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 Scienza e Tecnologie dell'Informazione - ISTI - C.N.R., Pisa
[3] Dipartimento di Informatica e Tlc - Università di Trento

Abstract. This paper presents a semantic-based environment for reasoning about the behaviour of mobile systems. The verification environment, called HAL, exploits a novel automata-like model which allows finite state verification of systems specified in the π-calculus. The HAL system is able to interface with several efficient toolkits (e.g. model checkers) to determine whether or not certain properties hold for a given specification. We report experimental results on some case studies.

1 Introduction

A *global computing* system is defined as a network of stationary and mobile components. The primary features of a global computing system are that its components are autonomous, software versioning is highly dynamic, the network's coverage is variable and often components reside over the nodes of the network (WEB services), membership is dynamic and often ad hoc without a centralized authority. Global computing systems must be made very robust since they are intended to operate in potentially hostile dynamic environments. This means that they are hard to construct correctly and very difficult to test in a controlled way. In this area formal analysis techniques and the corresponding verification technologies are important to gain confidence in correct behaviour and to weed out bugs and security hazards before a system is deployed. For instance, the growing demands on security have led to the development of formal models that allow specification and verification of cryptographic protocols (see [1, 10, 20, 27] to cite a few). Although significant progresses have been made in providing foundational models and effective verification techniques to support formal verification of global computing systems, current software engineering technologies provide limited solutions to some of the issues outlined above. The problem of formal verification of global computing systems still requires considerable research and dissemination efforts.

Automatic methods for verifying finite state concurrent systems have been shown to be surprisingly effective [9]. Indeed, finite state verification techniques

* Work partially supported by FET Projects IST-2001-33100 PROFUNDIS, IST-2001-32747 AGILE and MIUR Project NAPOLI.

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.

Unfortunately, finite state verification of global computing 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 [30]. The π-calculus 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 is the archetype of name passing or nominal process calculi. Name passing process calculi emphasize the principle that name mechanisms (e.g. local name generation, name exchanges, etc.) provide a suitable abstraction to formally explain a wide range of phenomena of global computing systems (see e.g.[41, 22]). The usefulness of names has been also emphasized in practice. For instance, Needham [35] pointed out the role of names for the security of distributed systems. The World Wide Web provides an excellent (perhaps the most important) example of the power of names and name binding/resolution.

Name passing process calculi have 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, standard automata-like models are infinite-state and infinite branching, thus making verification a difficult task.

History Dependent automata (HD-automata in short) have been proposed in [36, 32] as a new effective model for name passing calculi. Similarly to ordinary automata, HD-automata are made out of states and labeled transitions; their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt with as syntactic components of labels, but become explicit part of the operational model. This allows one to model explicitly name creation/deallocation, and name extrusion: these are the distinguished mechanisms of name passing calculi.

HD-automata can be abstractly understood as automata over a permutation model, whose ingredients are sets of names and of permutations (renaming substitutions) on these name sets. Names and name permutations have shown to play a fundamental role to describe and reason about formalisms with name-binding operations, and have been incorporated into various kinds of theories that aim at providing syntax-free models of name-passing calculi [19, 21, 25, 33, 34, 37].

HD-automata provide an intermediate, syntax independent, format to represent calculi equipped with mobility and distribution primitives [36]. 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 for 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. Indeed, the distinguished feature of our approach is the reduction of a specific name-based theory to a specific a name-less theory. This allows us to *re-use* both verification principles and automatic methods specifically developed for ordinary finite state automata. We refer to Honda [25] for the description of a general algebraic framework which provides formal mechanisms to establish representation theorems from name-based theories to name-less theories and back.

In this paper we focus on the usage of HD-automata as a theoretical foundations of an automata-based approach to the finite state verification of name passing process calculi. In particular, we exploit this theory as a basis for the design and development of effective and usable verification toolkits. This paper describes our experience in experimenting an environment, called the *HD Automata Laboratory* (HAL), for the finite state verification of systems specified in the π-calculus. 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 specifications presented in [32]. In particular, we exploit a high level logic with modalities indexed by π-calculus actions and we provide a mapping which translates these 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.

To illustrate the effectiveness and usability of our approach, we consider some case studies which allows us to demonstrate some common verification patterns which arise frequently when reasoning about π-calculus specifications.

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 our experiments on some case studies.

2 Background

In this section we present an overview of the main concepts and notations that we will use throughout the paper.

2.1 Ordinary Automata

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

Definition 1 (ordinary automaton). *An ordinary automaton is a 4-tuple* $\mathcal{A} = (Q, q^0, L, R)$*, where:*

- Q *is a finite set of states;*
- q^0 *is the initial state;*
- L *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* [29, 39].

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

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* $q_1' \mathcal{R}\, q_2'$*.*

Relation $\mathcal{R}$ *is a* bisimulation *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 their initial states* q_1^0, q_2^0 *are bisimilar, namely* $q_1^0 \mathcal{R}\, q_2^0$ *for some bisimulation* $\mathcal{R}$*.*

2.2 The π-calculus

Given a denumerable infinite set $\mathcal{N}$ of *names* (denoted by $a, \ldots, z$), the set of π-calculus *agents* over $\mathcal{N}$ are defined by the syntax[1]:

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

$$\alpha ::= \texttt{tau} \;\big|\; x!y \;\big|\; x?(y),$$

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 $\texttt{fn}(P)$ 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 $\texttt{fn}(P_A) \subseteq \{y_1 \ldots y_{r(A)}\}$) and we assume that each identifier in P_A is in the scope of a prefix (guarded recursion).

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

$$\mu ::= \texttt{tau} \;\big|\; x!y \;\big|\; x!(z) \;\big|\; x?y$$

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

The rules for the *early operational semantics* are defined in Table 1. As usual operational rules are defined modulo structural congruence, hence the symmetric versions of rules have been omitted.

[1] For convenience, we adopt the syntax that is used in the HAL framework to input π-calculus specifications. 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.

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 $\dfrac{P_1 \xrightarrow{\mu} P_1'}{P_1 \parallel P_2 \xrightarrow{\mu} P_1' \parallel P_2}$ if $\mathtt{bn}(\mu) \cap \mathtt{fn}(P_2) = \emptyset$	
COM $\dfrac{P_1 \xrightarrow{x!y} P_1' \quad P_2 \xrightarrow{x?y} P_2'}{P_1 \parallel P_2 \xrightarrow{\mathtt{tau}} P_1' \parallel P_2'}$	CLOSE $\dfrac{P_1 \xrightarrow{x!(y)} P_1' \quad P_2 \xrightarrow{x?y} P_2'}{P_1 \parallel P_2 \xrightarrow{\mathtt{tau}} (y)(P_1' \parallel P_2')}$ 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 [40]; they are based on direct comparison of the observable actions π-agents can perform. They can be strong or weak, early [31], late [30] or open [43]. In this paper we consider early bisimilarity since it provides the simplest setting for presenting the basic results of our framework. However, it is possible to treat also other behavioural equivalences and other dialects of the π-calculus (e.g. asynchronous π-calculus) [36].

Definition 3 (early bisimulation). *A binary relation $\mathcal{B}$ over a set of agents is a strong early simulation if, whenever $P \ \mathcal{B}\ Q$, we have that:*

- *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 $P' \ \mathcal{B}\ Q'$.*

Relation $\mathcal{B}$ is a strong early bisimulation if both $\mathcal{B}$ abd $\mathcal{B}^{-1}$ are simulations. Two agents are said strong early bisimilar*, written $P \simeq Q$, if there exists a bisimulation $\mathcal{B}$ such that $P \ \mathcal{B}\ Q$.*

2.3 A temporal logic for π-calculus agents

The standard approach to capturing correctness of π-calculus specification is through the use of a bisimulation equivalence. However, in some cases it could be more useful to check whether crucial properties (such as variety of safety and liveness properties) hold. This raises the obvious question of how the logic behaves with respect to a bisimulation equivalence. Usually, the logic behaves well provided that is *adequate* with respect to the bisimulation equivalence, namely two processes are bisimilar provided that they satisfy exactly the same set of logical formulae.

Several programming logics have been proposed to express and verify properties of π-calculus agents (e.g. [11, 31]). These logical formalisms are extensions, with π-calculus actions, name quantifications and parameterizations, of standard action-based logics [24, 26]. Here, we introduce the logic we use to specify behavioural properties of π-calculus agents. The logic, called π-logic, extends the modal logic introduced in [31] with some expressive modalities. Besides the strong *next* modality $EX\{\mu\}\ \phi$ of [31], the π-logic also includes two *eventually* temporal operators (notation $EF\phi$ and $EF\{\chi\}\phi$) that permit to express liveness and safety properties. The meaning of $EF\ \phi$ is that ϕ must be true sometimes in a possible future, and the meaning of $EF\{\chi\}\ \phi$ is that the truth of ϕ must be preceded by the occurrence of a sequence of actions χ. Derived temporal operators, defined on the top of the next and eventually operators, including a weak *next* modality $<\mu>\phi$, whose meaning is that a number of unobservable `tau` actions can be executed before the action μ[2], and the *always* operators $AG\ \phi$, whose meaning is that ϕ is true now and always in the future, and $AG\{\chi\}\ \phi$, whose meaning is that ϕ is true now and in all future states reachable performing sequences of actions χ.

The syntax of the π-logic is given by:

$$\phi ::= \mathit{true} \mid {\sim}\phi \mid \phi \,\&\, \phi' \mid EX\{\mu\}\phi \mid EF\ \phi \mid EF\{\chi\}\phi$$

where χ could be μ, ${\sim}\mu$ or $\bigvee_{i\in I}\mu_i$ where I is a finite set. We remark that we allow for a richer syntax of actions in $EF\{\chi\}\phi$ since this is useful for expressing constraints on the actions that can appear along the path.

The interpretation of the logic formulae is the following:

- $P \models \mathit{true}$ 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 $P' \models \phi$;
- $P \models EF\ \phi$ if and only if there exist $P_0, \dots, P_n$ and $\mu_1, \dots, \mu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\mu_1} P_1 \dots \xrightarrow{\mu_n} P_n$ and $P_n \models \phi$.
- $P \models EF\{\chi\}\phi$ if and only if there exist $P_0, \dots, P_n$ and $\nu_1, \dots, \nu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\nu_1} P_1 \dots \xrightarrow{\nu_n} P_n$, $P_n \models \phi$ and:
  - $\chi = \mu$: for all $1 \leq j \leq n$, $\nu_j = \mu$ or $\nu_j = \mathtt{tau}$;
  - $\chi = {\sim}\mu$: for all $1 \leq j \leq n$, $\nu_j \neq \mu$ or $\nu_j = \mathtt{tau}$;
  - $\chi = \bigvee_{i\in I}\mu_i$: for all $1 \leq j \leq n$, $\nu_j = \mu_i$ for some $i \in I$ or $\nu_j = \mathtt{tau}$.

The following derived operators can be defined:

- $\phi \vee \phi'$ stands for ${\sim}({\sim}\phi \,\&\, {\sim}\phi')$;
- $AX\{\mu\}\phi$ stands for ${\sim}EX\{\mu\}{\sim}\phi$. This is the dual version of the strong *next* operator;
- $<\mu>\phi$ stands for $EF\{\mathtt{tau}\}EX\{\mu\}\phi$. This is the weak *next* operator.
- $[\mu]\phi$ stands for ${\sim}<\mu>{\sim}\phi$. This is the dual version of the weak *next* operator;
- $AG\ \phi$ stands for ${\sim}EF\ {\sim}\phi$ and $AG\{\chi\}\ \phi$ stands for ${\sim}EF\{\chi\}\ {\sim}\phi$. These are the *always* operators.

[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 strong *next* by EX and the weak *next* by $<\_>$.

Standard results ensure that liveness and safety properties can be naturally expressed by means of π-logic formulae. Moreover, it has been proved [23] that the π-logic is adequate with respect to strong early bisimulation equivalence. This means that two π-calculus agents are early bisimilar provided that they satisfy the same properties that can be expressed in the π-logic.

The π-logic comes equipped with a model checking algorithm to determine whether or not that properties expressed as π-logic formulae hold for a π-calculus specification. The construction of the model checker for the π-logic exploits and re-uses the model checker implemented for the ACTL logic [14, 13]. The branching time temporal logic ACTL is the action based version of CTL [15]. ACTL is well suited to describe the behavior of a system in terms of the actions it performs at its working time. 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 splitted into these two steps to achieve modularity in the structure of the verification environment. Moreover, the intermediate representation allows for a more efficient implementation of the second translation step.

3.1 From π-calculus agents to HD-automata

HD-automata have been introduced in [32], 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 or causality and Petri Nets [36, 33, 34].

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 (HD-automaton). *A* history-dependent automaton (HD-automaton) *is a structure* $\mathcal{A} = (Q, q^0, L, \omega, q \xrightarrow[\sigma]{\lambda} q')$*, where:*

- *Q is a finite set of states;*
- *q^0 is the initial state;*
- *L is a set of action labels;*
- *ω is a function associating (finite sets of local) names to states:* $\omega : Q \longrightarrow \mathcal{P}_f(\mathcal{N})$*;*
- *$q \xrightarrow[\sigma]{\lambda} q'$ is the transition relation where $\sigma : \omega(q') \longrightarrow \omega(q) \cup \{*\}$ is the (injective) embedding function, and $*$ is a distinguished 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. In HD-automata, name creation must be handled explicitly, using $*$, whereas name discarding can occur silently.

As pointed out in [32] 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 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. 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 in the transition of P are *in* and *out* (i.e. the local names of the initial state) and the fresh name $*$. Moreover, labels of the form $in?(*)$ are used to denote the input of a fresh name.

The meaning of the names changes along the transitions (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

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

of Figure 1. The corresponding embedding function $\sigma : \{a, b\} \longrightarrow \{in, out, *\}$ is defined as $\sigma(a) = out, \sigma(b) = *$. 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 differ for an injective substitution only.

In [32] 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.

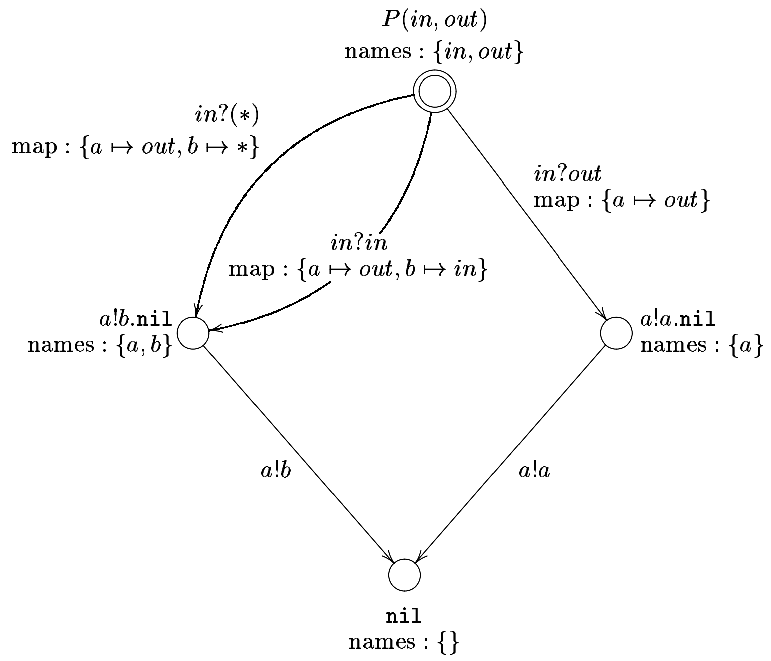


Fig. 1. The HD-automaton corresponding to the agent $P(in, out) := in?(x).out!x.\mathtt{nil}$.

Due to the private nature of the names appearing in the states of HD-automata, bisimulations cannot simply be defined as relations over states but 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 $t_1 : q_1 \xrightarrow[\sigma_1]{\lambda_1} q_1'$. To check bisimilarity, we have to find a transition $t_2 : q_2 \xrightarrow[\sigma_2]{\lambda_2} q_2'$ that matches t_1, i.e., the two transitions must have the same label according to bijection δ and the target states are bisimilar via the partial bijection δ' that is built from δ and from the transition embeddings σ_1 and σ_2.

Definition 5 (HD-bisimulation). *Let A_1 and A_2 be two HD-automata on the same set L of labels. An* HD-simulation *for A_1 and A_2 is a set of triples*

$$\mathcal{R} \subseteq \{\langle q_1, \delta, q_2 \rangle \mid q_1 \in Q_1,\ q_2 \in Q_2,\ \delta : \textit{partial bijection of } w_1(q_1) \textit{ and } w_2(q_2)\}$$

such that, whenever $\langle q_1, \delta, q_2 \rangle \in \mathcal{R}$ we have:

– *for each $t_1 : q_1 \xrightarrow[\sigma_1]{\lambda_1} q_1'$ there is some $t_2 : q_2 \xrightarrow[\sigma_2]{\lambda_2} q_2'$, and:*

- *$\lambda_2 = \delta^*(\lambda_1)$, where δ^* is a partial bijection between $w_1(q_1) \cup \{*\}$ and $w_2(q_2) \cup \{*\}$ such that $\delta^*(x) = \delta(x)$ if $\delta^*(x) \in w_2(q_2)$;*
- *$\langle q_1', \delta', q_2' \rangle \in \mathcal{R}$, where $\delta' = \sigma_2^{-1} \circ \delta^* \circ \sigma_1$.*

Relation $\mathcal{R}$ is an HD-bisimulation *if both $\mathcal{R}$ and $\mathcal{R}^{-1} = \{\langle q_2, \delta^{-1}, q_1 \rangle \mid \langle q_1, \delta, q_2 \rangle \in \mathcal{R}\}$ are HD-simulations.*
Two HD-automata A_1 and A_2 are HD-bisimilar, written $A_1 \sim A_2$ if their initial states are bisimilar according to the partial bijection that is the identity on $w_1(q_1^0) \cap w_2(q_2^0)$.

We can briefly comment on the previous definition. The mapping δ^* allows one to extend δ either by mapping the special symbol $*$ in t_1 into $*$ in t_2, or by mapping into $*$ a name of $w_1(q_1)$ not covered by δ. This second case is necessary since q_1 and q_2 may have different sets of free names, and hence different sets of input transitions. For instance, let us consider the agent $Q(in, out, w) := in?(x).out!x.\texttt{nil}$. This agent has the same behaviour of $P(in, out)$ but contains an extra name w. According to the definition of HD-bisimulation, the transition $Q(in, out, w) \xrightarrow{in?w} out!w.\texttt{nil}$ can be matched by the transition of $P(in, out)$ corresponding to the input of special symbol $*$.

In [32, 36] it has been shown that the definition of HD-bisimilarity applied to HD-automata obtained from π-calculus agents induces over π-calculus 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 they 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 ordinary 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 $in?(*)$, we have to give a global meaning, say v, to the fresh name $*$. Then, we reach the state $a!b.\texttt{nil}$, where the global meaning of names a and b is *out* and v, respectively. It is immediate to see that this corresponds to the π-calculus transition $P(in, out) \xrightarrow{in?(v)} out!v.\texttt{nil}$.[3] 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

[3] Notice that parentheses have been added around to name v in order to stress that v is used as a fresh name in the transition.

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.

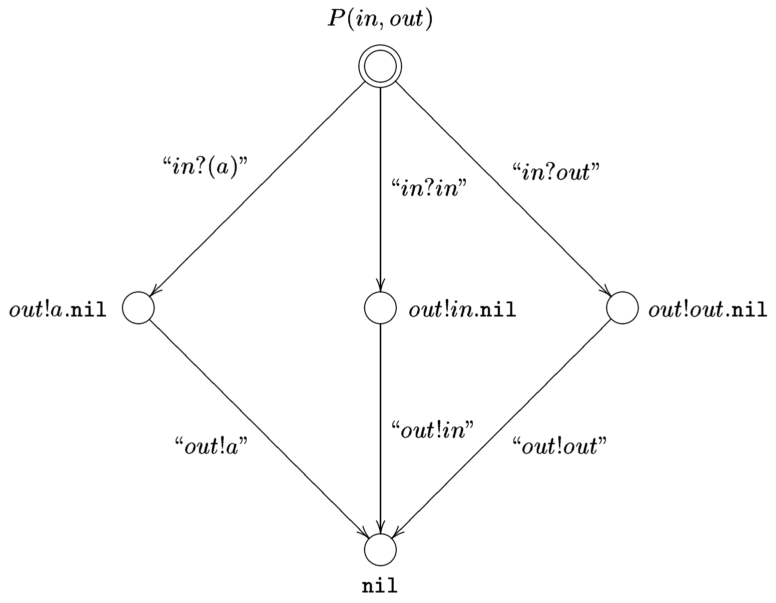


Fig. 2. The ordinary automaton corresponding to the HD-automaton of Figure 1.

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 bisimilar π-calculus agents are mapped into 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 guarantee this, 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. For instance, in the case of agent $Q(in, out, w) := in?(x).out!x.\mathtt{nil}$ one can see that name w does not play an active role and can hence be removed from the state of the HD-automaton corresponding to Q. As a consequence, also the transition $Q(in, out, w) \xrightarrow{in?w} out!w.\mathtt{nil}$ disappears, yielding an irredundant automaton. The irredundant HD-automaton is more compact than the starting "redundant" HD-automaton, but describes the same behaviors. More precisely, the irredundant HD-automaton and the staring HD-automaton and HD-bisimilar.

In [32] a simple and efficient algorithm is described to make irredundant the HD-automata corresponding to π-calculus agents without matching. HAL exploits an extension of this algorithm which is able to handle also a limited form of matching[4].

In [32, 36] it has been shown that the standard definition of bisimulation applied to ordinary automata obtained from *irredundant* HD-automata induces on the HD-automata a relation that coincides with HD-bisimilarity. This yields a procedure for checking the bisimilarity of two π-calculus agents. The two agents are translated into HD-automata. These are made irredundant and translated into ordinary automata. Finally, standard bisimilarity checking algorithms are exploited on the ordinary automata. The theoretical results of [32, 36] ensure the correctness of this procedure.

To conclude this section we show the expressiveness of HD-automata in handling bisimilarity.

Example 2. 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.

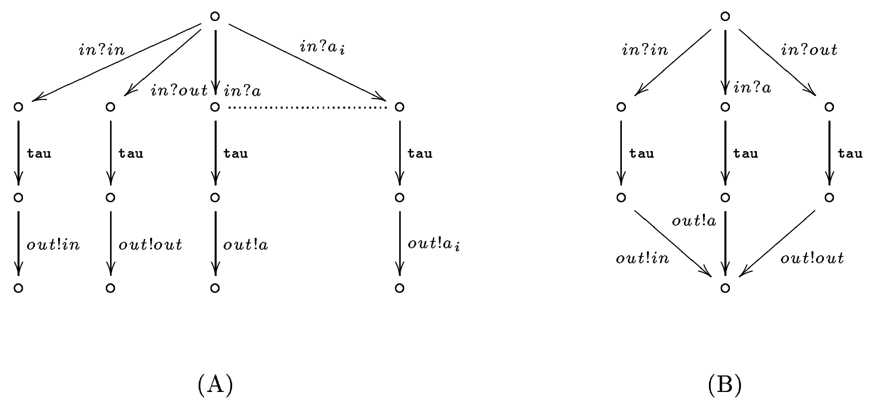


Fig. 3. State space representation of agent $Q(in, out)$.

[4] Intuitively, the names that appear in a matching must be bound and can never be the objects of bound output transitions.

4 From π-logic to ACTL

Our purpose is now to define an automatic verification procedure to model check whether or not a π-logic formula holds for a π-calculus specification. 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. This translation is possible using ACTL [14], for which an efficient model checker has been implemented [18] 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 action labels that occur in the transitions of the ordinary automaton associated with the agent P.

Definition 6. *Let $\theta = \{y'/y\}$. We define $\mu\theta$ as being the action μ' obtained from μ by replacing the occurrences of the name y with the name y'. Moreover, we define $true\theta = true$, $(\phi_1\&\phi_2)\theta = \phi_1\theta\&\phi_2\theta$, $(\sim\phi)\theta = \sim\phi\theta$, $(EX\{\mu\}\phi)\theta = EX\{\mu\theta\}\phi\theta$, $(EF\phi)\theta = EF\phi\theta$ and $(EF\{\chi\}\phi)\theta = EF\{\chi\theta\}\phi\theta$.*

Definition 7 (translation function). *Given a π-logic formula ϕ and a set of action labels S, the ACTL translation of ϕ is the ACTL formula $\mathcal{T}_S(\phi)$ defined as follows:*

- $\mathcal{T}_S(true) = true$
- $\mathcal{T}_S(\phi_1\&\phi_2) = \mathcal{T}_S(\phi_1)\&\mathcal{T}_S(\phi_2)$
- $\mathcal{T}_S(\sim\phi) = \sim\mathcal{T}_S(\phi)$
- $\mathcal{T}_S(EX\{\mu\}\phi) = \bigvee_{\mu' \in \mathcal{T}_S(\mu)} EX\{\mu'\}\mathcal{T}_S(\phi\theta)$ *where* $\theta = \{y'/y\}$ *if* $bn(\mu) = y$ *and* $bn(\mu') = y'$
- $\mathcal{T}_S(EF\phi) = EF\mathcal{T}_S(\phi)$
- $\mathcal{T}_S(EF\{\chi\}\phi) = E[true\{\bigvee_{\mu' \in \mathcal{T}_S(\chi)}\mu'\}U\mathcal{T}_S(\phi)]$

where:

- $\mathcal{T}_S(\mathtt{tau}) = \{\mathtt{tau}\}$
- $\mathcal{T}_S(x!y) = \{x!y\}$
- $\mathcal{T}_S(x!(y)) = \{x!(z) \in S \mid z \textit{ is a name }\}$
- $\mathcal{T}_S(x?y) = \{x?y\} \cup \{x?(z) \in S \mid z \textit{ is a name }\}$
- $\mathcal{T}_S(\sim\mu) = S \setminus \mathcal{T}_S(\mu)$
- $\mathcal{T}_S(\bigvee_{i\in I}\mu_i) = \bigcup_{i\in I}\mathcal{T}_S(\mu_i)$

Here, we assume that when $S = \emptyset$ then $\bigvee_{\mu' \in \mathcal{T}_S(\chi)} \phi = false$. Notice that the complexity of the translation has a worst case complexity which is exponential in the number of actions appearing in set S.

Example 3. 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 actions S used in the ordinary automaton of P. The translation of the formula is:

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

since the only bound input action in S is $in?(a)$. Note that the resulting ACTL formula holds in the ordinary automaton of P.

Assume now that S contains two bound input actions $in?(a)$ and $in?(b)$. In this case the translation yields the formula:

$$\begin{aligned} EX\{in?u\}EX\{out!u\}true \vee\ & EX\{in?(a)\}EX\{out!a\}true \\ \vee\ & EX\{in?(b)\}EX\{out!b\}true \end{aligned}$$

The correctness of the translation is shown in [23]. More precisely, let P be a π-calculus agent and let A be the corresponding ordinary automaton (namely, the automaton obtained by translating P into an HD-automaton, by making the HD-automaton irredundant, and by translating it into an ordinary automaton). Then P satisfies a π-logic formula ϕ if, and only if, A satisfies the formula $\mathcal{T}_S(\phi)$, where S are the action labels of A. We remark that ACTL is adequate with respect to the standard bisimulation on ordinary automata. Therefore, in the verification of formula $\mathcal{T}_S(\phi)$ it is possible to replace automaton A with a bisimilar automaton. This makes it possible, for instance, to minimize the automaton A before doing the actual verification.

We conclude this section observing that one of the advantages of model checking is that, if a formula is false, a counter-example is returned by the verification engine. This counter-example can guide the user in detecting and fixing the error. Currently, the counter-example is returned on the ordinary automaton, and the user is responsible of re-interpreting it on the starting π-calculus agent.

5 HAL Architecture

The previous sections outlined the theoretical foundations of an automata-based approach to the finite state verification of name passing process calculi. It remains to show that this theory can be exploited as a basis for the design and development of an effective and usable verification toolkit. This section and the one following explore this issue by describing our experience in experimenting an environment, called HAL for verifying finite state mobile systems represented in the π-calculus.

HAL has been implemented on top of the JACK environment [4]. The idea behind JACK[5] is to combine different specification and verification toolkits [28, 42, 3, 18], around a common format for representing ordinary automata: the FC2 file format [5]. FC2 allows interoperability among JACK tools. Moreover, tools can easily be added to the JACK system, thus extending its potential. 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 JACK front-ends allow specifications to be described both in textual form and in graphical form, by drawing automata. Moreover, JACK provides 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 several behavioral equivalences. Finally, 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.

The HAL toolkit is the component of JACK which provide facilities to deal with π-calculus specification by exploiting HD-automata. The goal of HAL is to verify properties of mobile systems specified in the π-calculus. Exploiting HAL facilities, π-calculus specifications are translated first into HD-automata and then in ordinary automata. Hence, the JACK bisimulation checkers can be used to verify bisimilarity. Automata minimization, according to bisimulation is also possible. HAL supports verification of logical formulae expressing properties of the behaviour of π-calculus specifications. The ACTL model checker provided by JACK can be used for verifying properties of π-calculus specifications, after that the π-logic formulae expressing the properties have been translated into ACTL formulae. Notice that the complexity of the model checking algorithm depends on the construction of the state space of the π-calculus agent to be verified, which is, in the worst case, exponential in the syntactic size of the agent.

The architecture of HAL is displayed in Figure 4. The current implementation 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 automata, and from π-logic formulae to ACTL formulae. The fourth module provides several routines that manipulate the internal representation of HD-automata. The routine for making HD-automata irredundant (see Section 3.2) is contained in this module. 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 (see also Figures 5 and 6). The Agent side allows:

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

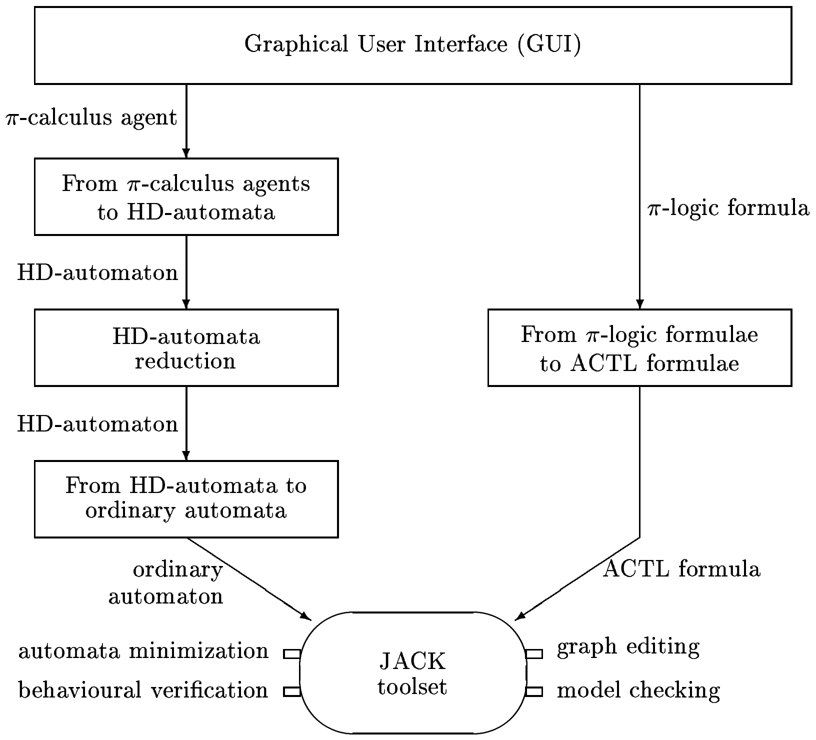
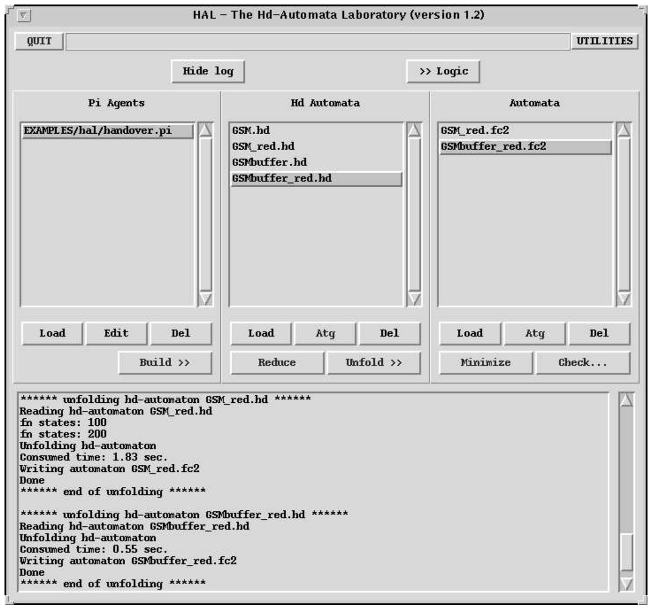


Fig. 4. The logical architecture of the HAL environment.

- the transformation of π-agents into HD-automata and then into ordinary automata (options **Build** and **Unfold**),
- the verification of equivalence of ordinary automata (option **Check**).

The Logic side allows a π-logic formula to be translated in the corresponding ACTL formula taking into account the specific automaton on which it will be checked (option **Translate**), and its verification through model checking (option **Check**).

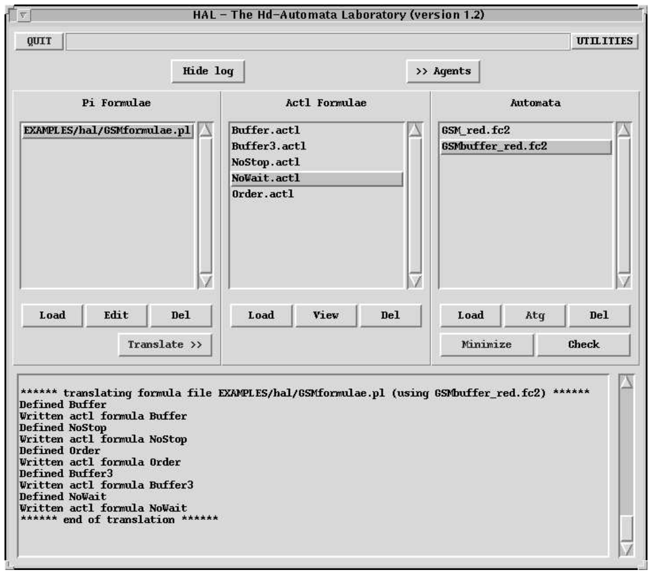
Several optimizations have been implemented. These optimizations reduce the state space of HD-automata, thus allowing a more efficient generation of the ordinary automata associated with π-calculus agents. An example of optimization is given by the reduction of `tau` chains (that are unbranched sequences 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



```
****** unfolding hd-automaton GSM_red.hd ******
Reading hd-automaton GSM_red.hd
fn states: 100
fn states: 200
Unfolding hd-automaton
Consumed time: 1.83 sec.
Writing automaton GSM_red.fc2
Done
****** end of unfolding ******

****** unfolding hd-automaton GSMbuffer_red.hd ******
Reading hd-automaton GSMbuffer_red.hd
Unfolding hd-automaton
Consumed time: 0.55 sec.
Writing automaton GSMbuffer_red.fc2
Done
****** end of unfolding ******
```

Fig. 5. The Agent side of HAL.



```
****** translating formula file EXAMPLES/hal/GSMformulae.pl (using GSMbuffer_red.fc2) ******
Defined Buffer
Written actl formula Buffer
Defined NoStop
Written actl formula NoStop
Defined Order
Written actl formula Order
Defined Buffer3
Written actl formula Buffer3
Defined NoWait
Written actl formula NoWait
****** end of translation ******
```

Fig. 6. The Logical side of HAL.

of input transitions is reduced. The semantic handling of constants is presented in [36]. Constants have to be declared in the π-calculus specifications.

The distinguished feature of our approach is the reduction of a specific name-based theory (the π-calculus) into an automata-like intermediate format (HD-automata). Theoretical results ensure soundness of this reduction. Furthermore, this allowed us the semantic re-use of both verification principles and automatic methods specifically developed for ordinary finite state automata. The main drawback of this approach is the generation of counterexamples when the π-calculus specification does not satisfies some properties. Indeed, counterexamples are generated by the JACK model checker but they are shown in the ordinary automata world. In other words, users are responsible of re-interpreting them on the orginal π-calculus specifications. This can be done, for instance, by exploiting the Autograph toolkit (a module of the JACK system) which provides services to animate and visualize ordinary automata.

HAL is written in C++ and compiles with the GNU C++ compiler. The GUI is written in Tcl/Tk. HAL is currently running on SUN stations (under SUN-OS) and on PC stations (under Linux). Recently, the HAL toolkit has been restructured and made available as a WEB service. By a few clicks in a browser at the URL `http://matrix.iei.pi.cnr.it:8080/halontheweb/` the HAL web service can be accessed remotely and its facilities can be exercised directly over the web.

6 Verification Case Studies

In this section we discuss some experimental results of HAL in the analysis of mobile systems specified in the π-calculus. The experiments have been run on a PC with a Pentium 4 - 1.80GHz processor and 512MB memory running Linux 2.4.18. The examples are available, along with the HAL bundle at URL: `http://matrix.iei.pi.cnr.it/projects/hal`.

6.1 Data Structures

The first example concerns reasoning about data structures represented as π-calculus processes. This is a simple exercise in reasoning about π-calculus specifications and yet allows the demonstration of common verification patterns which arise frequently when using π-calculus specifications.

To begin, let us consider the simplest example of a memory cell. A cell can be represented as a (recursive) process of the form

```
define Cell(i,o) = i?(c).o!c.Cell(i,o)
```

A process can store a new value via the channel `i` and read the stored value via the channel `o`. We shall exploit the `Cell` data structure to construct more interesting data structures. Figure 7 illustrates the HAL specification of two data structures: heap and buffer. For simplicity, we assume that the two data structures have a fixed size, and that names `in, out` are constants. For completeness,

π-spec	π-to-hd	red-hd	hd-to-aut	aut-min
`heap1`	2 (0.00)	2 (0.00)	2 (0.00)	2 (0.00)
`heap2`	7 (0.01)	7 (0.01)	11 (0.00)	6 (0.00)
`heap3`	22 (0.04)	22 (0.02)	75 (0.05)	20 (0.00)
`heap4`	74 (0.17)	74 (0.03)	700 (0.62)	70 (0.12)
`heap5`	277 (1.04)	277 (0.15)	8476 (15.42)	252 (2.21)
`heap6`	1154 (6.66)	1154 (0.90)	126125 (715.28)	924 (141.96)
`heap7`	5294 (46.00)	5294 (5.66)	-	-
`heap8`	26441 (338.26)	26441 (35.62)	-	-

Table 2. Heap specification: Model construction.

we also report the `build` statements. These statements are used to invoke the HAL facility that constructs the HD-automaton associated with a π-calculus process.

We expect that the specifications above satisfy certain properties. A main requirement is that whenever a value is inserted in the data structure, then it is possible to make it available in output. The following formula represents this property:

```
define Memory = AG([in?m]EF(<out!m>true))
```

Other interesting properties to be verified are

```
define NoDeadlock = AG(<in?*>true | <out!*>true)
```

(specifying that the evolution never reaches a deadlock state) and

```
define Order = AG([in?m][in?n] ~ (EF {~out!m} EX {out!n} true))
```

(specifying that the data structure adopts a FIFO policy, namely that, if name `m` is received before name `n`, then there is no path along which name `n` is emitted before name `m`).

Table 2 illustrates the results of the model creation for heap specifications. The model is computed by generating the HD-automaton associated to the π-calculus specification (column π-to-hd), the resulting HD-automaton is then made irredundant (column red-hd), the irredundant HD-automata is transformed into an ordinary automaton (column hd-to-aut) and then minimized (column min-aut). The numerical entries of the table give the number of state of the automaton and the construction time (in seconds). Notice that we were not able to construct the ordinary automata corresponding to `heap7` and `heap8` specifications because of state explosion. Table 3 illustrates the results of the model checking activity for the `heap4` specification. Tables 4 and 5 illustrate the results of our experiments for the buffer specifications. We remark that property `Order` is true only for buffers, while the other two properties are true for both structures.

Heaps:

```
define Heap1(in,out) = Cell(in,out)
define Heap2(in,out) = Cell(in,out)|Heap1(in,out)
define Heap3(in,out) = Cell(in,out)|Heap2(in,out)
define Heap4(in,out) = Cell(in,out)|Heap3(in,out)
define Heap5(in,out) = Cell(in,out)|Heap4(in,out)
define Heap6(in,out) = Cell(in,out)|Heap5(in,out)
define Heap7(in,out) = Cell(in,out)|Heap6(in,out)
define Heap8(in,out) = Cell(in,out)|Heap7(in,out)

const in
const out

build Heap1
build Heap2
build Heap3
build Heap4
build Heap5
build Heap6
build Heap7
build Heap8
```

Buffers:

```
define Buffer1(in,out) = Cell(in,out)
define Buffer2(in,out) = (c)(Cell(in,c)|Buffer1(c,out))
define Buffer3(in,out) = (c)(Cell(in,c)|Buffer2(c,out))
define Buffer4(in,out) = (c)(Cell(in,c)|Buffer3(c,out))
define Buffer5(in,out) = (c)(Cell(in,c)|Buffer4(c,out))
define Buffer6(in,out) = (c)(Cell(in,c)|Buffer5(c,out))
define Buffer7(in,out) = (c)(Cell(in,c)|Buffer6(c,out))
define Buffer8(in,out) = (c)(Cell(in,c)|Buffer7(c,out))

const in
const out

build Buffer1
build Buffer2
build Buffer3
build Buffer4
build Buffer5
build Buffer6
build Buffer7
build Buffer8
```

Fig. 7. Heap and buffer specifications.

π-spec	property	results
`heap4`	`Memory`	OK (0.01)
`heap4`	`NoDeadlock`	OK (0.00)
`heap4`	`Order`	NO (0.01)

Table 3. Heap specification: Model checking results.

π-spec	π-to-hd	red-hd	hd-to-aut	aut-min
`buffer1`	2 (0.00)	2 (0.00)	2 (0.00)	2 (0.00)
`buffer2`	7 (0.02)	5 (0.00)	8 (0.00)	7 (0.00)
`buffer3`	20 (0.03)	14 (0.00)	51 (0.02)	37 (0.00)
`buffer4`	67 (0.13)	51 (0.02)	504 (0.18)	297 (0.05)
`buffer5`	255 (0.80)	209 (0.08)	6370 (3.89)	3251 (3.26)
`buffer6`	1080 (4.88)	930 (0.38)	97473 (158.74)	45013 (1154.27)
`buffer7`	5017 (31.53)	4461 (2.02)	-	-
`buffer8`	25287 (213.58)	22977 (11.63)	-	-

Table 4. Buffer specification: Model construction.

6.2 λ-calculi

Although based on simple primitives, the π-calculus is very expressive. It can encode the λ-calculus and other functional programming formalisms. Similarly, a variety of imperative, object-oriented and concurrent programming may be reduced to the π-calculus. Several encodings of λ-calculus evaluation strageties into the π-calculus have been developed. All these encodings have three common features:

- function application is modeled as parallel composition,
- β-reduction is modeled as synchronization,
- the encoding is parameterized over a name which models the environment.

Here, we do not discuss the theories underlying the interpretation of λ-calculi into the π-calculus (we refer to [40] for the detailed treatment). Instead, we aim at showing how HAL can be exploited to reason about such encodings.

Let us consider the following three simple λ-calculus terms:

- $P = (\lambda x.x)$,
- $Q = (\lambda x.x)(\lambda x.x)$, and
- $R = (\lambda x.(xx))(\lambda x.x)$.

Their π-calculus interpretation is as follows:

```
define P(u) = u?(p). p?(x). p?(v) . x!v . nil

define FIX_P(z)   = z?(w). (FIX_P(z) | P(w))
define Q(u) = (v)(P(v) | (z)(p)(v!p. p!z. p!u. nil | FIX_P(z)))
```

π-spec	property	results
`buffer4`	`Memory`	OK (0.01)
`buffer4`	`NoDeadlock`	OK (0.02)
`buffer4`	`Order`	OK (0.03)

Table 5. Buffer specification: Model checking results.

π-encoding	π-to-hd	red-hd	hd-to-aut	aut-min
`P`	0.00	0.00	0.00	0.00
`Q`	0.02	0.00	0.00	0.00
`R`	0.03	0.00	0.00	0.00

Table 6. Interpreting λ terms: Model construction.

```
define FIX_x(z,x) = z?(w). (FIX_x(z,x) | x!w.nil)
define Aux(u) = u?(p). p?(x). p?(v) .
   (w)(x!w. nil | (z)(p) (w!p. p!z. p!v. nil | FIX_x(z,x)))
define R(u) = (v)(Aux(v) | (z)(p)(v!p. p!z. p!u. nil | FIX_P(z)))
```

Table 6 illustrates the results of the model creation for the encoding (we only report the construction time). Theoretical results guarantee that P is bisimilar to Q and R. Checking the two bisimilarities in HAL takes 0.01 sec. and 0.02 sec, respectively.

In order to experiment HAL with λ-calculus encodings, we introduce a "'wrong" encoding into the π-calculus of the three λ terms presented above:

```
define P(u) = u?(p). p?(x). p?(v) . x!v . nil

define Q(u) = (v)(P(v) | (z)(p)(v!p. p!z. p!u. nil | z?(w). P(w)))

define Aux(u) = u?(p). p?(x). p?(v) .
   (w)(x!w. nil | (z)(p) (w!p. p!z. p!v. nil | z?(w). x!w. nil))
define R(u) = (v)(Aux(v) | (z)(p)(v!p. p!z. p!u. nil | z?(w). P(w)))
```

Exploiting HAL facilities we compute that this encoding is not correct: P is bisimilar to Q (0.03 sec), but is *not* bisimilar to R (0.01 sec).

6.3 Security Protocols

Cryptography is an important mechanism for achieving security in distributed systems. The creation of unique names (nonces) is an essential primitive to identify sessions or to timestamp freshness of a message in security protocols. The generation of nonces is an instance of the dynamic name generation provided by the π-calculus. This observation has led to modeling keys as names, key generation as restriction, and communication on a cryptographic channel as communication on its key. For instance, the following π-calculus processes provide a simple specification of a set of cryptographic primitives with symmetric keys.

```
define CryptedMsg(cm,k,m) = cm?(x). (t)(k!t. t?(y). t?(r). [y=x] r!m.nil)

define Encrypt(r,k,m) = (cm)(r!cm. CryptedMsg(cm,k,m))

define Decrypt(r,k,cm) = (x)(cm!x.k?(t).t!x.t!r.nil)
```

Basically, the specification above describes a ciphertext (a cryptographic message) as an abstract object with the methods: Encrypt and Decrypt.

The spi calculus [1] is an extension of the π-calculus with basic primitives to represent ciphertexts, and constructs for generation of nonces and keys. Spi-like calculi have been used to specify and verify secrecy and authenticity properties of several cryptographic protocols. Moreover, there has been some work on designing and implementing toolkits to assist reasoning about security protocols. Here, we do not discuss in detail the issues of name passing calculi for security, but, we aim at showing the use of HAL facilities to assist specification and verification of security protocols. We illustrate some examples for secure communication and key exchange. In all case we model check properties over the models generated by the protocol specifications. Clearly, our security protocol specifications are lower level than those of other formalisms specifically designed to specify security protocols, since our goal is to illustrate the usability of HAL to tackle a variety of specification issues of global computing systems.

The following π-calculus specifications describe four simple security protocols. Processes SimpleSP1 and SimpleSP2 receive two plain-text messages over channel in, SimpleSP1 encrypts the first message; SimpleSP2, instead, encrypts the second message. The resulting cipher-text is sent along channel out.

```
define SimpleSP1(in,out) =
   (k)(in?(m1).in?(m2). (r)(Encrypt(r,k,m1) | r?(cm). out!cm.nil))

define SimpleSP2(in,out) =
   (k)(in?(m1).in?(m2). (r)(Encrypt(r,k,m2) | r?(cm). out!cm.nil))
```

Under the perfect encryption hypothesis, the external, hostile, environment is not able to decrypt the cipher text: the protocol does not leak the content of the cipher text. This is the standard notion of leaking formalized in terms of bisimilarity. Indeed, the two π-calculus processes are bisimilar: HAL takes 0.01 seconds to check bisimilarity.

Let us now consider processes SimpleSP3 and SimpleSP4:

```
define SimpleSP3(in,out) =
   (k)(in?(m1).in?(m2). (r)(Encrypt(r,k,m1) | r?(cm). out!cm.out!k.nil))

define SimpleSP4(in,out) =
   (k)(in?(m1).in?(m2). (r)(Encrypt(r,k,m2) | r?(cm). out!cm.out!k.nil))
```

In this case, the protocol does leak the encryption key k in the hostile environment, and HAL reports correctly that the two processes are *not* bisimilar.

We now consider other simple security protocols. Protocol SP1 models a cryptographic communication along a unsecure channel between two principals sharing a symmetric key. Protocol SP2 models a cryptographic communication where the initiator of the protocol encrypts, using a symmetric key, the session key. The message is then forwarded along the unsecure channel exploiting the session key.

```
define P1(in,bus,k) =
   in?(m). (r) (Encrypt(r,k,m) | r?(cm). bus!cm. nil )

define Q1(bus,out,k) =
   bus?(cm). (r)(Decrypt(r,k,cm) | r?(m). out!m. nil)

define SP1(in,bus,out) = (k) (P1(in,bus,k) | Q1(bus,out,k))

define P2(in,bus,k1) = in?(m). (r1)(r2)(k2)
   (Encrypt(r1,k1,k2) |
     r1?(cm1). bus!cm1. Encrypt(r2,k2,m) | r2?(cm2). bus!cm2. nil )

define Q2(bus,out,k1) =  bus?(cm1). bus?(cm2). (r1)
   (Decrypt(r1,k1,cm1) |
     r1?(k2). (r2)(Decrypt(r2,k2,cm2) | r2?(m). out!m. nil))

define SP2(in,bus,out) = (k1) (P2(in,bus,k1) | Q2(bus,out,k1))
```

To reason about these security protocols we consider the following properties whose meaning is straightforward. Table 7 illustrates the results of experimenting with HAL to verify these cryptographic protocols on these properties:

```
define AlwaysSuccess = AG([in?n]<out!n>true)

define PossibleSuccess = AG([in?m]EF(<out!m>true))

define NoWrongOutput = AG([in?n][out!m]false)
```

We see that it is not possible to ensure the success of communication (property `AlwaysSuccess`), since messages flow on public channels that are unreliable (e.g. messages can be intercepted). However, the communication *may* be successful (property `PossibleSuccess`), and in this case we can ensure the reception the right message (property `NoWrongOutput`).

To conclude this section, we consider a security protocol of larger size: the well known *Wide Mouth Frog* (WMF) protocol [2]. In the WMF protocol principal `A` transmits a secret message to principal `B` by a session key `k`. The session key is exchanged through a trust server `S`. The keys `kas`, `kbs` are secret keys for communicating from the server to principals `A` and `B`. The WMF protocol can be specified as follows. Table 8 illustrates the experimental results for the WMF protocol.

security protocol	π-to-hd	red-hd	hd-to-aut	aut-min
SP1	49 (0.09)	25 (0.00)	39 (0.01)	25 (0.00)
SP2	726 (4.56)	466 (0.12)	1329 (0.31)	270 (0.39)

security protocol	property	results	computation time
SP1	AlwaysSuccess	NO	0.00
SP1	PossibleSuccess	OK	0.01
SP1	NoWrongOutput	OK	0.00
SP2	AlwaysSuccess	NO	0.02
SP2	PossibleSuccess	OK	0.02
SP2	NoWrongOutput	OK	0.03

Table 7. Reasoning about security protocols.

security protocol	π-to-hd	red-hd	hd-to-aut	aut-min
WMF	2046 (24.89)	1079 (0.35)	4146 (1.39)	436 (1.72)

security protocol	property	results	computation time
WMF	AlwaysSuccess	NO	0.05
WMF	PossibleSuccess	OK	0.05
WMF	NoWrongOutput	OK	0.03

Table 8. Reasoning on the WMF protocol

```
define A(kas,in,bus) =
   in?(v).(k)(r)(Encrypt(r,kas,k) |
     r?(ck). bus!ck. (r)(Encrypt(r,k,v) |
       r?(cv). bus!cv. nil))

define B(kbs,out,bus) =
   bus?(ck).(r)(Decrypt(r,kbs,ck) |
     r?(k). bus?(cv). (r)(Decrypt(r,k,cv) |
       r?(v). out!v. nil))

define S(kas,kbs,bus) =
   bus?(ck).(r)(Decrypt(r,kas,ck) |
     r?(k). (r)(Encrypt(r,kbs,k) |
       r?(ck). bus!ck. nil))

define WMF(in,out,bus) =
   (kas)(kbs)(A(kas,in,bus)|B(kbs,out,bus)|S(kas,kbs,bus))
```

```
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)
```

Fig. 8. π-calculus specification of GSM modules.

6.4 The Handover Protocol for Mobile Telephones

The last case study concerns the specification of the core of the handover protocol for the GSM Public Land Mobile Network (GSM) proposed by the European Telecommunication Standards Institute. The specification is borrowed from that given in [44], which has been in turn derived from that in [38].

The π-calculus specification of the GSM protocol is

```
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))
```

`Centre` receives messages from the environment on channel `in`; these input actions are the only observable actions performed by `Centre`. Module `Car` sends the messages to the end user along the channel `out`; these outputs are the only visible actions performed by the `Car`. Modules `Centre` and `Car` interact via the base corresponding to the cell in which the car is located. The specification of modules `Car`, `Base`, `IdleBase` and `Centre` is reported in Figure 8. The behaviour of the four modules is briefly summarized below:

- `Car` brings a MobileStation and travels across two different geographical areas that provide services to end users;
- `Base` and `IdleBase` are Base Station modules; they interconnect the MobileStation and the MobileSwitching Centre;

– `Centre` is a MobileSwitching centre which controls radio communications within the whole area composed by the two cells.

The protocol starts when `Car` moves from one cell to the other. Indeed, `Centre` communicates to the MobileStation the name of the base corresponding to the new cell. The communication of the new channel name to the MobileStation is performed via the current base. All the communications of messages between the MobileSwitching centre and the MobileStation are suspended until the MobileStation receives the names of the new transmission channels. Then the base corresponding to the new cell is activated, and the communications between the MobileSwitching centre and the MobileStation continue through the new base.

The observable behaviour of the GSM protocol can be abstracted by a three position buffer. The buffer queues the messages and is specified by the module `GSMbuffer` as follows

```
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)
```

Figure 9 illustrates a version of the GSM protocol that models the MobileSwitching and MobileStation modules in a more realistic way. Indeed, the 'full' version exploits a protocol for establishing whether or not the car is crossing the boundary of a cell and entering the other cell.

It is possible to check that `GSM`, `GSMfull` and `GSMbuffer` have the same behaviour. Indeed, `GSM` and `GSMfull` are proved to be weakly bisimilar to `GSMbuffer`. Table 9 gives the performance figures of the model creation and of the bisimulation checks when `in` and `out` are assumed constant names. Table 10 gives the figures of the model creation when `in` and `out` are not constant names.

We expect that the GSM specifications satisfy some properties. Namely, the protocol is *reliable*: when a message has been sent then it is possible to receive it and in-order delivery is guaranteed. The logical formulae:

```
Reliable1 = AG([in?n]EF<out!n>true)

Reliable2 = AG([in?m][in?n] ~ (EF {~out!m} EX {out!n} true))
```

specify this property. The first formula states that whenever a message `m` is received from the external environment through the channel `in` then it will be

```
define 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))

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

define 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))

define 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)))

define 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)

define 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)))

define 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))

define 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))

define GSMfull(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))
```

Fig. 9. Full specification of the GSM protocol.

protocol	π-to-hd	red-hd	hd-to-aut	aut-min
`GSMbuffer`	12 (0.00)	12 (0.00)	49 (0.02)	49 (0.02)
`GSM`	55 (0.33)	35 (0.02)	135 (0.03)	49 (0.01)
`GSMfull`	260 (3.07)	146 (0.03)	557 (0.12)	49 (0.05)

`GSMbuffer` $\sim$ `GSM`	OK (0.14)
`GSMbuffer` $\sim$ `GSMfull`	OK (0.14)
`GSM` $\sim$ `GSMfull`	OK (0.13)

Table 9. GSM Protocol: Model creation and bisimilarity.

protocol	π-to-hd	red-hd	hd-to-aut	aut-min
`GSMbuffer`	65 (0.04)	65 (0.02)	164 (0.03)	163 (0.01)
`GSM`	211 (1.18)	167 (0.04)	407 (0.08)	163 (0.04)
`GSMfull`	960 (10.37)	676 (0.15)	1633 (0.31)	163 (0.15)

Table 10. Model creation without constant names.

eventually retransmitted to the end user via the channel `out`. The meaning of the second formula is that if name `m` is received before `n`, then there is no path along which name `n` is emitted before `m`.

We also expect that the formula

```
FastTransmission = AG([in?m1][in?m2][in?m3]
    ([out!m2]false &
[out!m3]false & <out!m1>true & [in?*]false))
```

will be satisfied. This formula states that whenever three messages are received in sequence through the channel `in`, then the first message will be retransmitted soon to the end user through the channel `out` before performing any input from channel `in`.

Other logical formulae expressing propeties are

```
NoStop = AG(EX{*?*}true | EX{*!*}true | EX{} true)
```

(that states that the protocol is always running) and

```
NoWait = AG(<in?*>true)
```

(that states that input operations have higher priority than the other operations of the protocol).

Notice that in the above formulae the shorthand `{*?*}` is used to indicate any input action and `in?*` is used to denote the reception of any name.

Assuming `in` and `out` constant names, the performance figures of the model checking are given in Table 11. All the properties are true, except `NoWait` (the handover phase has higher priority than the input of new messages).

GSMbuffer

π-logic-to-actl	(0.02)
`Reliable1`	OK (0.00)
`Reliable2`	OK (0.00)
`FastTrasmission`	OK (0.01)
`NoStop`	OK (0.00)
`NoWait`	NO (0.00)

GSM

π-logic-to-actl	(0.00)
`Reliable1`	OK (0.00)
`Reliable2`	OK (0.00)
`FastTrasmission`	OK (0.02)
`NoStop`	OK (0.00)
`NoWait`	NO (0.00)

GSMfull

π-logic-to-actl	(0.00)
`Reliable1`	OK (0.00)
`Reliable2`	OK (0.02)
`FastTrasmission`	OK (0.00)
`NoStop`	OK (0.00)
`NoWait`	NO (0.00)

Table 11. Model checking GSM properties.

7 Concluding Remarks

We have described HAL, an automata-based verification environment for the π-calculus. We illustrated the usability of HAL by reasoning on a variety of specifications. Our approach differs from others in that we do not place emphasis on a specification language but rather we exploit the intermediate, syntax independent, format provided by HD-automata. HD-automata give the basis to handle finite state verification in the case of modern specification calculi for global computing systems (like π-calculus, spi-calculus, etc.).

Our current research activity proceeds mainly on two fronts. The first line of reasearch aims at extending HAL to work directly on HD-automata without requiring the mapping to ordinary automata for verification. In particular, we are developing a module able to directly minimize HD-automata. Foundational results [33, 34, 16] guarantee existence of minimal HD-automata. Moreover, the development of verification algorithms that work directly on HD-automata would also have the advantage of presenting the results of the verification e.g., the counter-examples) in a form that is nearer to the π-calculus notations. Currently counter-examples are returned on the ordinary automaton world, and the user is responsible of re-interpreting them on the original π-calculus specifications.

In the second line of reaserach, we are extending the theory of HD-automata in several ways. We plan to extend the basic model to endow states with a structure intended to describe and observe the spatial organization of systems in a broad sense. In particular, we are exploiting HD-automata as a general model for spatial logics [7, 6], where typically one is able to express that if a state has a certain structure, then it satisfies some properties. Another research activity concerns the development of abstraction and composition techniques for HD-automata to avoid the state explosion problem on specifications of complex global computing systems. To this purpose, we plan to investigate how the techniques developed in [8] can be applied in our framework.

To end the paper we make a more detailed comparison with the *Mobility Workbench* [44] (MWB in short). In the MWB the verification of bisimulation equivalence between (finite control) π-calculus agents is made *on the fly* [17], 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 [11, 12] 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). On the largest example we considered, the GSM protocol, the time required by MWB for checking bisimilarity (running on a Pentium4/2GHz machine) of `GSMfull` and `GSMbuffer` is similar to the time required by HAL: MWB requires about 5 seconds, while HAL requires about 11 seconds.

There are several differences between our approach and the one adopted in the MWB that make it difficult to perform a precise comparison of the two verification environments. For instance, in HAL 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. The π-logic, 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.

The main difference between the two approaches is mainly *methodological.* HAL has been designed in order to be largely *language independent*: to handle a formalism different from the π-calculus is suffices to construct a translation module mapping the new formalism into HD-automata. The structure of the MWB, instead, is tailored to the language.

Acknowledgments We thank Bjorn Victor and the anonymous reviewers for their suggestions.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proc. Royal Society of London A* 246, 233-271, 1989.
3. A. Bouali and R. de Simone. Symbolic bisimulation minimization. In *Proc. CAV'92*, LNCS 663. Springer-Verlag, 1992.
4. A. Bouali , S. Gnesi, and S. Larosa. The integration Project for the JACK Environment. *Bulletin of the EATCS*, 54, October 1994, pp. 207–223.
5. A. Bouali, A. Ressouche, V. Roy, and R. De Simone. The FC2Tools set. In *Proc. CAV'96*, LNCS 1102. Springer Verlag, 1996.
6. L. Caires and L. Cardelli. A spatial logic for concurrency (Part II), In *Proc. CONCUR'02*, LNCS 2421, Springer, 2002.
7. L. Caires and L. Cardelli. A spatial logic for concurrency (Part I), *Information and Computation*, special issue on TACS 2001. To appear.
8. S. Chaki, S. Rajamani, and J. Rehof. Types as Models: model checking message-passing programs. In *Proc. POPL'02*, ACM Press, 2002.
9. 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.
10. E. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.
11. M. Dam. Model checking mobile processes. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
12. 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.
13. 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):761-778. North-Holland, 1993.

14. 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.
15. E.A. Emerson and J.Y. Halpern. Sometimes and Not Never Revisited: on Branching Time versus Linear Time Temporal Logic. *Journal of ACM*, 33(1), January 1986, pp. 151–178.
16. G. Ferrari, U. Montanari, and M. Pistore. Minimizing Transition Systems for Name Passing Calculi: A Co-algebraic Formulation. In *Proc. FOSSACS'02*, LNCS 2303, 2002.
17. J.-C. Fernandez and L. Mounier. "On the fly" verification of behavioral equivalences and preorders. In *Proc. CAV'91*, LNCS 575. Springer Verlag, 1991.
18. G. Ferro. *AMC: ACTL Model Checker. Reference Manual.* IEI-Internal Report, B4-47, 1994.
19. M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1999.
20. R. Focardi and R. Gorrieri. The Compositional Security Checker: A tool for the verification of information flow security properties. *IEEE Transaction on Software Engineering*, 23(9):550–571, 1997.
21. 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.
22. A. Gordon. Notes on Nominal Calculi for Security and Mobility, In *Proc. FOSAD Sumer School*, LNCS 2171, Springer, 2001.
23. S. Gnesi and G. Ristori. A Model Checking Algorithm for π-calculus agents. In *Advances in Temporal Logic.* Kluwer Academic Publishers, pp.339-357, 2000.
24. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM* 32(1), pp. 137-161, 1985.
25. K. Honda. Elementary Structures for Process Theory (1): Sets with Renaming. *Mathematical Structures in Computer Science* 10:617–663, Cambridge University Press, 2000.
26. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science* 27, pp. 333-354, 1983.
27. M. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proc. TACAS'96*, LNCS 1055, 147-166, 1996.
28. E. Madelaine and D. Vergamini. AUTO: A verification tool for distributed systems using reduction of finite automata networks. *Formal Description Techniques* II, 1990.
29. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
30. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
31. R. Milner, J. Parrow, and D. Walker. Modal logic for mobile processes. *Theoretical Computer Science* 114:149-171, 1993.
32. U. Montanari and M. Pistore. Checking bisimilarity for finitary π-calculus. In *Proc. CONCUR'95*, LNCS 962. Springer Verlag, 1995.
33. U. Montanari and M. Pistore. π-calculus, structured coalgebras and minimal HD-automata. In *Proc. MFCS'00*, LNCS 1893. Springer, 2000.
34. U. Montanari and M. Pistore. Structured Coalgebras and Minimal HD-automata for the π-Calculus. To appear in the MFCS 2000 special issue of *Theoretical Computer Science.*

35. R. Needham. In *Distributed Systems* (Mullender, Ed). Addison-Wesley, 1989.
36. M. Pistore. *History Dependent Automata.* PhD. Thesis TD-5/99, Università di Pisa, Dipartimento di Informatica, 1999.
37. A.M. Pitts and M.J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction (MPC'00)*, LNCS 1837. Springer, 2000.
38. F. Orava and J. Parrow. An Algebraic Verification of a Mobile Network. *Formal Aspects of Computing* 4, pp. 497-543, 1992.
39. D. Park *Concurrency and automata on infinite sequences.* LNCS 104, 1981.
40. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2002.
41. P. Sewell. *Applied Pi - A Brief Tutorial.* Technical Report 498, Computer Laboratory, University of Cambridge 2000.
42. V. Roy and R. De Simone. AUTO and autograph. In *Proc. CAV'90*, LNCS 531. Springer Verlag, 1990.
43. D. Sangiorgi. A theory of bisimulation for the π-calculus. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
44. 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 [14] 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.

Definition 8 (action formulae). *Given a set of observable actions Act, the language $\mathcal{AF}(Act)$ of the action formulae on Act is defined as follows:*

$$\chi ::= true \mid b \mid {\sim}\chi \mid \chi \,\&\, \chi$$

where b ranges over Act.

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

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 9 (ACTL syntax). *The syntax of the* ACTL *formulae is given by the grammar below:*

$$\begin{aligned}\phi &::= true \mid \phi \,\&\, \phi \mid {\sim}\phi \mid E\pi \mid A\pi \\ \pi &::= X\{\chi\}\phi \mid X\{\mathtt{tau}\}\phi \mid [\phi\{\chi\}U\phi] \mid [\phi\{\chi\}U\{\chi'\}\phi]\end{aligned}$$

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.

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 $E[true\{false\}U\phi]$.

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

Definition 10 (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 i^{th} state in the sequence, i.e. r_i, by $\sigma(i)$.

Definition 11 (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 $a \models \chi$,*
- *$a \models \chi \mathbin{\&} \chi'$ iff $a \models \chi$ and $a \models \chi'$.*

Definition 12 (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*
- *$s \models \phi \mathbin{\&} \phi'$ 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 $\alpha_j \models \chi$*

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 [14]. 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 $F_i = \{\psi \in \text{ACTL} : \mathcal{A}_i \text{ satisfies } \psi\}$, $i = 1, 2$.