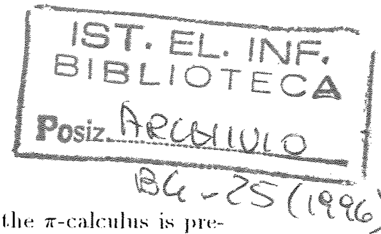
A Model Checking algorithm for π-calculus agents*

S. Gnesi[1], G. Ristori[1,2]

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



Abstract. In this paper an action-based logic for the π-calculus is presented. A model checker is built for this logic, following an automata-based approach. This is made possible by a result which allows finite state Labelled Transition Systems to be associated to a wide class of π-calculus agents, so that bisimilar Labelled Transition Systems are associated to bisimilar π-calculus agents. The model checker has then been built reusing an efficient model checker, that was developed to check the satisfiability of formulae of the action based logic ACTL on finite state Labelled Transition Systems, and implementing a sound translation from the π-logic into ACTL.

Category: Research paper

Topic areas: Pure temporal Logic, Specification and Verification.

Corresponding author:
Stefania Gnesi
Istituto di Elaborazione dell'Informazione - C.N.R.
Via S. Maria 46, 56100 Pisa (Italy)
fax: (+39)(+50)554342
email: gnesi@iei.pi.cnr.it

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

1 Introduction

The π-calculus [14] is a formalism suitable to model the behaviour of systems where the communication topology among processes can dynamically evolve when the computation progresses. Its primitives are simple but expressive: channel names can be created, communicated (thus giving the possibility of dynamically reconfigurating process acquaintances) and they are subjected to sophisticated scoping rules. Moreover names can be used to model objects [21] and higher order communication [18]. The π-calculus has greater expressive power than ordinary process calculi, but also a much more complicate theory. In particular, the usual operational models are infinite state and infinite branching.

Logics have been proposed [15, 2] 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 [11, 12].

Verification algorithms have been studied to check both behavioural and logical properties of π-calculus agents. In the case of infinite state systems verification techniques that rely on the representation of the state space of the system, reveal inadequate. For this class of systems, several decision procedures have been defined that rely on local methods [19] or on the symbolic representation of the state space [10]. This is the case of π-calculus for which a tableau based proof system, to check behavioural and logical properties, expressed in the logic [2], has been implemented [20].

A recent result [16] allows a finite representation of π-calculus agents to be given by associating to them finite state Labelled Transition Systems (i.e. finite automata). The theory of [16] ensures that equivalent Labelled Transition Systems are associated to equivalent π-calculus agents.

Starting from the result in [16], in this paper we address the issue of logic verification for the π-calculus. We present a logic, called π-logic, suitable for expressing properties of π-calculus agents, and a model checker for it, following an automata-based approach [1].

The model checker has been built using the implementation, described in [4], of the translation function of [16] and reusing an existing model checker, AMC [5, 6]. AMC checks the satisfiability of formulae of the action based logic ACTL [3] on finite state Labelled Transition Systems. The reuse is made possible by the definition of a sound translation function from π-logic formulae into ACTL ones. The model checker we propose supports verification of π-logic formulae, with modalities indexed by π-calculus actions, able to express a wide class of safety and liveness properties of π-calculus agents.

Our decision not to directly implement the model checker, but to reuse AMC, was due to the aim of rapidly building a model checker for the π-logic to experiment with the descriptive power of the new logic while relying on efficient and continuously upgraded tool.

The paper is organized as follows. Section 2 present some preliminary notions on Labelled Transition Systems and on the logic ACTL. In Section 3 a brief description of the π-calculus syntax and semantics is given and the π-logic is presented. Section 4 presents the structure of the proposed model checker. In

Section 5 the methodology that allows a π-calculus agent to be translated into a finite state Labelled Transition Systems is shown. Section 6 describes the translation into ACTL of the π-logic operators. In Section 7 an example is shown of the use of the model checker for the verification of logical properties of a dynamic protocol. Section 8 concludes the paper.

2 Preliminaries

In this Section we describe Labelled Transition Systems, that are a semantics model for concurrent systems, and the ACTL logic [3] (the action based version of the CTL temporal logic [8]) that is interpreted over Labelled Transition Systems.

Definition 1 Labelled Transition System. A Labelled Transition System (LTS in short) is a 4-tuple $\mathcal{A} = (Q, q_0, Act \cup \{\texttt{tau}\}, R)$, where:

- Q is a finite set of states;
- q_0 is the initial state;
- Act is a finite set of observable actions and tau is the unobservable action;
- $R \subseteq Q \times (Act \cup \{\texttt{tau}\}) \times Q$ is the transition relation. Whenever $(q, \alpha, q') \in R$ we will write $q \xrightarrow{\alpha} q'$.

Two LTSs are said *strongly equivalent*, or strongly bisimilar, if there exists a strong bisimulation relation between their initial states [13].

ACTL is a temporal logic that is suitable for describing the behaviour of systems that perform actions during their 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 (i.e. ACTL is a *branching time* temporal logic). The original definition of ACTL includes an auxiliary logic of action formulae.

Definition 2 Action formulae. Given a set of observable actions Act, the language $\mathcal{AF}(Act)$ of the action formulae on Act is defined as follows:

$$\chi ::= \texttt{true} \mid b \mid \neg\chi \mid \chi \ \&\ \chi$$

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 3 ACTL syntax. The syntax of the ACTL formulae is given by the grammar below:
$\phi ::= \texttt{true} \mid \phi \ \&\ \phi \mid \sim\phi \mid E\pi \mid A\pi$
$\pi ::= X\{\chi\}\phi \mid X\{\texttt{tau}\}\phi \mid [\phi\{\chi\}U\phi] \mid [\phi\{\chi\}U\{\chi'\}\phi]$
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 give the definition of the ACTL semantics, we need to introduce the notion of paths over a LTS.

Definition 4 Paths. Let $\mathcal{A} = (Q, q_0, Act \cup \{\texttt{tau}\}, R)$ be a LTS.

- 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 5 Action formulae semantics. The satisfaction relation $\models$ for action formulae is defined as follows:
$a \models \texttt{true}$ 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 $\chi | \chi'$ abbreviates $\sim (\sim \chi \,\&\! \sim \chi')$.

Definition 6 ACTL semantics. The satisfaction relation for ACTL formulae is defined in the following way:
$s \models \texttt{true}$ 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 $\sigma(1) \models \phi$;
$\sigma \models X\{\texttt{tau}\}\phi$ iff $\sigma = (\sigma(0), \texttt{tau}, \sigma(1))\sigma'$, 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 $\alpha_{j+1} = \texttt{tau}$ 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 $\alpha_j = \texttt{tau}$ or $\alpha_j \models \chi$.

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

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

The ACTL logic can be used to define *liveness* (something good eventually happens) and *safety* (nothing bad can happen) properties of concurrent systems. Moreover, the ACTL logic is *adequate* with respect to strong bisimulation equivalence on LTSs [3]. Adequacy means that given two LTSs $\mathcal{A}_1$ and $\mathcal{A}_2$, they are strongly bisimilar if and only if $F_1 = F_2$, where $F_i = \{\psi \in$ ACTL : $\mathcal{A}_i$ satisfies $\psi\}$, $i = 1, 2$.

3 A logic for the π-calculus

In this Section we present the logic formalism, called π-logic, that we define to express properties of π-calculus agents. We start with a brief description of the syntax and the operational semantics of the π-calculus, on which the logic formulae are interpreted.

Definition 7 π-calculus syntax. Given a denumerable infinite set $\mathcal{N}$ of *names* (denoted by $a, b, \ldots$), the π-calculus *agents* over $\mathcal{N}$ are defined as follows[3]:

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

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

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 we use indicate the set of free names of agent P with $\texttt{fn}(P)$. 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 *actions* that agents can perform are defined by the following syntax:

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

where x and y are free names of μ ($\texttt{fn}(\mu)$), whereas z is a bound name ($\texttt{bn}(\mu)$); finally $\texttt{n}(\mu) = \texttt{fn}(\mu) \cup \texttt{bn}(\mu)$. In the actions we can distinguish a subject, x and an object, y and z. The structural rules for the *early operational semantics* are defined in Table 1.

The strong early bisimulation [15] is given by the following definition:

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

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 $x \notin \mathrm{n}(\mu)$	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.

Definition 8. 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:

- 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 $(P, Q) \in B$.

In order to express properties of π-calculus agents, we introduce a temporal logic, called in the following π-logic, that is based on the logic formalism given in [15]. In [15], the modal operators of the classical Hennessy-Milner logic [11] are extended to deal with π-calculus actions.

Together with the *strong next* modality defined in [15], the π-logic also includes a *weak next* modality, whose meaning is that a number of `tau` can be executed before the occurrence of an action.[4] Moreover, to express general liveness and safety properties, the *eventually* temporal operator (notation $EF\phi$) is introduced.

Definition 9 **π-logic syntax and semantics.** The syntax of the π-logic is given by:

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

The interpretation of the logic formulae defined by the above syntax, is the following:

- $P \models \mathtt{true}$ holds always;

[4] The notation <_> is generally used in the framework of modal logics to denote the strong next modality, while ≪_≫ is used for the weak next modality. Here we adopted instead the ACTL-like notation [3], where the strong next is denoted by EX and the weak next by <_>. This notation is more convenient in this case, since π-logic formulae will be mapped into ACTL formulae.

- $P \models {\sim}\phi$ 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 $P' \models \phi$, $\mu = \texttt{tau}, x!y, x?y$;
- $P \models EX\{x!(y)\}\phi$ if and only if there exist P' and $w \notin fn(\phi) - \{y\}$ such that $P \xrightarrow{x!(w)} P'$ and $P' \models \phi\{w/y\}$;
- $P \models <\mu>\phi$ if and only if there exist $P_0, \ldots, P_n$, $n \geq 1$, such that $P = P_0 \xrightarrow{\texttt{tau}} P_1 \ldots \xrightarrow{\texttt{tau}} P_{n-1} \xrightarrow{\mu} P_n$ and $P_n \models \phi$, $\mu = \texttt{tau}, x!y, x?y$;
- $P \models <x!(y)>\phi$ if and only if there exist $P_0, \ldots, P_n$, $n \geq 1$, and $w \notin fn(\phi) - \{y\}$ such that $P = P_0 \xrightarrow{\texttt{tau}} P_1 \ldots \xrightarrow{\texttt{tau}} P_{n-1} \xrightarrow{x!(w)} P_n$ and $P_n \models \phi\{w/y\}$;
- $P \models EF\,\phi$ 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 $P_n \models \phi$.

Some derived operators can be defined:

- `false` stands for ${\sim}$`true`;
- $\phi \mid \phi'$ 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.

Theorem 10. *π-logic is adequate with respect to strong early bisimulation equivalence.*

Proof: In [15] it has been shown that the subset of the logic given by $\phi ::= \texttt{true} \mid {\sim}\phi \mid \phi \;\&\; \phi' \mid EX\{\mu\}\phi$ is adequate with respect to strong early bisimulation equivalence. It is easy to see that strongly equivalent agents satisfy the same set of weak logic formulae. Moreover, the eventually operator simply states that something will happen in the future, hence strongly equivalent agents satisfy also the same set of *eventually* logic formulae. As a consequence of these facts, the introduction of the weak next modalities and of the eventually operator only adds expressive power to the logic, without changing its adequacy with respect to strong early bisimulation equivalence.

It has to be noticed that the π-logic is a subset of the Propositional μ-calculus with name-passing [2]. The logic presented in [2] is an extension of the μ-calculus [12] in which it is possible to express name parameterizations and quantifications over the communication objects. It permits a broad class of properties of systems described by π-calculus agents to be expressed. The π-logic, with its simple syntax, can however cover an interesting class of properties of mobile systems without using fixed point operators. Moreover, as we will show in the following of this paper, an efficient model checking algorithm can be given for it.

4 A model checker for the π-calculus

Modal and temporal logics are provided of techniques for verification to show when systems have or fail to have certain properties expressed in a given logic. In the case of finite state systems a popular approach is to define an algorithm called model checker. Efficient automatic verification environments exist that exploit model checking theories starting from the first model checker, built for the CTL logic [1]. By using this approach, the complexity of the model checking problem is a function of the size of the system (number of states and transitions of its corresponding model) and of the size of the formula. For CTL and ACTL there are straightforward linear (in the size of the system and of the formula) time model checking algorithms. As more complex temporal logics are used, e. g. μ-calculus, more complex algorithms are developed. In general, these algorithms are exponential in the alternation depth of the formula [9].

In the case of infinite state systems automata-based model checking techniques reveal inadequate to check the satisfiability of systems properties. In this case, several decision procedure have been defined that rely on local methods [19], that are often presented using tableaux, or on the symbolic (or abstract) representation of the state space [10].

Agents defined over π-calculus are usually associated with infinite state models. Therefore, automatic verification strategies defined for them rely on the use of tableaux systems. The *Mobility Workbench* [20] (MWB in short) is the most popular tool for the verification of behavioural and logical properties for the π-calculus. The logic verification functionalities offered by the MWB are based on the implementation of the tableau-based proof system for the Propositional μ-calculus with name-passing [2]. Inference rules are applied to establish if an agent satisfies (or does not satisfy) a given logic formula.

In this paper, we will provide a classical model checking algorithm for checking the satisfiability of the π-logic formulae on π-calculus agents. This is made possible by exploiting the results in [16] that permit, given a finitary π-calculus agent a finite state LTS to be derived. The model checker for the π-logic is, then, built using this finite representation and re-using the existing model checker for the ACTL logic [5] by means of a sound translation function from the π-logic into ACTL.

5 From π-calculus agents to Labelled Transition Systems

In [16] it has been proved that finite state LTSs 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.

In order to associate to a π-calculus agent P a finite state LTS, a suitable finite set of object names is considered for representing all the bound output and input transitions of the agent. This set is defined by taking into account the set $an(P)$ of the semantically *active* names, or simply active names, of the agent.

The set $an(P)$ is a subset of the free names of the agent, that can be seen as syntactically active names. Formally, a name x is active in P, that is $x \in an(P)$, if and only if $(x)P \not\simeq P$. Active names have the nice property that bisimilar agents have the same set of active names. Notice that this is not true for free names, as the Example 1 shows. Informally, the names belonging to $an(P)$ are the free names that are used as subjects of the actions, or as objects of the free output actions, performed by the agent.

By assuming a total order among the names in $\mathcal{N}$, only one transition is used, in the LTS associated to the agent P, to represent the infinite set of transitions corresponding to a bound output $x!(y)$: it suffices to choose as label of the transition the string $x!(a)$, where a is the minimal element of the set of names given by $\mathcal{N} - an(P)$. Assume that P and Q are bisimilar agents, and that both P and Q can perform a bound output action with subject x. Since bisimilar agents have the same set of active names, these bound output actions are mapped into the same transition label $x!(a)$ of the corresponding LTSs.

The representation, in the LTS, of the infinite set of transitions corresponding to an input action $x?y$, performed by P, is done in a similar way: in the LTS there is a finite set of transitions labelled by strings like $x?u$, where u is an active name of P, along with a transition labelled with $x?(a)$, where a is minimal element of $\mathcal{N} - an(P)$.

The names chosen to represent, in the LTS of P, non active names, will be called in the following the *fresh names* of the LTS.

Example 1. Consider the bisimilar agents $P = x?(y).\texttt{nil}$ and $Q = x?(y).\texttt{nil} + (w)w!z.\texttt{nil}$. The two agents have different sets of free names, $\{x\}$ and $\{x, z\}$ respectively. However, they have the same set of active names $\{x\}$. Actually, the name z is a free name for Q but it is not a semantically active name in Q. In fact, we have that $(z)Q \simeq Q$.

Although it is in general undecidable whether $(x)P \simeq P$, in [16] algorithms for computing the active names and the finite state LTSs of finitary agents without matching[5] are given. These algorithms ensure that equivalent (i.e. strong or weak bisimilar) π-calculus agents are mapped into equivalent (i.e. strong or weak bisimilar) LTSs. The complexity of the construction of the state space of the π-calculus agent has a worst case complexity that is exponential in the syntactical size of the agent [16].

We do not give here a complete account of the algorithms. Instead, we give the description of the finite state LTS associated with an agent. From now on $\Downarrow P$ denotes the restriction $(y_1)\dots(y_n)P$, where $\{y_1,\dots,y_n\} = fn(P) - an(P)$. Notice that $fn(\Downarrow P)$ coincides with $an(P)$.

Definition 11. Let P_0 be a π-calculus agent. The LTS associated with P_0 is $(Q, \Downarrow P_0, Act \cup \{\texttt{tau}\}, \longmapsto)$, where $Q, Act, \longmapsto$ are defined by the following rules. For $P \in Q$:

[5] The same algorithm also works for the π-calculus with a restricted form of matching [17] that is general enough for most of the practical applications.

- if $P \xrightarrow{\texttt{tau}} P'$ then $\Downarrow P' \in Q$ and $\Downarrow P \overset{\texttt{tau}}{\mapsto} \Downarrow P'$;
- if $P \xrightarrow{x!y} P'$ then $\Downarrow P' \in Q$, $x, y! \in Act$ and $\Downarrow P \overset{x!y}{\mapsto} \Downarrow P'$;
- if $P \xrightarrow{x!(y)} P'$ and $y = min(\mathcal{N} - fn(P))$ then $\Downarrow P' \in Q$, $x!(y) \in Act$ and $\Downarrow P \overset{x!(y)}{\mapsto} \Downarrow P'$;
- if $P \xrightarrow{x?y} P'$ and $y \in fn(P)$ then $\Downarrow P' \in Q$, $x?y \in Act$ and $\Downarrow P \overset{x?y}{\mapsto} \Downarrow P'$;
- if $P \xrightarrow{x?y} P'$ and $y = min(\mathcal{N} - fn(P))$ then $\Downarrow P' \in Q$, $x?(y) \in Act$ and $\Downarrow P \overset{x?(y)}{\mapsto} \Downarrow P'$.

Notice that the set of the fresh names of the LTS can be easily recognized since the fresh names are those appearing enclosed in curly brackets in the transition labels of the LTS.

Example 2. Consider again the agents P and Q of the Example 1. They have associated the same LTS (see Figure 1), with two states, say s_0 and s_1, and two transitions $s_0 \overset{x?x}{\mapsto} s_1$ and $s_0 \overset{x?(a)}{\mapsto} s_1$, where we have assumed the name $a = min(\mathcal{N} - \{x\})$.

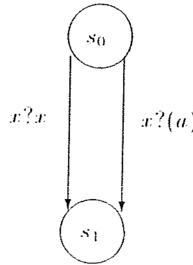


Fig. 1. The LTS of P and Q.

6 From π-logic to ACTL

In this Section we define the translation function that associates with each formula of the π-logic an ACTL formula. The translation we propose is defined by having in mind a precise soundness result: we want that a π-logic formula is satisfied by a π-calculus agent P if and only if the finite state LTS associated with P satisfies the corresponding ACTL formula. As a consequence, the translation of a formula is not unique, but it depends on the agent P. To be more precise, it depends on the set S of the fresh names of the LTS associated with the agent P.

The translation of the (strong or weak) next modalities in which the action arguments are free output or `tau` actions is immediate: it is given by the corresponding (strong or weak) ACTL next operator in which the action label is just the string corresponding to the action argument of the π-logic next modality to be translated.

The translation of the (strong or weak) next modalities in which the action arguments are input or bound output actions is more complex, as it depends on the given set of fresh names S. When the translation of a π-logic next bound output or input modality is considered, the action label of the corresponding ACTL next modality cannot be simply the action argument[6] of the π-logic next modality. In fact, we have to consider that a bound output or input action μ, performed by the π-calculus agent P, is represented in the LTS of P by labels that can be different from μ. Actually, whenever a π-logic next bound output modality has to be translated, we have to map its action argument, say $x!(y)$, to all the possible transition labels of the LTS of P that can represent $x!(y)$, that is to labels of the form $x!(s)$, where $s \in S$. When a π-logic next input modality has to be translated, two possibilities have to be taken into account. The first one is that an explicit transition labelled by the action argument of the π-logic next modality, say $x?y$, may exist (this is the case if y is an active name of the agent P). The second one is that it is possible that in the LTS no explicit transitions exist labelled by $x?y$. This means that we have to map $x?y$ also to all the possible labels that can represent $x?y$ in the LTS, that is to labels of the form $x?(s)$, where $s \in S$.

The translation of the π-logic formula $EF\,\phi$ is given simply by the ACTL formula $EF\,\phi'$, where ϕ' is the ACTL translation of ϕ.

Definition 12. Let $\theta = \{\alpha/y\}$. We define $\mu\theta$ be the action μ' obtained from μ by replacing the occurrences of the name y by the name α. Moreover, we define $\mathtt{true}\theta = \mathtt{true}$, $(\phi_1 \& \phi_2)\theta = \phi_1\theta \& \phi_2\theta$, $(\sim \phi)\theta = \sim \phi\theta$, $(EX\{\mu\}\phi)\theta = EX\{\mu\theta\}\phi\theta$, $(< \mu > \phi)\theta = < \mu\theta > \phi\theta$ and $(EF\phi)\theta = EF\phi\theta$.

Definition 13 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:

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

[6] Notice that in the action argument of a π-logic next modality we can distinguish the type of the action (silent, output, bound output, input) and eventually a subject and an object. On the contrary, the action labels of the ACTL next modalities are simply labels, that is string of characters.

- $\mathcal{T}_S(< x!y > \phi) =< x!y > \mathcal{T}_S(\phi)$
- $\mathcal{T}_S(< x!(y) > \phi) = |_{\alpha \in S} < x!(\alpha) > \mathcal{T}_S(\phi\theta)$, where $\theta = \{\alpha/y\}$
- $\mathcal{T}_S(< x?y > \phi) =< x?y > \mathcal{T}_S(\phi) \mid |_{\alpha \in S} < x?(\alpha) > \mathcal{T}_S(\phi\theta)$, where $\theta = \{\alpha/y\}$
- $\mathcal{T}_S(EF\phi) = EF\mathcal{T}_S(\phi)$

Notice that the complexity of the translation has a worst case complexity that is exponential in the number of the names appearing in the set S.

Example 3. Consider again the agent P of the Example 1. The agent P satisfies the π-logic formula $\phi = EX\{x?u\}\mathtt{true}$ for each name u, since $P \xrightarrow{x?u} \mathtt{nil}$ for each name u. We want to verify if the ACTL translation of the formula holds in the LTS 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 LTS of P (see Figure 1), that is $\{a\}$. Hence, the translation of the formula is:

$$EX\{x?u\}\mathtt{true} \mid EX\{x?(a)\}\mathtt{true}$$

Notice that the obtained ACTL formula holds in the LTS of P.

Theorem 14. *Let P be a π-calculus agent, with associated LTS $\mathcal{A}$ and ϕ be 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 $\mathcal{A}$.*

Proof sketch: In order to show the theorem, we use structural induction on ϕ. Since the logic has the negation operator, we have only to show one implication. We will show that whenever $P \models \phi$ then $\Downarrow P \models \mathcal{T}_S(\phi)$.
We will prove the theorem for $\phi = EX\{x?y\}\phi'$, the other cases are easier. We recall that $\mathcal{T}_S(EX\{x?y\}\phi') = EX\{x?y\}\mathcal{T}_S(\phi') \mid |_{\alpha \in S} EX\{x?(\alpha)\}\mathcal{T}_S(\phi'\{\alpha, y\})$.
Assume that $P \models EX\{x?y\}\phi'$. Then, we have that there exists P' such that $P \xrightarrow{x?y} P'$ and $P' \models \phi$. By induction, we get $\Downarrow P' \models \mathcal{T}_S(\phi)$. Now, we have to consider two cases, i.e. $y \in an(P)$ and $y \notin an(P)$.

case $y \in an(P)$
If $y \in an(P)$ then $\Downarrow P \overset{x?y}{\longmapsto} \Downarrow P'$. Hence, $\Downarrow P \models EX\{x?y\}\mathcal{T}_S(\phi')$, that implies $\Downarrow P \models \mathcal{T}_S(EX\{x?y\}\phi')$.
case $y \notin an(P)$
In this case, the LTS does not contain an explicit input transition labelled by $x?y$ from $\Downarrow P$. Instead, we have that there exists the transition $\Downarrow P \overset{x?(\beta)}{\longmapsto} \Downarrow P''$, where $\beta = min(\mathcal{N} - an(P))$, $P \xrightarrow{x?\beta} P''$ and $P'' = P'\{\beta/y\}$. We will show that $\Downarrow P \models EX\{x?(\beta)\}\mathcal{T}_S(\phi'\{\beta/y\})$. In order to show this, we have to show that $\Downarrow P'' \models \mathcal{T}_S(\phi'\{\beta/y\})$. Hence, we reduce to prove that $\Downarrow P'\{\beta/y\} \models \mathcal{T}_S(\phi'\{\beta/y\})$. To do this we can prove (by using structural induction on ϕ) that $P' \models \phi'$ implies $P'\{\beta/y\} \models \phi'\{\beta/y\}$, and then apply the induction hypothesis.

7 A verification example

In this Section we show an example of verification of some π-logic properties of a system specified by a π-calculus agent.

The system has been inspired by the principles used by a new generation of Web browsers, as that described in [7]. Differently from other Web browsers, that have a static knowledge of Internet data, protocols and behaviours, the browser we want to specify knows essentially none of them. However, it is able to dynamically encrease its capabilities by means of a (transparent) software migration across the network. When a request is raised by an user, the special software applications needed to display the data are automatically installed in the user's system. This allows content developers to feel free of adding new features to their programs, without the need of providing new browsers and servers with added capabilities. Moreover, the dynamic loading of the used protocols allows the end user to use a unique Web browser, instead of selecting some specialized browser able to access the data. The browser will search the target system for the software code to add to the local system in order to correctly serve the user request.

We will give the system specification by using the π-calculus formalism. Actually, π-calculus permits to describe in a suitable way the dynamic behaviour of the system. The system specification consists of two modules: the *browser* and the *protocol-handler*. The browser receives via the channel *local* the user request, that is the name of the *host* and the name of the desired *object*. Then, the *browser* sends to the *host* a request to access the given object. The *host* communicates to the *browser* the *class* of the protocol to be used to correctly display the data. This information is communicated by the *browser* to the *protocol-handler*, via the internal channel *req*. The *protocol-handler* can send to the *host* the request to get the *protocol* code to be loaded on the local system. After the protocol has been taken, the *protocol-handler* can load it on the local system. The object *data*, sent by the host and received by the *browser*, are then sent by the *browser* to the local system that is now able to process them correctly. In the following the π-calculus specification of the system is reported.

```
parseterm protocol-handler(req, ld) := (req?(host). req?(class).
  host!class. host?(protocol). ld!protocol. req!protocol.
  protocol-handler(req, ld)) endterm

parseterm browser(local,req) := (local?(host). local?(object).
  host!object. host?(class). req!host. req!class. req?(ack).
  host?(data). local!data. browser(local,req)) endterm

parseterm system(req,ld,local) := (req) (protocol-handler(req, ld)
  ||  browser(local,req)) endterm
```

Some of the formulae that we want to verify on the system to guarantee its correct behaviour are:

1. Whenever a request for the object o, located on the host h, is raised by the local system, then eventually the host will send a protocol p that will be loaded on the local system:
   $AG([local?h][local?o]EF < h?p >< ld!p > true)$
2. Whenever a host name h is communicated by the local system, it is always true that if the host communicates a protocol p and the protocol is loaded on the local system, then some data d are received from h and sent to the local system:
   $AG([local?h]AG([h?p][ld!p] < h?d >< local!d > true))$
3. It is eventually possible that a host name h is communicated by the local system and then it is eventually possible that h communicates a protocol p that is loaded on the local system. Then some data d will be received from h and sent to the local system:
   $EF(< local?h > EF(< h?p >< ld!p >< h?d >< local!d > true))$

In order to do model checking of these properties on the system, we have, as a first step, to generate the LTS associated with the system. In order to do this we use the tool described in [4]. The LTS obtained by performing this step has 13 states and 17 transitions and it is shown in Figure 2. Notice that the set of the fresh names of the LTS is $\{\sharp 0, \sharp 1\}$.

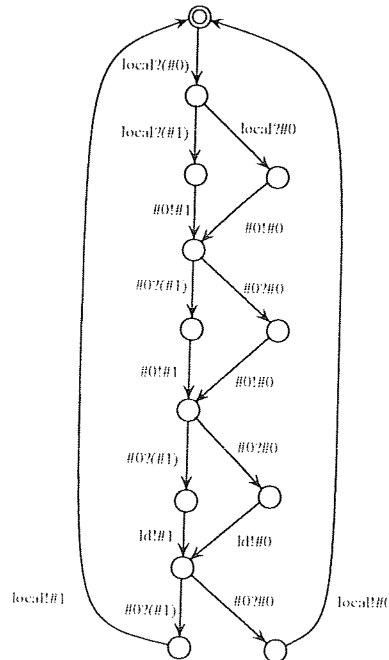


Fig. 2. The LTS associated with the system.

The next step we perform is the translation of the π-logic formulae into ACTL formulae. To do this we give in input to the automatic formulae translator that we have implemented the π-logic formulae and the LTS on which the translated formulae have to be interpreted. The translator, after the capturing of the information about the fresh names used in the LTS, translates the given formulae into the corresponding ACTL ones.

The resulting ACTL formulae can then be checked on the LTS, by using the ACTL model checker, AMC [5]. All the formulae result to be true in the LTS, as we expected.

8 Conclusions

The proposed model checker for the verification of logical properties of π-calculus agents, presented in this paper, exploits automata-based approaches to the logic verification. Once the finite LTS of a π-calculus agent has been generated [16, 4], efficient model checking (in the style of [1]) of π-logic formulae can be performed. The model checker of π-logic formulae has been realized by implementing a logic translator from π-logic formulae into ACTL formulae, and by using tools that have been integrated in the JACK verification environment [6, 4]. The JACK environment, that combines different specification and verification tools, permits to generate the LTS associated with a given π-calculus agent [4], and to reduce its size by using standard automata minimization tools. The model checking of π-logic formulae can be done, after a suitable translation into ACTL formulae, by using the AMC model checker for the ACTL logic, inside JACK.

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.

Some comparisons have to be done between the logic verification functionalities offered by the proposed model checker and those offered by another existing tool for the verification of logic formulae for the π-calculus: the *Mobility Workbench* [20]. The main difference between our approach and that adopted in the MWB is that in our environment the state space of a π-calculus agent is built once and for all, and then used 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. As a future development, we are studying model checking techniques, based on the same approach we have adopted for the verification of π-logic formulae, for the verification of properties expressed by means of a more powerful logic for π-calculus.

References

1. E.M. Clarke, E.A. Emerson, A.P. Sistla. 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.

2. M. Dam. Model checking mobile processes. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
3. 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.
4. G. Ferrari, G. Ferro, S. Gnesi, U. Montanari, M. Pistore, G. Ristori. An automata based verification environment for mobile processes. *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*, Enschede, NL, April 2-4 1997.
5. G. Ferro. AMC: ACTL Model Checker. Reference Manual. *IEI-Internal Report, B4-47*, 1994.
6. S. Gnesi. A formal verification environment for concurrent systems design. In *Proc. Automated Formal Methods Workshop*, ENTCS 5. Elsevier, 1996.
7. J. Gosling, H. McGilton. The Java Language Environment. A White Paper. *Sun Microsystems*, May 1996.
8. E. A. Emerson, J. Halpern. "Sometime" and "Not never" revisited: On branching versus linear time temporal logic. *JACM 33*, 1986, pp. 151-178.
9. E. A. Emerson, C. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In: *Proceedings of Symposium on Logics in Computer Science* (1986) 267-278.
10. M. Hennessy, H. Lin. Symbolic Bisimulations. *Dept. of Computer Science, University of Sussex*, 1/92, 1992.
11. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM* 32 (1), pp. 137-161, 1985.
12. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science* 27, pp. 333-354, 1983.
13. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
14. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
15. R. Milner, J. Parrow, D. Walker. Modal logic for mobile processes. In *Proc. CONCUR'91*, LNCS 527. Springer Verlag, 1992.
16. U. Montanari, M. Pistore. Checking bisimilarity for finitary π-calculus. In *Proc. CONCUR'95*, LNCS 962. Springer Verlag, 1995.
17. U. Montanari, M. Pistore. History Dependent Automata. To appear as Technical Report, Department of Computer Science, University of Pisa.
18. D. Sangiorgi. Expressing mobility in process algebras: first-order and higher-order paradigms. PhD Thesis CST-99-93, University of Edinburgh, 1992.
19. . C. Stirling. D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89, 161-177
20. B. Victor and F. Moller. The Mobility Workbench — A tool for the π-calculus. In *Proc. CAV'94*, LNCS 818. Springer Verlag, 1994.
21. D. Walker. π-calculus semantics for object-oriented programming languages. In *Proc. TACS'91*. Springer Verlag, 1995.

This article was processed using the LaTeX macro package with LLNCS style