APPENDIX 1.3.4

G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model checking verification environment for mobile processes. *Submitted to ACM TOSEM (under revision)*, 2002.

A model checking verification environment for mobile processes

G. Ferrari[1], S. Gnesi[2], U. Montanari[1], M. Pistore[3]

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

Abstract. In this paper a semantic-based environment for checking properties of π-calculus agents is presented. The verification environment exploits a novel foundational model which allows to associate ordinary finite state automata to a wide class of π-calculus agents, and is built on top of an efficient model checker that checks the satisfiability of formulae of a modal logic on finite state automata.

1 Introduction

Mobility is becoming increasingly important in the design and development of current and future wide area network (WAN) applications. This programming paradigm has been exploited in the design of several applications ranging from distributed information retrieval, to wireless-based services and had-hoc networking. We refer to [15, 34] for a detailed analysis and discussion of these issues.

Since WAN applications are inherently open and distributed, and exhibit crucial requirements like security and reliability, effective methods to support property certification are clearly required. With property certification we mean the ability of formally expressing and verifying properties (e.g. security levels of applications, trust in migrating components, safe and authorized resource accesses, and so on).

Automatic methods for verifying finite state concurrent systems have been showed to be surprisingly effective [4]. Indeed, finite state verification techniques have enjoyed substantial and growing use over the last years. For instance several communication protocols and hardware designs of considerable complexity have been formalized and proved correct by exploiting finite state verification techniques.

The advent of world-wide networks and wireless communications are contributing to a growing interest in dynamic and reconfigurable systems. Unfortunately, finite state verification of these kinds of systems is much more difficult. Indeed, in this case, even simple systems can generate infinite state spaces. An illustrative example is provided by the π-calculus [24]. The π-calculus is a paradigmatic example of a name passing process calculus. Its primitives are simple but expressive: channel names can be created, communicated (thus giving the possibility of dynamically reconfiguring process acquaintances) and they are

subjected to sophisticated scoping rules. The π-calculus has greater expressive power than ordinary process calculi, but the possibility of dynamically generating new names leads also to a much more complicated theory. In particular, the usual operational models are infinite-state and infinite branching, thus making verification via semantic equivalence a difficult task.

In the last few years two of the authors have developed a novel foundational model (and the corresponding proof techniques) to avoid the state explosion problem in the verification of systems of mobile processes specified in the π-calculus. This work has lead to the introduction of a finite state model, called *History Dependent Automata* (shortly *HD-automata*). HD-automata provides an adequate operational model for the π-calculus: the theory ensures that finite state, finite branching automata give a faithful representation of the behaviour of π-calculus agents [27–29]. As ordinary automata, HD-automata are composed of states and of transitions between states. However, states and transitions of HD-automata are enriched with sets of local names. In particular, each transition can refer to the names associated to its source state but can also introduce new names, which can then appear in the destination state. Hence, names are not global and static entities but they are explicitly represented within states and transitions and can be dynamically created.

More in general, HD-automata are expressive enough to represent formalisms equipped with mobility, locality, and causality primitives [29]. An important point is that for a wide class of processes (e.g. finitary π-calculus agents) the resulting HD-automata are finite state. Furthermore, it is possible to construct to each HD-automaton an ordinary automaton in such a way that equivalent HD-automata are mapped into equivalent ordinary automata, and finite state HD-automata are mapped into finite state ordinary automata. As a consequence, many practical and efficient verification techniques developed for ordinary automata can be smoothly adapted to the case of mobile processes.

Formally, HD automata are automata on top of a permutation algebras of states which describes the effect of name permutations (i.e. renaming) on state transitions. This information is sufficient to describe in a semantically correct way the creation, communication, and deallocation of names: all the features needed to describe and reason about formalisms with name-binding operations. Recently, several efforts have been devoted to understand naming issues independently of any particular formalism [16, 17]. The basic ingredients of these approaches are (again) a set of names and an action of its group of permutations (renaming substitutions).

This paper describes an environment which supports verification of mobile systems specified in the π-calculus: the *HD Automata Laboratory* (HAL). An overview of the current implementation of the HAL environment is presented. The HAL environment includes modules which implement decision procedures to calculate behavioural equivalences, and modules which support verification by model checking of properties expressed as formulae of suitable temporal logics. The construction of the model checker takes direct advantage of the finite representation of π-calculus agents presented in [27]. In particular, we introduce

an high level logic with modalities indexed by π-calculus actions and we provide a mapping which translates logical formulae into a classical modal logic for standard automata. The distinguished and innovative feature of our approach is that the translation mapping is driven by the finite state representation of the system (the π-calculus process) to be verified.

The paper is organized as follows. Section 2 reviews the π-calculus and the modal logic we use to express behavioural properties of π-calculus agents. This section introduces the main notations and definitions that will be used throughout the paper. We then proceed to introduce the translation mapping from π-calculus agents to HD-automata and from HD-automata to ordinary automata. The translation mapping from the higher order logic to the bare logic is presented in Section 4. Section 5 describes the main modules of the verification environment. Finally, Section 6 illustrates a verification case study: the Handover Protocol for Mobile Telephones.

2 Background

This section briefly reviews the π-calculus and introduces a basic modal logic to express properties about the behaviour of π-calculus agents. We start by reviewing the basic notions and notations of automata.

2.1 Ordinary Automata

Automata (or labelled transition systems) have been defined in several ways. We chose the following definition since it is rather natural and it can be easily modified to introduce HD-automata.

Definition 1. *An ordinary automaton is a 4-tuple $\mathcal{A} = (Q, q_0, Act, R)$, where:*

- *Q is a finite set of states;*
- *q_0 is the initial state;*
- *Act is a finite set of action labels;*
- *$R \subseteq Q \times Act \times Q$ is the transition relation. Whenever $(q, \lambda, q') \in R$ we will write $q \xrightarrow{\lambda} q'$.*

Several notions of behavioral preorders and equivalences have been defined on automata. Here, we review the notion of *bisimilarity* [23, 31].

Definition 2 (bisimulation on automata). *Let A_1 and A_2 be two automata on the same set L of labels. A relation $\mathcal{R} \subseteq Q_1 \times Q_2$ is a* simulation *for A_1 and A_2 if $q_1 \mathcal{R} q_2$ implies:*

for all $t_1 : q_1 \xrightarrow{\lambda} q_1'$ of A_1 there exists $t_2 : q_2 \xrightarrow{\lambda} q_2'$ of A_2 such that $q_1' \mathcal{R} q_2'$.

A relation $\mathcal{R} \subseteq Q_1 \times Q_2$ is a bisimulation *for A_1 and A_2 if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations.*
Two automata A_1 and A_2 are bisimilar, *written $A_1 \sim A_2$, if there is some bisimulation $\mathcal{R}$ for A_1 and A_2 such that $q_{01} \mathcal{R} q_{02}$.*

2.2 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[1]:

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

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

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).

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.

$$\text{TAU } \mathtt{tau}.P \xrightarrow{\mathtt{tau}} P \qquad \text{OUT } x!y.P \xrightarrow{x!y} P \qquad \text{IN } x?(y).P \xrightarrow{x?z} P\{z/y\}$$

$$\text{SUM } \frac{P_1 \xrightarrow{\mu} P'}{P_1 + P_2 \xrightarrow{\mu} P'} \qquad \text{PAR } \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \| P_2 \xrightarrow{\mu} P_1' \| P_2} \text{ if } \mathtt{bn}(\mu) \cap \mathtt{fn}(P_2) = \emptyset$$

$$\text{COM } \frac{P_1 \xrightarrow{x!y} P_1' \quad P_2 \xrightarrow{x?y} P_2'}{P_1 \| P_2 \xrightarrow{\mathtt{tau}} P_1' \| P_2'} \qquad \text{CLOSE } \frac{P_1 \xrightarrow{x!(y)} P_1' \quad P_2 \xrightarrow{x?y} P_2'}{P_1 \| P_2 \xrightarrow{\mathtt{tau}} (y)(P_1' \| P_2')} \text{ if } y \notin \mathtt{fn}(P_2)$$

$$\text{RES } \frac{P \xrightarrow{\mu} P'}{(x)P \xrightarrow{\mu} (x)P'} \text{ if } x \notin \mathtt{n}(\mu) \qquad \text{OPEN } \frac{P \xrightarrow{x!y} P'}{(y)P \xrightarrow{x!(z)} P'\{z/y\}} \text{ if } x \neq y, z \notin \mathtt{fn}((y)P')$$

$$\text{MATCH } \frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'} \qquad \text{IDE } \frac{P_A\{y_1/x_1, \ldots, y_{r(A)}/x_{r(A)}\} \xrightarrow{\mu} P'}{A(y_1, \ldots, y_{r(A)}) \xrightarrow{\mu} P'}$$

Table 1. Early operational semantics.

Several bisimulation equivalences have been introduced for the π-calculus [33]. They can be strong or weak, early [25], late [24] or open [36]. We now introduce the early bisimulation.

Definition 3. *A binary relation B over a set of agents is a strong early bisimulation if it is symmetric and, whenever $(P, Q) \in B$, we have that:*

[1] 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.

– *if* $P \xrightarrow{\mu} P'$ *and* $\mathtt{fn}(P,Q) \cap \mathtt{bn}(\mu) = \emptyset$, *then there exists* Q' *such that* $Q \xrightarrow{\mu} Q'$ *and* $(P', Q') \in B$.

Two terms are said *strong early bisimilar*, written $P \simeq Q$, if there exists a bisimulation B such that $(P, Q) \in B$.

2.3 A temporal logic for π-calculus agents

Some programming logics have been proposed [6, 25] 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 [19, 20].

We now introduce the logic we use to specify behavioural properties of π-calculus agents. The logic, called π-logic, extends the modal logic introduced in [25] with some expressive modalities. Indeed, together with the strong *next* modality $EX\{\mu\}$ defined in [25], 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.[2].

Moreover, to express general liveness and safety properties, the *eventually* temporal operator (notation $EF\,\phi$) 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:

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

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 $P = P_0 \xrightarrow{\mathtt{tau}} P_1 \ldots \xrightarrow{\mathtt{tau}} P_{n-1} \xrightarrow{\mu} P_n$ 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.

[2] 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 denote instead the the strong *next* by EX and the weak *next* by $<_>$

Theorem 1. *[18] The π-logic is adequate with respect to strong early bisimulation equivalence.*

For this logic we will show later on in this paper it is possible to provide a classical model checking algorithm to verify the satisfiability of the π-logic formulae on π-calculus agents. The construction of the model checker for the π-logic will exploit and re-use the model checker implemented for the ACTL logic [9, 8]. The branching time temporal logic ACTL is the action based version of CTL [10]. ACTL is well suited to describe the behavior of a system in terms of the actions it performs at its working time. In fact, ACTL embeds the idea of "evolution in time by actions" and is suitable for describing the various possible temporal sequences of actions that characterize a system behavior. The complete definition of ACTL syntax and semantics is presented in the Appendix.

3 From π-calculus agents to ordinary automata

In this section, we outline the translation steps 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 consists of two stages. The first stage constructs an intermediate representation of agent's behaviour taking advantage of the notion of HD-automaton. The second stage 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.

3.1 From π-calculus agents to HD-automata

HD-automata have been introduced in [27], with the name of π-automata, as a convenient structure to describe in a compact way the operational behaviours of π-calculus agents. HD-automata have been further generalized to deal with name passing process calculi, process calculi equipped with location, causality and Petri Nets [29, 28].

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. The creation of a new name gives rise to an infinite set of transitions: one for each choice of the new name. To handle this problems in HD-automata names appear explicitly in states, transitions and labels. Indeed, it is convenient to assume that the names which appear in a state, a transition or a label of a HD-automaton are *local* names and do not have a global identity. In this way, for instance, a single state of the HD-automaton can be used to represent all the states of a system that differ just for a bijective renaming. However, each transition is required to represent explicitly the correspondences between the names of source, target and label.

Definition 4. *A* history-dependent automaton *(*HD-automaton*) is a structure* $\mathcal{A} = (Q, q_0, Act, \omega, q \xrightarrow[\ell,\sigma]{\lambda} q')$*, where:*

- *Q is a finite set of states;*
- *q_0 is the initial state;*
- *Act is a set of action labels;*
- *ω is a function associating (finite sets of local) names to states:* $\omega : Q \longrightarrow \wp_f(\mathcal{N})$*;*
- *$q \xrightarrow[\ell,\sigma]{\lambda} q'$ is the transition relation where:*
  - *$\ell \in \omega(q)$ is the name referred to in the transition;*
  - *$\sigma : \omega(q') \longrightarrow \omega(q) \cup \{*\}$ is the (injective) embedding function, and $*$ is a new name.*

Function σ embeds the names of the target state in the names of the source state of the transition. The distinguished symbol $*$ is used to handle the creation of a new name: the name created during the transition is associated to $*$. Notice that the names that appear in the source and not in the target of the transition are discarded in the evolution. HD-automata require name creation must be handled explicitly, whereas name discarding can behave silently.

As pointed out in [27] the usage of local names allows modeling execution of input prefixes by a finite number of transitions: it is enough to consider as input values all the names which appear free in the source state plus just one fresh name. In other words, in the case of the HD-automata it does not make sense to have more transitions which differ just in the choice of the fresh name.

Example 1. Consider agent $P(in, out) := in?(x).out!x.\mathtt{nil}$. Figure 1 illustrates the corresponding HD-automaton.

Here, we adopted the following notational conventions. Local names of states (i.e. the result of function ω) are graphically represented by the finite set called *names*. The names which are used as input values are *in* and *out* (i.e. the local names of the initial state) and the fresh name x (to simplify reading of the graphical representation, we avoided usage of the distinguished symbol $*$). Moreover, labels of the form $in?(x)$ are used to denote the input of a fresh name.

The meaning of names of state changes (i.e. the embedding function from the names of the target state to the name of the source state) is represented by the function *map* labelling the transition. For instance, consider the transition

$$P(in, out) \xrightarrow[\text{map:}\{a \mapsto out, b \mapsto x\}]{in?(x)} a!b.nil$$

of Figure 1. The corresponding embedding function σ, $\sigma : \{a, b, *\} \longrightarrow \{in, out\}$ is defined to be $\sigma(a) = out, \sigma(*) = in$. Finally, in the HD-automaton of Figure 1, the targets of two input transitions originated from the initial state (namely the input of a new name, and the input of the name *in*) are merged: the corresponding agents, in fact, just differ for an injective substitution.

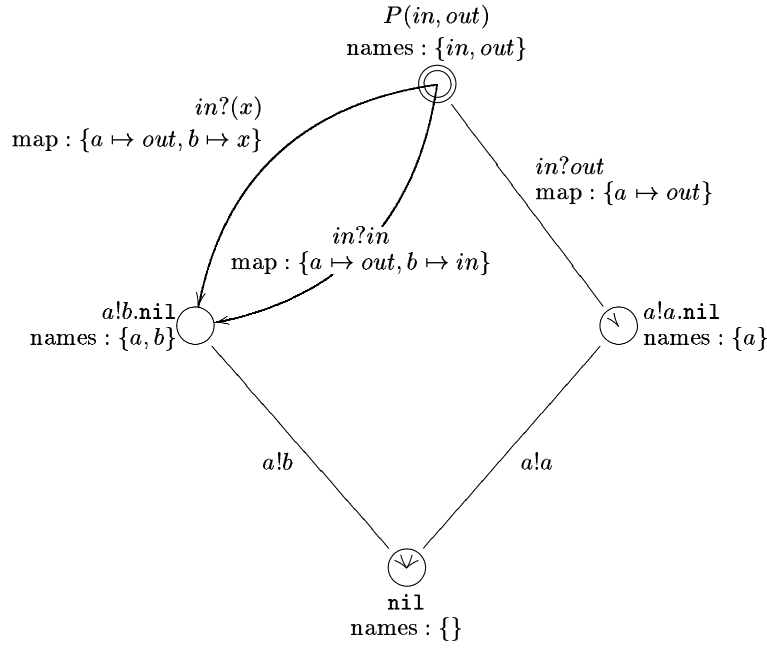

Fig. 1. The HD-automaton corresponding to the agent $P(in, out) := in?(x).out!x.\texttt{nil}$.

In [27] it has been proved that *finite state* HD-automata can be built for the class of *finitary* agents: 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.

Due to the private nature of the names appearing in the states of HD-automata, bisimulations cannot simply be relations on the states; they must also deal with name correspondences: a HD-bisimulation is a set of triples of the form $\langle q_1, \delta, q_2 \rangle$ where q_1 and q_2 are states of the HD-automata and δ is a partial bijection between the names of the states. The bijection is partial since we allow states with different numbers of names to be equivalent (in general equivalent π-calculus agents can have different sets of free names).

Suppose that we want to check if states q_1 and q_2 are (strongly) bisimilar via the partial bijection δ. Furthermore, suppose that q_1 can perform a transition $t_1 : q_1 \xrightarrow[\ell,\sigma]{\lambda} q_1'$. Then we have to find a transition $t_2 : q_2 \xrightarrow[\ell',\sigma']{\lambda} q_2'$ that matches t_1, i.e., not only the two transitions must have the same label, but also the names associated to the labels must be used consistently. In [27, 29] it has been showed that the definition of HD-bisimilarity applied to HD-automata obtained from π-calculus agents induces over π agents an equivalence relation which coincides with strong early bisimilarity.

3.2 From HD-automata to ordinary automata

The theory of HD-automata ensures that HD-automata provides a finite state faithful semantical representation of the behaviour of π-calculus agents. Indeed, it is possible to extract from the HD-automaton of a π-calculus agent its early operational semantics. This is done by a simple algorithm (basically a visit of the HD-automaton) which maintains the global meaning of the local names of the reached states. Intuitively, the algorithm behaves as follows When a fresh name is introduced by a transition of the HD-automaton, a global 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. Furthermore, assume that the global meaning of local names is the identity function (i.e. the function mapping 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 *out* and v, respectively. It is immediate to see that this corresponds to the early transition $P(in, out) \xrightarrow{in?v} out!v.\mathtt{nil}$. Clearly, we have a transition for all the possible choices of the fresh name v. In other words, this procedure yields an infinite state automaton. To obtain a finite state automaton it suffices to take as fresh name 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.

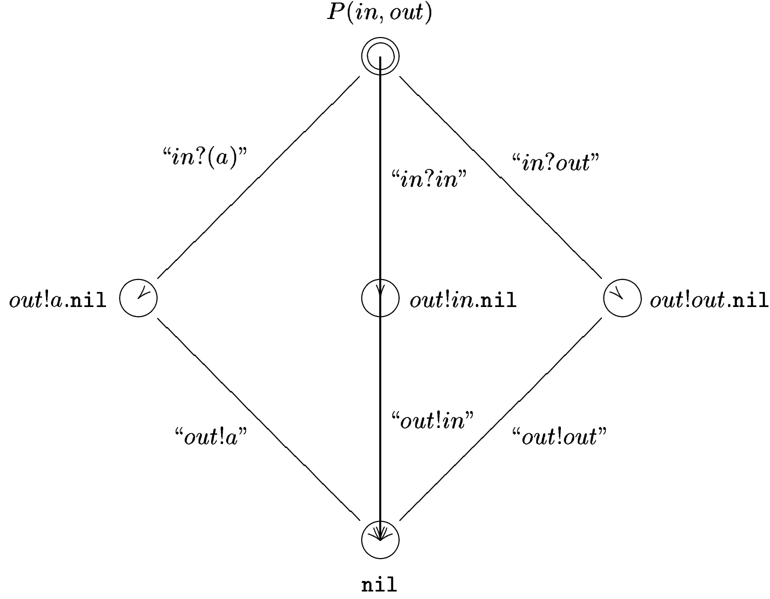

Fig. 2. The ordinary automaton corresponding to the HD-automaton of Figure 1

To sum up, we outlined a 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 irredundant 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 [27] 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.

To conclude this section we show the expressiveness of HD-automata in handling bisimilarity. Consider the π-calculus agent $Q(in, out) = (z)(in?(x).z!x.nil \parallel z?(y).out!y.nil)$. The standard π-calculus early operational semantics yields an infinite state and infinite branching labelled transtion system (see Figure 3 (A)). The ordinary automaton, instead, which results from the HD translation steps is displayed in Figure 3(B). It is straightforward to notice that agent $Q(in, out)$ is weakly bisimilar to agent $P(in, out)$ of Example 1.

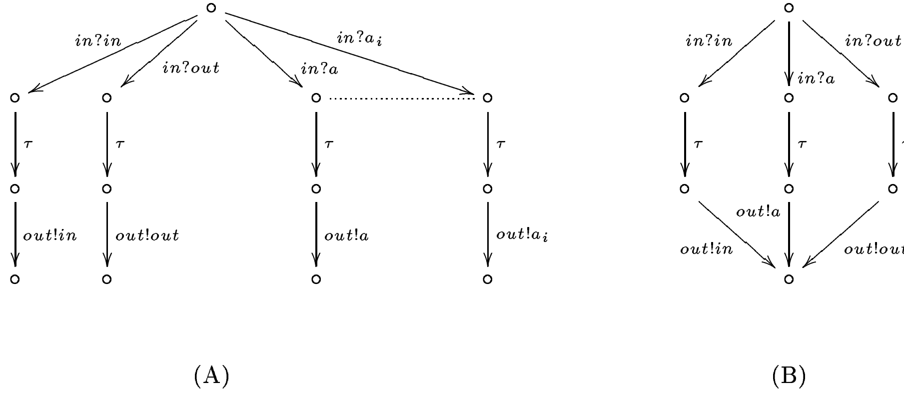

Fig. 3. State space representation of agent $Q(in, out) = (z)(in?(x).z!x.nil \parallel z?(y).out!y.nil)$

4 From π-logic to ACTL

Our purpose was to define an automatic verification procedure to model 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 [19]. Among these, we have chosen the ACTL logic [9], for which an efficient model checker has been implemented [14] and for which a sound translation exists.

In the rest of this section we present the translation function that associates an ACTL formula with a formula of π-logic. The translation is defined by having in mind a precise soundness result: we want a π-logic formula to be satisfied by a π-calculus agent P if and only if the finite state ordinary automaton associated with P satisfies the corresponding ACTL formula. The translation of a formula is thus not unique, but depends on the agent P. Specifically, it depends on the set S of the fresh names of the ordinary automaton associated with the agent P. Correctness properties (as well as other foundational results) of the translation are discussed in [18].

Definition 5. *Let $\theta = \{\alpha/y\}$. We define $\mu\theta$ as being the action μ' obtained from μ by replacing the occurrences of the name y with the name α. Moreover, we define $true\theta = true$, $(\phi_1 \& \phi_2)\theta = \phi_1\theta \& \phi_2\theta$, $(\sim \phi)\theta = \sim \phi\theta$, $(EX\{\mu\}\phi)\theta = EX\{\mu\theta\}\phi\theta$, $(< \mu > \phi)\theta = < \mu\theta > \phi\theta$ and $(EF\phi)\theta = EF\phi\theta$.*

Definition 6 (Translation function). *Given a π-logic formula ϕ and a set of names S, the* ACTL *translation of ϕ is the* ACTL *formula $\mathcal{T}_S(\phi)$ defined as follows:*

- $\mathcal{T}_S(true) = true$
- $\mathcal{T}_S(\phi_1 \& \phi_2) = \mathcal{T}_S(\phi_1) \& \mathcal{T}_S(\phi_2)$
- $\mathcal{T}_S(\sim \phi) = \sim \mathcal{T}_S(\phi)$
- $\mathcal{T}_S(EX\{\mathtt{tau}\}\phi) = EX\{\mathtt{tau}\}\mathcal{T}_S(\phi)$
- $\mathcal{T}_S(EX\{x!y\}\phi) = EX\{x!y\}\mathcal{T}_S(\phi)$
- $\mathcal{T}_S(EX\{x!(y)\}\phi) = \bigvee_{\alpha \in S} EX\{x!(\alpha)\}\mathcal{T}_S(\phi\theta)$, *where* $\theta = \{\alpha/y\}$
- $\mathcal{T}_S(EX\{x?y\}\phi) = EX\{x?y\}\mathcal{T}_S(\phi) \vee \bigvee_{\alpha \in S} EX\{x?(\alpha)\}\mathcal{T}_S(\phi\theta)$, *where* $\theta = \{\alpha/y\}$
- $\mathcal{T}_S(< \mathtt{tau} > \phi) = < \mathtt{tau} > \mathcal{T}_S(\phi)$
- $\mathcal{T}_S(< x!y > \phi) = < x!y > \mathcal{T}_S(\phi)$
- $\mathcal{T}_S(< x!(y) > \phi) = \bigvee_{\alpha \in S} < x!(\alpha) > \mathcal{T}_S(\phi\theta)$, *where* $\theta = \{\alpha/y\}$
- $\mathcal{T}_S(< x?y > \phi) = < x?y > \mathcal{T}_S(\phi) \vee \bigvee_{\alpha \in S} < x?(\alpha) > \mathcal{T}_S(\phi\theta)$, *where* $\theta = \{\alpha/y\}$
- $\mathcal{T}_S(EF\phi) = EF\mathcal{T}_S(\phi)$

In the above definition we have assumed that when $S = \emptyset$ then $\bigvee_{\alpha \in S}\phi = false$.

Note that the complexity of the translation has a worst case complexity which is exponential in the number of names appearing in set S.
This translation guarantees that if P is a π-calculus agent, with associated an ordinary automaton $\mathcal{A}$ and ϕ is a π-logic formula, then we have that $P \models \phi$ if and only if $\Downarrow P \models \mathcal{T}_S(\phi)$, where S is the set of fresh names of $\mathcal{A}$ [18].

Example 2. Let us consider agent $P(in, out)$ introduced in Example 1. Agent P satisfies the π-logic formula $\phi = EX\{in?u\}EX\{out!u\}true$ for each name u, since $P \xrightarrow{in?u}$ for each name u and then it performs an *out*! action with the corresponding name. We want to verify whether the ACTL translation of the formula holds in the ordinary automaton associated with P, hence we have to consider the ACTL translation of the formula with respect to the set of fresh names S used in the ordinary automaton of P, that is $\{a\}$. The translation of the formula is:

$$EX\{in?(a)\}EX\{out!a\}true$$

Note that the resulting ACTL formula holds in the ordinary automaton of P.

5 Model checking π-calculus agents

The translation procedures described in the previous Sections have been implemented on top of the JACK environment [1]. JACK is an environment based on the use of process algebras, automata and temporal logic formalisms, which supports many phases of the system development process.

The idea behind the JACK environment[3] was to combine different specification and verification tools [21, 35, 2, 14], around a common format for representing ordinary automata: the FC2 file format [3]. FC2 makes it possible to exchange automata between JACK tools. Moreover, tools can easily be added to the JACK system, thus extending its potential. Figure 4 illustrates the top-level interface of the JACK environment.

An ordinary automaton is represented in the FC2 format by means of a set of tables that keep the information about state names, arc labels, and transition relations between states.

The editing tools integrated in JACK allow specifications be described both in textual form and in graphical form, by drawing automata. Moreover, the tools provide sophisticated graphical procedures for the description of specifications as networks of processes. This supports hierarchical specification development.

Once the specification of a system has been written, JACK permits the construction of the global automaton corresponding to the behaviour of the overall system. Moreover, automata can be minimized with respect to various (bisimulation) equivalences. ACTL can be used to describe temporal properties and *model checking* can be performed to check whether systems (i.e. their models) satisfy the properties.

Besides facilities offered by the JACK environment for the specification and verification of concurrent systems it provides also facilities for the specification and verification of mobile systems. In particular it is possible to construct the HD-automata of π-calculus agents, and to map HD-automata into ordinary automata represented in the FC2 format through the functionalities offered by the HAL tool. HAL includes modules which implement decision procedures to calculate behavioural equivalences, and modules which support the model checking of properties expressed as formulae of the π-logic. The HAL architecture is displayed in figure 5;

By exploiting the HAL facilities π-calculus agents are translated into ordinary automata. Hence, the JACK bisimulation checker can be 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 FC2tools tools. HAL also supports verification of logical formulae expressing desired properties of the behaviour of π-calculus agents. The ACTL model checker AMC available in JACK can be used for verifying properties of π-calculus agents, after that the π-logic formulae expressing the properties have been translated into ACTL formulae.

The complexity of the model checking algorithm is due to the construction of the state space of the π-calculus agent to be verified, which is, in the worst case, exponential in the syntactical size of the agent.

The current implementation of HAL consists of five main modules all integrated inside the JACK environment. Three of these modules handle the translations from π-calculus agents to HD-automata, from HD-automata to ordinary

[3] Detailed information about JACK are available at:
`http://ghost.iei.pi.cnr.it/projects/JACK`.

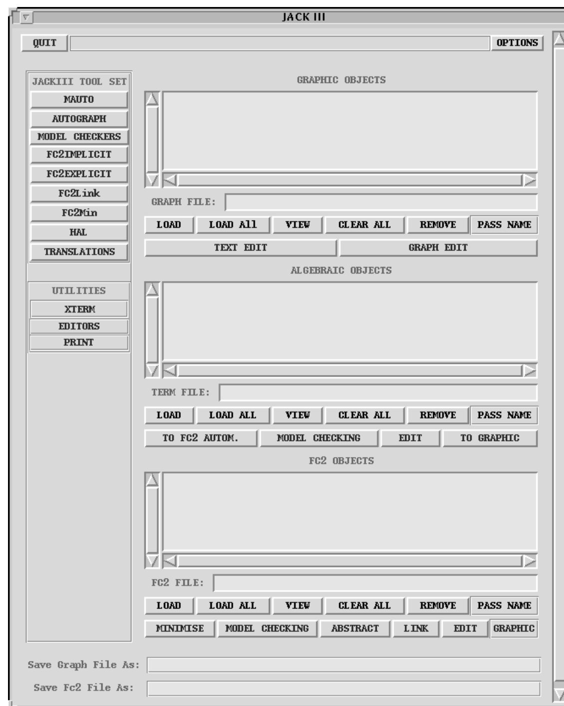

Fig. 4. JACK Main Menu

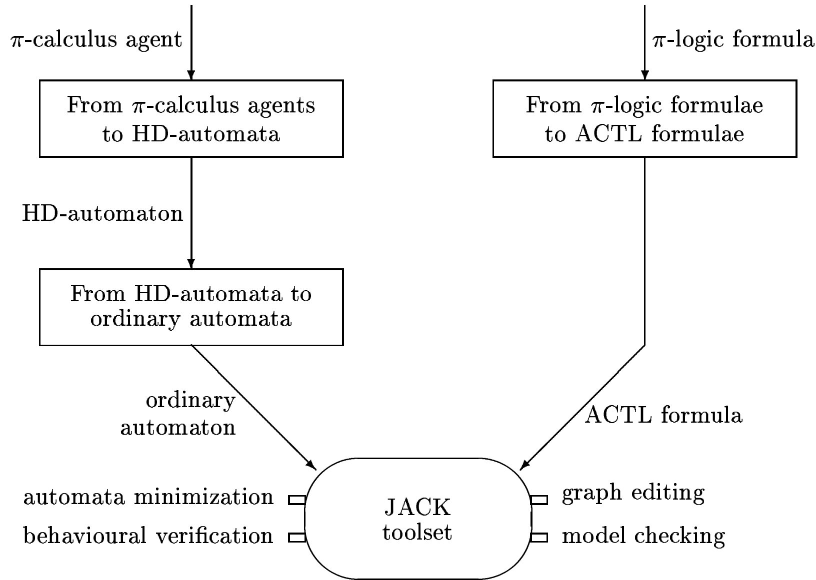

Fig. 5. The logical architecture of the HAL environment.

automata, and from π-logic formulae to ACTL formulae. The fourth module provides several routines that manipulate the internal representation of HD-automata. The last module provides HAL with a user-friendly Graphical User Interface (GUI). The HAL user interface is splitted in two sides: the Agent side and the Logical side. The Agent side allows a π-calculus agent to be transformed into a HD-automaton and then into an ordinary automaton (options **Build** and **Unfold**). The Logic side allows a π-logic formula to be translated in the corresponding ACTL formula taking into account the particular ordinary automaton on which then it will be checked.

Several optimizations have been 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. An example of optimization is given by the reduction of `tau` chains (that are unbranched sequences of `tau` transitions) to simple `tau` transitions (option **Reduce**).

Another optimization 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 stationary communication topologies, namely communication topologies which cannot be modified when computations progress). Since constant names are not consider as possible input values, the branching structure of input transitions is reduced. The semantic handling of constants is presented in [29]. Constants must be declared at the beginning of π-calculus agent specification.

HAL is written in C++ and compiles with the GNU C++ compiler (the GUI is written in Tcl/Tk). It is currently running on SUN stations (under SUN-OS) and on PC stations (under Linux).

6 Verification Case Study

In this section we illustrate the Model Checking facilities of HAL by presenting a simple but expressive case study.

6.1 The Handover Protocol for Mobile Telephones

The case study concerns 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 [37], which has been in turn derived from that in [30]. The specification consists of four modules:

- a MobileStation (Car) mounted in a car moving through two different geographical areas (cells), that provides services to an end user;
- a Mobile Switching Centre (Centre) that is the controller of the radio communications within the whole area composed by the two cells;
- the Base Station modules (Base and IdleBase) 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 GSM is reported below.

```
define Car(talk,switch,out) =
    talk?(msg).out!msg.Car(talk,switch,out) +
    switch?(t).switch?(s).Car(t,s,out)

define Base(talkcentre,talkcar,give,switch,alert) =
    talkcentre?(msg).talkcar!msg.
             Base(talkcentre,talkcar,give,switch,alert)
    +
    give?(t).give?(s).switch!t.switch!s.give!give.
             IdleBase(talkcentre,talkcar,give,switch,alert)

define IdleBase(talkcentre,talkcar,give,switch,alert) =
    alert?(empty).Base(talkcentre,talkcar,give,switch,alert)

define Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap) =
    in?(msg).tca!msg.Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap)
    +
    tau.ga!tp.ga!sp.ga?(empty).ap!ap.Centre(in,tcp,tp,gp,sp,ap,tca,ta,ga,sa,aa)

define GSM(in,out) =
    (tca)(ta)(ga)(sa)(aa)(tcp)(tp)(gp)(sp)(ap)
    |(Car(ta,sa,out),
      Base(tca,ta,ga,sa,aa),
      IdleBase(tcp,tp,gp,sp,ap),
      Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap))
```

There are two kinds of correctness checking that can be performed by exploiting HAL facilities. One is the checking that the specification of the the Handover Protocol is bisimilar to a more abstract service specification, that models the intended behaviour of the system. The other is the (model) checking of some interesting properties, expressed as π-logic formulae.

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

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

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

define GSMbuffer(in,out) = S0(in,out)
```

We expect that both the specifications above satisfy 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

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

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.

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

We also expect that the formulae:

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

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 activities (performed on a Sun Workstation Ultra 1) is presented below.

Figure 6 illustrates the generation of the HD-automata for both the specifications, i.e. GSM and $GSMbuffer$.

Figure 7 shows the construction of the ordinary automata (represented in the FC2 format) associated to the specifications. Then GSM and $GSMbuffer$ have then been shown to be equivalent using the bisimulation checking facilities offered by HAL through the use of FC2tools.

Figure 8 illustrates the translation of the high level property into the low level property expressed as a ACTL formula. Notice that the translation mapping is driven by automata representation.

Figure 9 and Figure 10 illustrates the model checking of the ACTL formulae, obtained in the previous step, by exploiting the AMC model checker.

In the following tables we summarize the figures (states, transitions and times) of the different steps of a typical session of verification for the handover protocol (we report in the table the invocation of the functionalities of the main

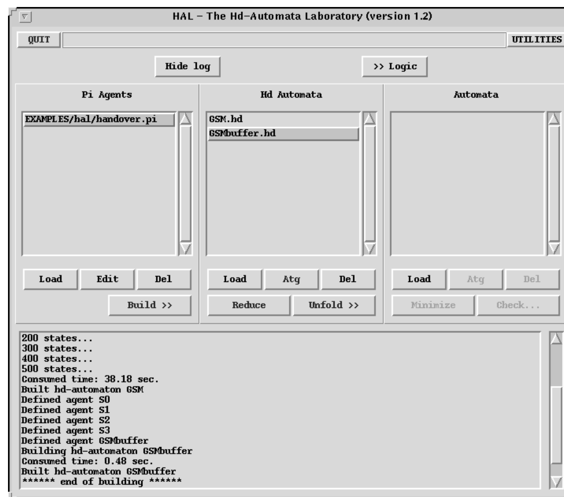

Fig. 6. HAL Build Menu

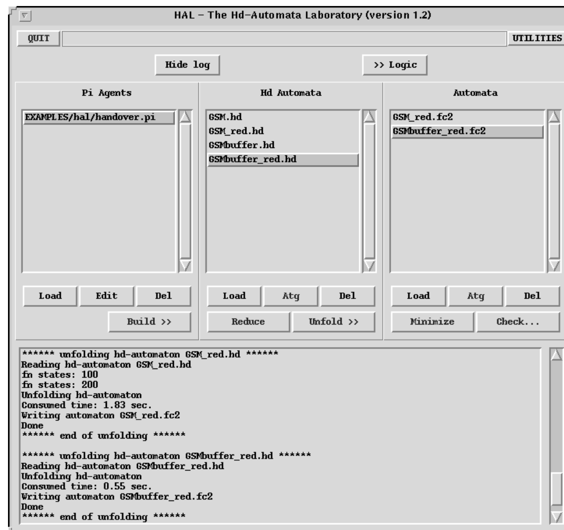

Fig. 7. HAL Unfold Menu

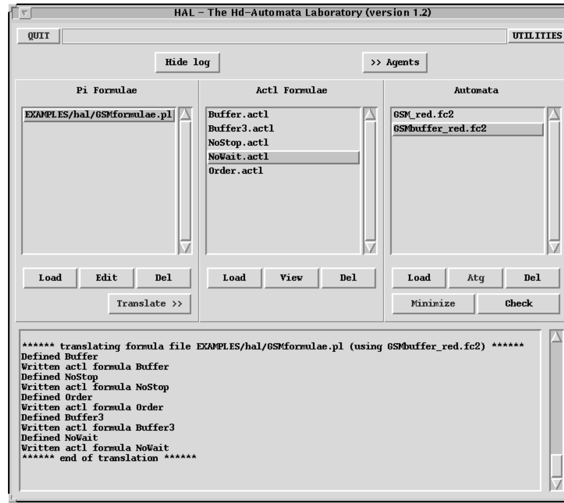

Fig. 8. HAL Translate Menu

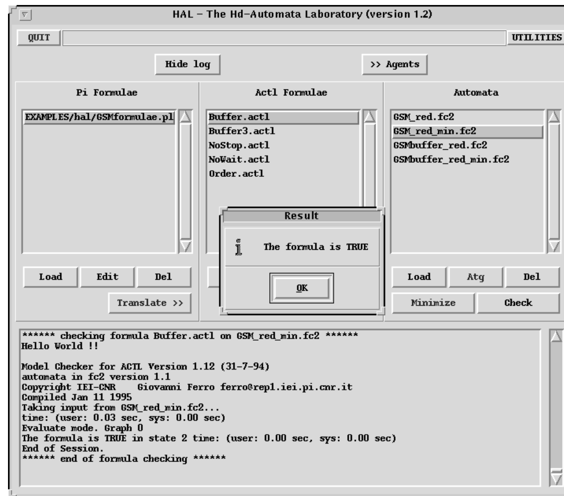

Fig. 9. HAL Model Checking

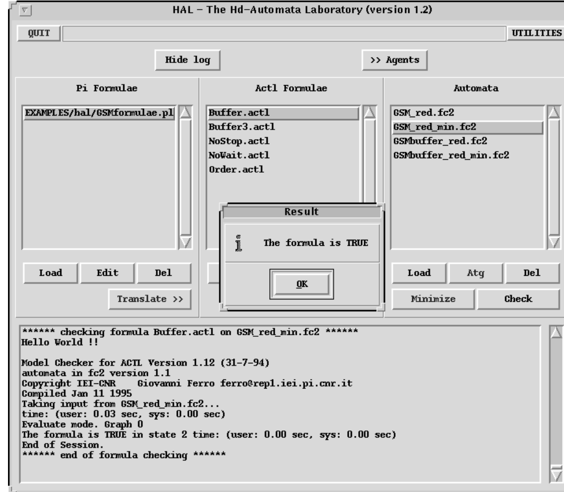

Fig. 10. HAL Model Checcking

HAL modules). We also describes the number of states and transitions of the automata that are built in the different steps.

We have considered first the original specifications of GSM and $GSMbuffer$ as given previously (see Table 2 and Table 3). We then have added to the specification *in* and *out* as constants obtaining hence better figures (see Table 4 and Table 5).

Table 2. Performance issue (1)

command	states	transitions	time
hdaut `:=` `buildHD` *GSM.pi*	506	745	37.52 sec.
hdaut-red `:=` `reduceHD-red` *GSM.hd*	245	484	1.19 sec.
aut `:=` `buildFC2` *hdaut-red*	545	1062	1.54 sec.
min-aut `:=` `minimize` *aut*	49	91	3.45 sec.
`verify` *no-loss-of-messages* `on` *min-aut*	—	—	6 sec.

Finally, we report the figures (see Table 6 and Table 7) also in the case of the original π-calculus specification of handover protocol for the GSM Public Land Mobile Network as given in [30]. Notice that in this case, we were not able to build the HD-automaton without constant declarations (due to state explosion).

Table 3. Performance issue (2)

command	states	transitions	time
hdaut `:=` `buildHD` *GSMbuffer.pi*	65	131	0.46 sec.
hdaut-red `:=` `reduceHD-red` *GSMbuffer.hd*	65	131	0.13 sec.
aut `:=` `buildFC2` *hdaut-red*	164	320	0.54 sec.
min-aut `:=` `minimize` *aut*	49	91	1.88 sec.
`verify` *no-loss-of-messages* `on` *min-aut*	—	—	6 sec.

Table 4. Performance issue (3)

command	states	transitions	time
hdaut `:=` `buildHD` *GSM-const.pi*	124	172	9.43 sec.
hdaut-red `:=` `reduceHD-red` *GSM-const.hd*	52	100	0.3 sec.
aut `:=` `buildFC2` *hdaut-red*	188	358	0.59 sec.
min-aut `:=` `minimize` *aut*	49	91	2.5 sec.
`verify` *no-loss-of-messages* `on` *min-aut*	—	—	6 sec.

Table 5. Performance issue(4)

command	states	transitions	time
hdaut `:=` `buildHD` *GSMbuffer-const.pi*	12	23	0.1 sec.
hdaut-red `:=` `reduceHD-red` *GSMbuffer-const.hd*	12	23	0.03 sec.
aut `:=` `buildFC2` *hdaut-red*	49	92	0.18 sec.
min-aut `:=` `minimize` *aut*	49	91	0.57 sec.
`verify` *no-loss-of-messages* `on` *min-aut*	—	—	6 sec.

Table 6. Performance issue

command	states	transitions	time
hdaut `:=` `buildHD` *handover.pi*	state explosion	—	sec.
hdaut-red `:=` `reduceHD-red` *handover.hd*	—	—	sec.
aut `:=` `buildFC2` *hdaut-red*	—	—	sec.
min-aut `:=` `minimize` *aut*	—	—	sec.
`verify` *no-loss-of-messages* `on` *min-aut*	—	—	sec.

Table 7. Performance issue

command	states	transitions	time
hdaut `:=` `buildHD` *handover-const.pi*	37199	47958	4473 sec.
hdaut-red `:=` `reduceHD-red` *handover-const.hd*	11015	21774	81.44 sec.
aut `:=` `buildFC2` *hdaut-red*	32263	62990	101.71 sec.
min-aut `:=` `minimize` *aut*	49	91	10 sec.
`verify` *no-loss-of-messages* `on` *min-aut*	—	—	6 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.

Our current work on the HAL environment is proceeding on two fronts. On the one hand, we are extending the environment with new modules. In particular, we are developing a *minimization* module. Minimal automata play a central role in the theory and the practice of finite state verification. The theory guarantees that the minimal automaton is indistinguishable from the original one with respect to many behavioral properties. Moreover, the problem of deciding observational equivalences is reduced to the problem of computing the minimal automaton. Foundational results on the basic model [28, 12] are driving the design and the implementation of a module able to directly minimize HD-automata (without mapping them into ordinary automata). On the other hand, we are working on experimenting the environment to deal with specification and verification of WAN applications.

In particular, we are adddressing the problem of verifyng properties of security protocols. The creation of nonces to identify sessions in security protocols is an instance of the idea of having dynamic name generation. In the last years, several techniques for finding flaws of security protocols have been developed (we refer to [22] for a critical review of the state-of-the-art on verification of security protocols). All of these techniques are basically based on the analysis of finite

state systems, and typically can ensure error freedom only for a finite amount of the behaviour of protocols. Even if many protocols do not include iterations, an unbound number of principals may take part into the interleaved sessions of the protocol; moreover, many known attacks exploit the mixed interleaving of different sessions of the same protocol [26]. In general, it is not possible to deduce the safety of the protocol from the safety of a bounded number of interleaved sessions.

Another issue we are approaching consists of the development of a design methodology for composition of software components based on the notion *programmmable coordination via naming.* Coordination is a key concept for modeling and designing WAN applications. Coordinators are the basic mechanisms to adapt components to the network environment changes. For instance, coordinators are in charge of supporting and monitoring the execution of dynamically loaded modules. The main idea of our approach is to identify all the principals (e.g. authorities, software components) involved within a coordination activity by naming schemes. A programmable coordination policy is modeled as a structured operation acting over names. For instance, the coordination policy which redirects bindings to resources of mobile agents roaming the net is naturally described by a structured action over names (bindings) of resources. Similarly, the coordination policy which detects the accessible resources (namely the current execution environment) of incoming mobile agents fits in this framework.

To end the paper we make a more detailed comparison with related works. The *Mobility Workbench* [37] (MWB in short). In the MWB the verification of bisimulation equivalence between (finite control) π-calculus agents is made *on the fly* [13], 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 [6, 7] 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, although expressive enough to describe interesting safety and liveness properties of π-calculus agents, is less expressive than the Propositional μ-calculus with name-passing used in the MWB.

References

1. A. Bouali , S. Gnesi, S. Larosa. The integration Project for the JACK Environment. *Bulletin of the EATCS*, 54, October 1994, pp. 207–223.
2. A. Bouali and R. de Simone. Symbolic bisimulation minimization. In *Proc. CAV'92*, LNCS 663. Springer-Verlag, 1992.

3. A. Bouali, A. Ressouche, V. Roy and R. De Simone. The FC2Tools set. In *Proc. CAV'96*, LNCS 1102. Springer Verlag, 1996.
4. E. Clarke and J. Wing Eds. Formal Methods: State of the Art and Future Directions. Strategic Directions in Comp. Res. Formal Methods WG Rep. *ACM Comp. Surv.*, December 1996.
5. Clarke, E.M., Emerson, E.A., Sistla, A.P., "Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specification," ACM Transaction on Programming Languages and Systems, 8(2), April 1986, pp. 244–263.
f
6. M. Dam. Model checking mobile processes. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
7. M. Dam. Proof systems for π-Calculus Logics. To appear on *Logics for Concurrency and Synchronization* Studies in Logics and Computation, Oxford University Press, 2001.
8. R. De Nicola, A. Fantechi, S. Gnesi and G. Ristori, An action-based framework for verifying logical and behavioural properties of concurrent systems, *Computer Networks and ISDN Systems* **25** 7 (North-Holland, 1993) 761-778.
9. 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.
10. Emerson, E.A., Halpern, J.Y., "Sometimes and Not Never Revisited: on Branching Time versus Linear Time Temporal Logic," Journal of ACM, 33 (1), January 1986, pp. 151–178.
11. G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, G. Ristori, Verifying Mobile Processes in the HAL Environment. *Computer Aided Verification* (CAV'98), LNCS 1427, 1998.
12. G. Ferrari, U. Montanari, M. Pistore. Minimizing Transition Systems for Name Passing Calculi: A Co-algebraic Formulation Technical Report, Dipartimento di Informatica, Università di Pisa, 2001.
13. J.-C. Fernandez and L. Mounier. "On the fly" verification of behavioral equivalences and preorders. In *Proc. CAV'91*, LNCS 575. Springer Verlag, 1991.
14. G. Ferro. *AMC: ACTL Model Checker. Reference Manual.* IEI-Internal Report, B4-47, 1994.
15. A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. *Transactions on Software Engineering*, 24(5):342-361, 1998.
16. M.P. Fiori, G.D. Plotkin and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1999.
17. M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1999.
18. S. Gnesi, G. Ristori. *A Model Checking Algorithm for π-calculus agents*. In *Advances in Temporal Logic*. Kluwer Academic Publishers, pp.339 - 357, 2000.
19. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM* 32 (1), pp. 137-161, 1985.
20. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science* 27, pp. 333-354, 1983.
21. E. Madelaine and D. Vergamini. AUTO: A verification tool for distributed systems using reduction of finite automata networks. Formal Description Techniques II, 1990.

22. C. Meadows Open Issues in Formal Methods for Cryptographic Protocol Analysis, Proceedings of DISCEX 2000, IEEE Computer Society Press, 237-250, January, 2000.

23. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

24. R. Milner, J. Parrow and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

25. R. Milner, J. Parrow and D. Walker. Modal logic for mobile processes. In *Theoretical Computer Science* 114:149-171, 1993.

26. Jonathan K. Millen. A necessarily parallel attack. In Nevin Heintze and Edmund Clarke, editors, *Workshop on Formal Methods and Security Protocols, Part of the Federated Logic Conference*, Trento, Italy, 1999.

27. U. Montanari and M. Pistore. Checking bisimilarity for finitary π-calculus. In *Proc. CONCUR'95*, LNCS 962. Springer Verlag, 1995.

28. U. Montanari and M. Pistore. π-calculus, structured coalgebras and minimal HD-automata. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science 2000*, volume 1893 of *Lecture Notes in Computer Science*. Springer, 2000.

29. M. Pistore. *History Dependent Automata.* PhD. Thesis TD-5/99, Università di Pisa, Dipartimento di Informatica, 1999.

30. F. Orava and J. Parrow. An Algebraic Verification of a Mobile Network" *Formal Aspects of Computing* 4, pp. 497-543.

31. D. Park Concurrency and automata on infinite sequences. Lecture Notes in Computer Science, 104, 1981

32. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

33. P. Quaglia. *The π-calculus with explicit substitutions.* PhD thesis TD-09/96. Dipartimanto di Informatica, Università di Pisa, 1996.

34. G. Catalin-Roman, G. Picco, A. Murphy. Software Engineering for mobility. *The Future of Software Engineering* (A. Filkelstein Ed), pages 241-258, ACM Press, 2000.

35. V. Roy and R. De Simone. AUTO and autograph. In *Proc. CAV'90*, LNCS 531. Springer Verlag, 1990.

36. D. Sangiorgi. A theory of bisimulation for the π-calculus. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.

37. B. Victor and F. Moller. The Mobility Workbench — A tool for the π-calculus. In *Proc. CAV'94*, LNCS 818. Springer Verlag, 1994.

Appendix: ACTL: Action Based CTL

ACTL [?] is a branching time temporal logic suitable to express properties of reactive systems whose behaviour is characterized by the actions they perform. Indeed, ACTL embeds the idea of "evolution in time by actions" and logical formulae take their meaning on labelled transition systems. ACTL can be used to define both *liveness* (something good eventually happens) and *safety* (nothing bad can happen) properties of concurrent systems.

Definition 7 (Action formulae). *Given a set of observable actions Act, the language $\mathcal{AF}(Act)$ of the action formulae on Act is defined as follows:*

$$\chi ::= true \mid b \mid \neg\chi \mid \chi \,\&\, \chi$$

where b ranges over Act.
ACTL is a branching time temporal logic of state formulae (denoted by ϕ), in which a path quantifier prefixes an arbitrary path formula (denoted by π).

Definition 8 (ACTL syntax). *The syntax of the ACTL formulae is given by the grammar below:*
$\phi ::= true \mid \phi \,\&\, \phi \mid \sim \phi \mid E\pi \mid A\pi$
$\pi ::= X\{\chi\}\phi \mid X\{\mathtt{tau}\}\phi \mid [\phi\{\chi\}U\phi] \mid [\phi\{\chi\}U\{\chi'\}\phi]$
where χ, χ' range over action formulae, E and A are path quantifiers, and X and U are the next and the until operators respectively.

In order to present the ACTL semantics, we need to introduce the notion of paths over an ordinary automaton.

Definition 9 (Paths). *Let $\mathcal{A} = (Q, q_0, Act \cup \{\mathtt{tau}\}, R)$ be an ordinary automaton.*

- *σ is a path from $r_0 \in Q$ if either $\sigma = r_0$ (the empty path from r_0) or σ is a (possibly infinite) sequence $(r_0, \alpha_1, r_1)(r_1, \alpha_2, r_2)\ldots$ such that $(r_i, \alpha_{i+1}, r_{i+1}) \in R$.*
- *The concatenation of paths is denoted by juxtaposition. The concatenation $\sigma_1\sigma_2$ is a partial operation: it is defined only if σ_1 is finite and its last state coincides with the initial state of σ_2. The concatenation of paths is associative and has identities. Actually, $\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3$, and if r_0 is the first state of σ and r_n is its last state, then we have $r_0\sigma = \sigma r_n = \sigma$.*
- *A path σ is called maximal if either it is infinite or it is finite and its last state has no successor states. The set of the maximal paths from r_0 will be denoted by $\Pi(r_0)$.*
- *If σ is infinite, then $|\sigma| = \omega$.*
  If $\sigma = r_0$, then $|\sigma| = 0$.
  If $\sigma = (r_0, \alpha_1, r_1)(r_1, \alpha_2, r_2)\ldots(r_n, \alpha_{n+1}, r_{n+1})$, $n \geq 0$, then $|\sigma| = n + 1$. Moreover, we will denote the ith state in the sequence, i.e. r_i, by $\sigma(i)$.

Definition 10 (Action formulae semantics). *The satisfaction relation* $\models$ *for action formulae is defined as follows:*

$a \models true$	*always*
$a \models b$	*iff* $a = b$
$a \models \sim \chi$	*iff not* $a \models \chi$
$a \models \chi \,\&\, \chi'$	*iff* $a \models \chi$ *and* $a \models \chi'$

As usual, *false* abbreviates $\sim$ *true* and $\chi \vee \chi'$ abbreviates $\sim (\sim \chi \,\&\!\sim \chi')$.

Definition 11 (ACTL semantics). *Let* $\mathcal{A} = (Q, q_0, Act \cup \{\mathtt{tau}\}, R)$ *be an ordinary automaton. Let* $s \in Q$ *and* σ *be a path. The satisfaction relation for* ACTL *formulae is defined in the following way:*

- $s \models true$ *always*
- $s \models \phi \,\&\, \phi'$ *iff* $s \models \phi$ *and* $s \models \phi'$
- $s \models \sim \phi$ *iff not* $s \models \phi$
- $s \models E\pi$ *iff there exists* $\sigma \in \Pi(s)$ *such that* $\sigma \models \pi$
- $s \models A\pi$ *iff for all* $\sigma \in \Pi(s)$, $\sigma \models \pi$
- $\sigma \models X\{\chi\}\phi$ *iff* $\sigma = (\sigma(0), \alpha_1, \sigma(1))\sigma'$, *and* $\alpha_1 \models \chi$, *and* $\sigma(1) \models \phi$
- $\sigma \models X\{\mathtt{tau}\}\phi$ *iff* $\sigma = (\sigma(0), \mathtt{tau}, \sigma(1))\sigma'$, *and* $\sigma(1) \models \phi$
- $\sigma \models [\phi\{\chi\}U\phi']$ *iff there exists* $i \geq 0$ *such that* $\sigma(i) \models \phi'$, *and for all* $0 \leq j < i$: $\sigma = \sigma'(\sigma(j), \alpha_{j+1}, \sigma(j+1))\sigma''$ *implies* $\sigma(j) \models \phi$, *and* $\alpha_{j+1} = \mathtt{tau}$ *or* $\alpha_{j+1} \models \chi$
- $\sigma \models [\phi\{\chi\}U\{\chi'\}\phi']$ *iff there exists* $i \geq 1$ *such that* $\sigma = \sigma'(\sigma(i-1), \alpha_i, \sigma(i))\sigma''$, *and* $\sigma(i) \models \phi'$, *and* $\sigma(i-1) \models \phi$, *and* $\alpha_i \models \chi'$, *and for all* $0 < j < i$: $\sigma = \sigma'_j(\sigma(j-1), \alpha_j, \sigma(j))\sigma''_j$ *implies* $\sigma(j-1) \models \phi$ *and* $\alpha_j = \mathtt{tau}$ *or* $\alpha_j \models \chi$

As usual, *false* abbreviates $\sim$ *true* and $\phi \vee \phi'$ abbreviates $\sim (\sim \phi \,\&\!\sim \phi')$. Moreover, we define the following derived operators:

- $EF\phi$ stands for $E[true\{true\}U\phi]$.
- $AG\phi$ stands for $\sim EF \sim \phi$.
- $< a > \phi$ stands for $E[true\{false\}U\{a\}\phi]$.
- $< \mathtt{tau} > \phi$ stands for $E[true\{false\}U\phi]$.

ACTL logic can be used to define *liveness* (something good eventually happens) and *safety* (nothing bad can happen) properties of concurrent systems. Moreover, ACTL logic is *adequate* with respect to strong bisimulation equivalence on ordinary automata [9]. Adequacy means that two ordinary automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are strongly bisimilar if and only if $F_1 = F_2$, where $F_i = \{\psi \in \text{ACTL} : \mathcal{A}_i \text{ satisfies } \psi\}$, $i = 1, 2$.