Verification of Sets of Infinite State Processes Using Program Transformation*

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[1]

(1) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
(2) DISP, University of Roma Tor Vergata, I-00133 Roma, Italy
{fioravanti,adp,proietti}@iasi.rm.cnr.it

(Extended Abstract)

1 Introduction

This paper describes a method for verifying properties of systems which are finite sets of infinite state processes (see, for instance, [13,16,20]). Our method is an enhancement of the one presented in [8]. Systems and properties are specified by using *Constraint Logic Programming* [10] (CLP for short), and the inference engine for verifying properties is provided by a technique based on *Unfold/Fold Transformations* [3,15,21]. The enhancement we propose in this paper allows us to deal with a finite set of processes of arbitrary cardinality, rather than a finite set of processes of fixed cardinality. An essential feature of our method is that we consider CLP programs where the constraint theory is the *Weak Monadic Second Order Theory* of k successors, denoted by WSkS (see [23] for a brief presentation). Within this theory each process in a given state is represented as a finite string, and relations between finite sets of processes are expressed by using formulas of WSkS (see Section 3).

A *system* is a finite set of processes which evolve in parallel. At any given instant in time, one process evolves by making a transition from a state to a new state. Correspondingly, the system makes a transition from a finite set of states to a new finite set of states, or as we say, from a *configuration* to a new configuration. We assume that each process may be infinite state, that is, during its evolution it may be in an infinite number of distinct states. Thus, the set of possible configurations is infinite.

The evolution of a system S starting from an initial configuration c is formalized as a *computation tree*, called T_S, which is defined as follows: (i) the root of T_S is c, and (ii) every node c_i of T_S has a child node c_j iff there exists in S a transition from configuration c_i to configuration c_j. We assume that in every computation tree, every node has at least one child node.

A property of a system is expressed as a formula of the *Computational Tree Logic* [4] (CTL, for short). CTL is a branching time temporal logic whose formulas are built from elementary properties by using: (i) the logical connectives $\neg$, $\wedge$, and $\vee$, (ii) the quantifiers over paths: A ('for all paths') and E ('for some path'), and (iv) the quantifiers along paths: G ('for all configurations on the path'), F

* In Pre-Proceedings LOPSTR'01, Paphos, Cyprus, 28-30 November, 2001.

('for some configuration on the path'). The notion of truth of a CTL formula φ in a configurations c of a system S, is formalized via the satisfaction relation $T_S, c \models \varphi$, where T_S is the computation tree for S whose root is c. This relation is inductively defined according to the structure of φ. We assume to know the value of $T_S, c \models \varphi$ for each elementary property φ.

Now we outline our method for verifying that, for all initial configurations c of a given system S, the property φ holds.

Verification Method.

Step 1. (*System and Property Specification*) We introduce: (i) a constraint $init(X)$ which characterizes the initial configurations, that is, c is an initial configuration iff $init(c)$ holds, and (ii) a locally stratified CLP program P_S which defines a binary predicate sat such that for all configurations c,

$$T_S, c \models \varphi \text{ iff } sat(c, \varphi) \in M(P_S)$$

where $M(P_S)$ denotes the perfect model of the program P_S [1].

Step 2. (*Proof Method*) We introduce a new predicate f defined by the clause F: $f(X) \leftarrow init(X), sat(X, \varphi)$, where X is a variable. We then apply the transformation rules, and from program $P_S \cup \{F\}$ we derive a new program P_f. If the clause $f(X) \leftarrow init(X)$ occurs in P_f then for all initial configurations c, we have that $T_S, c \models \varphi$ holds.

Some remarks are now in order.

Remark 1. The choice of the perfect model as a semantics of the program P_S is due to the fact that, by definition, $T_S, c \models \neg\varphi$ holds iff $T_S, c \models \varphi$ does not hold, and this fact can be expressed by the clause:

$$N\colon\ sat(X, \neg\varphi) \leftarrow \neg\, sat(X, \varphi)$$

where $\neg$ in the head is interpreted as a function symbol, while $\neg$ in the body is interpreted as negation by (finite or infinite) failure. Since clause N is locally stratified and the other clauses for sat will not contain negated atoms (see Section 3.3), the semantics of negation by failure is captured by the perfect model (recall that for locally stratified programs the perfect model is equal to the stable model and the well-founded model [1]).

Remark 2. The correctness of our verification method is a consequence of the following two facts: (i) the transformation rules preserve perfect models, and (ii) perfect models are models of the completion of a program.

Remark 3. Our verification method is incomplete, in the sense that there exist a system S, a configuration c, and a property φ, such that $T_S, c \models \varphi$ holds, and yet there is no sequence of applications of the transformation rules which leads from the program $P_S \cup \{f(X) \leftarrow init(X), sat(X, \varphi)\}$ to the program P_f containing the clause $f(X) \leftarrow init(X)$. This incompleteness limitation cannot be overcome, because the problem of verifying properties of finite sets of infinite state processes is undecidable and not semidecidable. This is a consequence of the fact that the uniform verification of parameterized systems consisting of finite state processes is undecidable [2].

We will illustrate our verification method by considering the case of a system of N processes which use the *bakery protocol* for ensuring mutual exclusion [11]. This protocol is described in Section 2. In Section 3 we indicate how Step 1 of our verification method is realized. For the definition of the program P_S we introduce constraint logic programs whose constraints are formulas of the WSkS theory. These programs will be called CLP(WSkS) programs. The WSkS theory is a well known decidable theory [22] and it allows us to express properties of finite sets of finite strings over an alphabet of k symbols. We will use WSkS for encoding both the elementary properties of the configurations and the transition relation between configurations. In Section 4 we show how Step 2 of our verification method is realized by applying suitable rules for program transformation. These rules are adaptations to the case of CLP(WSkS) of the unfold/fold rules for generic CLP programs similar to those presented in [7,8].

2 The N-Process Bakery Protocol for Mutual Exclusion

In this section we describe the N-process bakery protocol. The techniques we will use for our description are tailored to this protocol. However, we think that they are quite general and can be used for many other systems and protocols.

The N-process bakery protocol is a protocol for ensuring mutual exclusion in a system S made out of N processes which use a shared resource. The mutual exclusion property is ensured iff the shared resource is used by at most one process at a time.

A *state* of a process p in the system S is the pair $\langle n_p,\ s_p\rangle$, where n_p is a *counter*, that is, a natural number, and s_p is a *control state*, that is, an element of the set $\{t, w, u\}$ (short for *think*, *wait*, and *use*, respectively). The transitions for process p in S are the following ones:
(1) from state $\langle n_p,\ t\rangle$ to state $\langle m{+}1,\ w\rangle$, where m is the maximum value of the counters of the processes in S;
(2) from state $\langle n_p,\ w\rangle$ to state $\langle n_p,\ u\rangle$, iff for any other process q in S, different from p, we have that either $n_q = 0$ or $n_p < n_q$;
(3) from state $\langle n_p,\ u\rangle$ to state $\langle 0,\ t\rangle$.

A *configuration* c of the system S is a set I_c of N distinct elements, each one corresponding to a process. For reason of simplicity, we will identify each element of I_c with its corresponding process. We assume that, for each configuration, the following two functions are defined:
(i) the *counter function* cn from I_c to the set of natural numbers such that, for every process $p \in I_c$, cn returns the value of the counter of the process p, and
(ii) the *control state function* cs from I_c to $\{t, w, u\}$ such that for every process $p \in I_c$, cs returns the value of the control state of p.

An initial configuration c of the system S is any set I_c of N distinct elements such that cn is the constant function returning 0 and cs is the constant function returning t. The transition relation between configurations is defined as follows. Given a configuration $c1$, if process p in I_{c1} makes a transition from state $\langle n_1,\ s_1\rangle$ to state $\langle n_2,\ s_2\rangle$, then there is a transition from configuration $c1$ to the new

configuration $c2$ which is equal to $c1$ except that I_{c2} is obtained from I_{c1} by substituting p by a new element $\tilde{p}$ such that $cn(\tilde{p}) = n_2$ and $cs(\tilde{p}) = s_2$.

We want to verify the mutual exclusion property for the bakery protocol which says that the system S will never reach a configuration where at least two processes use the shared resource. This safety property can be expressed by the CTL formula $\neg EF(\mathit{unsafe})$, where *unsafe* is an elementary property which holds in a configuration c iff there are at least two distinct processes $p_1, p_2 \in I_c$ such that $cs(p_1) = cs(p_2) = u$.

Formally, we need to prove that for all initial configurations c of the system S we have that $T_S, c \models \neg EF\ \mathit{unsafe}$ holds. Indeed, $T_S, c \models \neg EF\ \mathit{unsafe}$ means that in the computation tree for S whose root is the configuration c, there is no path which starts from c and has a configuration where *unsafe* is true.

3 System Specification using Monadic Second Order Logic and CLP

Given a system of processes, in order to specify the elementary properties of its configurations and the transitions from configuration to configuration, we use the weak monadic second order logic of k successors, denoted WSkS [23]. We do so by first encoding a process as a string and a system configuration as a finite set of strings. Then, since WSkS allows us to express properties of finite sets of finite strings, we express the elementary properties of the configurations and the transitions from configuration to configuration by using WSkS formulas.

In order to specify CTL properties of the configurations (that is, the *sat* relation indicated at Step 1 of our verification method), we will introduce CLP programs whose constraints are WSkS formulas.

3.1 The Logic WSkS

The syntax of WSkS is defined as follows. The *first order terms* (or simply, *terms*) of WSkS are constructed out of the constant ε, the individual variables $x, y, z, \ldots$, and the k unary function symbols $s_1, \ldots, s_k$. We will use the postfix notation so that, for instance, $s_1(s_2(\varepsilon))$ is written an $\varepsilon s_2 s_1$. We will omit the ε when no confusion arises. For instance, $\varepsilon s_1 s_1 s_2 s_4 s_1$ and $x s_2 s_1 s_2$ are terms, and $\varepsilon s_1 s_1 s_2 s_4 s_1$ is also written as $s_1 s_1 s_2 s_4 s_1$.

Set variables are denoted by $X, Y, Z, \ldots$

Atomic formulas are: either (i) equalities between terms $t_1 = t_2$, or (ii) inequalities between terms $t_1 \leq t_2$, or (iii) membership atomic formulas $t \in X$, where t is a term and X is a set variable.

The *formulas* of WSkS are constructed from the atomic formulas by using the usual logical connectives and the quantifiers over individual variables and set variables.

The *semantics* of WSkS formulas is defined by considering the structure $\mathcal{D}_k$ which is the tuple $\langle \{s_1, \ldots, s_k\}^*, =, \leq, \in \rangle$. The carrier $\{s_1, \ldots, s_k\}^*$ of $\mathcal{D}_k$ is the set of the finite strings of successor symbols, $=$ denotes string equality, $\leq$ denotes

prefix ordering over strings, and $\in$ denotes membership of a string to a *finite* set of strings. (For simplicity, we use the same notation for the predicate symbols and their semantic interpretation.) We say that a closed formula φ of WSkS holds iff $\mathcal{D}_k \models \varphi$.

3.2 Specifying the Configurations and the Transition Relation

For our N-process bakery protocol we use the 5 successor symbols: 1, 2, t, w, and u. Thus, we consider the WS5S logic. For instance, $x11221u$ and $11w222$ are terms of WS5S. A process is represented as a term of the form $1^n s 2^m$, where: (i) 1^n and 2^m are (possibly empty) sequences of 1's and 2's, respectively, and (ii) $s \in \{t, w, u\}$. For a process $1^n s 2^m$ we have that: (i) the prefix 1^n is its counter (the empty string ε represents the counter 0), (ii) the symbol s is its control state, and (iii) the suffix 2^m is used to allow different processes with the same <counter, control state> pair. A configuration is a set of processes and it is represented as a set of terms each of which is of the form $1^n s 2^m$.

Now we show that the counter function *cn,* the control state function *cs,* the initial configurations, the transition relation, and the *unsafe* property are all definable in WS5S. We start off by defining the following predicates by means of WS5S formulas (between parentheses we have written the corresponding intended meanings):

- $\mathit{is\text{-}cnt}(x)$ iff $\exists X\ ((\forall y\ y \in X \rightarrow (y{=}\varepsilon \vee \exists z\ (y{=}z1 \wedge z{\in}X))) \wedge x{\in}X)$
  (x is a term of the form 1^n for some $n{\geq}0$, i.e., x is a counter)
- $\mathit{is\text{-}cs}(x)$ iff $x{=}t \vee x{=}w \vee x{=}u$
  ($x{\in}\{t, w, u\}$, i.e., x is a control state)
- $\mathit{is\text{-}proc}(x)$ iff $\exists X\ ((\forall y\ y \in X \rightarrow (\exists c \exists s\ y = cs \wedge \mathit{is\text{-}cnt}(c) \wedge \mathit{is\text{-}cs}(s)) \vee \exists z\ (y{=}z2 \wedge z{\in}X))) \wedge x{\in}X)$
  (x is a term of the form $1^n s 2^m$ for some $n, m{\geq}0$ and $s{\in}\{t, w, u\}$, i.e., x is a process)
- $\mathit{cnfg}(X)$ iff $\forall x\ (x{\in}X \rightarrow \mathit{is\text{-}proc}(x))$
  (X is a set of processes, i.e., X is a configuration)

The counter function *cn* and the control state function *cs* can be expressed by the following binary predicates:

- $\mathit{cn}(x,c)$ iff $\mathit{is\text{-}proc}(x) \wedge \mathit{is\text{-}cnt}(c) \wedge c{\leq}x \wedge (\forall y\ (y{\leq}x \wedge \mathit{is\text{-}cnt}(y)) \rightarrow y{\leq}c)$
  (x is a process whose counter is c)
- $\mathit{cs}(x,s)$ iff $\mathit{is\text{-}proc}(x) \wedge \mathit{is\text{-}cs}(s) \wedge y{\leq}x \wedge (\exists z\ (\mathit{is\text{-}cnt}(z) \wedge y{=}zs)$
  (x is a process whose control state is s)

Now we define some predicates which we will use in the definition of the transition relation, the initial configurations, and the unsafe configurations. We will write $x \notin X$, $x \neq y$, and $x < y$, instead of $\neg(x \in X)$, $\neg(x = y)$, and $x \leq y \wedge x \neq y$, respectively.

- $delete(X, c, s, Y)$ iff $cnfg(X) \wedge \exists x\ (x \in X \wedge x \notin Y\ \wedge cn(x, c) \wedge cs(x, s) \wedge (\forall y\ x \neq y \rightarrow (y \in X \leftrightarrow y \in Y)))$
  (configuration X is obtained by deleting from configuration Y a process with counter c and control state s)
- $add(X, c, s, Y)$ iff $delete(Y, c, s, X)$
- $max(X,m)$ iff $cnfg(X) \wedge \exists x\ (x \in X \wedge cn(x, m)) \wedge \forall y\ \forall c\ ((y \in X \wedge cn(y, c)) \rightarrow c \leq m)$
  (the maximum counter in configuration X is m)
- $min(X,n)$ iff $cnfg(X) \wedge \exists x\ (x \in X \wedge cn(x, n)) \wedge \forall y\ \forall c\ ((y \in X \wedge y \neq x \wedge cn(y, c)) \rightarrow (c = \varepsilon \vee n < c))$
  (in configuration X there exists a process x which has a counter n such that the counter of any other process in X is either 0 or greater than n)

Now we define three predicates whose disjunction specifies the transition relation between configurations. These three predicates correspond to the transition of a process from *think* to *wait*, from *wait* to *use*, and from *use* to *think*, respectively.

- $tw(X, Y)$ iff $\exists c \exists m \exists Z\ delete(X, c, t, Z) \wedge max(X, m) \wedge add(Z, m1, w, Y)$
  ($Y = (X - \{x\}) \cup \{y\}$ where x is a process in X with control state t, and y is a process with control state w and counter $m+1$ such that m is the maximum counter in X)
- $wu(X, Y)$ iff $\exists n \exists Z\ min(X, n) \wedge delete(X, n, w, Z) \wedge add(Z, n, u, Y)$
  ($Y = (X - \{x\}) \cup \{y\}$ where x is a process in X with control state w and counter n such that the counter of any other process in X is either 0 or greater than n, and y is a process with control state u and counter n)
- $ut(X, Y)$ iff $\exists c \exists Z\ delete(X, c, u, Z) \wedge add(Z, \varepsilon, t, Y)$
  ($Y = (X - \{x\}) \cup \{y\}$ where x is a process in X with control state u, and y is a process with counter 0 and control state t)

The initial configurations and the unsafe configurations are expressed by the following formulas:

$init(X)$ iff $\forall x\ x \in X \rightarrow (cn(x, \varepsilon) \wedge cs(x, t))$
$unsafe(X)$ iff $\exists c \exists d \exists Y \exists Z\ delete(X, c, u, Y) \wedge delete(Y, d, u, Z)$

3.3 Specifying CTL Properties via CLP(WSkS)

In order to realize Step 1 of our verification method, we introduce CLP(WSkS) programs and we use them to encode the satisfaction relation $\models$ as the predicate *sat*.

A CLP(WSkS) program is a conjunction of many sorted first order formulas [6]. There are three sorts: *string*, *stringset*, and *tree*, interpreted as finite strings, finite sets of strings, and finite trees, respectively.

Terms of CLP(WSkS) programs are either WSkS terms, or set variables, or ordinary terms (that is, terms constructed out of variables, constants, and function symbols distinct from those used for WSkS terms and set variables). WSkS

terms will be assigned the *string* sort, set variables will be assigned the *stringset* sort, and ordinary terms will be assigned the *tree* sort. Each n-ary predicate of CLP(WSkS) programs is assigned a unique sort which consists of an n-tuple $\langle i_1, \ldots, i_n \rangle$ of sorts. For instance, the predicate $\in$ is of sort $\langle \mathit{string}, \mathit{stringset} \rangle$. We assume that CLP(WSkS) programs are constructed by complying with the sorts of predicates and terms. An *atom* is an atomic formula whose predicate symbol is not in $\{\leq, =, \in\}$. As usual, a *literal* is either an atom or a negated atom. A CLP(WSkS) clause is of the form:

$A \leftarrow c, L_1, \ldots, L_n$

where A is an atom, c is a formula of WSkS, and $L_1, \ldots, L_n$ are literals. We can extend to CLP(WSkS) the definitions of *locally stratified* program and *perfect model*, by adapting the corresponding definitions for logic programs [1]. Given a locally stratified CLP(WSkS) program P, we denote by $M(P)$ the perfect model of P.

Now we return to our N-process system S. The following locally stratified CLP(WSkS) program P_S specifies the predicate *sat* of Step 1 of our verification method.

$\mathit{sat}(C, \mathit{unsafe}) \leftarrow \mathit{unsafe}(C)$
$\mathit{sat}(C, \neg F) \leftarrow \neg\ \mathit{sat}(C, F)$
$\mathit{sat}(C, \mathit{ef}(F)) \leftarrow \mathit{sat}(C, F)$
$\mathit{sat}(C, \mathit{ef}(F)) \leftarrow \mathit{tw}(C, Y),\ \mathit{sat}(Y, \mathit{ef}(F))$
$\mathit{sat}(C, \mathit{ef}(F)) \leftarrow \mathit{wu}(C, Y),\ \mathit{sat}(Y, \mathit{ef}(F))$
$\mathit{sat}(C, \mathit{ef}(F)) \leftarrow \mathit{ut}(C, Y),\ \mathit{sat}(Y, \mathit{ef}(F))$

We have that for all configurations c,

$$T_S, c \models \neg EF\ \mathit{unsafe} \quad \text{iff} \quad \mathit{sat}(c, \neg \mathit{ef}(\mathit{unsafe})) \in M(P_S).$$

4 Verifying the N-Process Bakery Protocol via Program Transformation

In this section we show how Step 2 of our verification method is performed. We verify the safety of the N-process bakery protocol by applying the unfold/fold transformation rules [7,8] to the constraint logic program P_S defined at the end of Section 3.3. For the reader's convenience, we recall from [8] the folding rule for CLP programs with negation. This rule is different from the one by Seki [19] and it can be shown that, under suitable restrictions, it preserves the perfect model semantics. By $FV(\varphi)$ we denote the set of free variables occurring in φ.

Constrained Atomic Folding. From a program P and a set of *definition clauses Defs* we derive a new program Q as follows. Let C be the clause $A \leftarrow c, G_1, L, G_2$ in P, where L is either the atom B or the negated atom $\neg B$, and let D be the definition clause $\mathit{newp}(v_1, \ldots, v_n) \leftarrow d, B$. Suppose that: (i) the WSkS formula $\forall w_1, \ldots, w_m (c \rightarrow d)$ holds, where $\{w_1, \ldots, w_m\} = FV(c \rightarrow d)$ and (ii) $FV(D) \cap FV(A, c, G_1, G_2) \subseteq \{v_1, \ldots, v_n\}$.
If L is B then $Q = (P - \{C\}) \cup \{A \leftarrow c, G_1, \mathit{newp}(v_1, \ldots, v_n), G_2\}$.

If L is $\neg B$ then $Q = (P - \{C\}) \cup \{A \leftarrow c, G_1, \neg newp(v_1, \ldots, v_n), G_2\}$.

We start off by introducing the following definition clause, corresponding to the clause F of Step 2 of our verification method as presented in Section 1:

d1. $bakery\text{-}safe(X) \leftarrow init(X),\ sat(X, \neg ef(unsafe))$

Our goal is to transform the program $P_S \cup \{\text{d1}\}$ into a program P_f which contains a clause of the form $bakery\text{-}safe(X) \leftarrow init(X)$. The issue concerning the automation of this transformation process will be briefly discussed in Section 5. The constraints we will use during the transformation are given by the WSkS formulas: $init(X)$, $unsafe(X)$, $tw(X,Y)$, $wu(X,Y)$, and $ut(X,Y)$.

Step 2 of the verification method is divided in the two phases 2.1 and 2.2. Now we will describe these two phases and then we will see them in action in the verification of the N-process bakery protocol.

Phase 2.1 consists in the iteration of a sequence of applications of the transformation rules as follows: (i) we unfold all definition clauses, (ii) we remove all clauses with unsatisfiable constraints, and (iii) if we are able to fold all clauses which have *sat* literals in the body, then we terminate Phase 1; otherwise we introduce some new definition clauses which will allow us to fold all such clauses, and we return to Point (i).

Phase 2.2 consists in iterating the following two transformations: (i) we delete definitions of *useless predicates*, that is, predicates for which we can infer that they do not hold in the perfect model by looking at the constrained facts and the predicate dependency graph, and (ii) we unfold clauses w.r.t. literals with useless predicates.

We start Phase 1 by unfolding clause 1, thereby obtaining

2. $bakery\text{-}safe(X) \leftarrow init(X),\ \neg\ \underline{sat(X, ef(unsafe))}$

In clause 2 and in the clauses below, we have underlined the atoms selected for folding or unfolding. The constraint $init(X)$ is satisfiable and clause 2 *cannot be folded* using the definition clause d1. Thus, we introduce the new definition clause

d3. $newp1(X) \leftarrow init(X),\ \underline{sat(X, ef(unsafe))}$

By using clause d3 we fold clause 2, and we obtain

4. $bakery\text{-}safe(X) \leftarrow init(X),\ \neg\ newp1(X)$

We proceed by applying the unfolding rule to the newly introduced clause d3, thereby obtaining

5. $newp1(X) \leftarrow init(X) \wedge unsafe(X)$
6. $newp1(X) \leftarrow init(X) \wedge tw(X,Y),\ \underline{sat(Y, ef(unsafe))}$
7. $newp1(X) \leftarrow init(X) \wedge wu(X,Y),\ sat(Y, ef(unsafe))$
8. $newp1(X) \leftarrow init(X) \wedge ut(X,Y),\ sat(Y, ef(unsafe))$

Clauses 5, 7 and 8 are removed, because their bodies contain unsatisfiable constraints. Clause 6 cannot be folded using the already introduced definition clauses d1 and d3, because the implication $\forall X \forall Y\ (init(X) \wedge tw(X,Y)) \rightarrow init(Y)$ does not hold. In order to fold clause 6, we introduce the new definition clause

d9. $newp2(X) \leftarrow c(X),\ \underline{sat(X, ef(unsafe))}$

where $c(X)$ is a new constraint defined by the following WS5S formula:

$(\forall x\ x \in X \rightarrow ((cs(x,t) \wedge cn(x,\varepsilon)) \vee (cs(x,w) \wedge \exists c(cn(x,c) \wedge \varepsilon < c))))$

We have that $\forall X\, \forall Y\ (init(X) \wedge tw(X,Y)) \rightarrow c(Y)$ holds and thus, we can fold 6 using d9. We obtain

10. $newp1(X) \leftarrow init(X) \wedge tw(X,Y),\ newp2(Y)$

By unfolding the definition clause d9 we obtain

11. $newp2(X) \leftarrow c(X) \wedge unsafe(X)$
12. $newp2(X) \leftarrow c(X) \wedge tw(X,Y),\ \underline{sat(Y, ef(unsafe))}$
13. $newp2(X) \leftarrow c(X) \wedge wu(X,Y),\ \underline{sat(Y, ef(unsafe))}$
14. $newp2(X) \leftarrow c(X) \wedge ut(X,Y),\ \underline{sat(Y, ef(unsafe))}$

Clauses 11 and 14 have unsatisfiable constraints in their bodies and we remove them. We fold clause 12 by using the already introduced definition clause d9 and we obtain

15. $newp2(X) \leftarrow c(X) \wedge tw(X,Y),\ newp2(Y)$

However, clause 13 cannot be folded by using a definition clause introduced so far. Thus, in order to fold it we introduce the following new definition clause

d16. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X),\ \underline{sat(X, ef(unsafe))}$

where the constraint $d(X)$ is the WSkS formula:

$(\exists y\ y \in X \wedge cs(y,u)) \wedge$
$(\forall x\ x \in X \rightarrow ((cs(x,t) \wedge cn(x,\varepsilon)) \vee$
$\qquad (cs(x,w) \wedge \exists c\ (cn(x,c) \wedge \varepsilon < c\ )) \vee$
$\qquad (cs(x,u) \wedge \exists n\ (cn(x,n) \wedge min(X,n)))))$

We have that: $\forall X\, \forall Y\ (c(X) \wedge wu(X,Y)) \rightarrow (d(Y) \wedge \neg unsafe(Y))$ and we can fold clause 13 using clause d16. We get

17. $newp2(X) \leftarrow c(X) \wedge wu(X,Y),\ \ newp3(Y)$

We now proceed by applying the unfolding rule to the definition clause d16 and we obtain

18. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X) \wedge unsafe(X)$
19. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X) \wedge tw(X,Y),\ \underline{sat(Y, ef(unsafe))}$
20. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X) \wedge wu(X,Y),\ \underline{sat(Y, ef(unsafe))}$
21. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X) \wedge ut(X,Y),\ \underline{sat(Y, ef(unsafe))}$

We remove clauses 18 and 20 which have unsatisfiable constraints in their bodies, and we fold clauses 19 and 21 by using the definition clauses d16 and d9, respectively. Indeed the following two implications hold:

$\forall X\, \forall Y\ (d(X) \wedge \neg unsafe(X) \wedge tw(X,Y)) \rightarrow (d(Y) \wedge \neg unsafe(Y))$
$\forall X\, \forall Y\ (d(X) \wedge \neg unsafe(X) \wedge ut(X,Y)) \rightarrow c(Y)$

We obtain

22. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X) \wedge tw(X,Y),\ newp3(Y)$
23. $newp3(X) \leftarrow d(X) \wedge \neg\ unsafe(X) \wedge ut(X,Y),\ newp2(Y)$

Since no new definition clauses need be introduced, we terminate Phase 1 of our transformation process. The program derived so far consists of the clauses 4, 10, 15, 17, 22 and 23.

We now proceed by performing Phase 2 of our transformation. Predicates *newp*1, *newp*2, and *newp*3 are useless because their definitions do not contain any constrained fact. Therefore we may remove clauses 10, 15, 17, 22, and 23 and we derive a program consisting of clause 4 only. We now unfold clause 4 w.r.t. the literal $\neg$ *newp*1(X), whose predicate is useless. We obtain

24. *bakery-safe*$(X) \leftarrow$ *init*(X)

and, according to Step 2 of our verification method, this concludes the proof of the safety property for the N-process bakery protocol.

5 Related Work and Conclusions

Model checking is a well established technique for the verification of *finite state concurrent systems* [4]. Recently, there have been several attempts to develop verification methods for *parameterized systems*, that is, systems consisting of an arbitrary number of finite state processes. Among them the method described in [18] is closely related to ours, in that it uses unfold/fold transformation of programs for generating induction proofs of safety properties for parameterized systems. However, there are some differences between the present paper and [18]. We use constraint logic programs with locally stratified negation to specify concurrent systems and their properties while [18] uses definite logic programs. Correspondingly, we use a different set of transformation rules. Moreover, the class of parameterized systems is strictly smaller than the class of systems we consider here. Indeed, we consider systems with an arbitrary number of *infinite* state processes.

Other methods for the verification of concurrent systems which have been recently proposed in the literature, are based on (constraint) logic programming. In particular:
(i) The method described in [12] uses partial deduction and abstract interpretation of logic programs for verifying safety properties of infinite state systems. (ii) The method presented in [9] uses logic programs with linear arithmetic constraints and Presburger arithmetic to verify safety properties of Petri nets. (iii) Similarly to [9], also the method presented in [5] uses constraint logic programs to represent infinite state systems. This method can be applied to verify CTL properties of such systems by computing approximations of least and greatest fixed points via abstract interpretation. (iv) In [17] it is shown that by using tabulation-based logic programming systems it is possible to efficiently verify μ-calculus properties of finite state transitions systems expressed in a CCS-like language. (v) The method in [14] uses CLP with finite domains, extended with constructive negation and tabled resolution, for finite state local model checking.

With respect to the abovementioned methods, the distinctive feature of the method presented in this paper is that we use the WSkS logic, together with CLP and program transformation, to deal with systems consisting of an arbitrary number of infinite state processes.

Proofs of properties for this larger class have been presented, among others, in the following papers.

In [13] the authors introduce a proof technique which is based on induction and model checking. Proofs are carried out by solving a finite number of model checking problems on a finite abstraction of the initial system and they are mechanically checked. The technique is illustrated by proving that the N-process bakery protocol is starvation free.

In [16] the author presents a proof of the mutual exclusion for the N-process version of the ticket protocol based on the Owicki-Gries assertional method. The proof has been mechanically checked by using the Isabelle theorem prover.

In [20] the author presents a proof of the mutual exclusion for the N-process bakery protocol. The proof is carried out by using a combination of theorem proving and model checking based on an abstraction of the protocol itself which reduces it to the case of two processes only.

Similarly to the techniques presented in the above three papers [13,16,20], each step of our verification method can be mechanized, but the construction of the whole proof requires some human guidance. However, in contrast to [13,16,20] in our approach the parameter N representing the number of processes is *invisible*, no explicit induction on N is performed, and no abstraction of the set of processes is needed.

The verification method presented in this paper is an enhancement of the *rules + strategies* transformation method proposed in [8] for verifying CTL properties of systems consisting of a fixed number of infinite state processes. In [8] we proved the mutual exclusion property for the 2-process bakery protocol by using CLP programs with constraints consisting of linear inequations over the reals. That proof can easily be extended to the case of any fixed number of processes by using CLP programs over the same constraint theory. Here, however, we proved the mutual exclusion property for the N-process bakery protocol, *uniformly* for any N, by using CLP programs with constraints over a weak monadic second order theory with k successors.

The verification method described in the present paper can be automated following the approach presented in [8], but due to the lack of an implementation of a solver for the WSkS theory, the proof of the mutual exclusion property for the N-process bakery protocol, presented in Section 4 was made by hand.

In the future we plan to integrate our CLP program transformation system MAP with a solver for constraints over WSkS. This integrated system would allow us to experiment with other protocols and other properties of interest.

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. 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. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

5. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th International Conference TACAS'99*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.
6. H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
7. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
8. 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.
9. 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.
10. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
11. Leslie 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. K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *CAV 2000*, Lecture Notes in Computer Science 1855, pages 312–327. Springer-Verlag, 2000.
14. Ulf Nilsson and Johan Lübcke. Constraint logic programming for local and symbolic model-checking. In J. Lloyd et al., editor, *CL 2000: Computational Logic*, number 1861 in Lecture Notes in Artificial Intelligence, pages 384–398, 2000.
15. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
16. L. Prensa-Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In *Formal Methods for Parallel Programming: Theory and Applications, FMPPTA 2001*. IEEE Computer Society Press, 2001.
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. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
20. N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000: Concurrency Theory*, number 1877 in Lecture Notes in Computer Science, pages 1–16, State College, PA, August 2000. Springer-Verlag.
21. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.
22. J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.
23. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer, 1997.