Verifying Parameterized Protocols by Transforming Stratified Logic Programs

(Preliminary Version)

Alberto Pettorossi[1], Maurizio Proietti[2], Valerio Senni[1]

(1) DISP, University of Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy
pettorossi@info.uniroma2.it, senni@disp.uniroma2.it
(2) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
proietti@iasi.rm.cnr.it

Abstract. We propose a method for the specification and the automated verification of temporal properties of parameterized protocols. Our method is based on logic programming and program transformation. We specify the properties of parameterized protocols by using an extension of stratified logic programs. This extension allows premises of clauses to contain first order formulas over arrays of parameterized length. A property of a given protocol is proved by applying suitable unfold/fold transformations to the specification of that protocol. We demonstrate our method by proving that the parameterized Peterson's protocol among N processes, for any $N \geq 2$, ensures the mutual exclusion property.

1 Introduction

Protocols are rules that govern the interactions among concurrent processes. In order to guarantee that these interactions enjoy some desirable properties, many sophisticated protocols have been designed and proposed in the literature. These protocols are, in general, difficult to verify because of their complexity and ingenuity. This difficulty has motivated the development of methods for the formal specification and the automated verification of properties of protocols. One of the most successful methods is *model checking* [4]. It can be applied to every protocol that can be formalized as a *finite state system*, that is, a set of transitions over a finite set of states.

Usually, the number of the interacting concurrent processes is not known in advance. Thus, people have designed protocols which can work properly for any number of interacting processes. These protocols are said to be *parameterized* w.r.t. the number of processes.

In this paper we will address the problem of proving properties of parameterized protocols by using the program transformation methodology. The formulas which we manipulate in our transformations, describe the parameterized protocols and the processes themselves. These formulas include the so called *array formulas* which will be presented below.

We will demonstrate our proof method by considering the parameterized Peterson's protocol. It is used for ensuring mutually exclusive access to a given resource which is shared among N processes. Each of these N processes wants to access and possibly modify the shared resource. The number N is the parameter of the parameterized protocol.

We assume that for any i, with $1 \leq i \leq N$, the i-th process consists of an infinite loop whose body is made out of two portions of code: (i) a portion called *critical section*, denoted *cs*, in which the process accesses and modifies the resource, and (ii) a portion called *non-critical section*, denoted *ncs*, in which the process does not access the resource. We also assume that every process is initially in its non-critical section.

We want the following property of the computation of the given system of N processes to hold.

Mutual Exclusion: the statements of the critical section are executed by any one of the N processes while no other process is executing a statement of the critical section.

The parameterized Peterson's protocol consists in adding two portions of code to every process: a first portion to be executed before entering into the critical section, and a second portion to be executed after exiting from the critical section.

We have two arrays $Q[1, \ldots, N]$ and $S[1, \ldots, N-1]$ of integers which are shared among the N processes. The N elements of the array Q are initially set to 0 and may get values from 0 to $N-1$. The $N-1$ elements of the array S are initially set to 1 and may get values from 1 to N.

We also have the array $J[1, \ldots, N]$ whose elements are initially set to 1 and may get values from 1 to N. The array J is *not* shared, because for $i = 1, \ldots, N$, every process i reads and writes $J[i]$ only, but it does nothing on the other elements of the array J.

Process i is of the form given below (see also Figure 1 where it is represented in the form of a finite state diagram).

---

while *true* **do** // Process i
  non-critical section of process i;
  for $J[i] = 1, \ldots, N-1$ **do**
    begin $Q[i] := J[i]$; $S[J[i]] := i$;
      λ: **if** $(\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$ **then** *skip* **else** *goto* λ;
    end;
  critical section of process i;
  $Q[i] := 0$;
    od

The N processes execute their portions of code in a concurrent way. We assume that some operations are atomic and, in particular:
- the tests '$J[i] < N$' and '$\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$' are atomic, and

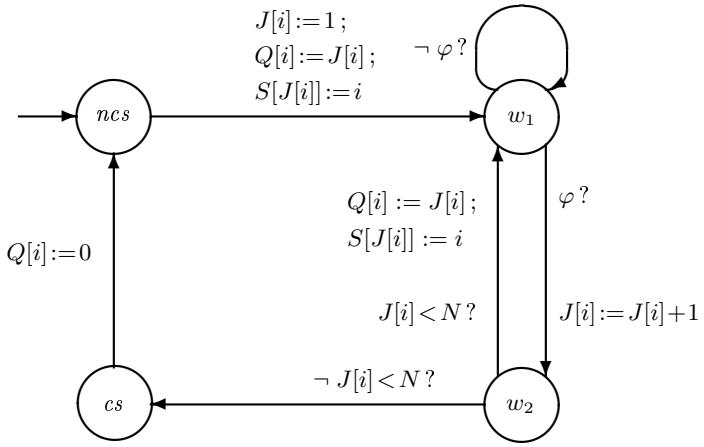

Fig. 1. Finite state diagram corresponding to process i of a system of N processes using Peterson's protocol. The formula φ is $\forall k(k \neq i \rightarrow Q[k] < J[i]) \vee S[J[i]] \neq i$.

- the assignments '$Q[i] := 0;$' and '$J[i] := J[i] + 1;$', and the two sequences of assignments '$J[i] := 1;\ Q[i] := J[i];\ S[J[i]] := i;$' and '$Q[i] := J[i];\ S[J[i]] := i;$' are all atomic.

Actually, less stringent atomicity conditions are sufficient for ensuring that Peterson's protocol guarantees mutual exclusion.

We assume that the value of N does *not* change over time, in the sense that while the computation progresses, neither a new process is constructed nor an existing process is destroyed.

In the original paper [15] G. L. Peterson does *not* prove the mutual exclusion property of its parameterized protocol in a formal way. He leaves that proof to the reader by saying that the correctness of the protocol in the case of N processes, for $N \geq 2$, can be derived from the informal proof provided for the case of two processes because, for each value of $J[i] = 1, \ldots, N-1$, at least one process is discarded from the set of those which may enter into the critical section. In other words, when going from 'level' $J[i]$ to 'level' $J[i]+1$, at least one process is eliminated from the set of those competing for entering into the critical section.

In Peterson's protocol, the value of the variable $J[i]$ of process i indicates the level that process i has reached since it first requested to enter into its critical section. When process i has completed its non-critical section and requests to enter into the critical section, it goes to the state w_1, while its level $J[i]$ has value 1. When process i goes from state w_1 back to state w_1 through state w_2, it increases its level from $J[i]$ to $J[i]+1$.

For each level $J[i] = 1, \ldots, N-1$, process i tests whether or not the following property φ holds, where:

$$\varphi \equiv \quad \forall k(k \neq i \rightarrow Q[k] < J[i]) \vee S[J[i]] \neq i$$

If φ holds then process i enters the next level $J[i]+1$. When process i has successfully tested the property φ at the final level $N-1$, it can enter into its critical section.

In order to formally show that Peterson's protocol ensures mutual exclusion we cannot use directly the model checking technique. Indeed, since the parameter N is unbounded, the parameterized Peterson's protocol can be viewed as a system with an infinite number of states. Now, in order to reduce a system with an infinite number of states to a system with a finite number of states, and thus, be able to apply model checking, one needs to apply an abstraction which, however, is not easily mechanizable.

In this paper we propose an alternative method for the specification and the automated verification of properties of parameterized protocols which does *not* need an abstraction. Our method is based on logic programming and program transformation.

We consider properties of parameterized protocols that can be expressed by using the CTL branching time temporal logic [4]. We formally specify these temporal properties by using an extension of stratified logic programs where premises of clauses may contain first order formulas over arrays of parameterized length. Our specification method is demonstrated in Section 2 by considering the mutual exclusion property of the parameterized Peterson's protocol. Then, in Section 3, we show that this mutual exclusion property can be proved by transforming the given specification using the unfold/fold transformation rules. Finally, in Section 4 we briefly illustrate the related work in the area of the verification of parameterized protocols.

2 Specifying Parameterized Protocols

In this section we present our method for the specification of temporal properties of parameterized protocols. Our method is an extension of other methods based on logic programming and constraint logic programming [6,7,10,12,19]. The main new feature of our method is that in the specifications of protocols we use a first order theory of arrays.

Similarly to the model checking approach, we represent a protocol as a set of transitions between states which are assumed to belong to a possibly infinite set. The transition relation is specified as a binary predicate t, defined by a set of statements of the form:

$$t(a, a') \leftarrow \tau$$

where a and a' are terms representing states and τ is a first order formula. We assume that the transition $t(a, a') \leftarrow \tau$ is not recursive, that is, t does not occur in τ.

For the representation of states and transitions, it is often useful to consider arrays of length N, where N is the number of processes that participate in the protocol. Thus, in order to specify the transition relation, we assume that the

formula τ is an *array formula*, that is, a formula of the first order theory of arrays.

Array formulas are arbitrarily quantified formulas constructed as usual in first order logic starting from the following predicates and function symbols: (i) the equality $=$ between constants (such as the labels ncs, w_1, w_2, and cs of the states of Figure 1), between natural numbers, and between arrays, (ii) the inequalities $<$ and $\leq$, and the disequality $\neq$ between natural numbers, (iii) the constant 0, (iv) the successor $+1$ and predecessor -1, (v) the unary function *length* denoting the length of an array, and (vi) the binary function $-[-]$ such that $A[i]$ denotes the i-th element of the array A.

Statements with array formulas in their premises extend the usual syntax of clauses in logic programs. However, we can translate a non-recursive statement of the form $H \leftarrow \tau$ into a *stratified* set of clauses. Indeed, the predicates used in array formulas can be easily defined by logic programs. For example, the predicate $A[i] = e$ can be defined by a ternary predicate *member* as follows:

$member([E|A], 1, E) \leftarrow$
$member([E_1|A], I, E) \leftarrow I = J+1 \wedge member(A, J, E)$

Moreover, any non-recursive statement of the form $H \leftarrow \tau$ can be translated into a stratified set of clauses by applying the Lloyd-Topor transformation [13]. For instance, given the statement:

$p \leftarrow \forall n\, \exists A\, \forall i\, (i \geq 1 \ \wedge \ n \geq i \ \rightarrow \ \exists e\, A[i] = e)$

by applying the Lloyd-Topor transformation we get:

$p \leftarrow \neg newp1$
$newp1 \leftarrow \neg newp2(N)$
$newp2(N) \leftarrow \neg newp3(N, A)$
$newp3(N, A) \leftarrow I \geq 1 \wedge N \geq I \wedge \neg newp4(A, I)$
$newp4(A, I) \leftarrow member(A, I, E)$

When specifying protocols we will use statements with array formulas, instead of the corresponding clausal translations, because statements are more concise and intuitive. By abuse of language, we will use the term 'clause' also to indicate any statement in which array formulas may occur.

Let us now show how the parameterized Peterson's protocol is specified by using clauses with array formulas.

A *state* is represented by a term of the form $s(P, J, Q, S)$, where:

1. P is the array $P[1, \ldots, N]$ such that, for $i = 1, \ldots, N$, $P[i]$ belongs to the set $\{cs, ncs, w_1, w_2\}$ and represents the state of process i. The constants cs and ncs denote the critical and the non-critical section, respectively, while the constants w_1 and w_2 denote two distinct waiting states.
2. Q and S are shared arrays such that, for $i = 1, \ldots, N$, $Q[i]$ belongs to $\{0, \ldots, N-1\}$ and, for $i = 1, \ldots, N-1$, $S[i]$ belongs to $\{1, \ldots, N\}$. These two arrays can be read and modified by the individual processes.
3. $J[1, \ldots, N]$ is an array such that, for $i = 1, \ldots, N$, $J[i]$ belongs to $\{1, \ldots, N\}$ and represents a local variable that can be read and modified by process i only.

The transition relation is defined by six clauses $T_1, \ldots, T_6$, where for $r = 1, \ldots, 6$, T_r is of the form:

$T_r: \quad t(s(P,J,Q,S), s(P',J',Q',S')) \leftarrow \tau_r(s(P,J,Q,S), s(P',J',Q',S'))$

and $\tau_r(s(P,J,Q,S), s(P',J',Q',S'))$ is an array formula defined as follows:

transition $ncs \rightarrow w_1$:

$$\begin{array}{l}\tau_1(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\ \quad \exists i(P[i]{=}ncs \wedge \\ \quad\quad P'[i]{=}w_1 \ \wedge \ J'[i]{=}1 \ \wedge \ Q'[i]{=}J'[i] \ \wedge \ S'[J'[i]]{=}i \wedge \\ \quad\quad \forall k(k{\neq}i \ \rightarrow \ (P'[k]{=}P[k] \ \wedge \ Q'[k]{=}Q[k] \ \wedge \ J'[k]{=}J[k])) \wedge \\ \quad\quad \forall k(k{\neq}J[i] \ \rightarrow \ S'[k]{=}S[k]))\end{array}$$

transition $w_1 \rightarrow w_1$:

$$\begin{array}{l}\tau_2(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\ \quad \exists i(P[i]{=}w_1 \ \wedge \ \exists k(k{\neq}i \ \wedge \ Q[k]{\geq}J[i]) \ \wedge \ S[J[i]]{=}i) \wedge \\ \quad P'{=}P \ \wedge \ J'{=}J \ \wedge \ Q'{=}Q \ \wedge \ S'{=}S\end{array}$$

transition $w_1 \rightarrow w_2$:

$$\begin{array}{l}\tau_3(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\ \quad \exists i(P[i]{=}w_1 \ \wedge \ (\forall k(k{\neq}i \ \rightarrow \ Q[k]{<}J[i]) \ \vee \ S[J[i]]{\neq}i) \wedge \\ \quad\quad P'[i]{=}w_2 \ \wedge \ J'[i]{=}J[i]{+}1 \wedge \\ \quad\quad \forall k(k{\neq}i \ \rightarrow \ P'[k]{=}P[k] \ \wedge \ J'[k]{=}J[k])) \wedge \\ \quad Q'{=}Q \ \wedge \ S'{=}S\end{array}$$

transition $w_2 \rightarrow w_1$:

$$\begin{array}{l}\tau_4(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\ \quad \exists i(P[i]{=}w_2 \ \wedge \ J[i] < \mathit{length}(P) \wedge \\ \quad\quad P'[i]{=}w_1 \ \wedge \ Q'[i]{=}J[i] \ \wedge \ S'[J[i]]{=}i \wedge \\ \quad\quad \forall k(k{\neq}i \ \rightarrow \ (P'[k]{=}P[k] \ \wedge \ Q'[k]{=}Q[k])) \wedge \\ \quad\quad \forall k(k{\neq}J'[i] \ \rightarrow \ S'[k]{=}S[k])) \wedge \\ \quad J'{=}J\end{array}$$

transition $w_2 \rightarrow cs$:

$$\begin{array}{l}\tau_5(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\ \quad \exists i(P[i]{=}w_2 \ \wedge \ J[i]{=}\mathit{length}(P) \ \wedge \ P'[i]{=}cs \wedge \\ \quad\quad \forall k(k{\neq}i \ \rightarrow \ P'[k]{=}P[k])) \wedge \\ \quad J'{=}J \ \wedge \ Q'{=}Q \ \wedge \ S'{=}S\end{array}$$

transition $cs \rightarrow ncs$:

$$\begin{array}{l}\tau_6(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\ \quad \exists i(P[i]{=}cs \ \wedge \ P'[i]{=}ncs \ \wedge \ Q'[i]{=}0 \wedge \\ \quad\quad \forall k(k{\neq}i \ \rightarrow \ (P'[k]{=}P[k] \ \wedge \ Q'[k]{=}Q[k]))) \wedge \\ \quad S'{=}S \ \wedge \ J'{=}J\end{array}$$

We express the properties of interest of the parameterized protocols by using the CTL branching time temporal logic [4]. In particular, the mutual exclusion property of Peterson's protocol is expressed by the temporal formula:

$$\mathit{initial} \rightarrow \neg EF \ \mathit{unsafe}$$

where *initial* and *unsafe* are atomic properties of states which we will specify below. This temporal formula holds at a state a iff if a is an *initial* state then there exists no *unsafe* state in the future of a.

The truth of a CTL temporal formula is defined by the following locally stratified logic program, where the predicate $holds(a, f)$ means that the temporal formula f holds at state a:

$holds(X, F) \leftarrow atomic(X, F)$
$holds(X, \neg F) \leftarrow \neg holds(X, F)$
$holds(X, F \rightarrow G) \leftarrow \neg holds(X, F)$
$holds(X, F \rightarrow G) \leftarrow holds(X, F) \wedge holds(X, G)$
$holds(X, ef(F)) \leftarrow holds(X, F)$
$holds(X, ef(F)) \leftarrow t(X, X') \wedge holds(X', ef(F))$

The unary constructor ef encodes the temporal logic operator EF. Other temporal operators can be defined by using locally stratified logic programs [7,12]. For reasons of simplicity, here we have restricted ourselves to the operator EF which is the only operator needed for specifying the mutual exclusion property of Peterson's protocol.

The atomic properties of the states are specified by a set of non-recursive clauses of the form:

$atomic(a, p) \leftarrow \psi$

where ψ is an array formula stating that the atomic formula p holds at state a. In particular, *initial* and *unsafe* are atomic properties specified as follows.

$atomic(s(P, J, Q, S), initial) \leftarrow \forall i\ (P[i]{=}ncs \wedge J[i]{=}1 \wedge Q[i]{=}0 \wedge S[i]{=}1)$

The premise of the above clause will also be written as $init\_state(s(P, J, Q, S))$, and it means that in an initial state every process is in its non-critical section, J is an array whose elements are all 1's, Q is an array whose elements are all 0's, and S is an array whose elements are all 0's.

$atomic(s(P, J, Q, S), unsafe) \leftarrow \exists i, j\ (P[i] = cs \wedge P[j] = cs \wedge i \neq j)$

The premise of the above clause will also be written as $unsafe\_state(s(P, J, Q, S))$, and it means that in an unsafe state two distinct processes are in critical section.

Let *Peterson* be the program consisting of the clauses that define the binary predicates *holds* and *atomic*, and the binary transition relation t. *Peterson* is a locally stratified program which, therefore, has a perfect model [1], denoted by $M(Peterson)$. We will prove that the temporal formula $initial \rightarrow \neg EF\ unsafe$ holds for every state, by showing that:

$M(Peterson) \models \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$

Notice that X ranges over terms of the form $s(P, J, Q, S)$ where the length of the array P of the states of the processes is the parameter N. Thus, by showing the above formula we show that Peterson's protocol ensures mutual exclusion for any number N of processes.

3 Transformational Verification of the Parameterized Peterson's Protocol

In this section we prove that the mutual exclusion property holds for the parameterized Peterson's protocol by using program transformation. As a first step we introduce the statement:

$mutex \leftarrow \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$

which, by applying the Lloyd-Topor transformation [13], is transformed into the following set of clauses:

1. $mutex \leftarrow \neg new1$
2. $new1 \leftarrow new2(X)$
3. $new2(X) \leftarrow holds(X, initial) \wedge holds(X, ef(unsafe))$

We have that:

$M(Peterson) \models \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$ iff

$M(Peterson \cup \{1, 2, 3\}) \models mutex$

We will show that $M(Peterson \cup \{1, 2, 3\}) \models mutex$ by applying unfold/fold transformation rules that preserve the perfect model [9,20] and deriving from the program $Peterson \cup \{1, 2, 3\}$ a new program T which contains the clause $mutex \leftarrow$.

The unfold/fold transformation rules are guided by a transformation strategy similar to the ones presented in [7,16]. We do not indicate this strategy here and we only show it in action in our verification of Peterson's protocol.

First we transform clause 3 with the objective of deriving the specialized definitions corresponding to the instances of the atom $holds(X, F)$ for various values of the state X and the formula F. Indeed, by doing so we will discover that $new2(X)$, which corresponds to the instance of $holds(X, F)$ where X is an initial state and F is $ef(unsafe)$, is false. Then, by unfolding, we can easily derive the clause $mutex \leftarrow$.

Starting from clause 3, we apply the following transformation steps: (i) we unfold clause 3, thereby deriving a new set, say G, of clauses, (ii) we manipulate the array formulas occurring in the clauses of that set G, by replacing these formulas by equivalent ones, and we remove each clause whose body contains an unsatisfiable formula, (iii) we introduce new predicate definitions and we fold every instance of $holds(X, F)$. Starting from every new predicate definition which has been introduced, we repeat the above transformation steps (i), (ii), and (iii) until we are able to fold every instance of $holds(X, F)$ by using a predicate definition introduced at a previous step.

As already mentioned in [7,16], this unfolding/definition/folding procedure is not ensured to terminate in general, because properties of programs are in general undecidable. However, for many classes of programs and properties, this procedure terminates and, for those classes, it behaves as a decision procedure.

Let us now show some of the transformation steps for verifying that Peterson's protocol enjoys the mutual exclusion property. By unfolding clause 3 we get:

4. $new2(s(P, J, Q, S)) \leftarrow init\_state(s(P, J, Q, S)) \ \wedge \ unsafe\_state(s(P, J, Q, S))$
5. $new2(s(P, J, Q, S)) \leftarrow init\_state(s(P, J, Q, S)) \wedge$
   $t(s(P, J, Q, S), s(P', J', Q', S')) \wedge$
   $holds(s(P', J', Q', S'), ef(unsafe))$

By unfolding clause 5 w.r.t. the atom t we get six new clauses, one for each clause $T_1, \ldots, T_6$ defining the transition relation (see previous Section 2). The clauses

derived from $T_2, \ldots, T_6$ are removed because their bodies contain unsatisfiable array formulas. Also clause 4 is removed because the formula

$$\mathit{init\_state}(s(P,J,Q,S)) \ \wedge \ \mathit{unsafe\_state}(s(P,J,Q,S))$$

occurring in its body is unsatisfiable. Thus, the only clause derived from clause 3 after the unfolding and removal steps is:

$$\begin{array}{lll} 6.\ \mathit{new2}(s(P,J,Q,S)) \leftarrow & \mathit{init\_state}(s(P,J,Q,S)) \wedge \\ & \tau_1(s(P,J,Q,S), s(P',J',Q',S')) \wedge \\ & \mathit{holds}(s(P',J',Q',S'), \mathit{ef}(\mathit{unsafe})) \end{array}$$

where $\tau_1(s(P,J,Q,S), s(P',J',Q',S'))$ is the array formula defined in the previous section. Let us consider the formula $c_1(s(P',J',Q',S'))$ defined as follows:

$$\begin{array}{l} c_1(s(P',J',Q',S')) \equiv \\ \quad \exists P,J,Q,S \ (\mathit{init\_state}(s(P,J,Q,S)) \wedge \tau_1(s(P,J,Q,S), s(P',J',Q',S'))) \end{array}$$

This formula $c_1(s(P',J',Q',S'))$ characterizes the set of successor states of the state $s(P,J,Q,S)$. We have that the following equivalence holds:

$$\begin{array}{lll} c_1(s(P',J',Q',S')) \equiv \exists i(P'[i]{=}w_1 \ \wedge \ Q'[i]{=}J'[i] \ \wedge \ S'[J'[i]]{=}i \ \wedge J'[i]{=}1 \ \wedge \\ \qquad\qquad \forall k(k \neq i \rightarrow (P'[k] = \mathit{ncs} \ \wedge \ Q'[k] = 0))) \end{array}$$

In order to fold the atom $\mathit{holds}(s(P',J,Q',S'), \mathit{ef}(\mathit{unsafe}))$ in the body of clause 6 by applying the folding rule of [9], we need to introduce a new predicate definition of the form:

$$\begin{array}{lll} 7.\ \mathit{new3}(s(P,J,Q,S)) \leftarrow & \mathit{genc}_1(s(P,J,Q,S)) \wedge \\ & \mathit{holds}(s(P,J,Q,S), \mathit{ef}(\mathit{unsafe})) \end{array}$$

where $\mathit{genc}_1(s(P,J,Q,S))$ is a *generalization* of $c_1(s(P,J,Q,S))$, in the sense that $\forall P,J,Q,S \ (c_1(s(P,J,Q,S)) \rightarrow \mathit{genc}_1(s(P,J,Q,S)))$ holds. As usual in the program transformation approach, this generalization step requires ingenuity. Here we will not address the problem of how to mechanize the generalization steps. This crucial issue is left for future research. In our example we continue the derivation by introducing the following array formula $\mathit{genc}_1(s(P,J,Q,S))$ which expresses the fact that, for any set of processes which are in state w_1, there exists a process i which has been the last one to enter w_1 and to set $S[J[i]] = i$:

$$\begin{array}{lll} \mathit{genc}_1(s(P,J,Q,S)) \equiv \exists i(P[i]{=}w_1 \ \wedge \ Q[i]{=}J[i] \ \wedge \ J[i]{=}1 \ \wedge \ S[J[i]]{=}i) \wedge \\ \qquad\qquad \forall k((P[k]{=}\mathit{ncs} \ \wedge \ Q[k]{=}0) \vee \\ \qquad\qquad\quad (P[k]{=}w_1 \ \wedge \ Q[k]{=}J[k] \wedge J[k]{=}1)) \end{array}$$

By folding clause 6 using the newly introduced clause 7 we get:

$$\begin{array}{lll} 6.\mathrm{f}\ \ \mathit{new2}(s(P,Q,S,J)) \leftarrow & \mathit{init\_state}(s(P,J,Q,S)) \wedge \\ & \tau_1(s(P,J,Q,S), s(P',J',Q',S')) \wedge \\ & \mathit{new3}(s(P',J',Q',S')) \end{array}$$

Now, starting from clause 7, we repeat the transformation steps (i), (ii), and (iii) described above, until we are able to fold every instance of $holds(X, F)$ by using a predicate definition introduced at a previous step. By doing so we derive the following program S where:
- $genc_1$ is defined as indicated above,
- $genc_2, \ldots, genc_8$ are defined as indicated in the Appendix,
- $\tau_1, \ldots, \tau_6$ are the array formulas that define the transition relation as indicated in Section 2, and
- the arguments a and a' stand for the states $s(P, J, Q, S)$ and $s(P', J', Q', S')$, respectively.

Program S

1. $mutex \leftarrow \neg new1$
2. $new1 \leftarrow new2(X)$

6.f $new2(a) \leftarrow initial(a) \wedge \tau_1(a, a') \wedge new3(a')$

8. $new3(a) \leftarrow genc_1(a) \wedge \tau_1(a, a') \wedge new3(a')$
9. $new3(a) \leftarrow genc_1(a) \wedge \tau_2(a, a') \wedge new3(a')$
10. $new3(a) \leftarrow genc_1(a) \wedge \tau_3(a, a') \wedge new4(a')$
11. $new3(a) \leftarrow genc_1(a) \wedge \tau_3(a, a') \wedge new5(a')$

12. $new4(a) \leftarrow genc_2(a) \wedge \tau_1(a, a') \wedge new8(a')$
13. $new4(a) \leftarrow genc_2(a) \wedge \tau_2(a, a') \wedge new4(a')$
14. $new4(a) \leftarrow genc_2(a) \wedge \tau_3(a, a') \wedge new4(a')$
15. $new4(a) \leftarrow genc_2(a) \wedge \tau_4(a, a') \wedge new4(a')$
16. $new4(a) \leftarrow genc_2(a) \wedge \tau_5(a, a') \wedge new9(a')$

17. $new5(a) \leftarrow genc_3(a) \wedge \tau_1(a, a') \wedge new5(a')$
18. $new5(a) \leftarrow genc_3(a) \wedge \tau_2(a, a') \wedge new5(a')$
19. $new5(a) \leftarrow genc_3(a) \wedge \tau_3(a, a') \wedge new4(a')$
20. $new5(a) \leftarrow genc_3(a) \wedge \tau_3(a, a') \wedge new5(a')$
21. $new5(a) \leftarrow genc_3(a) \wedge \tau_4(a, a') \wedge new6(a')$
22. $new5(a) \leftarrow genc_3(a) \wedge \tau_5(a, a') \wedge new7(a')$

23. $new6(a) \leftarrow genc_4(a) \wedge \tau_1(a, a') \wedge new6(a')$
24. $new6(a) \leftarrow genc_4(a) \wedge \tau_2(a, a') \wedge new6(a')$
25. $new6(a) \leftarrow genc_4(a) \wedge \tau_3(a, a') \wedge new6(a')$
26. $new6(a) \leftarrow genc_4(a) \wedge \tau_4(a, a') \wedge new6(a')$
27. $new6(a) \leftarrow genc_4(a) \wedge \tau_5(a, a') \wedge new7(a')$

28. $new7(a) \leftarrow genc_5(a) \wedge \tau_2(a, a') \wedge new7(a')$
29. $new7(a) \leftarrow genc_5(a) \wedge \tau_6(a, a') \wedge new6(a')$

30. $new8(a) \leftarrow genc_6(a) \wedge \tau_1(a, a') \wedge new8(a')$
31. $new8(a) \leftarrow genc_6(a) \wedge \tau_2(a, a') \wedge new8(a')$
32. $new8(a) \leftarrow genc_6(a) \wedge \tau_3(a, a') \wedge new8(a')$
33. $new8(a) \leftarrow genc_6(a) \wedge \tau_4(a, a') \wedge new8(a')$
34. $new8(a) \leftarrow genc_6(a) \wedge \tau_5(a, a') \wedge new10(a')$

35. $new9(a) \leftarrow genc_7(a) \wedge \tau_1(a, a') \wedge new10(a')$
36. $new9(a) \leftarrow genc_7(a) \wedge \tau_6(a, a') \wedge new2(a')$

37. $new10(a) \leftarrow genc_8(a) \wedge \tau_1(a, a') \wedge new10(a')$
38. $new10(a) \leftarrow genc_8(a) \wedge \tau_2(a, a') \wedge new10(a')$
39. $new10(a) \leftarrow genc_8(a) \wedge \tau_3(a, a') \wedge new10(a')$
40. $new10(a) \leftarrow genc_8(a) \wedge \tau_4(a, a') \wedge new10(a')$
41. $new10(a) \leftarrow genc_8(a) \wedge \tau_6(a, a') \wedge new8(a')$

An inspection of program S reveals that predicates $new1$ through $new10$ are *useless*, that is, for every predicate p in the set $\{new1, \ldots, new10\}$ and for every clause C that defines p in S, there exists a predicate q in $\{new1, \ldots, new10\}$ which occurs positively in the body of C. The removal of the clauses that define useless predicates preserves the perfect model of the program [9]. Thus, we remove clauses 2, 6.f, and 8 through 41, and we derive a program consisting of clause 1 only. By unfolding clause 1 we get the final program T, which consists of the clause $mutex \leftarrow$ only. Thus, $M(T) \models mutex$ and we have proved that:

$M(Peterson) \models \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$

that is, for any initial state and for any number N of processes, the mutual exclusion property holds for Peterson's protocol for N processes.

4 Related Work and Conclusions

The protocol verification method presented in this paper is based on the program transformation approach proposed in [16] for the verification of properties of locally stratified logic programs. We use locally stratified logic programs which are extended with array formulas and the properties we consider are temporal properties of parameterized systems, that is, systems consisting of an *arbitrary* number of *finite state* processes.

As yet, our method is *not* fully mechanical and human intervention is needed for the following two tasks: (i) the verification of array formulas and (ii) the introduction of new definitions by generalization.

The problem of verifying array formulas has been addressed in several papers (see [21] for a short survey). In general this problem is undecidable. However, some decidable fragments such as the *quantifier-free extensional theory of arrays*, have been identified [21]. Unfortunately, the array formulas considered in this paper cannot be reduced to formulas of the quantifier-free extensional theory of arrays. As an alternative approach we are working on the design of (necessarily incomplete) transformational strategies which would allow us to check validity of most array formulas that occur in practice in the verification of parameterized protocols.

The introduction of suitable new definitions by generalization is a typical issue of the program transformation methodology [3] and it corresponds to the discovery of suitable invariants of the protocol to be verified. There is no universal method for automating these generalization steps. However, we believe that suitable techniques can be devised by focusing on specific classes of protocols.

Other verification methods based on the transformational approach presented in [16] are those described in [7] and [8]. In [7] it is presented a method for verifying CTL properties of systems consisting of a *fixed number* of infinite state processes. The method of [7] makes use of locally stratified constraint logic programs, where the constraints are linear equations and disequations on real numbers. In this paper we have followed a paradigm similar to constraint logic programming where, however, the constraints are array formulas. The method presented here can be easily extended to deal with parameterized infinite state systems by considering, for instance, arrays of infinite state processes.

The paper [7] describes the verification of the mutual exclusion property for the parameterized Bakery protocol [11]. That paper uses locally stratified logic programs extended with formulas of the Weak Monadic Second Order Theory of k-Successors (WSkS), which describes monadic properties of strings. The array formulas considered in this paper are more expressive than WSkS formulas, because array formulas may express polyadic properties. However, as already mentioned, the general theory of array formulas is undecidable, while the theory WSkS is decidable.

Other transformational approaches to the verification of concurrent systems have been proposed in [12,18,19].

The method described in [12] uses partial deduction and abstract interpretation of logic programs for verifying safety properties of infinite state systems. Partial deduction is strictly less powerful than unfold/fold program transformation, which, on the other hand, is more difficult to mechanize when unrestricted transformations are considered. One of the main objectives of our future research is the design of suitably restricted unfold/fold transformations which are powerful enough for verification purposes and yet amenable to mechanization.

The work presented in [18,19] is the most similar to ours. Indeed, the authors of [18,19] use unfold/fold rules for transforming programs and proving properties of parameterized concurrent systems. Our paper differs from [18,19] in that, instead of using definite logic programs, we use logic programs with locally stratified negation and array formulas for the specification of concurrent systems and their properties. As a consequence, also the transformation rules we consider are different and more general than those used in [18,19].

Besides the above mentioned transformational methods, some more verification methods based on (constraint) logic programming have been proposed in the literature [6,10,14,17].

The methods proposed in [14,17] deal with finite state systems only. In particular, the method presented in [14] uses CLP with finite domains, extended with constructive negation and tabled resolution, for finite state local model checking, and the method described in [17] uses tabled logic programming to efficiently verify μ-calculus properties of finite state systems expressed in the CCS calculus.

The methods presented in [6,10] deal with infinite state systems. In particular, the method presented in [6] uses constraint logic programs to represent infinite state systems. This method can be applied to verify CTL properties of

infinite state systems by computing approximations of least and greatest fixed points via abstract interpretation. An extension of this method has also been used for the verification of parameterized cache coherence protocols [5]. The method described in [10] uses logic programs with linear arithmetic constraints and Presburger arithmetic to verify safety properties of Petri nets. However, parameterized systems that use arrays, like Peterson's protocol, cannot be directly specified and verified using the methods in [6,10], because in general array formulas cannot be encoded as constraints over the real numbers or Presburger formulas.

Several verification techniques for parameterized systems have been presented also outside the area of logic programming (see [22] for a survey of some of these techniques). These techniques extend finite state model checking with various forms of *induction* (for proving properties for every value of the parameter) or *abstraction* (for reducing the verification of a parameterized system to the verification of a finite state system).

We do not have the space here for discussing the relationships of our work with all these techniques. We only want to mention the technique presented in [2], which is also applied for the verification of the parameterized Peterson's protocol. The technique proposed in [2] can be applied for verifying in an automatic way safety properties of all systems that satisfy a so-called *stratification* condition. Indeed, when this restriction holds for a given parameterized system, then the verification task can be reduced to the verification of a number of finite-state systems that are instances of the given parameterized system for suitable values of the parameter. However, Peterson's protocol does *not* satisfy the stratification condition and its treatment with the technique proposed in [2] requires a substantial amount of ingenuity.

Finally, we want to notice that techniques based on deduction and transformation which have been developed in the area of logic programming, seem particularly promising when moving from the problem of verifying finite state systems to the problem of verifying infinite state systems and parameterized systems, because in the latter verification logical reasoning plays a crucial role.

Appendix

Below we give the definitions of the array formulas $genc_2$ through $genc_8$ occurring in the program S.

$$
\begin{array}{l}
genc_2(s(P,J,Q,S)) \equiv \\
\quad \exists i,k(1 \leq k \leq length(P) \wedge \\
\qquad ((P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k-1 \ \wedge \ S[J[i]]=i) \vee \\
\qquad \ (P[i]=w_2 \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=k \ \wedge \ S[J[i]-1]=i)) \wedge \\
\qquad \forall j(j \neq i \ \rightarrow \ P[j]=ncs \ \wedge \ Q[j]=0))
\end{array}
$$

$$
\begin{array}{l}
genc_3(s(P,J,Q,S)) \equiv \\
\quad \exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=1 \ \wedge \ S[J[i]]=i) \wedge
\end{array}
$$

$$\forall k((P[k]=ncs \ \wedge \ Q[k]=0) \vee$$
$$(P[k]=w_1 \ \wedge \ Q[k]=J[k] \ \wedge \ J[k]=1) \wedge$$
$$(P[k]=w_2 \ \wedge \ Q[k]=J[k]-1 \ \wedge \ J[k]=2))$$

$$genc_4(s(P,J,Q,S)) \equiv$$
$$\exists m(m \leq length(P) \wedge$$
$$\forall k(1 \leq k \leq m \ \rightarrow \ \exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k-1 \ \wedge \ S[J[i]]=i)) \wedge$$
$$\forall j((P[j]=ncs \ \wedge \ Q[j]=0) \vee$$
$$\exists k(k \leq m \ \wedge \ ((P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=k-1) \vee$$
$$(P[j]=w_2 \ \wedge \ Q[j]=J[j]-1 \ \wedge \ J[j]=k)))))$$

$$genc_5(s(P,J,Q,S)) \equiv$$
$$\exists i(P[i]=cs \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=length(P) \ \wedge \ S[J[i]-1]=i) \wedge$$
$$\forall k(1 \leq k \leq length(P)-1 \ \rightarrow$$
$$\exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k \ \wedge \ S[J[i]]=i))$$

$$genc_6(s(P,J,Q,S)) \equiv$$
$$\exists l,n(n \leq length(P) \wedge$$
$$((P[l]=w_1 \ \wedge \ Q[l]=J[l] \ \wedge \ J[l]=n-1 \ \wedge \ S[J[l]]=l) \vee$$
$$(P[l]=w_2 \ \wedge \ Q[l]=J[l]-1 \ \wedge \ J[l]=n \ \wedge \ S[J[l]-1]=l)) \wedge$$
$$\exists m(m \leq n \wedge$$
$$\forall k(1 \leq k \leq m \ \rightarrow \ \exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k-1 \ \wedge \ S[J[i]]=i)) \wedge$$
$$\forall j((P[j]=ncs \ \wedge \ Q[j]=0) \vee$$
$$\exists k(k \leq m \ \wedge \ ((P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=k) \vee$$
$$(P[j]=w_2 \ \wedge \ Q[j]=J[j]-1 \ \wedge \ J[j]=k))))))$$

$$genc_7(s(P,J,Q,S)) \equiv$$
$$\exists i(P[i]=cs \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=length(P) \ \wedge \ S[J[i]-1]=i \wedge$$
$$\forall k(k \neq i \ \rightarrow \ (P[k]=ncs \ \wedge \ Q[k]=0)))$$

$$genc_8(s(P,J,Q,S)) \equiv$$
$$\exists i(P[i]=cs \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=length(P) \ \wedge \ S[J[i]-1]=i \wedge$$
$$\exists m(m \leq length(P) \wedge$$
$$\forall l(1 \leq l \leq m \ \rightarrow \ \exists j(P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=l-1 \ \wedge \ S[J[j]]=j)) \wedge$$
$$\forall j(j \neq i \ \rightarrow \ ((P[j]=ncs \ \wedge \ Q[j]=0) \vee$$
$$\exists k(k \leq m \wedge$$
$$((P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=k-1) \vee$$
$$(P[j]=w_2 \ \wedge \ Q[j]=J[j]-1 \ \wedge \ J[j]=k)))))))$$

References

1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
2. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 221–234. ACM, July 2001.

3. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
4. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
5. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state systems using program transformation. In A. Pettorossi, editor, *Proceedings of LOPSTR 2001, Eleventh International Workshop on Logic-based Program Synthesis and Transformation*, Lecture Notes in Computer Science 2372, pages 111–128. Springer-Verlag, 2002.
9. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer, 2004.
10. L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with z-counters. *Constraints*, 2(3/4):305–335, 1997.
11. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
12. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of LOPSTR '99, Venice, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 1999.
13. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
14. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.
15. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
16. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer, 2000.
17. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV '97*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 1997.
18. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *CAV 2001*, pages 25–37, 2001.
19. A. Roychoudhury and C. R. Ramakrishnan. Unfold/fold transformations for automated verification. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 261–290. Springer, 2004.

20. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
21. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Press, 2001.
22. L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.