Action-based Model Checking

(and its applications to distributed, mobile, object-oriented systems)

A Tutorial

A. Fantechi, DSI - University of Florence, Italy

S. Gnesi, ISTI – CNR, Pisa, Italy

FMOODS 2003

Introduction

This tutorial aims to present the possibilities offered by model checking tools for the verification of distributed, mobile, object-oriented systems, modelled by formalisms derived by ***process algebras.***

Outline of the tutorial

First part:

- *action-based logics interpreted over LTSs,*
- *discussion about their expressive power*
- *relations with process algebras.*
- *model checking algorithms and tools*

Second part:

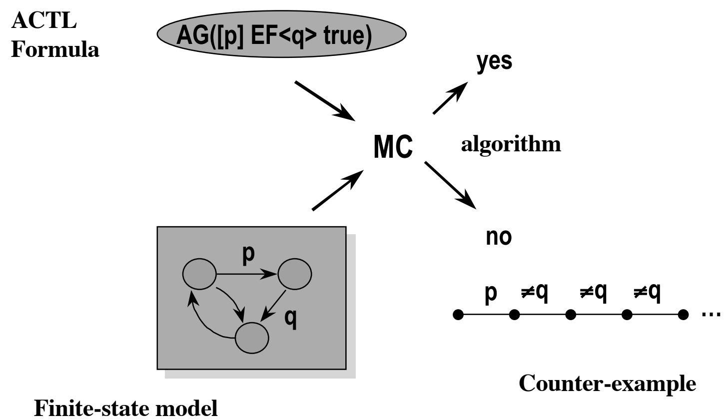
- *examples of applications*
- *infinite state systems*
- *extensions to mobility*
- *…and what about UML????*

Model checking

Model Checking:
"checking that a given structure is a model for a given logical formula".

in practice:
- specification language based on finite state automata, process algebras, Petri nets
- model checking: proving that a formula in temporal logic (expressing a desired property) is satisfied by the automaton (or automata) that specifies the system

Model Checking (Clarke/Emerson, Queille/Sifakis- 1981)



States or actions?

CTL, the branching time temporal logic most used in model checking (EMC, SMV,…) is based on predicates on the states

CTL models are Kripke Structures, that is transition systems where the states are labelled by a set of atomic predicates

LTL, the linear time temporal logic used by SPIN has linear models: execution traces, sequences of states labelled by atomic predicates

Success of model checking techniques in hardware technology: a state is a bit vector

States or actions?

Model checking techniques in software technology: states are variables'values.

But often many aspects of software (and the most interesting ones for the classes of reactive, concurrent, distributed software) are often seen as events, and event-based transistion systems are often used as models

Process algebras have been recognized, since two decades, as a useful mean to model the behaviour of a system, abstracting from data and functions.

States or actions?

In process algebras, a system is seen as a set of processes, and each process is characterised by the actions performed in the time by the process.

Hence a model checker focusing on actions, rather than states, is able to address the whole word of systems commonly modelled with process algebras, and with languages derived by process algebras. These systems include concurrent and distributed systems, mobile systems, and cover the behavioural aspects of object-orented systems.

The ACTL logic: syntax

ACTL Syntax --> **□ ::= true | ~ □| □□□ | □□| □□**

(state formulae)

□::= X□□ | X□□ | □ □U □ □' | □ □U □'

(path formulae)

Action formulae --> **□ ::= □ | ~ □| □□□ | □| □**

- ***ACTL Semantics***

  The satisfation of an ACTL formula is inductively defined over labelled transition systems

Preliminary definitions

A labelled transition systems (LTS) is a 4-tuple A=(*S*, *s0*, *Act*, □), where:

***S* is a finite set of states; *s0* is the initial state;**

***Act* is a finite set of observable actions;**

□ □ S□*Act*□S is the transition relation between states.

We denote by *s -a*□ *s*', *a* □ *Act*, the transition from the state *s* to the state *s'* by executing *a*; in particular, *s -a*□ *s'* indicates that a system in state *s* can perform a transition to state *s'* by executing the action *a*.

D_a(s) = {s' | s -a□ s'} set of successors of s

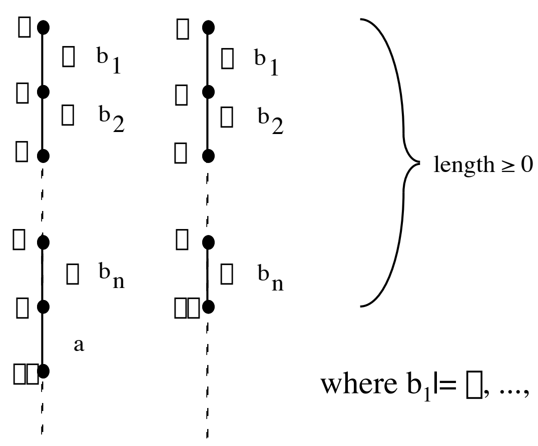
□(s) is the set of paths starting from s; a path is a sequence of successive transitions.

ACTL semantics

$s \models_{TS} true$		always;
$s \models_{TS} \phi \wedge \phi'$	iff	$s \models_{TS} \phi$ and $s \models_{TS} \phi'$;
$s \models_{TS} \phi \vee \phi'$	iff	$s \models_{TS} \phi$ or $s \models_{TS} \phi'$;
$s \models_{TS} \neg\phi$	iff	not $s \models_{TS} \phi$;
$s \models_{TS} E\gamma$	iff	there exists a path $\pi \in \Pi(s)$ such that $\pi \models_{TS} \gamma$;
$s \models_{TS} A\gamma$	iff	for all paths $\pi \in \Pi(s)$, $\pi \models_{TS} \gamma$;
$\pi \models_{TS} X_\chi \phi$	iff	$\lvert\pi\rvert \geq 1$ and $\pi(1) \in D_{\kappa(\chi)}(\pi(0))$ and $\pi(1) \models_{TS} \phi$;
$\pi \models_{TS} X_\tau \phi$	iff	$\lvert\pi\rvert \geq 1$ and $\pi(1) \in D_{\{\tau\}}(\pi(0))$ and $\pi(1) \models_{TS} \phi$;
$\pi \models_{TS} \phi\, {}_\chi U \phi'$	iff	there exists $i \geq 1$ such that $\pi(i) \models_{TS} \phi'$, and for all $1 \leq j \leq i-1$: $\pi(j) \models_{TS} \phi$ and $\pi(j+1) \in D_{\kappa(\chi)_\tau}(\pi(j))$;
$\pi \models_{TS} \phi\, {}_\chi U_{\chi'} \phi'$	iff	there exists $i \geq 2$ such that $\pi(i) \models_{TS} \phi'$ and $\pi(i) \in D_{\kappa(\chi')}(\pi(i-1))$, and for all $1 \leq j \leq i-1$: $\pi(j) \models_{TS} \phi$ and $\pi(j) \in D_{\kappa(\chi)_\tau}(\pi(j-1))$.

Action indexed Untils

□ □U□' □'   □ □U □'



where b_1|= □, ..., b_n |= □ and a |= □'

Derived modalities

Several useful modalities can be defined, starting from the basic ones.

$EF\phi$ for $E(tt_{tt}U\phi)$, and $AF\phi$ for $A(tt_{tt}U\phi)$;

these are called the *eventually* operators

$EG\phi$ for $\neg AF\neg\phi$, and $AG\phi$ for $\neg EF\neg\phi$;

these are called the *always* operators

Hennessy-Milner "weak" modalities, [ ] < >:

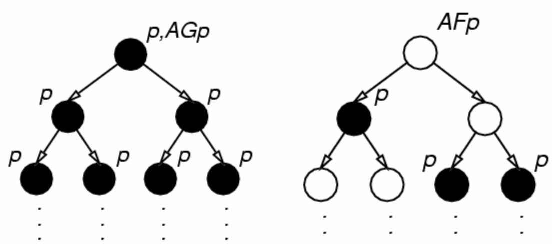
$\langle a\rangle\phi = E(tt_{ff}U_a\,\phi)$ ***(There exists a visible transition labelled by a)***

$[a]\phi = \neg\langle a\rangle\neg\phi$ ***(For all transitions labelled by a,….)***

Universal derived modalities

$AG\ p$ "globally p"

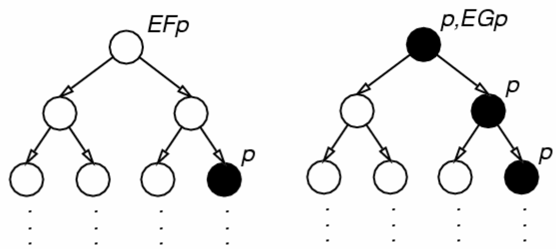
$AF\ p$ "inevitably p"



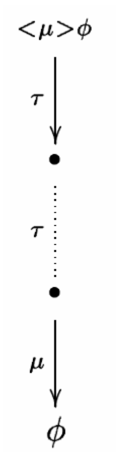
Existential derived modalities

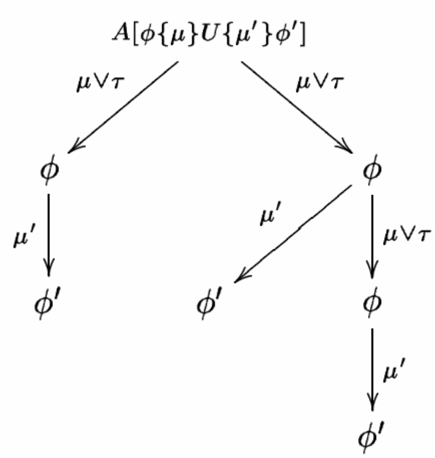
$EF\ p$ "possibly p"

$EG\ p$ "?"



Other examples





Safety and liveness properties

Classical distinction of properties of reactive systems:

- *Liveness properties* (something good eventually happens)
- *Safety properties* (nothing bad can happen)

$$AF\ EX_{good}\ true$$

$$AG\ \neg EX_{bad}\ true \quad (= \neg EF\ EX_{bad}\ true\ )$$

Relations between ACTL and process algebras

A logic L is adequate with respect to an equivalence $\sim$, if for every pair of processes q and q', q $\sim$ q' holds if and only if q and q' satisfy the same set of ACTL formulas.

The ACTL logic is adequate with respect to strong bisimulation equivalence on LTSs

Therefore minimization by strong bisimulation preserves ACTL formulae.

Relations between ACTL and process algebras

Distinctive feature of ACTL and of Process algebras (vs. CTL/Kripke Structures): explicitation of the unobservable action Tau
(in Kripke Structure internal moves are modelled by stuttering)

The ACTL-X (ACTL without next) logic is adequate with respect to branching bisimulation equivalence on LTSs

Variants of ACTL and other action-based logics

"Machine readable" syntax

E[□{□}U{□'}□'] instead of E(□ □U□ □')
EX{□}true instead of Ex□ true

Strong version of HML modalities:

<a>□ = EX_a □ [a]□ = ¬ <a> ¬ □

Unless [Meolic]

E[□{□}W{□'}□'] = E[□{□}U{□'}□'] | EG □{□}
A[□{□}U{□'}□'] = A[□{□}W{□'}□'] & AF {□'}□'

Eventually an action

EF{□}□ = E[true{true}U{□}□]

Variants of ACTL and other action-based logics

Expressive power of ACTL w.r.t. CTL:

CTL have more expressive power than ACTL (CTL can predicate over conjunctions of actions labeling a transition)

Expressive power w.r.t. mu-calculus:

ACTL properties can be expressed in mu-calculus

Variants of ACTL and other action-based logics

□-ACTL is an extension with a fixed point operator of ACTL

□-ACTL embeds the idea of "evolution in time by actions"

□ ::= true | ~ □ | □&□ | E□ | A□ | <a> □ | □Y.□ (Y)

□ ::= X{a}□ | T□ | □ {a} U {a'} □' | □ {a} U□'

The satisfation of a □-ACTL formula is defined inductively over a Labelled Transition System

Relations between ACTL and process algebras

- **□-ACTL is adequate w.r.t. strong bisimulation □**

  A logic L is adequate with respect to □, if for every pair of processes q and q', q □ q' holds if and only if q and q' satisfy the same set of ACTL formulas.

- **□-ACTL is able to express safety, liveness and cyclic properties of concurrent systems.**
- **□-ACTL has the same expressive power than □-calculus**
- **□-ACTL is expressive w.r.t. strong bisimulation.**

Basic model checking algorithm

We present a model checking algorithm for (a subset of) ACTL, directly derived by the Clarke-Emerson-Sistla 1986 algorithm

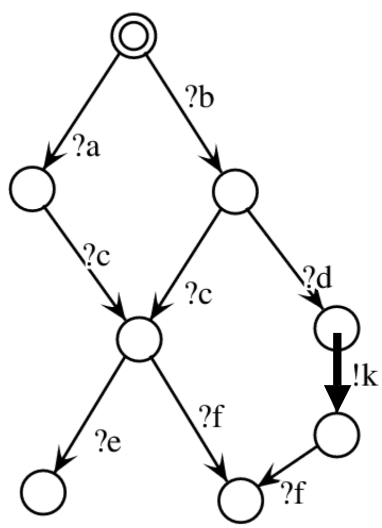
The algorithm proceeds visiting the automaton and labelling each state with the set of subformulae of the formula to be checked, that are satisifed in that state *(following the satisfaction relation |= used for defining the semantics of the logic).*

The subset considered is composed of state formulae:

P := true | ~P | EX{act}P | E[P{act}U P]

(Explicit) Model checking algorithm

```
for i =1 to length(p0)
 for each subformula p of p0  of length i
  case on the form of p
    p =  true    /* nothing to do */
    p  = q and  r:    for each s  in  S
        if q in L(s) and r in L(s) then add q  and  r to L(s)
        end
    p = ~q:             for each s in S
                           if q in L(s) then add  ~q to L(s)
                        end
    p = EX{a}q:         for each s in S
                        if (for some t in S, s-a→t, q in L(t)) then add EX{a}q toL(s)
                        end
    p = E[q {a}U r]:    for each s in S
                        if r in L(s)
                        then add E[q {a}U r] to L(s)
                        end
                        for j  = 1 to card(S)
                            for each s in S
                                    if  q in L(s) and (for some in S, s-a→t or s-□→t, E[q {a}U r] in L(t))
                                    then add E[q {a}U r] to L(s)
                            end
                        end
    end of case
  end
end
```

Looking for a "bad" transition

Formula expressing:
there exists no path that executes the bad transition.

`~ EF EX{!k} true`

Length = 2

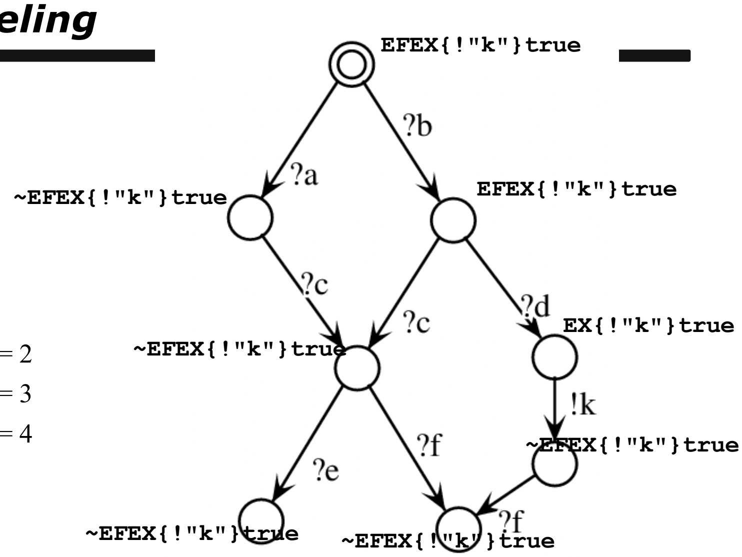
Length = 3

Length = 4

Note that `EF p = E[true{true}U p]`

Labeling

Length = 2
Length = 3
Length = 4



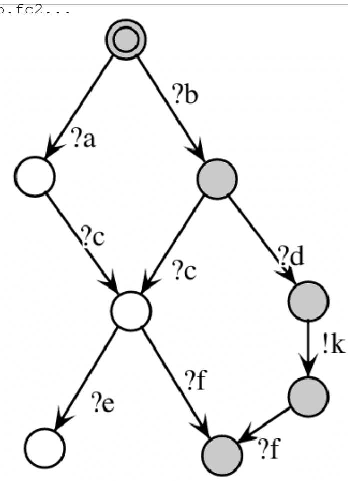
Counterexample

- The *labeling* alghoritm permits, in the caso of negative result, to find out the reason of the result, since it is possible to find all those states that do not verify some significant subformula, and hence contribute to the failure of the verification;

in general, model-checking alghoritms are able to provide a *counterexample*: for example, a path in the model that does not verify the (sub)formula. ☐

Counterexample generation with AMC

```
top >  load "/home/fantechi/jackov/Tzeroprimo.fc2"
Taking input from /home/fantechi/jackov/Tzeroprimo.fc2...
time: (user: 0.00 sec, sys: 0.00 sec)

top >  eval
Evaluate mode. Graph 0

|=  ~EF  EX{!k} true

The formula is FALSE in state 0 time: (user: 0.0(


|=  why
why: (~ EF EX {!"k"} true) false 0

    0 :(EF EX {!"k"} true)(1) is true

choice ?
|=  why
why: (EF EX {!"k"} true) true 0

    0 :
   |
   | labelled by : ?1
   |
    2 :
   |
   | labelled by : ?3
   |
    4 :(EX {!"k"} true)(1) is true
```



Computational Complexity

Computational Complexity of this algorithm is in the worst case (Until formulae):

O(length(p0) * card(S) * (card(S)+card(R)))

where card(S) is the number of states and card(R) is the number of transitions.

An immediate optimization in the search produces a complexity:

O(length(p0) * (card(S)+card(R))) □

State space explosion

Though the explicit model chcking algorithms are linear in the state space, is this space that often have a challenging size.

Indeed, in concurrent systems, the state space grows ***exponentially*** with the number of independent processes.

The state space is explicitly represented (e.g. by a matrix representing the transition relation), so tools capacity is limited by memory (no more than one million states)

A symbolic representation of the state space has been adopted to address large state spaces.

Symbolic Model Checking

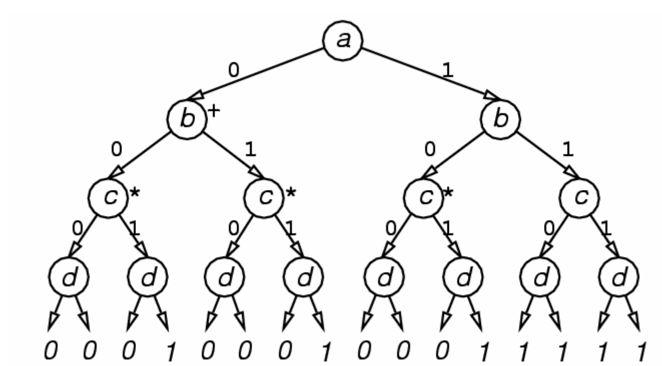
Method used by most “industrial strength” model checkers: uses boolean encoding for state machine and sets of states.

Can handle much larger designs – hundreds of state variables.

BDDs traditionally used to represent boolean functions.

Ordered Decision Tree

Ordered decision tree for the function $ab + cd$:



Ordered Decision Tree

Rules for obtaining the function value:

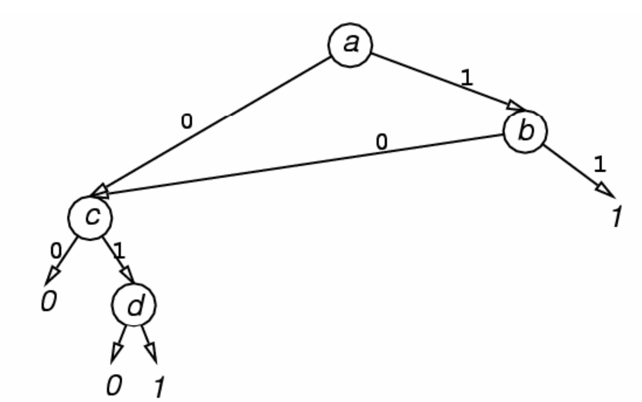
- Start at the root of the tree.
- At each node, take the "0" or "1" branch depending on the truth value of the given boolean variable.
- The leaf you reach gives the truth value of the function.

Requirement: the variables appear in the same order on any path from root to leaf. This requirement of a fixed order is very important for the results that follow.

Note: this tree is just an encoding of the truth table of the function.

Binary Decision Diagram

Binary Decision Diagram for the function $ab + cd$:



Binary Decision Diagram

The BDD is obtained from the ordered decision tree by the following steps:

- Combine isomorphic subtrees.
- Eliminate redundant nodes (nodes with identical children).

BDD encoding

How to represent state-transition graphs with Ordered Binary Decision Diagrams:

Assume that states are encoded by n boolean variables

$v_1, v_2, \ldots, v_n$.

Possible actions are encoded by m boolean variables

$a_1, a_2, \ldots a_m$

The Transition relation T will be given as a boolean formula in terms of the state and action variables:

$T(v_1,\ldots, v_n, a_1, a_2, \ldots a_m, v'_1,\ldots, v'_n)$

Which gives true if there is a transition leaving the current state $v_1,\ldots, v_n$, labelled by action $a_1, a_2, \ldots a_m$, and with next state $v'_1,\ldots, v'_n$

Now convert T to a OBDD!!

Similar encodings can be done for ACTL formulae, through encoding boolean operators, quantifiers, and fixed points.

Model checking then amounts to a few implication-like boolean operators on the BDDs of the LTS and of the formula

The evaluation of this operator is also reduced to a BDD traversal

→ very efficient in terms of memory occupation
→ can handle millions of states easily

Other techniques for dealing with State Explosion

- Use of simmetries in the model
- Abstraction techniques
- Compositional Reasoning (Assume/Guarantee)
- Abstract Interpretation
- LTS Minimization by equivalence
- Partial Order Reduction
- Model checking "On the Fly"(local model checking): *lo stato globale dell'automa non viene costruito completamente prima di applicare l'algoritmo di etichettamento, ma vengono via via generate solo le regioni dell'automa che sono strettamente necessarie a verificare la formula*

ACTL model checkers

AMC - *Explicit model checker for ACTL inside the JACK verification environment (ISTI- CNR)*

FMC - *On the fly model checker for ACTL inside the JACK verification environment (ISTI- CNR)*

Evaluator - *inside the CADP verification environment (INRIA): on-the-fly model-checking of regular alternation-free mu-calculus formulas on LTSs. Regular alternation-free mu-calculus allows direct encodings of ACTL .*

EST - *Efficient Symbolic Tool: BDD- based model checker for ACTL (University of Maribor)*

...

INFINITE STATE PROCESSES

Non-finiteness due to:

- **Data, variables,**
- **inherent behaviour of the system under specification: I.e. Context-free or more.**

we do not build an abstract (with respect to values) model on which the properties are proved.

we define a suitable chain of finite labeled transition systems based on the operational semantics: this allows us to choose the approximation level case by case.

F

CCS process algebra

- ***CCS Syntax*** **--> p::= □.p | nil | p+p | p||p | p\A | x | p[f]**
- ***CCS Semantics*** **--> Operational semantics**
  Labelled transition systems
  (finite-non finite state)
  Behavioural equivalences

Example BAG:

X = p1. (g1.nil | X) + p2.(g2.nil | X) ***non finite state***

p1,p2 insertion of data values 1,2
g1,g2 extraction of data values 1,2

F

ACTL subsets

Finite ACTL

We say that □ is a finite ACTL formula if □ is an ACTL formula without until operators.

Positive ACTL

We say that □ is a positive ACTL formula if □ is an ACTL formula without negations (we admit negations in the action formulae).

The positive finite ACTL subset is defined analogously.

Depth of a finite formula

If □ is a finite ACTL formula, the depth of □ is the maximum number of nested next operators occurring in □

Formulae preserved by preordes

A preorder ≤ over *T* preserves a formula φ if:

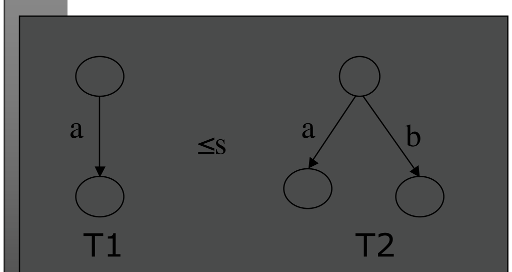
(TS_1 |= φ and TS_1 ≤ TS_2) implies TS_2 |= φ

≤*s*: Simulation preorder does not preserve universal positive formulae

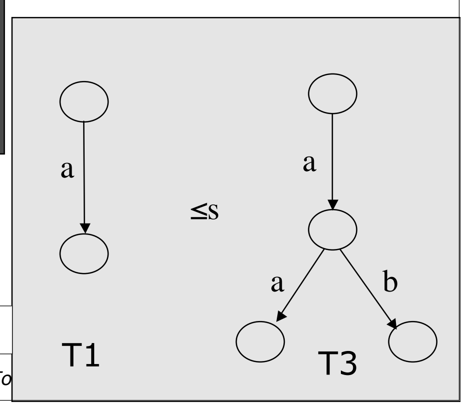
Example:

Each path starts with an action a: $\mathbf{AX_a}$ true

Simulation preorder and ACTL properties



T1 =|AXa true
T2 ≠ |AXa true
T3 =|AXa true



Branching complete-simulation

Let TS_1 and TS_2 be LTSs and let $s_1 \in S_1$ and $s_2 \in S_2$.
We say that s_2 ***BC-simulates*** s_1 if there exists a strong ***BC-Simulation*** that relates s_1 and s_2. $R \subseteq S_1 \times S_2$ is a strong BC-simulation if $(s_1, s_2) \in R$:

- either $s_1 -/\rightarrow$ or

1) $s_1 \xrightarrow{a} s_1'$ implies s_2': $s_2 \xrightarrow{a} s_2'$ and $(s_1', s_2') \in R$
2) $s_2 \xrightarrow{a} s_2'$ implies s_1': $s_1 \xrightarrow{a} s_1'$ and $(s_1', s_2') \in R$

TS_2 ***BC-simulates*** TS_1 ($TS_1 \leq_{bc} TS_2$) if a branching complete simulation R exists such that $(s_{01}, s_{02}) \in R$

Bc-simulation

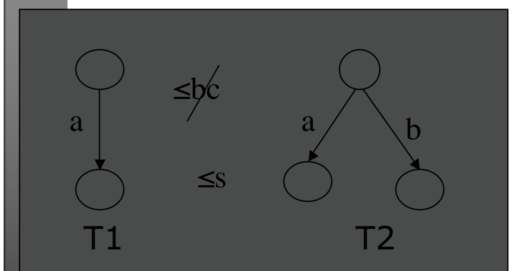
$\leq_{bc}$ preserves the whole positive fragment of ACTL:

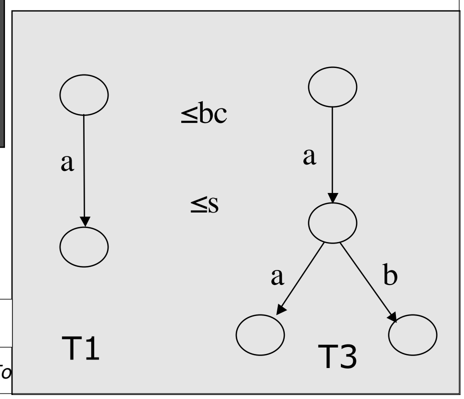
Proposition
Let TS_1 and TS_2 be LTSs and let φ be a Positive ACTL formula:

- ($TS_1 \models \varphi$ and $TS_1 \leq_{bc} TS_2$) implies $TS_2 \models \varphi$

$\leq_{bc}$ is more adequate than $\leq_s$ for proving properties.
The set of formulae preserved by $\leq_s$ is strictly included in the set of formulae preserved by $\leq_{bc}$.

BC-Simulation and ACTL properties





T1 =|AXa true
T2 ≠ |AXa true
T3 =|AXa true

Model checking by approximation chains

To model-check if a CCS description of a system enjoys properties specified as ACTL formulae, its LTS is usually first built

This does not work when the system has a infinite-state representation

Our approach for proving the validity of properties on infinite-state system **consists in checking** the properties **on** the elements of an **"increasing"approximation chain** of finite transition systems, **until the properties are verified.**

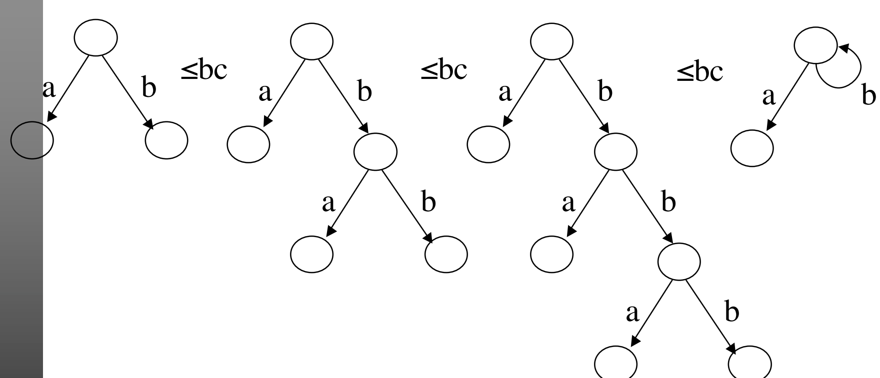
Thus, in general, this is a semi-decision procedure for proving properties.

Bc-simulation and approximation chains

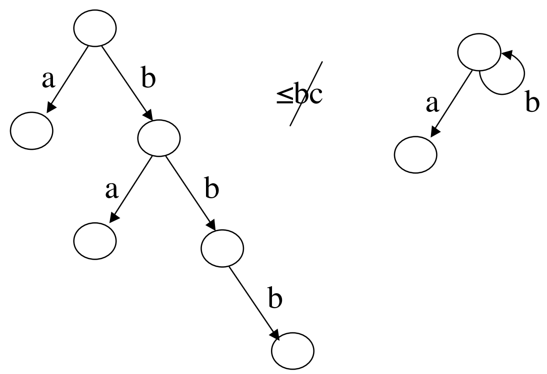
Using the Bc-simulation preorder is it possible to define chains of finite **LTSs that approximates the behavior of possible infinite-state process.**

Each element of the chain is finite and each Ts_i is completely included in Ts_{i+1}.

Approximation chains can be used to define a decision procedure to verify the satisfiability of ACTL formulae on infinite-state processes



Approximating chains



Example: BAG

X= p1. (g1.nil | X) + p2.(g2.nil | X)

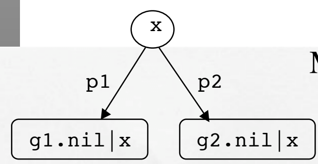
- **The bag is not a set,therefore there exists a computation path on which it is possible to get twice the same value from the bag consecutively: EF<g_1><g_1> true**

- **It is possible, on some (but finitely many) states to do a *put***
- **immediately followed by a *get* action: EFEG<p_1><g_1> true**

- **There exists a computation path on which it is possible to do infinitely often *put* actions: EGAF(E (true $_{true}$U $_{p1|p2}$ true).**

- **It is always possible to perform a *put* action: AG EX$_{p1|p2}$true**

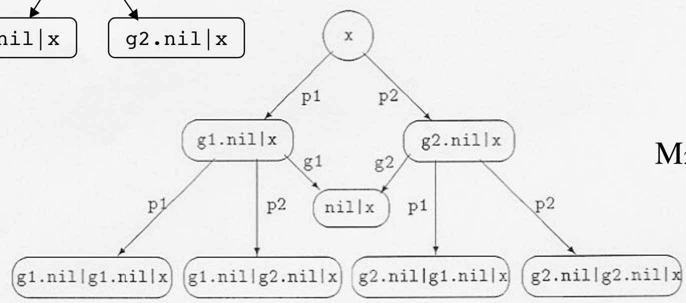
X= p1. (g1.nil | X) + p2.(g2.nil | X)



M1(X)

$M_1(X)$ |≠ EF<g_1><g_1> true

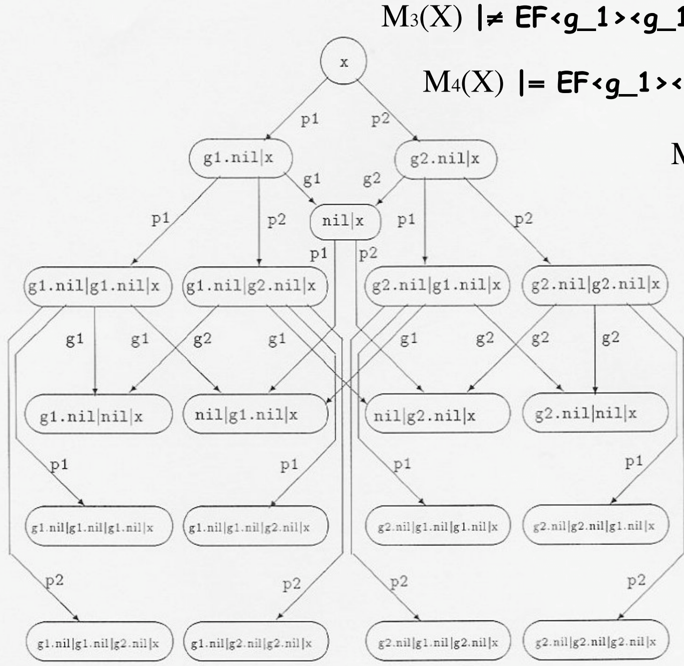
$M_2(X)$ |≠ EF<g_1><g_1> true



$M_2(X)$

$M_3(X)$ |≠ EF<g_1><g_1> true

$M_4(X)$ |= EF<g_1><g_1>true



$M_3(X)$

Example: BAG

X= p1. (g1.nil | X) + p2.(g2.nil | X)

Properties,

EFEG<p_1><g_1> true,
EGAF(E (true $_{true}$U $_{p1\ |p2}$ true),
AG EX$_{p1|\ p2}$true
are not verified by any Mi for each i

Their satisfiability implies detecting a cycle in the transition system.
By induction on the length of the chain Mi it can be proved that no cycle belongs to the transition systems of the chain.

SS approximations

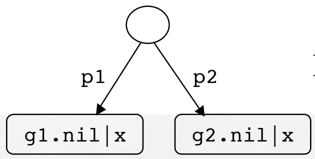
We have used another way of approximating systems which is based on a different operational semantics, which **allows us to prove a greater** set of properties than those proved by M_i, **the SS semantics** [DI93].

It is **more abstract than SOS**, since the SS rules **have built in some behavioral equivalence axioms**, i.e. they accomplish some simplifications on the terms during the derivations.

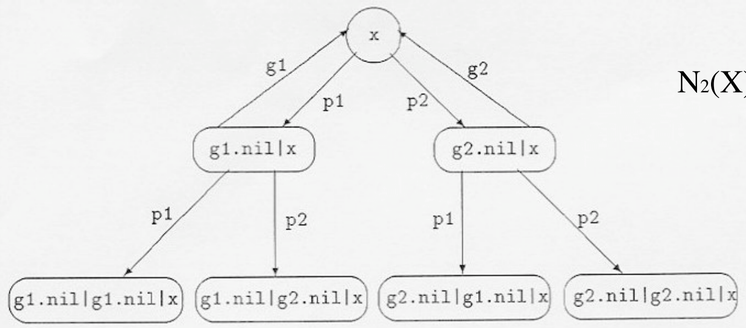
In this way it is possible to obtain **more succinct LTSs** than those obtainable with the standard SOS rules.

Sometimes, the transition system obtained by using the SS rules is finite while the SOS one is not.

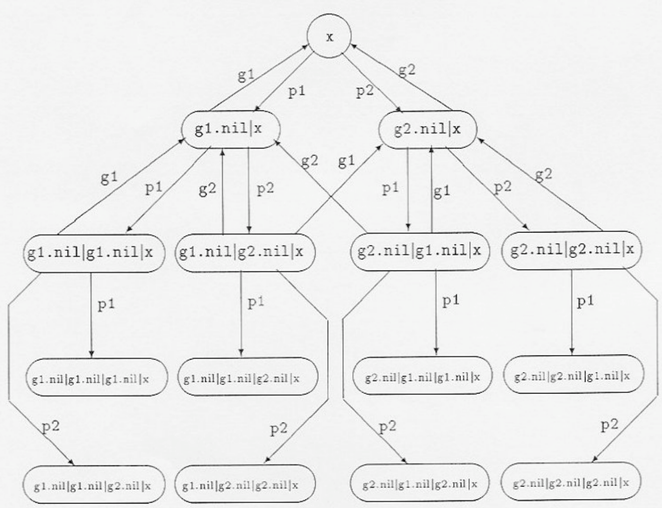
SS approximations chains



N1(X)



N2(X)



N3(X)

Example: BAG

X= p1. (g1.nil | X) + p2.(g2.nil | X)

Properties,
EF<g_1><g_1> true
is verified by N3

EFEG<p_1><g_1> true,
EGAF(E (true $_{true}$U $_{p1|p2}$ true),
are verified by N2

AG EX$_{p1|\ p2}$true
is not verified by any Ni for each i

□□logic: an extension of ACTL for mobility

Mobility is introduced, referring to the □□calculus formalism as a basic process algebra able to express the typical issues of mobility.

Model checking tools operating over pi-calcualus agent are offered by the Mobility Workbench (MWB) and by the HD-Automata Laboratory (HAL). The core of HAL are the HD-automata: they are used as a common format for the various history-dependent languages. The HAL environment includes modules which support verification of behavioral properties of pi-calculus agents expressed as formulae of suitable temporal logics.

Variants of ACTL: Mobile systems

□-logic syntax -->

□ ::= ***true*** **| ~ □| □□ □ | □ X{□}□ | <□> □ | □ F □ | □ F{□}□**

□ ::= tau | x!y |x!(y) | x?y

□ X{□}□ **strong next**

<□> □**weak next**

□ F □ **eventually**

□ F{□}□ **eventually guarded by □**

As usual [□] □, AG □can be defined by duality

□-logic is adequate with respect to strong early bisimulation equivalence

□-logic semantics

- $P \models true$ holds always;
- $P \models {\sim}\phi$ if and only if not $P \models \phi$;
- $P \models \phi \,\&\, \phi'$ if and only if $P \models \phi$ and $P \models \phi'$;
- $P \models EX\{\chi\}\phi$ if and only if:
  - **case** $\chi = \mu$
    there exists P' such that $P \xrightarrow{\mu} P'$ and $P' \models \phi$;
  - **case** $\chi = {\sim}\mu$
    there exists P' such that $P \xrightarrow{\mu'} P'$ and $\mu' \neq \mu$ and $P' \models \phi$;
  - **case** $\chi = \bigvee_i \mu_i$
    there exist P', i such that $P \xrightarrow{\mu_i} P'$ and $P' \models \phi$;
- $P \models EF\,\phi$ if and only if there exist $P_0, \ldots, P_n$ and $\mu_1, \ldots, \mu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\mu_1} P_1 \ldots \xrightarrow{\mu_n} P_n$ and $P_n \models \phi$.
- $P \models EF\{\chi\}\phi$ if and only if there exist $P_0, \ldots, P_n$ with $n \geq 1$, such that:
  - **case** $\chi = \mu$
    $P = P_0 \xrightarrow{\mu} P_1 \ldots \xrightarrow{\mu} P_n$ and $P_n \models \phi$;

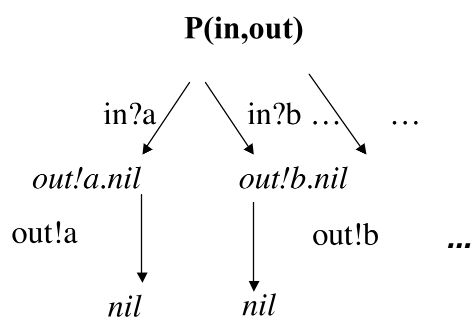
P(in,out) ::= in?(x). out! x nil

x,in,out ∈ N

N infinite sets of names

in, out: channels

x : place holder



THE SEMANTICS MODEL OF P IS:
INFINITE STATE
INFINITE BRANCHING

JACK for MOBILITY
HD -automata

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.

In HD-automata names appear explicitly in states, transitions and labels (local names) . Local names do not have a global identity.

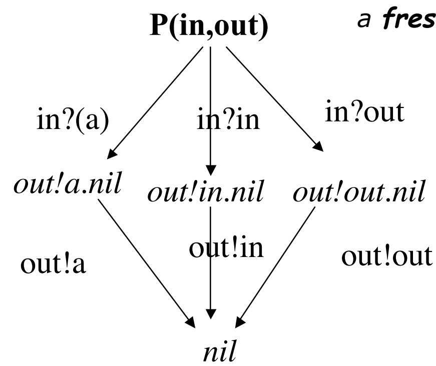
In this way, for instance, a single state of the HD-automaton can be used to represent all the states of a system that differ just for a bijective renaming.

FROM HD-AUTOMATA TO LTSs

P(in,out) ::= in?(x). out! x nil

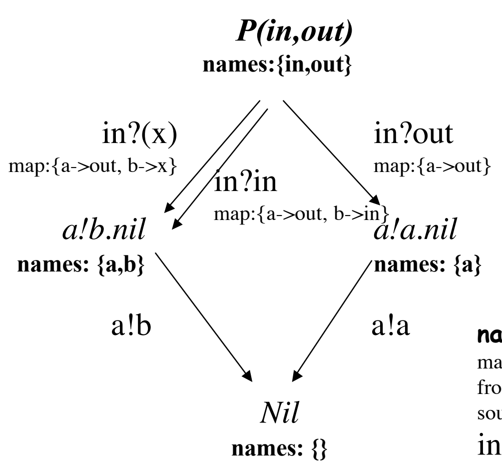
in,out **are the active names of P**

a ***fresh name***



FROM π-calculus to HD-AUTOMATA

P(in,out) ::= in?(x). out! x nil



names:{…} local names

map:{a->out, b->x} embedding function from names of the target state to the source state

in?(x) **input fresh name**

A translation function exists from □-logic to ACTL

soundness : a □-logic formula is satisfied by a □ -calculus agent P if and only if the finite state ordinary automaton associated with P satisfies the corresponding ACTL formula

The translation of a formula is thus not unique, but depends on the agent P. Specifically, it depends on the set S of the fresh names of the ordinary automaton associated with the agent P.

- $\mathcal{T}_S(true) = true$
- $\mathcal{T}_S(\phi_1 \& \phi_2) = \mathcal{T}_S(\phi_1) \& \mathcal{T}_S(\phi_2)$
- $\mathcal{T}_S(\sim\phi) = \sim\mathcal{T}_S(\phi)$
- $\mathcal{T}_S(EX\{\chi\}\phi) = \bigvee_{\mu' \in \mathcal{T}_S(\chi)} EX\{\mu'\}\mathcal{T}_S(\phi\theta)$ where $\theta = \{y'/y\}$ if $bn(\chi) = y$ 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:

- $\mathcal{T}_S(\mathtt{tau})) = \{\mathtt{tau}\}$
- $\mathcal{T}_S(x!y) = \{x!y\}$

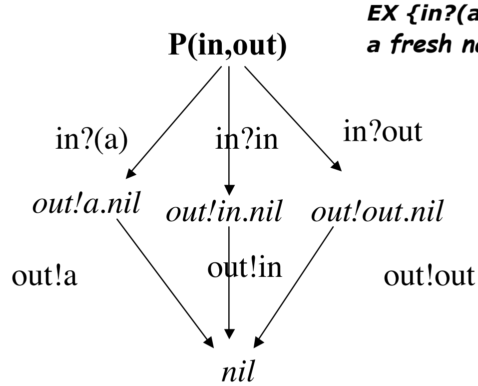
Model checking facilities

P(in,out) ::= in?(x). out! x nil

EX {in?u}EX{out!u} true (□-logic)

EX {in?(a)}EX{out!a} true (ACTL)

a fresh name



JACK for MOBILITY

The generation of the ordinary automaton associated with a pi-calculus agent consists of two stages.

The first stage constructs an intermediate representation of agent's behaviour as HD-automaton

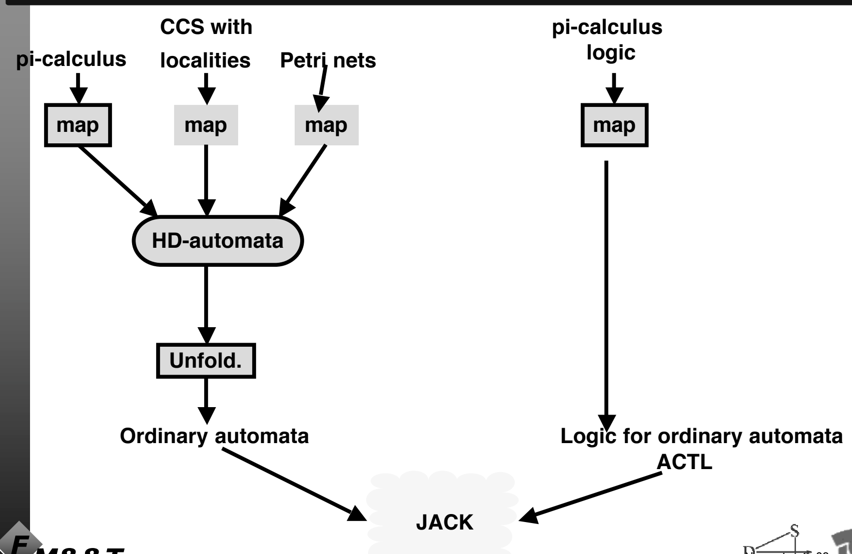
The second stage builds the ordinary automaton starting from the HD-automaton.

The generation of the ordinary automaton has been split into these two steps to achieve modularity in the structure of the verification environment and allows a more efficient implementation of the second translation step.

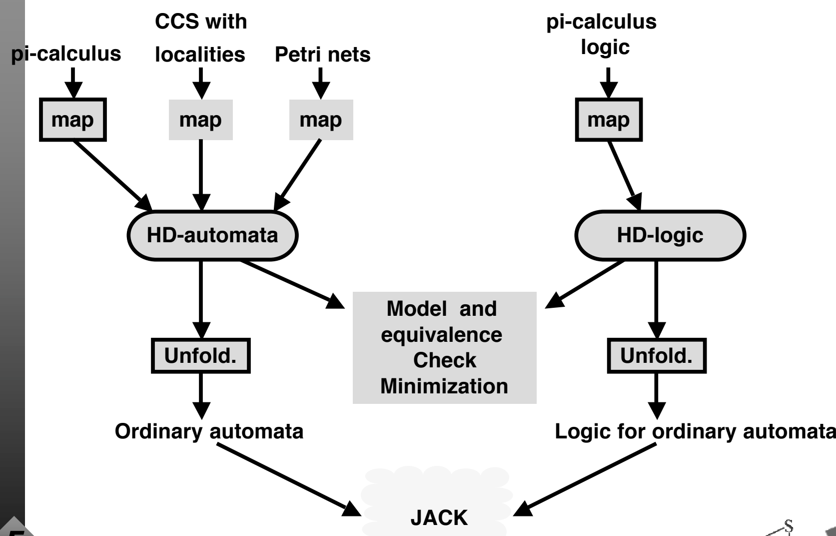
The HAL environment: an overview

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

The HAL environment: an overview



The HAL environment: an overview



HAL applications
handover protocol

The specification, GSM, describing the core of system is basically composed by four modules:
a Mobile Station MS, mounted in a car moving through two different
Geographical areas (cells), that provides services to an end user;
a Mobile Switching Centre MSC, that is the controller of the radio
communications within the whole area composed by the two cells;
the Base Station modules BSa and BSp, that are the interfaces between
the Mobile Station;
the Mobile Switching Centre.

HAL applications

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

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

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

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

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


HAL applications

There are two kinds of correctness checking that can be performed by exploiting HAL facilities.

One is the checking that the specification of the The Handover Protocol is (early) bisimilar to a more abstract service specification, that models the intended behaviour of the system.

The other is the (model) checking of some interesting properties.

HAL applications

a more abstract representation of GSM

```
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)
```


HAL applications

Some logic formulae, describing that the protocol is reliable:**, have been expressed -logic formulae.**

no messages are lost: that is whenever a message is received from the external environment through an input channel then it will be eventually retransmitted to the end user via the output channel.

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

in-order delivery**: whenever three messages are sequentiallyreceived in sequence through an input channel then the first message can be soon retransmitted to the end user through the output channel.**

AG[in?m][in?n] ~ EF {~ out!n } EX { out!m } true

HAL applications

We summarize the figures (states, transitions and times) of the different steps of a typical session of verification for the handover protocol (GSM spec)

	States	Trans.	Time
buildHD GSM.pi	**506**	**745**	**37.52 sec.**
reduceHD-red GSM.hd	**245**	**484**	**1.19 sec.**
buildFC2	**545**	**1062**	**1.54 sec.**
Minimize automaton	**49**	**91**	**3.45 sec.**
Model checking			**6 sec.**

»

UMC model checker

A most recent approach to the specification of distributed object-oriented processes is by using the proper UML diagrams, such as the state diagrams, to express the behavioural aspects of the overall design. Again, this tutorial will address the issue of how action-based model checking can be upgraded to deal with UML state diagrams: the UMC model checker will be presented as an example.

□-ACTL$^+$

Evolution formulae:

□ ::= tt | `[target.]event[(args)]` | □ | □□□ | □ □

□-ACTL$^+$ formulae:

□ ::= true | $□_1$ □ $□_2$ | □ □ |
    `EX`$_{\{□\}}$□ | `AX`$_{\{□\}}$□ | `EF`$_{\{□\}}$□ | `EF` □ | `min Y:`□(Y) | `Y` |
    `ASSERT(VAR=value)`

L2TS

Doubly Labelled Transition System: (Q, q0, Act*+{tau}, R, $\mathcal{L}$)

(Q, q0, Act*+{tau}, R) is a LTS

$\mathcal{L}$ is a labelling function $\mathcal{L}$: Q □ $\mathcal{AP}$

$\mathcal{AP}$ is a finite set of atomic propositions

(tipically of the form VAR=value)

□-ACTL+ semantics 1

transition label |= evolution formula   is the Satisfaction relation

□ |= tt   holds always

□ |= □ □   iff not □ |= □ □

□ |= □$_1$ □ □$_2$ iff □ |= □$_1$ and □ |= □$_2$

□ |= □   iff □ = tau

□ |= [target.]event[(args)]   iff □ = e_1;…;e_n 1 □ n and
□ i, 1 □ i □ n : e_i = target.event(args)

□-ACTL+ semantics 2

state |= formula   is the Satisfaction relation

q |= ASSERT(VAR=value)   iff VAR=value □ $\mathcal{L}$(q)

q |= true   holds always

q |= □ □   iff not q |= □ □

q |= □$_1$ □ □$_2$ iff q |= □$_1$ and q |= □$_2$

q |= EX$_{\{□\}}$□   iff □q' tale che q —□□ q', q' |= □ , □ |= □

□-ACTL$^+$ semantics 3

$q \models \mathtt{AX}_{\{\square\}}\square$ iff $\square q' : q \longrightarrow^{\square\square} q'$, and
$\square q' : q \longrightarrow^{\square\square} q'$, $q' \models \square$, $\square \models \square$

$q_0 \models \mathtt{EF}\square$ iff $\square q_1 \ldots q_n ,\ \square_1 \ldots \square_n ,\ 0 \square n :\ q_n \models \square$
and $\square\, i : 0 \square i < n,\ q_i \longrightarrow^{\square_{i+1}\square} q_{i+1}$

$q_0 \models \mathtt{EF}_{\{\square\}}\square$ iff $\square q_1 \ldots q_n ,\ \square_1 \ldots \square_n ,\ 0 \square n :\ q_n \models \square$
and $\square\, i : 0 \square i < n,\ q_i \longrightarrow^{\square_{i+1}\square} q_{i+1} ,\ \square_i \models \square$

$q \models \mathtt{min\ Y{:}}\square(Y)$ iff $\square\, n: 0 \square n \Big(\square^{n}(\square \mathrm{true}) \Big)$,
where $\square^{0}(Y) = \square(\square \mathrm{true})$, $\square^{n+1}(Y) = \square(\square^{n}(Y))$

□-ACTL$^+$

`EX` $_{\{\mathtt{Chart.my\_event}\}}$ `true`
in the current configuration the system can perform an evolution in which a state machine sends the signal my_event to the state machine Chart.

`EX` $_{\{\mathtt{my\_event(3)}\}}$ `true`
in the current configuration the system can perform an evolution in which a state machine sends the signal my_event(3)to some other state machine.

`AG ((EX` $_{\{\mathtt{my\_event}\}}$`true) -> ASSERT(Chart.Status=1))`
the signal my_event can be sent, only when the object is in status 1.

UMC

•**Development of Model checking techniques based on:**

On the fly* model checking *UML State machines

- **we have defined a procedure which builds the model of a complex system starting from its component subsystems (a network) while evaluating a formula.**
- **a network represents a concurrent system as a collection of synchronized agents working in parallel.**
- **It is possible in this way to verify interesting properties also on systems for which the state explosion problem makes other verification tools inapplicable**

On the fly model checking

```
Evaluate (F: Formula, S: State) is
  if we have already done this evaluation and
            the result is available then
     return the already known result
  elsif we are are already trying to evaluate F in S  then
     return  true or false depending on maximum or minimum
            fixed point semantics
  else
     Keep track of the fact that we are trying to evaluate F in S
     --      (e.g. push the pair (F,S) in a stack)
     for each sub-formula F'and
            next state S' which needs to be evaluated loop
        call recursively  Evaluate (F' S');
       if the result of Evaluate (F' S') is sufficient
              to establish the result of evaluate (F,S) then
         exit from the loop;
       end if
     end loop
     --  (at this point we have in any case a final
         result)
     Keep track of the fact that we are
             no longer trying to evaluate F in S;
         (e.g. pop the pair (F,S) from the stack)
     Possibly keep track of the performed evaluation and result (e.g. push the triple (F, S, result) in a hash table)
     return the final result
  end if
end Evaluate;
```

UMC assumptions

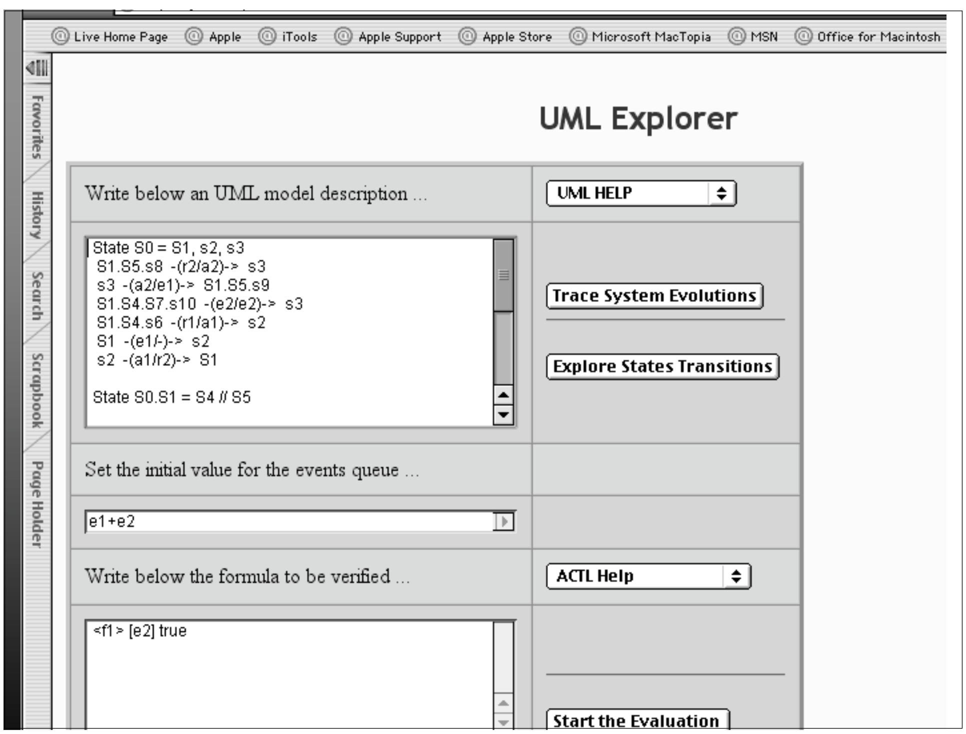
The whole sequence of actions constituting the actions part of statechart transition, is supposed to be executed as an indivisible atomic activity.

Given a model constituted by more than one state machine, a system evolution is constituted by any single evolution of any single state machine.

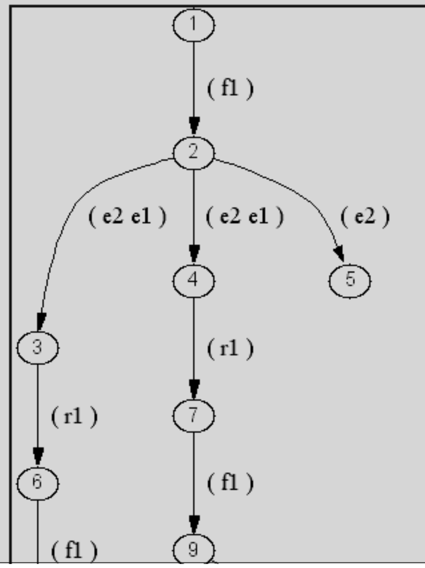
The propagation of signals inside a state machine and among state machines is considered instantaneous, and loss free.

The events queue associated with a state machine handles its events in a FIFO way.

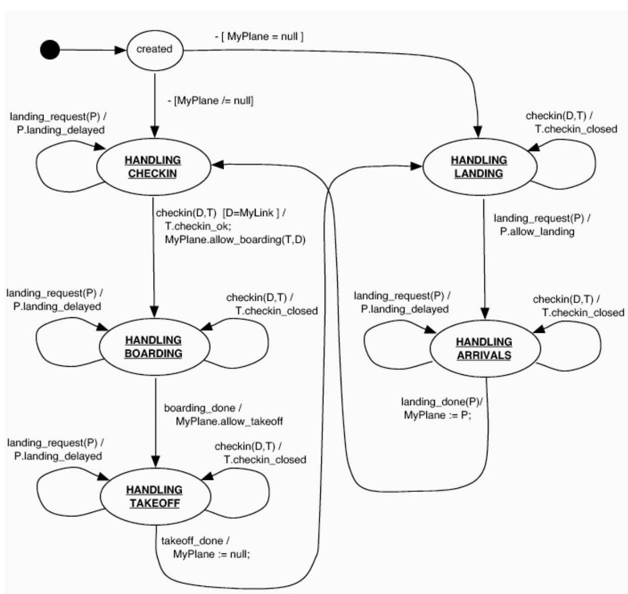
The relative priority of a join transition is always well defined and statically fixed.



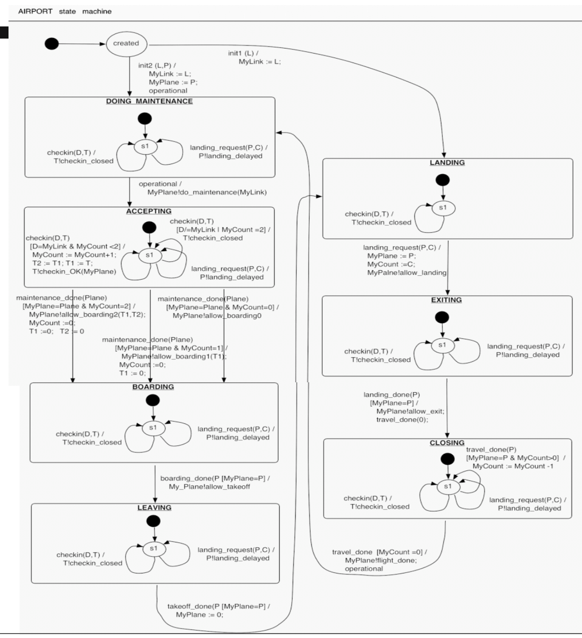
Events Queue	(e1,e2)
Active States	(S0.S1.S4.s6, S0.S1.S5.s8)
Fireable Transitions	1) S0.S1.S4: (s6) -(e1/f1)-> (S7)
Maximal Sets	1



UMC and the airport case study



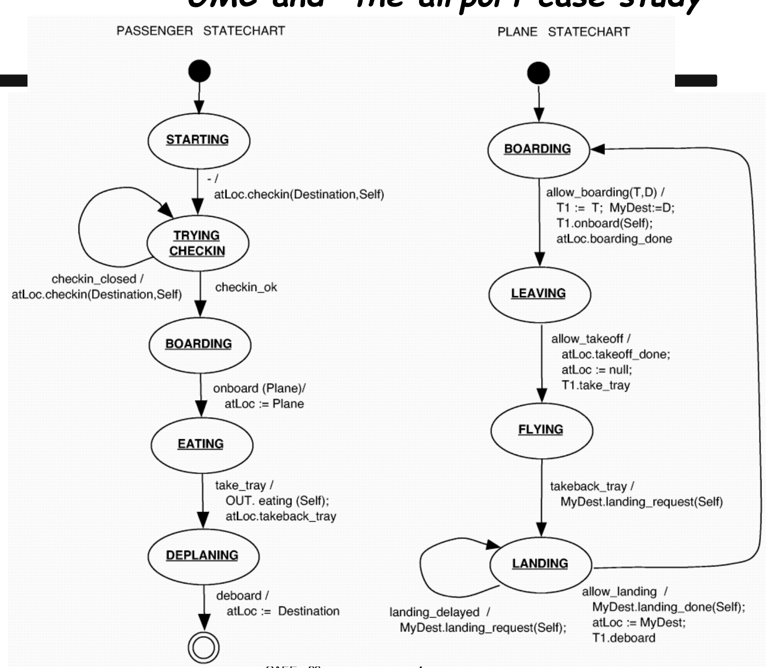
UMC and the airport case study



M&T

November 3, 03

UMC and the airport case study



M&T

Proving properties

We want to check if it is true that passengers can eat, only when their plane is flying
(we have only one plane in the model "Plane1").
This can be done by checking the following formula:

```
AG ( ( EX { eating }  true ) ->
ASSERT(Plane1.Status=1) )
```

The formula:

```
   max Z : < ( ~ eating ) & ~ checkin_closed > Z
```

The above formula,, is true. I.e. there is some infinite path in which nobody ever eats, and no check-in request are denied.

Airport case study

Let us consider one of the simplest scenarios, constituted by two airports (Airport1, Airport2), two Passenger (Traveler1, initially located at Airport1, and Traveler2, initially located at Airport2), and a single Plane (Plane1) traveling between Airport1 and Airport2.

The system is composed by 5 objects, and originates a LTS containing 18131 states and 55379 transitions.

Open research issues

Another interesting research issue is the study of counterexamples: one area of interest about counterexamples is given by the possibility of generating test cases from them; due to the common practice that sees verification of hardware and software ecomponents often effectively carried on by testing, can we use counterexample to enhance testing coverage?

*Particularly interesting in this sense is the possibility to abandon simple **linear counterexample** in favour of **tree-like counterexamples** or **counterexample autoamata.***

References – Model checking classics

E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of programs: workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag,

J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programmin*g, volume 137 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):pages 244–263, 1986.

E. M Clarke, O. Grunberg, D. A. Pele, *Model Checking*, MIT Press, 1999

J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):pages 142–170, 1992. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

References – Action based Model checking

De Nicola R, Vaandrager F.W. Actions versus State Based Logics for Transition Systems. Proc. Ecole de Printemps on Semantics of Concurrency, Lecture Notes in Computer Science vol. 469, 1990, 407-419.

R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori, "An action based framework for verifying logical and behavioural properties of concurrent systems", Computer Networks and ISDN Systems, Vol. 25, N. 7, pp. 761-778, North Holland, Febbraio1993.

C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, D. Romano, "Formal Verification Environment for Railway Signalling System Design", Formal Methods in System Design, Vol. 12, pp.139-161, 1998.

S. Gnesi, G.Ristori. A Model Checking Algorithm for p-calculus agents. In Advances in Temporal Logic, H. Barringer, M. Fisher, D. Gabbay, G. Gough eds., Applied Logic Series, Vol. 16, Kluwer Academic Publishers, 2000, pp.339-358.

A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, E. Tronci A Symbolic Model Checker for ACTL, Applied Formal Methods -- FM-Trends 98, LNCS 1641, Springer - Verlag, 1999.

S. Gnesi and F. Mazzanti, On the Fly Verification of Networks of Automata, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), special session on Current limits to automated verification for distributed systems, CSREA Press, 1999 (Invited paper).

G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, G. Ristori Verifying Mobile Processes in the HAL Environment, CAV'98, LNCS 1427, Springer - Verlag, 1998.

C. Bernardeschi, A. Fantechi, S. Gnesi, "Formal Validation of Fault-tolerance Mechanisms inside GUARDS", Journal of Reliability Engineering and System Safety, Vol. 71, n.3, Feb. 2001, pp. 261-270.

N. De Francesco, A. Fantechi, S. Gnesi, P. Inverardi, "Finite Approximations for Model Checking Non-finite-state Processes" , The Computer Journal, vol 44 n. 2, 2001.

R. Meolic, T. Kapus, Z. Brezocnik, "An Action Computation Tree Logic With Unless Operator", SEEFM 2003, Thessaloniki, Greece, November 20, 2003.

M. Hennessy, R. Milner, "Algebraic Laws for Nondeterminism and Concurrency", Journal of ACM, vol. 32, n. 1, January 1985, pp. 137-161.

R. Mateescu,M. Sighireanu Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany), April 2000

A, Bouali, S, Gnesi, S. Larosa S. The integration Project for the JACK Environment. Bulletin of the EATCS, 54, 1994, 207-223.