consistent, so is T'. Since T' and A' share the same set of index counters, the first three points of Definition 1 hold for σ', and therefore for σ. Since T' is transition consistent, there exists a transition $\langle s, t\rangle$, $s = \langle \alpha, \iota\rangle$ and $t = \langle \beta, \kappa\rangle$ such that $\sigma_q'(x_p^o) = \beta(a_p)[\sigma_q'(i_p) - 1]$, whenever $\sigma_q'(i_p) > 0$, for all $1 \leq p \leq k$, $0 \leq q < |\sigma'|$. By the definition of B, we have $\sigma_q(y_p) = \beta(a_p)[\sigma_q'(i_p)] = \beta(a_p)[\sigma_q(i_p)]$. Last, for all parameters $p_r^o \in \mathbf{p}^o$, we have $\sigma_q(p_r^o) = \kappa(b_r)$, by Definition 2.

We are left with showing that indeed $\Sigma(B) = \{t \mid \text{exists } s \in \Sigma(A) \text{ s.t. } \langle s, t\rangle \in \Theta(T)\}$. "$\subseteq$" Let $t = \langle \beta, \kappa\rangle \in \Sigma(B)$. If $t \in \Sigma(B)$, then there exists $\sigma \in Tr(B)$ such that $\sigma \vdash t$. By the definition of B,

$\sigma' = \sigma\downarrow_{\mathbf{x}^{io}\cup\mathbf{p}^{io}\cup\mathbf{i}\cup\mathbf{z}\cup\mathbf{u}} \in Tr(B') = Tr(A') \cap Tr(T')$ such that $\sigma'_q(x^o_p) = \beta(a_p)[\sigma'_q(i_p)-1]$, whenever $\sigma'_q(i_p) > 1$, for all $1 \leq p \leq k,\ 0 \leq q < |\sigma'|$. Since $\sigma' \in Tr(T')$, there exists a state s such that $\sigma' \vdash \langle s,t\rangle$. Obviously, $\sigma'\downarrow_{\mathbf{x}^{io}\cup\mathbf{p}^{io}\cup\mathbf{i}\cup\mathbf{u}} \in Tr(T)$ and $\sigma'\downarrow_{\mathbf{x}^{io}\cup\mathbf{p}^{io}\cup\mathbf{i}\cup\mathbf{u}} \vdash \langle s,t\rangle$, hence $\langle s,t\rangle \in \Theta(T)$. By Definition 2, we have that $\sigma'\downarrow_{\mathbf{x}^{i}\cup\mathbf{p}^{i}\cup\mathbf{i}} \vdash s$. Moreover, as $\sigma' \in Tr(A')$, $(\sigma'\downarrow_{\mathbf{x}^{i}\cup\mathbf{p}^{i}\cup\mathbf{i}})[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i] \in Tr(A)$ and $(\sigma'\downarrow_{\mathbf{x}^{i}\cup\mathbf{p}^{i}\cup\mathbf{i}})[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i] \vdash s$ as well. Hence $s \in \Sigma(A)$.

"$\supseteq$" Let $s \in \Sigma(A)$ such that $\langle s,t\rangle \in \Theta(T)$. Since A and T are compatible, there exist traces $\sigma \in Tr(A)$ and $\rho \in Tr(T)$ such that $\sigma \vdash s$ and $\sigma \equiv \rho[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i]$. From σ and ρ, we can now build a trace $\pi \in Tr(A') \cap Tr(T') = Tr(B')$ such that:

- $\pi\downarrow_{\mathbf{x}^{i}\cup\mathbf{p}^{i}\cup\mathbf{i}} = \rho\downarrow_{\mathbf{x}^{i}\cup\mathbf{p}^{i}\cup\mathbf{i}} = (\sigma\downarrow_{\mathbf{x}\cup\mathbf{p}\cup\mathbf{i}})[\mathbf{x}^i/\mathbf{x}][\mathbf{p}^i/\mathbf{p}]$
- $\pi\downarrow_{\mathbf{z}} = \sigma\downarrow_{\mathbf{z}}$
- $\pi\downarrow_{\mathbf{x}^{o}\cup\mathbf{p}^{o}\cup\mathbf{u}} = \rho\downarrow_{\mathbf{x}^{o}\cup\mathbf{p}^{o}\cup\mathbf{u}}$

This is because A' does not constrain $\mathbf{x}^o$, $\mathbf{p}^o$, and $\mathbf{u}$, whereas T' does not constrain $\mathbf{z}$. Moreover, $\rho \vdash \langle s,t\rangle$ implies $\pi \vdash \langle s,t\rangle$. We can now extend $\pi \in Tr(B')$ to a trace $\pi' \in Tr(B)$ such that $\pi' \vdash t$. Hence $t \in \Sigma(B)$. □

The lemma above guarantees composability of a transducer with a counter automaton, under the compatibility condition of Definition 4. However, this condition cannot be applied in practice, due to undecidability reasons[3]. In the following, we give sufficient compatibility conditions that can easily be applied in practice.

Lemma 5 *If A is a state-complete counter automaton with value counters $\mathbf{x} = \{x_1,\ldots,x_k\}$, index counters $\mathbf{i} = \{i_1,\ldots,i_k\}$, and parameters $\mathbf{p} = \{p_1,\ldots,p_m\}$, and T is any transducer with input counters $\mathbf{x}^i$, index counters $\mathbf{i}$, and input parameters $\mathbf{p}^i$ such that $\Delta(T)[\mathbf{x}/\mathbf{x}^i] \rightarrow \Delta(A)$, then A is compatible with T.*

Proof: Let $s \in \Sigma(A)$ be a state such that $\langle s,t\rangle \in \Theta(T)$, for some state t. Hence there exists $\rho \in Tr(T)$ such that $\rho \vdash \langle s,t\rangle$, i.e., $\rho[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i] \vdash s$. As $\rho \in Tr(T)$, $\rho \models \Delta(T)$, and since $\Delta(T)[\mathbf{x}/\mathbf{x}^i] \rightarrow \Delta(A)$, we also have $\rho[\mathbf{x}/\mathbf{x}^i] \models \Delta(A)$. Let σ be a trace over the counters of A such that $\sigma \equiv \rho[\mathbf{x}/\mathbf{x}^i][\mathbf{p}/\mathbf{p}^i]$. By Proposition 3, we have $\sigma \models \Delta(A)$ and $\sigma \vdash s$. Since A is state-complete, we obtain $\sigma \in Tr(A)$. By Definition 4, A is compatible with T. □

We have reduced the problem of checking compatibility to the problems of checking state-completeness and comparing dependencies on index counters. Both criteria can now be guaranteed in a sound (but not necessarily complete) way by some syntactic conditions that will be introduced later on. Namely, we prove that a counter automaton A generated from a formula is state-complete, and we give sufficient syntactic conditions to guarantee that $\Delta(T)[\mathbf{x}/\mathbf{x}^i] \rightarrow \Delta(A)$, when A is generated from a formula and T from a program.

[3]Trace inclusion is undecidable for counter automata.

Theorem 1 *If A is a state-consistent and state-complete counter automaton with value counters $\mathbf{x} = \{x_1,\ldots, x_k\}$, index counters $\mathbf{i} = \{i_1,\ldots,i_k\}$, and parameters $\mathbf{p} = \{p_1,\ldots,p_m\}$, 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 $\Delta(B) \rightarrow \Delta(T)[\mathbf{x}/\mathbf{x}^i]$.*

Proof: By Lemma 5 A is compatible with T, and by Lemma 4, there exists B such that $\Sigma(B) = \{t \mid \exists\, s \in \Sigma(A) \,.\, \langle s,t\rangle \in \Theta(T)\}$. For the second point, notice that, in the proof of Lemma 4, $\Delta(B) = \Delta(B')[\mathbf{x}/\mathbf{x}^i] \rightarrow (\Delta(A') \wedge \Delta(T'))[\mathbf{x}/\mathbf{x}^i] = \Delta(A) \wedge \Delta(T) = \Delta(T)$. The step $\Delta(B') \rightarrow \Delta(A') \wedge \Delta(T')$ is because $B' = A' \otimes T'$ and uses the "$\rightarrow$" direction of the proof of the first point of Lemma 3. $\square$

4 Singly Indexed Logic

We consider three types of variables. The *scalar variables* $b, b_1, b_2, \ldots \in BVar$ 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* $i, i_1, i_2, \ldots \in IVar$ and *array variables* $a, a_1, a_2, \ldots \in AVar$ are used in array terms. The sets $BVar$, $IVar$, and $AVar$ are assumed to be pairwise disjoint.

$n, m, \ldots$	$\in$	$\mathbb{Z}$	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

Figure 1: Syntax of the Single Index Logic

Figure 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 $B_1(b_1), \ldots, B_n(b_n)$ are bound terms with free variables $b_1, \ldots, b_n \in BVar$, respectively, we write any Presburger formula φ on terms $a_1[B_1], \ldots, a_n[B_n]$ as a shorthand for $(\bigwedge_{k=1}^{n} \forall j \,.\, j = B_k \rightarrow a_k[i] = b'_k) \wedge \varphi[b'_1/a_1[B_1], \ldots, b'_n/a_n[B_n]]$, where $b'_1, \ldots, b'_n$ 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 $n \in \bigcap_{a[i+m] \text{ occurs in } \upsilon}[-m, |\alpha|-m-1]$, if $\models \gamma[n/i][\iota(\mathbf{b})/\mathbf{b}]$, then also $\models \upsilon[n/i][\iota(\mathbf{b})/\mathbf{b}][\alpha(\mathbf{a})/\mathbf{a}]$. 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 $\exists i_1 \ldots \exists i_n \forall i_1 \ldots \forall i_m$. As shown in [12] (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 ([12]) *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 [12]). 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 [12], 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 (normalized) 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.[4] 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. For an atomic proposition υ on array values of type (1)–(3), we define τ_υ and $\overline{\upsilon}$ as follows:

(a) $\tau_\upsilon \stackrel{\Delta}{=} \{i_p\}$ and $\overline{\upsilon} \stackrel{\Delta}{=} x_p \sim \overline{B}$ if υ is $a_p[i] \sim B$, where $1 \leq p \leq k$,

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

(b) $\tau_\upsilon \triangleq \{i_p\}$ and $\overline{\upsilon} \triangleq i_p - x_p \sim n$ if υ is $i - a_p[i] \sim n$, where $1 \leq p \leq k$, and

(c) $\tau_\upsilon \triangleq \{i_p, i_q\}$ and $\overline{\upsilon} \triangleq x_p - x'_q \sim n$ if υ is $a_p[i] - a_q[i+1] \sim n$, where $1 \leq p, q \leq k$.

For a set of index counters $I = \{i_{p_1}, i_{p_2}, \ldots, i_{p_l}\}$ where $1 \leq l \leq k$ and $p_j \in \{1, ..., k\}$ for each $1 \leq j \leq l$, we denote by $tick(I) \triangleq \bigwedge_{j=1}^{l} i'_{p_j} = i_{p_j} + 1$, by $idle(I) \triangleq \bigwedge_{j=1}^{l} i'_{p_j} = i_{p_j} \wedge x'_{p_j} = x_{p_j}$, by $I \sim \ell \triangleq \bigwedge_{j=1}^{l} i_{p_j} \sim \ell$, and by $I' \sim \ell \triangleq \bigwedge_{j=1}^{l} i'_{p_j} \sim \ell$, where $\sim \in \{<, >, =\}$ and ℓ is any linear term. For any set of counters U, let $const(U) = \bigwedge_{u \in U} u' = u$.

The construction of A_φ is defined recursively on the structure of φ:

- 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:
  - $X = \{p_q \mid b_q \in FV(\varphi) \cap BVar,\ 1 \leq q \leq m\}$,
  - $Q = \{q_i, q_f\}$,
  - $q_i \xrightarrow{\bar{\varphi} \wedge \bigwedge_{x \in X} x' = x} q_f$ and $q_f \xrightarrow{\bigwedge_{x \in X} x' = x} q_f$.

- If φ is $\forall i \,.\, f \leq i \leq g \rightarrow \upsilon$, then $A_\varphi = \langle X, Q, \{q_i\}, \rightarrow, \{q_f\}\rangle$ where:
  - $X = \{x_p, i_p \mid a_p \in FV(\varphi) \cap AVar,\ 1 \leq p \leq k\} \cup \{p_q \mid b_q \in FV(\varphi) \cap BVar,\ 1 \leq q \leq m\} \cup \{w_N\}$
  - $Q = \{q_i, q_1, q_2, q_3, q_f\}$
  - Assuming $const(\{p_q \mid b_q \in FV(\varphi) \cap BVar,\ 1 \leq q \leq m\})$ to be an implicit transition constraint, h=1 if υ is $a_p[i] - a_q[i+1] \sim n,\ 1 \leq p, q \leq k$, and h=0 otherwise, the transition relation is defined as shown in Figure 2.

- If φ is $\neg(\forall i \,.\, f \leq i \leq g \rightarrow \upsilon)$, then $A_\varphi = \langle X, Q, \{q_i\}, \rightarrow, \{q_f\}\rangle$ where:
  - $X = \{x_p, i_p \mid a_p \in FV(\varphi) \cap AVar,\ 1 \leq p \leq k\} \cup \{v_q \mid b_q \in FV(\varphi) \cap BVar,\ 1 \leq q \leq m\} \cup \{w_N\}$
  - $Q = \{q_i, q_1, q_2, q_3, q_f\}$
  - Assuming $const(\{v_q \mid b_q \in FV(\varphi) \cap BVar,\ 1 \leq q \leq m\})$ to be an implicit transition constraint, h=1 if υ is $a_p[i] - a_q[i+1] \sim n,\ 1 \leq p, q \leq k$ and h=0 otherwise, the transition relation is defined as shown in Figure 3.

Intuitively, the automaton A_φ or $A_{\neg\varphi}$ for a formula $\forall i \,.\, f \leq i \leq g \rightarrow \upsilon$ 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

$$q_i \xrightarrow{\tau_\upsilon=0 \,\wedge\, idle(\tau_\upsilon)} q_i \qquad q_j \xrightarrow{idle(\tau_\upsilon)} q_j \text{ for all } j \in \{1,2,3,f\}$$

$$q_i \xrightarrow{\tau_\upsilon+1<\overline{f}\leq\overline{g} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, 0=\tau_\upsilon<w_N-1} q_1 \qquad q_1 \xrightarrow{\tau_\upsilon+1<\overline{f} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, \tau_\upsilon<w_N-1} q_1$$

$$q_1 \xrightarrow{\tau_\upsilon+1=\overline{f} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, \tau_\upsilon<w_N-1} q_2 \qquad q_2 \xrightarrow{\tau_\upsilon<\overline{g} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, \overline{\upsilon} \,\wedge\, \tau_\upsilon<w_N-1} q_2$$

$$q_2 \xrightarrow{\tau_\upsilon=\overline{g} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, \overline{\upsilon} \,\wedge\, \tau_\upsilon<w_N-1} q_3 \qquad q_3 \xrightarrow{tick(\tau_\upsilon) \,\wedge\, \tau_\upsilon<w_N-1} q_3$$

$$q_3 \xrightarrow{tick(\tau_\upsilon) \,\wedge\, \tau_\upsilon=w_N-1} q_f$$

$$q_i \xrightarrow{\overline{f}\leq\tau_\upsilon<\overline{g} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, \overline{\upsilon} \,\wedge\, 0=\tau_\upsilon<w_N-1} q_2 \qquad q_i \xrightarrow{\tau_\upsilon+1=\overline{f}\leq\overline{g} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, 0=\tau_\upsilon<w_N-1} q_2$$

$$q_i \xrightarrow{\overline{f}\leq\tau_\upsilon=\overline{g} \,\wedge\, tick(\tau_\upsilon) \,\wedge\, \overline{\upsilon} \,\wedge\, 0=\tau_\upsilon<w_N-1} q_3 \qquad q_i \xrightarrow{(\overline{f}>\overline{g}\vee\overline{f}\leq\overline{g}<0) \,\wedge\, tick(\tau_\upsilon) \,\wedge\, 0=\tau_\upsilon<w_N-1} q_3$$

$$q_2 \xrightarrow{tick(\tau_\upsilon) \,\wedge\, \overline{\upsilon} \,\wedge\, \tau_\upsilon=w_N-1} q_f \text{ if } h=0 \qquad q_2 \xrightarrow{tick(\tau_\upsilon) \,\wedge\, \tau_\upsilon=w_N-1} q_f \text{ if } h=1$$

$$q_1 \xrightarrow{tick(\tau_\upsilon) \,\wedge\, \tau_\upsilon=w_N-1} q_f$$

$$q_i \xrightarrow{tick(\tau_\upsilon) \,\wedge\, \overline{f}\leq\tau_\upsilon\leq\overline{g} \,\wedge\, \overline{\upsilon} \,\wedge\, 0=\tau_\upsilon=w_N-1} q_f \text{ if } h=0 \qquad q_i \xrightarrow{tick(\tau_\upsilon) \,\wedge\, (\overline{f}>\overline{g}\vee\overline{f}\leq\overline{g}<0) \,\wedge\, 0=\tau_\upsilon=w_N-1} q_f \text{ if } h=0$$

$$q_i \xrightarrow{tick(\tau_\upsilon) \,\wedge\, 0<\overline{f}\leq\overline{g} \,\wedge\, 0=\tau_\upsilon=w_N-1} q_f \text{ if } h=0 \qquad q_i \xrightarrow{tick(\tau_\upsilon) \,\wedge\, 0=\tau_\upsilon=w_N-1} q_f \text{ if } h=1$$

Figure 2: Transition rules of the automaton A_φ for $\varphi \equiv \forall i \,.\, f \leq i \leq g \rightarrow \upsilon$

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.

Figure 4.1 shows the CA A_φ for $\varphi : \forall i \,.\, f \leq i \leq g \rightarrow \upsilon$. Figure 4.1 shows the CA A_φ for $\varphi : \neg(\forall i \,.\, f \leq i \leq g \rightarrow \upsilon)$.

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 $\varphi \equiv \forall i \,.\, f \leq i \leq g \rightarrow \upsilon$, $\delta(\varphi) \triangleq \delta(\neg\varphi) \triangleq \begin{cases} \top & \text{if } \upsilon \text{ is } a_p[i] \sim B \text{ or } i - a_p[i] \sim n, \\ i_p = i_q & \text{if } \upsilon \text{ is } a_p[i] - a_q[i+1] \sim n, \end{cases}$
- $\delta(\varphi_1 \wedge \varphi_2) = \delta(\varphi_1) \wedge \delta(\varphi_2)$.

Theorem 3 *Given a satisfiable formula φ of $\exists^*\forall^*$-**SIL**, the following hold for the CA A_φ, defined in the previous:*

1. *A_φ is state consistent,*
2. *A_φ is state complete,*
3. *A_φ and φ correspond,*
4. *$\delta(A_\varphi) \leftrightarrow \Delta(A_\varphi)$.*

$$q_i \xrightarrow{\tau_\upsilon=0\ \wedge\ idle(\tau_\upsilon)} q_i \qquad q_j \xrightarrow{idle(\tau_\upsilon)} q_j \text{ for all } j \in \{1,2,3,f\}$$

$$q_i \xrightarrow{\tau_\upsilon+1<\overline{f}\leq\overline{g}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ 0=\tau_\upsilon<w_N-1} q_1 \qquad q_1 \xrightarrow{\tau_\upsilon+1<\overline{f}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ \tau_\upsilon<w_N-1} q_1$$

$$q_1 \xrightarrow{\tau_\upsilon+1=\overline{f}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ \tau_\upsilon<w_N-1} q_2 \qquad q_2 \xrightarrow{\tau_\upsilon<\overline{g}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ \overline{\upsilon}\ \wedge\ \tau_\upsilon<w_N-1} q_2$$

$$q_2 \xrightarrow{\tau_\upsilon\leq\overline{g}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ \neg\overline{\upsilon}\ \wedge\ \tau_\upsilon<w_N-1} q_3 \qquad q_3 \xrightarrow{tick(\tau_\upsilon)\ \wedge\ \tau_\upsilon<w_N-1} q_3$$

$$q_i \xrightarrow{\overline{f}\leq\tau_\upsilon\leq\overline{g}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ \neg\overline{\upsilon}\ \wedge\ 0=\tau_\upsilon<w_N-1} q_3 \qquad q_3 \xrightarrow{tick(\tau_\upsilon)\ \wedge\ \tau_\upsilon=w_N-1} q_f$$

$$q_i \xrightarrow{\overline{f}\leq\tau_\upsilon<\overline{g}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ \overline{\upsilon}\ \wedge\ 0=\tau_\upsilon<w_N-1} q_2 \qquad q_i \xrightarrow{\tau_\upsilon+1=\overline{f}\leq\overline{g}\ \wedge\ tick(\tau_\upsilon)\ \wedge\ 0=\tau_\upsilon<w_N-1} q_2$$

$$q_2 \xrightarrow{tick(\tau_\upsilon)\ \wedge\ \neg\overline{\upsilon}\ \wedge\ \tau_\upsilon=w_N-1} q_f \text{ if } h=0 \qquad q_i \xrightarrow{tick(\tau_\upsilon)\ \wedge\ \overline{f}\leq\tau_\upsilon\leq\overline{g}\ \wedge\ \neg\overline{\upsilon}\ \wedge\ 0=\tau_\upsilon=w_N-1} q_f \text{ if } h=0$$

Figure 3: Transition rules of the automaton $A_{\neg\varphi}$ for $\varphi \equiv \forall i\ .\ f \leq i \leq g \rightarrow \upsilon$

Proof: By induction on the structure of φ:

- $\varphi = \forall i\ .\ f \leq i \leq g \rightarrow \upsilon$. This case is by analysis of the CA in Figure 4.1.
- $\varphi = \neg(\forall i\ .\ f \leq i \leq g \rightarrow \upsilon)$. This case is by analysis of the CA in Figure 4.1.
- φ is a Presburger constraint on $\mathbf{b}$ and $|a|$, $a \in \mathbf{a}$. This case is by the analysis of the CA.
- $\varphi = \psi_1 \wedge \psi_2$. Since φ is satisfiable, there exists a state $\langle\alpha,\iota\rangle \in [\![\varphi]\!] = [\![\psi_1]\!] \cap [\![\psi_2]\!]$. By the induction hypothesis, A_{ψ_1} corresponds to ψ_1, hence there exists a trace $\sigma_1 \in Tr(A_{\psi_1})$ such that $\sigma_1 \vdash \langle\alpha,\iota\rangle$, i.e. $\langle\alpha,\iota\rangle \in \Sigma(A_{\psi_1})$. Symmetrically, $\langle\alpha,\iota\rangle \in \Sigma(A_{\psi_2})$, therefore $\Sigma(A_{\psi_1}) \cap \Sigma(A_{\psi_2}) \neq \emptyset$. By applying Lemma 3 and the induction hypothesis, we obtain that

  $$\Delta(A_\varphi) = \Delta(A_{\psi_1}) \wedge \Delta(A_{\psi_2}) \ \leftrightarrow\ \delta(\psi_1) \wedge \delta(\psi_2) = \delta(\varphi)$$

  and that $A_\varphi = A_{\psi_1} \otimes A_{\psi_2}$ is state complete, since A_{ψ_1} and A_{ψ_2} are. The fact that A_φ is state complete follows directly from the induction hypothesis. We are left with proving that A_φ corresponds to φ. Let X_i denote in the following the sets of counters of A_{ψ_i}, and let $\mathbf{a}_i$, $\mathbf{b}_i$ denote the sets of array and bound variables corresponding to X_i, $i = 1,2$.

  - Let $\sigma \in Tr(A_\varphi)$ be a trace. By Lemma 1, we have $\sigma{\downarrow_{X_i}} \in Tr(A_{\psi_i})$, $i = 1,2$. By the induction hypothesis, there exist states $s_i = \langle\alpha_i,\iota_i\rangle \in [\![\psi_i]\!]$ such that $\sigma{\downarrow_{X_i}} \vdash s_i$, $i = 1,2$. By Proposition 1 we can build a state $s = \langle\alpha,\iota\rangle$ such that $\langle\alpha{\downarrow_{\mathbf{a}_i}},\iota{\downarrow_{\mathbf{b}_i}}\rangle = s_i$, $i = 1,2$. Hence $\sigma \vdash s$ and $s \in [\![\psi_1 \wedge \psi_2]\!] = [\![\varphi]\!]$.
  - Let $s \in [\![\varphi]\!]$ be a state. Since $s \in [\![\psi_1]\!] \cap [\![\psi_2]\!]$, by the induction hypothesis there exists two traces $\sigma_i \in Tr(A_{\psi_i})$ such that $\sigma_i \vdash s$, $i = 1,2$. By an argument similar to the one used in the proof of Lemma 3, we can build a sequence $\sigma \in (X_1 \cup X_2 \rightarrow \mathbb{Z})^*$ such that $\sigma{\downarrow_{X_i}} \in Tr(A_{\psi_i})$, $i = 1,2$, and $\sigma \vdash s$. By Lemma 1, we have $\sigma \in Tr(A_{\psi_1} \otimes A_{\psi_2}) = Tr(A_\varphi)$.

□

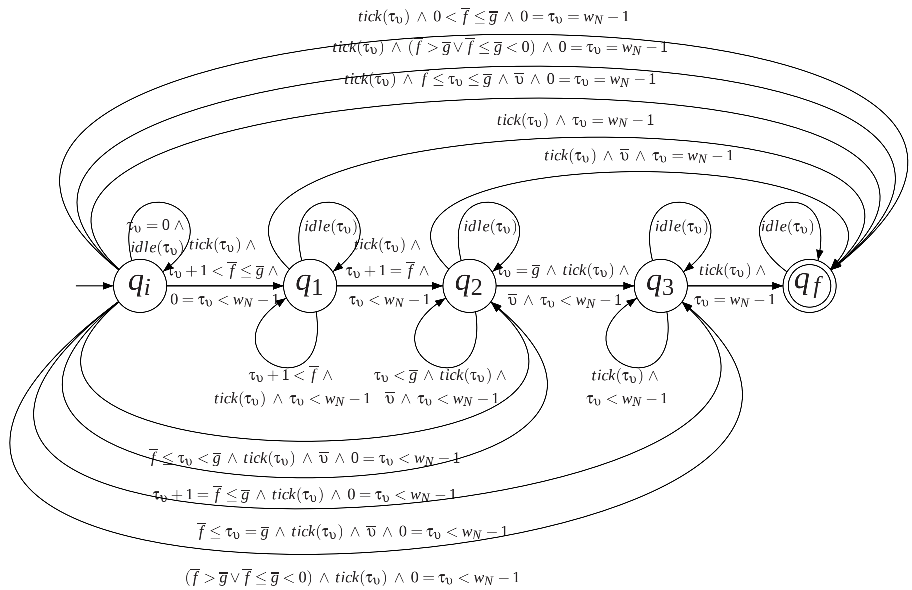


Figure 4: The counter automaton for the SIL formulae $\forall i \,.\, f \leq i \leq g \rightarrow \upsilon$ for the $h = 0$ case (for the other case, the three transitions from q_i to q_f are replaced with only one transition labeled with $tick(\tau_\upsilon) \wedge 0 = \tau_\upsilon = w_N - 1$, and $\overline{\upsilon}$ is removed in the transition $q_2 \rightarrow q_f$)

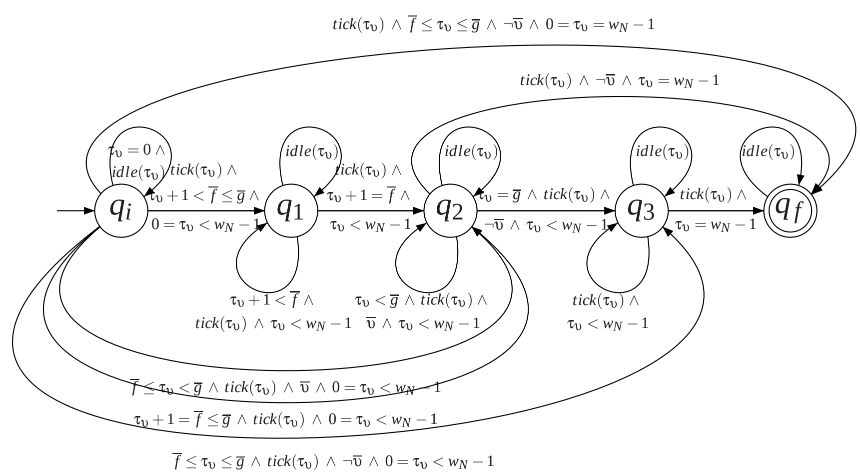


Figure 5: The counter automaton for the SIL formulae $\neg(\forall i \,.\, f \leq i \leq g \rightarrow \upsilon)$ for the $h = 0$ case (for the other case, two transitions, namely $q_i \rightarrow q_f$ and $q_2 \rightarrow q_f$, are removed)

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 $\mathbb{Z}^k$. It is known [6, 4] that ρ^n and ρ^* are Presburger definable if ρ is 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[5], and, moreover, if ρ is a finite union of n difference bound relations, this takes $O(n \times 4k^2)$ time[6].

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 labeling 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 counter automaton $A_\rho = \langle X, \{q\}, \{q\}, q \xrightarrow{\rho} q, \{q\}\rangle$ as the FCADBC $A_\rho^K = \langle X, Q_\rho^K, \{q_1\}, \rightarrow_\rho^K, Q_\rho^K\rangle$, where $Q_\rho^K = \{q_1, q_2, ..., q_K\}$ and $\rightarrow_\rho^K$ 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 $\mathcal{A}_\rho^K$ (cf. Figure 6), is obtained from A_ρ^K 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 $q_K \rightarrow q_K$, allows to tighten the abstraction, according to the K parameter. Notice that the $\mathcal{A}_\rho^K$ 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 6 *Given a relation $\rho(X,X')$ on $X = \{x_1, x_2, \ldots, x_k\}$, the following hold:*

- $Tr(A_\rho) = Tr(A_\rho^K) \subseteq Tr(\mathcal{A}_\rho^K)$, *for all* $K > 0$,

[5] $\mathcal{D}(\rho)$ 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 $\phi(\mathbf{z}) \;:\; \forall X \forall X' \,.\, \rho(X,X') \rightarrow \bigwedge_{x_i,x_j \in X \cup X'} x_i - x_j \leq z_{ij}$.

[6] 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, $1 \leq i \leq n$.

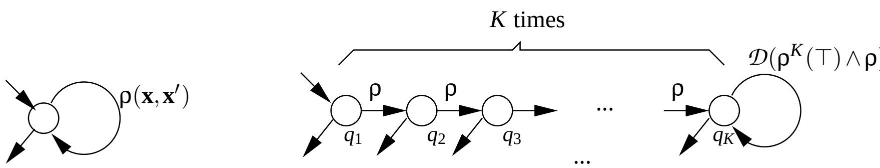


Figure 6: K-abstraction of a relation

- $Tr(\mathcal{A}_\rho^{K_2}) \subseteq Tr(\mathcal{A}_\rho^{K_1})$ *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 $\mathcal{A}_1^K, \mathcal{A}_2^K, \ldots, \mathcal{A}_n^K$, whose intersection of sets of recognized states is a superset of the original one, i.e., $\Sigma(A) \subseteq \bigcap_{i=1}^n \Sigma(\mathcal{A}_i^K)$. Let A be a state-consistent CA with counters X partitioned into value counters $\mathbf{x} = \{x_1, \ldots, x_k\}$, index counters $\mathbf{i} = \{i_1, \ldots, i_k\}$, parameters $\mathbf{p} = \{p_1, \ldots, p_m\}$ 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, 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 $[i_{s_1}], [i_{s_2}], \ldots, [i_{s_n}]$, where $1 \leq s_1, s_2, \ldots, s_n \leq k$. Let us consider n identical copies of A: $A_1, A_2, \ldots, A_n$. 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 Figure 6. Thus we obtain $\Sigma(A) \subseteq \bigcap_{j=1}^n \Sigma(\mathcal{A}_j^K)$, by Lemma 6.

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 $\upsilon_1^j, \ldots, \upsilon_p^j$ 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 $\theta_1^j, \ldots, \theta_q^j$ 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[7]. Let $\Theta_j = \bigvee_{t=1}^q \mathcal{D}(\Upsilon_j^*) \circ \theta_t^j$, and let A_{Θ_j} be the

[7]Since Υ_j is a difference bound relation, by [6, 4], we have that Υ_j^* is Presburger definable.

one-state self-loop automaton whose transition is labeled by Θ_j, and $\mathcal{A}_j^K$ be the K-abstraction of A_{Θ_j} (cf. Figure 6). 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 7 *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 Figure 6.[8]

For a relation $\varphi(X, X')$, $X = \mathbf{x} \cup \mathbf{p}$, let $\mathcal{T}_i(\varphi)$ be the **SIL** formula obtained by replacing each:

- unprimed value counter $x_s \in FV(\varphi) \cap \mathbf{x}$ by $a_s[i]$, $1 \leq s \leq k$,
- primed value counter $x_s' \in FV(\varphi) \cap \mathbf{x}'$ by $a_s[i+1]$, $1 \leq s \leq k$,
- parameter $v_s \in FV(\varphi) \cap \mathbf{v}$ by b_s, $1 \leq s \leq m$.

For the rest, fix an automaton $\mathcal{A}_j^K$ of the form from Figure 6 for some $1 \leq j \leq n$, and let $q_p \xrightarrow{\rho} q_{p+1}$, $1 \leq p < K$, be its sequential part, and $q_K \xrightarrow{\lambda} q_K$ its self-loop. Let $[i_{s_j}] = \{i_{t_1}, i_{t_2}, \ldots, i_{t_q}\}$ be the set of relevant index counters for $\mathcal{A}_j^K$, and let $\mathbf{x}_r = \mathbf{x} \setminus \{x_{t_1}, \ldots, x_{t_q}\}$ be the set of redundant value counters. With these notations, the desired formula is defined as $\varphi_j = (\bigvee_{l=1}^{K-1} \tau(l)) \vee (\exists b \,.\, b \geq 0 \wedge \tau(K) \wedge \omega(b))$, where:

$$\tau(l) \ : \ \bigwedge_{s=0}^{l-1} \mathcal{T}_s(\exists \mathbf{i}, \mathbf{x}_r, \mathbf{x}_r', \mathbf{w}.\ \rho)$$

$$\begin{aligned} \omega(b) \ : \ & (\forall i \,.\, K \leq \mathrm{j} < K + b \rightarrow \mathcal{T}_i(\exists \mathbf{i}, \mathbf{x}_r, \mathbf{x}_r', \mathbf{w}.\ \lambda)) \ \wedge \\ & \mathcal{T}_0(\exists \mathbf{i}, \mathbf{x}, \mathbf{x}', \mathbf{w}.\ \lambda^b[K/i_{t_1}, \ldots, i_{t_q}][K + b - 1/i_{t_1}', \ldots, i_{t_q}']) \end{aligned}$$

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.[9]

[8] 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 Figure 6 after the abstraction. Such a CA may be converted to **SIL** by describing each component by a formula as above, parameterized 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.

[9] Since λ is difference bound relation, λ^b can be defined by a Presburger formula [6, 4].

Intuitively, $\tau(l)$ describes arrays corresponding to runs of $\mathcal{A}_j^K$ from q_1 to q_l, for some $1 \leq l \leq K$, 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 $1 \leq i \leq n$, 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

We consider programs consisting of assignments, conditional statements, and non-nested while loops in the syntax shown in Figure 7. We consider a very 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 (e.g., [18]). For simplicity of the further constructions, we assume that no *out-of-bound array references* occur in the programs. However, the approach can be extended to take care of such references if extended as described in Appendix B.

Considering the program statements given in Figure 7, we have developed a strongest post-condition calculus for the $\exists^*\forall^*$-**SIL**, given in Appendix A. 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.

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 $\Theta_L = \{\langle s, t \rangle \mid \text{there is a run of } L \text{ from } \langle l, s \rangle \text{ to } \langle l', t \rangle\}$ the transition relation induced by L. 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 initializing i_p and i_q and (2) for each branch of L finished by an index increment statement $\texttt{incr}(I)$, $i_p \in I \iff i_q \in I$. The equivalence relation $\simeq_{\delta_L}$ on index counters is defined as before: $i_p \simeq_{\delta_L} i_q$ iff $\models \delta_L \rightarrow i_p = i_q$.

Assume that we are given a loop L as in Figure 8 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

$$
\begin{array}{lcl}
a, a_1, a_2, \ldots \in AVar & \ldots & \text{array variables} \\
i, i_1, i_2, \ldots \in IVar & \ldots & \text{index variables} \\
b, b_1, b_2, \ldots \in BVar & \ldots & \text{scalar variables}
\end{array}
\qquad
\begin{array}{lcl}
n \in \mathbb{Z} & \ldots & \text{integer constants} \\
c \in \mathbb{N} & \ldots & \text{natural constants}
\end{array}
$$

$$
\begin{array}{lcll}
LHS_L & ::= & b \mid a[i+c] & \text{left-hand sides in loops} \\
RHS_L & ::= & LHS_L\text{+}n \mid i+n & \text{right-hand sides in loops}^{(1)} \\
ASGN_L & ::= & LHS_L = RHS_L; & \text{assignements in loops} \\
CND_L & ::= & CND_L \;\&\&\; CND_L \mid RHS_L \leq RHS_L & \text{conditions in loops} \\
INC & ::= & incr(\{[i,]^*\, i\}); & \text{index increments}^{(2)} \\
IF_L & ::= & if\ (CND_L)\ ASGN_L^*\ INC & \text{conditional statements in loops}^{(3)} \\
 & & [\ else\ if\ (CND_L)\ ASGN_L^*\ INC\ ]^* & \\
 & & else\ ASGN_L^*\ INC & \\
IDX & ::= & [\ a : i = b+n,\ ]^*\ a : i = b+n & \text{index declaration and initialization}^{(4)} \\
WHILE & ::= & while_{IDX}\ (CND_L)\ IF_L & \text{while loops} \\
 & & & \\
LHS_P & ::= & b \mid a[b+c] & \text{left-hand sides outside of loops} \\
RHS_P & ::= & LHS_P\text{+}n & \text{right-hand sides outside of loops}^{(1)} \\
ASGN_P & ::= & LHS_P = RHS_P; & \text{assignements outside of loops} \\
CND_P & ::= & CND_P \;\&\&\; CND_P \mid RHS_P \leq RHS_P & \text{conditions outside of loops} \\
IF_P & ::= & if\ (CND_P)\ [\ ASGN_P \mid IF_P \mid WHILE\ ]^* & \text{conditional statements outside of loops}^{(3)} \\
 & & [\ else\ if\ (CND_P)\ [\ ASGN_P \mid IF_P \mid WHILE\ ]^*\ ]^* & \\
 & & else\ [\ ASGN_P \mid IF_P \mid WHILE\ ]^* & \\
 & & & \\
PROGRAM & ::= & [\ ASGN_P \mid IF_P \mid WHILE\ ]^+ & \text{array programs}
\end{array}
$$

(1) If n is zero, we skip it.
(2) Each index variable may be incremented at most once in the increment statement *incr*.
(3) If the condition is *true*, we skip the if keyword and the *else* branch.
(4) We assume a 1:1 correspondence between arrays and indices in the loop. The indices are local to the loop.

Figure 7: Syntax of the considered array programs

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

Figure 8: A while loop

$d_E(a) = max\{c \mid a[i+c] \text{ occurs in } E\}$ 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, $p^{i/o}_r$, $1 \le r \le k$, 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).
- $Q = \{q_0, q_{pre}, q_{loop}, q_{suf}, q_{fin}\} \cup \{q^r_l \mid 1 \le r \le h, 0 \le l < n_r\}$.
- 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 initialized).
  - 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).
  - For each $1 \le l \le h$, $1 \le r \le n_l$, $q^l_{r-1} \xrightarrow{\xi(S^l_r)} q$ where $q = q^l_r$ if $r < n_l$, and $q = q_{loop}$ otherwise (the automaton simulates one branch of the loop).
  - $q_{loop} \xrightarrow{\varphi} q_{suf}$, $\varphi = \neg\xi(C) \wedge \bigwedge_{1\le r\le m}(w_r = p^o_r)$ (T_L finished the simulation of the actual execution of L).
  - For each $\simeq_{\delta_L}$-equivalence class I_j, $1 \le j \le n$, 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. The translation of the increment statements is a bit more involved as it implies "shifting" of the contents of the window containing the array entries that the program can currently manipulate:

- $\xi(n) := n$ for $n \in \mathbb{Z}$, $\xi(b_r) := w_r$ for $1 \leq r \leq m$, $\xi(i_r) := i_r$ for $1 \leq r \leq k$, and $\xi(a_r[i_r + c]) := w^o_{r,c}$ for $1 \leq r \leq k$, $c \in \mathbb{N}$,
- $\xi(LHS_L\text{+}n) := \xi(LHS_L) + n$ for $n \in \mathbb{Z}$,
- $\xi(LHS_L = RHS_L) := (\xi(LHS_L))' = \xi(RHS_L)$,
- $\xi(RHS_{L,1} \leq RHS_{L,2}) := \xi(RHS_{L,1}) \leq \xi(RHS_{L,2})$,
- $\xi(CND_{L,1}\ \&\&\ CND_{L,2}) := \xi(CND_{L,1}) \wedge \xi(CND_{L,2})$,
- $\xi(incr(I)) := \bigwedge_{i_r \in I} \xi(incr(i_r))$ for $I \subseteq IVar$, and
- $\xi(incr(i_r)) := x^{i\,\prime}_r = w^i_{r,1} \wedge \bigwedge_{1<l\leq d_L(a_r)} w^{i\prime}_{r,l-1} = w^i_{r,l} \wedge x^{o\prime}_r = w^o_{r,0} \wedge$
  $\bigwedge_{0<l\leq d_L(a_r)} w^{o\prime}_{r,l-1} = w^o_{r,l} \wedge w^{i\prime}_{r,d_L(a_r)} = w^{o\prime}_{r,d_L(a_r)} \wedge i'_r = i_r + 1$,
  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 $d_L(a_r) = 0$.

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 whose meaning shifts 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 initialized via evaluating the appropriate initializing 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_r^i counters to the output x_r^o 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_r^i 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 variable, 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 initialized 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:*

- T_L *is a transition-consistent transducer,*
- $\Theta(L)=\Theta(T_L)$, *and*
- $\Delta(T_L)\rightarrow\delta_L$.

The last point 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.

Note that the described translation is not intended to be as optimal as possible—it can for sure be optimized and one can also use common static analyses for a further optimization of the obtained transducers. For example, one can always in a standard way (using substitutions of the assigned values) compress each loop of T_L simulating a single branch of L into a self-loop.

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 optimized 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 Appenix C.

Table 1: Case studies

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

The `init` example is the classical initialization 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, optimizations 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.

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

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. 1

[2] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *Proc. of PLDI'07, ACM SIGPLAN*, 2007. 1

[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. 1

[4] M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, *LNCS* 4052. Springer, 2006. 4.2, 7, 9

[5] A.R. Bradley, Z. Manna, and H.B. Sipma. What's Decidable About Arrays? In *Proc. of VMCAI'06*, *LNCS* 3855. Springer, 2006. 1

[6] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, *LNCS* 1427. Springer, 1998. 4.2, 7, 9

[7] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proc. of POPL'02*. ACM, 2002. 1

[8] 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. 1

[9] 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. 1

[10] D. Gopan, T.W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Aarray Operations. In *POPL'05*. ACM, 2005. 1

[11] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *POPL'08*. ACM, 2008. 1

[12] P. Habermehl, R. Iosif, and T. Vojnar. A Logic of Singly Indexed Arrays. In *Proc. of LPAR'08*, *LNAI* 5330. Springer, 2008. (document), 1, 4, 2, 4, 4.1

[13] P. Habermehl, R. Iosif, and T. Vojnar. What Else is Decidable about Integer Arrays? In *Proc. of FoSSaCS'08*, *LNCS* 4962. Springer, 2008. 1

[14] N. Halbwachs and M. Péron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI'08*. ACM, 2008. 1

[15] R. Jhala and K. McMillan. Array Abstractions from Proofs. In *CAV'07*, *LNCS* 4590. Springer, 2007. 1

[16] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE'09*, LNCS. Springer, 2009. 1

[17] S.K. Lahiri and R.E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In *CAV'04*, *LNCS* 3114. Springer, 2004. 1

[18] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992. 5

[19] K. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS'08*, *LNCS* 4963. Springer, 2008. 1

[20] 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. 1

A Post-condition Calculus

In this section we give rules to obtain the strongest postcondition for **SIL** formulae expressing configurations of arrays w.r.t. statements of the program. We also explain how to decide verification conditions involving **SIL** formulae.

Assignements It is sufficient to consider basic assignements of the forms (1) $b = RHS$ where b is a scalar variable and RHS does not contain b and (2) $a[b+c] = RHS$ where a is an array, b a scalar variable, c a positive constant and RHS does not contain a. Other assignements can be simulated by several of these basic assignements.

Let us start with assignements of the form $b = RHS$. Then $post(\phi)$ is defined as

$$\exists b'.\phi[b'/b] \wedge b = RHS$$

where $\phi[b'/b]$ is the formula ϕ with all free occurrences of b replaced by b'.

We continue with assignements of the form $a[b+c] = RHS$. We suppose that the only occurences of b in ϕ are free. Then $post(\phi)$ is defined as

$$\exists a.\phi[a/a[b+c]] \wedge a[b+c] = RHS$$

where $\phi[a/a[b+c]]$ is the formula where all "occurrences" of $a[b+c]$ are replaced by a. Here an occurrence of $a[b+c]$ is an occurence of $a[t]$ where t is some index term which has potentially the value of $b+c$. We obtain $\phi[a/a[b+c]]$ in the following way:

- Replace ϕ by an equivalent formula ϕ' where occurrences of $a[b+c]$ are isolated. ϕ' is given as follows.
  - Replace all subformulae of the form $\psi = \forall j.G \rightarrow V$ in ϕ by $\psi' \wedge \psi''$ which is obtained as follows. Let T be the set of index terms accessing a in V. Then replace the guard G by $G \wedge \bigwedge_{t \in T} b+c \neq t$ to obtain ψ'. Let C be the set of constants appearing in T. Then ψ'' is given as $\bigwedge_{c' \in C}(G \rightarrow V)[(b+c-c')/j]$
- Now replace all occurrences of $a[b+c]$ in ϕ' by a.

Conditionals statements We consider a simple statement of the form $if\ \ COND_p\ \ ASGN_p$ $else\ \ ASGN'_p$. The other conditional statements can be handled in a very similar way.

Let φ be a **SIL** formula. Then the postcondition of the statement is given by the disjunction of the postcondition of $ASGN_p$ of $\varphi \wedge COND_p$ and of the postcondition of $ASGN'_p$ of $\varphi \wedge \neg COND_p$.

Closure under post-condition of the $\exists^*\forall^*$ fragment It is clear that if φ is a formula in $\exists^*\forall^*$-**SIL**, then its post-condition is in $\exists^*\forall^*$-**SIL** as well since only one existential quantifier is added by the post-condition computation.

Deciding verification conditions The user can specify a postcondition ψ in the $\forall^*$ fragment of **SIL**, i.e. all formulae which when written in prenex normal form have the quantifier prefix $\forall i_1 \ldots \forall i_m$, where $i_1, \ldots, i_m$ are index variables. ψ can contain freely occurring array and scalar variables.

Then, checking if a formula φ in $\exists^*\forall^*$**-SIL** obtained as a computed postcondition entails ψ is decidable since the validity of $\varphi \rightarrow \psi$ is equivalent to the non-satisfiability of $\varphi \wedge \neg\psi$. This formula is in $\exists^*\forall^*$**-SIL** whose satisifiability problem is decidable due to Theorem 2.

B Transducers Checking for Out-of-Bound Array References

In order to be able to check for *out-of-bound array references*, we may assume that the set of scalar variables used in the given program contains a variable b_a for each array variable $a \in AVar$, which stores the assumed length of the array a.[10] We can now extend the translation from loops to transducers, which we presented in Section 5.1, as follows. We translate a loop L of the form depicted in Figure 8 to the transducer $T_L = \langle X, Q, \{q_0\}, \rightarrow, \{q_{fin}, q_{err}\}\rangle$ defined below:

- $X = \{x_r^i, x_r^o, i_r \mid 1 \leq r \leq k\} \cup \{w_{r,l}^i \mid 1 \leq r \leq k, 1 \leq l \leq d_L(a_r)\} \cup \{w_{r,l}^o \mid 1 \leq r \leq k, 0 \leq l \leq d_L(a_r)\} \cup \{p_r^i, p_r^o, w_r \mid 1 \leq r \leq m\} \cup \{w_N\}$ where $x_r^{i/o}$, $1 \leq r \leq k$, are input/output array counters, $p_r^{i/o}$, $1 \leq r \leq k$, are parameters storing input/output scalar values, and w_r, $1 \leq r \leq m$, are working counters used for the manipulation of arrays and scalars (w_N stores the common length of arrays).

- $Q = \{q_0, q_{pre}, q_{loop}, q_{suf}, q_{fin}, q_{oob}, q_{err}\} \cup \{q_l^r \mid 1 \leq r \leq h, 0 \leq l < n_r\}$.

- 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:

  – $q_0 \xrightarrow{\varphi} q_{pre}$, $\varphi = \bigwedge_{1 \leq r \leq m}(w_r = p_r^i) \wedge w_N > 0 \wedge \bigwedge_{1 \leq r \leq k}(i_r = 0 \wedge x_r^i = w_{r,0}^o) \wedge \bigwedge_{\substack{1 \leq r \leq k \\ 1 \leq l \leq d_L(a_r)}}(w_{r,l}^i = w_{r,l}^o)$. The counters are initialized.

  – For each $\simeq_{\delta_L}$-equivalence class I_j, $1 \leq j \leq n$, $q_{pre} \xrightarrow{\varphi} q_{pre}$ with $\varphi = \bigwedge_{1 \leq r \leq k}(i_r < \xi(e_r)) \wedge \xi(incr(I))$. T_L copies the initial parts of the arrays untouched by L.

  – $q_{pre} \xrightarrow{\varphi} q_{loop}$, $\varphi = \bigwedge_{1 \leq r \leq k} i_r = \xi(e_r)$. T_L starts simulating L.

  – For each $1 \leq l \leq h$, $q_{loop} \xrightarrow{\varphi} q_0^l$ with $\varphi = \xi(C) \wedge \neg o(C) \wedge \bigwedge_{1 \leq r < l}(\neg\xi(C_r) \wedge \neg o(C_r)) \wedge \xi(C_l) \wedge \neg o(C_l)$ where $C_h = \top$. The automaton chooses a branch of the loop to simulate. Moreover, there are special transitions for the case of an out-of-bound access: For each $1 \leq l \leq h$ and each disjunct ψ of $o(C_l)$, $q_{loop} \xrightarrow{\varphi'} q_{oob}$ with $\varphi' = \xi(C) \wedge \bigwedge_{1 \leq r < l}(\neg\xi(C_r)) \wedge \psi \wedge \varphi_{out}$, and there are also transitions $q_{loop} \xrightarrow{\varphi''} q_{oob}$ where $\varphi'' = \psi \wedge \varphi_{out}$ for each disjunct ψ in $o(C)$. The formula $\varphi_{out} = \bigwedge_{1 \leq r \leq m}(w_r = p_r^o)$ defines the output values of scalar variables.

[10] Note that b_a is a program variable whose contents may differ from the uniform length of the simulated arrays w_N that we introduce in our model in order to deal with (possibly padded or, on the other hand, shortened) arrays having the same length. We can have a situation when $b_a < w_N$ meaning that the real array is padded in our model by cells which will not be used within a simulation of the loop. On the other hand, it may even be the case that $w_n < b_a$ which may happen when not all elements of a are really used in the loop (in which case they may effectively be left out within the simulation).

– For each $1 \leq l \leq h$, $1 \leq r \leq n_l$, $q^l_{r-1} \xrightarrow{\xi(S^l_r) \wedge \neg o(S^l_r)} q$ where $q = q^l_r$ if $r < n_l$ and $q = q_{loop}$ otherwise. The automaton simulates one branch of the loop. For the case of out-of-bound accesses, there are also transitions $q^l_{r-1} \xrightarrow{\psi \wedge \varphi_{out}} q_{oob}$ for each disjunct ψ in $o(S^l_r)$.

– $q_{loop} \xrightarrow{\neg\xi(C) \wedge \neg o(C) \wedge \varphi_{out}} q_{suf}$. The automaton finishes the simulation of the loop body.

– For each $\simeq_{\delta_L}$-equivalence class I_j, $1 \leq j \leq n$, $i_r \in I_j$, and $t \in \{suf, oob\}$, $q_t \xrightarrow{\varphi} q_t$ where $\varphi = i_r < w_N \wedge \xi(incr(I_j))$. These transitions copy the final parts of the arrays untouched by the loop. The copy is synchronous for dependent arrays. Note that we do not have to worry about the dependent arrays having different lengths b_a as T_L is now not really accessing the useful contents of the arrays, and the arrays are padded to the same total length w_N.

– For $(t, t') \in \{(suf, fin), (oob, err)\}$, $q_t \xrightarrow{\varphi} q_{t'}$ where $\varphi = \bigwedge_{1 \leq r \leq k} i_r = w_N$. All arrays are entirely processed.

In the above, we assume the same syntactical translation ξ of particular assignement statements, index increments, initialization expressions, and conditions as in Section 5.1. It remains to define the test for an out-of-bound access within a condition or assignement statement E: $o(E) = \bigvee_{1 \leq r \leq k . d_E(a_r) \neq \perp} i_p + d_E(a_r) \geq b_{a_r}$.

In order to be able to describe the effect of the above translation, let us assume that whenever an out-of-bound array reference happens in a run of a program, the control is transferred to a special terminal error line err without modifying the contents of the program variables in any way. Given a loop L, we then denote by $\Theta_{fin}(L)$ the set of pairs of states $\langle s, t \rangle$ such that there is a run of L from $\langle l_1, s \rangle$ to $\langle l_2, t \rangle$, $l_2 \neq err$, assuming that L starts at the line l_1 and l_2 is the line immediately following L. Moreover, let $\Theta_{err}(L)$ be the set of pairs of states $\langle s, t \rangle$ such that there is a run of L from $\langle l_1, s \rangle$ to $\langle err, t \rangle$. Further, given the transducer T_L derived for a loop L, let T_L^{fin} and T_L^{err} be the transducers obtained from T_L by restricting its set of final states to $\{q_{fin}\}$ or $\{q_{err}\}$, respectivelly. We can now give an alternative to Theorem 5 characterizing correctness of the extended construction.

Theorem 6 *Given a program loop L, the following hold:*

- T_L *is a transition-consistent transducer,*
- $\Theta_r(L) = \Theta(T_L^r)$ *for* $r \in \{fin, err\}$, *and*
- $\Delta(T_L) \rightarrow \delta_L$.

C Details of the Considered Case Studies

C.1 Array Partition

Input: An array a, parameter b_1 denoting the number of useful cells in a.
Data: b_2, b_3 – auxiliary variables
Output: Array a_2 contains non-negative elements, array a_3 contains negative elements.

```
/* φ : b_1 ≥ 0                                                    */
b_2 := 0;
b_3 := 0;
while _{a_1:i_1=0, a_2:i_2=0, a_3:i_3=0}(i_1 < b_1) do
    if a_1[i_1] ≥ 0 then
        a_2[i_2] := a_1[i_1];
        b_2 := b_2 + 1;
        i_1 + +;
        i_2 + +;
    else
        a_3[i_3] := a_1[i_1];
        b_3 := b_3 + 1;
        i_1 + +;
        i_3 + +;
    end
end
/* ψ : ∀i.(0 ≤ i ≤ b_2 − 1) ⇒ (a_2[i] ≥ 0) ∧ ∀i.(0 ≤ i ≤ b_3 − 1) ⇒ (a_3[i] < 0)   */
```

Algorithm 1: Array partition program

Formula φ_{in} expressing the program state at the point where the program enters the loop is constructed according to the postcondition calculus, as described in appendix A. This formula is then translated to a counter automaton A_{in} according to section 4.1. Then, a loop transducer A_L is constructed in accordance with section 5.1. The automaton A_{out} describing a program state at the point where the program leaves the loop is created according to lemma 4.

In the split example, we have

$$\varphi_{in} = v_1^i \geq 0 \wedge v_2^i = 0 \wedge v_3^i = 0,$$

$$A_{in} = (\{v_1^i, v_1^i, v_1^i\}, \{q_i, q_f\}, \{q_i\}, \rightarrow, \{q_f\}), \text{ where}$$

$$\rightarrow = \{q_i \xrightarrow{v_1^i \geq 0 \wedge v_2^i = 0 \wedge v_3^i = 0 \wedge const(v_1^i, v_1^i, v_1^i)} q_f, q_f \xrightarrow{const(v_1^i, v_1^i, v_1^i)} q_f\}.$$

Automaton A_{out} is depicted in figure C.1. We don't show the automaton A_L as its structure is exactly the same as in A_{out} and labeling of A_L's transitions is a subset of A_{out}'s transitions. Notably, guards specifying post-image value counters (y_1, y_2, y_3) are not present in A_L. Correspondence between program variables and counters in A_{out} is given in table C.1.

Figure 9: Array partition - correspondence between program variables and counters

array variable	index counter	val. counter at loop exit	val. counter at loop entry	transducer output
a_1	i_1	y_1	x^i_1	x^o_1
a_2	i_2	y_2	x^i_2	x^o_1
a_3	i_3	y_3	x^i_3	x^o_1

scalar variable	parameter at loop exit	parameter at loop entry
b_1	v_1	v^i_1
b_2	v_2	v^i_2
b_3	v_3	v^i_3

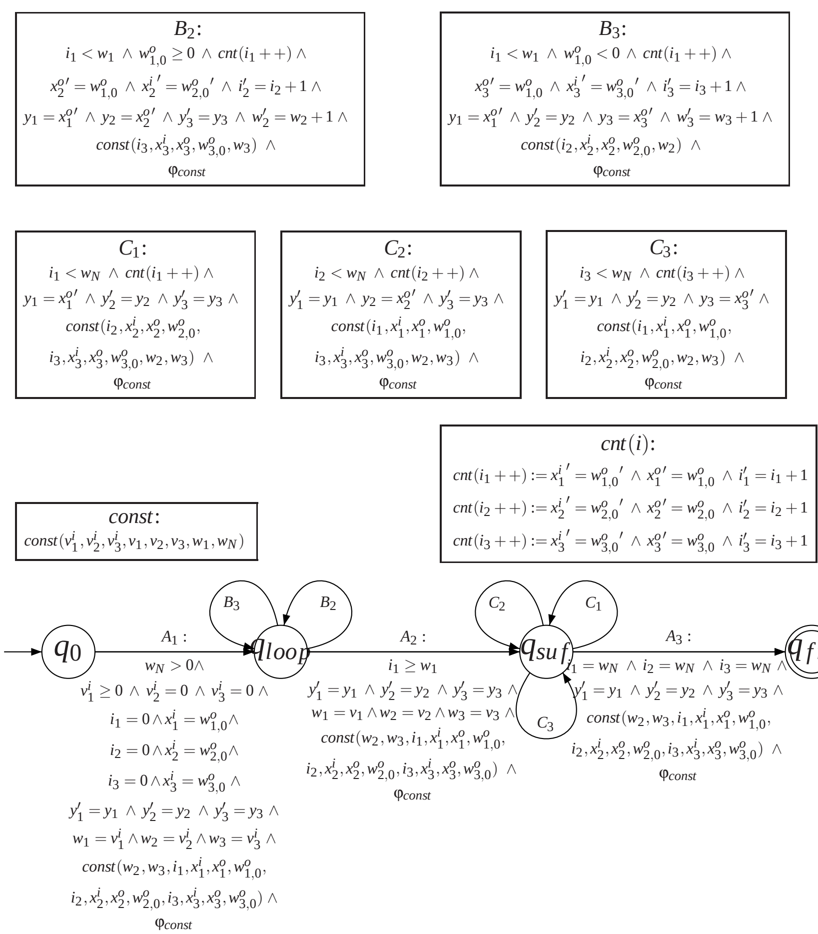


Figure 10: Array partition - loop post-image represented by A_{out}

C.1.1 From CA to SIL

Arrays a_1, a_2, a_3 are independent.

Let us focus on computing the abstraction w.r.t. a_2 equivalence class. The case for a_3 is similar (and the case of a_1 is simpler).

The abstraction from a the non-flat CA to a flat CA with DBC constraints is schematized in figure C.1.1.

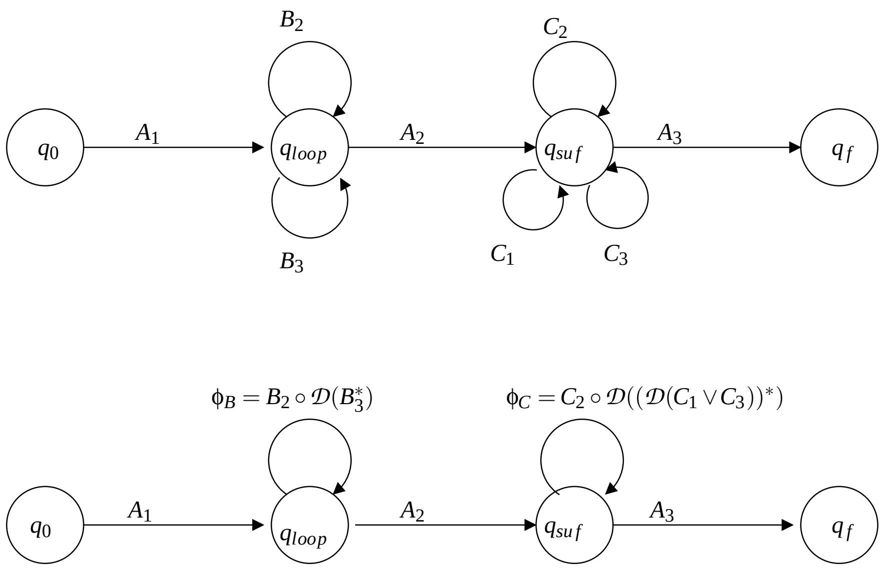


Figure 11: Array partition - abstraction of the post-image

C.1.2 State q_{loop}

Remark that relations are *partitioned* i.e, the set of counters restricted on every line are disjoint.

$$
\begin{array}{rcl}
B_2 & : & i_1 < w_1 \wedge i_1' = i_1 + 1 \wedge w_1' = w_1 \\
& \wedge & w_{1,0}^o >= 0 \wedge {x_1^i}' = {w_{1,0}^o}' \wedge x_1^{o\prime} = w_{1,0}^o \wedge x_2^{o\prime} = w_{1,0}^o \wedge y_1 = x_1^{o\prime} \wedge y_2 = x_2^{o\prime} \\
& \wedge & {x_2^i}' = {w_{2,0}^o}' \\
& \wedge & i_2' = i_2 + 1 \\
& \wedge & w_2' = w_2 + 1 \\
& \wedge & Id(w_{3,0}^o, x_3^o, x_3^i, y_3, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_3, i_3, w_N)
\end{array}
$$

$$
\begin{array}{rl}
B_3 \quad : & i_1 < w_1 \wedge i_1' = i_1 + 1 \wedge w_1' = w_1 \\
\wedge & w_{1,0}^o < 0 \wedge x_1^{i\,\prime} = w_{1,0}^{o\;\prime} \wedge x_1^{o\prime} = w_{1,0}^o \wedge x_3^{o\prime} = w_{1,0}^o \wedge y_1 = x_1^{o\prime} \wedge y_3 = x_3^{o\prime} \\
\wedge & x_3^{i\,\prime} = w_{3,0}^{o\;\prime} \\
\wedge & i_3' = i_3 + 1 \\
\wedge & w_3' = w_3 + 1 \\
\wedge & Id(w_{2,0}^o, x_2^o, x_2^i, y_2, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_2, i_2, w_N)
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{D}(B_3^*) \quad : & i_1' \geq i_1 \wedge w_1' = w_1 \\
\wedge & \top \\
\wedge & \top \\
\wedge & i_3' \geq i_3 \\
\wedge & w_3' \geq w_3 \\
\wedge & Id(w_{2,0}^o, x_2^o, x_2^i, y_2, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_2, i_2, w_N)
\end{array}
$$

$$
\begin{array}{rl}
\phi_B = B_2 \circ \mathcal{D}(B_3^*) \quad : & i_1' \geq i_1 + 1 \wedge i_1 < w_1 \wedge w_1' = w_1 \\
\wedge & i_2' = i_2 + 1 \\
\wedge & w_2' = w_2 + 1 \\
\wedge & w_{1,0}^o \geq 0 \wedge x_2^{o\prime} = w_{1,0}^o = y_1 = y_2 \\
\wedge & x_2^{i\,\prime} = w_{2,0}^{o\;\prime} \\
\wedge & i_3' \geq i_3 \\
\wedge & w_3' \geq w_3 \\
\wedge & Id(v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_N)
\end{array}
$$

$$
\mathcal{T}(\exists \mathbf{i} \exists \mathbf{x}_r \exists \mathbf{w} \,.\, \phi_B) \quad : \quad \mathcal{T}(y_2 \geq 0) = a_2[\mathrm{j}] \geq 0
$$

$$
\begin{array}{rl}
\phi_B^n \quad : & i_1' \geq i_1 + n \wedge i_1 < w_1 \wedge w_1' = w_1 \wedge \\
& i_2' = i_2 + n \wedge w_2' = w_2 + n \wedge \\
& x_2^{i\,\prime} = w_{2,0}^o \wedge \\
& i_3' \geq i_3 + n \wedge w_3' \geq w_3 + n \wedge Id(v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_N)
\end{array}
$$

$$
\mathcal{T}(\exists \mathbf{i} \exists \mathbf{x} \exists \mathbf{w} \,.\, (A_1 \circ \phi_B^n \circ A_2)) \quad : \quad \mathcal{T}(v_2 = n) = b_2 = n
$$

C.1.3 State q_{suf}

$$\begin{aligned}
C_1 &= i_1 < w_N \wedge i_1' = i_1 + 1 \wedge w_N' = w_N \\
&\wedge \; {x_1^i}' = {w_{1,0}^o}' \wedge x_1^{o\prime} = w_{1,0}^o \wedge y_1 = x_1^{o\prime} \\
&\wedge \; Id(y_2, y_3, i_2, i_3, w_{2,0}^o, x_2^o, x_2^i, w_{3,0}^o, x_3^o, x_3^i, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_1, w_2, w_3)
\end{aligned}$$

$$\begin{aligned}
C_2 &= i_2 < w_N \wedge i_2' = i_2 + 1 \wedge w_N' = w_N \\
&\wedge \; {x_2^i}' = {w_{2,0}^o}' \wedge x_2^{o\prime} = w_{2,0}^o \wedge y_2 = x_2^{o\prime} \\
&\wedge \; Id(y_1, y_3, i_1, i_3, w_{1,0}^o, x_1^o, x_1^i, w_{3,0}^o, x_3^o, x_3^i, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_1, w_2, w_3)
\end{aligned}$$

$$\begin{aligned}
C_3 &= i_3 < w_N \wedge i_3' = i_3 + 1 \wedge w_N' = w_N \\
&\wedge \; {x_3^i}' = {w_{3,0}^o}' \wedge x_3^{o\prime} = w_{3,0}^o \wedge y_3 = x_3^{o\prime} \\
&\wedge \; Id(y_2, y_1, i_2, i_1, w_{2,0}^o, x_2^o, x_2^i, w_{1,0}^o, x_1^o, x_1^i, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_1, w_2, w_3)
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}(C_1 \vee C_3) &= i_1 \leq i_1' \leq i_1 + 1 \\
&\wedge \; i_3 \leq i_3' \leq i_3 + 1 \\
&\wedge \; w_N' = w_N \\
&\wedge \; Id(y_2, i_2, w_{2,0}^o, x_2^o, x_2^i, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_1, w_2, w_3)
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}((\mathcal{D}(C_1 \vee C_3))^*) &= i_1 \leq i_1' \\
&\wedge \; i_3 \leq i_3' \\
&\wedge \; w_N' = w_N \\
&\wedge \; Id(y_2, i_2, w_{2,0}^o, x_2^o, x_2^i, v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_1, w_2, w_3)
\end{aligned}$$

$$\begin{aligned}
\phi_C = C_2 \circ \mathcal{D}((\mathcal{D}(C_1 \vee C_3))^*) &= i_2' = i_2 + 1 \wedge i_2 < w_N \wedge w_N' = w_N \\
&\wedge \; {x_2^i}' = {w_{2,0}^o}' \wedge w_{2,0}^o = x_2^{o\prime} = y_2 \\
&\wedge \; i_1' \geq i_1 \\
&\wedge \; i_3' \geq i_3 \\
&\wedge \; Id(v_1^i, v_2^i, v_3^i, v_1, v_2, v_3, w_1, w_2, w_3)
\end{aligned}$$

C.1.4 The SIL formula

The formula is $\exists \mathfrak{n} \forall \mathfrak{j} \,.\, 0 \leq \mathfrak{j} < \mathfrak{n} \rightarrow a_2[\mathfrak{j}] \geq 0 \wedge b_2 = \mathfrak{n}$. This can be further simplied to $\forall \mathfrak{j} \,.\, 0 \leq \mathfrak{j} < b_2 \rightarrow a_2[\mathfrak{j}]$. Notice that $K = 0$ here, which simplifies things a lot. For i_3 we obtain in a similar way $\forall \mathfrak{j} \,.\, 0 \leq \mathfrak{j} < b_3 \rightarrow a_3[\mathfrak{j}]$.

C.2 Insertion of an Element

Input: A sorted array a, parameter b_1 denoting the number of used cells (see precondition) in the input array a, b_2 is an element to insert.
Data: b_3, b_4 – auxiliary variables
Output: Element b_2 is inserted into a (hence the number of useful cells in a increases by one), the value of a variable b_5 is an index of a where b_2 was inserted. The scalar variable b_5 has a value of an index where b_2 was inserted.

/* $\psi_{pre} : \forall i.(0 \leq i \leq b_1 - 2) \Rightarrow (a[i] \leq a[i+1]) \wedge 0 \leq b_1$ */

$b_3 := b_2$;
$b_5 := 0$;
while ${}_{a:i=0}(i < b_1)$ **do**
  if $(a[i] \leq b_2)$ **then**
    $b_5 := b_5 + 1$;
    $i++$;
  else
    $b_4 := a[i]$;
    $a[i] := b_3$;
    $b_3 := b_4$;
    $i++$;
  end
end
$a[b_1] := b_3$;
b_1++;
/* $\psi_{post} :\ 0 \leq b_5 \leq d_0 \ \wedge\ \forall i.(0 \leq i \leq b_1 - 2) \Rightarrow (a[i] \leq a[i+1]) \ \wedge$
$\forall i.(b_5 \leq i \leq b_5) \Rightarrow (a[i] = b_2])$ */

Algorithm 2: Insertion of an element into a sorted array

Formula φ_{in} expressing the program state at the point where the program enters the loop is constructed according to the postcondition calculus, as described in appendix A.

$$\varphi_{in} := \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4, \text{ where } \varphi_1 := \forall i.(0 \leq i \leq b_1 - 2) \Rightarrow (a[i] \leq a[i+1]),$$

$$\varphi_2 := 0 \leq b_1,\ \varphi_3 := b_3 = b_2,\ \varphi_4 := b_5 = 0$$

This formula is then translated to a counter automaton A_{in} according to a construction described in section 4.1. So, for each formula φ_i, $i \in \{1,2,3,4\}$, we construct an automaton A_{φ_i}. A counter transducer A_L is constructed for the program loop (in accordance with section 5.1). The automata representation of the post-image of the φ_{in} and φ_{loop} is constructed by making product of all A_{φ_i}, $i \in \{1,2,3,4\}$ and A_L and by adding new value counters (denoted by y in examples) in accordance with section 4.1.

It is convenient to make the product of the automata describing presburger constraints and of the loop transducer first (we can prevent some redundant guards to appear in the post-image automaton by this). By this, we obtain $A_L{}' = A_{\varphi_2} \otimes A_{\varphi_3} \otimes A_{\varphi_4} \otimes A_L$ and then we make a product $A_{out} = A_1 \otimes A_L{}'$ to obtain the post-image. For A_1, see figure C.2, for A_L figure C.2 and for A_L^{out} figure C.2.

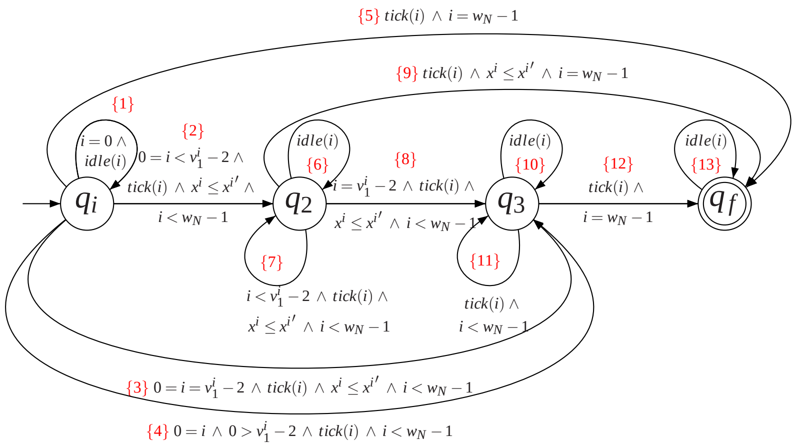


Figure 12: Counter automaton A_1 representing the forall subformula of the precondition.

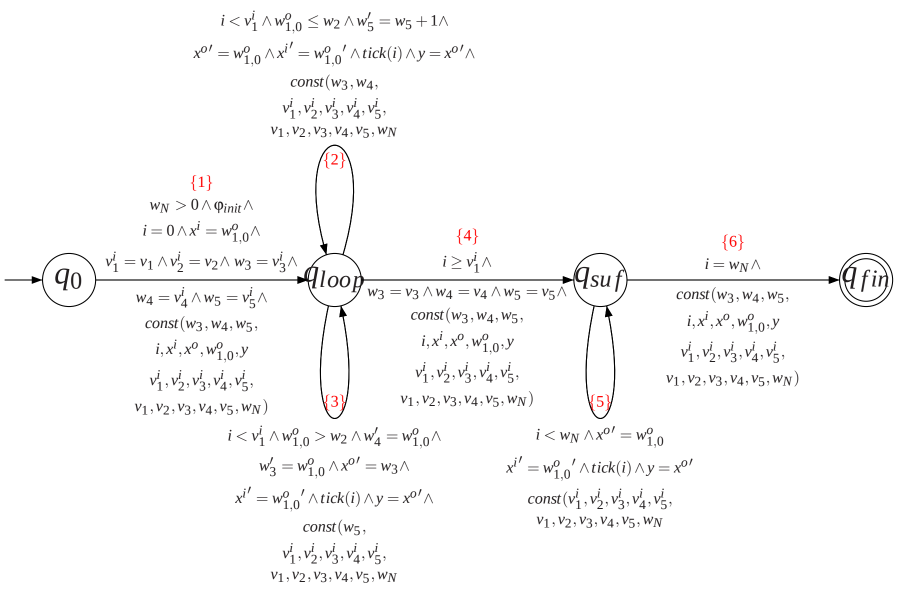


Figure 13: Product of a loop counter transducer A_L with automata A_2, A_3 and A_4 (note: $\varphi_{init} := 0 \le v_1^i \wedge v_3^i = v_2^i \wedge v_5^i = 0$).

Automaton A_{out} can be simplified further. Firstly, every non-loop, non-initial and non-final state can be eliminated by composing its incoming and outcoming transitions. Formally, let $in(s)$ ($out(s)$) be a set of all incoming (outcoming) transitions of state s. Every state s which is neither initial nor final nor it contains any loops can be eliminated by composing every $t \in in(s)$ with every $t' \in out(s)$. In our framowork, this elimination can be applied to states where all incoming transitions are idle or where all outcoming transitions are idle (otherwise we would lose state-consistency). With this additional condition, this simplification can be applied to states (q_i, q_{loop}), (q_f, q_{loop}), (q_f, q_{suf}) in the insert example.

Secondly, every non-trivial strongly connected component *scc* of the automaton can be analysed by constructing a dependency graph of its transitions. This technique may discover that loops (or transitions, generally) can be performed only in some order. If this leads to simplification of a *scc* (e.g. *scc* becomes flat), then this *scc* is replaced by the dependency graph. In the insert example, this is the case of state (q_2, q_{loop}), where the dependency analysis discovers that a transition $\varphi_{(7,3)}$ can never be followed by $\varphi_{(7,2)}$ (analogically for state (q_3, q_{loop}) and transitions $\varphi_{(11,3)}$ and $\varphi_{(11,2)}$).

Both previous optimizations have been implemented in the FLATA library. The result of this optimization, an automaton $\varphi_{out}{}'$, can be seen in figure C.2. Moreover, by a simple analysis of the loop, it can be found out that variables b_1 and b_2 are never assigned, so the loop transducer does not have to use auxiliary counters w_1 and w_2.

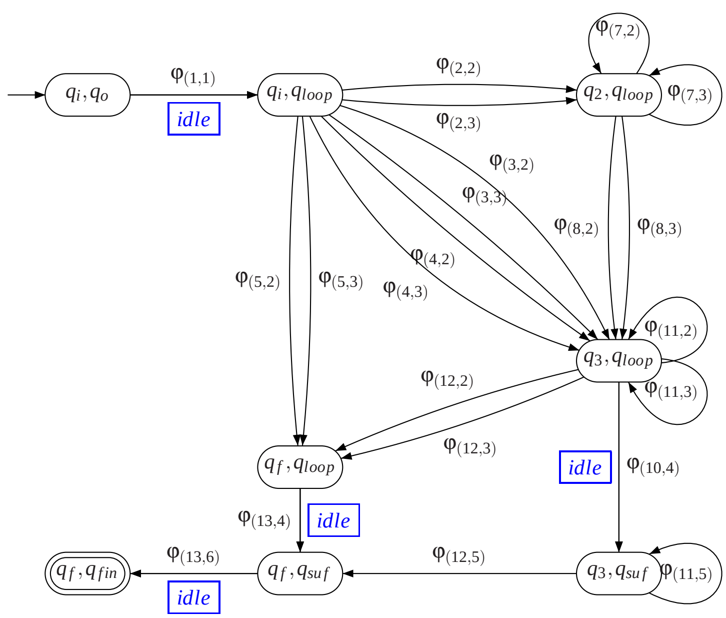


Figure 14: Insert example, the post-image automaton A_{out} (idle transitions are marked, the rest are ticks).

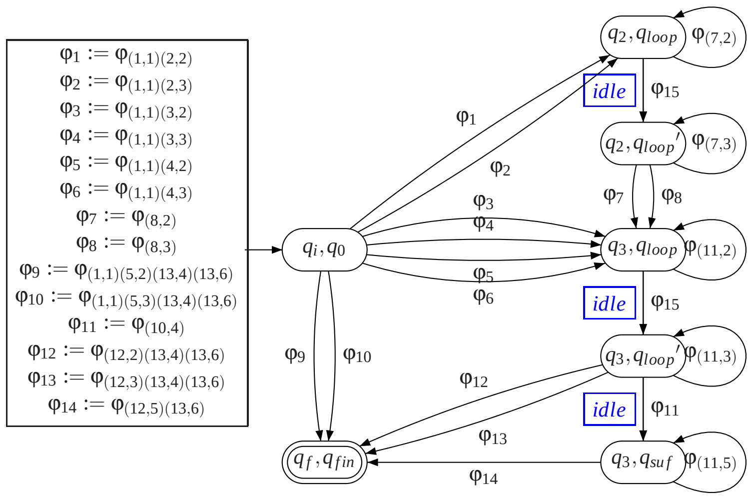


Figure 15: Simplification $A_{out}{}'$ of the post-image automaton A_{out}.

Definitions of transition labels:

$$
\begin{array}{lll}
\varphi_1 & : & 0 = i < v^i_1 - 2 \ \wedge \ x^i \leq (x^i)' \ \wedge \ i < w_N - 1 \\
& \wedge & w_N > 0 \wedge 0 \leq v^i_1 \wedge v^i_3 = v^i_2 \wedge v^i_5 = 0 \wedge x^i = w^o_{1,0} \wedge v^i_1 = v_1 \wedge v^i_2 = v_2 \wedge w_3 = v^i_3 \wedge w_4 = v^i_4 \wedge w_5 = v^i_5 \\
& \wedge & w^o_{1,0} \leq w_2 \wedge w'_5 = w_5 + 1 \wedge x^{o\prime} = w^o_{1,0} \wedge x^{i\prime} = w^o_{1,0}{}' \wedge tick(i) \wedge y = x^{o\prime} \\
& \wedge & const(w_3, w_4, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{array}
$$

$$
\begin{array}{lll}
\varphi_2 & : & 0 = i < v^i_1 - 2 \ \wedge \ x^i \leq (x^i)' \ \wedge \ i < w_N - 1 \\
& \wedge & w_N > 0 \wedge 0 \leq v^i_1 \wedge v^i_3 = v^i_2 \wedge v^i_5 = 0 \wedge x^i = w^o_{1,0} \wedge v^i_1 = v_1 \wedge v^i_2 = v_2 \wedge w_3 = v^i_3 \wedge w_4 = v^i_4 \wedge w_5 = v^i_5 \\
& \wedge & w^o_{1,0} > w_2 \wedge w'_4 = w^o_{1,0} \wedge w'_3 = w^o_{1,0} \wedge x^{o\prime} = w_3 \wedge x^{i\prime} = w^o_{1,0}{}' \wedge tick(i) \wedge y = x^{o\prime} \\
& \wedge & const(w_5, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{array}
$$

$$
\begin{array}{lll}
\varphi_3 & : & 0 = i = v^i_1 - 2 \ \wedge \ x^i \leq (x^i)' \ \wedge \ i < w_N - 1 \\
& \wedge & w_N > 0 \wedge 0 \leq v^i_1 \wedge v^i_3 = v^i_2 \wedge v^i_5 = 0 \wedge x^i = w^o_{1,0} \wedge v^i_1 = v_1 \wedge v^i_2 = v_2 \wedge w_3 = v^i_3 \wedge w_4 = v^i_4 \wedge w_5 = v^i_5 \\
& \wedge & w^o_{1,0} \leq w_2 \wedge w'_5 = w_5 + 1 \wedge x^{o\prime} = w^o_{1,0} \wedge x^{i\prime} = w^o_{1,0}{}' \wedge tick(i) \wedge y = x^{o\prime} \\
& \wedge & const(w_3, w_4, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{array}
$$

$$
\begin{aligned}
\varphi_4 \quad : \quad & 0 = i = v_1^i - 2 \ \wedge \ x^i \leq (x^i)' \ \wedge \ i < w_N - 1 \\
\wedge \quad & w_N > 0 \wedge 0 \leq v_1^i \wedge v_3^i = v_2^i \wedge v_5^i = 0 \wedge x^i = w_{1,0}^o \wedge v_1^i = v_1 \wedge v_2^i = v_2 \wedge w_3 = v_3^i \wedge w_4 = v_4^i \wedge w_5 = v_5^i \\
\wedge \quad & w_{1,0}^o > w_2 \wedge w_4' = w_{1,0}^o \wedge w_3' = w_{1,0}^o \wedge x^{o\prime} = w_3 \wedge x^{i\prime} = {w_{1,0}^o}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_5, v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_5 \quad : \quad & 0 = i \ \wedge \ v_1^i = 1 \ \wedge \ i < w_N - 1 \\
\wedge \quad & w_N > 0 \wedge v_3^i = v_2^i \wedge v_5^i = 0 \wedge i = 0 \wedge x^i = w_{1,0}^o \wedge v_1^i = v_1 \wedge v_2^i = v_2 \wedge w_3 = v_3^i \wedge w_4 = v_4^i \wedge w_5 = v_5^i \\
\wedge \quad & w_{1,0}^o \leq w_2 \wedge w_5' = w_5 + 1 \wedge x^{o\prime} = w_{1,0}^o \wedge x^{i\prime} = {w_{1,0}^o}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_3, w_4, v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_6 \quad : \quad & 0 = i \ \wedge \ v_1^i = 1 \ \wedge \ i < w_N - 1 \\
\wedge \quad & w_N > 0 \wedge v_3^i = v_2^i \wedge v_5^i = 0 \wedge i = 0 \wedge x^i = w_{1,0}^o \wedge v_1^i = v_1 \wedge v_2^i = v_2 \wedge w_3 = v_3^i \wedge w_4 = v_4^i \wedge w_5 = v_5^i \\
\wedge \quad & w_{1,0}^o > w_2 \wedge w_4' = w_{1,0}^o \wedge w_3' = w_{1,0}^o \wedge x^{o\prime} = w_3 \wedge x^{i\prime} = {w_{1,0}^o}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_5, v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_7 \quad : \quad & i = v_1^i - 2 \ \wedge \ x^i \leq (x^i)' \ \wedge \ i < w_N - 1 \\
\wedge \quad & w_{1,0}^o \leq w_2 \wedge w_5' = w_5 + 1 \wedge x^{o\prime} = w_{1,0}^o \wedge x^{i\prime} = {w_{1,0}^o}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_3, w_4, v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_8 \quad : \quad & i = v_1^i - 2 \ \wedge \ x^i \leq (x^i)' \ \wedge \ i < w_N - 1 \\
\wedge \quad & w_{1,0}^o > w_2 \wedge w_4' = w_{1,0}^o \wedge w_3' = w_{1,0}^o \wedge x^{o\prime} = w_3 \wedge x^{i\prime} = {w_{1,0}^o}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_5, v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_9 \quad : \quad & 0 = i = w_N - 1 \\
\wedge \quad & w_N > 0 \wedge 0 \leq v_1^i \wedge v_3^i = v_2^i \wedge v_5^i = 0 \wedge x^i = w_{1,0}^o \wedge v_1^i = v_1 \wedge v_2^i = v_2 \wedge w_3 = v_3^i \wedge w_4 = v_4^i \wedge w_5 = v_5^i \\
\wedge \quad & i = v_1^i - 1 \wedge w_3' = v_3 \wedge w_4' = v_4 \wedge w_5' = v_5 \wedge \\
\wedge \quad & w_{1,0}^o \leq w_2 \wedge w_5' = w_5 + 1 \wedge x^{o\prime} = w_{1,0}^o \wedge x^{i\prime} = {w_{1,0}^o}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_3, w_4, v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{10} \quad : \quad & 0 = i = w_N - 1 \\
\wedge \quad & w_N > 0 \wedge 0 \leq v^i_1 \wedge v^i_3 = v^i_2 \wedge v^i_5 = 0 \wedge x^i = w^o_{1,0} \wedge v^i_1 = v_1 \wedge v^i_2 = v_2 \wedge w_3 = v^i_3 \wedge w_4 = v^i_4 \wedge w_5 = v^i_5 \\
\wedge \quad & i = v^i_1 - 1 \wedge w'_3 = v_3 \wedge w'_4 = v_4 \wedge w'_5 = v_5 \wedge \\
\wedge \quad & w^o_{1,0} > w_2 \wedge w'_4 = w^o_{1,0} \wedge w'_3 = w^o_{1,0} \wedge x^{o\prime} = w_3 \wedge x^{i\prime} = {w^o_{1,0}}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_5, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{11} \quad : \quad & i \geq v^i_1 \wedge w_3 = v_3 \wedge w_4 = v_4 \wedge w_5 = v_5 \wedge \\
\wedge \quad & const(w_3, w_4, w_5, i, x^i, x^o, w^o_{1,0}, y, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

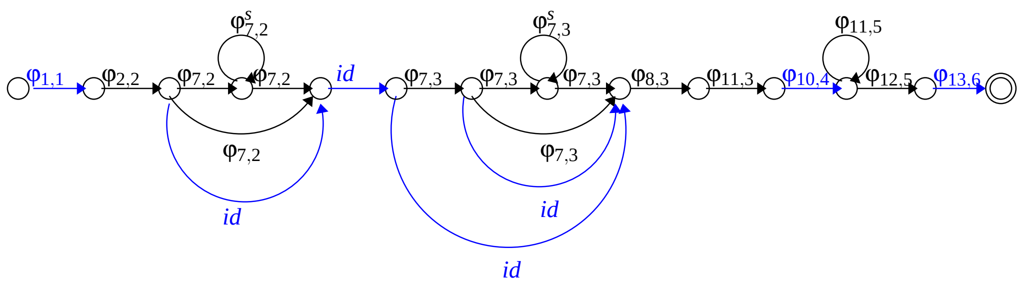
$$
\begin{aligned}
\varphi_{12} \quad : \quad & i = w_N - 1 \\
\wedge \quad & i = v^i_1 - 1 \wedge w'_3 = v_3 \wedge w'_4 = v_4 \wedge w'_5 = v_5 \wedge \\
\wedge \quad & w^o_{1,0} \leq w_2 \wedge w'_5 = w_5 + 1 \wedge x^{o\prime} = w^o_{1,0} \wedge x^{i\prime} = {w^o_{1,0}}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_3, w_4, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{13} \quad : \quad & i = w_N - 1 \\
\wedge \quad & i = v^i_1 - 1 \wedge w'_3 = v_3 \wedge w'_4 = v_4 \wedge w'_5 = v_5 \wedge \\
\wedge \quad & w^o_{1,0} > w_2 \wedge w'_4 = w^o_{1,0} \wedge w'_3 = w^o_{1,0} \wedge x^{o\prime} = w_3 \wedge x^{i\prime} = {w^o_{1,0}}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(w_5, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{14} \quad : \quad & i = w_N - 1 \\
\wedge \quad & x^{o\prime} = w^o_{1,0} \wedge x^{i\prime} = {w^o_{1,0}}' \wedge tick(i) \wedge y = x^{o\prime} \\
\wedge \quad & const(v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
\end{aligned}
$$

$$
\varphi_{15} \quad : \quad const(w_3, w_4, w_5, i, x^i, x^o, w^o_{1,0}, y, v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v_1, v_2, v_3, v_4, v_5)
$$

C.2.1 Postcondition computation



Where
$\varphi^s_{7,2} = \varphi_{7,2} \wedge x^i = w^o_{1,0} \wedge y' = {w^o_{1,0}}'$ (and implies $y \leq y'$)
$\varphi^s_{7,3} = \varphi_{7,3} \wedge w_4 = w_3 \leq x^i = w^o_{1,0} \wedge y' = w'_3$ (and implies $y \leq y'$)

C.3 Array Rotation

Input: An array a_1, parameter b_1 denoting the number of used cells in a_1.
Data: Auxiliary array a_2, which used for specification of postcondition, keeps a copy of a_1, auxiliary scalar variable b_2 holding value of the first element of a_1.
Output: Array a_1 is rotated by one to the left.

/* $\psi_{pre}: \; b_1 > 0 \; \wedge \; \forall i.(0 \leq i \leq b_1 - 1) \Rightarrow (a_1[i] = a_2[i])$ */

$b_2 := a_1[0];$
while $_{a:i=0}(i \leq b_1 - 2)$ **do**
  $a_1[i] := a_1[i+1];$
  $i++;$
end
$a_1[b_1 - 1] := b_2;$

/* $\psi_{post}: \; \forall i.(0 \leq i \leq b_1 - 2) \Rightarrow (a_1[i] = a_2[i+1]) \; \wedge$
$\forall i.(b_1 - 1 \leq i \leq b_1 - 1) \Rightarrow (a_1[i] = b_2)$ */

Algorithm 3: Array rotation

A loop post-image counter automaton is constructed in a same manner as in split and insert examples. By a simple analysis of the loop, it can be found out that variable b_1 is never assigned, so the loop transducer does not have to use auxiliary counter w_1. Furthermore, we have to deal with variables which are not used in the loop in this example:

- scalar variables – scalar variable b_2 is not used in the loop, the product is modified in the way that the initial transition contains a $v_2^i = v_2$ constraint
- array variables – array variable a_2 is not used in the loop, the product automaton is modified in the way that every transition contains a $x_2^i = y_2$ constraint. (i_1 and i_2 are dependent in the precondition) and i_2 ticks if and only if i_1 ticks.

$$
\begin{aligned}
\varphi_1 \quad : \quad & 0 = \{i_1, i_2\} = w_N - 1 = v_1^i - 2 \\
\wedge \quad & x_1^i = x_2^i \wedge x_1^i = v_2^i \\
\wedge \quad & w_N > 0 \wedge v_1^i = v_1 \wedge v_2^i = v_2 \\
\wedge \quad & x_1^i = w_{1,0}^o \wedge w_{1,1}^o = w_{1,1}^i \\
\wedge \quad & x_1^{o\prime} = w_{1,1}^o \wedge x_1^{i\prime} = w_{1,1}^i \wedge w_{1,0}^{o\prime} = w_{1,1}^o \wedge w_{1,1}^{o\prime} = w_{1,1}^{i\prime} \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

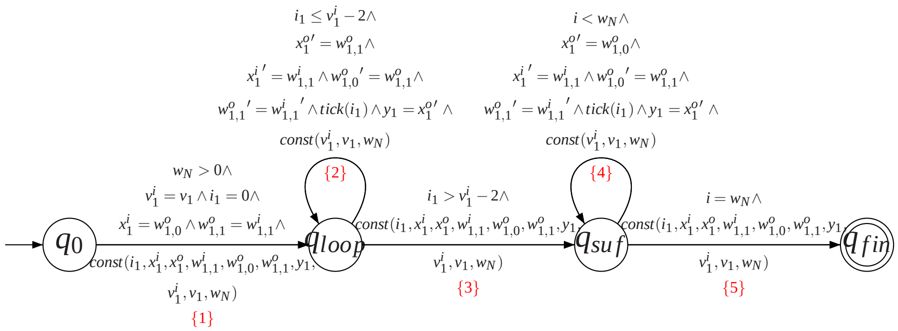


Figure 16: Loop automaton A_L

$$
\begin{aligned}
\varphi_2 \quad : \quad & 0 = \{i_1, i_2\} = w_N - 1 = v_1^i - 1 \\
\wedge \quad & x_1^i = x_2^i \wedge x_1^i = v_2^i \\
\wedge \quad & w_N > 0 \wedge v_1^i = v_1 \wedge v_2^i = v_2 \\
\wedge \quad & x_1^i = w_{1,0}^o \wedge w_{1,1}^o = w_{1,1}^i \\
\wedge \quad & x_1^{o\prime} = w_{1,0}^o \wedge {x_1^i}' = w_{1,1}^i \wedge {w_{1,0}^o}' = w_{1,1}^o \wedge {w_{1,1}^o}' = {w_{1,1}^i}' \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_3 \quad : \quad & 0 = \{i_1, i_2\} < w_N - 1 \wedge \{i_1, i_2\} < v_1^i - 1 \\
\wedge \quad & x_1^i = x_2^i \wedge x_1^i = v_2^i \\
\wedge \quad & w_N > 0 \wedge v_1^i = v_1 \wedge v_2^i = v_2 \\
\wedge \quad & x_1^i = w_{1,0}^o \wedge w_{1,1}^o = w_{1,1}^i \\
\wedge \quad & x_1^{o\prime} = w_{1,1}^o \wedge {x_1^i}' = w_{1,1}^i \wedge {w_{1,0}^o}' = w_{1,1}^o \wedge {w_{1,1}^o}' = {w_{1,1}^i}' \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_4 \quad : \quad & 0 = \{i_1, i_2\} = v_1^i - 1 < w_N - 1 \\
\wedge \quad & x_1^i = x_2^i \wedge x_1^i = v_2^i \\
\wedge \quad & w_N > 0 \wedge v_1^i = v_1 \wedge v_2^i = v_2 \\
\wedge \quad & x_1^i = w_{1,0}^o \wedge w_{1,1}^o = w_{1,1}^i \\
\wedge \quad & x_1^{o\prime} = w_{1,0}^o \wedge {x_1^i}' = w_{1,1}^i \wedge {w_{1,0}^o}' = w_{1,1}^o \wedge {w_{1,1}^o}' = {w_{1,1}^i}' \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

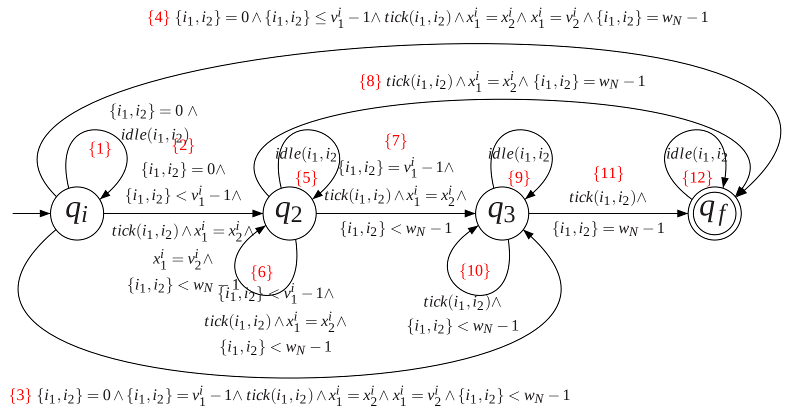


Figure 17: Counter automaton A_1 representing the forall subformula of the precondition (optimized by $b_1 > 0$ formula – 2 transitions were be removed).

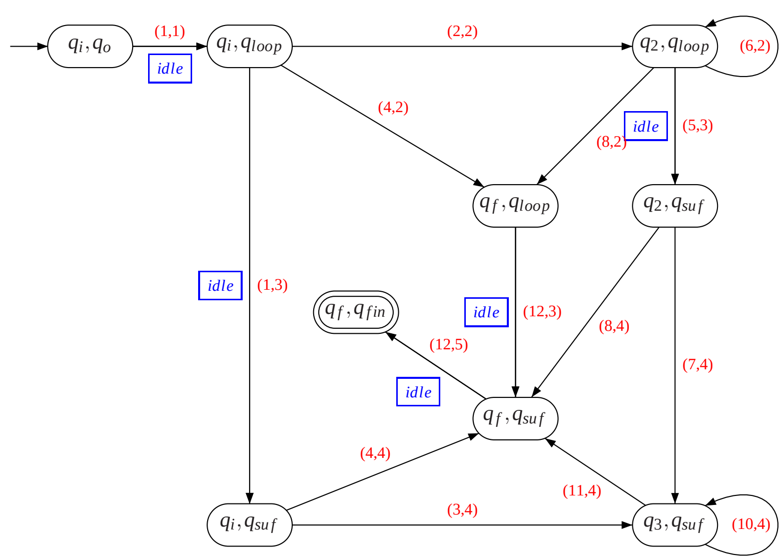


Figure 18: Rotation example, the product automaton (idle transitions are marked, the rest are ticks)

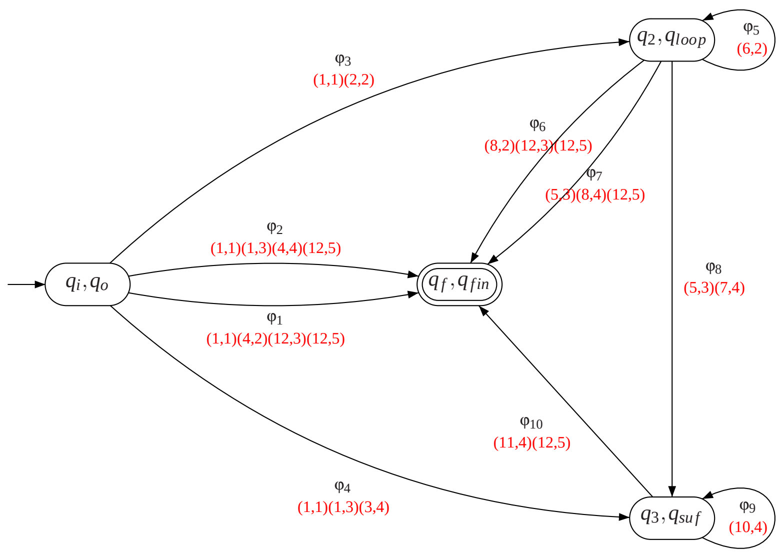


Figure 19: Rotation example, the simplified product automaton (idle transitions composed)

$$
\begin{aligned}
\varphi_5 \quad : \quad & \{i_1, i_2\} < w_N - 1 \wedge \{i_1, i_2\} < v_1^i - 1 \\
\wedge \quad & x_1^i = x_2^i \\
\wedge \quad & x_1^{o\prime} = w_{1,1}^o \wedge x_1^{i\,\prime} = w_{1,1}^i \wedge w_{1,0}^{o\,\prime} = w_{1,1}^o \wedge w_{1,1}^{o\,\prime} = w_{1,1}^{i\,\prime} \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_6 \quad : \quad & \{i_1, i_2\} = w_N - 1 = v_1^i - 2 \\
\wedge \quad & x_1^i = x_2^i \\
\wedge \quad & x_1^{o\prime} = w_{1,1}^o \wedge x_1^{i\,\prime} = w_{1,1}^i \wedge w_{1,0}^{o\,\prime} = w_{1,1}^o \wedge w_{1,1}^{o\,\prime} = w_{1,1}^{i\,\prime} \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_7 \quad : \quad & \{i_1, i_2\} = w_N - 1 > v_1^i - 2 \\
\wedge \quad & x_1^i = x_2^i \\
\wedge \quad & x_1^{o\prime} = w_{1,0}^o \wedge x_1^{i\,\prime} = w_{1,1}^i \wedge w_{1,0}^{o\,\prime} = w_{1,1}^o \wedge w_{1,1}^{o\,\prime} = w_{1,1}^{i\,\prime} \wedge tick(i_1, i_2) \wedge y_1 = x_1^{o\prime} \wedge x_2^i = y_2 \\
\wedge \quad & const(v_1^i, v_1, w_N, v_2^i, v_2)
\end{aligned}
$$

$$\begin{array}{lll} \varphi_8 & : & \{i_1, i_2\} = v^i_1 - 1 < w_N - 1 \\ & \wedge & x^i_1 = x^i_2 \\ & \wedge & x^{o\prime}_1 = w^o_{1,0} \wedge {x^i_1}' = w^i_{1,1} \wedge {w^o_{1,0}}' = w^o_{1,1} \wedge {w^o_{1,1}}' = {w^i_{1,1}}' \wedge tick(i_1, i_2) \wedge y_1 = x^{o\prime}_1 \wedge x^i_2 = y_2 \\ & \wedge & const(v^i_1, v_1, w_N, v^i_2, v_2) \end{array}$$

$$\begin{array}{lll} \varphi_9 & : & \{i_1, i_2\} < w_N - 1 \\ & \wedge & x^{o\prime}_1 = w^o_{1,0} \wedge {x^i_1}' = w^i_{1,1} \wedge {w^o_{1,0}}' = w^o_{1,1} \wedge {w^o_{1,1}}' = {w^i_{1,1}}' \wedge tick(i_1, i_2) \wedge y_1 = x^{o\prime}_1 \wedge x^i_2 = y_2 \\ & \wedge & const(v^i_1, v_1, w_N, v^i_2, v_2) \end{array}$$

$$\begin{array}{lll} \varphi_{10} & : & \{i_1, i_2\} = w_N - 1 \\ & \wedge & x^{o\prime}_1 = w^o_{1,0} \wedge {x^i_1}' = w^i_{1,1} \wedge {w^o_{1,0}}' = w^o_{1,1} \wedge {w^o_{1,1}}' = {w^i_{1,1}}' \wedge tick(i_1, i_2) \wedge y_1 = x^{o\prime}_1 \wedge x^i_2 = y_2 \\ & \wedge & const(v^i_1, v_1, w_N, v^i_2, v_2) \end{array}$$

C.3.1 Postcondition computation

We unfold the φ_5 loop once. Then, we translate the ticks before the unfolded loop:

$$\begin{array}{rcl} \varphi_3 \wedge \varphi_5^{-1}(\top) & : & \varphi_3 \wedge w^o_{1,1} = w^i_{1,1} \wedge {x^i_1}' = {x^i_2}' \wedge {x^i_2}' = {y_2}' \\ \mathcal{T}_0(\varphi_3 \wedge \varphi_5^{-1}(\top)) & : & a_2[0] = b_2 \wedge a_1[0] = a_2[1] \\ \varphi_5 \wedge \varphi_5^{-1}(\top) & : & \varphi_5 \wedge {x^i_1}' = {x^i_2}' \wedge {x^i_2}' = {y_2}' \\ \mathcal{T}_1(\varphi_5 \wedge \varphi_5^{-1}(\top)) & : & a_1[1] = a_2[2] \end{array}$$

Now we translate the loop:

$$\begin{array}{rcl} \varphi^s_5 = \varphi_5(\top) \wedge \varphi_5 & : & \varphi_5 \wedge w^o_{1,1} = w^i_{1,1} \\ \mathcal{T}_i(\varphi^s_5) & : & a_1[i] = a_2[i+1] \\ \mathcal{T}_0({\varphi^s_5}^n) & : & n + 2 = b_2 - 1 \end{array}$$

Here we used the initial value of i in the unfolded loop and the exit condition $i' = b_2 - 1$.

Putting it all together we obtain:

$$\exists n \,.\, a_1[0] = a_2[1] \wedge a_1[1] = a_2[2] \wedge (\forall i \,.\, 2 \leq i < n + 2 \rightarrow a_1[i] = a_2[i+1]) \wedge n + 2 = b_2 - 1$$

By straightforward simplifications:

$$\forall i \,.\, 0 \leq i < b_2 - 1 \rightarrow a_1[i] = a_2[i+1]$$

C.4 Zero Array

Input: An array a, parameter b_1 denoting the number of used cells in a.
Output: All used cells in a are equal to 0.

/* $\psi_{pre}: \ b_1 \geq 0$ */

while $_{a:i=0}(i \leq b_1 - 1)$ **do**
    $a[i] := 0;$
    $i++;$
end

/* $\psi_{post}: \ \forall i.(0 \leq i \leq b_1 - 1) \Rightarrow (a[i] = 0)$ */

Algorithm 4: Zero array

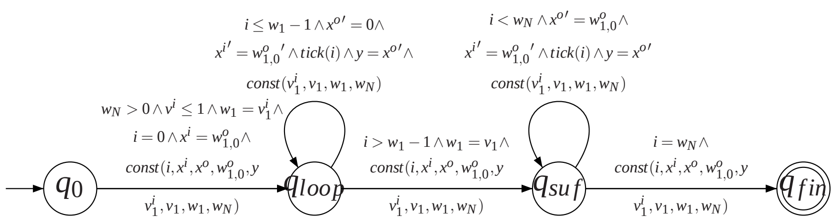


Figure 20: Automata representation of the post-image of the zero array program's loop