An Automata Based Verification Environment for Mobile Processes*

G. Ferrari[1], G. Ferro[12], S. Gnesi[2], U. Montanari[1], M. Pistore[1], G. Ristori[12]

[1] Dipartimento di Informatica, Università di Pisa
[2] Istituto di Elaborazione dell'Informazione - C.N.R., Pisa

Abstract. A verification environment for the π-calculus is presented. The environment takes a direct advantage of a general theory which allows to associate ordinary finite state automata to a wide class of π-calculus agents, so that equivalent automata are associated to equivalent π-calculus agents. A key feature of the approach is the reuse of efficient algorithms and verification techniques which have been developed and implemented for ordinary automata.

1 Introduction

Finite state labelled transition systems (also called ordinary automata) have been widely used to specify many interesting concurrent systems, e.g., communication protocols, mutual exclusion algorithms, hardware components. Moreover, efficient algorithms and practical verification techniques have been developed for ordinary automata. Hence, they form the basis of the existing semantics based verification environments for concurrent systems (e.g., [2, 4, 10]).

Verification techniques based on finite state representations of system behaviours cannot be directly applied to those concurrent systems where behaviours may refer to past steps of the ongoing computation, as happens for *history-dependent* concurrent systems. In this case, even simple agents can generate infinite state systems. An illustrative example is provided by the so called *mobile* systems, i.e. systems where the communication topology among processes can dynamically evolve when the computation progresses. Here, it is the topology of channels which depends on the history of system evolution.

The π-calculus [16] gives an example of this situation. Its primitives are simple but expressive: channel names can be created, communicated (thus giving the possibility of dynamically reconfigurating process acquaintances) and they are subjected to sophisticated scoping rules. The π-calculus has greater expressive power than ordinary process calculi, but also a much more complicated theory. In particular, the usual operational models are infinite-state and infinite branching. Hence, even though the π-calculus generalizes CCS [15], the semantic-based verification tools developed for CCS cannot be directly reused for the π-calculus.

* Work partially founded by CNR Integrated Project *Metodi e Strumenti per la Progettazione e la Verifica di Sistemi Eterogenei Connessi mediante Reti di Comunicazione* and Esprit Working Group *CONFER2*.

Similar considerations apply to the case of CCS with *localities* [3] (locations can be dynamically created and referred to when computations evolve), and to CCS with *causality* [6] (the actions a process performs refer to the actions they depend on).

To overcome these problems a generalization of the theory of ordinary automata to handle the π-calculus [18] and, more generally, history dependent behaviours of processes [19, 21], has been proposed. This has lead to the introduction of a kind of automata, called HD-automata [20], which provide a finite state and finite branching representation of history dependent behaviours. Moreover, it is possible to associate to each HD-automaton an ordinary automaton. As a consequence, many practical and efficient verification techniques developed for ordinary automata can be smoothly adapted to history dependent calculi.

This paper presents a prototype version of a verification environment for the π-calculus. The construction of the environment takes a direct advantage of the finite representation of π-calculus agents presented in [18]. In fact, the environment includes implementations of facilities which allow a π-calculus agent to be translated into an ordinary automaton. The theory of [18] ensures that equivalent ordinary automata are associated to equivalent π-calculus agents. Hence, existing equivalence checkers for ordinary automata can be used to calculate whether or not π-calculus agents are equivalent. The environment also supports verification of logical formulae expressing desired properties of the behaviour of π-calculus agents. Instead of requiring to write the formulae to be checked in the low-level logic for ordinary automata, we found convenient to introduce a logic with modalities indexed by π-calculus actions and to implement a translation of this π-logic into a standard temporal logic for ordinary automata. Again, existing model checkers can be used to verify whether or not a formula holds. This verification facility for π-calculus agents has been implemented on top of the JACK environment [10].

2 The π-calculus

Given a denumerable infinite set $\mathcal{N}$ of *names* (denoted by $a, \ldots, z$), the π-calculus *agents* over $\mathcal{N}$ are defined by the syntax[3]:

$$P ::= \mathtt{nil} \mid \alpha.P \mid P_1 \| P_2 \mid P_1 + P_2 \mid (x)P \mid [x = y]P \mid A(x_1, \ldots, x_{r(A)})$$

$$\alpha ::= \mathtt{tau} \mid x!y \mid x?(y),$$

where $r(A)$ is the range of the *agent identifier* A. The occurrences of y in $x?(y).P$ and $(y)P$ are bound; *free names* are defined as usual and $\mathtt{fn}(P)$ indicates the set of free names of agent P. For each identifier A there is a definition $A(y_1, \ldots, y_{r(A)}) := P_A$ (with y_i all distinct and $\mathtt{fn}(P_A) \subseteq \{y_1 \ldots y_{r(A)}\}$) and we assume that each identifier in P_A is in the scope of a prefix (guarded recursion).

[3] For convenience, we adopt the syntax of the agents we use to input agents in the environment. We use $(x)P$ for the restriction, $x?(y).P$ for input prefixes and $x!y.P$ for output prefixes. The syntax of the other operators is standard.

The *actions* that agents can perform are defined by the following syntax:

$$\mu ::= \mathtt{tau} \mid x!y \mid x!(z) \mid x?y;$$

where x and y are free names of μ ($\mathtt{fn}(\mu)$), whereas z is a bound name ($\mathtt{bn}(\mu)$); finally $\mathtt{n}(\mu) = \mathtt{fn}(\mu) \cup \mathtt{bn}(\mu)$.

The structural rules for the *early operational semantics* are defined in Table 1.

TAU $\mathtt{tau}.P \xrightarrow{\mathtt{tau}} P$	OUT $x!y.P \xrightarrow{x!y} P$	IN $x?(y).P \xrightarrow{x?z} P\{z/y\}$
SUM $\dfrac{P_1 \xrightarrow{\mu} P'}{P_1 + P_2 \xrightarrow{\mu} P'}$	PAR $\dfrac{P_1 \xrightarrow{\mu} P_1'}{P_1 \Vert P_2 \xrightarrow{\mu} P_1' \Vert P_2}$ if $\mathtt{bn}(\mu) \cap \mathtt{fn}(P_2) = \emptyset$	
COM $\dfrac{P_1 \xrightarrow{x!y} P_1' \quad P_2 \xrightarrow{x?y} P_2'}{P_1 \Vert P_2 \xrightarrow{\mathtt{tau}} P_1' \Vert P_2'}$	CLOSE $\dfrac{P_1 \xrightarrow{x!(y)} P_1' \quad P_2 \xrightarrow{x?y} P_2'}{P_1 \Vert P_2 \xrightarrow{\mathtt{tau}} (y)(P_1' \Vert P_2')}$ if $y \notin \mathtt{fn}(P_2)$	
RES $\dfrac{P \xrightarrow{\mu} P'}{(x)P \xrightarrow{\mu} (x)P'}$ if $x \notin \mathtt{n}(\mu)$	OPEN $\dfrac{P \xrightarrow{x!y} P'}{(y)P \xrightarrow{x!(z)} P'\{z/y\}}$ if $x \neq y, z \notin \mathtt{fn}((y)P')$	
MATCH $\dfrac{P \xrightarrow{\mu} P'}{[x=x]P \xrightarrow{\mu} P'}$	IDE $\dfrac{P_A\{y_1/x_1, \ldots, y_{r(A)}/x_{r(A)}\} \xrightarrow{\mu} P'}{A(y_1, \ldots, y_{r(A)}) \xrightarrow{\mu} P'}$	

Table 1. Early operational semantics.

Several bisimulation equivalences have been introduced for the π-calculus. They can be early [17], late [16] or open [25], and their *weak* counterparts (we refer to [23] for a complete overview). Moreover, logics have been proposed [17, 5] to express properties of π-calculus agents. These logics are extensions, with π-calculus actions and names quantifications and parameterizations, of classical action-based logics [12, 13].

3 From π-calculus agents to ordinary automata

In this section, we outline the translation steps presented in [18] that permit, given a π-calculus agent, to generate the finite state and finitely branching ordinary automaton representing the agent's behaviour. The generation of the ordinary automaton associated with a π-calculus agent has been implemented by two translation modules. The first module builds an intermediate representation, the *History Dependent Automaton* (HD-automaton in short). The second module builds the ordinary automaton starting from the HD-automaton. The generation of the ordinary automaton has been split into these two steps to achieve modularity in the structure of the verification environment. Moreover, the intermediate representation allows a more efficient implementation of the second translation step.

The ordinary automata representation is possible only for *finitary* agents with a restricted form of matching. An agent is finitary if there is a bound to the number of parallel components of all the agents reachable from it. In particular, all the *finite control* agents, i.e. the agents without parallel composition inside recursion, are finitary.

3.1 From π-calculus agents to HD-automata

HD-automata have been introduced in [18], with the name of π-automata, as a convenient structure to describe in a compact way the operational behaviours of π-calculus agents. Due to the mechanism of input, the ordinary operational semantics of the π-calculus requires an infinite number of states also for very simple agents. Many of these states, however, just differ for an injective renaming.

Like ordinary automata, HD-automata have a set of states and a set of transitions between states. States and transitions, however, are more structured entities. In fact, a finite set of names is associated to each state: these are the names which are syntactically significant in that state (in the case of π-calculus they correspond to the free names of the agents). These names are considered private, local to the state, since a single state represents a whole class of agents differing for injective renamings only. This implies that each transition must describe explicitly which name of the source state corresponds to a given name of the target state. This correspondence is represented via an injective mapping from the names of the target state to the names of the source plus the fresh names introduced by the transition.

Due to the usage of local names, a finite number of transitions is needed to model input prefixes. In fact, it is enough to consider as input values all the names which appear free in the source state plus just one fresh name: in the HD-automata it does not make sense to have more transitions which differ just in the choice of the fresh name. In [18] it has been proved that *finite* HD-automata can be built for the class of *finitary* agents.

Example 1. Consider agent $P(in, out) := in?(x).out!x.\mathtt{nil}$. Figure 1 illustrates the corresponding HD-automaton. Here, the names which are used as input values are in and out (already present in the initial state) and the fresh name x. Notice that parenthesis appears in the label of a transition when it corresponds to the input of a fresh name. In the HD-automaton of Figure 1, the targets of two input transitions (the one corresponding to the fresh name and the one for in) are merged: the corresponding agents, in fact, just differ for an injective substitution.

3.2 From HD-automata to ordinary automata

It is possible to extract from the HD-automaton of a π-calculus agent its early operational semantics. This is done by a visit of the HD-automaton during which the global meaning of the private names of the reached states is maintained. When a fresh name is introduced by a transition of the HD-automaton, a global

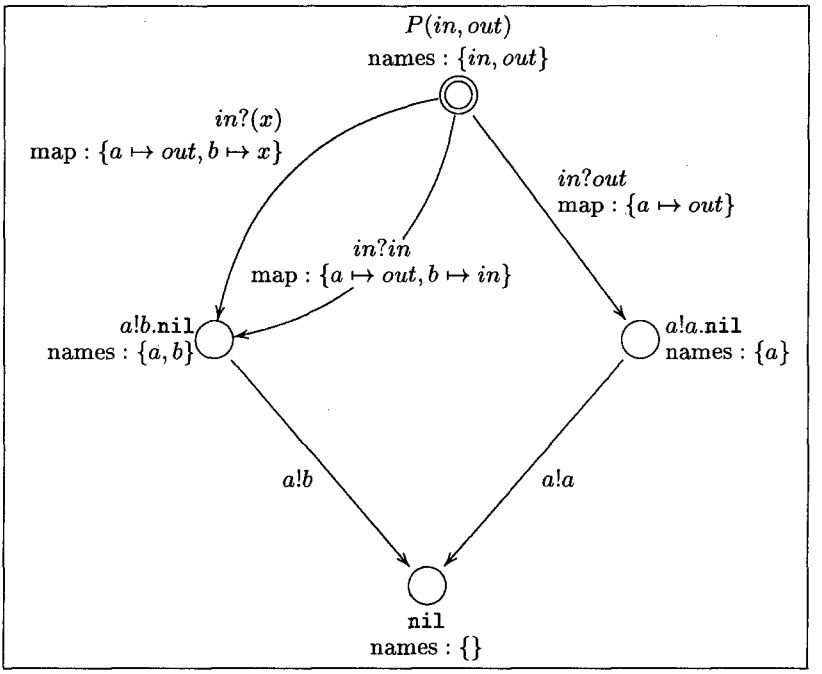

Fig. 1. The HD-automaton corresponding to the agent $P(in, out) := in?(x).out!x.\mathtt{nil}$.

instantiation has to be chosen for that name. For instance, suppose we are visiting the HD-automaton of Figure 1 starting from the initial state and from the identical mapping (i.e. the one which maps local names *in* and *out* into global names *in* and *out*, respectively). If we choose the transition $in(x)$, we have to give a global meaning, say v, to the fresh name x. Then, we reach the state $a!b.\mathtt{nil}$, where the global meaning of names a and b is respectively *out* and v. This corresponds to the early transition $P(in, out) \xrightarrow{in?v} out!v.\mathtt{nil}$.

Obviously, by choosing all the possible instantiations for the fresh names of the HD-transitions, we get an infinite automaton. To obtain a finite state automaton it suffices to take the first name which has been not already used. In this way, a finite state automaton is obtained from each finite HD-automaton.

The ordinary automaton obtained from the HD-automaton of Figure 1 is displayed in Figure 2. In the ordinary automata labels of transitions appear in quotation marks, to stress the fact that they are just strings.

To sum up, we outlined an effective procedure to map (a significant class of) π-calculus agents into finite state automata. It is not true in general, however, that equivalent (i.e. strong or weak bisimilar) π-calculus agents are mapped into equivalent (i.e. strong or weak bisimilar) ordinary automata. In fact, due to the mechanism for generating fresh names, this is true only if we can guarantee that two bisimilar agents have the same set of free names. To this purpose, the HD-automaton has to be made *irredundant* in a pre-processing phase. The irre-

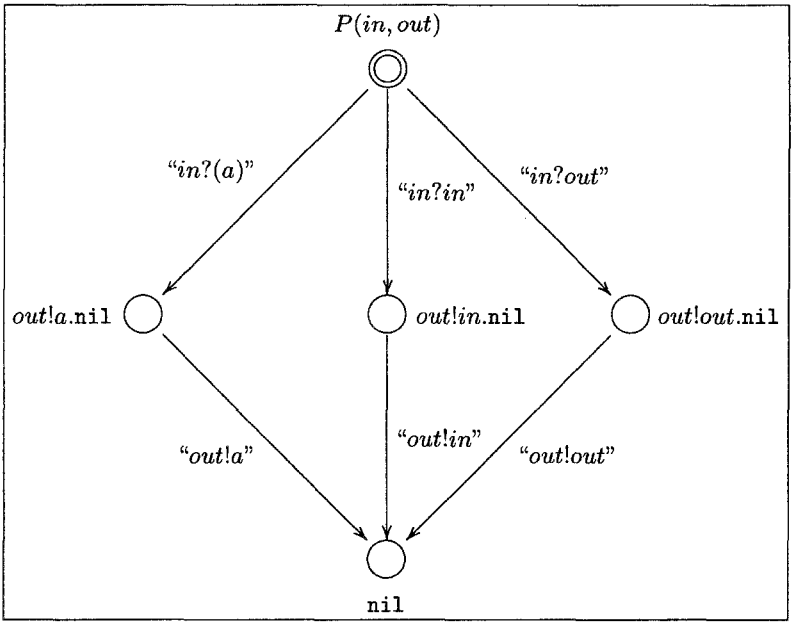

Fig. 2. The ordinary automaton corresponding to the HD-automaton of Figure 1.

dundant construction discards all the names which appear in the states of the HD-automaton but which do not play any active role in the computations from that state. In [18] a simple and efficient algorithm is described to make irredundant the HD-automata corresponding to π-calculus agents without matching. The same algorithm also works for the π-calculus with a restricted form of matching.

4 The π-logic

In this section we present the logic formalism, called π-logic, that we use to express properties of π-calculus agents. The π-logic is based on the logic formalism given in [17], in which the modal operators of the classical Hennessy-Milner logic [12] are extended to deal with π-calculus input and output actions.

Together with the strong *next* modality $EX\{\mu\}$ defined in [17], the π-logic also includes a weak *next* modality $<\mu>$ whose meaning is that a number of unobservable `tau` actions can be executed before μ.[4] Moreover, to express general liveness and safety properties, the *eventually* temporal operator (notation $EF\,\phi$)

[4] The notation <_> is generally used in the framework of modal logics to denote the strong *next* modality, while $\ll\_\gg$ is used for the weak *next* modality. Here we adopted instead the ACTL-like notation [7], where the strong *next* is denoted by EX and the weak *next* by <_>. This notation is more convenient in this case, since π-logic formulae will be mapped into ACTL formulae.

is introduced. The meaning of $EF\phi$ is that ϕ must be true sometime in a possible future.

The syntax of the π-logic is given by:

$$\phi ::= true \mid \sim\phi \mid \phi \,\&\, \phi' \mid EX\{\mu\}\phi \mid <\mu>\phi \mid EF\,\phi.$$

The interpretation of the logic formulae is the following:

- $P \models true$ holds always;
- $P \models \sim\phi$ if and only if not $P \models \phi$;
- $P \models \phi \,\&\, \phi'$ if and only if $P \models \phi$ and $P \models \phi'$;
- $P \models EX\{\mu\}\phi$ if and only if there exists P' such that $P \xrightarrow{\mu} P'$ and $P' \models \phi$;
- $P \models <\mu>\phi$ if and only if there exist $P_0, \ldots, P_n$, $n \geq 1$, such that $P = P_0 \xrightarrow{\texttt{tau}} P_1 \ldots \xrightarrow{\texttt{tau}} P_{n-1} \xrightarrow{\mu} P_n$ and $P_n \models \phi$;
- $P \models EF\,\phi$ if and only if there exist $P_0, \ldots, P_n$ and $\mu_1, \ldots, \mu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\mu_1} P_1 \ldots \xrightarrow{\mu_n} P_n$ and $P_n \models \phi$.

As usual, the following derived operators can be defined:

- $\phi \mid \phi'$ stands for $\sim(\sim\phi \,\&\, \sim\phi')$;
- $[\mu]\phi$ stands for $\sim<\mu>\sim\phi$. This is the dual version of the weak *next* operator;
- $AG\,\phi$ stands for $\sim EF\,\sim\phi$. This is the *always* operator, whose meaning is that ϕ is true now and always in the future.

The π-logic has been shown to be adequate with respect to strong early bisimulation equivalence [11].

Our purpose is to define an automatic verification procedure to check the satisfiability of a formula of the π-logic over a π-calculus agent. In Section 3 we have shown that it is possible to derive an ordinary automaton for finitary π-calculus. Hence, if we were able to translate formulae of the π-logic into "ordinary" logic formulae, it should be possible to use existing model checking algorithms to check the satisfiability of "ordinary" logic formulae over ordinary automata. As "ordinary" logics, we mean all the action-based logics that have been defined starting from the Hennessy-Milner logic [12]. Among these, we have chosen the ACTL logic [7]. The ACTL logic is adequate with respect to the strong bisimulation equivalence over ordinary automata [7], and an efficient model checker has been implemented and for which a sound translation exists [11]. Such translation is parameterized by a set of names, that is borrowed from the ordinary automaton associated to the π-agent. The translation has a worst case complexity that is exponential in the size of such set of names.

The translation of the *next* modalities indexed over output or `tau` actions is immediate (it is given by the corresponding ACTL operator in which the action argument is the action label). The translation of the *next* input (or bound output) modalities are more complex, since they have to deal with the generation of fresh names. To this purpose, syntactical scoping rules are applied so that the correct bindings are maintained through the formula when a fresh name is used as input object. The translation of the π-logic modality $<\_>$ is given in terms

of the corresponding ACTL weak operator <_>, using the same idea for input and bound output actions. The translation of the π-logic formula $EF\,\phi$ is given simply by the ACTL formula $EF\,\phi'$, where ϕ' is the ACTL translation of ϕ.

We illustrate the translation method of the *next* input modality with an example. Let ϕ be the formula $EX\{in?u\}EX\{out!u\}true$; the translation of ϕ is:

$$\phi' = EX\{\text{“}in?u\text{”}\}EX\{\text{“}out!u\text{”}\}true \mid EX\{\text{“}in?(a)\text{”}\}EX\{\text{“}out!a\text{”}\}true.$$

Consider now the agent $P(in, out) := in?(x).out!x.\mathtt{nil}$ of Example 1. The agent P satisfies the formula ϕ, since $P \xrightarrow{in?u} out!u.\mathtt{nil} \xrightarrow{out!(u)} \mathtt{nil}$ for all names u. The set of fresh names in the ordinary automaton of P (see Figure 2) is just $\{a\}$, hence to verify that P satisfies ϕ we can feed the ACTL model checker with the ordinary automaton of P and the ACTL formula ϕ'. It is easy to see that ϕ' holds in the initial state of the ordinary automaton of P. In fact, ϕ' is satisfied if there exists an explicit path, starting from the initial state, in which the actions $in?u$ and $out!u$ are executed in sequence. Otherwise, it must exist a transition from the initial state such that a fresh name, represented by a, is taken in input, followed by the output of that name, and this is actually true for the ordinary automaton of P.

5 The Verification Environment

The translation procedures described in sections 3 and 4 have been implemented on top of the JACK environment [10]. The idea behind the JACK environment[5] was to combine different specification and verification tools [14, 24, 1, 9], around a common format for representing ordinary automata: the FC2 file format [2]. FC2 makes it possible to exchange automata between JACK tools. Moreover, tools can easily be added to the JACK system, thus extending its potential.

The current prototype version of the verification environment provides facilities to construct the HD-automata of π-calculus agents, and to map HD-automata into ordinary automata represented in the FC2 format. Hence, the JACK bisimulation checker MAUTO is used to verify (strong and weak) bisimilarity of π-calculus agents. Automata minimization, according to weak bisimulation is also possible, by using the functionalities offered in JACK by the HOGGAR tool. Moreover, the ACTL model checker AMC is used for verifying properties of mobile processes, after that the π-logic formulae expressing the properties have been translated into ACTL formulae. The global architecture of the verification environment is shown in figure 3.

The complexity of our methodology is given by the construction of the state space of the π-calculus agent to be verified, that is, in the worst case, exponential in the syntactical size of the agent.

[5] Detailed information about JACK are available at:
`http://rep1.iei.pi.cnr.it/Projects/JACK`.

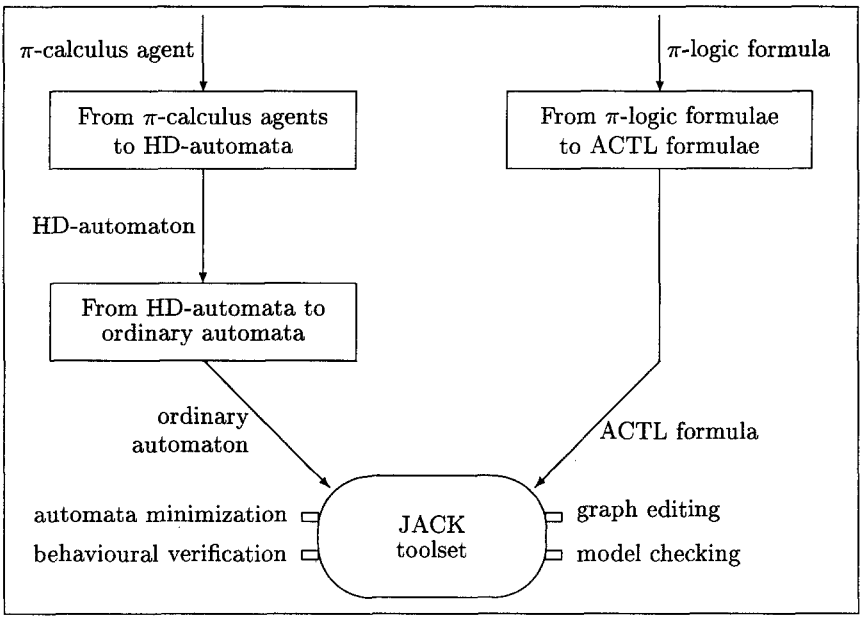

Fig. 3. The architecture of the verification environment.

Some optimizations have been also implemented. These optimizations reduce the state space of HD-automata, thus allowing a more efficient generation of the ordinary automata associated with π-calculus agents. The first optimization consists of the reduction of `tau` chains (that are unbranched sequences of `tau` transitions) to simple `tau` transitions. The other consists of the introduction of *constant* declarations. Constant names are names that cannot be used as objects of input or output actions (for instance, names that represent static channels). Since constant names are not consider as possible input values, the branching for input transitions is reduced.

6 A case study: The Handover Protocol for Mobile Telephones

In this section we illustrate the features of the verification environment. The case study we consider is the specification of the core of the handover protocol for the GSM Public Land Mobile Network proposed by the European Telecommunication Standards Institute. The specification is borrowed from that given in [26], which has been in turn derived from that in [22]. The specification consists of four modules:

- a Mobile Station MS, mounted in a car moving through two different geographical areas (cells), that provides services to an end user;

- a Mobile Switching Centre MSC, that is the controller of the radio communications within the whole area composed by the two cells;
- the Base Station modules BSa and BSp, that are the interfaces between the Mobile Station and the Mobile Switching Centre.

The observable actions performed by the Mobile Switching Centre are the input of the messages transmitted from the external environment through the channel *in*. The observable actions performed by the Mobile Station are the transmissions, via the channel *out*, of the messages to the end user. The communications between the Mobile Switching Centre and the Mobile Station happen via the base corresponding to the cell in which the car is located. When the car moves from one cell to the other, the Mobile Switching Centre starts a procedure to communicate to the Mobile Station the names of the new transmission channels, related to the base corresponding to the new cell. The communication of the new channel names to the Mobile Station is done via the base that is in use at the moment. All the communications of messages between the Mobile Switching Centre and the Mobile Station are suspended until the Mobile Station receives the names of the new transmission channels. Then the base corresponding to the new cell is activated, and the communications between the Mobile Switching Centre and the Mobile Station continue through the new base. The π-calculus specification of the Mobile Telephones is displayed in Table 2.

There are two kinds of correctness verifications that can be done in the environment. One is the checking that the specification of the System is bisimilar to a more abstract service specification. The other one is the checking of some interesting properties, expressed as π-logic formulae, that the specification must satisfy to meet the desired behaviour.

The abstract service specification (a three position buffer where the messages are queued) is described by the π-calculus agent $S0$ described below:

```
S0(in,out) := in?(v). S1(in,out,v) + tau. S0(in,out)

S1(in,out,v1) := in?(v). S2(in,out,v1,v) + out!v1. S0(in,out) + tau.
out!v1. S0(in,out)

S2(in,out,v1,v2) := in?(v). S3(in,out,v1,v2,v) + out!v1. S1(in,out,v2) +
tau. out!v1. out!v2. S0(in,out)

S3(in,out,v1,v2,v3) := out!v1. S2(in,out,v2,v3)
```

We expect that the specification satisfies the requirement that *no messages are lost*, that is whenever a message *msg* is received from the external environment through the channel *in* then it will be eventually retransmitted to the end user via the channel *out*. The formula:

$$AG([in?msg]EF < out!msg > true)$$

represents this property. Moreover, we require that the formula

$$AG([in?msg0][in?msg1][in?msg2] < out!msg0 > true)$$

```
CC(fa,fp,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel) :=
in?(v).fa!data.fa!v.CC(fa,fp,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel) +
l?(mnew).fa!ho_cmd.fa!mnew.(fp?(c).[c=ho_com]fa!ch_rel.fa?(mold).l!mold.

CC(fp,fa,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel) +
fa?(c).[c=ho_fail]l!mnew.CC(fa,fp,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel))

HC(l,m) := l!m.l?(m).HC(l,m)

MSC(fa,fp,m,in,data,ho_cmd,ho_com,ho_fail,ch_rel) := (l)(HC(l,m) ||
CC(fa,fp,l,in,data,ho_cmd,ho_com,ho_fail,ch_rel))

BSa(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) :=
f?(c).([c=data]f?(v).m!data.m!v.
BSa(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) +
[c=ho_cmd]f?(v).m!ho_cmd.m!v. (f?(c).[c=ch_rel]f!m.
BSp(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) +
m?(c).[c=ho_fail]f!ho_fail.
BSa(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail)))

BSp(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) :=
m?(c).[c=ho_acc]f!ho_com.BSa(f,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail)

MS(m,data,ho_cmd,out,ho_acc,ho_fail) :=
m?(c).([c=data]m?(v).out!v.MS(m,data,ho_cmd,out,ho_acc,ho_fail) +
[c=ho_cmd] m?(mnew). (mnew!ho_acc.MS(mnew,data,ho_cmd,out,ho_acc,ho_fail) +
m!ho_fail.MS(m,data,ho_cmd,out,ho_acc,ho_fail)))

P(fa,fp,in,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) :=
(m)(MSC(fa,fp,m,in,data,ho_cmd,ho_com,ho_fail,ch_rel) ||
BSp(fp,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail))

Q(fa,out,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) :=
(m)(BSa(fa,m,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) ||
MS(m,data,ho_cmd,out,ho_acc,ho_fail))

System(in,out) := (ho_acc)(ho_com)(data)(ho_cmd)(ch_rel)(ho_fail)
(fa)(fp)(P(fa,fp,in,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail) ||
Q(fa,out,ho_acc,ho_com,data,ho_cmd,ch_rel,ho_fail))
```

Table 2. Mobile telephones in π-calculus

holds. Namely, whenever three messages $msg0$, $msg1$ and $msg2$ are received in sequence through the channel *in*, then $msg0$ can be soon retransmitted to the end user through the channel *out*. We also expect that the formulae:

$$AG([in?msg] < out!msg > true)$$

$$AG([in?msg1][in?msg2] < out!msg1 > true)$$

are not satisfied. Indeed, it may happen that a message msg, just received through the channel *in*, cannot be soon given in output through the channel *out*: there can be other messages received before msg that are waiting for being transmitted through *out*. Similarly, it can be false that if two messages $msg1$ and $msg2$ are received in sequence through the channel *in* then $msg1$ can be soon retransmitted through the channel *out*. This is because another message (that has been received before $msg1$) may exist and still waiting for being retransmitted via *out*.

An account of the verification steps performed by exploiting the facilities of the environment (running on a Sun Workstation Spark Ultra 1) is presented below.

6.1 Automata generation

1. *Input of the specifications.* We give as input to the verification environment the the agents System and $S0$. We also declare that the names *in* and *out* must be considered as constant names.
2. *Generation of the HD-automata.* We build the HD-automata associated to System and to $S0$. Moreover, `tau`-chains are removed from the HD-automaton of the System to reduce its state space.

```
>aSystem := build System
used time 3973.75 sec.
number of states : 37199
number of transitions : 47958

>remove-tau-chains aSystem
number of states : 11015
number of transitions : 21774

>aS0 := build S0
used time 0.0834961 sec.
number of states : 13;
number of transitions : 25
```

3. *Generation and minimization of the ordinary automata.* We build the ordinary automata (represented in the FC2 format) associated to System and to $S0$. The ordinary automaton of the System has 32,263 states and 62,990 and it has been built in 441.983 sec. The ordinary automaton of $S0$ has 50 states and 94 transitions. We then minimize the ordinary automaton associated to

System by the weak bisimulation minimization facility offered in JACK by the HOGGAR tool. The minimized ordinary automaton thus obtained has 49 states and 91 transitions.

6.2 Bisimulation checking and logic verification

1. *Bisimulation checking.* We verify the weak equivalence between the minimized automaton and the ordinary automaton of the abstract service specification, by using the *obseq* functionality offered inside JACK by the MAUTO tool:

```
 MAUTO
Version v2-6beta (17 Nov 1994)
@ set A = include-fc2-automaton "System_W";; A : Automaton
@ set B = include-fc2-automaton "Spec";; B : Automaton
@ obseq(A,B);;
True : Bool
```

2. *Model checking.* We perform the model checking of the ACTL formulae, obtained as a result of the logic translation, on the ordinary automaton associated with the System, by using the model checking functionalities offered in JACK by the AMC tool.

```
Model Checker for ACTL Version 1.12.1 (31-7-94)
top > load "System.fc2"
Taking input from System.fc2...
time: (user: 7.07 sec, sys: 0.21 sec)
top >eval
AG((~(<"in?msg">~EF(<"out!msg">true) | <"in?(#0)">~EF(<"out!#0">true)
...
The formula is TRUE in state 0 time: (user: 0.91 sec, sys: 0.00 sec)

AG((~(<"in?msg0">~~(<"in?msg1">~~(<"in?msg2">~<"out!msg0">true
...
The formula is TRUE in state 0 time: (user: 15.66 sec, sys: 0.40 sec)

AG((~(<"in?msg">~<"out!msg">true | <"in?(#0)">~<"out!#0">true
...
The formula is FALSE in state 0 time: (user: 0.96 sec, sys: 0.00 sec)

AG((~(<"in?msg1">~~(<"in?msg2">~<"out!msg1">true | <"in?(#0)">
 ...
The formula is FALSE in state 0 time: (user: 3.94 sec, sys: 0.00 sec)
```

7 Concluding Remarks

We presented an automata-based verification environment for the π-calculus. Our approach requires the construction of the whole state space of the agents. Hence, the complexity of our methodology is given by the construction of the state space of the π-calculus agent to be verified, that is, in the worst case, exponential in the syntactical size of the agent.

To end the paper we make a more detailed comparison with related works. In the *Mobility Workbench* [26] (MWB in short) the verification of bisimulation equivalence between (finite control) π-calculus agents is made *on the fly* [8], that is the state spaces of the agents are built during the construction of the bisimulation relation. Checking bisimilarity is, in the worst case, exponential in the syntactical size of the agents to be checked. The model checking functionality offered by the MWB is based on the implementation of a tableau-based proof system [5] for the Propositional μ-calculus with name-passing (an extension of μ-calculus in which it is possible to express name parameterization and quantifications over the communication objects). The main difference between our approach and the one adopted in the MWB is that in our environment the state space of a π-calculus agent is built once and for all. Hence, it can be minimized with respect to some minimization criteria and then used for behavioural verifications and for model checking of logical properties. It has to be noticed that the π-logic we use is expressive enough to describe interesting safety and liveness properties of π-calculus agents, without using fixed point operators. However, it is less expressive than the Propositional μ-calculus with name-passing used in the MWB.

As a future development, we plan to extend the verification environment to deal with other history dependent calculi [3, 6]. Due to the modular structure of the environment it suffices to add HD-translation modules following the constructions of [19, 21].

References

1. A. Bouali and R. De Simone. Symbolic bisimulation minimization. In *Proc. CAV'92*, LNCS 663. Springer-Verlag, 1992.
2. A. Bouali, A. Ressouche, V. Roy and R. De Simone. The FC2Tools set. In *Proc. CAV'96*, LNCS 1102. Springer Verlag, 1996.
3. G. Boudol, I. Castellani, M. Hennessy and A. Kiehn. Observing localities. *Theoretical Computer Science*, 114: 31–61, 1993.
4. R. Cleveland, J. Parrow and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems* 15(1):36-72, 1003.
5. M. Dam. Model checking mobile processes. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
6. Ph. Darondeau and P. Degano. Causal trees. In *Proc. ICALP'89*, LNCS 372. Springer-Verlag, 1989.

7. R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Proc. Ecole de Printemps on Semantics of Concurrency*, LNCS 469. Springer Verlag, 1990.
8. J.-C. Fernandez and L. Mounier. "On the fly" verification of behavioral equivalences and preorders. In *Proc. CAV'91*, LNCS 575. Springer Verlag, 1991.
9. G. Ferro. AMC: ACTL Model Checker. Reference Manual. IEI-Internal Report, B4-47, 1994.
10. S. Gnesi. A formal verification environment for concurrent systems design. In *Proc. Automated Formal Methods Workshop*, ENTCS 5. Elsevier, 1996.
11. S. Gnesi, G. Ristori. A Model Checking algorithm for π-calculus agents. IEI-Internal Report, B4-25, October 1996.
12. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM* 32 (1), pp. 137-161, 1985.
13. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science* 27, pp. 333-354, 1983.
14. E. Madelaine and D. Vergamini. AUTO: A verification tool for distributed systems using reduction of finite automata networks. Formal Description Techniques II, 1990.
15. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
16. R. Milner, J. Parrow and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
17. R. Milner, J. Parrow and D. Walker. Modal logic for mobile processes. In *Proc. CONCUR'91*, LNCS 527. Springer Verlag, 1992.
18. U. Montanari and M. Pistore. Checking bisimilarity for finitary π-calculus. In *Proc. CONCUR'95*, LNCS 962. Springer Verlag, 1995.
19. U. Montanari and M. Pistore. Minimal Transition Systems for History-Preserving Bisimulation. To appear *Proc. STACS'97*, LNCS, 1997.
20. U. Montanari and M. Pistore. History Dependent Automata. To appear as Technical Report, Department of Computer Science, University of Pisa.
21. U. Montanari, M. Pistore and D. Yankelevich. Efficient minimization up to location equivalence. In *Proc. ESOP'96*, LNCS 1058. Springer-Verlag, 1996.
22. F. Orava and J. Parrow. An Algebraic Verification of a Mobile Network" *Formal Aspects of Computing* 4, pp. 497-543.
23. P. Quaglia. *The π-calculus with explicit substitutions.* PhD thesis TD-09/96. Dipartimanto di Informatica, Università di Pisa, 1996.
24. V. Roy and R. De Simone. AUTO and autograph. In *Proc. CAV'90*, LNCS 531. Springer Verlag, 1990.
25. D. Sangiorgi. A theory of bisimulation for the π-calculus. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
26. B. Victor and F. Moller. The Mobility Workbench — A tool for the π-calculus. In *Proc. CAV'94*, LNCS 818. Springer Verlag, 1994.