Transformational Verification of Parameterized Protocols Using Array Formulas

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* [5]. It can be applied to any protocol that can be formalized as a *finite state system*, that is, a finite set of transitions over a finite set of states.

Usually, the number of interacting concurrent processes is not known in advance. Thus, people have designed protocols that can work properly for any number of interacting processes. These protocols are said to be *parameterized* with respect to the number of processes. Several extensions of the model checking technique based upon *abstraction* and *induction* have been proposed in the literature for the verification of parameterized protocols (see, for instance, [3,18,27,29]). However, since the general problem of verifying temporal properties of parameterized protocols is undecidable [2], these extensions cannot be fully mechanical.

In this paper we propose an alternative verification method based on *program transformation* [4]. Our main objective is to establish a correspondence between protocol verification and program transformation, so that the large number of semi-automatic techniques developed in the field of program transformation can be applied to the verification of properties of parameterized protocols.

Since arrays are often used in the design of parameterized protocols, we will consider a specification language that allows us to write *array formulas*, that is, first order formulas over arrays. We will specify a parameterized protocol and a property of interest by means of a logic program whose clause bodies may contain array formulas. Our verification method works by transforming this logic program, in which we assume that the head of the clause specifying the property has predicate *prop*, into a new logic program where the clause $prop \leftarrow$ occurs. Our verification method is an extension of many other techniques based on logic programming which have been proposed in the literature [7,9,11,15,19,22,23].

We will demonstrate our method by considering the parameterized Peterson's protocol [20]. This protocol ensures mutually exclusive use of a given resource which is shared among N processes. The number N is the parameter of the parameterized protocol. In order to formally show that Peterson's protocol ensures mutual exclusion, we cannot use the model checking technique directly. Indeed, since the parameter N is unbounded, the parameterized Peterson's protocol, as it stands, cannot be viewed as a finite state system. Now, one can reduce it to a finite state system, thereby enabling the application of model checking, by using the above mentioned techniques based on abstraction [3]. However, it is not easy to find a powerful abstraction function which works for the many protocols and concurrent systems one encounters in practice.

In contrast, our verification method based on program transformation does not rely on an abstraction function which is applied once at the beginning of the verification process, but it relies, instead, on a *generalization strategy* which is applied *on demand* during the construction of the proof, possibly many times, depending on the structure of the portion of proof constructed so far. This technique provides a more flexible approach to the problem of proving properties of protocols with an infinite state space.

The paper is structured as follows. In Section 2 we recall the parameterized Peterson's protocol for mutual exclusion which will be used throughout the paper as a working example. In Section 3 we present our specification method which makes use of an extension of stratified logic programs where bodies of clauses may contain first order formulas over arrays of parameterized length. We consider properties of parameterized protocols that can be expressed by using formulas of the branching time temporal logic CTL [5] and we show how these properties can be encoded by stratified logic programs with array formulas. Then, in Section 4, we show how CTL properties can be proved by applying unfold/fold transformation rules to a given specification. In Section 5 we discuss some issues regarding the automation of our transformation method. Finally, in Section 6 we briefly discuss the related work in the area of the verification of parameterized protocols.

2 Peterson's mutual exclusion protocol

In this section we provide a detailed description of the parameterized Peterson's protocol [20]. The goal of this protocol is to ensure the mutually exclusive access to a resource that is shared among N (≥ 2) processes. Let assume that for any

i, with $1 \leq i \leq N$, process i 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 uses the resource, and (ii) a portion called *non-critical section*, denoted *ncs*, in which the process does not use the resource. We also assume that every process is initially in its non-critical section.

We want to establish the following *Mutual Exclusion* property of the computation of the given system of N processes: *for all i and j in $\{1, \ldots, N\}$, while process i executes a statement of its critical section, process j, with $j \neq i$, does not execute any statement of its critical section.*

The parameterized Peterson's protocol consists in adding two portions of code to every process: (i) a first portion to be executed before entering the critical section, and (ii) a second portion to be executed after exiting the critical section (see in Figure 1 the code relative to process i).

Peterson's protocol makes use of two arrays $Q[1, \ldots, N]$ and $S[1, \ldots, N]$ of natural numbers, which are shared among the N processes. The N elements of the array Q may get values from 0 to $N-1$ and are initially set to 0. The N elements of the array S may get values from 1 to N and their initial values are not significant (in [20] it is assumed that they are all 1's). Notice that in [20] the array S is assumed to have $N-1$ elements, not N as we do. Indeed, the last element $S[N]$ is never used by Peterson's protocol. Its introduction, however, allows us to write formulas which are much simpler.

In Peterson's protocol we also have the array $J[1, \ldots, N]$ whose i-th element, for $i = 1, \ldots, N$, is a local variable of process i and may get values from 1 to N. Notice that the array J is *not* shared and indeed, for $i = 1, \ldots, N$, process i reads and/or writes $J[i]$ only.

In Figure 2 process i is represented by a finite state diagram. In that diagram a transition from state a to state b is denoted by an arrow from a to b labelled by a test t and a statement s. We have omitted from the label of a transition the test t when it is *true*. Likewise, we have omitted the statement s when it is *skip*. A transition is said to be *enabled* iff its test t evaluates to *true*. An enabled transition takes place by executing its statement s.

For $i = 1, \ldots, N$, process i is deterministic in the sense that in any of its states at most one transition is enabled. However, in the given system of N processes, it may be the case that more than one transition is enabled (obviously, no two enabled transitions belong to the same process). In that case we assume that exactly one of the enabled transitions takes place. Note that we do not make any fairness assumption so that, for instance, if the same configuration of enabled transitions occurs again in the future, nothing can be said about the transition which will actually take place in that repeated configuration.

The N processes execute their code in a concurrent way according to the following four atomicity assumptions. Here and in what follows, we denote by φ the formula $\forall k\, (k \neq i \;\rightarrow\; Q[k] < J[i]) \vee (S[J[i]] \neq i)$.

(1) The assignments '$Q[i] := 0$' and '$J[i] := 1$' are atomic,
(2) the tests '$\neg J[i] < N$' and '$\neg\varphi$' are atomic,
(3) the sequence of the test '$J[i] < N$' followed by the two assignments '$Q[i] := J[i];\ S[J[i]] := i$' is atomic, and

```
while $true$ do
  $ncs$ : non-critical section of process $i$;
          $J[i] := 1$;
  $w$:    while $J[i] < N$ do
              $Q[i] := J[i]$;  $S[J[i]] := i$;
  $\lambda$:        if $\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$ then $J[i] := J[i]+1$ else goto $\lambda$
          od;
  $cs$ :  critical section of process $i$;
          $Q[i] := 0$
od
```

Fig. 1. Process i of a system of N processes using Peterson's protocol.

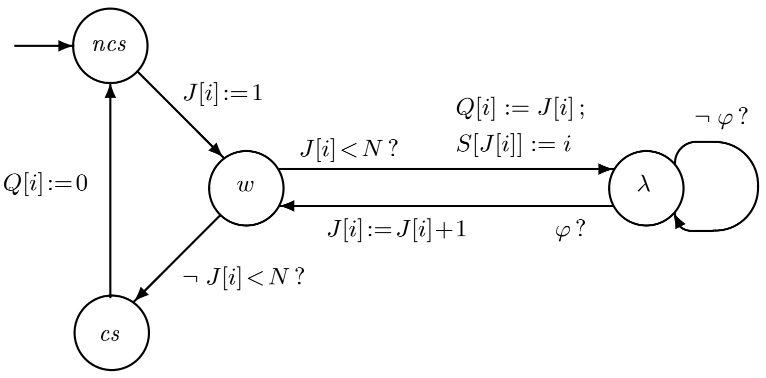

Fig. 2. Finite state diagram corresponding to process i of a system of N processes using Peterson's protocol. ncs is the initial state. The formula φ stands for $\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$.

(4) the sequence of the test 'φ' followed by the assignment '$J[i] := J[i]+1$' is atomic.

We have made these atomicity assumptions (which correspond to the labels of the transitions of the diagram of Figure 2) for keeping the presentation of our proof of the mutual exclusion property as simple as possible. However, this property has also been proved by using our method which we will present in Section 4, under weaker assumptions, in which one only assumes that every single assignment and test is atomic [26]. (In particular, in [26] it is assumed that each test $k \neq i$ and '$Q[k] < J[i]$' in the formula φ, and *not* the entire formula φ, is atomic. Likewise, it is assumed that in the transition from state w to state λ, each assignment '$Q[i] := J[i]$' and '$S[J[i]] := i$', and *not* the sequence '$Q[i] := J[i]; S[J[i]] := i$' of assignments, is atomic.)

We assume that the number N of processes 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 [20], the proof of the mutual exclusion property of the parameterized Peterson's protocol is left to the reader. The author of [20] simply says that it can be derived from the proof provided for the case of two processes (and, actually, that proof is an informal one) by observing that, for each value of $J[i] = 1, \ldots, N-1$, at least one process is discarded from the set of those which may enter their critical section. Thus, at the end of the for-loop, at most one process may enter its critical section.

In Peterson's protocol, the value of the variable $J[i]$ of process i indicates, as we will now explain, the 'level' that process i has reached since it first requested to enter its critical section (and this request was done by starting the execution of the while-loop with label w, see Figure 1). When process i completes its non-critical section and requests to enter its critical section, it goes to state w where its level $J[i]$ is 1. When process i completes one execution of the body of the while-loop with label w (that is, it goes from state w to state λ and back to state w), it increases its level by one unit. For each level $J[i]=1, \ldots, N-1$, process i tests whether or not property φ holds, and for $J[i]=1, \ldots, N-2$, if φ holds at level $J[i]$, then process i goes to the next level up, that is, $J[i]$ is increased by one unit. If φ holds at the final level $N-1$, then process i enters its critical section.

3 Specification of Parameterized Protocols Using Array Formulas

In this section we present our method for the specification of parameterized protocols and their temporal properties. The main novelty of our method with respect to other methods based on logic programming is that in the specification of protocols we use the first order theory of arrays introduced below.

Similarly to the model checking approach, we represent a protocol as a set of transitions between states. Notice, however, that in the case of parameterized protocols the number of states may be infinite. For the formal specification of the transition relation we consider a *typed first order language* [16] with the following two types: (i) $\mathbb{N}$, denoting natural numbers, and (ii) $\mathbb{A}$, denoting arrays of natural numbers. A *state* is represented by a term of the form $s(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables of type $\mathbb{N}$ or $\mathbb{A}$. The *transition relation* is specified by a set of statements of the form:

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

where t is a fixed binary predicate symbol, a and a' are terms representing states, and τ is an *array formula* defined as we now describe.

An array formula is a typed first order formula constructed by using a language consisting of: (i) *variables* of type $\mathbb{N}$, (ii) *variables* of type $\mathbb{A}$ (called *array variables*), (iii) the constant 0 of type $\mathbb{N}$ and the successor function *succ* of type $\mathbb{N} \rightarrow \mathbb{N}$, and (iv) the following predicates, whose informal meaning is given between parentheses (the names *rd* and *wr* stand for *read* and *write*, respectively):

ln of type $\mathbb{A} \times \mathbb{N}$ ($ln(A, l)$ means 'the array A has length l')
rd of type $\mathbb{A} \times \mathbb{N} \times \mathbb{N}$ ($rd(A, i, n)$ means 'in the array A the i-th element is n')
wr of type $\mathbb{A} \times \mathbb{N} \times \mathbb{N} \times \mathbb{A}$ ($wr(A, i, n, B)$ means 'the array B is equal to the array A except that the i-th element of B is n')

$=_{\mathbb{N}}$, $<$, $\leq$, all of type $\mathbb{N}\times\mathbb{N}$ (equality and inequalities between natural numbers)
$=_{\mathbb{A}}$ of type $\mathbb{A}\times\mathbb{A}$ (equality between arrays)

Given a term n of type $\mathbb{N}$, the term $succ(n)$ will also be written as $n+1$. For reasons of simplicity, we will write $=$, instead of $=_{\mathbb{N}}$ and $=_{\mathbb{A}}$, when the type of the equality is clear from the context.

Array formulas are constructed as usual in typed first order logic by using the connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$, and the quantifiers $\forall$ and $\exists$. However, for every statement of the form $t(a,a') \leftarrow \tau$ which specifies a transition relation, we assume that every array variable occurring in τ is not quantified within τ itself.

The semantics of a statement of the form $t(a,a') \leftarrow \tau$ is defined in a transformational way by transforming this statement into a stratified set of clauses. This set of clauses is obtained by applying the variant of the Lloyd-Topor transformation for typed first order formulas described in [10], called the *typed Lloyd-Topor transformation*. This transformation works like the Lloyd-Topor transformation for untyped first order formulas [16], except that it adds *type atoms* to the bodies of the transformed clauses so that each variable ranges over the domain specified by the corresponding type atom. In our case, the transformation adds the type atoms $nat(N)$ and $array(A)$ for each occurrence of a variable N of type $\mathbb{N}$ and a variable A of type $\mathbb{A}$, respectively. The definition of the predicates *nat* and *array* is provided by the following definite clauses:

$nat(0) \leftarrow$ $\qquad\qquad$ $array([\,]) \leftarrow$
$nat(N{+}1) \leftarrow nat(N)$ $\qquad\qquad$ $array([A|As]) \leftarrow nat(A) \wedge array(As)$

Note that in these clauses arrays are represented as lists. These four clauses are included in a set, called *Arrays*, of definite clauses that also provide the definitions of the predicates *ln*, *rd*, *wr*, $=_{\mathbb{N}}$, $<$, $\leq$, and $=_{\mathbb{A}}$ of our first order language of arrays. In particular, *Arrays* contains the clauses:

$ln([\,],0) \leftarrow$
$ln([A|As],L) \leftarrow L{=}N{+}1 \;\wedge\; ln(As,N)$
$rd([A|As],1,D) \leftarrow A{=}D$
$rd([A|As],L,D) \leftarrow L{=}K{+}1 \;\wedge\; rd(As,K,D)$
$wr([A|As],1,D,[B|Bs]) \leftarrow B{=}D \;\wedge\; As{=}Bs$
$wr([A|As],L,D,[B|Bs]) \leftarrow A{=}B \;\wedge\; L{=}K{+}1 \;\wedge\; wr(As,K,D,Bs)$

We omit to list here the usual clauses defining the predicates $=_{\mathbb{N}}$, $<$, $\leq$, and $=_{\mathbb{A}}$. As an example of application of the typed Lloyd-Topor transformation, let us consider the following statement:

$t(s(A),s(B)) \leftarrow \exists n\, \forall i\; wr(A,i,n,B)$

where: (i) A and B are array variables, and (ii) $s(A)$ and $s(B)$ are terms representing states. By applying the typed Lloyd-Topor transformation to this statement, we get the following two clauses:

$t(s(A),s(B)) \leftarrow array(A) \wedge array(B) \wedge nat(N) \wedge \neg newp(A,N,B)$
$newp(A,N,B) \leftarrow array(A) \wedge nat(I) \wedge nat(N) \wedge array(B) \wedge \neg wr(A,I,N,B)$

Given a statement of the form $H \leftarrow \tau$, where H is an atom and τ is an array formula, we denote by $LT_t(H \leftarrow \tau)$ the set of clauses which are derived by applying the typed Lloyd-Topor transformation to $H \leftarrow \tau$. For reasons of conciseness, in what follows we will feel free to write statements with array formulas, instead of the corresponding set of clauses, and by abuse of language, statements with array formulas will also be called ‘clauses’.

Let us now specify the parameterized Peterson’s protocol for N processes by using statements with array formulas. In this specification a *state* is represented by a term of the form $s(P, J, Q, S)$, where:

– P is an array of the form $[p_1, \ldots, p_N]$ such that, for $i = 1, \ldots, N$, p_i is a constant in the set $\{ncs, cs, w, \lambda\}$ representing the state of process i (see Figure 2). In order to comply with the syntax of array formulas, the constants ncs, cs, w, and λ should be replaced by distinct natural numbers, but, for reasons of readability, in the formulas below we will use the more expressive identifiers ncs, cs, w, and λ.
– J is an array of the form $[j_1, \ldots, j_N]$, where, for $i = 1, \ldots, N$, j_i belongs to the set $\{1, \ldots, N\}$ and is a local value in the sense that it can be read and written by process i only.
– Q and S are arrays of the form $[q_1, \ldots, q_N]$ and $[s_1, \ldots, s_N]$, respectively, where, for $i = 1, \ldots, N$, q_i belongs to the set $\{0, \ldots, N{-}1\}$ and s_i belongs to the set $\{1, \ldots, N\}$. These two arrays Q and S are shared in the sense that they can be read and written by any of the N processes.

The transition relation of the parameterized Peterson’s protocol is defined by the seven statements $T_1, \ldots, T_7$ which we now introduce. For $r = 1, \ldots, 7$, statement T_r is of the form:

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

where $\tau_r(s(P, J, Q, S), s(P', J', Q', S'))$ is an array formula defined as follows (see also Figure 2).

1. For the transition from ncs to w:
$$\begin{aligned}&\tau_1(s(P, J, Q, S), s(P', J', Q', S')) \equiv_{def}\\ &\qquad \exists i\; (rd(P, i, ncs) \;\wedge\; wr(P, i, w, P') \;\wedge\; wr(J, i, 1, J')) \wedge\\ &\qquad Q'{=}Q \;\wedge\; S'{=}S\end{aligned}$$

2. For the transition from w to λ:
$$\begin{aligned}&\tau_2(s(P, J, Q, S), s(P', J', Q', S')) \equiv_{def}\\ &\qquad \exists i, k, l\; (rd(P, i, w) \;\wedge\; wr(P, i, \lambda, P') \;\wedge\; rd(J, i, k) \wedge\\ &\qquad\qquad wr(Q, i, k, Q') \;\wedge\; wr(S, k, i, S') \;\wedge\; ln(P, l) \;\wedge\; k{<}l\,) \wedge\\ &\qquad J'{=}J\end{aligned}$$

3. For the transition from λ to λ:
$$\begin{aligned}&\tau_3(s(P, J, Q, S), s(P', J', Q', S')) \equiv_{def}\\ &\qquad \exists i, k, m, n\; (rd(P, i, \lambda) \;\wedge\; rd(J, i, m) \;\wedge \neg\, (k{=}i) \;\wedge\; rd(Q, k, n) \wedge\\ &\qquad\qquad n{\geq}m \;\wedge\; rd(S, m, i)) \wedge\\ &\qquad P'{=}P \;\wedge\; J'{=}J \;\wedge\; Q'{=}Q \;\wedge\; S'{=}S\end{aligned}$$

4. For the transition from λ to w when $\forall k\, (k \neq i \rightarrow Q[k] < J[i])$ holds:
$$\begin{aligned}&\tau_4(s(P, J, Q, S), s(P', J', Q', S')) \equiv_{def}\\ &\qquad \exists i, l, m\; (rd(P, i, \lambda) \wedge wr(P, i, w, P') \wedge rd(J, i, m) \wedge ln(P, l) \wedge\\ &\qquad\qquad \forall k, n((1{\leq}k{\leq}l \;\wedge\; rd(Q, k, n) \wedge \neg\, (k{=}i)) \;\rightarrow\; n{<}m) \wedge\\ &\qquad\qquad wr(J, i, m{+}1, J')) \wedge\\ &\qquad Q'{=}Q \;\wedge\; S'{=}S\end{aligned}$$

5. For the transition from λ to w when $S[J[i]] \neq i$ holds:
   $\tau_5(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
   $$\exists i,m\ (rd(P,i,\lambda)\ \wedge\ wr(P,i,w,P')\ \wedge$$
   $$rd(J,i,m)\ \wedge\ \neg\, rd(S,m,i)\ \wedge\ wr(J,i,m{+}1,J'))\ \wedge$$
   $$Q'{=}Q\ \wedge\ S'{=}S$$
6. For the transition from w to cs:
   $\tau_6(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
   $$\exists i,m\ (rd(P,i,w)\ \wedge\ wr(P,i,cs,P')\ \wedge$$
   $$rd(J,i,m)\ \wedge\ ln(P,l)\ \wedge\ m{\geq}l)\wedge$$
   $$J'{=}J\ \wedge\ Q'{=}Q\ \wedge\ S'{=}S$$
7. For the transition from cs to ncs:
   $\tau_7(s(P,J,Q,S), s(P',J',Q',S')) \equiv_{def}$
   $$\exists i\ (rd(P,i,cs)\ \wedge\ wr(P,i,ncs,P')\ \wedge\ wr(Q,i,0,Q'))\ \wedge$$
   $$J'{=}J\ \wedge\ S'{=}S$$

We will express the properties of parameterized protocols by using the branching time temporal logic CTL [5]. In particular, the mutual exclusion property of Peterson's protocol will be expressed by the following temporal formula:

$initial \rightarrow \neg\, EF\ unsafe$

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

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

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

Other connectives, such as $\vee$, $\rightarrow$ and $\leftrightarrow$, defined as usual in terms of $\wedge$ and $\neg$, can be used in CTL formulas. The unary constructor *ef* encodes the temporal operator *EF*. Other temporal operators, such as the operator AF which is needed for expressing *liveness* properties, can be defined by using locally stratified logic programs [9,15]. Here, for reasons of simplicity, we have restricted ourselves to the operator *EF* which is the only operator needed for specifying the mutual exclusion property (which is a *safety* property).

The atomic properties of the states are specified by a set of statements of the form:

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

where a is a term representing a state, p is a constant representing an atomic property, and α is an array formula stating that p holds at state a. We assume that the array variables occurring in α are not quantified within α itself. In particular, the *initial* and *unsafe* atomic properties are defined by the following two statements A_1 and A_2.

A_1: $\mathit{atomic}(s(P,J,Q,S), \mathit{initial}) \leftarrow$
$$\exists l\,(\forall k\,(1 \le k \le l \rightarrow (\mathit{rd}(P,k,\mathit{ncs}) \,\wedge\, \mathit{rd}(Q,k,0))) \wedge \mathit{ln}(P,l) \,\wedge\, \mathit{ln}(J,l) \,\wedge\, \mathit{ln}(Q,l) \,\wedge\, \mathit{ln}(S,l))$$

A_2: $\mathit{atomic}(s(P,J,Q,S), \mathit{unsafe}) \leftarrow \exists i,j\,(\mathit{rd}(P,i,\mathit{cs}) \,\wedge\, \mathit{rd}(P,j,\mathit{cs}) \,\wedge \neg\,(j=i))$

The premise of A_1, which will also be denoted by $\mathit{init\_state}(s(P,J,Q,S))$, expresses the fact that in an initial state every process is in its non-critical section, Q is an array whose elements are all 0's, and the arrays P, J, Q, and S have the same length. The premise of A_2, which will also be denoted by $\mathit{unsafe\_state}(s(P,J,Q,S))$, expresses the fact that in an unsafe state at least two distinct processes are in their critical section.

Now we formally define when a CTL formula holds for a specification of a parameterized protocol. Let us consider a protocol specification *Spec* consisting of the following set of statements:

$$\mathit{Spec}: \ \{T_1,\ldots,T_m,A_1,\ldots,A_n\}$$

where: (i) $T_1,\ldots,T_m$ are statements that specify a transition relation, and (ii) $A_1,\ldots,A_n$ are statements that specify atomic properties. We denote by P_{Spec} the following set of clauses:

$$P_{\mathit{Spec}}: \ LT_t(T_1) \cup \ldots \cup LT_t(T_m) \cup LT_t(A_1) \cup \ldots \cup LT_t(A_n) \cup \mathit{Arrays} \cup \mathit{Holds}$$

Given a specification *Spec* of a parameterized protocol and a CTL formula φ, we say that

$$\varphi \ \textit{holds for Spec} \ \text{ iff } \ M(P_{\mathit{Spec}}) \models \forall X \ \mathit{holds}(X,\varphi)$$

where $M(P_{\mathit{Spec}})$ denotes the perfect model of P_{Spec}. Note that the existence of $M(P_{\mathit{Spec}})$ is guaranteed by the fact that P_{Spec} is locally stratified [1]. In the next section we will prove the mutual exclusion property for the parameterized Peterson's protocol by proving that

$$M(P_{\mathit{Peterson}}) \models \forall X \ \mathit{holds}(X, \mathit{initial} \rightarrow \neg\, \mathit{ef}(\mathit{unsafe})) \qquad \text{(ME)}$$

where *Peterson* is the specification of the parameterized Peterson's protocol consisting of the set $\{T_1,\ldots,T_7,A_1,A_2\}$ of statements we have listed above.

Note that the above formula (ME) guarantees the mutual exclusion property of the parameterized Peterson's protocol for any number N (≥ 2) of processes. Indeed, in (ME) the variable X ranges over terms of the form $s(P,J,Q,S)$ and the parameter N of Peterson's protocol is the length of the arrays P, J, Q, and S.

4 Transformational Verification of Parameterized Protocols

In this section we describe our method for the verification of CTL properties of parameterized protocols. This method follows the approach based on program transformation which has been proposed in [21]. As an example of application of our method, we prove that the mutual exclusion property holds for the parameterized Peterson's protocol.

Suppose that, given a specification *Spec* of a parameterized protocol and a CTL property φ, we want to prove that φ holds for *Spec*, that is, $M(P_{\mathit{Spec}}) \models \forall X \ \mathit{holds}(X,\varphi)$. We start off by introducing the statement:

$$\mathit{prop} \leftarrow \forall X\ \mathit{holds}(X, \varphi)$$

where *prop* is a new predicate symbol. By applying the Lloyd-Topor transformation (for untyped formulas) to this statement and by using the equivalence:

$$M(P_{\mathit{Spec}}) \models \forall X, F\ (\neg\, \mathit{holds}(X, F) \leftrightarrow \mathit{holds}(X, \neg\, F))$$

we get the following two clauses:

1. $\mathit{prop} \leftarrow \neg\, \mathit{new}1$
2. $\mathit{new}1 \leftarrow \mathit{holds}(X, \neg\, \varphi)$

Our verification method consists in showing $M(P_{\mathit{Spec}}) \models \forall X\ \mathit{holds}(X, \varphi)$ by applying unfold/fold transformation rules that preserve the perfect model [9,25] and deriving from the program $P_{\mathit{Spec}} \cup \{1, 2\}$ a new program T which contains the clause $\mathit{prop} \leftarrow$.

The soundness of our method is a straightforward consequence of the fact that both the Lloyd-Topor transformation and the unfold/fold transformation rules preserve the perfect model, that is, the following holds:

$$M(P_{\mathit{Spec}}) \models \forall X\ \mathit{holds}(X, \varphi)\ \text{ iff }\ M(P_{\mathit{Spec}} \cup \{1, 2\}) \models \mathit{prop}\ \text{ iff }\ M(T) \models \mathit{prop}$$

Notice that in the case where T contains no clause for *prop*, we conclude that $M(P_{\mathit{Spec}} \cup \{1, 2\}) \not\models \mathit{prop}$ and, thus, $M(P_{\mathit{Spec}}) \models \exists X\ \mathit{holds}(X, \neg\, \varphi)$. Unfortunately, our method is necessarily incomplete due to the undecidability of CTL for parameterized protocols. Indeed, the unfold/fold transformation may not terminate or it may terminate by deriving a program T that contains one or more clauses of the form $\mathit{prop} \leftarrow \mathit{Body}$, where *Body* is not the empty conjunction.

The application of the unfold/fold transformation rules is guided by a transformation strategy which extends the ones presented in [9,21] to the case of logic programs with array formulas. Now we outline this strategy and then we will see it in action in the verification of the mutual exclusion property of the parameterized Peterson's protocol.

Our transformation strategy is divided into two phases, called Phase A and Phase B, respectively.

In Phase A we compute a specialized definition of $\mathit{holds}(X, \neg\, \varphi)$ as we now describe. Starting from clause 2 above, we perform the following transformation steps: (i) we unfold clause 2, thereby deriving a new set, say *Cls*, of clauses, (ii) we manipulate the array formulas occurring in the clauses of *Cls*, by replacing these formulas by equivalent ones and by removing each clause whose body contains an unsatisfiable formula, (iii) we introduce definitions of new predicates and we fold every instance of $\mathit{holds}(X, F)$. Starting from each definition of a new predicate, we repeatedly perform the above three transformation steps (i), (ii), and (iii). We stop when we are able to fold all instances of $\mathit{holds}(X, F)$ by using predicate definitions already introduced at previous transformation steps.

In Phase B we derive a new program T where as many predicates as possible are defined either by a single fact or by an empty set of clauses, in the hope that *prop* is among such predicates. In order to derive program T we use the unfolding rule and the *clause removal* rule. In particular, we remove all clauses that define *useless* predicates [9]. Recall that: (i) the set U of all useless predicates of a program P is defined as the largest set such that for every predicate p in U and for every clause C that defines p in P, there exists a predicate q in U which occurs positively in the body of C, and (ii) the removal of the clauses that define useless predicates preserves the perfect model of the program at hand [9].

This two-phase transformation technique has been fruitfully used for proving properties of infinite state systems in [9].

Let us now show how the mutual exclusion property of the parameterized Peterson's protocol can be verified by using our method based on program transformation. The property φ to be verified is $\mathit{initial} \rightarrow \neg\, \mathit{ef}(\mathit{unsafe})$. Thus, we start from the statement:

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

and by applying the Lloyd-Topor transformation, we get the following two clauses:

1. $\mathit{mutex} \leftarrow \neg\, \mathit{new}1$
2. $\mathit{new}1 \leftarrow \mathit{holds}(X, \mathit{initial} \wedge \mathit{ef}(\mathit{unsafe}))$

Now we apply our transformation strategy starting from $P_{Peterson} \cup \{1, 2\}$, where $P_{Peterson}$ is the program which encodes the specification of the parameterized Peterson's protocol as described in Section 3. Let us now show some of the transformation steps performed during Phase A of this strategy. By unfolding clause 2 we get:

3. $\mathit{new}1 \leftarrow \mathit{init\_state}(s(P, J, Q, S))\ \wedge\ \mathit{unsafe\_state}(s(P, J, Q, S))$
4. $\mathit{new}1 \leftarrow \mathit{init\_state}(s(P, J, Q, S))\ \wedge\ t(s(P, J, Q, S), s(P', J', Q', S')) \wedge$
   $\mathit{holds}(s(P', J', Q', S'), \mathit{ef}(\mathit{unsafe}))$

Clause 3 is removed because the array formula

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

occurring in its body is unsatisfiable (indeed, every process is initially in its non-critical section and, thus, the initial state is not unsafe). In the next section we will discuss the issue of how to mechanize satisfiability tests.

We unfold clause 4 with respect to the atom with predicate t and we get seven new clauses, one for each statement $T_1, \ldots, T_7$ defining the transition relation (see Section 3). The clauses derived from $T_2, \ldots, T_7$ are removed because their bodies contain unsatisfiable array formulas. Thus, after these unfolding steps and removal steps, from clause 2 we get the following clause only:

5. $\mathit{new}1 \leftarrow \mathit{init\_state}(s(P, J, Q, S))\ \wedge\ \tau_1(s(P, J, Q, S), s(P', J', Q', S')) \wedge$
   $\mathit{holds}(s(P', J', Q', S'), \mathit{ef}(\mathit{unsafe}))$

where $\tau_1(s(P, J, Q, S), s(P', J', Q', S'))$ is the array formula defined in Section 3. Now let us consider the formula $c_1(s(P', J', Q', S'))$ defined as follows:

$$\begin{aligned} &c_1(s(P', J', Q', S')) \equiv_{def} \\ &\qquad \exists P, J, Q, S\ (\mathit{init\_state}(s(P, J, Q, S)) \wedge \tau_1(s(P, J, Q, S), s(P', J', Q', S')))\end{aligned}$$

By eliminating from it the existentially quantified variables P, J, Q and S, we obtain the following equivalence:

$$\begin{aligned} &c_1(s(P', J', Q', S')) \leftrightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (C)\\ &\qquad \exists l, i\ (\forall k((1 \le k \le l\ \wedge \neg\,(k = i)) \rightarrow (rd(P', k, ncs)\ \wedge\ rd(Q', k, 0))) \wedge \\ &\qquad\qquad rd(P', i, w)\ \wedge\ rd(J', i, 1)\ \wedge\ rd(Q', i, 0) \\ &\qquad\qquad ln(P', l)\ \wedge\ ln(J', l)\ \wedge\ ln(Q', l)\ \wedge\ ln(S', l))\end{aligned}$$

Now, in order to fold clause 5 w.r.t. the atom $holds(s(P', J', Q', S'), ef(unsafe))$, a suitable condition has to be fulfilled (see the folding rule for constraint logic programs described in [9]). Let us present this condition in the case of programs with array formulas that we consider in this paper.

Suppose that we are given a clause of the form $H \leftarrow \alpha \wedge holds(X, \psi) \wedge G$ and we want to fold it by using a (suitably renamed) clause of the form $newp(X) \leftarrow \beta \wedge holds(X, \psi)$. This folding step is allowed only if we have that $M(Arrays) \models \forall(\alpha \rightarrow \beta)$ holds, that is, $\alpha \wedge \neg\beta$ is unsatisfiable in $M(Arrays)$. If this condition is fulfilled, then by folding we obtain the new clause $H \leftarrow \alpha \wedge newp(X) \wedge G$.

Now, in order to fold clause 5, we introduce a new predicate definition of the form:

6. $new2(s(P, J, Q, S)) \leftarrow genc_1(s(P, J, Q, S)) \wedge holds(s(P, J, Q, S), ef(unsafe))$

The formula $genc_1(s(P, J, Q, S))$ is a *generalization* of $c_1(s(P, J, Q, S))$, in the sense that the following holds:

$$M(Arrays) \models \forall P, J, Q, S\ (c_1(s(P, J, Q, S)) \rightarrow genc_1(s(P, J, Q, S)))$$

This ensures that the condition for folding is fulfilled.

As usual in program transformation techniques, this generalization step from c_1 to $genc_1$ requires ingenuity. We will not address here the problem of how to mechanize this generalization step and the other generalization steps required in the remaining part of our program derivation. However, some aspects of this crucial generalization issue will be discussed in Section 5.

In our verification of the parameterized Peterson's protocol we introduce the following array formula $genc_1(s(P, J, Q, S))$ which holds iff zero or more processes are in state w and the remaining processes are in state ncs:

$$\begin{aligned} & genc_1(s(P, J, Q, S)) \equiv_{def} \qquad\qquad (G)\\ & \quad \exists l\ (\forall k\ (1 \le k \le l\ \rightarrow ((rd(P, k, ncs)\ \wedge\ rd(Q, k, 0)) \vee \\ & \qquad\qquad (rd(P, k, w)\ \wedge\ rd(J, k, 1)\ \wedge\ rd(Q, k, 0)))) \wedge \\ & \qquad ln(P, l)\ \wedge\ ln(J, l)\ \wedge\ ln(Q, l)\ \wedge\ ln(S, l)) \end{aligned}$$

This formula defining $genc_1(s(P, J, Q, S))$ is indeed a generalization of the formula $c_1(s(P, J, Q, S))$, as the reader may check by looking at the above equivalence (C). By folding clause 5 using the newly introduced clause 6 we get:

5.f $new1 \leftarrow init\_state(s(P, J, Q, S))\ \wedge\ \tau_1(s(P, J, Q, S), s(P', J', Q', S')) \wedge$
$new2(s(P', J', Q', S'))$

Now, starting from clause 6, 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 transformation step. By doing so we terminate Phase A and we derive the following program R where:

- $genc_1$ is defined as indicated in (G),
- $genc_2, \ldots, genc_8$ are defined as indicated in the Appendix,
- $\tau_1, \ldots, \tau_7$ are the array formulas that define the transition relation as indicated in Section 3, and
- the arguments a and a' stand for the states $s(P, J, Q, S)$ and $s(P', J', Q', S')$, respectively.

1. $mutex \leftarrow \neg\, new1$ Program R

5.f $new1 \leftarrow initial(a) \wedge \tau_1(a,a') \wedge new2(a')$

7. $new2(a) \leftarrow genc_1(a) \wedge \tau_1(a,a') \wedge new2(a')$
8. $new2(a) \leftarrow genc_1(a) \wedge \tau_2(a,a') \wedge new3(a')$
9. $new2(a) \leftarrow genc_1(a) \wedge \tau_6(a,a') \wedge new7(a')$

10. $new3(a) \leftarrow genc_2(a) \wedge \tau_1(a,a') \wedge new3(a')$
11. $new3(a) \leftarrow genc_2(a) \wedge \tau_2(a,a') \wedge new3(a')$
12. $new3(a) \leftarrow genc_2(a) \wedge \tau_3(a,a') \wedge new3(a')$
13. $new3(a) \leftarrow genc_2(a) \wedge \tau_4(a,a') \wedge new4(a')$
14. $new3(a) \leftarrow genc_2(a) \wedge \tau_5(a,a') \wedge new5(a')$

15. $new4(a) \leftarrow genc_3(a) \wedge \tau_1(a,a') \wedge new4(a')$
16. $new4(a) \leftarrow genc_3(a) \wedge \tau_2(a,a') \wedge new4(a')$
17. $new4(a) \leftarrow genc_3(a) \wedge \tau_2(a,a') \wedge new6(a')$
18. $new4(a) \leftarrow genc_3(a) \wedge \tau_4(a,a') \wedge new4(a')$
19. $new4(a) \leftarrow genc_3(a) \wedge \tau_6(a,a') \wedge new7(a')$

20. $new5(a) \leftarrow genc_4(a) \wedge \tau_1(a,a') \wedge new5(a')$
21. $new5(a) \leftarrow genc_4(a) \wedge \tau_2(a,a') \wedge new5(a')$
22. $new5(a) \leftarrow genc_4(a) \wedge \tau_3(a,a') \wedge new5(a')$
23. $new5(a) \leftarrow genc_4(a) \wedge \tau_4(a,a') \wedge new5(a')$
24. $new5(a) \leftarrow genc_4(a) \wedge \tau_5(a,a') \wedge new5(a')$
25. $new5(a) \leftarrow genc_4(a) \wedge \tau_6(a,a') \wedge new8(a')$

26. $new6(a) \leftarrow genc_5(a) \wedge \tau_1(a,a') \wedge new6(a')$
27. $new6(a) \leftarrow genc_5(a) \wedge \tau_2(a,a') \wedge new6(a')$
28. $new6(a) \leftarrow genc_5(a) \wedge \tau_3(a,a') \wedge new6(a')$
29. $new6(a) \leftarrow genc_5(a) \wedge \tau_4(a,a') \wedge new6(a')$
30. $new6(a) \leftarrow genc_5(a) \wedge \tau_5(a,a') \wedge new6(a')$
31. $new6(a) \leftarrow genc_5(a) \wedge \tau_6(a,a') \wedge new9(a')$

32. $new7(a) \leftarrow genc_6(a) \wedge \tau_1(a,a') \wedge new7(a')$
33. $new7(a) \leftarrow genc_6(a) \wedge \tau_2(a,a') \wedge new9(a')$
34. $new7(a) \leftarrow genc_6(a) \wedge \tau_7(a,a') \wedge new2(a')$

35. $new8(a) \leftarrow genc_7(a) \wedge \tau_3(a,a') \wedge new8(a')$
36. $new8(a) \leftarrow genc_7(a) \wedge \tau_7(a,a') \wedge new5(a')$

37. $new9(a) \leftarrow genc_8(a) \wedge \tau_1(a,a') \wedge new9(a')$
38. $new9(a) \leftarrow genc_8(a) \wedge \tau_2(a,a') \wedge new9(a')$
39. $new9(a) \leftarrow genc_8(a) \wedge \tau_3(a,a') \wedge new9(a')$
40. $new9(a) \leftarrow genc_8(a) \wedge \tau_5(a,a') \wedge new9(a')$
41. $new9(a) \leftarrow genc_8(a) \wedge \tau_7(a,a') \wedge new6(a')$

Now we proceed to Phase B of our strategy. Since in program R the predicates $new1$ through $new9$ are useless, we remove clause 5.f, and clauses 7 through 41, and by doing so, we derive a program consisting of clause 1 only. By unfolding clause 1 we get the final program T, which consists of the clause $\mathit{mutex} \leftarrow$ only. Thus, $M(T) \models \mathit{mutex}$ and we have proved that:

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

As a consequence, we have that for any initial state and for any number $N(\geq 2)$ of processes, the mutual exclusion property holds for the parameterized Peterson's protocol.

5 Mechanization of the Verification Method

In order to achieve a full mechanization of our verification method, two main issues have to be addressed: (i) how to test the satisfiability of array formulas, and (ii) how to perform suitable generalization steps.

Satisfiability tests for array formulas are required at the following two points of Phase A of the transformation strategy described in Section 4: (1) at Step (ii), when we remove each clause whose body contains an unsatisfiable array formula, and (2) at Step (iii), when we fold each clause whose body contains a *holds* literal.

In order to clarify Point (2), we recall that, before applying the folding rule [9], we need to test that in $M(\mathit{Arrays})$ the array formula occurring in the body of the clause to be folded implies the array formula occurring in the body of the clause that we use for folding. For instance, in Section 4 before folding clause 5 using clause 6, we need to prove that:

$$M(\mathit{Arrays}) \models \forall P, J, Q, S\ (c_1(s(P, J, Q, S)) \rightarrow \mathit{genc}_1(s(P, J, Q, S)))$$

which holds iff the following formula:

$$c_1(s(P, J, Q, S)) \wedge \neg\, \mathit{genc}_1(s(P, J, Q, S)) \qquad (CG)$$

is unsatisfiable in $M(\mathit{Arrays})$.

Now the problem of testing the satisfiability of array formulas is in general undecidable. (The reader may refer to [28] for a short survey on this subject.) However, some decidable fragments of the theory of arrays, such as the *quantifier-free extensional theory of arrays*, have been identified [28]. Unfortunately, the array formulas occurring in our formalization of the parameterized Peterson's protocol cannot be reduced to formulas in those decidable fragments. Indeed, due to the assumptions made in Section 3 on the array formulas which are used in the specifications of protocols, we need to test the satisfiability of array formulas where the variables of type $\mathbb{A}$ are not quantified, while the variables of type $\mathbb{N}$ can be quantified in an unrestricted way.

In order to perform the satisfiability tests required by our verification of the parameterized Peterson's protocol, we have followed the approach based on program transformation which has been proposed in [21]. Some of these satisfiability tests have been done in a fully automatic way by using the MAP transformation system [17], which implements the unfold/fold proof strategy described in [21]. Examples of array formulas whose unsatisfiability we have proved in an automatic way include: (i) the formula occurring in the body of clause 3 shown in Section 4, and (ii) the formula (CG) shown above in this section. Some other satisfiability tests have been done in a semi-automatic way, by interleaving fully automatic applications of the unfold/fold proof strategy and some manual applications of the unfold/fold transformation rules.

Generalization steps are needed when, during Step (iii) of Phase A of our transformation strategy, a new predicate definition is introduced to fold the instances of the atom $\mathit{holds}(X, F)$. The introduction of suitable new definitions by generalization is a crucial issue of the program transformation methodology [4]. The invention of these definitions corresponds to the discovery of suitable invariants of the protocol to be verified. Due to the undecidability of CTL for parameterized protocols, it is impossible to provide a general, fully automatic technique which performs the suitable generalization steps in all cases. However, we have followed an approach that, in the case of the parameterized Peterson's protocol, works in a systematic way. This approach extends to the case of logic programs with array formulas some generalization techniques which are used for the specialization of (constraint) logic programs [8,14] and it can be briefly described as follows.

The new predicate definitions introduced during Step (iii) of Phase A of the transformation strategy are arranged as a tree *DefsTree* of clauses. The root of *DefsTree* is clause 2. Given a clause N, the children of N are the predicate definitions which are introduced to fold the instances of $\mathit{holds}(X, F)$ in the bodies of the clauses obtained by unfolding N at Step (i) and not removed at Step (ii).

If the new predicate definitions are introduced without any guidance, then we may construct a tree *DefsTree* with infinite paths, and the transformation strategy may not terminate. In order to avoid the construction of infinite paths and achieve the termination of the transformation strategy, before adding a new predicate definition D to *DefsTree* as a child of a clause N, we match D against every clause A occurring in the path from the root of *DefsTree* to N. Suppose that A is of the form $\mathit{newp}(X) \leftarrow \alpha \wedge \mathit{holds}(X, \psi)$ and D is of the form $\mathit{newq}(X) \leftarrow \delta \wedge \mathit{holds}(X, \psi)$. If the array formula α is *embedded* (with respect to a suitable ordering) into the array formula δ, then instead of introducing D, we introduce a clause of the form $\mathit{gen}(X) \leftarrow \gamma \wedge \mathit{holds}(X, \psi)$, where γ is a generalization of both α and δ, that is, both $M(\mathit{Arrays}) \models \forall\ (\alpha \rightarrow \gamma)$ and $M(\mathit{Arrays}) \models \forall\ (\delta \rightarrow \gamma)$ holds.

Thus, in order to fully mechanize our generalization technique we need to address the following two problems: (i) the introduction of a formal definition of the *embedding* relation between array formulas, and (ii) the computation of the array formula γ from α and δ. Providing solutions to these two problems is beyond the scope of the present paper. However, a possible approach to follow for solving these problems consists in extending to logic programs with array formulas the notions that have been introduced in the area of specialization of (constraint) logic programs (see, for instance, [8,14]).

6 Related Work and Conclusions

The method for protocol verification presented in this paper is based on the program transformation approach which has been proposed in [21] for the verification of properties of locally stratified logic programs. We consider concurrent systems of *finite state* processes. We assume that systems are *parameterized*, in the sense that they consist of an *arbitrary* number of processes. We also assume that parameterized systems may use arrays of parameterized length. The properties of the systems we want to verify, are the temporal logic properties which can be expressed in CTL (Computational Tree Logic) [5]. Our method consists in: (i) encoding a parameterized system and the property to be verified as a locally stratified logic program extended with array formulas, and then (ii) applying suitable unfold/fold transformations to this program so to derive a new program where it is immediate to check whether or not the property holds.

In general, the problem of verifying CTL properties of parameterized systems is undecidable [2] and thus, in order to find decision procedures, one has to consider subclasses of systems where the problem is decidable. Some of these decidable subclasses in the presence of arrays have been studied in [13], but unfortunately, our formalization of the parameterized Peterson's protocol does not belong to any of those classes, because it requires more than two arrays of natural numbers, and also requires assignments and reset operations.

As yet, our method is *not* fully mechanical and human intervention is needed for: (i) the test of satisfiability for array formulas, and (ii) the introduction of new definitions by generalization. We have discussed these two issues in Section 5.

Other verification methods for concurrent systems based on the transformational approach are those presented in [9,10,15,23,24].

In [9] it is presented a method for verifying CTL properties of systems consisting of a *fixed number* of infinite state processes. That method 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 an approach similar to constraint logic programming, but in our setting we have array formulas, instead of constraints. The method presented here can easily be extended to deal with parameterized infinite state systems by considering, for instance, arrays of infinite state processes.

The paper [10] describes the verification of the mutual exclusion property for the parameterized Bakery protocol which was introduced in [12]. In [10] the authors use locally stratified logic programs extended with formulas of the Weak Monadic Second Order Theory of k-Successors (WSkS) which expresses monadic properties of strings. The array formulas considered in this paper are more expressive than WSkS formulas, because array formulas can express polyadic properties. However, there is price to pay, because in general the theory of array formulas is undecidable, while the theory WSkS is decidable.

The method described in [15] 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 easily mechanizable and yet powerful enough for the verification of program properties.

The work presented in [23,24] is the most similar to ours. The authors of these two papers use unfold/fold rules for transforming programs and proving properties of parameterized concurrent systems. Our paper differs from [23,24] 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 [23,24].

Besides the above mentioned transformational methods, some more verification methods based on (constraint) logic programming have been proposed in the literature [7,11,19,22].

The methods proposed in [19,22] deal with finite state systems only. In particular, the method presented in [19] uses constraint logic programming with finite domains, extended with constructive negation and tabled resolution, for finite state local model checking, and the method described in [22] uses tabled logic programming to efficiently verify μ-calculus properties of finite state systems expressed in the CCS calculus.

The methods presented in [7,11] deal with infinite state systems. In particular, [7] describes a method which is based on constraint logic programming and can be applied for verifying CTL properties of infinite state systems by computing approximations of least and greatest fixpoints via abstract interpretation. An extension of this method has also been used for the verification of parameterized cache coherence protocols [6]. The method described in [11] uses logic programs with linear arithmetic constraints and Presburger arithmetic for the verification of safety properties of Petri nets. Unfortunately, however, parameterized systems that use arrays, like Peterson's protocol, cannot be directly specified and verified using the methods considered in [7,11] because, in general, array formulas cannot be encoded as constraints over real numbers or Presburger formulas.

Several other verification techniques for parameterized systems have been proposed in the literature outside the area of logic programming (see [29] for a survey of some of these techniques). These techniques extend finite state model checking with various forms of *abstraction* (for reducing the verification of a parameterized system to the verification of a finite state system) or *induction* (for proving properties for every value of the parameter).

We do not have space here to discuss the relationships of our work with the many techniques for proving properties based on abstraction. We only want to mention the technique proposed in [3], which has also been applied for the verification of the parameterized Peterson's protocol. That technique can be applied for verifying in an automatic way safety properties of all systems that satisfy a so-called *stratification* condition. Indeed, when this condition holds for a given parameterized system, then the verification task can be reduced to the verification of a finite 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 [3] requires a significant amount of ingenuity.

Our verification method is also related to the verification techniques based on induction (see, for instance, [18]). These techniques use interactive theorem proving tools where many tasks are mechanized, but the construction of a whole proof requires substantial human guidance. Our method has advantages and disadvantages with respect to these techniques based on induction. On one hand, in our approach we need neither explicit induction on the parameter of the system nor the introduction of suitable induction hypotheses. On the other hand, as already mentioned, our method needs suitable generalization steps which cannot be fully mechanized.

Appendix

Below we give the definitions of the array formulas $genc_2$ through $genc_8$ occurring in the program R of Section 4.

$$
\begin{aligned}
& genc_2(s(P,J,Q,S)) \equiv_{def} \\
& \quad \exists i,l(rd(P,i,\lambda) \wedge l>1 \wedge rd(J,i,1) \wedge rd(Q,i,1) \wedge rd(S,1,i) \wedge \\
& \qquad \forall k(1\leq k\leq l \rightarrow ((rd(P,k,ncs) \wedge rd(Q,k,0)) \vee \\
& \qquad\qquad (rd(P,k,w) \wedge rd(J,k,1) \wedge rd(Q,k,0)) \vee \\
& \qquad\qquad (rd(P,k,\lambda) \wedge rd(J,k,1) \wedge rd(Q,k,1)))) \wedge \\
& \qquad ln(P,l) \wedge ln(J,l) \wedge ln(Q,l) \wedge ln(S,l))
\end{aligned}
$$

$$
\begin{aligned}
& genc_3(s(P,J,Q,S)) \equiv_{def} \\
& \quad \exists i,k,l(2\leq k<l \wedge ((rd(P,i,w) \wedge rd(J,i,k+1) \wedge rd(Q,i,k) \wedge rd(S,k,i)) \vee \\
& \qquad\qquad (rd(P,i,\lambda)\wedge rd(J,i,k)\wedge rd(Q,i,k)\wedge rd(S,k,i))) \wedge \\
& \qquad \forall j((1\leq j\leq l \wedge \neg(j=i)) \rightarrow ((rd(P,j,ncs) \wedge rd(Q,j,0)) \vee \\
& \qquad\qquad (rd(P,j,w) \wedge rd(J,j,1) \wedge rd(Q,j,0)))) \wedge \\
& \qquad ln(P,l) \wedge ln(J,l) \wedge ln(Q,l) \wedge ln(S,l))
\end{aligned}
$$

$$
\begin{aligned}
& genc_4(s(P,J,Q,S)) \equiv_{def} \\
& \quad \exists m,l(1\leq m\leq l \wedge \forall k(1\leq k<m \rightarrow \\
& \qquad\qquad \exists i(rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \wedge \\
& \qquad \forall j(1\leq j\leq l \rightarrow ((rd(P,j,ncs) \wedge rd(Q,j,0)) \vee \\
& \qquad\qquad \exists k(1\leq k<m \wedge ((rd(P,j,w) \wedge rd(J,j,k+1) \wedge rd(Q,j,k)) \vee \\
& \qquad\qquad\qquad (rd(P,j,\lambda) \wedge rd(J,j,k) \wedge rd(Q,j,k)))))) \wedge \\
& \qquad ln(P,l) \wedge ln(J,l) \wedge ln(Q,l) \wedge ln(S,l))
\end{aligned}
$$

$$
\begin{aligned}
& genc_5(s(P,J,Q,S)) \equiv_{def} \\
& \quad \exists i,k,l(2\leq k<l \wedge ((rd(P,i,w) \wedge rd(J,i,k+1) \wedge rd(Q,i,k) \wedge rd(S,k,i)) \vee \\
& \qquad\qquad (rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \wedge \\
& \qquad \exists m(1\leq m \leq k \wedge \\
& \qquad\quad \forall u(1\leq u\leq m \rightarrow \\
& \qquad\qquad \exists j(rd(P,j,\lambda) \wedge rd(J,j,u) \wedge rd(Q,j,u) \wedge rd(S,u,j))) \wedge \\
& \qquad\quad \forall n((1\leq n\leq l \wedge \neg(n=i)) \rightarrow \\
& \qquad\qquad ((rd(P,n,ncs) \wedge rd(Q,n,0)) \vee \\
& \qquad\qquad \exists r(1\leq r\leq m \wedge \\
& \qquad\qquad\quad ((rd(P,n,w) \wedge rd(J,n,r+1) \wedge rd(Q,n,r)) \vee \\
& \qquad\qquad\quad (rd(P,n,\lambda) \wedge rd(J,n,r) \wedge rd(Q,n,r))))))) \wedge \\
& \qquad ln(P,l) \wedge ln(J,l) \wedge ln(Q,l) \wedge ln(S,l))
\end{aligned}
$$

$$
\begin{array}{l}
genc_6(s(P,J,Q,S)) \equiv_{def} \\
\quad \exists i,l,u(rd(P,i,cs) \wedge rd(J,i,u{+}1) \wedge rd(Q,i,u) \wedge rd(S,u,i) \ \wedge \ u{+}1{=}l \ \wedge \\
\qquad\qquad \forall j((1{\leq}j{\leq}l \wedge \neg\,(j{=}i)) \rightarrow ((rd(P,k,ncs) \wedge rd(Q,k,0)) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (rd(P,k,w) \wedge rd(J,k,1) \wedge rd(Q,k,0)))) \ \wedge \\
\qquad\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{array}
$$

$$
\begin{array}{l}
genc_7(s(P,J,Q,S)) \equiv_{def} \\
\quad \exists i,l,u(rd(P,i,cs) \wedge rd(J,i,u{+}1) \wedge rd(Q,i,u) \ \wedge \ u{+}1{=}l \ \wedge \\
\qquad\qquad \forall k(1{\leq}k{<}l \ \rightarrow \ \exists i(rd(P,i,\lambda) \wedge rd(J,i,k) \wedge rd(Q,i,k) \wedge rd(S,k,i))) \ \wedge \\
\qquad\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{array}
$$

$$
\begin{array}{l}
genc_8(s(P,J,Q,S)) \equiv_{def} \\
\quad \exists i,l,u(rd(P,i,cs) \wedge rd(J,i,u{+}1) \wedge rd(Q,i,u) \ \wedge \ u{+}1{=}l \ \wedge \\
\qquad\qquad \exists m(1{\leq}m{\leq}l \wedge \forall n(1{\leq}n{<}m \ \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \exists j(rd(P,j,\lambda) \wedge rd(J,j,n) \wedge rd(Q,j,n) \wedge rd(S,n,j))) \ \wedge \\
\qquad\qquad\quad \forall j((1{\leq}j{\leq}l \wedge \neg\,(j{=}i)) \rightarrow \\
\qquad\qquad\qquad ((rd(P,j,ncs) \wedge rd(Q,j,0)) \vee \\
\qquad\qquad\qquad\quad \exists k(1{\leq}k{<}m \ \wedge \\
\qquad\qquad\qquad\qquad ((rd(P,j,w) \wedge rd(J,j,k{+}1) \wedge rd(Q,j,k)) \vee \\
\qquad\qquad\qquad\qquad\ (rd(P,j,\lambda) \wedge rd(J,j,k) \wedge rd(Q,j,k))))))) \ \wedge \\
\qquad\qquad ln(P,l) \ \wedge \ ln(J,l) \ \wedge \ ln(Q,l) \ \wedge \ ln(S,l))
\end{array}
$$

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. K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.
3. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proceedings of CAV 2001*, Lecture Notes in Computer Science 2102, pages 221–234. Springer, July 2001.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
6. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
7. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In K.-K. Lau, editor, *Proceedings of LOPSTR 2000, London, UK, 24-28 July, 2000*, LLNCS 2042, pages 125–146. Springer, 2001.
9. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of VCL '01, Florence, Italy*, DSSE-TR-2001-3, pages 85–96. Univ. of Southampton, UK, 2001.
10. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state systems using program transformation. In *Proceedings of LOPSTR '01*, Lecture Notes in Computer Science 2372, pages 111–128. Springer, 2002.

11. 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.
12. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
13. R. Lazic, T. C. Newcomb, and A. W. Roscoe. On model checking data-independent systems with arrays with whole-array operations. In *Communicating Sequential Processes: The First 25 Years*, LNCS 3525, pages 275–291. Springer, 2004.
14. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.
15. 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*, LNCS 1817, pages 63–82. Springer, 1999.
16. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.
17. MAP group. The MAP transformation system. Available from: `http://www.iasi.rm.cnr.it/~proietti/system.html`, 1995–2005.
18. K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *Proceedings of CAV '00*, LNCS 1855, pages 312–327. Springer, 2000.
19. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proceedings of CL '00*, LNAI 1861, pp. 384–398. Springer, 2000.
20. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
21. 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*, LNAI 1861, pages 613–628. Springer, 2000.
22. 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 *Proceedings of CAV '97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
23. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *Proceedings of CAV '01*, pages 25–37, 2001.
24. 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*, LNCS 3049, pages 261–290. Springer, 2004.
25. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
26. V. Senni. Transformational verification of the parameterized Peterson's protocol. Unpublished note, July 2005.
27. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000*, LNCS 1877, pages 1–16, Springer, 2000.
28. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS '01*, pages 29–37. IEEE Press, 2001.
29. 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.