Automatic Proofs of Protocols via Program Transformation

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[3]

(1) Computer Science Department, University of L'Aquila, I-67100 L'Aquila, Italy
fioravanti@di.univaq.it
(2) DISP, University of Roma Tor Vergata, I-00133 Roma, Italy
pettorossi@info.uniroma2.it
(3) 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 protocols which regulate the activities of multiagent systems. The set of states of those systems may be infinite so that, in general, the verification of a property of a multiagent system cannot be performed by an exhaustive inspection. We specify a given multiagent system by means of a constraint logic program P with locally stratified negation, and we specify a given temporal property to be verified by means of an atomic formula A. In order to verify that the given temporal property holds, we transform the program P into an equivalent program T such that the fact $A \leftarrow$ belongs to T. Our transformation method consists of a set of rules and an automatic strategy that guides the application of the rules. Our method is sound for verifying properties of protocols that are expressible in the CTL logic [5]. Although our method is incomplete for proving properties of infinite state systems, it is able to verify important properties of several protocols which are used in practice.

1.1 Introduction

Many models of computation have been considered in the literature. Among these are: (i) the centralized, *sequential* model based on the von Neumann architecture, (ii) the centralized, *parallel* model based on the parallel random machine with the 'concurrent-read and exclusive-write' restriction on its registers, and (iii) the distributed, *multiagent system* model. Other models of computation proposed in the literature include those based on rewriting systems, logical deductions, quantum computations, and DNA computations. In this paper we want to consider, in particular, the multiagent system model and we want to analyze a few issues that we think are of major relevance for that model.

The multiagent system model can be understood as a *set* of agents, each of which can be viewed as a von Neumann computer. These agents interact with each other and cooperate by exchanging messages with the aim of achieving a common goal, maybe in the presence of antagonistic agents. The cooperation among agents is realized via protocols that ensure suitable global properties of the multiagent system. While computation progresses, a global knowledge of the system of agents is built up, starting from a local, maybe imprecise, knowledge of each individual agent. In the multiagent model the following concepts are important: (i) the *local computations* (also called *local rules*), which regulate the activity of each agent, (ii) the *protocols* (also called *metarules*), which establish the way agents interact with each other, (iii) the *communications*, which specify the messages that can be exchanged among agents, and (iv) the *distributivity* property, which indicates the degree of interaction among agents.

Before illustrating these concepts, let us present a simple example of multiagent system that will help the reader to fix his/her own ideas. In this example we consider the problem of computing a directed spanning tree of a given undirected, connected, finite graph $G = \langle N, E\rangle$, where N is a set of nodes and E is a set of edges (that is, ordered pairs of nodes). Since G is undirected we assume that for every edge $\langle n, m\rangle$ there exists the symmetric edge $\langle m, n\rangle$. For reasons of simplicity, we also assume that in G there is no edge of the form $\langle n, n\rangle$. This spanning tree is computed by several cooperating agents, each of which acting at a node of the given graph G. For every node $n \in N$ we define the two sets $P(n) = \{p \,|\, \langle p, n\rangle \in E\}$ and $S(n) = \{s \,|\, \langle n, s\rangle \in E\}$, which are the sets of the so called *predecessor nodes* of n and *successor nodes* of n, respectively.

The agent at node n can communicate only with the agents which are at the nodes of $P(n) \cup S(n)$, and the messages it sends are based only on the local knowledge it has. At the end of the computation, when the spanning tree has been computed, each agent knows the edges which insist on its node and belong to the computed spanning tree.

The multiagent system has to construct a directed spanning tree of the given graph G with a given node n_0 as its root. During the execution of the algorithm a node may be either *unmarked* or *marked*. Initially all nodes are unmarked. The computation performed by the agents at the nodes is done according to the following local rules $R1$–$R3$, which transform the given graph G into a directed spanning tree of G with root n_0.
(i) Rule $R1$ is applied by the agent at the root node n_0. This agent deletes all edges arriving at the root, marks the root, and sends a *mark-token* to every node in $S(n_0)$.
(ii) Rule $R2$ is applied by an agent at an unmarked, non-root node n iff there is at least one incoming mark-token from a node, say i, of $P(n)$. The agent at node n deletes all edges arriving at n from a node different from i, marks the node n, and sends a mark-token to every node of $S(n)$.
(iii) Rule $R3$ is applied by the agent at a marked node n iff all agents at the nodes in $S(n)$ have sent an *end-token* to node n. In particular, rule $R3$ is

applied at every node n such that $S(n)=\emptyset$. The agent at node n applies rule $R3$ by sending an end-token to each node in $P(n)$. (Notice that each time this rule is applied, $P(n)$ has exactly one element.) This rule $R3$ is needed for detecting the termination of the algorithm.

The algorithm terminates iff every node in $S(n_0)$ has sent an end-token to the root n_0. One can show that when this happens, the given graph G has been transformed into a directed subgraph T of the original graph G and T is a tree with root n_0.

In this spanning tree example the protocol consists of the metarules with which every agent should comply. The protocol establishes that: (i) all nodes are initially unmarked and the fixed node n_0 is the root of the spanning tree to be computed, (ii) the local rules are applied in an *atomic* way, in the sense that when one of them is applied to a node n, no rule can be applied to a node of $P(n) \cup S(n)$, and (iii) the termination of the algorithm occurs when the root n_0 has received an end-token from each of its children.

Communications among agents take place by sending messages. In the case of our spanning tree algorithm, these messages are either mark-tokens or end-tokens. In general, the messages may be any first-order or higher-order value. They may also be agents themselves [3]. In this case the agents sent as messages, once they reach destination, will operate in a concurrent way together with the agents residing at the destination nodes. In a multiagent system messages may also modify the topology of the communication channels while the computation proceeds. This phenomenon can be modeled within, for instance, the π-calculus [16].

The notion of distributivity in a multiagent system is related to the amount of global knowledge which is shared by the individual agents during the computation. We say that distributivity is high if every agent in the multiagent system knows a small amount of global knowledge, and it is low if this amount is big. We will not give a formal definition of distributivity: for our purposes here it will be enough to say that the spanning tree algorithm that we have described above, has high distributivity, while the usual, centralized algorithm which performs a depth-first visit of the given graph and keeps a global representation of it, has low distributivity.

The main objective of this paper is to present a technique for proving correctness of protocols. Before entering into this matter, we would like to stress the relevance of the correctness of protocols for multiagent systems by considering the familiar example of the n-queens problem for $n = 4$. We assume that each queen is a distinct agent. A position of a queen in the 4×4 board is denoted by a pair of numbers in $\{1, 2, 3, 4\} \times \{1, 2, 3, 4\}$. For instance, a queen in position $\langle 1, 2\rangle$ is placed in the first row and in the second column.

Let us consider the initial configuration whose positions are: $\langle 1, 1\rangle$, $\langle 2, 2\rangle$, $\langle 3, 3\rangle$, and $\langle 4, 4\rangle$ (that is, the queens are in the diagonal of the board from bottom-left to top-right). Let us also assume the metarules (or protocol) by which: (i) *one* queen at a time may make a move, while all other queens do not move, and (ii) a queen may go only to a position which is safe. Under these metarules there is no solution to the 4-queens problem because no queen can move.

However, there is a solution to the 4-queens problem if we consider the following metarules: (i) each queen can move only along her column, and (ii) any two mutually attacking queens make their next moves, while the other queens do not move, so that after the move, at least one of the two queens is free from attacks by any other queen. (We assume that each queen knows whether or not she is free from attacks by any other queen, and in this sense she has a global knowledge of the board.)

This protocol is correct, in the sense that there exists a sequence of board configurations which leads to a final configuration where no queen is under attack. One such a sequence is the following:

$$\begin{array}{rlcl} & \langle 1,1\rangle,\ \langle 2,2\rangle,\ \langle 3,3\rangle,\ \langle 4,4\rangle & \rightarrow & \{\text{moves of queens 1 and 2}\} \rightarrow \\ \rightarrow & \langle 1,2\rangle,\ \langle 2,4\rangle,\ \langle 3,3\rangle,\ \langle 4,4\rangle & \rightarrow & \{\text{moves of queens 3 and 4}\} \rightarrow \\ \rightarrow & \langle 1,2\rangle,\ \langle 2,4\rangle,\ \langle 3,1\rangle,\ \langle 4,3\rangle. & & \end{array}$$

Notice that the protocol we have presented, leaves unspecified the order in which the pairs of mutually attacking queens should be considered.

1.2 Specifying Protocols Using Logic Programs

In this section we present a method for specifying protocols and their properties by means of logic programs.

The behaviour of a multiagent system that evolves over time according to a given protocol, can be modeled by means of a *state transition system.* This modeling approach has been used in *model checking* techniques for the formal verification of concurrent systems [5]. A *state* is a configuration of a system which is identified by an assignment of values to the system variables. A *transition* between states models an action performed by an agent.

Formally, a state transition system is given by: (i) a set S of *states*, (ii) an *initial state* $s_0 \in S$, and (iii) a *transition relation* $t \subseteq S \times S$. We assume that t is a *total* relation, that is, for every state $s_1 \in S$ there exists a state $s_2 \in S$, called *successor state* of s_1, such that $t(s_1, s_2)$ holds. A *computation path* starting from a state s_1 is an *infinite* sequence of states $s_0\, s_1 \ldots$ such that, for every $i \geq 0$, $t(s_i, s_{i+1})$ holds. In this section and in the next one, we deal with *finite state systems*, that is, we assume that the system variables range over finite sets of values and, thus, the set of states is finite.

The transition relation t can be defined by clauses of a logic program. For instance, the finite state system depicted in Figure 1.1 can be specified by the relation t defined by the following four *unit* clauses:

$$t(s_0, s_1) \leftarrow \qquad t(s_1, s_0) \leftarrow \qquad t(s_1, s_2) \leftarrow \qquad t(s_2, s_2) \leftarrow$$

The properties of the evolution over time of a state transition system are specified by using a temporal logic called *Computation Tree Logic* (or *CTL*, for short [5]). We suppose that, for each state $s \in S$, we are given a set of

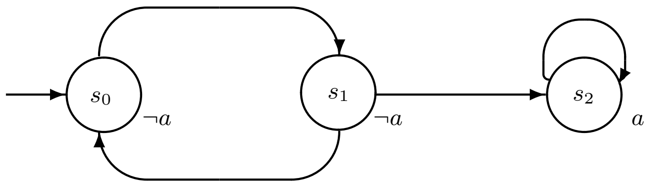

Figure 1.1. A finite state transition system. The formula φ placed near state s_i indicates that φ holds in s_i. s_0 is the initial state.

elementary properties that hold in s. These elementary properties are specified by a binary relation *elem* such that $elem(s,p)$ holds iff the elementary property p holds in s. The relation *elem* can be defined by a logic program. For instance, if we assume that no elementary property holds in s_0 and s_1, and the elementary property a holds in s_2 (see Figure 1.1), then the relation *elem* is defined by the following unit clause:

$elem(s_2, a) \leftarrow$

The formulas of CTL are built from the given set of elementary properties by using: (i) the connectives: $\neg$ ('not') and $\wedge$ ('and'), (ii) the following quantifiers along a computation path: g ('for all states on the path' or 'globally'), f ('there exists a state in the path' or 'in the future'), x ('next time'), and u ('until'), and (iii) the quantifiers over computation paths: a ('for all paths') and e ('there exists a path'). In this paper we will consider only the subset of the formulas of CTL that can be constructed by using the connectives and the two combinations *ef* and *af* of quantifiers. Thus, we assume that the syntax of a temporal formula φ is as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid ef\,\varphi \mid af\,\varphi$$

where p is an elementary property. We also use the following abbreviations: (i) $eg\,\varphi \equiv \neg af\,\neg\varphi$ and (ii) $ag\,\varphi \equiv \neg ef\,\neg\varphi$.

The semantics of temporal formulas is given by: (i) a *Kripke structure* $\mathcal{K}$ [5], which is a state transition system together with a relation *elem* specifying a set of elementary properties for each state s of $\mathcal{K}$, and (ii) a satisfaction relation $\mathcal{K}, s \models \varphi$ denoting that a formula φ holds in a state s of $\mathcal{K}$. The relation $\mathcal{K}, s \models \varphi$ is inductively defined as follows:

$\mathcal{K}, s \models p$ iff p is an elementary property such that $elem(s,p)$ holds
$\mathcal{K}, s \models \neg\varphi$ iff it is not the case that $\mathcal{K}, s \models \varphi$
$\mathcal{K}, s \models \varphi_1 \wedge \varphi_2$ iff $\mathcal{K}, s \models \varphi_1$ and $\mathcal{K}, s \models \varphi_2$
$\mathcal{K}, s \models ef\,\varphi$ iff there exists a computation path $s_0\, s_1 \ldots$ such that (i) $s = s_0$ and (ii) for some $n \geq 0$ we have that $\mathcal{K}, s_n \models \varphi$
$\mathcal{K}, s \models af\,\varphi$ iff for all computation paths $s_0\, s_1 \ldots$ if $s = s_0$ then there exists $n \geq 0$ such that $\mathcal{K}, s_n \models \varphi$.

For instance, let $\mathcal{K}_0$ be the Kripke structure consisting of the transition system of Figure 1.1, where the elementary property a holds in state s_2 only. Then the following properties hold:

$\mathcal{K}_0, s_0 \models \mathit{ef}\ a$ $\qquad\qquad$ $\mathcal{K}_0, s_0 \models \mathit{af}\ \mathit{ef}\ a$
$\mathcal{K}_0, s_0 \models \neg \mathit{af}\ a$ $\qquad\qquad$ $\mathcal{K}_0, s_0 \models \mathit{ag}\ \mathit{ef}\ a$

In order to verify that a temporal formula φ holds in the initial state s_0 of a given Kripke structure $\mathcal{K}$, we encode the satisfaction relation $\mathcal{K}, s_0 \models \varphi$ as a predicate $sat(s_0, \varphi)$ defined by a logic program $P_\mathcal{K}$. This program can be constructed by induction on the structure of the formula φ as follows [9]:

S1. $sat(X, F) \leftarrow elem(X, F)$
S2. $sat(X, \neg F) \leftarrow \neg sat(X, F)$
S3. $sat(X, F1 \wedge F2) \leftarrow sat(X, F1) \wedge sat(X, F2)$
S4. $sat(X, \mathit{ef}\ F) \leftarrow sat(X, F)$
S5. $sat(X, \mathit{ef}\ F) \leftarrow t(X, Y) \wedge sat(Y, \mathit{ef}\ F)$
S6. $sat(X, \mathit{af}\ F) \leftarrow sat(X, F)$

together with, for each state $s \in S$, a clause of the form:

S7. $sat(s, \mathit{af}\ F) \leftarrow sat(s_1, \mathit{af}\ F) \wedge \ldots \wedge sat(s_k, \mathit{af}\ F)$

where $s_1, \ldots, s_k$ are the successor states of s. Program $P_\mathcal{K}$ also includes the clauses that define the relations *elem* and t for the given Kripke structure $\mathcal{K}$.

In program $P_\mathcal{K}$ clauses S4 and S5 express that the formula $\mathit{ef}\ \varphi$ holds in a state s iff either φ holds in s or $\mathit{ef}\ \varphi$ holds in a successor state of s. Clauses S6 and S7 express that the formula $\mathit{af}\ \varphi$ holds in a state s iff either φ holds in s or $\mathit{af}\ \varphi$ holds in all successor states of s.

Program $P_\mathcal{K}$ is a logic program with *locally stratified* negation and, thus, it has a unique *perfect model*, denoted by $M(P_\mathcal{K})$ [1]. For every Kripke structure $\mathcal{K}$, state s_0, and temporal formula φ, we have that [9]:

$\mathcal{K}, s_0 \models \varphi \quad \text{iff} \quad sat(s_0, \varphi) \in M(P_\mathcal{K})$

Unfortunately, it is often the case that we cannot use SLDNF resolution [1, 13] to check whether or not $sat(s_0, \varphi)$ belongs to $M(P_\mathcal{K})$, because SLDNF resolution may not terminate for the goal $sat(s_0, \varphi)$ for many temporal formulas φ. This is due to the presence of clauses S5 and S7 and to the fact that SLDNF resolution is not able to detect the presence of infinite loops.

Tabled resolution [4] overcomes this limitation of SLDNF resolution in the case of finite state systems by maintaining a table of predicate calls and avoiding their repeated evaluation. However, tabled resolution is no longer effective when we consider logic programs encoding *infinite state systems*, that is, when the transition relation t is infinite (see Section 1.4). Indeed, in that case, when we evaluate the goal $sat(s_0, \varphi)$ by using tabled resolution, infinitely many different predicate calls may be generated.

In the next section we will show a technique based on *program transformation* [2] for checking whether or not $sat(s_0, \varphi)$ belongs to $M(P_\mathcal{K})$. This technique is complete for programs $P_\mathcal{K}$ encoding finite state systems. Moreover, in Section 1.4 we will show that our technique is also able to verify that $sat(s_0, \varphi)$ belongs to $M(P_\mathcal{K})$ also for a large class of programs $P_\mathcal{K}$ encoding infinite state systems.

1.3 Transformation Rules and Strategies for the Verification of Protocols

In this section we describe our method for verifying temporal properties of protocols. This method is based on suitable transformation rules (see rules R1–R4 below) and on a transformation strategy, called *UFV* (short for Unfold/Fold Verification), for guiding the application of these rules. Our verification method consists of two steps as indicated below.

The Verification Method.
Step 1. Given a Kripke structure $\mathcal{K}$, an initial state s_0, and a temporal formula φ, we construct a program $P_{\mathcal{K}}$ such that $\mathcal{K}, s_0 \models \varphi$ iff $sat(s_0, \varphi) \in M(P_{\mathcal{K}})$ as indicated in Section 1.2.

Step 2. We introduce the clause δ_0: $new0 \leftarrow sat(s_0, \varphi)$, where $new0$ is a new predicate symbol. Then we apply the transformation rules R1–R4 according to the transformation strategy *UFV* and from program $P_{\mathcal{K}} \cup \{\delta_0\}$ we derive a transformed program T such that: $new0 \in M(P_{\mathcal{K}} \cup \{\delta_0\})$ iff $new0 \in M(T)$.
Finally, we inspect program T and
(i) if the unit clause $new0 \leftarrow$ occurs in T then $\mathcal{K}, s_0 \models \varphi$, and
(ii) if no clause with head $new0$ occurs in T then $\mathcal{K}, s_0 \not\models \varphi$.

The correctness of our verification method is a consequence of the following facts: (i) by construction we have that $\mathcal{K}, s_0 \models \varphi$ iff $sat(s_0, \varphi) \in M(P_{\mathcal{K}})$, (ii) since δ_0: $new0 \leftarrow sat(s_0, \varphi)$ is the only clause that defines $new0$ in $P_{\mathcal{K}} \cup \{\delta_0\}$, we have that $sat(s_0, \varphi) \in M(P_{\mathcal{K}})$ iff $new0 \in M(P_{\mathcal{K}} \cup \{\delta_0\})$, (iii) since the transformation rules, when applied according to the transformation strategy *UFV*, preserve the perfect model, $new0 \in M(P_{\mathcal{K}} \cup \{\delta_0\})$ iff $new0 \in M(T)$, and finally, (iv) by the definition of perfect model, (iv.1) if $new0 \leftarrow$ occurs in T then $new0 \in M(T)$ and (iv.2) if no clause with head $new0$ occurs in T then $new0 \notin M(T)$.

Let us return to our example of Figure 1.1 and let us suppose that we want to verify that $\mathcal{K}_0, s_0 \models \neg ef\, \neg ef\, a$ holds, that is, for every state reachable from the initial state s_0 it is possible to get to a state where a holds. Let P_0 be the program constructed at Step 1 as indicated in Section 1.3. Step 2 of our verification method consists in transforming the program $P_0 \cup \{\delta_0\}$ where δ_0 is the clause $new0 \leftarrow sat(s_0, \neg ef\, \neg ef\, a)$ into a program where $new0 \leftarrow$ occurs. We will present this transformation in Section 1.3.3.

1.3.1 The Transformation Rules

The process of transforming a given program $P_{\mathcal{K}}$ thereby deriving program T, can be formalized as a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where: (i) $P_0 = P_{\mathcal{K}}$, (ii) $P_n = T$, and (iii) for $i = 0, \ldots, n-1$, program P_{i+1} is obtained from program P_i by applying one of the transformation rules listed below. These rules are variants of the rules considered in the literature for transforming logic programs (see, in particular, [20, 21]).

The *atomic definition* rule allows us to introduce a new predicate definition by adding to program P_i a new clause whose body consists of one atom only. We can use this rule to add clause δ_0 to program $P_{\mathcal{K}}$ at the beginning of Step 2 of our verification method.

R1. Atomic Definition. We introduce a new clause, called a *definition*, of the form:

$\delta: \ newp(X_1, \ldots, X_m) \leftarrow A$

where: (i) *newp* is a predicate symbol not occurring in $P_0, \ldots, P_i$, (ii) $X_1, \ldots, X_m$ are the variables occurring in A, and (iii) the predicate of A occurs in P_0. By *atomic definition* (or *definition*, for short), we derive the new program $P_{i+1} = P_i \cup \{\delta\}$. For $i \geq 0$, Defs_i denotes the set of definitions introduced during the transformation sequence $P_0, \ldots, P_i$. In particular, $\mathit{Defs}_0 = \emptyset$.

The *unfolding* rule corresponds to a symbolic computation step. It replaces a clause γ in P_i by the set of all clauses that can be derived by applying a resolution step w.r.t. a literal L occurring in the body of γ. We have a *positive* and a *negative* unfolding rule, according to the case where L is a positive or negative literal. Notice that in the negative unfolding rule (see case R2n below) the literal L should be either valid or failed. We say that an atom A is *valid* in a program P iff there exists a unit clause $H \leftarrow$ in P such that A is an instance of H. We say that an atom A is *failed* in P iff there exists no clause $H \leftarrow G$ in P such that A is unifiable with B. The negated atom $\neg A$ is valid iff A is failed and $\neg A$ is failed iff A is valid.

R2. Unfolding. Let $\gamma: \ H \leftarrow G_1 \wedge L \wedge G_2$ be a clause in P_i and let P'_i be a variant of P_i without common variables with γ. We consider the following two cases.

(R2p: *Positive Unfolding*) Let L be a positive literal. By *unfolding* γ *w.r.t.* L we derive the set of clauses

$$\begin{aligned} \mathit{\Gamma}: \ \{(H \leftarrow G_1 \wedge G \wedge G_2)\vartheta \mid \ & \text{(i) } K \leftarrow G \text{ is a clause in } P'_i \text{ and} \\ & \text{(ii) } L \text{ and } K \text{ are unifiable with mgu } \vartheta\} \end{aligned}$$

We derive the new program $P_{i+1} = (P_i - \{\gamma\}) \cup \mathit{\Gamma}$.

(R2n: *Negative Unfolding*) Let L be a negative literal.
(i) If L is valid in P'_i, then by *unfolding* γ *w.r.t.* L we derive the clause

$\eta: \ H \leftarrow G_1 \wedge G_2$

and we derive the new program $P_{i+1} = (P_i - \{\gamma\}) \cup \{\eta\}$.
(ii) If L is failed in P'_i, then by *unfolding* γ *w.r.t.* L we derive the new program $P_{i+1} = P_i - \{\gamma\}$.

The *atomic folding* rule allows us to replace an atom A which is an instance of the body of a definition by the corresponding instance of the head of the definition.

R3. Atomic Folding. Let $\gamma: \ H \leftarrow G_1 \wedge L \wedge G_2$ be a clause in P_i and let $\delta: N \leftarrow A$ be a clause in Defs_i without common variables with γ. We consider the following two cases.

(R3p: *Positive Folding*) Let L be the atom $A\vartheta$ for some substitution ϑ. By *folding* γ *w.r.t.* A *using* δ we derive the clause

$\eta: \; H \leftarrow G_1 \wedge N\vartheta \wedge G_2$

and we derive the new program $P_{i+1} = (P_i - \{\gamma\}) \cup \{\eta\}$.

(R3n: *Negative Folding*) Let L be the negated atom $\neg A\vartheta$. By *folding* γ *w.r.t.* $\neg A$ *using* δ, we derive the clause

$\eta: \; H \leftarrow G_1 \wedge \neg N\vartheta \wedge G_2$

and we derive the new program $P_{i+1} = (P_i - \{\gamma\}) \cup \{\eta\}$.

The following *clause removal* rule may be used for removing from P_i a redundant clause γ, that is, a clause γ such that $M(P_i) = M(P_i - \{\gamma\})$. Let us first introduce the following definitions. The set of *useless predicates* in a program P is the maximal set U of predicate symbols occurring in P such that a predicate p is in U iff for every clause $p(\ldots) \leftarrow G$ in P, the body G is of the form $G_1 \wedge q(\ldots) \wedge G_2$ for some predicate q in U. A clause is *useless* iff the predicate in its head is useless.

R4. Clause Removal. Let γ be a clause in P_i. By *clause removal* we derive the new program $P_{i+1} = P_i - \{\gamma\}$ if one of the following cases occurs:

(R4s: *Removal of Subsumed Clause*) γ is a clause $H \leftarrow G$ and H is valid in $P_i - \{\gamma\}$;

(R4u: *Removal of Useless Clause*) γ is useless in P_i.

1.3.2 The Transformation Strategy

During Step 2 of our verification method the transformation rules are applied according to the following transformation strategy *UFV*.

The Transformation Strategy UFV.

Input: A program $P_{\mathcal{K}}$ and a clause $\delta_0 : new0 \leftarrow sat(s_0, \varphi)$. They encode the Kripke structure $\mathcal{K}$ and the property φ to be verified in the initial state s_0.
Output: A transformed program T such that $new0 \in M(P_{\mathcal{K}} \cup \{\delta_0\})$ iff $new0 \in M(T)$.

```
Phase A. $P_A := \emptyset$;   $Defs := \{\delta_0\}$;   $\Delta := \{\delta_0\}$;
WHILE there exists a clause $\delta \in \Delta$
DO $Unfold(\delta, \Gamma)$;
   $Define\&Fold(\Gamma, Defs, NewDefs, \Phi)$;
   $P_A := P_A \cup \Phi$;   $Defs := Defs \cup NewDefs$;   $\Delta := (\Delta - \{\delta\}) \cup NewDefs$
END-WHILE
Phase B. $Remove\&Unfold(P_A, T)$
```

The *UFV* strategy is divided into two phases: Phase A and Phase B. Phase A takes program $P_{\mathcal{K}}$ and clause δ_0 as input and returns program P_A as output. Initially program P_A is the empty set of clauses. During Phase A we use the following two sets of clauses: (1) *Defs*, which is the set of definitions introduced during the transformation process, and (2) Δ, which is the set of definitions that have been introduced but not yet unfolded. At the beginning of Phase A we apply the definition rule and we add clause δ_0 to *Defs* and Δ. Then the strategy performs a WHILE-DO loop whose body consists of applications of the unfolding rule according to the *Unfold* procedure (see below), followed by applications of the definition and folding rules according to the *Define&Fold* procedure (see below).

The *Unfold* procedure takes a definition clause $\delta \in \Delta$ and derives a set Γ of clauses by applying one or more times the positive unfolding rule starting from clause δ. Every clause in Γ is of the form $H \leftarrow L_1 \wedge \ldots \wedge L_n$, where $n \geq 0$ and each literal L_i is either an atom of the form $sat(s, \psi)$ or a negated atom of the form $\neg sat(s, \psi)$. The *Define&Fold* procedure takes as input: (i) the set Γ of clauses and (ii) the set *Defs* of definitions, and returns as output: (i) a set *NewDefs* of new definitions and (ii) a set Φ of transformed clauses. *NewDefs* is the set of definitions of the form $newp \leftarrow sat(s, \psi)$ such that (i.1) a literal of the form either $sat(s, \psi)$ or $\neg sat(s, \psi)$ occurs in the body of a clause in Γ, and (i.2) there is no definition of the form $newq \leftarrow sat(s, \psi)$ in *Defs*. The set Φ consists of: (ii.1) all unit clauses belonging to Γ, together with (ii.2) all clauses derived by (positive or negative) folding of each clause in Γ w.r.t. each *sat* literal in its body using a definition in $Defs \cup NewDefs$. After the execution of the *Define&Fold* procedure the clauses of Φ are added to P_A, the definitions of *NewDefs* are added to *Defs* and, in the set Δ, clause δ is replaced by the clauses of *NewDefs*.

Phase B is realized by the *Remove&Unfold* procedure, which transforms P_A by repeatedly removing useless clauses, subsumed clauses, and applying the positive and the negative unfolding rule w.r.t. valid or failed literals. Upon termination this procedure returns a program T where $new0$ is either valid or failed, that is, either (i) the unit clause $new0 \leftarrow$ occurs in T or (ii) no clause with head $new0$ occurs in T. In case (i) we have that $\mathcal{K}, s_0 \models \varphi$ and in case (ii) we have that $\mathcal{K}, s_0 \not\models \varphi$.

For a Kripke structure $\mathcal{K}$ with a finite set of states the *UFV* strategy terminates. In particular, only a finite number of definitions will be introduced during the execution of the WHILE-DO loop because only a finite number of distinct atoms of the form $sat(s, \psi)$ can be generated. Indeed s is an element of a finite set of states and ψ is a (proper or not) subformula of the given formula φ. Thus, the *UFV* strategy is a decision procedure for checking whether or not $\mathcal{K}, s_0 \models \varphi$ holds for any given finite state Kripke structure $\mathcal{K}$, initial state s_0, and temporal formula φ.

1.3.3 An Example of Application of the Verification Method

In this section we will see our transformation strategy in action for the verification of a property of the finite state system of Figure 1.1. We want to show that in that system, for every state which is reachable from the initial state s_0, it is possible to get to a state where a holds, that is, $\mathcal{K}_0, s_0 \models ag\ ef\ a$. The initial program P_0 which encodes the Kripke structure $\mathcal{K}_0$, is the following:

1. $sat(s_2, a) \leftarrow$
2. $sat(S, \neg F) \leftarrow \neg sat(S, F)$
3. $sat(S, \mathit{ef}\ F) \leftarrow sat(S, F)$
4. $sat(s_0, \mathit{ef}\ F) \leftarrow sat(s_1, \mathit{ef}\ F)$
5. $sat(s_1, \mathit{ef}\ F) \leftarrow sat(s_0, \mathit{ef}\ F)$
6. $sat(s_1, \mathit{ef}\ F) \leftarrow sat(s_2, \mathit{ef} F)$
7. $sat(s_2, \mathit{ef}\ F) \leftarrow sat(s_2, \mathit{ef} F)$

Program P_0 has been obtained by unfolding the program $P_{\mathcal{K}}$ which encodes a generic Kripke structure $\mathcal{K}$, by using the clauses which define the relations *elem* and t relative to $\mathcal{K}_0$ (see Section 1.2). We have not considered the clauses for temporal formulas of the form $F1 \wedge F2$ and $\mathit{af} F$ because they are not needed during the application of the transformation strategy.

The clause δ_0 is $new0 \leftarrow sat(s_0, \neg \mathit{ef}\ \neg \mathit{ef}\ a)$. (Recall that $\mathit{ag}\ \mathit{ef}\ \varphi$ is equivalent to $\neg \mathit{ef}\ \neg \mathit{ef}\ \varphi$.) By applying the *Unfold* procedure, we unfold clause δ_0 using clause 2 and we get:

8. $new0 \leftarrow \neg sat(s_0, \mathit{ef} \neg\ \mathit{ef}\ a)$.

Then we apply the *Define&Fold* procedure. We introduce the definition:

9. $new1 \leftarrow sat(s_0, \mathit{ef} \neg\ \mathit{ef}\ a)$

and we fold clause 8 using clause 9. We get the following clause:

10. $new0 \leftarrow \neg new1$

At the end of the *Define&Fold* procedure, we have that the set Φ of clauses and the program P_A are both equal to {clause 10}. We also have that *Defs* = $\{\delta_0, \text{clause } 9\}$ and *NewDefs* = Δ = {clause 9}. Thus, we perform again the body of the WHILE-DO loop of Phase A of the *UFV* strategy. We apply the *Unfold* procedure and we unfold clause 9. We derive the following clauses:

11. $new1 \leftarrow \neg sat(s_0, \mathit{ef}\ a)$
12. $new1 \leftarrow \neg sat(s_1, \mathit{ef}\ a)$
13. $new1 \leftarrow \neg sat(s_2, \mathit{ef}\ a)$

Thus, Γ = {clause 11, clause 12, clause 13}. We apply again the *Define&Fold* procedure. The new value of *NewDefs* is the set of the following three clauses:

14. $new2 \leftarrow sat(s_0, \mathit{ef}\ a)$
15. $new3 \leftarrow sat(s_1, \mathit{ef}\ a)$
16. $new4 \leftarrow sat(s_2, \mathit{ef}\ a)$

which are then used to fold clauses 11, 12, and 13. These folding steps generate the following three clauses which are added to P_A:

17. $new1 \leftarrow \neg new2$
18. $new1 \leftarrow \neg new3$
19. $new1 \leftarrow \neg new4$

Now we apply for each clause in *NewDefs* the *Unfold* and *Define&Fold* procedures. For instance, starting from clause 14, by positive unfolding we get:

20. $new2 \leftarrow sat(s_1, \mathit{ef}\ a)$

and then by folding this clause using clause 15 we get:

21. $new2 \leftarrow new3$

Analogously, from clauses 15 and 16 by applying the *Unfold* and *Define&Fold* procedures we get:

22. $new3 \leftarrow new2$
23. $new3 \leftarrow new4$
24. $new4 \leftarrow$
25. $new4 \leftarrow new4$

This concludes Phase A and the derived program P_A consists of the clauses 10, 17, 18, 19, 21, 22, 23, 24, and 25. During Phase B we apply (positive and negative) unfolding and subsumption to the clauses in P_A and we get the final program T which consists of the following four unit clauses:

26. $new0 \leftarrow$ 28. $new3 \leftarrow$
27. $new2 \leftarrow$ 24. $new4 \leftarrow$

Thus, since in this program there is the clause $new0 \leftarrow$, we conclude that $\mathcal{K}_0, s_0 \models ag\ ef\ a$ holds.

1.4 Verification of Infinite State Systems

Our transformational approach for the verification of properties of protocols can be extended from finite state systems to infinite state systems. In order to specify infinite state systems we extend the method described in Section 1.2 by using *constraint logic programs* [15]. These programs generalize logic programs by allowing the bodies of the clauses to contain *constraints*. Constraints are formulas that define relations over some given domains, such as real numbers, integers, and trees. For our purposes here, constraints are simply first order formulas whose predicate symbols are taken from a distinct set. These constraints can be evaluated by using *constraint solvers* that are realized by *ad hoc* algorithms.

The semantics of a constraint c is defined by the usual first order satisfaction relation $\mathcal{D} \models c$, where $\mathcal{D}$ is a fixed interpretation. The notion of perfect model can be extended from logic programs to constraint logic programs in a straightforward way [9].

A transition relation t on an infinite set of states can be specified by using constraints in the body of the clauses that define t. For instance, the clause $t(X,Y) \leftarrow X > 0 \wedge Y = X+1$ specifies a transition relation on the set of the integer numbers. We will see a more elaborate example in Section 1.4.1.

In order to encode the satisfaction relation $\mathcal{K}, s \models \varphi$ for a Kripke structure $\mathcal{K}$ with an infinite set of states, we can construct a constraint logic program $P_{\mathcal{K}}$ similarly to what we have described in Section 1.2. However, in order to encode the satisfiability of a temporal formula of the form $af\ \varphi$, the method of Section 1.2 introduces a clause for each state of $\mathcal{K}$ and this is impossible for infinite state systems. Fortunately, this problem can be overcome for a large class of infinite state systems by using constraints as indicated in [9] (see also Section 1.4.1 for an example).

In order to extend our verification method to the case of infinite state systems, we need to extend the transformation rules and the transformation strategy presented in Section 1.3 to the case of constraint logic programs. The extensions of the definition, unfolding, folding, and clause replacement rules can be found in [8, 9]. Moreover, we will use the following two transformation rules which specifically refer to constraints: (i) Rule R4f, for deleting a clause whose body contains an unsatisfiable constraint, and (ii) Rule R5, for replacing a constraint by an equivalent one.

R4f. Removal of Clauses with Unsatisfiable Body. Let γ be a clause of the form $H \leftarrow c \wedge G$ in P_i. Suppose that c is unsatisfiable, that is, $\mathcal{D} \models \neg\exists(c)$, where $\exists(c)$ is the existential closure of c. Then, we derive the new program $P_{i+1} = P_i - \{\gamma\}$.

R5. Constraint Replacement. Let $\gamma_1 : \; H \leftarrow c_1 \wedge G$ be a clause in P_i. Suppose that for some constraint c_2, we have that:

$\mathcal{D} \models \forall\,(\exists Y_1 \ldots \exists Y_k \, c_1 \leftrightarrow \exists Z_1 \ldots \exists Z_m \, c_2)$

where: (i) $Y_1, \ldots, Y_k$ are the variables occurring free in c_1 and not in $\{H, G\}$, (ii) $Z_1, \ldots, Z_m$ are the variables occurring free in c_2 and not in $\{H, G\}$, and $\forall(\varphi)$ denotes the universal closure of formula φ. Then by *constraint replacement* we derive the clause

$\gamma_2 : \;\; H \leftarrow c_2 \wedge G$

and we derive the new program $P_{i+1} = (P_i - \{\gamma_1\}) \cup \{\gamma_2\}$.

The transformation strategy *UFV* presented in Section 1.3.2 can be extended to constraint logic programs that encode infinite state systems. During the execution of this strategy, we apply the modified transformation rules for constraint logic programs. In particular, during the execution of the *Define&Fold* procedure, when applying the definition rule, we introduce new definitions of the form:

$\mathit{NewH} \leftarrow c(X) \wedge \mathit{sat}(X, \psi)$

where: (i) X is a variable ranging over states, (ii) $c(X)$ is a constraint representing a possibly infinite set of states, and (iii) ψ is a temporal formula.

The main issue that arises when dealing with infinite state systems is that the termination of the *UFV* strategy is no longer guaranteed. Indeed, for any given temporal formula ψ, an infinite number of constrained atoms of the form $c(X) \wedge \mathit{sat}(X, \psi)$ with non-equivalent constraints may be generated and, thus, an infinite number of non-equivalent definitions may be introduced.

For instance, during the verification of the mutual exclusion property of the Bakery protocol (see Section 1.4.1 below), starting from the initial definition $\delta_0 : \mathit{new}0 \leftarrow \mathit{sat}(\langle \mathit{think}, 0, \mathit{think}, 0\rangle, \neg \mathit{ef\ unsafe})$, the *UFV* strategy introduces an infinite sequence of definitions of the form:

δ_1: $\mathit{new}1 \leftarrow A2\!=\!0 \wedge B2\!=\!0 \wedge \mathit{sat}(\langle \mathit{think}, A2, \mathit{think}, B2\rangle, \mathit{ef\ unsafe})$
δ_2: $\mathit{new}2 \leftarrow A2\!=\!1 \wedge B2\!=\!0 \wedge \mathit{sat}(\langle \mathit{wait}, A2, \mathit{think}, B2\rangle, \mathit{ef\ unsafe})$
δ_3: $\mathit{new}3 \leftarrow A2\!=\!3 \wedge B2\!=\!0 \wedge \mathit{sat}(\langle \mathit{wait}, A2, \mathit{think}, B2\rangle, \mathit{ef\ unsafe})$
$\ldots$
δ_k: $\mathit{newk} \leftarrow A2\!=\!2k\!-\!3 \wedge B2\!=\!0 \wedge \mathit{sat}(\langle \mathit{wait}, A2, \mathit{think}, B2\rangle, \mathit{ef\ unsafe})$
$\ldots$ (for $k\!>\!1$)

and the *UFV* strategy does not terminate. We can often overcome this non-termination problem by introducing a *generalization operator* between clauses, and modifying the *Define&Fold* procedure used of the *UFV* strategy as we now indicate. Suppose that a constrained literal L of the form either $d(X) \wedge \mathit{sat}(X, \psi)$ or $d(X) \wedge \neg\mathit{sat}(X, \psi)$ occurs in the body of a clause γ belonging to the set Γ of the clauses derived by the *Unfold* procedure. In order to fold γ w.r.t. L we proceed as follows: (i) if γ can be folded using a definition δ belonging to the set *Defs* of all definitions introduced so far, then we fold γ using δ, otherwise (ii) we consider a clause in *Defs* of the form:

$NewH1 \leftarrow c(X) \wedge sat(X, \psi)$

and a clause of the form:

$NewH2 \leftarrow d(X) \wedge sat(X, \psi)$

and we introduce the generalized clause:

δ_{gen}: $GenH \leftarrow genc(X) \wedge sat(X, \psi)$

where $\mathcal{D} \models \forall X\, (c(X) \rightarrow genc(X))$ and $\mathcal{D} \models \forall X\, (d(X) \rightarrow genc(X))$. Then we fold γ using δ_{gen}. For example, during the verification of the Bakery protocol, from clauses δ_2 and δ_3, by using a generalization operator we introduce the new clause:

$gen(A2) \leftarrow A2 \geq 1 \wedge B2 = 0 \wedge sat(\langle wait, A2, think, B2\rangle, ef\ unsafe)$

Suitable generalization operators which ensure the termination of the *UFV* strategy for a large class of infinite state systems can be found in [9].

1.4.1 Examples of Verification of Infinite State Systems

In this section we use our verification method for proving various properties of infinite state systems. In particular, we consider the Bakery protocol [11] and we verify that it satisfies the mutual exclusion and starvation freedom properties. Then, at the end of the section, we report on some more experimental results concerning the proofs of properties of several other protocols.

Let us consider two agents A and B which want to access a shared resource in a mutual exclusive way by using the Bakery protocol. The state of agent A is represented by a pair $\langle A1, A2\rangle$, where $A1$ is an element of the set $\{think, wait, use\}$ of *control states*, and $A2$ is a counter that takes values from the set of natural numbers. Analogously, the state of agent B is represented by a pair $\langle B1, B2\rangle$. The *state* of the system consisting of the two agents A and B, whose states are $\langle A1, A2\rangle$ and $\langle B1, B2\rangle$, respectively, is represented by the 4-tuple $\langle A1, A2, B1, B2\rangle$. The transition relation t of the two agent system from an old state *OldState* to a new state *NewState*, is defined as follows:

TA. $t(OldState, NewState) \leftarrow tA(OldState, NewState)$
TB. $t(OldState, NewState) \leftarrow tB(OldState, NewState)$

where the transition relation tA for the agent A is given by the following clauses whose bodies are conjunctions of constraints (see also Figure 1.2):

A1. $tA(\langle think, A2, B1, B2\rangle, \langle wait, A21, B1, B2\rangle) \leftarrow A21 = B2+1$
A2. $tA(\langle wait, A2, B1, B2\rangle, \langle use, A2, B1, B2\rangle) \leftarrow A2 < B2$
A3. $tA(\langle wait, A2, B1, B2\rangle, \langle use, A2, B1, B2\rangle) \leftarrow B2 = 0$
A4. $tA(\langle use, A2, B1, B2\rangle, \langle think, A21, B1, B2\rangle) \leftarrow A21 = 0$

The following analogous clauses define the transition relation tB for the agent B:

B1. $tB(\langle A1, A2, \mathit{think}, B2\rangle, \langle A1, A2, \mathit{wait}, B21\rangle) \leftarrow B21 = A2+1$
B2. $tB(\langle A1, A2, \mathit{wait}, B2\rangle, \langle A1, A2, \mathit{use}, B2\rangle) \leftarrow B2 < A2$
B3. $tB(\langle A1, A2, \mathit{wait}, B2\rangle, \langle A1, A2, \mathit{use}, B2\rangle) \leftarrow A2 = 0$
B4. $tB(\langle A1, A2, \mathit{use}, B2\rangle, \langle A1, A2, \mathit{think}, B21\rangle) \leftarrow B21 = 0$

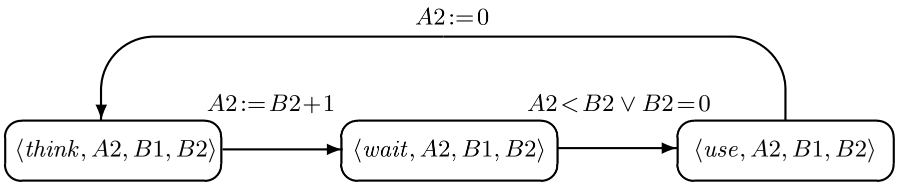
$A2 := 0$
$A2 := B2+1$
$A2 < B2 \vee B2 = 0$
$\langle \mathit{think}, A2, B1, B2\rangle$
$\langle \mathit{wait}, A2, B1, B2\rangle$
$\langle \mathit{use}, A2, B1, B2\rangle$

Figure 1.2. The Bakery protocol: a graphical representation of the transition relation tA for the agent A. The assignment $X := e$ on the arc from state s_1 to state s_2 tells us that the value of the variable X in s_2 is obtained from the value of the expression e in s_1. The boolean expression b on the arc from state s_1 to state s_2 tells us that the transition from s_1 to s_2 takes place iff b holds.

Notice that the two agent system has an infinite number of states, because counters may increase in an unbounded way, as indicated by the underlined states of the following computation path starting from the initial state $\langle \mathit{think}, 0, \mathit{think}, 0\rangle$:

$\langle \mathit{think}, 0, \mathit{think}, 0\rangle$, $\underline{\langle \mathit{wait}, 1, \mathit{think}, 0\rangle}$, $\langle \mathit{wait}, 1, \mathit{wait}, 2\rangle$, $\langle \mathit{use}, 1, \mathit{wait}, 2\rangle$,
$\langle \mathit{think}, 0, \mathit{wait}, 2\rangle$, $\langle \mathit{think}, 0, \mathit{use}, 2\rangle$, $\langle \mathit{wait}, 3, \mathit{use}, 2\rangle$, $\underline{\langle \mathit{wait}, 3, \mathit{think}, 0\rangle}$, ...

We may apply our verification method for checking the mutual exclusion property of the Bakery protocol. This property is expressed by the CTL formula: $\neg \mathit{ef}\ \mathit{unsafe}$, where for all states s,

$\mathit{elem}(s, \mathit{unsafe})$ holds iff s is of the form $\langle \mathit{use}, A2, \mathit{use}, B2\rangle$,

that is, *unsafe* holds iff both agents A and B are in the control state *use*.

The initial program $P_{\mathcal{K}}$ which encodes the Kripke structure of the Bakery protocol with two agents A and B, is the following one:

1. $\mathit{sat}(\langle \mathit{use}, A2, \mathit{use}, B2\rangle, \mathit{unsafe}) \leftarrow$
2. $\mathit{sat}(S, \neg F) \leftarrow \neg \mathit{sat}(S, F)$
3. $\mathit{sat}(S, \mathit{ef}\ F) \leftarrow \mathit{sat}(S, F)$
4. $\mathit{sat}(S, \mathit{ef}\ F) \leftarrow t(S, T) \wedge \mathit{sat}(T, \mathit{ef}\ F)$

together with the clauses TA, TB, A1–A4, and B1–B4, which define the transition relation t. The initial clause δ_0 for the mutual exclusion property is:

$\mathit{new}0_{me} \leftarrow \mathit{sat}(\langle \mathit{think}, 0, \mathit{think}, 0\rangle, \neg \mathit{ef}\ \mathit{unsafe})$

For the Bakery protocol we may also want to prove the starvation freedom property which ensures that an agent, say A, which requests the shared resource, will eventually get it. This property is expressed by the CTL formula: $\mathit{ag}\ (\mathit{waitA} \rightarrow \mathit{af}\ \mathit{useA})$, which is equivalent to: $\neg \mathit{ef}\ (\mathit{waitA} \wedge \neg \mathit{af}\ \mathit{useA})$. For the elementary properties *waitA* and *useA*, the satisfaction relation is defined by the clauses:

$sat(\langle wait, A2, B1, B2\rangle, waitA) \leftarrow$
$sat(\langle use, A2, B1, B2\rangle, useA) \leftarrow$

For the starvation freedom property the initial clause δ_0 is:

$new0_{sf} \leftarrow sat(\langle think, 0, think, 0\rangle, \neg ef\ (waitA \wedge \neg af\ useA))$

The clauses for $sat(X, \neg F)$, $sat(X, F1 \wedge F2)$, and $sat(X, ef\ F)$ are clauses S2–S5 (see Section 1.2). We do not have the space here to list all clauses for $sat(X, af\ F)$. These clauses include clause S6 (see Section 1.2) together with one or more clauses of the form S7 (see Section 1.2) for each state s of the form $\langle A1, A2, B1, B2\rangle$, where $A1$ and $B1$ belong to $\{think, wait, use\}$. For instance, the clause for the state $\langle think, A2, think, B2\rangle$ is:

$sat(\langle think, A2, think, B2\rangle, af\ F) \leftarrow A21 = B2+1 \wedge B21 = A2+1$
$\quad \wedge\ sat(\langle wait, A21, think, B2\rangle, af\ F) \wedge sat(\langle think, A2, wait, B21\rangle, af\ F)$

The remaining clauses for $sat(X, af\ F)$ can be found in [9].

By using our experimental constraint logic program transformation system MAP [14] we have been able to automatically verify the mutual exclusion and the starvation freedom properties of the Bakery protocol.

We have verified some more properties of various other protocols by using our system MAP running on a Linux machine with a 900 MHz clock, and the results of these experiments are reported in the following Table 1.1. The verification times we have obtained demonstrate that our system performs well w.r.t. the DMC system [6] and the other systems cited in [6, 7].

Protocol	Property	Verification Time
Bakery	$\neg ef\ unsafe$	0.2
(mutual exclusion)	$ag\ (waitA \rightarrow af\ useA)$	2.3
Ticket	$\neg ef\ unsafe$	0.6
(mutual exclusion)	$ag\ (waitA \rightarrow af\ useA)$	3.0
Berkeley RISC	$\neg ef\ (dirty \geq 2)$	2.0
(cache coherence)	$\neg ef\ (dirty \geq 1 \wedge shared \geq 1)$	1.3
Xerox Dragon	$\neg ef\ (dirty \geq 2)$	1.3
(cache coherence)	$\neg ef\ (dirty \geq 1 \wedge shared\_clean \geq 1)$	0.9
	$\neg ef\ (dirty \geq 1 \wedge shared\_dirty \geq 1)$	1.0
DEC Firefly	$\neg ef (dirty \geq 2)$	0.4
(cache coherence)	$\neg ef\ (dirty \geq 1 \wedge shared \geq 1)$	0.4
Illinois University	$\neg ef\ (dirty \geq 2)$	0.3
(cache coherence)	$\neg ef\ (dirty \geq 1 \wedge shared \geq 1)$	0.3
MESI	$\neg ef\ (dirty \geq 2)$	0.3
(cache coherence)	$\neg ef\ (dirty \geq 1 \wedge shared \geq 1)$	0.2

Table 1.1. Experimental results of the verification of some properties of various protocols. The protocols and the properties are taken from [6]. The verification time is expressed in seconds.

1.5 Conclusions and Related Work

We have presented a method for verifying CTL properties of protocols for multiagent systems specified as constraint logic programs. For systems which have a finite number of states, the method is complete and can be used as a decision procedure. For systems which have an infinite number of states, the method may not terminate. However, for a large class of infinite state systems, the method terminates if we use suitable generalization operators. We have applied our method for proving safety and liveness properties of several infinite state protocols.

Our verification method is related to others presented in the literature for the proofs of properties of concurrent systems which use the logic programming paradigm. Among them we mention the following ones.

In [18] the authors present XMC, a model checking system implemented in the tabulation-based logic programming language XSB. XMC can verify temporal properties expressed in the alternation-free fragment of the μ-calculus of finite state systems specified in a CCS-like language.

In [17] a model checker is presented for verifying CTL properties of finite state systems, by using logic programs with finite constraint domains that are closed under conjunction, disjunction, variable projection and negation. The verification process is performed by executing a constraint logic program encoding the semantics of CTL in an extended execution model that uses constructive negation and tabled resolution.

In [10] an automatic method for verifying safety properties of infinite state Petri nets with parametric initial markings is presented. The method constructs the reachability set of the Petri net being verified by computing the least fixpoint of a logic program with Presburger arithmetic constraints.

A method for the verification of some CTL properties of infinite state systems using constraint logic programming is described in [7]. Suitable constraint logic programs which encode the system and the property to be verified, are introduced, and then, the CTL properties are verified by computing exact and approximated least and greatest fixed points of those programs.

Finally, the use of program transformation for verifying properties of infinite state systems has been investigated in [12, 19]. In particular, (i) specialization of logic programs and abstract interpretation are used in [12] for the verification of safety properties of infinite state systems, and (ii) unfold/fold transformation rules are applied in [19] for proving safety and liveness properties of parameterized finite state systems with various network topologies.

Acknowledgements

Many thanks to Dr. A. Jankowski, Prof. A. Skowron, and Dr. M. Szczuka for their kind invitation at the International Workshop MSRAS 2004. We would like also to thank Prof. G. Delzanno, Prof. S. Etalle, and Prof. M. Leuschel for helpful discussions and comments.

References

1. K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *J. Logic Programming*, 19, 20:9–71, 1994.
2. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.
3. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
4. W. Chen and D.S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
5. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
6. G. Delzanno. Automatic verification of parametrized cache coherence protocol. In *Proc. CAV 2000*, LNCS 1855, 55–68. Springer, 2000.
7. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland (ed.) *Proc. TACAS '99*, LNCS 1579, 223–239. Springer, 1999.
8. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
9. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. R. 544, IASI-CNR, Roma, Italy, 2001.
10. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Proc. CONCUR '97*, LNCS 1243, 96–107. Springer-Verlag, 1997.
11. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, 1974.
12. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In *Proc. LOPSTR '99*, LNCS 1817, 63–82. Springer, 1999.
13. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987.
14. MAP group. The MAP transformation system. Available from: `http://www.iasi.rm.cnr.it/~proietti/system.html`, 1995–2004.
15. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.
16. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Part I and II. *Information and Computation*, 100(1):1–77, 1992.
17. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proc. CL 2000*, LNAI 1861, 384–398. Springer, 2000.
18. 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 *Proc. CAV '97*, LNCS 1254, 143–154. Springer, 1997.
19. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Verification of parameterized systems using logic program transformations. In *Proc. TACAS 2000*, LNCS 1785, 172–187. Springer, 2000.
20. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
21. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund (ed.) *Proc. 2nd Int. Conf. Logic Programming*, 127–138, Uppsala, Sweden, 1984.