Concurrent Constraint Automata

Laurent Fribourg Marcos Veloso Peixoto
email: fribourg@dmi.ens.fr veloso@dmi.ens.fr

Laboratoire d'Informatique, URA 1327 du CNRS
Département de Mathématiques et d'Informatique
Ecole Normale Supérieure
45, rue d'Ulm, 75005 Paris - France

LIENS 93-10

May 1993

Abstract

We address the problem of the specification and the proof of properties of concurrent systems which manipulate an unbounded number of data. We propose an approach based on an extended notion of automata, called "Concurrent Constraint Automata (CCA)". A CCA is an automaton with constraints and synchronous communication. By automata with constraints, we mean a state machine whose actions and states contain parameters that take their value in the set of natural numbers.
With each transition is associated an arithmetic constraint that must be satisfied by the the action and states parameters for enabling the transition. The synchronous communication is realized by means of a handshaking mechanism: two actions are executed simultaneously, their parameters being equalized. Each CCA will be represented as a logic program with arithmetic constraints. Using bottom-up evaluation techniques, we will show that, for a certain class of constraints, the *language inclusion problem is decidable*: given two CCA AUT_1 and AUT_2, it is possible to say whether or not all the sequences of actions accepted by AUT_1 are accepted by AUT_2. We show how to apply this decidability result to prove properties of concurrent systems which manipulate an unbounded number of data. Such properties cannot be proved (and not even specified) within the framework of finite-state systems, such as CCS or CSP. On the other hand, we are not able to address temporal issues, such as liveness or safety, within our framework. Our method can thus be seen as complementary to the standard methods (e.g., model checking, bisimulation) that are used for proving properties of concurrent systems.

1 Introduction

We are concerned with the problem of specifying concurrent systems and proving properties of these specifications. One approach is to use process algebras, such as CCS [13] or CSP [9]. A second approach is to use concurrent logic programming languages, such as Concurrent Prolog [19] , or more generally concurrent constraint programming languages [18]. We propose here another approach based on the notion of Concurrent Constraint Automata (CCA).

A CCA is an automaton with constraints and synchronous communication. By automata with constraints, we mean a state machine whose actions and states are parameterized. The parameters take their values on the domain of constraints, viz. arithmetic. With each transition is associated a constraint that must be satisfied by the the action and states parameters for enabling the transition. The synchronous communication is realized by means of a handshaking mechanism: two actions are executed simultaneously, their parameters being equalized. Each CCA will be represented as a logic program with arithmetic constraints. Using bottom-up evaluation techniques, we will show that, for a certain class of constraints, the *language inclusion problem is decidable*: given two CCA AUT_1 and AUT_2, it is possible to say whether or not all the sequences of actions accepted by AUT_1 are accepted by AUT_2.

CCA bear resemblance to concurrent constraint programming and to process algebras. As in concurrent constraint programming, the communication is done by exchanging information that takes its value over the domain constraint. An essential difference is that the communication is done here synchronously by parameter passing, and not asynchronously through independent update and query operations (see [1]). As process algebras, CCA are extended forms of finite-state machines. In contrast to transition systems, CCA contain parameters which are defined over an infinite domain. On the other hand the semantics that we consider here for CCA are those used for automata (language equivalence), which are stronger than those used for process algebras (observational equivalence) and cannot capture temporal notions such as safety or liveness.

The plan of the paper is as follows:

In sections 2 and 3 we present some basic definitions and explain how to represent automata by logic programs. In section 4 we define generalized automata with arithmetic constraints. In section 5 we define the operations of intersection, shuffle product and completion of generalized automata. Section 6 describes a method to solve the inclusion language problem. In section 7 we show that any generalized automaton is "equivalent" to a deterministic automaton. In section 8 we define the operation of parallel composition of automata and we give an example of application of automata operations, for specifying and proving properties of the alternating bit protocol.

2 Automata

In this section, we recall some basic notions.

Definition .1 *Let Σ be a finite set of actions and Ξ a finite set of states. An* automaton *is a quadruple $< \Xi, s_0, \Sigma, \Delta >$, where:*

1. $s_0 \in \Xi$ *is the initial state of the process*

2. Δ *is a subset of* $\Xi \times \Sigma \times \Xi$

The fact that a triple (s_i, a_k, s_j) belongs to Δ means that a transition from state s_i to state s_j can take place, executing action a_k . This transition is denoted by $s_i \stackrel{a_k}{\longmapsto} s_j$.

Our automata do not have an explicit set of final states. Any state is considered as final.

Let us assume that Σ contains r distinct actions, i.e. $\Sigma = \{a_1, \ldots, a_r\}$ and that Ξ contains $v+1$ distinct states, i.e. $\Xi = \{s_0, \ldots, s_v\}$.

Given an automaton $AUT = < \Xi, s_0, \Sigma, \Delta >$, the *language recognized* by AUT is defined by:
$Rec(AUT) = \{ [a_{k_1}, a_{k_2}, \ldots, a_{k_q}] \ / \ \exists i_1, \ldots, i_q \in \{0, \ldots, v\}$ such that
$s_0 \stackrel{a_{k_1}}{\longmapsto} s_{i_1} \stackrel{a_{k_2}}{\longmapsto} \ \ldots \ \stackrel{a_{k_q}}{\longmapsto} s_{i_q} \}$

Example 1

Let us consider the automaton AUT_0 represented in figure 1, with initial state s_0 .

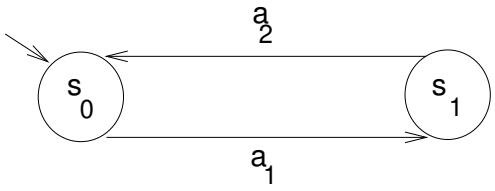

Figure 1: Automaton AUT_0

We have:

$Rec(AUT_0) = \{ [\,] \} \cup \{ [a_1] \} \cup \{ [a_1, a_2] \} \cup \{ [a_1, a_2, a_1] \} \cup \{ [a_1, a_2, a_1, a_2] \} \cup$
$\{ [a_1, a_2, a_1, a_2, a_1] \} \cup \ldots$

3 Logic Programs

Given an automaton, we want to find a logic program such that any sequence of actions recognized by the automaton corresponds to an answer for a certain goal, and vice-versa.

The construction of such a program is quite straightforward (see, e.g. [20]). Given an automaton $AUT =< \Xi, s_0, \Sigma, \Delta >$, the program Π_{AUT} is defined, using the binary predicate aut, as follows:

$aut([\,], S)$

$\left\{ aut([a_k|L], s_i) \leftarrow aut(L, s_j), \right\}$
for all i, j, k such that $(s_i, a_k, s_j) \in \Delta$

It is easy to see that given a list $\overline{l}$ of actions, the goal $\leftarrow aut(\overline{l}, s_0)$ succeeds in Π_{AUT} iff $\overline{l} \in Rec(AUT)$.

As we will see later, for the application of our proof method, it is desirable that the initial state appears in the base rule and not in the goal. Using magic sets techniques (see [2]; cf. [15]), we transform the program Π_{AUT} into a program Π'_{AUT} of the form:

$$aut'([\,], s_0)$$

$$\left\{ aut'([a_k|L], s_j) \leftarrow aut'(L, s_i), \right\} \quad \text{for all } i, j, k \text{ such that } (s_i, a_k, s_j) \in \Delta$$

The goal $\leftarrow aut'(\overline{l}, S)$ succeeds in Π'_{AUT} iff $\leftarrow aut(\overline{l}^{rev}, s_0)$ succeeds in Π_{AUT} (i.e. iff $\overline{l}^{rev} \in Rec(AUT)$). The expression $\overline{l}^{rev}$ denotes the reverse list of $\overline{l}$.

Example 2

Let us consider the automaton AUT_0 of example 1. This process can be represented by the program Π'_{AUT_0}:

$$\begin{array}{l} aut'_0([\,], s_0) \\ aut'_0([a_1|L], s_1) \leftarrow aut'_0(L, s_0) \\ aut'_0([a_2|L], s_1) \leftarrow aut'_0(L, s_0) \end{array}$$

The goal $\leftarrow aut'_0([a_2, a_1, a_2, a_1], S)$ succeeds in Π'_{AUT_0}, which means that $[a_2, a_1, a_2, a_1]^{rev}$ (i.e. $[a_1, a_2, a_1, a_2]$) is a sequence of actions recognized by AUT_0.

4 Generalized Automata with Arithmetic Constraints

In this section we generalize the notion of automata: both actions and states will contain arithmetic parameters and transitions between states will be enabled only if certain arithmetic constraints on these parameters are satisfied.

The letters M, N and P (with possible subscripts or primes) will denote vectors of arithmetic variables. The letter M will be used as an argument of actions, while N and P will be used as arguments of states. The letters $\overline{m}$, $\overline{n}$ and $\overline{p}$ will denote arithmetic values of the variables M, N and P, respectively. The letter $\overline{l}$ will denote a ground list of actions of the form $[a_{k_1}(\overline{m}_1), a_{k_2}(\overline{m}_2), \ldots, a_{k_q}(\overline{m}_q)]$. We can assume, without any loss of generality, that all state parameters have the same arity.

Definition .2 *Let Ξ be a set of (parameterized) states and let Σ be a set of (parameterized) actions. A* generalized automaton with arithmetic constraints *is a quadruple $< \Xi, s_0, \Sigma, \widehat{\Delta} >$, where:*

1. *$s_0 \in \Xi$ is the initial state of the process*
2. *$\widehat{\Delta}$ is a set of quadruples $< s_i, a_k, s_j, cond_{i,k,j} >$, where $s_i \in \Xi$, $s_j \in \Xi$, $a_k \in \Sigma$ and $cond_{i,k,j}$ is an arithmetic relation.*

The quadruple $< s_i, a_k, s_j, cond_{i,k,j} >$ will be abbreviated as $\delta_{i,k,j}$.

Informally, the fact that a quadruple $< s_i, a_k, s_j, cond_{i,k,j} >$ is in $\widehat{\Delta}$ means that a transition from state $s_i(N)$ to state $s_j(P)$ executing action $a_k(M)$ is enabled when constraint $cond_{i,k,j}(N, M, P)$ is satisfied. This transition is represented by the diagram of figure 2. The parameters M, N and P are "local" variables to the transition.

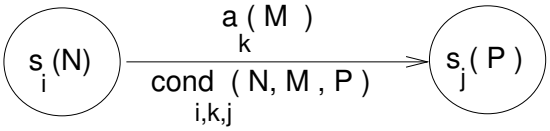

Figure 2: Transition diagram

Given a generalized automaton $AUT = < \Xi, s_0, \Sigma, \widehat{\Delta} >$, the *language recognized* by AUT is:

$Rec(AUT) = \{ [a_{k_1}(M_1), a_{k_2}(M_2), \ldots, a_{k_q}(M_q)] \ / \ \exists i_1, \ldots, i_q \in \{ 0, \ldots, v \}$ such that $\delta_{0,k_1,i_1}, \delta_{i_1,k_2,i_2}, \ldots, \delta_{i_{q-1},k_q,i_q} \in \widehat{\Delta}$ and $\exists N_0, N_1, \ldots, N_q (cond_{0,k_1,i_1}(N_0, M_1, N_1) \wedge \ldots \wedge cond_{i_{q-1},k_q,i_q}(N_{q-1}, M_q, N_q))$ holds$\}$

With the generalized automaton AUT, we associate the following logic program Π_{AUT}:

$aut([\,], s_0(N))$

$\left\{ aut([a_k(M)|L], s_j(P)) \leftarrow cond_{i,k,j}(N, M, P), aut(L, s_i(N)) \right\}$

for all i, j, k such that $\delta_{i,k,j} \in \widehat{\Delta}$

The domain of interpretation of the variables M, N and P of the program is the set of natural numbers. The expression $cond_{i,k,j}(N, M, P)$ denotes an arithmetic constraint (see [10]).

Given a list $\overline{l}$ of actions, say $[a_{k_1}(\overline{m}_1), \ldots, a_{k_q}(\overline{m}_q)]$, it can be seen that the goal $\leftarrow aut(\overline{l}, S)$ succeeds in Π_{AUT} iff the reverse list $[a_{k_q}(\overline{m}_q), \ldots, a_{k_1}(\overline{m}_1)]$ belongs to $Rec(AUT)$.

In the following, a generalized automaton AUT will be characterized by its associated logic program Π_{AUT}. We will also depict an automaton by its transition diagram. In order to account for the mechanism of "state" unification, we will represent a transitions $\delta_{i,k,j}$ followed by a transition $\delta_{i,k',j'}$ by:

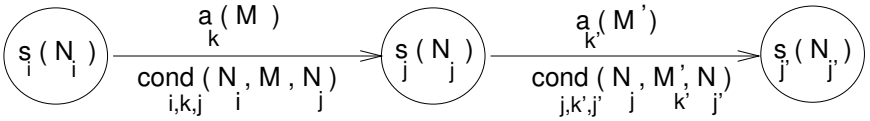

Figure 3: Composition of the transitions $\delta_{i,k,j}$ and $\delta_{i,k',j'}$

Note that an automata diagram (see, e.g., figure 4) cannot account for the mechanism of variable renaming that should take place whenever a state is reached for the second time within a path.

Example 3

Let AUT_1 be the automaton represented by the program Π_{AUT_1}:

$$
\begin{array}{l}
aut_1([\,], s_0(N)) \\
aut_1([a_1(M)|L], s_1(P)) \quad \leftarrow \quad M = N,\ aut_1(L, s_0(N)) \\
aut_1([a_2(M)|L], s_0(P)) \quad \leftarrow \quad M = P,\ aut_1(L, s_1(N))
\end{array}
$$

The recognized language is:

$$
\begin{array}{l}
Rec(AUT_1) \ = \ \{\,[\,]\,\} \ \cup \ \{\ [a_1(M)]\ \} \ \cup \ \{\ [a_1(M), a_2(M')]\ \} \ \cup \\
\qquad\qquad \{\ [a_1(M), a_2(M'), a_1(M')]\ \} \ \cup \ \{\ [a_1(M), a_2(M'), a_1(M'), a_2(M'')]\ \} \ \cup \ \ldots
\end{array}
$$

This automaton depicted on figure 4. (Note that N_0, N_1, M_1 and M_2 are "local" variables to each transition.)

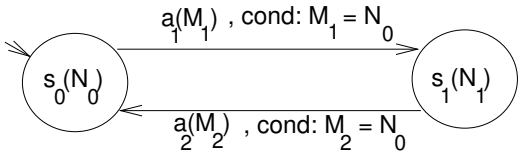

Figure 4: Automaton AUT_1

The goal $\leftarrow aut_1([a_2(5), a_1(7), a_2(7), a_2(3)]$, for example, succeeds in Π_{AUT_1}, which means that $[a_2(3), a_2(7), a_1(7), a_2(5)] \in Rec(AUT_1)$.

5 Automata Basic Operations

In this section, we extend the classical operations of intersection and shuffle product (see, e.g., [5]) to generalized automata. We also define an operation of completion of automata.

Let AUT and AUT' be two generalized automata defined respectively by $< \Xi, s_0, \Sigma, \widehat{\Delta} >$ and $< \Xi', s'_0, \Sigma', \widehat{\Delta}' >$. We can assume that the set of states Ξ and Ξ' do not overlap (this can always be achieved by renaming the states of one of them).

Let us recall that, for the sake of simplicity, $< s_i, a_k, s_j, cond_{i,k,j} >$ is abbreviated as $\delta_{i,k,j}$ and $< s'_{i'}, a'_{k'}, s'_{j'}, cond_{i',k',j'} >$ as $\delta'_{i',k',j'}$.

The programs Π_{AUT} and $\Pi_{AUT'}$ associated with the automata AUT and AUT' are:

$$
\begin{array}{l}
\quad aut([\,], s_0(N)) \\
\left\{\ aut([a_k(M)|L], s_j(P)) \quad \leftarrow \quad cond_{i,k,j}(N, M, P),\ \ aut(L, s_i(N))\ \right\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for all } i, j, k \text{ such that } \delta_{i,k,j} \ \in \ \widehat{\Delta}
\end{array}
$$

$$
\begin{array}{l}
\quad aut'([\,], s'_0(N')) \\
\left\{\ aut'([a'_{k'}(M')|L], s'_{j'}(P')) \quad \leftarrow \quad cond'_{i',k',j'}(N', M', P'),\ \ aut'(L, s'_{i'}(P')\ \right\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for all} \quad i', j', k' \text{ such that } \delta'_{i',k',j'} \ \in \ \widehat{\Delta}'
\end{array}
$$

5.1 Intersection

Definition .3 *The* intersection automaton *of AUT and AUT' is the generalized automaton AUT^{inter} characterized by the program:*

$$aut^{inter}([\,], s_0(N), s'_0(N'))$$

$$\left\{ aut^{inter}([a_k(M)|L], s_j(P), s'_{j'}(P')) \leftarrow cond_{i,k,j}(N,M,P), cond'_{i',k',j'}(N',M',P'), \; M = M', \; aut^{inter}(L, s_i(N), s'_{i'}(N')) \right\}$$

$$\text{for all } i, i', k, k', j, j' \text{ such that } \delta_{i,k,j} \in \widehat{\Delta}, \; \delta'_{i',k',j'} \in \widehat{\Delta}', \; a_k = a'_{k'}$$

It can be easily proved:

Proposition .4 $Rec(AUT^{inter}) = Rec(AUT) \cap Rec(AUT')$

5.2 Shuffle-product

Given the two subsets A and A' of lists of actions, the shuffle product $A \sqcup\!\sqcup A'$ denotes the set of lists obtained by "interleaving" a list of action $\overline{l}$ of A with a list of action $\overline{l}'$ of A'.

Definition .5 *The* shuffle product automaton *of AUT and AUT' is the generalized automaton $AUT^{shuffle}$ characterized by the program:*

$$aut^{shuffle}([\,], s_0(N), s'_0(N'))$$

$$\left\{ aut^{shuffle}([a_k(M)|L], s_j(P), s'_{i'}(N')) \leftarrow cond_{i,k,j}(N,M,P), \; aut^{shuffle}(L, s_i(N), s'_{i'}(N')) \right\}$$

$$\text{for all } i' \text{ and all } i, k, j \text{ such that } \delta_{i,k,j} \in \widehat{\Delta}$$

$$\left\{ aut^{shuffle}([a'_{k'}(M')|L], s_i(N), s'_{j'}(P')) \leftarrow cond'_{i',k',j'}(N',M',P'), \; aut^{shuffle}(L, s_i(N), s'_{i'}(N')) \right\}$$

$$\text{for all } i \text{ and all } i', k', j' \text{ such that } \delta'_{i',k',j'} \in \widehat{\Delta}'$$

It can be easily proved:

Proposition .6 $Rec(AUT^{shuffle}) = Rec(AUT) \sqcup\!\sqcup Rec(AUT')$.

5.3 Completion

We want to construct an automaton that recognizes any list $\overline{l}$ of actions and that is able to decide whether or not $\overline{l}$ is recognized by AUT.

In this section, we consider only deterministic automata: an automaton is deterministic iff, for any action a_k and any state s_i, there is at most one transition labeled with a_k which starts from s_i. This assumption does not entail any loss of generality, since any generalized automaton is equivalent to a deterministic one (see section 7).

Definition .7 *The* completed automaton *of AUT is the generalized automaton AUT^{comp} characterized by the program:*

$aut^{comp}([\,], s_0(N))$

$$\left\{ aut^{comp}([a_k(M)|L], s_j(P)) \leftarrow cond_{i,k,j}(N,M,P),\ aut^{comp}(L, s_i(N)) \right\}$$
$$\text{for all } i,k,j \text{ such that } \delta_{i,k,j} \in \widehat{\Delta}$$

$$\left\{ aut^{comp}([a_k(M)|L], s_{error}) \leftarrow (\forall Q\ \neg cond_{i,k,j}(N,M,Q)),\ aut^{comp}(L, s_i(N)) \right\}$$
$$\text{for all } i,\ k \text{ such that: } \exists j(\delta_{i,k,j} \in \widehat{\Delta})\ ^{1}$$

$$\left\{ aut^{comp}([a_k(M)|L], s_{error}) \leftarrow aut^{comp}(L, s_i(N)) \right\}$$
$$\text{for all } i,\ k \text{ such that: } \forall j(\delta_{i,k,j} \notin \widehat{\Delta})\ ^{2}$$

$$\left\{ aut^{comp}([a_k(M)|L], s_{error}) \leftarrow aut^{comp}(L, s_{error}) \right\}$$
$$\text{for all } k$$

Note that the constraint $\forall\ Q\ \neg cond_{i,k,j}(N,M,Q)$ can always be replaced by an equivalent quantifier-free arithmetic expression (say $cond''_{i,k,j}(N,M)$) if $cond_{i,k,j}$ is a *linear* arithmetic expression (i.e., contains only $<=\ +\ \neq$, but no $\times$).

It can be easily proved:

Proposition .8 *For any ground list $\overline{l}$, any arithmetic vector $\overline{n}$ and any $i \in \{1,\ldots,v\}$:*

- *The goal $\leftarrow aut^{comp}(\overline{l}, s_i(\overline{n}))$ succeeds in Π^{comp}_{AUT} iff the goal $\leftarrow aut(\overline{l}, s_i(\overline{n}))$ succeeds in Π_{AUT}.*
- *The goal $\leftarrow aut^{comp}(\overline{l}, s_{error})$ succeeds in Π^{comp}_{AUT} iff the goal $\leftarrow aut(\overline{l}, S)$ fails in Π_{AUT}.*

It follows from proposition .8 that, as desired, the automata AUT^{comp} is able to decide whether or not a list $\overline{l}$ of actions is recognized by AUT.

5.4 An example of completed automaton

Consider the automaton AUT_2 characterized by program Π_{AUT_2}.

$aut_2([\,], s_0(N))$
$aut_2([a_1(M)|L], s_1(P)) \leftarrow M = P,\ aut_2(L, s_0(N))$
$aut_2([a_2(M)|L], s_0(P)) \leftarrow M = P,\ aut_2(L, s_1(N))$
$aut_2([a_3(M)|L], s_1(P)) \leftarrow M = N,\ aut_2(L, s_1(N))$

This automaton is depicted on figure 5.

The language recognized by AUT_2 is:

$$Rec(AUT_2) = \{[\,]\} \cup \{[a_1(M)]\} \cup \{[a_1(M), a_3(M)]\} \cup \{[a_1(M), a_2(M')]\} \cup \{[a_1(M), a_3(M), a_2(M')]\} \cup \{[a_1(M), a_2(M'), a_1(M'')]\} \cup \ldots$$

The program $\Pi^{comp}_{AUT_2}$ associated with AUT_2^{comp} is:

[1] case where there is a (single) transition labeled with a_k and starting from s_i
[2] case where there is no transition labeled with a_k and starting from s_i

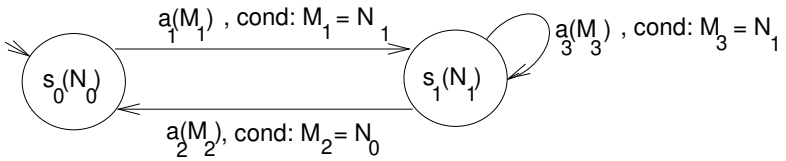

Figure 5: Automaton AUT_2

$$
\begin{array}{lcll}
aut_2^{comp}([\,], s_0(N)) & & & \\
aut_2^{comp}([a_1(M)|L], s_1(P)) & \leftarrow & M = P,\ aut_2^{comp}(L, s_0(N)) & \\
aut_2^{comp}([a_2(M)|L], s_0(P)) & \leftarrow & M = P,\ aut_2^{comp}(L, s_1(N)) & \\
aut_2^{comp}([a_3(M)|L], s_1(P)) & \leftarrow & M = N,\ aut_2^{comp}(L, s_1(N)) & \\
aut_2^{comp}([a_3(M)|L], s_{error}) & \leftarrow & aut_2^{comp}(L, s_0(N)) & \\
aut_2^{comp}([a_2(M)|L], s_{error}) & \leftarrow & aut_2^{comp}(L, s_0(N)) & \\
aut_2^{comp}([a_1(M)|L], s_{error}) & \leftarrow & aut_2^{comp}(L, s_1(N)) & \\
aut_2^{comp}([a_3(M)|L], s_{error}) & \leftarrow & M \neq N,\ aut_2^{comp}(L, s_1(N)) & \\
aut_2^{comp}([a_1(M)|L], s_{error}) & \leftarrow & (\forall Q\ M \neq Q),\ aut_2^{comp}(L, s_0(N)) & (*) \\
aut_2^{comp}([a_2(M)|L], s_{error}) & \leftarrow & (\forall Q\ M \neq Q),\ aut_2^{comp}(L, s_1(N)) & (**) \\
aut_2^{comp}([a_1(M)|L], s_{error}) & \leftarrow & aut_2^{comp}(L, s_{error}) & \\
aut_2^{comp}([a_2(M)|L], s_{error}) & \leftarrow & aut_2^{comp}(L, s_{error}) & \\
aut_2^{comp}([a_3(M)|L], s_{error}) & \leftarrow & aut_2^{comp}(L, s_{error}) &
\end{array}
$$

Note that the constraints $\forall Q\ M \neq Q$ and $\forall Q\ M \neq Q$ in clauses $(*)$ and $(**)$ can be replaced by "false". Hence these clauses can be removed.

The automaton AUT_2^{comp} is depicted on figure 6.

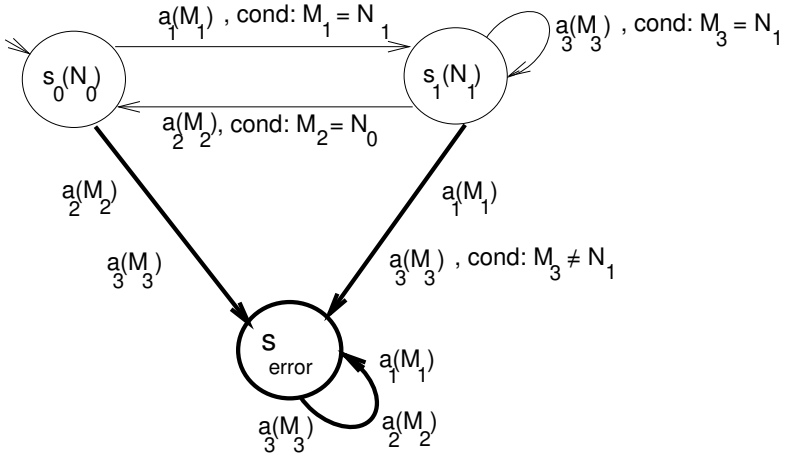

Figure 6: Automaton aut_2^{comp}

The goal $\leftarrow aut_2^{comp}([a_2(7), a_3(5), a_1(5)], s_0(0))$ succeeds in $\Pi_{AUT_2}^{comp}$ since $\leftarrow aut_2([a_2(7), a_3(5), a_1(5)], s_0(0))$ succeeds in Π_{AUT_2}.

The goal $\leftarrow aut_2^{comp}([a_2(7), a_3(5), a_1(3)], s_{error})$ succeeds in $\Pi_{AUT_2}^{comp}$ since $\leftarrow aut_2([a_2(7), a_3(5), a_1(3)], S)$ fails in Π_{AUT_2}.

6 Language Inclusion Problem

In this section, we study the language inclusion problem for generalized automata with arithmetic constraints, that is, given two generalized automata AUT and AUT', decide whether or not $Rec(AUT) \subseteq Rec(AUT')$.

From proposition .8 and the definition of the predicates aut and aut', it follows:

Proposition .9 *Given two generalized automata AUT and AUT',*
$Rec(AUT) \subseteq Rec(AUT')$ *iff* $\forall L \; \forall S \; \forall S' \; (\; aut(L, S) \; \wedge \; aut'^{comp}(L, S') \; \Rightarrow S' \neq s'_{error})$.

We now describe a method to decide the validity of:

$$aut(L, S) \; \wedge \; aut'^{comp}(L, S') \Rightarrow \; S' \neq s'_{error} \qquad (***)$$

This is done following the steps below (cf. [6]):

1. Construct the program Π^{inter} associated with the intersection automaton of AUT and AUT'^{comp}.
2. Construct the program Π'^{inter} obtained removing the list argument L and the action parameters M from the clauses of Π^{inter}.
3. Compute the output of the bottom-up evaluation process of program Π'^{inter}.

The output of the bottom-up evaluation of Π'^{inter} computed at step 3 is of the form $OUT_1 \cup OUT_2$, where:

$$OUT_1 \quad = \quad \bigcup_{\substack{i,j \\ i \in \{1,\ldots,v\} \\ j \in \{1,\ldots,v'\}}} \{ \; (s_i(N), s'_j(N')) \; / \; \varphi_{i,j}(N, N') \;\; \text{holds} \; \}$$

$$OUT_2 \quad = \quad \bigcup_{i \in \{1,\ldots,v\}} \{ \; (s_i(N), s'_{error}) \; / \; \xi_i(N) \;\; \text{holds} \; \}$$

and $\varphi_{i,j}$ and ξ_i denote arithmetic formulas.

The formula $(***)$ holds iff OUT_2 is empty.

The language inclusion problem is thus decidable when the bottom-up evaluation process of step 3 terminates. This bottom-up evaluation is guaranteed to terminate when the arithmetic constraints of AUT and AUT' contain just the symbols $=$, $\neq$, $<$ (but neither $+$ nor $\times$): in this case, the program Π'^{inter} can be finitely bottom-up evaluated using Revesz's Method [16].

Note that, even in the case where some arithmetic constraints in AUT and AUT' contain the symbol $+$, it is sometimes possible to solve the inclusion problem, by using suitable bottom-up evaluation techniques (see, e.g., [7] [11]).

Example 4

Consider the automaton AUT_1 depicted on the figure 4 and the automaton AUT_2 depicted on figure 5. Our goal is to prove that $Rec(AUT_1) \subseteq Rec(AUT_2)$.

The program $\Pi^{comp}_{AUT_2}$ associated with AUT_2^{comp} is given in subsection 5.4.

The next step consists in verifying if:

$$aut_1(L,S) \wedge aut_2^{comp}(L,S') \Rightarrow \ S' \neq s'_{error}$$

This is done following the steps 1, 2 and 3 described previously.

1. Construction of program Π^{inter}, resulting from the intersection of AUT_1 and AUT_2^{comp}. (Note that the states s_i and the parameters M_i and N_i of automaton AUT_2^{comp} are renamed as s'_i, M'_i and N'_i, respectively.)

$$
\begin{array}{lcl}
aut^{inter}([\,], s_0(N), s'_0(N')) & & \\
aut^{inter}([a_1(M)|L], s_1(P), s'_1(P')) & \leftarrow & M = M', M = N, M' = P', \\
 & & \quad aut^{inter}(L, s_0(N), s'_0(N')) \\
aut^{inter}([a_2(M)|L], s_0(P), s'_0(P')) & \leftarrow & M = M', M = P, M' = P', \\
 & & \quad aut^{inter}(L, s_1(N), s'_1(N')) \\
aut^{inter}([a_2(M)|L], s_0(P), s'_{error}) & \leftarrow & M = M', M = P, aut^{inter}(L, s_1(N), s'_0(N')) \\
aut^{inter}([a_1(M)|L], s_1(P), s'_{error}) & \leftarrow & M = M', M = N, aut^{inter}(L, s_0(N), s'_1(N')) \\
aut^{inter}([a_1(M)|L], s_1(P), s'_{error}) & \leftarrow & M = M', M = N, aut^{inter}(L, s_0(N), s'_{error}) \\
aut^{inter}([a_2(M)|L], s_0(P), s'_{error}) & \leftarrow & M = M', M = P, aut^{inter}(L, s_1(N), s'_{error})
\end{array}
$$

2. Elimination of the list variable L and the parameters M_i from program Π^{inter}, thus generating the program Π'^{inter} :

$$
\begin{array}{lcl}
aut'^{inter}(s_0(N), s'_0(N')) & & \\
aut'^{inter}(s_1(P), s'_1(P')) & \leftarrow & N = P', \ aut'^{inter}(s_0(N), s'_0(N')) \\
aut'^{inter}(s_0(P), s'_0(P')) & \leftarrow & P = P', \ aut'^{inter}(s_1(N), s'_1(N')) \\
aut'^{inter}(s_0(P), s'_{error}) & \leftarrow & aut'^{inter}(s_1(N), s'_0(N')) \\
aut'^{inter}(s_1(P), s'_{error}) & \leftarrow & aut'^{inter}(s_0(N), s'_1(N')) \\
aut'^{inter}(s_1(P), s'_{error}) & \leftarrow & aut'^{inter}(s_0(N), s'_{error}) \\
aut'^{inter}(s_0(P), s'_{error}) & \leftarrow & aut'^{inter}(s_1(N), s'_{error})
\end{array}
$$

3. The output of the bottom-up evaluation can be computed by Revesz's method [16]. Here, we have:

   $OUT_1 \ = \ \{ \ (s_0(N), s'_0(N')) \ \} \ \cup \ \{ \ (s_1(N), s'_1(N')) \ \}$

   $OUT_2 \ = \ \emptyset$

Since $OUT_2 \ = \ \emptyset$, we have:

$Rec(AUT_1) \subseteq Rec(AUT_2)$

7 Deterministic automata

Given a generalized non-deterministic automaton AUT, we now show how to construct a deterministic automaton AUT^{det} such that AUT and AUT^{det} recognize the same list of actions. Let us recall that an automaton is deterministic iff, for any action a_k and any state s_i, there is at most one transition labeled with a_k which starts from s_i.

Let $\rho(i,k)$ be the set $\{\ j\ /\ \delta_{i,k,j}\ \in\ \widehat{\Delta}\ \}$. Intuitively, the set $\rho(i,k)$ contains the indices of all the states s_j that are reached from state s_i executing action a_k. For a deterministic automaton the set $\rho(i,k)$ has at most one element, for any i and k.

The transformation is made by steps, where in each step, we merge all the states s_j (if there is more than one) that are reached from a selected state s_{i_0}, executing a selected action a_{k_0}.

More formally, in each step of our process we do the following:

1. Select i_0 and k_0 such that $\rho(i_0,k_0)$ has more than one element, say r.
2. Transform the current automaton AUT into $\widetilde{AUT}$, defined by:

$\widetilde{aut}([\,],s_0(N))$

$$\widetilde{aut}([a_{k_0}(M)|L],s_w(J,P)) \ \leftarrow \bigvee_{\substack{j \\ \delta_{i_0,k_0,j}\in\widehat{\Delta}}} (\ cond_{i_0,k_0,j}(N,M,P)\ \wedge\ J=j\ ),\ \widetilde{aut}(L,s_{i_0}(N))$$

$$\left\{\ \widetilde{aut}([a_k(M)|L],s_j(P)) \ \leftarrow\ cond_{i,k,j}(N,M,P)\ \wedge\ I=i,\ \widetilde{aut}(L,s_w(I,N))\ \right\}$$
for all i,k,j such that $\delta_{i,k,j}\ \in\ \widehat{\Delta},\ \delta_{i_0,k_0,i}\ \in\ \widehat{\Delta}$ and $\delta_{i_0,k_0,j}\ \notin\ \widehat{\Delta}$

$$\left\{\ \widetilde{aut}([a_k(M)|L],s_w(J,P)) \ \leftarrow\ cond_{i,k,j}(N,M,P),\ \wedge\ I=i\ \wedge\ J=j,\ \widetilde{aut}(L,s_w(I,N))\ \right\}$$
for all i,k,j such that $\delta_{i,k,j}\ \in\ \widehat{\Delta},\ \delta_{i_0,k_0,i}\ \in\ \widehat{\Delta}$ and $\delta_{i_0,k_0,j}\ \in\ \widehat{\Delta}$

$$\left\{\ \widetilde{aut}([a_k(M)|L],s_w(J,P)) \ \leftarrow\ cond_{i,k,j}(N,M,P)\ \wedge\ J=j,\ \widetilde{aut}(L,s_i(N))\ \right\}$$
for all i,k,j such that $\delta_{i,k,j}\ \in\ \widehat{\Delta},\ \delta_{i_0,k_0,i}\ \notin\ \widehat{\Delta}$ and $\delta_{i_0,k_0,j}\ \in\ \widehat{\Delta}$

$$\left\{\ \widetilde{aut}([a_k(M)|L],s_j(P)) \ \leftarrow\ cond_{i,k,j}(N,M,P),\ \ \widetilde{aut}(L,s_i(N))\ \right\}$$
for all i,k,j such that $\delta_{i,k,j}\ \in\ \widehat{\Delta},\ \delta_{i_0,k_0,i}\ \notin\ \widehat{\Delta}$ and $\delta_{i_0,k_0,j}\ \notin\ \widehat{\Delta}$

where w is a new state index and I and J are integer variables.

Since each step decreases the number of states (by $r-1$) and since AUT has a finite number of states, this process is guaranteed to terminate and to generate a deterministic automaton AUT^{det}.

It can be easily proved:

Proposition .10 $Rec(AUT) = Rec(AUT^{det})$

Example 5

Consider the automaton AUT_3 represented by:

$$
\begin{array}{lcl}
aut_3([\,], s_0(N)) & & \\
aut_3([a_1(M)|L], s_1(P)) & \leftarrow & M = P,\ aut_3(L, s_0(N)) \\
aut_3([a_1(M)|L], s_3(P)) & \leftarrow & M = N,\ aut_3(L, s_0(N)) \\
aut_3([a_3(M)|L], s_3(P)) & \leftarrow & M = N,\ aut_3(L, s_1(N)) \\
aut_3([a_2(M)|L], s_2(P)) & \leftarrow & N = P,\ aut_3(L, s_1(N)) \\
aut_3([a_2(M)|L], s_4(P)) & \leftarrow & M = P,\ aut_3(L, s_3(N)) \\
aut_3([a_1(M)|L], s_2(P)) & \leftarrow & aut_3(L, s_4(N))
\end{array}
$$

This automaton is depicted on figure 7.

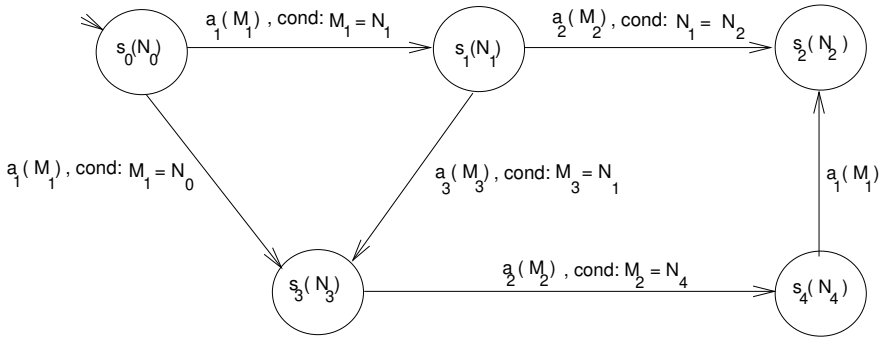

Figure 7: Automaton AUT_3

At the first step of our process, we generate the automaton $\widetilde{AUT}_3$ (depicted on figure 8):

$$
\begin{array}{lcl}
\widetilde{aut}_3([\,], s_0(N)) & & \\
\widetilde{aut}_3([a_1(M)|L], s_5(J, P)) & \leftarrow & (M = P \wedge J = 1) \vee (M = N \wedge J = 3),\ \widetilde{aut}_3(L, s_0(N)) \\
\widetilde{aut}_3([a_3(M)|L], s_5(J, P)) & \leftarrow & M = N \wedge I = 1 \wedge J = 3,\ \widetilde{aut}_3(L, s_5(I, N)) \\
\widetilde{aut}_3([a_2(M)|L], s_2(P)) & \leftarrow & N = P \wedge I = 1,\ \widetilde{aut}_3(L, s_5(I, N)) \\
\widetilde{aut}_3([a_2(M)|L], s_4(P)) & \leftarrow & M = P \wedge I = 3,\ \widetilde{aut}_3(L, s_5(I, N)) \\
\widetilde{aut}_3([a_1(M)|L], s_2(P)) & \leftarrow & \widetilde{aut}_3(L, s_4(N))
\end{array}
$$

The next step generates the automaton $\widetilde{\widetilde{AUT}}_3$ (depicted on figure 9):

$$
\begin{array}{lcl}
\widetilde{\widetilde{aut}}_3([\,], s_0(N)) & & \\
\widetilde{\widetilde{aut}}_3([a_1(M)|L], s_5(J, P)) & \leftarrow & (M = P \wedge J = 1) \vee (M = N \wedge J = 3),\ \widetilde{\widetilde{aut}}_3(L, s_0(N)) \\
\widetilde{\widetilde{aut}}_3([a_3(M)|L], s_5(J, P)) & \leftarrow & M = N \wedge I = 1 \wedge J = 3,\ \widetilde{\widetilde{aut}}_3(L, s_5(I, N)) \\
\widetilde{\widetilde{aut}}_3([a_2(M_2)|L], s_6(J, P)) & \leftarrow & (M = P \wedge I = 3 \wedge J = 4) \vee (N = P \wedge I = 1 \wedge J = 2), \\
 & & \qquad \widetilde{\widetilde{aut}}_3(L, s_5(I, N)) \\
\widetilde{\widetilde{aut}}_3([a_1(M_1)|L], s_6(J, P)) & \leftarrow & I = 4 \wedge J = 2,\ \widetilde{\widetilde{aut}}_3(L, s_6(I, N))
\end{array}
$$

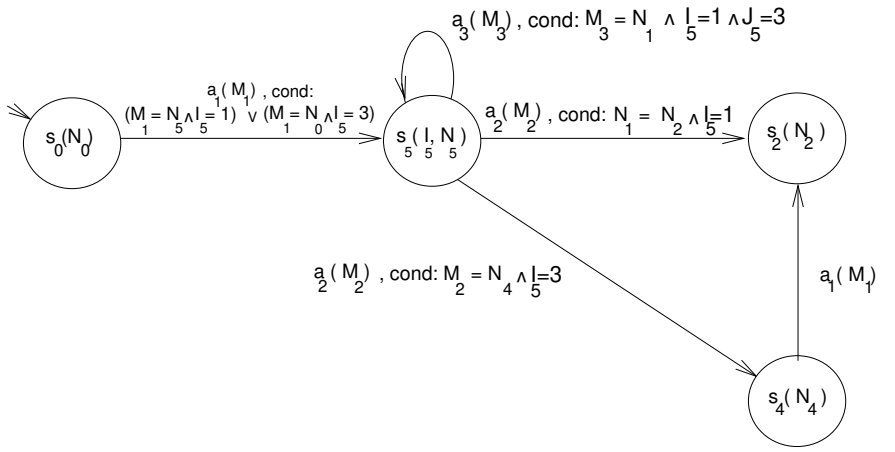

Figure 8: Automaton $\widetilde{AUT}_3$

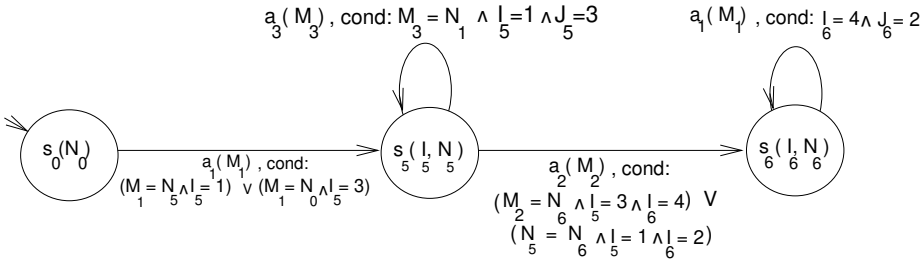

Figure 9: Automaton $\widetilde{\widetilde{AUT}}_3$

The automaton $\widetilde{\widetilde{AUT}}_3$ is deterministic, so our process terminates after two iterations ($AUT_3^{det} = \widetilde{\widetilde{AUT}}_3$).

8 Parallel composition

An operation of parallel composition for generalized automata can be defined using a combination of the intersection and the shuffle product operations. (This is similar in spirit to what is done in CSP [9].) Informally speaking, the parallel composition operation corresponds to the intersection of automata for the transitions that we want to synchronize and corresponds to the shuffle product otherwise.

Henceforth, we will assume that a special action τ belongs to Σ. This action (called "silent action") represents a spontaneous transition between states. We also assume that clauses of the form $aut([\tau|L], s_i(P)) \leftarrow N = P,\ aut(L, s_i(N))$ belong implicitly to the program Π_{AUT}.

We now explain how to perform the parallel composition of CCA on a typical example. (This construction can be generalized without any problem.)

Let AUT and AUT' be defined by the following programs.

$$
\begin{array}{lcl}
aut([\,], s_0(N)) & & \\
aut([a_1(M)|L], s_1(P)) & \leftarrow & M = P,\ aut(L, s_0(N)) \\
aut([a_2(M)|L], s_0(P)) & \leftarrow & M = P,\ aut(L, s_1(N))
\end{array}
$$

$$
\begin{array}{lcl}
aut'([\,], s'_0(N')) & & \\
aut'([a_3(M')|L], s'_1(P')) & \leftarrow & M' = N',\ aut'(L, s'_0(N')) \\
aut'([a_4(M')|L], s'_0(P')) & \leftarrow & aut'(L, s'_1(N'))
\end{array}
$$

The automata AUT and AUT' are depicted on figure 10.

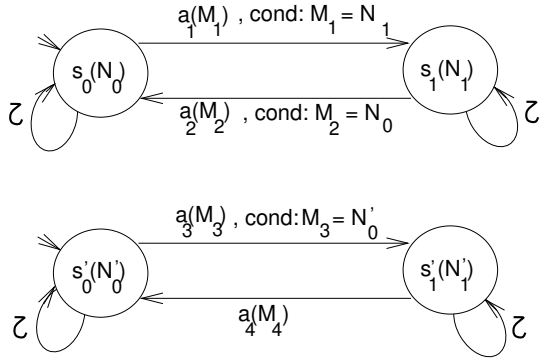

Figure 10: Automata AUT and AUT'

Suppose we want to *synchronize* the transition associated with the action $a_2(M)$ of AUT with the transition associated with the action $a_3(M')$ of AUT'. The actions a_2 and a_3 must have the same arity. First, we rename both actions $a_2(M)$ and $a_3(M')$ as τ. Then, we characterize the parallel composition automaton AUT^{paral} by the program:

$$
\begin{array}{lcl}
aut^{paral}([\,], s_0(N), s'_0(N')) & & \\
aut^{paral}([a_1(M)|L], s_1(P), s'_0(P')) & \leftarrow & M = P,\ aut^{paral}(L, s_0(N), s'_0(N')) \\
aut^{paral}([a_1(M)|L], s_1(P), s'_1(P')) & \leftarrow & M = P,\ aut^{paral}(L, s_0(N), s'_1(N')) \\
aut^{paral}([a_4(M)|L], s_0(P), s'_0(P')) & \leftarrow & aut^{paral}(L, s_0(N), s'_1(N')) \\
aut^{paral}([a_4(M)|L], s_1(P), s'_0(P')) & \leftarrow & aut^{paral}(L, s_1(N), s'_1(N')) \\
aut^{paral}([\tau|L], s_0(P), s'_1(P')) & \leftarrow & M = P,\ M' = N',\ M = M', \\
 & & \quad aut^{paral}(L, s_1(N), s'_0(N'))
\end{array}
$$

The 2^{nd}, 3^{rd} , 4^{th} and 5^{th} clauses come from the program associated with the shuffle product of AUT and AUT', while the last clause comes from the program associated with the intersection of AUT and AUT', with the additional constraint $M = M'$.

The automaton AUT^{paral} is depicted on figure 11.

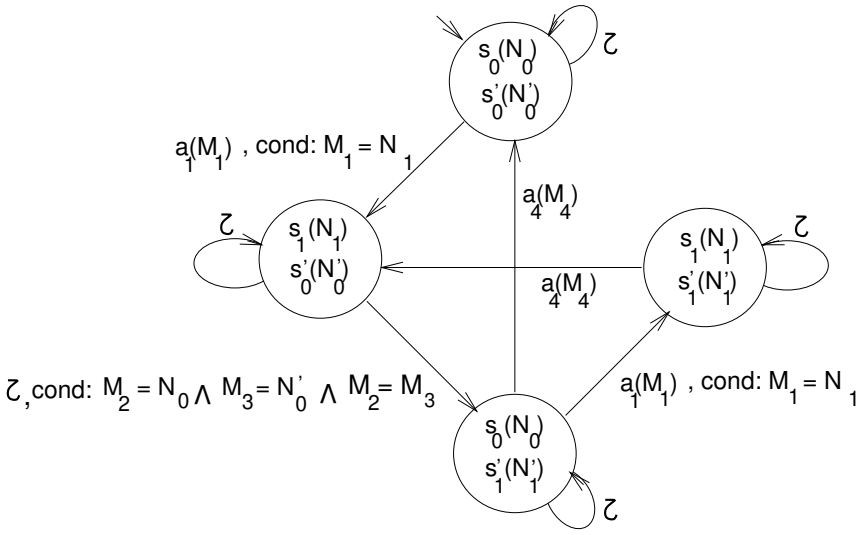

Figure 11: Automaton AUT^{paral}

9 The Alternating Bit Protocol

In this section, we apply the operation of parallel composition of automata for specifying the *Alternating Bit Protocol* and we explain how to use the decision procedure of section 6 to prove properties of this protocol. The description of this protocol is taken from [8]. (Apart from some minor differences.)

The Alternating Bit Protocol is a process composed of four entities: $SENDER$, $RECEIVER$, $MEDIUM_1$ and $MEDIUM_2$.Each of these entities can be characterized by a generalized automaton. The table below (see [8]) gives the possible actions executed shared by them.

Actions	From	To	Description
$put(M)$	environment	$SENDER$	getting a message from the environment
$send(M,B)$	$SENDER$	$MEDIUM_1$	transmission of a message
$rdt(M,B)$	$MEDIUM_1$	$RECEIVER$	re-transmission of a message
$rdte$	$MEDIUM_1$	$RECEIVER$	loss of a message
$get(M)$	$RECEIVER$	environment	sending a message to the environment
$sack(B)$	$MEDIUM_2$	$SENDER$	transmission of a bit control
$sacke$	$MEDIUM_2$	$SENDER$	loss of the bit control
$rack(M)$	$RECEIVER$	$MEDIUM_1$	re-transmission of a bit control

The alternating bit protocol can be regarded as the result of the parallel communication of the entities $SENDER, RECEIVER$, $MEDIUM_1$ and $MEDIUM_2$, as represented on figure 12(see [8]).

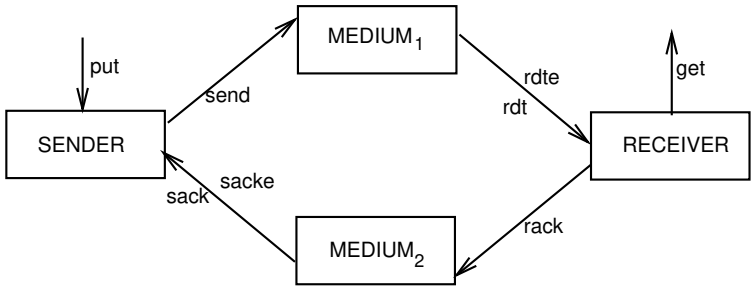

Figure 12: Alternating Bit Protocol

9.1 Process $SENDER$

The $SENDER$ process receives a message (via put), and transmits this message after adding to it a control bit B (via $send$). This bit control is initialized with the value 0 and has it value changed before each action put.

The $SENDER$ process receives a message (via put), and transmits that message after adding to it a control bit B (via $send$). This bit control is initialized with the value 0 and has it value changed before each action put.

The $SENDER$ re-sends a message if one of the following situations:

- The $SENDER$ receives a notification of loss of the message $sacke$.
- After a certain amount of time without any action (which is represented by a silent action τ), the $SENDER$ spontaneously re-send the message to prevent the occurrence of a deadlock.

We represent this process by the logic program Π_{sender}:

$$
\begin{array}{lcl}
aut_{sender}([\,], initsend(0)) & & \\
aut_{sender}([put(M)|L], readysend(M,B)) & \leftarrow & aut_{sender}(L, initsend(B)) \\
aut_{sender}([send(M,B)|L], readysack(M,B)) & \leftarrow & aut_{sender}(L, readysend(M,B)) \\
aut_{sender}([sack(1)|L], readysack(M,0)) & \leftarrow & aut_{sender}(L, readysack(M,0)) \\
aut_{sender}([sack(0)|L], readysack(M,1)) & \leftarrow & aut_{sender}(L, readysack(M,1)) \\
aut_{sender}([sacke|L], readysend(M,B)) & \leftarrow & aut_{sender}(L, readysack(M,B)) \\
aut_{sender}([\tau|L], readysend(M,B)) & \leftarrow & aut_{sender}(L, readysack(M,B)) \\
aut_{sender}([sack(0)|L], initsend(1)) & \leftarrow & aut_{sender}(L, readysack(N,0)) \\
aut_{sender}([sack(1)|L], initsend(0)) & \leftarrow & aut_{sender}(L, readysack(N,1))
\end{array}
$$

Note that, for the sake of conciseness, the form of the clauses of program Π_{sender} has been slightly simplified. For example, the clause:

$$aut_{sender}([put(M)|L], readysend(M',B)) \leftarrow M = M', B = B', \quad aut_{sender}(L, initsend(B'))$$

has been written as:

$$aut_{sender}([put(M)|L], readysend(M,B)) \leftarrow aut_{sender}(L, initsend(B))$$

The automaton AUT_{sender} is depicted on figure 13.

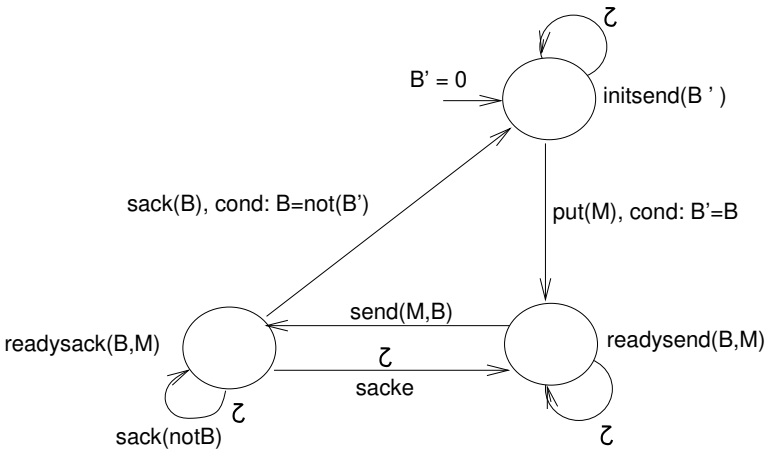

Figure 13: Process $SENDER$

9.2 Process $MEDIUM_1$

The $MEDIUM_1$ process receives a message (via $send$) with a control bit B. Then, it can behave in three different ways:

- Re-send the message and the control bit, without changing their values. In this case, the action $rdt(M, B)$ is executed.
- Lose the message and send a notification accusing this lost. In this case, the action $rdte$ is executed.
- Lose the message without sending any notification. This corresponds to the silent action τ.

We represent this process by the logic program Π_{MEDIUM_1}:

$$
\begin{array}{lcl}
medium_1([\,], initm1) & & \\
medium_1([send(M,B)|L], finalm1(M,B)) & \leftarrow & medium_1(L, initm1) \\
medium_1([rdt(M,B)|L], initm1) & \leftarrow & medium_1(L, finalm1(M,B)) \\
medium_1([rdte|L], initm1) & \leftarrow & medium_1(L, finalm1(M,B)) \\
medium_1([\tau|L], initm1) & \leftarrow & medium_1(L, finalm1(M,B))
\end{array}
$$

The automaton AUT_{medium_1} is depicted on figure 14.

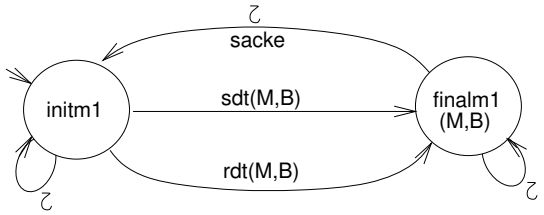

Figure 14: Process $MEDIUM_1$

9.3 Process $Medium_2$

The behaviour of $MEDIUM_2$ is analogous to the behaviour of $MEDIUM_1$. The only difference is the name of the actions that are executed.

We represent this process by the program Π_{MEDIUM_2}:

$$
\begin{array}{lcl}
medium_2([\,], initm2) & & \\
medium_2([rack(B)|L], finalm2(B)) & \leftarrow & medium_2(L, initm2) \\
medium_2([sacke|L], initm2) & \leftarrow & medium_2(L, finalm2(B)) \\
medium_2([\tau|L], initm2) & \leftarrow & medium_2(L, finalm2(B))
\end{array}
$$

The automaton AUT_{medium_2} is depicted on figure 15.

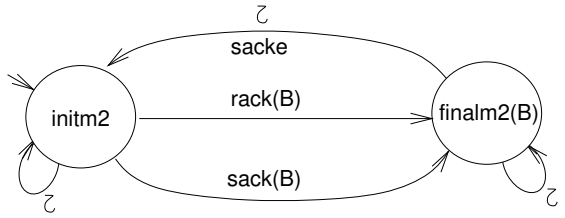

Figure 15: Process $MEDIUM_2$

9.4 Process $RECEIVER$

The $RECEIVER$ process behaves in the following ways:

- If it receives a message (via rdt), with a correct control bit B, this message is transmitted to the environment (via get) and a bit control is re-send with value B (via $rack$) .
- If it receives a message (via rdt), with a incorrect bit B, the receiver waits for another message to be received.
- If a message lost takes place (via $rdte$), a notification of message lost is omited (via $rack$).
- In order to avoid deadlocks, after a certain amount of time (represented by a silent action τ), a notification of message lost is omited by the $RECEIVER$ (via $rack$).

We represent this process by the program $\Pi_{RECEIVER}$:

$$
\begin{array}{lcl}
receiver([\,], initreceive(B)) & & \\
receiver([rdt(M,B)|L], readyget(M,B)) & \leftarrow & receiver(L, initreceive(B)) \\
receiver([get(M)|L], readyrackcorrect(B)) & \leftarrow & receiver(L, readyget(M,B)) \\
receiver([rack(0)|L], initreceive(1)) & \leftarrow & receiver(L, readyrackcorrect(0)) \\
receiver([rack(1)|L], initreceive(0)) & \leftarrow & receiver(L, readyrackcorrect(1)) \\
receiver([rdt(N,1)|L], initreceive(0)) & \leftarrow & receiver(L, initreceive(0)) \\
receiver([rdt(N,0)|L], initreceive(1)) & \leftarrow & receiver(L, initreceive(1)) \\
receiver([rdte|L], readyrackincorrect(B)) & \leftarrow & receiver(L, initreceive(B))
\end{array}
$$

$$
\begin{array}{lcl}
receiver([\tau|L], initreceive(B)) & \leftarrow & receiver(L, readyrack(B)) \\
receiver([rack(1)|L], initreceive(0)) & \leftarrow & receiver(L, readyrackincorrect(0)) \\
receiver([rack(0)|L], initreceive(1)) & \leftarrow & receiver(L, readyrackincorrect(1))
\end{array}
$$

The automaton $AUT_{receiver}$ is depicted on figure 16.

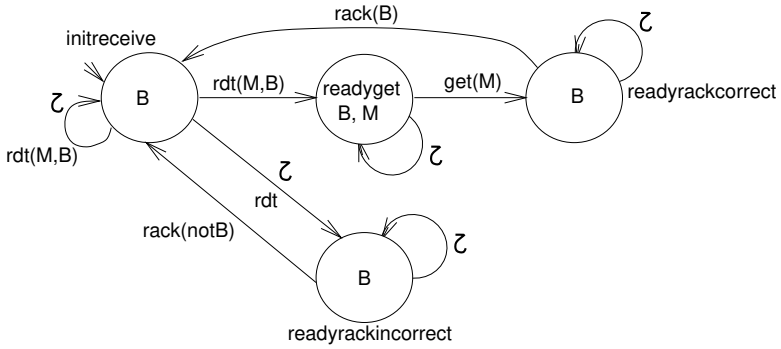

Figure 16: Process $RECEIVER$

9.5 Parallel Composition

The generalized automaton AUT_{bit} associated with the Alternating Bit Protocol is obtained by applying the operation of parallel composition to the automata associated with $SENDER$, $RECEIVER$, $MEDIUM_1$ and $MEDIUM_2$. (The synchronized actions are those which are shared by these entities (see figure 12).

Applying the parallel operation to the programs that characterize those four processes, we generate the program Π_{bit}:

$$
\begin{array}{lcl}
bit([\,], initsend(0), initm1, initreceive(0), initm2) & & \\
bit([put(M)|L], readysend(M,B), Q, R, S) & \leftarrow & bit(L, initsend(B), Q, R, S) \\
bit([\tau|L], readysack(M,0), Q, R, initm2) & \leftarrow & bit(L, readysack(M,0), Q, R, finalm2(1)) \\
bit([\tau|L], readysack(M,1), Q, R, initm2) & \leftarrow & bit(L, readysack(M,1), Q, R, finalm2(0)) \\
bit([\tau|L], readysend(M,B), Q, R, initm2) & \leftarrow & bit(L, readysack(M,B), Q, R, finalm2(B1)) \\
bit([\tau|L], initsend(1), Q, R, initm2) & \leftarrow & bit(L, readysack(M,0), Q, R, finalm2(0)) \\
bit([\tau|L], initsend(0), Q, R, initm2) & \leftarrow & bit(L, readysack(M,1), Q, R, finalm2(1)) \\
bit([\tau|L], P, initm1, readyget(M,B), S) & \leftarrow & bit(L, P, finalm1(M,B), initreceive(B), S) \\
bit([\tau|L], P, initm1, initreceive(0), S) & \leftarrow & bit(L, P, finalm1(M,1), initreceive(0), S) \\
bit([\tau|L], P, initm1, initreceive(1), S) & \leftarrow & bit(L, P, finalm1(M,0), initreceive(1), S) \\
bit([\tau|L], P, initm1, readyrackincorrect(B), S) & \leftarrow & bit(L, P, finalm1(M,B1), initreceive(B), S) \\
bit([get(M)|L], P, Q, readyrackcorrect(B), S) & \leftarrow & bit(L, P, Q, readyget(M,B), S) \\
bit([\tau|L], P, Q, initreceive(1), finalm2(0)) & \leftarrow & bit(L, P, Q, readyrackcorrect(0), initm2) \\
bit([\tau|L], P, Q, initreceive(0), finalm2(1)) & \leftarrow & bit(L, P, Q, readyrackcorrect(1), initm2) \\
bit([\tau|L], P, Q, initreceive(0), finalm2(1)) & \leftarrow & bit(L, P, Q, readyrackincorrect(0), initm2) \\
bit([\tau|L], P, Q, initreceive(1), finalm2(0)) & \leftarrow & bit(L, P, Q, readyrackincorrect(1), initm2) \\
bit([\tau|L], P, initm1, R, S) & \leftarrow & bit(L, P, finalm1(M,B1), R, S) \\
bit([\tau|L], readysend(M,B), Q, R, S) & \leftarrow & bit(L, readysack(M,B), Q, R, S) \\
bit([\tau|L], P, Q, R, initm2) & \leftarrow & bit(L, P, Q, R, finalm2(B1)) \\
bit([\tau|L], P, Q, readyrackincorrect(B), S) & \leftarrow & bit(L, P, Q, initreceive(B), S)
\end{array}
$$

$bit([\tau|L], readysack(M,B), finalm1(M,B), R, S) \leftarrow bit(L, readysend(M,B), initm1, R, S)$

Let us now explain how to apply the decision procedure of section 6 for proving functional properties of processes. Consider, for example, the property $\psi(\overline{l})$:

Within a list of actions $\overline{l}$, it is impossible to have one action $put(\overline{m}_1)$ followed by one action $get(\overline{m}_2)$, if $\overline{m}_1 \neq \overline{m}_2$.

It is easy to construct an automaton AUT_ψ such that:

$$\overline{l} \ \in \ Rec(AUT_\psi) \ \Leftrightarrow \ \psi(\overline{l})$$

This automaton is depicted on figure 17 and is characterized by the program:

$$\begin{array}{lcl}
aut_\psi([\,], N) & & \\
aut_\psi([put(M)|L], s_1(P)) & \leftarrow & M = P,\ aut_\psi(L, s_0(N)) \\
aut_\psi([get(M)|L], s_0(P)) & \leftarrow & M = N,\ aut_\psi(L, s_1(N)) \\
aut_\psi([\tau|L], s_1(P)) & \leftarrow & aut_\psi(L, s_1(N)) \\
aut_\psi([get(M)|L], s_0(P)) & \leftarrow & aut_\psi(L, s_0(N)) \\
aut_\psi([\tau|L], s_0(P)) & \leftarrow & aut_\psi(L, s_0(N))
\end{array}$$

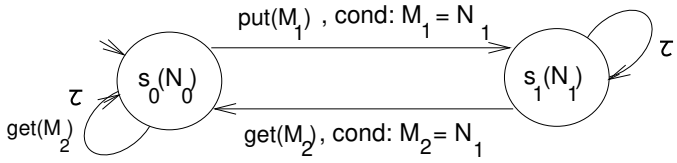

Figure 17: Property ψ

The verification that any sequence of actions executed by the Alternating Bit Protocol satisfies ψ reduces to the verification that $AUT_{bit} \subseteq AUT_\psi$. This can be done using the procedure of section 6.

10 Conclusion

We have presented an automata-based approach for specifying processes that manipulate objects defined over infinite domains. We have shown how to model the parallel composition of such processes and how to prove their functional properties: for example, we have explained how to prove that, in the alternating bit protocol, the message received at one side coincides with the message emitted at the other side. Such properties cannot be proved (and not even specified) within the framework of process algebras or finite-state systems. This explains why standard correctness proofs of protocols disregard the problems involving an unbounded number of data.

Wolper has shown, that under the assumption of "data-independence", a functional property over an infinite domain can be reduced to a property over a finite domain [21]: the property

can then be proved using standard techniques, such as model checking [4] [14] [17] or bisimulation [13]. This data-independence assumption holds in the case of the alternating bit protocol, but not for more realistic protocols. In contrast, our method can be applied even when the protocol is data-dependent (e.g., in the case of a "sliding window" protocol, where the window size is a parameter instead of a given constant). On the other hand, we are not able to address temporal issues, such as liveness or safety, within our framework. Our method can thus be seen as complementary to the standard methods used for proving concurrent systems properties. A possible extension to our theory consists in considering tree automata (cf.[3] [12]).

Acknowledgements: We would like to thank Hubert Garavel, Klaus Havelund and Michel Boyer for many helpful discussions.

References

[1] F.S. de Boer, J.N. Kok, C. Palamidessi and J.J.M.M. Rutten. "A Paradigm for Asynchronous Communication and its Application to Concurrent Constraint Programming", *Logic Programming Languages: Constraints, Functions and Objects*, K.R. Apt, J.W. de Bakker and J.J.M.M Rutten, eds., the MIT Press, chapter 4, pp. 82-114.

[2] F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc. ACM Symp. on Principles of Databases Systems*, Boston, 1986, pp. 1-15.

[3] A.C. Caron, J.L.Coquide and M. Dauchet. "Encompassment Properties and Automata with Constraints", To appear in *Proc. 5th Conf. on Rewriting Techniques and Applications*, Montreal, June, 1993.

[4] E.M. Clarke, E.A. Emerson and A.P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", *Proc. 10th ACM Symp. on Principles of Programming* Languages, Austin, 1984, pp. 117-126.

[5] S. Eilenberg. *Automata, Languages and Machines*, Academic Press, 1974.

[6] L. Fribourg. "Mixing List Recursion and Arithmetic", *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, 1991, pp. 419-429.

[7] L. Fribourg and M. Veloso Peixoto. "Bottom-up Evaluation of Datalog Programs with Arithmetic Constraints", *Technical Report LIENS 92-13*, Ecole Normale Supérieur, June 1992.

[8] H. Garavel and D. Pilaud. *Description et Applications du Language LOTOS*, Notes de cours 3$^{\text{ème}}$ année, ENSIMAG.

[9] C. A. R. Hoare. *Communicating Sequencial Processes*, Prentice-Hall Intl., 1985.

[10] J. Jaffar and J.L. Lassez. "Constraint Logic Programming", *Proc. 14th ACM Symp. on Principles of Programming Languages*, 1987, pp. 111-119.

[11] F. Kabanza, J. M. Stevenne and P. Wolper. "Handling Infinite Temporal Data", *Proc. 9th ACM Symp. on Principles of Database Systems*,Nashville, 1990, pp. 392-403.

[12] D. Lugiez and J-L.Moysset. "Tree Automata Help Solve Equational Formulae in AC-Theories", *Technical Report CRIN 93-RR-044*, Centre de Recherche en Informatique de Nancy, April 1993.

[13] R. Milner. *Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, New-York, 1980.

[14] J. Queille and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR", *Proc. 5th Int. Symp. in Programming*, Lecture Notes in Computer Science 137, Springer-Verlag, 1981.

[15] R. Ramakrishnan. "Parallelism in Logic Programs", *Proc. 17th ACM Symp. on Principles of Programming Languages*, New York, 1990, pp. 246-260.

[16] P. Revesz. "A Closed Form For Datalog Queries with Integer Order", *Proc. Int. Conf. on Database Theory*, Lecture Notes in Computer Science 470, 1990, pp. 187-201

[17] J.L. Richier, C. Rodriguez, J. Sifakis and J. Voiron. "Verification in XESAR of the Sliding Window Protocol", IFIP WG-6.1 *Proc. 7th Int. Conf. on Protocol Specification, Testing and Verification*, Amsterdam: North Holland, 1987.

[18] V.A. Saraswat and M. Rinard. "Concurrent Constraint Programming", *Proc. 17th ACM Symp. on Principles of Programming Languages*, New York, 1990, pp. 232-245.

[19] E.Y. Shapiro. "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys 21(3)*, 1989, pp. 412-450.

[20] L. Sterling and E.Y. Shapiro. *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, 1986

[21] P. Wolper. "Expressing Interesting Properties of Programs in Propositional Temporal Logic", *Proc. 13th ACM Symp. on Principles of Programming Languages*, 1986, pp. 184-193.