Verifying CTL Properties* of Infinite State Systems by Specializing Constraint Logic Programs

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

The goal of automated verification of systems is the definition and the implementation of logical frameworks which allow one: (i) to formally specify these systems, and (ii) to prove their properties in an automatic way. These logical frameworks require formalisms both for the description of the systems and the description of their properties.

In this paper we assume that a system makes transitions from states to states and the evolution of a system can be formalized using a *computation tree,* which is defined as follows. Given a system S and its initial state s_0, the root of the computation tree for S is s_0, and every node s_i of the computation tree for S has a child node s_j iff there exists in S a transition from state s_i to state s_j. The state s_j is called a *successor* state of s_i. We assume that in every system for every state s_i there exists at least one successor state. Notice that the set of all states of a system may be finite or infinite.

We also assume that the description of the properties of any system can be done in the *Computation Tree Logic* formalism [4] (CTL for short). CTL formulas describe properties of computation trees, and they are built using: (i) atomic state properties, (ii) logical connectives: $\neg, \wedge, \vee$, (iii) quantifiers over paths: A ('for all paths') and E ('for some path'), and (iv) quantifiers along paths: G ('for all states on the path'), F ('for some state on the path'). CTL formulas are very expressive, and in particular, one may use them to describe the so called *safety* and *liveness* properties. Given a CTL formula φ and state s, the semantics of CTL defines the satisfaction relation $s \models \varphi$ which holds whenever φ is true in s [4].

In this paper we will present a method for verifying CTL properties of possibly infinite state systems by using Constraint Logic Programming [7] (CLP, for short) and Program Specialization. Our method is applicable to a large class of concurrent systems, like those described by [14].

* Revised version of the extended abstract presented at VCL'01, 2nd ACM-Sigplan Workshop on Verification and Computational Logic, Florence, Italy, September 4, 2001.

Constraint Logic Programming extends standard logic programming by incorporating mechanisms for solving constraints over some given constraint domains by using domain specific efficient algorithms. The following are constraint domains which are usually considered: (i) the domain of inequations over real or rational numbers, (ii) the domain of boolean formulas, and (iii) every finite domain. CLP programs turn out to be very suitable for modelling infinite state systems because an infinite set of states can be described by constraints over the state space.

For our purpose of verifying CTL properties of systems which may require the use of negated formulas, we will consider CLP programs with *locally stratified negation* [1]. We assume that the semantics of a program P in this class is provided by the unique perfect model, denoted by $M(P)$, which coincides with the unique stable model, and the total well-founded model. The reader may refer to [1,7,11] for the definition of these models and other notions of logic programming or constraint logic programming which we do not recall here.

Program specialization is a program transformation technique whose goal is the adaptation of a program to the context where it is used. In [5] we have developed a general framework for the automatic specialization of constraint logic programs over a generic constraint domain $\mathcal{D}$. The specialization also improves the computational properties w.r.t. a given class of goals, while it preserves the *least $\mathcal{D}$-model* [8]. In this paper we use an extension of that framework to deal with locally stratified CLP programs.

Our specialization technique is *correct* w.r.t. the perfect model semantics in the sense that, given a locally stratified CLP program P and an atom A whose predicate is defined in P, and given a program Q which is a specialization of P w.r.t. A, for every ground instance A_g of A,

$$A_g \in M(P) \quad \text{iff} \quad A_g \in M(Q). \qquad (1)$$

Verification Method. Our method for verifying whether or not a system S in its initial state s_0, satisfies a CTL property φ, consists of the following two steps.

Step 1. We introduce a CLP program P_S which defines a binary predicate *sat* such that $s_0 \models \varphi$ iff $sat(s_0, \varphi) \in M(P_S)$. We assume that s_0 and φ are ground terms.

Step 2. We introduce a new 0-ary predicate f defined by the clause $f \leftarrow sat(s_0, \varphi)$ and thus, $sat(s_0, \varphi) \in M(P_S)$ iff $f \in M(P_S \cup \{f \leftarrow sat(s_0, \varphi)\})$. We then apply our program specialization technique and transform the program $P_S \cup \{f \leftarrow sat(s_0, \varphi)\}$ into a specialized program P_f. By the correctness of program specialization, stated by the equivalence (1) above, we have that $f \in M(P_S \cup \{f \leftarrow sat(s_0, \varphi)\})$ iff $f \in M(P_f)$.

Putting Step 1 and Step 2 together, we have that $s_0 \models \varphi$ iff $f \in M(P_f)$, and we can check whether or not $s_0 \models \varphi$ as follows: (i) if the unit clause $f \leftarrow$ occurs in P_f then $s_0 \models \varphi$, and (ii) if no clause with head f occurs in P_f (that is, f has an empty definition in P_f) then it is not the case that $s_0 \models \varphi$.

The structure of our paper is as follows. In Section 2 we show how Step 1 of our verification method can be realized by encoding CTL properties of a finite or infinite state system as a constraint logic program with locally stratified negation. In Section 3 we describe the program specialization technique which we use to realize Step 2 of our verification method. In Section 4 we show how our method works for the verification of a safety and a liveness property of the bakery protocol for mutual exclusion [9]. Finally, in Section 5 we compare our work with related verification techniques described in the literature.

2 Encoding CTL Properties as Constraint Logic Programs

Given a system S, its initial state s_0, and a CTL property φ, Step 1 of our verification method is realized by providing the recursive definition of the relation $s_0 \models \varphi$ as a locally stratified program P_S. Step 1 can be performed in an automatic way for a very large class of concurrent systems, namely those which are state transition systems with *enabling conditions* and *actions* [14] with conditions and actions which can be expressed by constraints over the values of *state variables*.

The following simple example will clarify the reader's ideas. This example will be used throughout this paper to illustrate our approach. Let $S0$ be the system whose set of states is a subset of $\{a, b\} \times \mathbf{Z}$, where $\mathbf{Z}$ is the set of integers. Let the initial state of $S0$ be the pair $\langle a, 0\rangle$, and let us assume that the transitions between states are the following ones:

$$\begin{array}{ll} & \langle a, 1\rangle \rightarrow \langle b, 1\rangle \\ \forall y \in \mathbf{Z}. & \langle a, y\rangle \rightarrow \langle a, y+2\rangle \\ \forall y \in \mathbf{Z}. & \langle b, y\rangle \rightarrow \langle b, y-1\rangle \end{array}$$

The system $S0$ can be depicted as shown in Figure 1.

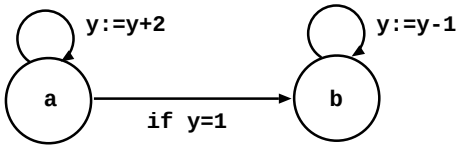

Figure 1. The system $S0$ whose set of states is $\{a, b\} \times \mathbf{Z}$.

We want to verify that the property that starting from the initial state $\langle a, 0\rangle$, the system $S0$ never reaches a state $\langle x, y\rangle$ with $y < 0$, for some $x \in \{a, b\}$. This property is expressed by the satisfaction relation $\langle a, 0\rangle \models \neg EF\ neg$ which asserts that in the initial state $\langle a, 0\rangle$ the CTL formula $\neg EF\ neg$ is true, where *neg* is the atomic state property which holds in a state $\langle x, y\rangle$ iff $y < 0$.

The satisfaction relation $s \models \varphi$ between the state s and the CTL formula φ, is recursively defined as follows:

$s \models p$ iff p is an atomic state property and p holds in s
$s \models \neg\varphi$ iff it is not the case that $s \models \varphi$

$s \models EF\,\varphi$ iff either $s \models \varphi$
or $\exists$ a state s_1 such that (i) $\exists$ a transition from s to s_1 and
(ii) $s_1 \models EF\,\varphi$

For the system $S0$ the relation $s \models \varphi$ can be encoded by the following locally stratified program P_{S0} written a Prolog-like syntax, where the state $\langle x, y\rangle$ is denoted by `s(X,Y)`:

```
sat(s(X,Y),neg) :- Y<0.
sat(S,not(F)) :- \+ sat(S,F).
sat(S,ef(F)) :- sat(S,F).
sat(s(a,Y),ef(F)) :- Y1=Y+2, sat(s(a,Y1),ef(F)).
sat(s(a,Y),ef(F)) :- Y=1, sat(s(b,Y),ef(F)).
sat(s(b,Y),ef(F)) :- Y1=Y-1, sat(s(b,Y1),ef(F)).
```

Thus, we have that:

$\langle a, 0\rangle \models \neg EF\ neg$ iff `sat(s(a,0),not(ef(neg)))` $\in M(P_{S0})$.

3 Verification of CTL Properties via Program Specialization

Step 2 of our verification method is realized by using an automatic program specialization technique which is derived from the one described in [5] and it is a particular case of the *program transformation* technique based on *rules* and *strategies*. The transformation rules ensure the correctness of the specialized program w.r.t. the given initial program. Thus, given a system S with initial state s_0, and a CTL property φ, after Step 1 and Step 2, whereby we introduce the predicate f, and we derive the specialized program P_f, we have that:

$$s_0 \models \varphi \quad \text{iff} \quad f \in M(P_f).$$

The specialization strategy guides the application of the rules with the aim of deriving a program P_f where the definition of f is either (i) the unit clause $f \leftarrow$ or (ii) it is the empty definition. As already mentioned, in case (i) $s_0 \models \varphi$ holds, while in case (ii) $s_0 \models \varphi$ does not hold.

We will show that our specialization strategy terminates for all initial programs $P_S \cup \{f \leftarrow sat(s_0, \varphi)\}$, but due to the undecidability of CTL for infinite state systems, in some cases our specialization strategy may produce a program P_f in which the definition of f is neither the unit clause $f \leftarrow$ nor the empty definition.

However, we have that our strategy is complete for finite state systems.

3.1 The Rules for Program Specialization

Now we introduce some notions which we use below for describing our program specialization technique.

We assume that every atom is *pure*, that is, it is of the form $p(X_1, \ldots, X_m)$, where $X_1, \ldots, X_m$ are distinct variables. A *constrained atom* is the conjunction of a constraint and an atom. *Goals* are (possibly empty) conjunctions of atoms.

A *constrained goal* is the conjunction of a constraint and a goal. Conjunction is commutative and, thus, the order of constraints and atoms in the body of a clause is immaterial. The empty conjunction (of constraints or atoms) is *true*. *Clauses* are of the form $H \leftarrow c, G$.

The assumption that all atoms are pure is not restrictive because, for instance, any clause γ with occurrences of non-pure atoms, can be transformed into a clause δ with occurrences of pure atoms only, such that δ is equivalent to γ w.r.t. the least $\mathcal{D}$-model semantics. This can be done by adding suitable equality constraints. For example, the clause: $p(X+1) \leftarrow X \leq 0$, $r(X-1)$ can be transformed into the equivalent clause: $p(Y) \leftarrow X \leq 0$, $r(Z)$, $Y = X+1$, $Z = X-1$, where all atoms are pure.

We define the set of *useless predicates* of a program P to be the maximal set U of predicates occurring in P such that the predicate p is in U iff the body of each clause defining p in P contains a positive literal whose predicate is in U.

We say that the atom A is *failed* in a program P iff A does not unify with the head of any clause in P. We say that A is *valid* in a program P iff P contains a unit clause whose head has the predicate symbol of A.

The process of *specializing* a given program P whereby deriving program P_s, can be formalized as a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where $P_0 = P$, $P_n = P_s$ and, for $k = 0, \ldots, n-1$, program P_{k+1} is obtained from program P_k, called the current program, by applying one of the transformation rules are listed below. These rules are an extension of the rules presented in [5] to the case of CLP programs with locally stratified negation.

R1. Constrained Atomic Definition. Introduce a new predicate defined by a *definition* clause: $newp(X_1, \ldots, X_n) \leftarrow c, A$ where c, A is a constrained atom.
R2. Positive Unfolding. Replace clause $H \leftarrow c, G_1, A, G_2$ where A is an atom, by the set of clauses $\{H \leftarrow c,\ A = A_j,\ c_j,\ G_1, G_j, G_2 \mid j = 1, \ldots, m\}$, where $\{A_j \leftarrow c_j, G_j \mid j = 1, \ldots, m\}$ is the set of *all* renamed apart clauses in the current program such that the atoms A and A_j have the same predicate symbol.
R3. Negative Unfolding. Let γ be the clause $H \leftarrow c, G_1, \neg A, G_2$. If A is failed in the current program then replace γ by the clause $H \leftarrow c, G_1, G_2$. If A is valid in the current program then remove γ.
R4. Constrained Atomic Folding. Replace clause $A \leftarrow c, G_1, L, G_2$, by clause $A \leftarrow c, G_1, L', G_2$, where L' is $newp(X_1, \ldots, X_n)$ if L is B, or $\neg newp(X_1, \ldots, X_n)$ if L is $\neg B$, provided that there exists a renamed apart definition clause $newp(X_1, \ldots, X_n) \leftarrow d, B$ such that $\mathcal{D} \models c \rightarrow d$.
R5. Removal of Clauses with Unsatisfiable Body. Remove clause $A \leftarrow c, G$ if the constraint c is unsatisfiable.
R6. Removal of Useless Clauses. Remove all clauses whose head predicate is useless in the current program.
R7. Removal of Subsumed Clauses. If the current program contains a unit clause $p(X_1, \ldots, X_n) \leftarrow$, then remove all clauses whose head predicate symbol is p.
R8. Contextual Constraint Replacement. Given a set $\mathbf{C}$ of constrained atoms, replace clause $p(X_1, \ldots, X_n) \leftarrow c_1, G$ by $p(X_1, \ldots, X_n) \leftarrow c_2, G$, if for

some constraint c_2, we have that for every constrained atom $c, p(Y_1, \ldots, Y_n)$ in $\mathbf{C}$, $\mathcal{D} \models (c, X_1 = Y_1, \ldots, X_n = Y_n) \rightarrow (c_1 \leftrightarrow c_2)$, that is, in the constraint domain $\mathcal{D}$ if $c, X_1 = Y_1, \ldots, X_n = Y_n$ holds then the constraints c_1 and c_2 are equivalent.

3.2 The Specialization Strategy

The specialization process is performed according to a strategy which guides the application of the rules R1-R8 above. Our specialization strategy is parameterized by: (i) a function *solve* for solving constraints over the constraint domain $\mathcal{D}$, (ii) an *unfolding function* for controlling the unfolding process, (iii) a clause *generalization function* for controlling the introduction of new predicate definitions. Once these parameters have been chosen, our strategy can be applied in a fully automatic way.

The specialization strategy is divided into three phases.

Phase A. We consider the program $P_S \cup \{f \leftarrow sat(s_0, \varphi)\}$ and we iterate the procedures *Unfold-Replace* and *Define-Fold* as we now explain. During the *Unfold-Replace* procedure we unfold the program to be specialized by using the given unfolding function, and we solve the constraints in the derived clauses by using the given function *solve*. We then apply the *Define-Fold* procedure and we fold the clauses we have derived. For folding we make use of already available definitions and, possibly, some new definitions introduced by using the clause generalization function. Phase A terminates with output program P_A when no new definitions need to be introduced for performing the folding steps.

Phase B. We consider program P_A and, by applying the contextual constraint replacement rule, from each clause defining a predicate, say p, we remove the constraints which hold before the execution of the clause. These constraints are determined by computing the least upper bound of the set of constraints which occur in the clauses containing a call of p (see [5] for details). Under suitable conditions, that we discuss below, the output of Phase B is a program P_B which, by construction, admits a finite stratification so that $P_B = S_1 \cup \ldots \cup S_n$.

Phase C. We consider the program P_B and by working bottom-up on the strata $S_1, \ldots, S_n$, we simplify the definition of every predicate p with the aim of deriving either the unit clause $p(\ldots) \leftarrow$ or the empty definition. During this phase we apply the following rules: (i) positive and negative unfolding, (ii) removal of useless and subsumed clauses, and (iii) contextual constraint replacement.
The algorithm for Phase C is as follows.

$P_f := \emptyset$
for $i := 1, \ldots, n$ **do**
  repeat
  $S' := S_i$;
  Apply to S_i, as long as possible, the rule for removing subsumed clauses;
  Apply to S_i, as long as possible, the negative unfolding rule and the positive unfolding rule w.r.t. valid and failed atoms in the current program;

Replace all clauses in S_i of the form $H \leftarrow c$ by the clause $H \leftarrow$ provided that $\mathcal{D} \models \forall(\exists Y\, c)$ where Y is the set of variables in c
which do not occur in H
until $S' = S_i$
Remove the useless clauses of S_i;
$P_f := P_f \cup S_i$;
end-for

Finally, (i) if the unit clause $f \leftarrow$ belongs to P_f, we conclude that φ is true in S, and otherwise, (ii) if there are no clauses in P_f defining f, we conclude that φ is false in S. If neither case (i) nor case (ii) occurs, we cannot conclude that φ is true in S and we cannot conclude that φ is false in S.

Now we illustrate how our specialization strategy works by applying it to program P_{S0} described in Section 2. We start by introducing a new definition clause δ

```
property :- X=0, sat(s(a,X), not(ef(neg))).
```

which encodes the property to be verified. The specialization of program P_{S0} w.r.t. the atom `property` proceeds as follows. The output of Phase A is the following program P_A:

```
property :- A=0, \+(newsat1(A)).
newsat1(A) :- A=0, newsat2(A).
newsat1(A) :- B=2, A=0, newsat3(B).
newsat3(A) :- A>1, newsat4(A).
newsat3(A) :- B=2+A, A>1, newsat3(B).
```

During Phase B we apply the contextual constraint replacement rule and we obtain the following program P_B:

(1) `property :- A=0, \+ newsat1(A).`
(2) `newsat1(A) :- newsat2(A).`
(3) `newsat1(A) :- B=2, newsat3(B).`
(4) `newsat3(A) :- newsat4(A).`
(5) `newsat3(A) :- B=2+A, newsat3(B).`

We start Phase C by computing a minimal stratification of the program P_B. We get: $P_B = S_1 \cup S_2$, where $S_1 = \{2, 3, 4, 5\}$ and $S_2 = \{1\}$. Then, we start processing the lowest stratum S_1.

Stratum S_1. Since there is no clause defining `newsat2` and `newsat4`, we apply the positive unfolding rule to clauses 2 and 4 and we remove them from S_1. We get: $S_1 = \{3, 5\}$. Predicates `newsat1` and `newsat3` are useless in S_1, so we remove their definitions and we obtain $S_1 = \emptyset$. The iteration on stratum S_1 ends with $P_f = \emptyset$.

Stratum S_2. The current program contains no clause defining `newsat1` and this allows us to apply the negative unfolding rule to clause 1 obtaining $S_2 = \{$`property :- A=0.`$\}$. We solve the constraint in the body of the only clause of S_2 w.r.t. the variables in its head, and we obtain $S_2 = \{$`property.`$\}$. The output of Phase C of our specialization strategy is $P_f = \{$`property.`$\}$ which gives us a proof that property $\neg EF\ neg$ is true in $S0$.

Let us now briefly comment on our three-phase strategy.

The output program P_A of Phase A of the specialization strategy admits a finite stratification, if we choose a clause generalization function which does not generalize the second argument of *sat* encoding a CTL formula. Indeed, by construction, each clause of P_A with a negative literal in the body, will be of the form:

$newp(\ldots) \leftarrow \ldots, \neg newq(\ldots), \ldots$

and the predicates *newp* and *newq* are defined by clauses of the form:

$newp(\ldots) \leftarrow sat(\ldots, \varphi_p)$
$newq(\ldots) \leftarrow sat(\ldots, \varphi_q)$

where: (i) φ_p and φ_q are ground terms encoding CTL formulas, and (ii) φ_q is a proper subformula of φ_p.

Since the contextual constraint replacement rule preserves stratification, also the output program P_B of Phase B admits a finite stratification.

Our specialization strategy terminates if the following facts hold: (i) the *Unfold-Replace* procedure terminates, and (ii) the set of predicate definitions introduced by the generalization function is finite. Notice that Phases B and C always terminate. In the example presented in the next section we will provide an unfolding function and a generalization function for which Phase A of our specialization strategy terminates.

4 Verification of the Bakery Protocol

The bakery protocol describes the behavior of a system consisting of two processes, A and B, which run in parallel and try to access a shared resource. The state s_A of process A is represented by a pair $\langle c_A, a\rangle$ where c_A is an element of the set $\{think, wait, use\}$ of *control states*, and a is a counter which takes as value a non negative rational number. The possible transitions for process A are the following (see also Figure 2):
(1) if the current state is $\langle think, a\rangle$, then the next state is $\langle wait, b+1\rangle$,
(2) if the current state is $\langle wait, a\rangle$ and, either $a < b$ or $b = 0$, then the next state is $\langle use, a\rangle$,
(3) if the current state is $\langle use, a\rangle$, then the next state is $\langle think, 0\rangle$.

Analogously for process B, by interchanging a and b.

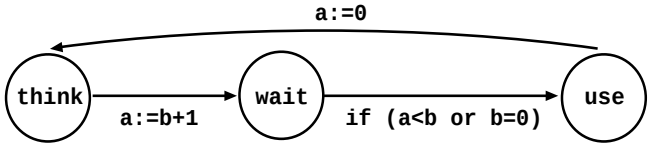

Figure 2. Transitions of the process A.

The generic state s of the system consisting of the two processes A and B together, is represented by the 4-tuple $\langle c_A, a, c_B, b\rangle$, which is initially set to $s_0 = \langle think, 0, think, 0\rangle$.

This system has an infinite number of states, because counters may increase in an unbounded way, as the following sequence of transitions illustrates:
$\langle think, 0, think, 0\rangle \rightarrow \langle wait, 1, think, 0\rangle \rightarrow \underline{\langle wait, 1, wait, 2\rangle} \rightarrow \langle use, 1, wait, 2\rangle \rightarrow \langle think, 0, wait, 2\rangle \rightarrow \langle think, 0, use, 2\rangle \rightarrow \underline{\langle wait, 3, use, 2\rangle \rightarrow} \langle wait, 3, think, 0\rangle \rightarrow \underline{\langle wait, 3, wait, 4\rangle} \rightarrow \ldots$

We have applied our specialization method to the verification of two properties of the bakery protocol: (i) the mutual exclusion property, and (ii) the starvation freedom property. The mutual exclusion property says that 'the system will never reach a state where both processes are using the shared resource'. It is a *safety* property in the sense that during the evolution of the system 'something (bad) may never happen'. The starvation freedom property says that 'if a process wants to use a resource then it will eventually get it'. It is a *liveness* property in the sense that during the evolution of a system 'something (good) eventually happens'.

The mutual exclusion property can be expressed by the CTL formula $s_0 \models \neg EF(unsafe)$, where *unsafe* is an atomic state property which holds iff both processes are in control state *use*. The starvation freedom property for a process can be expressed by the CTL formula $s_0 \models AG(wait \rightarrow AF(use))$ which is equivalent to $s_0 \models \neg EF(wait \wedge \neg AF(use))$, where *wait* and *use* are atomic state properties which hold iff the process is in control state *wait* and *use*, respectively.

Our two-step verification method works as follows.

Step 1. We introduce a CLP program P_{bakery}, which is shown in Appendix, such that for the mutual exclusion property we have: $s_0 \models \neg EF(unsafe)$ iff $\mathtt{sat(s_0, not(ef(unsafe)))} \in M(P_{bakery})$, and
for the starvation freedom property we have: $s_0 \models \neg EF(wait \wedge \neg AF(use))$ iff $\mathtt{sat(s_0, not(ef(and(wait, not(af(use))))))} \in M(P_{bakery})$.

Step 2. We choose the parameters of our strategy as follows. (i) The *solve* function is the clp(q,r) solver of [6] for simplifying conjunctions of linear equations and inequations over real numbers. (ii) The unfolding function takes a definition of the form $newp(X_1, \ldots, X_n) \leftarrow c, A$ and it unfolds it w.r.t. A. Then it unfolds the derived clauses w.r.t. the atoms of the form $sat(S, F)$, where either F is an atomic state property or the outermost connective of F is one of the following: *and*, *or*, *not*. (iii) The generalization function takes a clause $H \leftarrow c, A$ and it generalizes it to the clause $H \leftarrow d, A$, where the constraint d is defined as follows starting from the constraint c. Let $\mathcal{L}(c)$ be the set of constraints whose generic element r is defined as follows:

$r ::= true \mid t = t \mid t > t \mid t \geq t \mid r \wedge r$

where $t \in vars(c) \cup \{0\}$. The constraint d is the least constraint of $\mathcal{L}(c)$ (in the implication ordering) which is entailed by c. Notice that $vars(d) \subseteq vars(c)$.

For instance, the generalization of the clause:

```
new1(A,B) :- A=1, B=2, sat(s(wait,A,wait,B), ef(unsafe))
```

is the clause:

```
new2(A,B) :- A>0, B>A, sat(s(wait,A,wait,B), ef(unsafe)).
```

The definition of the set $\mathcal{L}(c)$ of constraints used by the clause generalization function is based upon the transitions which can be made by the system. This choice can easily be automated because this set is constructed by using the constants and the constraint predicate symbols occurring in the transitions.

We have verified the mutual exclusion and the starvation freedom properties by specializing P_{bakery} w.r.t. the atoms `safe` and `starvfree`, respectively, which are defined as follows:

```
safe :- A=0, B=0,
        sat(s(think,A,think,B), not(ef(unsafe))).
starvfree :- A=0, B=0,
        sat(s(think,A,think,B), not(ef(and(wait,not(af(use)))))).
```

The whole verification process was performed automatically by using the MAP transformation system (available at `http://www.iasi.rm.cnr.it/~fioravan`).

5 Related Work and Conclusions

During the last years many logic-based techniques have been developed for automatically verifying properties of systems, the most successful of them being model checking [2]. The success of model checking is also due to the use of Binary Decision Diagrams which provide a very compact symbolic representation of a possibly very large, but finite, set of states. In order to overcome this finiteness restriction, some efforts have recently been devoted for the incorporation into model checking of abstraction and deduction techniques [15].

Recent papers also demonstrate the usefulness of logic programming and constraint logic programming as a basis for the verification of finite or infinite state systems. In particular, in [13] the authors present XMC, a model checking system implemented in the tabulation-based logic programming language XSB. XMC can verify μ-calculus properties of finite state transitions systems expressed in a CCS-like language, with performances comparable to that of state-of-the-art model checkers. In [3] model checking using constraint logic programming is described and some CTL operators of infinite state concurrent systems are expressed in terms of least and greatest fixed points of the CLP programs used to describe the systems. In [10] the authors define a method based on partial deduction of logic programs, augmented with abstract interpretation, for solving coverability problems of infinite state Petri nets. In [12] a finite state local model checker is defined for CTL, by using CLP with finite domains, extended with constructive negation and tabled resolution.

This paper presents some preliminary results obtained when applying to the verification of infinite state systems an extension of the techniques developed in [5] for specializing constraint logic programs. We performed some experiments on a simplified version of the bakery algorithm [9] which is a protocol for mutual exclusion between two processes. We proved that this protocol ensures both mutual exclusion and starvation freedom.

We believe that the integration of CLP as modeling language, and program specialization as inference system, can provide a very flexible tool for the verification of infinite state systems. Indeed, constraints provide a natural representation for infinite sets of values (e.g., $X \geq 0$ describes the infinite set of non-negative real numbers), and the declarativeness of logic programming makes it easy to model a large variety of systems and properties.

Future work on the application of specialization of CLP program for the verification of infinite state systems will include: experimentation with different choices of constraint domains, unfolding functions, and generalization operators. We will also plan to experiment with different classes of systems and properties.

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. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
3. 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.
4. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.
5. F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In K.-K. Lau, editor, *Proceedings of LOPSTR'2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, London, UK, 24-28 July, 2000*, Lecture Notes in Computer Science 2042, pages 125–146. Springer-Verlag, 2001.
6. C. Holzbaur. OFAI clp(q,r) manual, Edition 1.3.2. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
7. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
8. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.
9. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
10. Michael Leuschel and Helko Lehmann. Solving coverability problems of petri nets by partial deduction. In *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 268–279, N.Y., September 20–23 2000. ACM Press.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
12. 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.
13. 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.
14. A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

15. Natarajan 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.

Appendix

The CLP program P_{bakery}.

```
sat(s(u,A,u,B),unsafe).
sat(s(w,A,B,C),wait).
sat(s(u,A,B,C),use).

sat(A,or(B,C)) :- sat(A,B).
sat(A,or(B,C)) :- sat(A,C).
sat(A,and(B,C)) :- sat(A,B), sat(A,C).
sat(A,not(B)) :- \+ sat(A,B).

sat(A,ef(B)) :- sat(A,B).
sat(s(t,A,S,B),ef(C)) :- D=B+1, A>=0, B>=0, sat(s(w,D,S,B),ef(C)).
sat(s(w,A,S,B),ef(C)) :- A<B, A>=0, sat(s(u,A,S,B),ef(C)).
sat(s(w,A,S,B),ef(C)) :- B=0, A>=0, sat(s(u,A,S,B),ef(C)).
sat(s(u,A,S,B),ef(C)) :- D=0, A>=0, B>=0, sat(s(t,D,S,B),ef(C)).
sat(s(S,A,t,B),ef(C)) :- D=A+1, A>=0, sat(s(S,A,w,D),ef(C)).
sat(s(S,A,w,B),ef(C)) :- B<A, B>=0, sat(s(S,A,u,B),ef(C)).
sat(s(S,A,w,B),ef(C)) :- A=0, B>=0, sat(s(S,A,u,B),ef(C)).
sat(s(S,A,u,B),ef(C)) :- D=0, B>=0, A>=0, sat(s(S,A,t,D),ef(C)).

sat(A,af(B)) :- sat(A,B).
sat(s(t,T1,t,T2),af(P)) :- T3=T2+1, T4=T1+1,
        sat(s(w,T3,t,T2),af(P)), sat(s(t,T1,w,T4),af(P)).
sat(s(t,T1,u,T2),af(P)) :- T3=T2+1, T4=0,
        sat(s(w,T3,u,T2),af(P)), sat(s(t,T1,t,T4),af(P)).
sat(s(t,T1,w,T2),af(P)) :- T3=T2+1, T2<T1,
        sat(s(w,T3,w,T2),af(P)), sat(s(t,T1,u,T2),af(P)).
sat(s(t,T1,w,T2),af(P)) :- T3=T2+1, T1=0,
        sat(s(w,T3,w,T2),af(P)), sat(s(t,T1,u,T2),af(P)).
sat(s(t,T1,w,T2),af(P)) :- T3=T2+1, T1>0, T1=<T2,
        sat(s(w,T3,w,T2),af(P)).
sat(s(u,T1,u,T2),af(P)) :- T3=0, T4=0,
        sat(s(t,T3,u,T2),af(P)), sat(s(u,T1,t,T4),af(P)).
sat(s(u,T1,w,T2),af(P)) :- T3=0, T2<T1,
        sat(s(t,T3,w,T2),af(P)), sat(s(u,T1,u,T2),af(P)).
sat(s(u,T1,w,T2),af(P)) :- T3=0, T1=0,
        sat(s(t,T3,w,T2),af(P)), sat(s(u,T1,u,T2),af(P)).
sat(s(u,T1,w,T2),af(P)) :- T3=0, T1>0, T1=<T2,
        sat(s(t,T3,w,T2),af(P)).
sat(s(w,T1,w,T2),af(P)) :- T1<T2, T1=0,
        sat(s(u,T1,w,T2),af(P)), sat(s(w,T1,u,T2),af(P)).
sat(s(w,T1,w,T2),af(P)) :- T1<T2, T1>0,
        sat(s(u,T1,w,T2),af(P)).
sat(s(w,T1,w,T2),af(P)) :- T2=0, T1=0,
        sat(s(u,T1,w,T2),af(P)), sat(s(w,T1,u,T2),af(P)).
sat(s(u,T2,t,T1),af(P)) :- T3=T2+1, T4=0,
        sat(s(u,T2,w,T3),af(P)), sat(s(t,T4,t,T1),af(P)).
```

```
sat(s(w,T2,t,T1),af(P)) :- T3=T2+1, T2<T1,
        sat(s(w,T2,w,T3),af(P)), sat(s(u,T2,t,T1),af(P)).
sat(s(w,T2,t,T1),af(P)) :- T3=T2+1, T1=0,
        sat(s(w,T2,w,T3),af(P)), sat(s(u,T2,t,T1),af(P)).
sat(s(w,T2,t,T1),af(P)) :- T3=T2+1, T1>0, T1=<T2,
        sat(s(w,T2,w,T3),af(P)).
sat(s(u,T2,u,T1),af(P)) :- T3=0, T4=0,
        sat(s(u,T2,t,T3),af(P)), sat(s(t,T4,u,T1),af(P)).
sat(s(w,T2,u,T1),af(P)) :- T3=0, T2<T1,
        sat(s(w,T2,t,T3),af(P)), sat(s(u,T2,u,T1),af(P)).
sat(s(w,T2,u,T1),af(P)) :- T3=0, T1=0,
        sat(s(w,T2,t,T3),af(P)), sat(s(u,T2,u,T1),af(P)).
sat(s(w,T2,u,T1),af(P)) :- T3=0, T1>0, T1=<T2,
        sat(s(w,T2,t,T3),af(P)).
sat(s(w,T2,w,T1),af(P)) :- T1<T2, T1=0,
        sat(s(w,T2,u,T1),af(P)),
        sat(s(u,T2,w,T1),af(P)).
sat(s(w,T2,w,T1),af(P)) :- T1<T2, T1>0,
        sat(s(w,T2,u,T1),af(P)).
sat(s(w,T2,w,T1),af(P)) :- T2=0, T1=0,
        sat(s(w,T2,u,T1),af(P)), sat(s(u,T2,w,T1),af(P)).
```