UMC User Guide
(version 2.5)

Franco Mazzanti

Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"
ISTI-CNR
Via A.Moruzzi 1
56124 Pisa, Italy
franco.mazzanti@isti.cnr.it

1 Introduction

The goal of the UMC project is to experiment in three different directions:

- We are interested in exploring and exploiting the advantages given by the "on the fly" approach to model construction and checking [5,2,9].

- We are interested in investigating the kind of user interface which might help a non-expert user in taking advantage of formal specifications and verification techniques.

- We are interested in testing the appropriateness of the UML [19,26] methodology for the specification and verification of the dynamic behaviour of a system.

This experimentation is carried out through the actual development of a new verification tool (UMC), specifically tailored to the aims of the project. The immediate purpose of the project is definitively not that one of building a heavyweight verification machine (e.g. targeting the verification of systems with a very large number of states), even if the gained experience might certainly be useful for possible future extensions moving also in this direction.
The development of the tool builds over the experience obtained with the previous development of FMC (see [13]), another "on the fly" model checker for networks of automata (specified in the fc2 format or as a collection of regular CCS / basic Lotos agents).
In our case, the model under investigation is specified by a textual description of a set of UML statechart diagrams - one for each class of objects which constitutes the system - and by a set of objects instantiations.
The properties to be verified are specified as mu-ACTL+ formulas: a temporal logic with the power of full mu-calculus, which includes the high level composite operators typical of branching time action based logic ACTL [6].
The reference dynamic semantics for UML statecharts is as far as possible corresponding to the "official" semantics as given in [26], as already formally described in [1,21,29] (with some limitations).
In Section 2 we will describe how the underlying model to be analysed can be defined, and in Section 3 we will describe the temporal logic which can be used to express the properties to be verified. In Section 4 we will describe the how to use the tool both through its simple command-line interface and its web interface. In Section 5 we give an overview of the overall structure of UMC and of then verification algorithm used by the tool. In Section 6 we show a small but complete example of model verification.

2 The definition of the UMC Model

A complete UMC model is constituted by a sequence of class and object declarations.
Classes can model active or non-active objects (an active object has its dynamic behaviour described by the statechart associated to its class).
A state machine (with its events queue) is associated only to the active objects of the system.
Non-active objects play the role of "interfaces" towards the outside of the system, and can only be the target of signals.
There is a predefined non-active `OUT` Class, and a predefined `OUT` object, which can be used to model the sending of signals to the outside of the system, and there is a predefined non-active `ERR` Class, and a predefined `ERR` object, which can be used to model the notification or error signals to the outside of the system; further non-active classes and objects can be defined by defining classes without statechart.
At least one active object must be defined, and the declaration of an object must appear after the declaration of the class to which it belongs.
In Appendix B is given the complete grammar for UMC models. In the following two sections we describe in more detail the two model components, namely classes and objects.

Classes

The behaviour of a the set of objects belonging to a class is defined by a statechart diagram associated with the class itself.
In particular, the definition of a class statechart consists in the introduction of

- the class name
- the list of events which trigger the transitions of the objects of the class
- the list of attributes (variables) local to the objects of the class
- the structure of the states of the class (nodes of the statechart diagram)
- the transitions of the objects of the class (edges of the statechart diagram)

The attributes (local variables) of a class, and the parameters of events can be explicitly typed, and the allowed types are just the "int", "bool" and "object" types.

The structural definition of the states of a statechart consists in a sequence of state definitions which starts from the definition of the top level state. The definition of outer states must precede the definition of its nested substates. The top level state of a statechart must be a composite sequential state.
A composite sequential state is defined by a list of substates (which can be composite sequential states, parallel states or simple states). The first substate of a composite state is assumed to be its default initial substate.
For any simple state appearing in the definition of a composite sequential state it is not necessary to give any further explicit definition.
A composite parallel state is defined as a parallel composition of several composite sequential substates also called "regions" of the parallel state. For each region must subsequently be given an explicitly definition as a composite sequential state.

The definition of a transition contains an optional transition label, a set of source states, a trigger, an optional guard, an optional list of actions and a set of target states.
Each state of the source or target is identified by its composite_name which is a path (at most starting from the top state) which univoquely identifies the state.

A transition with more than one source is called a "join" transition.
In the case of joins transitions, the first state in the source list is required to be "the most transitively nested source state". In this sense the first state univoquely determinates the priority of the transition.
A transition with more than one target is called a "fork" transition.
The trigger of a transition can be an event declaration (matching one of the definitions already given in the events section of the chart), or the hyphen symbol ("-") which means the absence of any explicit trigger (i.e. a "completion transition"). If the trigger is an event declaration with formal parameters, the name of the parameters can be used inside the actions part of the transition.
The guard (if present) is a simple form of boolean expression involving the chart variables.
Each action of the actions part can be either a signal or a local assignment.
A signal is similar to an event declaration, but its arguments are constituted by integer expressions instead that by formal parameters. If a signal is preceded by a destination specification the meaning is that the signal is sent to the events queue of the specified destination chart. If the destination chart name does not identify an explicitly defined chart then the signal is considered to be just a external signal observable in the environment. If no destination is specified, then the signal is supposed to be sent to the local events queue.
An assignment specifies a target variable name and a source integer expression.

Examples

Let us consider the statechart diagrams shown in Figure 1,3 and 5. This statechart diagram can be declared in UMC format using the textual description respectively shown in Figures 2, 4 and 6.

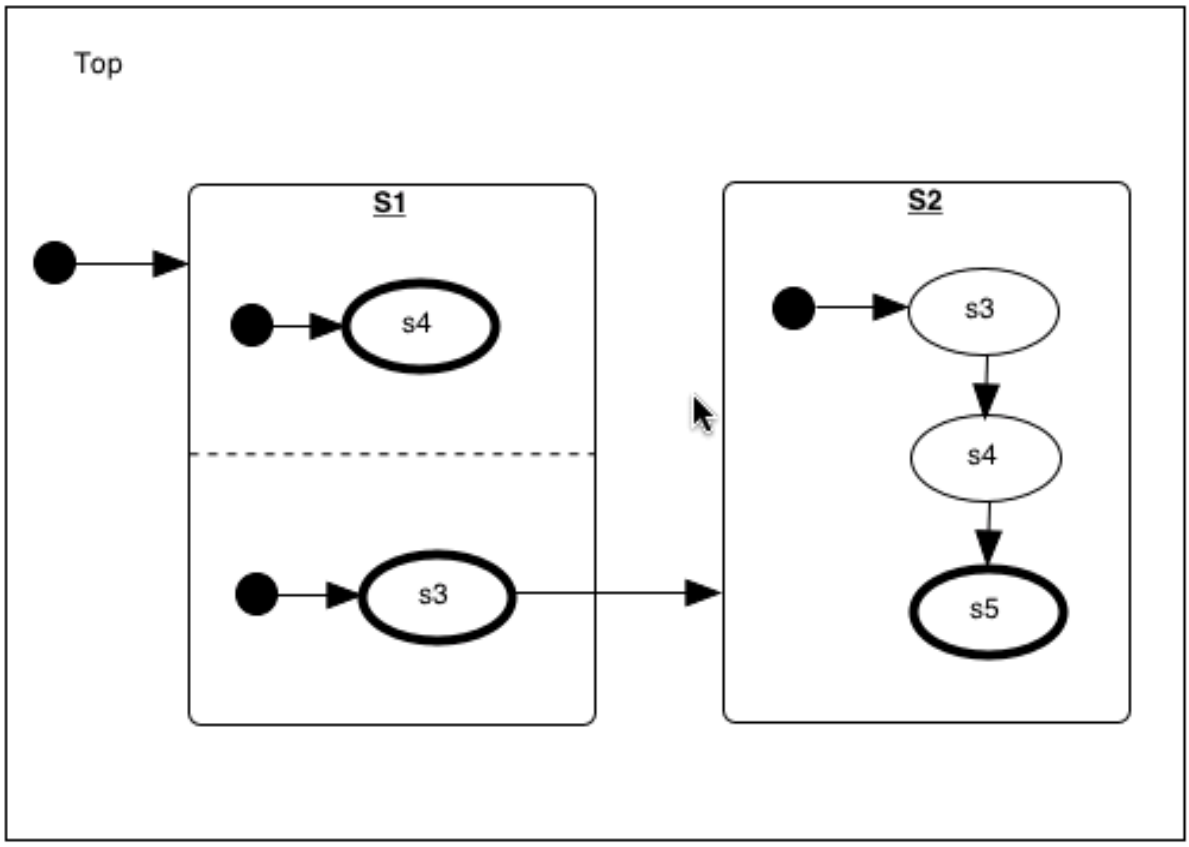

Figure 1: A first sample statechart diagram

```
Class Main
 State Top =  S1, S2    -- S1 is the initial state of Top
 State S1 = R1 // R2
 State S2 = s3, s4      -- s3 is the initial state of S2
 State S1.R1 = s4
 State S1.R2 = s3       -- simple state are not explicitly declared
 Transitions:
    S1.R2.s3 -> S2
    S2.s3 -> s4
    S2.s4 -> s5
-- initial default transitions are not explicitly declared
```

Figure 2 A first sample textual representation of a statechart diagram

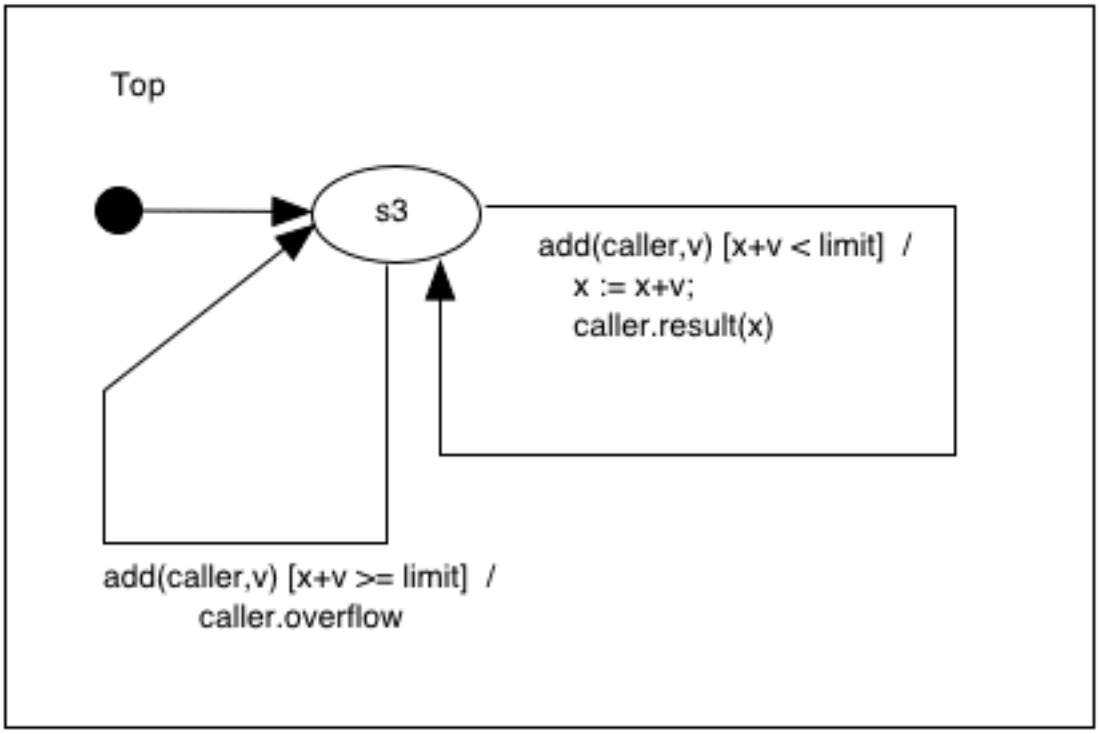

Figure 3: A second sample statechart diagram

```
Class Counter
Events: add(caller:obj, v:int)
Vars:  x:int, limit:int
State Top =  s1
Transitions:
  s1 -(add(caller,v) [x+v <limit] /x:=x+v; caller.result(x) )-> s1
  s1 -(add(caller,v) [x+v >=limit] / caller.overflow )-> s1
```

Figure 4 A second sample textual representation of a statechart diagram

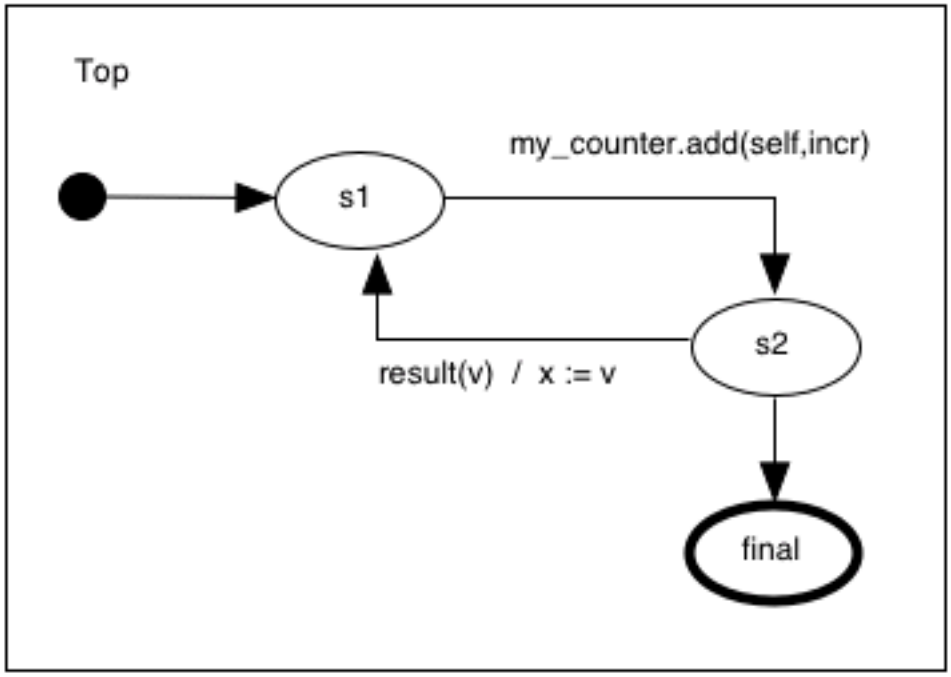

Figure 5: A third sample statechart diagram

```
Class User
Events: overflow, result(v:int)
Vars:  my_counter:obj, x:int, incr:int
State Top =  s1, s2
Transitions:
   s1 -(mycounter.add(self,incr) )-> s2
   s2 -( result(v) / x:=v  )-> s1
   s2 -( overflow )-> final
```

Figure 6 A second sample textual representation of a statechart diagram

Objects

Once the needed classes are have their behaviour defined by the appropriate statechart, we can define the actual evolving system as a set of object instances. Each object instance is declared by an object declaration which introduces the object name, the name of its class, and possibly any specific initial values for its attributes.
This initial values can be literals or names of other objects (possibly also objects which will be declared later in the text).

Example 1:

The initial configuration of a system is constituted by one object instance for each of the classes "Counter" and "User":

```
Object My_User: User (my_counter => My_Counter, incr => 1)
Object My_Counter: Counter (limit => 10)
```

3 UMC Logics

The logics supported by UMC is an extension of the action based branching time temporal logic mu-ACTL[7,13], which is in turn an extension with a fixed-point operator of the action based logic ACTL defined in [6], and clearly has the expressive power of full modal mu-calculus [19]. Formulas can be built over basic state predicates using the usual boolean operators, the classic temporal logic modal operators, and the fixed point operators.

State Predicates

The simplest form of logic formula is a StatePredicate.
A StatePredicate can be either the formula "`true`", or the formula "`false`", or a predicate on the internal structure of a system configuration .
The formula "`true`" clearly holds for any state, the formula "`false`" never holds in any state, and the formula "`ASSERT(obj.attr = val)`" holds for all the states in which the attribute `attr` of object `obj` has value `val`. The obj part of the attribute can be omitted is there system is constituted by a single active object.

Boolean logical operators

Logic formulas can be composed with the usual boolean operators which have the classical meaning.
The ~ operator has precedence over the others `&` | `->` relational operators.
Since no specific binding priority between the relational operators is defined, the boolean composition of formulas should not be ambiguous (and parenthesis should be explicitly used to disambiguate when needed).

Next temporal operators

The basic temporal operators are the "next" operators which allow to specify that a certain formula should be true in a certain states reachable in one step from the current state.
They have the `EX` or `AX` form.
"`EX {action-predicate} subformula`" holds in the current state if (and only if) there exists a transition from the current state, whose label satisfies the `action-predicate`, which leads to state in which the `subformula` holds.

"`AX { action-predicate} subformula`" holds in the current state if (and only if) all the transitions which exit from the current state have a label which satisfies the `action-predicate`, and lead to state in which `subformula` holds. Moreover the current state must not be final (i.e at least a transition must exist)[1].

An `action predicate` is just a boolean composition of action expressions.
The action expression "`tau`" is satisfied only by the system evolutions which do not generate any signal (i.e. internal transitions or stuttering transitions).
The action expression "`true`" is satisfied only by the system evolutions which generate at least a signal.

---

[1] The `AX` operator can be considered a derived operator since "`AX{act} Formula`" is equivalent to "`(EX{act} Formula) & (~EX{act} ~Formula) & (~EX{~act } true) & (~ET true)`"

The action expression "`false`" is not satisfied by any system evolutions.
The action expression "`event_id`" is satisfied only by the system evolutions which do generate a signal "`event_id`", possibly being sent to another state_machine, and possibly with some parameters (notice that event_id might not be the only signal generated by the evolution).
The actions expression "`target!event_id`" is satisfied only by system evolution which generate a signal event_id being sent to state machine "`target`".
The actions expression "`event_id(args)`" is satisfied only by system evolution which generate a signal `event_id` with the specifies `args`,
The actions expression "`target!event_id(args)`" is satisfied only by system evolution which generate a signal event_id with the specifies `args`, being sent to object denoted by "`target`".

For example, the action expression "`event_id`" is satisfied by the system evolutions labeled with any of the following labels:

"`event_id`", "`event_id(3)`", "`target!event_id`", "`target!event_id(0)`",
"`first_event_id; target!event_id(1,2,3); other_event_id`".

The action predicate "`pred1 | pred2`" is satisfied only by the system evolutions which satisfy either `pred1` or `pred2`.
The action predicate "`pred1 & pred2`" is satisfied only by the system evolutions which satisfy both `pred1` or `pred2`.
The action predicate "`~pred1`" is satisfied only by the system evolutions which do generate a signal not satisfying `pred1`, and which do not generate a signal satisfy `pred1`. Inside `pred1` the action expression "`tau`" is not allowed. [2] Notice that, for example, "`~true`" is never satisfied by any evolution.

Recursion operators

The power of mu-calculus is achieved through the two fixed point operators max and min. The fixedpoint identifier can obviously only appear inside the fixedpoint body of the formula
In state s the formula: `max Z : FixedPointBody(Z)` is true if and only if the following infinite formula is true in s:

```
FixedPointBody(false) ))  |
FixedPointBody(FixedPointBody(false))  |
FixedPointBody(FixedPointBody(FixedPointBody(false)))  |
...  |  ... |
```

In state s the formula: `min Z : FixedPointBody(Z)` is true if and only if the following infinite formula is true is s:

```
FixedPointBody(true)    &
FixedPointBody(FixedPointBody(true))  &
FixedPointBody(FixedPointBody(FixedPointBody(true)))  &
...  |  ... |
```

For example, the following formula: `max Z: EX{a} Z` holds in a state s1 if and only if there is a next state s2 reachable with an evolutions satisfying "`a`" in which `Z` holds again (i.e. if there exists an infinite path of "`a`" evolutions sending the "`a`" signal starting from s1).

---

[2] This semantics allows to preserve important properties like `{b & ~b}` being equivalent to `{false}`

Of the two fixpoint operators only one might be considered primitive as the other one can be easily derived from the first. I.e:

- `max Z: FixpointBody(Z) is equivalent to: ~min Z: ~FixpointBody(~Z)`
- `min Z: FixpointBody(Z) is equivalent to: ~max Z: ~FixpointBody(~Z)`

Even if monotonicity of a fixpoint formula is not strictly required, it is not advised to make use of non-monotonic formulas because their meaning in not necessarily intuitive.
E.g. the formula "`max Z: ~Z`", holds for any state because its semantics is given by:
" `false | ~ false | ~ ~ false | ...`"
The above is equivalent to " `false | true | ...` " whose value is "`true`" .
It is likely that in future version of the tool the monotonicity of the formula will be required and checked.

Several higher level operators are usually found in branching time logics, which are also directly supported by UMC. "Future", "Globally", "Until" "box" and "diamond" are among the most widespread.

Clearly , max and min recursion operators give to the logic a far greater power than simple plain ACTL. For example the following property: "*There exist an path which is an infinite sequence of* `b` *signals*" has no equivalents in plain ACTL, but can be simply written in mu-ACTL+ as:

```
max Z: (EX {b} Z )
```

Eventually (alias Future)

The formula: "`EF Formula`" holds in a state s1 if and only if `Formula` holds in s1 or in one of the states reachable in one or more steps from s1.
The formula: "`AF Formula`" holds in a state s1 if and only if `Formula` holds in s1 or in one of the states reachable in one or more steps from s1, for all the possible paths starting from s1.[3]

Globally

The formula: "`EG Formula`" holds in a state s1 if and only if `Formula` holds in s1 and in all the states reachable in one or more steps from s1, along at least one path starting from s1.
The formula: "`AG Formula`" holds in a state s1 if and only if `Formula` holds in s1 for all the states reachable in one or more steps from s1 (along any path).[4]

Until

The formula "`E[ Formula1 U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if `Formula2` holds in a state s2 reachable in one or more steps from s1, and `Formula1` holds in all the intermediate states along the path from s1 to s2 (s1 included, s2 excluded).[5]

---

[3] "`EF Formula`" can be translated as "`min Z: (Formula | EX Z)`" or as "`~AG ~Formula`"
"`AF Formula`" can be translated as "`min Z: (Formula | AX Z)`" or as "`~EG ~Formula`"

[4] "`EG Formula`" can be translated as "`min Z: (Formula & EX Z)`" or as "`~AF ~Formula`"
"`AG Formula`" can be translated as "`min Z: (Formula & AX Z)`" or as "`~EF ~Formula`"

[5] "`E[ Formula1 U Formula2 ]`" can be translated as:
"`min Z: ( Formula2 | (Formula1 & EX Z ))`"

The formula "`A[ Formula1 U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if, for all paths p starting from s1: `Formula2` holds in a state $s2_p$ reachable in one or more steps from s1 and, `Formula1` holds in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded).[6]

The formula "`E[ Formula1 { act } U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if `Formula2` holds in a state s2 reachable in one or more steps from s1 performing only internal actions (labelled `tau`) or actions satisfying `act`, and `Formula1` holds in all the intermediate states along the path from s1 to s2 (s1 included, s2 excluded).[7]

The formula "`A[ Formula1 { act} U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if, for all paths p starting from s1: `Formula2` holds in a state $s2_p$ reachable in one or more steps from s1 performing only internal actions (labelled `tau`) or actions satisfying `act`, and `Formula1` holds in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded).[8]

The formula "`E[ Formula1 { act1 }U { act2 }Formula2 ]`" holds in a state s1 if exists at least one path p starting from s1 made by a (possibly empty) sequence of internal evolutions (labelled `tau`) or evolutions whose label satisfies `act1` followed by a conclusive evolution whose label satisfies `act2` which leads to a state $s2_p$ in which holds `Formula2`, and in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded) `Formula1` holds.[9]

The formula "`A[ Formula1 { act1 }U { act2 } Formula2 ]`" holds in a state s1 if and only if all paths p starting from s1 are made by a (possibly empty) sequence of internal evolutions (labelled `tau`) or evolutions whose label satisfies `act1` followed by a conclusive evolution whose label satisfies `act2` which leads to a state $s2_p$ in which holds `Formula2`, and in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded) `Formula1` holds.[10]

So far, the above are the only currently supported flavours of "until" operators.
Further flavours are sometimes used in other logics , as for example versions of "weak until", or the "release" operator which is the dual of the normal "until".

Diamond

---

[6] "`A[ Formula1 U Formula2 ]`" can be translated as:
`"min Z: ( Formula2 | (Formula1 & AX Z ))"`

[7] "`E[ Formula1 { act }U Formula2 ]`" can be translated as:
`"min Z: ( Formula2 | (Formula1 & EX{act | tau} Z ))"`

[8] "`A[ Formula1 U Formula2 ]`" can be translated as
`"min Z: ( Formula2 | (Formula1 & AX {act | tau} Z ))"`

[9] "`E[ Formula1 { act }U { act2 }Formula2 ]`" can be translated as:
`" min Z: Formula1 & ( (EX {act2} Formula2) | EX {act1 | tau} Z} )`
`" E [ true { false } U { act } Formula ]` is equivalent to `"< act > Formula"`

[10] "`A[ Formula1 U { act2 } Formula2 ]`" can be translated as:
```
min Z: ( Formula1 & ( ~(EX {act1 & ~ act2} ~Z)  & (~EX {tau} ~Z) &
 (~EX {act2 & ~ act1} ~Formula2) & (~EX {act2 & act1} ~(Formula2|Z) )
```

The formula: "`< act> Formula`" holds in a state s1 if and only if `Formula` holds in s1 or in one of the states reachable in one or more steps from s1.performing only internal actions (labelled `tau`) or actions satisfying `act`.

Notice that the diamond operation in UMC has a "weak" semantics (for backward compatibility with ACTL), while in the mu-calculus it usually has a "strong" semantics.
The strong "`< act > Formula`" operator of the mu-calculus can be expressed as:

```
EX{ act } Formula.
```
[11]

Box

The formula "`[ act ] Formula`" holds in state s1 if the formula "`Formula`" holds.in all the states (if any) reachable from s1 performing an evolution whose label satisfies `act`, possibly after a finite sequence of internal evolutions (labelled `tau`). [12]

Notice that the box operation in UMC has a "weak" semantics (for backward compatibility with ACTL), while in the mu-calculus it usually has a "strong" semantics.
The strong "`[ act ] Formula`" operator of the mu-calculus is here expressed as:

```
" ~ EX.{ act } ~ Formula"
```

Examples

Let us consider some simple logic properties:

S1) *There is a reachable deadlock (i.e. a final state) somewhere.*

```
EF FINAL
```

S2) *There is a reachable livelock (i.e. a loop of* `tau` *evolutions) somewhere.*

```
EF  max Y: ET Y
```

S3) *The sending of signal* `a` *is always eventually possible (i.e. infinitely possible).*

```
AG EF EX{a} true
```

Let us consider the following basic existential (fairness related) properties:

E1) *There exist a path such that predicate* `a` *holds infinitely often*. [13]

If "`a`" is an evolution predicate this property can be written as:

```
max Z:  min W: ( (EX{a} Z )  |  (EX {tau | ~a} W) )
```

---

[11] "`< act> Formula`" can be translated as: "`min Z: ((EX {act } Formula ) | EX { tau } Z )`"

[12] "`[ act ] Formula`" can be translated as:
"`max Z: ((~EX {tau} ~Z) & ~(EX {act} ~Formula))`" and is equivalent to:
`~< act > ~Formula`

[13] I.e. at least one path satisfies the pattern: $(\sim a^* \; a)^\infty$

If "a" is a configuration predicate [14] this property can be written as:

```
max Z:  min W: ( (assert(a) & EX Z) |  (~assert(a) & EX  W) )
```

E2) *There exist a path from a certain point of which predicate* a *does no longer hold*.[15]

If "a" is an evolution predicate this property can be written as:

```
min W: max Z: ( ( EX{ tau | ~a} Z ) | ( EX{a} W ) )
```

If "a" is a configuration predicate [16]this property can be written as:

```
min W:  max Z: ( ( ~assert(a) & EX Z ) | ( assert(a) & EX W ) )
```

E3) *There exist a path such that predicate* a *and predicate* b *hold infinitely often*.[17]

If "a" and "b" are both evolution predicates this property can be written as:

```
max Z:
  min W: ( (EX{a} min KA: ( (EX {b} Z) | (EX {tau | ~b} KA) ) |
           (EX {tau | ~a} W)  )
```

E4) *There exist a path from certain point of which predicate* a *holds infinitely often and predicate* b *does no longer hold*.[18]

If "a" and "b" are both evolution predicates this property can be written as:

```
EF  max Z:
      min W: ( (EX {a & ~b} Z) |
               (EX { tau | (~a & ~b} W) )
```

If "a" is a configuration predicate and "b" is an evolution predicate this property can be written as:

```
EF max Z:
    min W: ( ( assert(a)  & EX {tau | ~b} Z)  |
             ( ~ assert(a)  & EX {tau | ~b} W) )
```

We can now derive some interesting <u>universal, fairness related, formulas,</u> like:

UF1) *For all paths, if visible action* a *is <u>done</u> infinitely often, then also* b *is <u>done</u> infinitely often*.

This property, in fact can also be stated as:

Does not exist a path such that from a certain point a *is done infinitely often, and* b *is not done anymore*. (I.e. it is the negation of a kind of **E4**)

I.e. `~ EF max Z:`

---

[14] In CTL* this property would be stated as: E(GFa)

[15] I.e. at least one path satisfies the pattern: any* (~a) ∞

[16] In CTL* this property would be stated as: E(FG~a)

[17] I.e. at least one path satisfies the pattern : (~a* a ~b* b)∞

[18] I.e. at least one path satisfies the pattern : (any)* ((a & ~b)* (a & ~b)) ∞

```
min W: ((EX {a & ~b} Z)  |
        (EX {tau | (~a & ~b)} W))
```

UF2) *For all paths, if visible action* `a` *is possible an infinite number of times, then a is done an infinite number of times.* **(strong fairness)**

This property, in fact can also be stated as:

Does not exist a path such that `a` *is infinitely possible, and from a certain point* `a` *is not done anymore* (Again, this it is the negation of a kind of **E4**)

I.e. `~ EF max Z:`

```
min W: ( ( (EX{a} true) & EX {tau | ~a} Z)  |
         ( (~EX{a} true ) & EX {tau | ~a} W ) )
```

4 The UMC tool and how to use it

The UMC environment is actually two faced. From one side we have a basic command-line oriented "umc" tool, easily portable to many platforms, which implements the basic exploration and verification features.
From another side we have an experimental www interface (an html browser is needed to view it), which integrates the basic umc features with graphic features and further abstraction features, and which allows to use the tool remotely from the web (with all the advantages and disadvantages of the case).

4.1 Command line interface

"umc" is in its original form is a command line oriented tool which is called with a parameter which is the name of a "<file>.umc" document containing the textual description of an UML model.

Starting umc

```
--> umc mymodel.umc
```

Once started, umc enters into a loop inside which it accepts logic formulas to be evaluated, or umc commands to be executed.

```
--------- umc  version 2.5  -----------

Loading the model from file: mymodel.umc
Enter an ACTL formula, or a command, or exit with "."
|-
```

Verifying a Formula

If a logic formula is inserted, it is immediately evaluated and further input is awaited.

```
|- EF ET true
 ----------  reprinted formula -----------
 EF ET true
 ----------------------------------------
 Starting Evaluation with LTS Bound set to 1024
 The formula is  FALSE
|-
```

Looking at the explanations of the result

Once a logic formula has been evaluated, we can ask for an explanation of how the result has been deduced, which results in the printing of all the deduction steps leading to the result.
This is achieved with the "why" command.

```
|- why
----------------------------------------
, the formula:  EF ET true
   is FOUND_FALSE in State C1
because
 The formula:  ET true
   is FOUND_FALSE in State C1
|-
```

Browsing into the current configuration detail

The "info" command allow to observe the internal details of the current configuration. These details include, for each active object of the model, the name of the object, the status of its event queue, the status of its variables, the current trigger (if any), the set of currently active states, and the set of the currently fireable transitions.

```
|- info
---- CURRENT CONFIGURATION : C1 -----
 OBJECT NAME           = MyModel
 OBJECT QUEUE          =
 CURRENT TRIGGER       =
 CURRENT VARIABLES     =
 ACTIVE STATES        = Top.s1
 FIREABLE TRANSITIONS = {{#1}}
--------------------------------------------------
|-
```

Tracing the system evolutions tree

The "trace" command, which takes as argument the depth at which the trace should be truncated, allows to trace the possible system evolutions, starting from the current configuration.
For each evolution it is shown the source and target configuration, the evolving object, and the sequence of signals (if any) generated by the evolution.
If the showstuttering preference is set to true (this is the default case), umc generates a dummy
`OUT.lostevent(event)` signal each time a stuttering evolution occurs (i.e. when an object discards the current triggering event because no fireable transitions are available).

```
|- trace 4
C1.MyModel -()-> C2     (#1)
   C2 is FINAL.
|-
```

Interactively exploring the system evolutions tree

The explore command starts a nested exploration cycle inside which the user is allowed to interactively select one of the possible evolutions from the current state, display some information about the current configuration, or to climb back in the evolutions tree.

```
|- explore
Enter a number to select an evolution,
 'i' to get more info on this state,
 '.' to exit, 'b' to go back one step,
 'tn' to trace the LTS tree for n levels.
Current Configuration:  C1
Possible Evolutions:
 1: - a -> C2     (#1)
<explore> i
---- CURRENT CONFIGURATION : C1 -----
 OBJECT NAME          = MyModel
 OBJECT QUEUE         =
 CURRENT TRIGGER      =
 CURRENT VARIABLES    =
 ACTIVE STATES       = Top.s1
 FIREABLE TRANSITIONS = {{#1}}
-----------------------------------------------------
Current Configuration:  C1
Possible Evolutions:
 1: - a -> C2     (#1)
<explore> 1
Current Configuration:  C2
Possible Evolutions:
 1: - OUT.lostevent(a) -> C3     ()
<explore> b
Current Configuration:  C1
Possible Evolutions:
 1: - a -> C2     (#1)
<explore> .
|-
```

Moving to another specific configuration

The initial configuration of the system has name "C1"; as the exploration of the system evolutions proceeds, new configurations are discovered "on the fly" and named "C2", "C3" ,.... and so on.
Notice that the name of the configuration depends from the order in which the system evolutions are explored, and different names can be given to the same actual configurations, in different run of the tools, if the system configurations are explored in a different order.
The "initial" command set the initial configuration. (i.e.- C1) as the current configuration .
The "moveto Ci " command set "Ci" as current configuration, if such a configuration has indeed been generated.

```
|- trace 2
C1.MyModel -(a)-> C2     (#1)
   C2.MyModel -(OUT.lostevent(a))-> C3     ()
|- moveto 2
Current Configuration: C2
|- initial
Current Configuration: C1
|-
```

Viewing and adjusting some tool preferences and parameters

The behaviour of the "umc" tool can be customized according to some parameters.
The "set" command allows to observe the current status of them or change their value.

```
|- set
initial_lts_depth=1024
max_explanation_depth=25
max_evolutions_depth=6
showstuttering=TRUE
|- set showstuttering=FALSE
|-
```

The initial_lts_depth parameter affects the way in which the on-the-fly evaluation of a formula proceeds. In particular it set the initial depth limit at which a depth-first subcomputation should be aborted, in favour of other less deep subcomputations. As a limit case, when set to 1,
the evaluations procedes if a breadth first way (even if in a quite inefficient way in the current version of the tool).
The max_explanation_depth parameter constrains the maximal depth of the explanations steps visualized in consequence of a "`why`" command.
The max_evolutions_depth parameter constrains the maximal depth of the explanations steps visualized in consequence of a "`trace`" command without arguments.
The showstuttering parameter defines whether or not to make visible stuttering transitions by adding fake "OUT.lostevent" signals in stuttering transitions (absence of stuttering is usually an interesting property of the system).

Getting help

A brief summary of the available commands is visualized by issuing the "`help`" command (or "`?`").

Exiting

Finally, the main umc loop is exited, and the execution of umc abandoned, by issuing the "exit" command, or simply typing a dot (".").

4.2 Web interface

The Web interface to umc is much more complex and rich of features, since it exploits several other packages and tools developed as part of the project or available from the net.

Among these externally developed tools we must mention the Graphviz package (for the generation of ps and jpg images), the Ghostview package (for the translations of ps info pdf), the Fc2tools package (for the minimization of automata according to some equivalences), Tcl/Tk .
Other internally developed tools are the xmi2umc package (for the extraction of umc classes from an XMI model description of an UML system), the umctotab and tofc2, totdot tools for the translation of a finite UMC model into specifically formatted finite LTS usable by other tools.

The UMC www page is constituted by three main sections: a menu of commands (in the form of buttons) on the left side , a writeable textarea in the upper part, and a set of control buttons and status lights in the bottom.
The writeable text-area is the place where we must define the umc model to analyse (initially it contains the sample model shown in Section 6).
We can introduce the model by directly writing into the area, or by extracting a set of class definitions from a local .xmi file ("`Extract from XMI ...` " button), or by loading an UMC model description from a local text document ("`Load from UMC` ... " button).

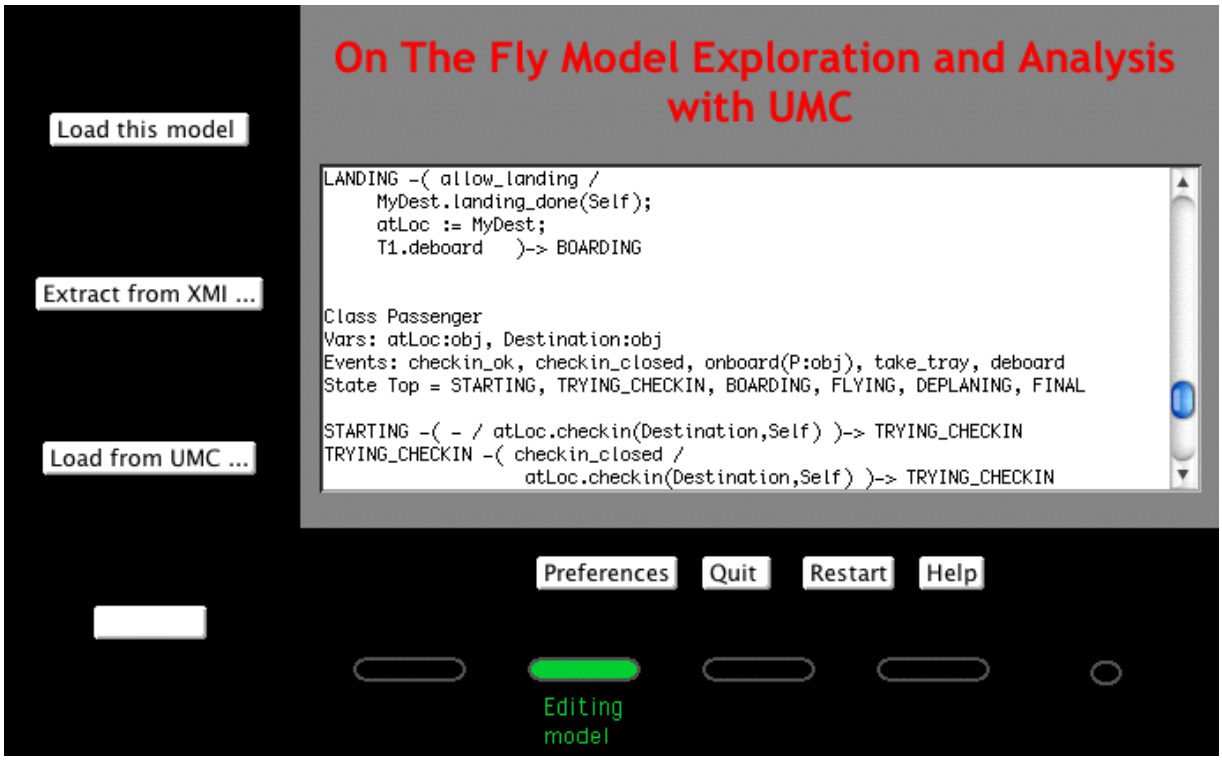

Figure 7 UMC startup page

Once the model has been defined, we should pass to the basic umc tool; this is done with the "Load his model " button. Once loaded, the menu of the left sides changes, as new activities become possible
(supposing the model does not contain syntactic errors)..

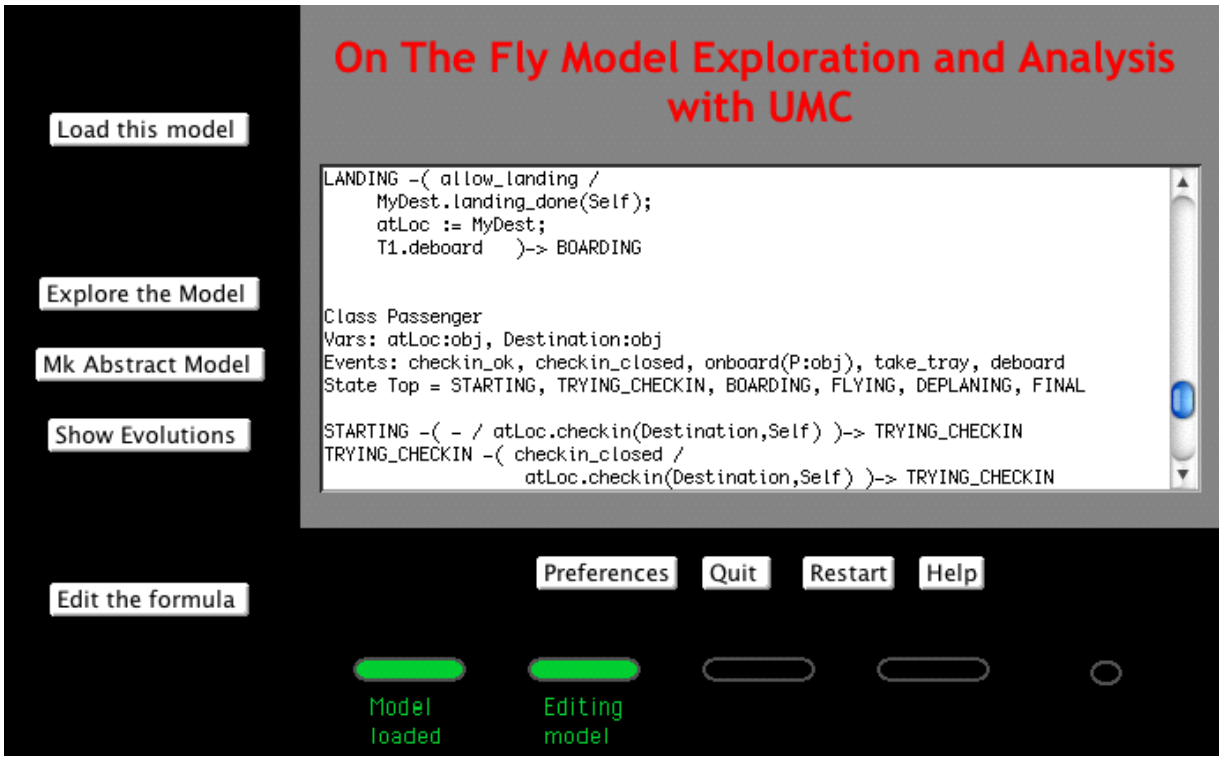

Figure 8 UMC - model loaded page

At this point the possible new activities are the following:

Exploring the configurations

Selecting the "`Explore the Model`" button, a new window is opened which contains a detailed, graphical description of the (initial) configuration. In that window are shown all the details of the active objects (their events queue, the status of their attributes, the set of possible evolutions, and the statechart of the objects with the currently active simple states highlighted in red.
By clicking on one of the possible evolutions of an active object the window is updated with the target configuration reached by forming such evolution.

Observing the evolutions map

Selecting the "`Show Evolutions`" button, a new window is opened which contains a graphical map of the possible system evolutions starting from the current configuration.
The nodes in the graph represent the system configurations and the edges the possible system evolutions.
Edges are labelled with the signals generated by that evolution.
The generated graph is truncated after a certain depth (user customizable). By clicking over one of its deepest leaves a new map is generated starting from that node. Clicking on one of the internal nodes, instead, a new window is opened containing the detailed graphical description of that configuration.

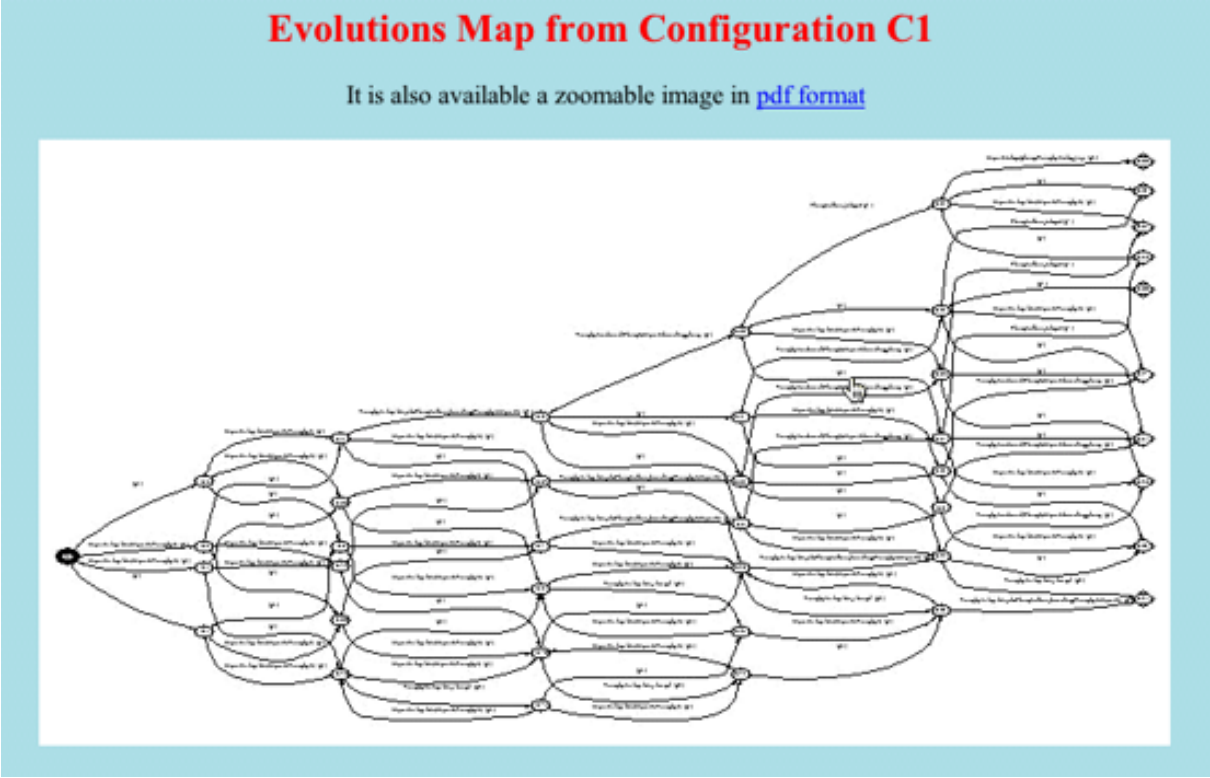

Figure 9 UMC - sample evolutions map

If the image is to big to be reproduced in its original scale, it is scaled down to fix a predefined maximum width. The result, however, can easily become unreadable for complex graphs. For this reason it is provided the possibility to browse the graph embedded into a pdf document, with the advantage of being able to zoom inside complex graphs for observing both its overall structure and all its internal details. Unfortunately with this format it is lost the possibility of clicking over a node to view the details of the specific configuration.

Abstracting the concrete model.

Selecting the "Mk Abstract Model" button it is possible to specify the aspects of the system which we are interested to observe, and generate a simplified model which is an abstraction (minimized w.r.t. divergence sensitive branching equivalence) or the original one but which is equivalent to it w.r.t. the observed aspects.
In particular we can specify that we want to observe only a subset of the set of signal generated by the system,
or we can specify that we are interested to observe the evolutions and the run-time value of certain attributes of certain objects. In the following we show an example of the page which allows the select the observations to which we are interested.

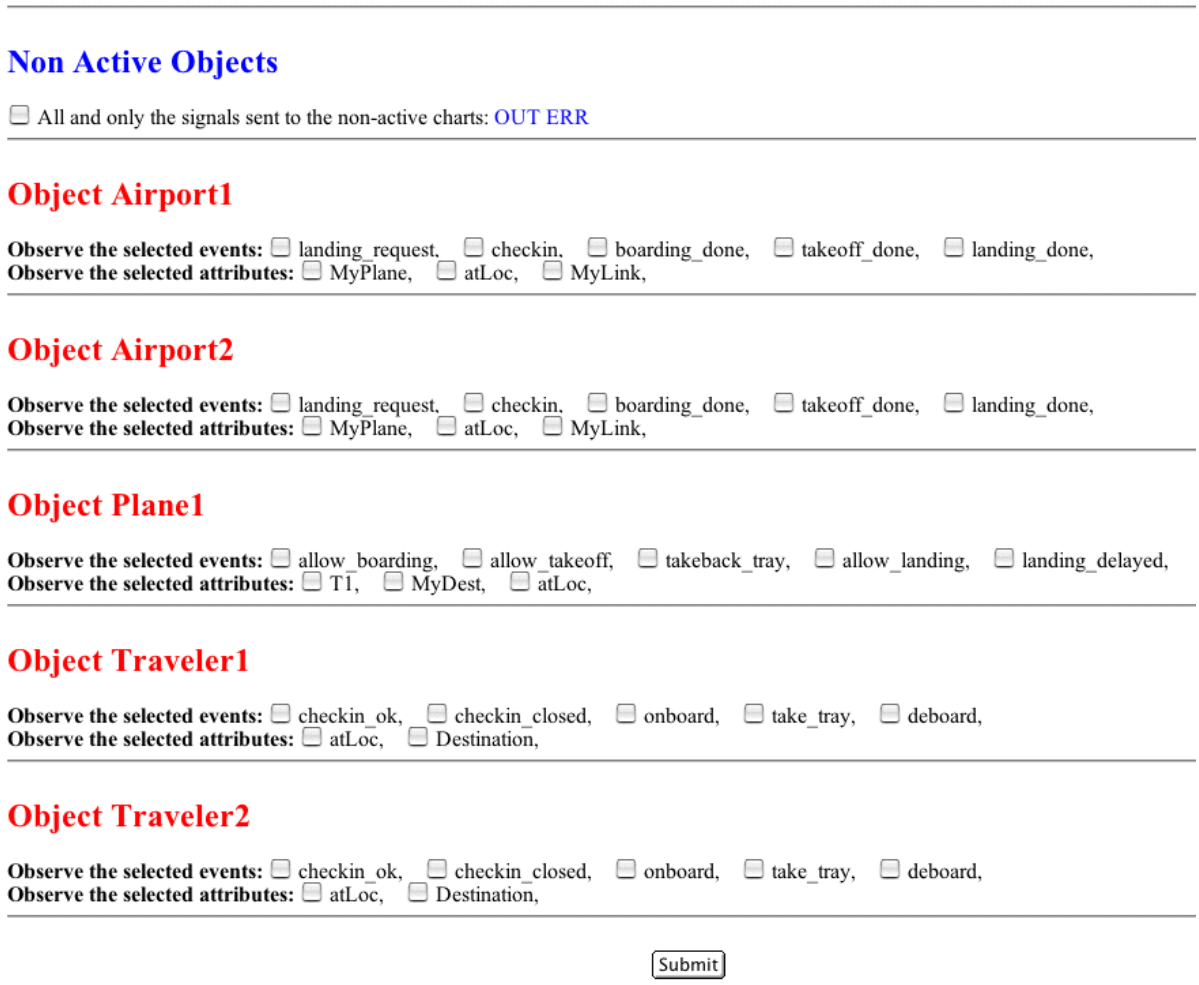

Figure 10 UMC observations selections page

Let us suppose that we are interested to observe only the "`atloc`" attribute of object `Traveler1`. Once that aspect is selected, and the " button clicked, a minimized graph of the possible system evolutions is generated.
The result is shown in the picture below, from which we can easily observe that in the modelled system there is the possibility for Traveler1 to cycle infinitely while with its `atLoc` attribute unchanged, or there is the possibility to eventually change it first to the "`Plane1`" value, and then to the "`Airport2`" value.

Figure 11 : result of observing "atLoc"

Let us now suppose that we are interested to observe the interactions between the object "`Airport1`" and the object "`Plane1`" with respect to the landing phase (i.e. the events landing_request and landing_done of `airport1`).
The result is shown in the picture below, from which we can observe an anomalous behaviour of the system, caused by the fact that there is a possible evolution in which `Plane1` issues a "`landing_request`" to `Airport1`, and that request is followed by an infinite cycle (at S_2) which does not lead to the issuing of the "`landing_done`" signal. Is we would like to

have more details about this behaviour we might generate further abstractions including more events in the observations.

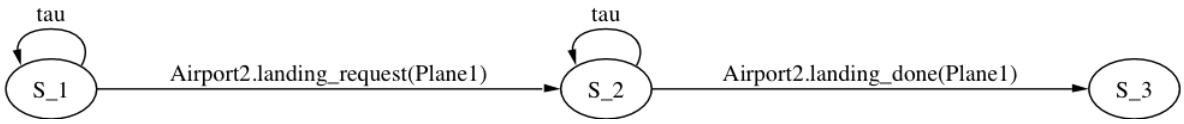

Figure 12 : result of observing "landing_request - landing_done"

Editing and checking a formula

Selecting the "Edit the formula" button the textarea is cleared from the model, and becomes available for the specification of a login formula to be verified (by default there is sample formula the sample model).
Once the formula has been written we start the evaluation selecting the "Eval the formula" button.

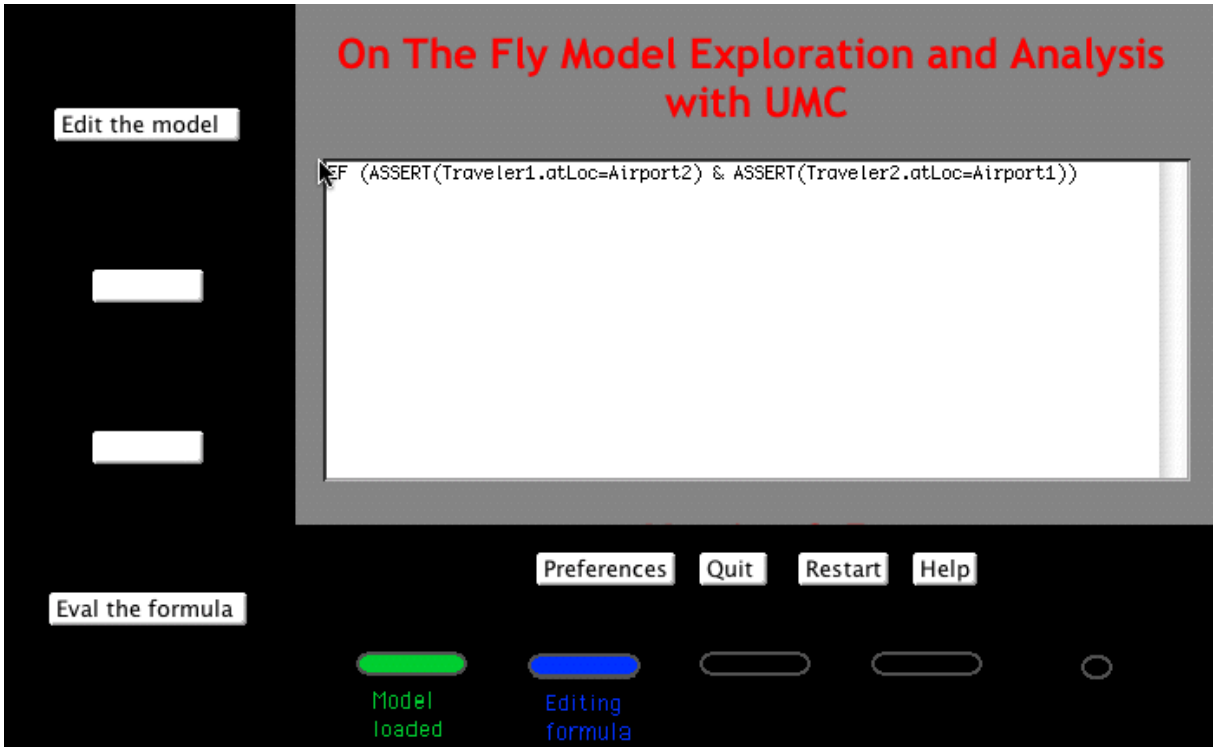

Figure 13: Formula Editing page

Browsing the explanation of the result of the evaluation.

When the evaluation completes, the result of the evaluation is shown in the textarea below the checked formula.
If we want more details on how the result was achieved we can select the "Explain Result" button, which opens a windows in which all the logic steps which have led to the result are explained.

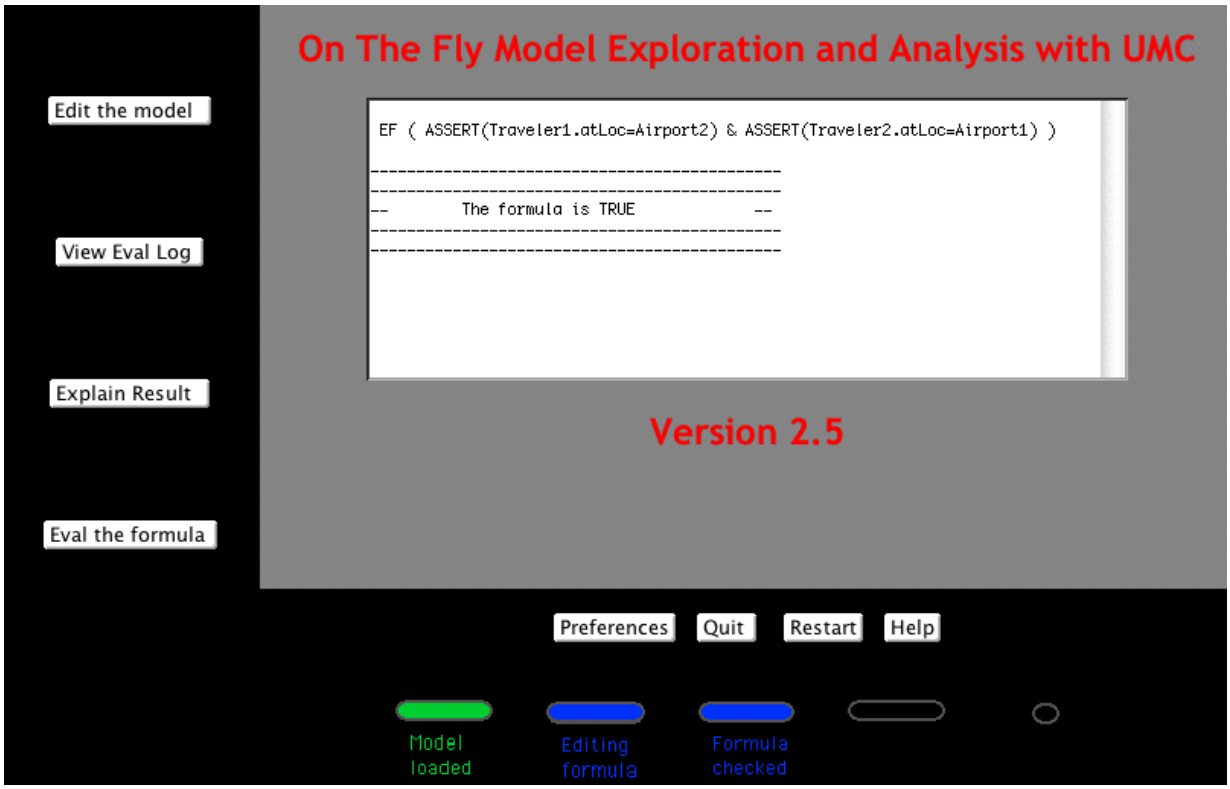

Figure 14: Evaluation done page

In the following is shown just the initial fragment of such explanation.

If we are not interested in the explanation, we can verify another formula on the same model by just modifying the textarea and selecting again the "`Eval the formula`" button.
During the evaluation we can observe some intermediate messages from the umc (e.g. the number of logic steps computed), by selecting the "View Eval Log" button.

Customizing some user preferences

The behaviour of the underlying "umc" tool can be customized according to some parameters, when the "`Preferences`" button is the control part of the window is selected. The possible options are the same as those explained in Section 5.1 under the title "Viewing and adjusting some tool preferences and parametrs".

5 How UMC works

The UMC structure is constituted essentially by 7 main modules. Two modules are constituted by the parser of the UMC textual model description format, and by the parser of the mu-ACTL+ logic formulae. Then we have two archiving modules: a module implementing a database of explored system configurations, and a module implementing a database of started/ completed/ in progress / aborted computations (of logic formulae at configurations). Moreover we have a module which abstracts from the internal details of system configurations, providing abstract iterations routines over the possible evolutions from a configuration into the next reachable configurations. This abstract iteration module is used by two other modules: a logic module which implements the on -the-fly verification algorithm

for mu-ACTL+, and an exploration module which is support the interactive exploration of the system evolutions.

Figure 15. UMC explanation of result

Our approach to the "on the fly" model checking of a m-ACTL+ logic formula has been initially presented in [13]. In that case the system to be verified was defined by a network of synchronised agents working in parallel. The model checker, named FMC, was included in Jack [4], an environment based on the use of process algebras, automata and temporal logic formalisms, supporting many phases of the system development process. The model checker presented here, UMC, is based on the same ideas of FMC, but working over a set of communicating (i.e. exchanging signals) UML State machines. Even though the code for both tools FMC and UMC has been almost completely rewritten several times, the underlying logic schema has remained the same.

The basic idea behind FMC and UMC is that, given a system state, the validity of a formula on that state can be evaluated analysing the transitions allowed in that state, and analysing the validity of some sub-formula in only some of the next reachable states, in a recursive way, as shown by the following schema (Env represents the "current context" in which a given formula is evaluated, which gives a precise meaning (in terms of already started computations) to the free variables of the formula):

```
Evaluate (F: Formula, Env: Environment, S: State) is
  if we have already done this evaluation and
              the result is available then
     return the already known result
  elsif we are already trying to evaluate F in S with Env 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 with Env
     --      (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', Env', S' );
        if the result of Evaluate (F' Env' S') is sufficient
                to establish the result of evaluate (F, Env, 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 with Env;
     Possibly keep track of the performed evaluation and result;
     return the final result
  end if
end Evaluate;
```

Figure 16: Schema of the on the fly evaluation algorithm

This approach seems promising when applied to UML state machines (or groups of communicating state machines) because it can easily be extended also to the case of potentially infinite state space, as it may happen for UML state machines. Indeed, a problem of the above evaluation schema is that, in case of infinite state machines, it might fail to produce a result even for some cases in which a result might be produced in a finite number of steps. This is a consequence of the "depth first" recursive structure of algorithm. The solution taken to solve this problem consists in adopting a bounded model checking approach [2], i.e. the evaluation is started assuming a certain value as maximum depth limit of the evaluation. In this case if a result of the evaluation a formula is given inside the requested depth, then the result holds for the whole system, otherwise the depth limit is increased and the evaluation restarted.
This approach, initially introduced in UMC to overcome the problem of infinite state machines, happens to be quite useful also for another reason. Setting a small initial depth limit, and a small automatic increment of it at each re-evaluation failure, when we finally find a result we can have a reasonable (almost minimal) explanation for it, and this could be very useful also in the case of finite states machines.
More specific details about the dynamic semantics of UML models, and related UMC assumption, are described in Appendix D.

The web interface

The web interface is an additional layer constructed over the basic umc tool. It is constituted by a set of cgi scripts which are called whenever an umc operation is requested, and which allow to subsequently check whether the requested operations is completed. The client-side part of the interface (the umc html page) implements a connectionless kind of interaction with server, asking for operations and subsequently cyclically rechecking whether that operation has been performed (this approach is needed because in general a verification of a formula might require a long time which would trigger internal timeouts both from in the client browser and in the web server side).
The umc client html code makes use of several features (dynamic html, frames, javascript) and seems to works both on reasonably recent browsers like Netscape Navigator from version 4.7 up, and with Internet Explorer.

6 The Airport example

Let as consider as a toy example, a system constituted by two airports, two passengers (one at each airport), and a plane. The plane is supposed to carry exactly one passenger and flies (if it has passengers) between the two airports. Departing passengers try to check in at the airport and than board the plane. We contemplate only one observable action performed by the passengers during the flight, namely the consumption of a meal. The complete dynamic behaviour of the objects of classes Passenger, Airport and Plane is shown below both in the form of statecharts diagrams, and in the umc textual form.

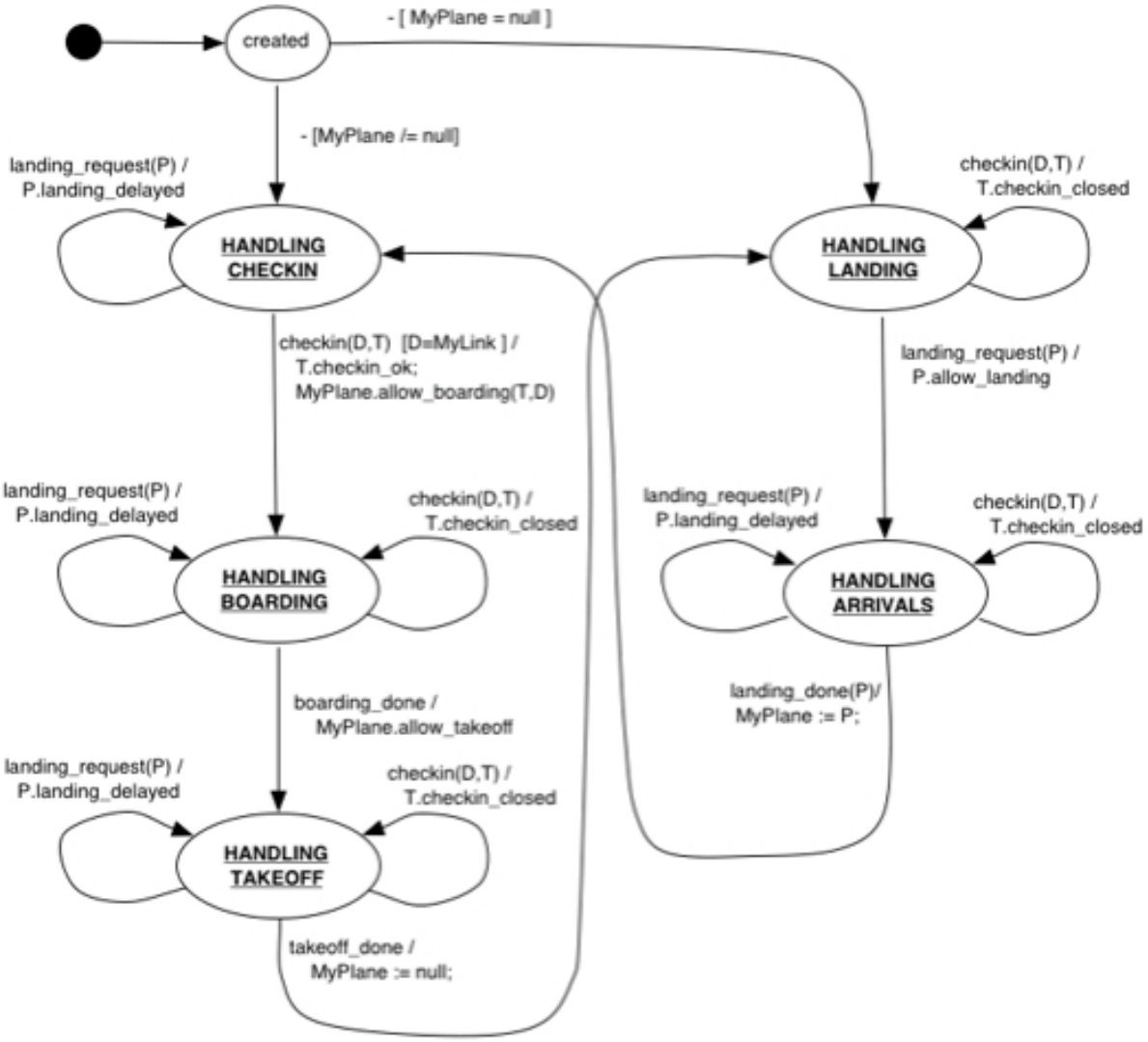

Figure 17 : The airport class statechart

```
Class Airport
Vars: MyPlane:obj, atLoc:obj, MyLink:obj
Events: landing_request(P:obj), checkin(D:obj,T:obj),
         boarding_done, takeoff_done, landing_done(P:obj)
State Top = created, HANDLING_CHECKIN, HANDLING_BOARDING,
     HANDLING_TAKEOFF, HANDLING_LANDING, HANDLING_ARRIVALS
Transitions:
  created -( -[MyPlane=null] )-> HANDLING_LANDING
  created -( -[MyPlane/=null] )-> HANDLING_CHECKIN
  HANDLING_CHECKIN -( landing_request(P) /
      P.landing_delayed )-> HANDLING_CHECKIN
  HANDLING_CHECKIN -( checkin(D,T) [D=MyLink] /
      T.checkin_ok;
      MyPlane.allow_boarding(T,D)-> HANDLING_BOARDING
  HANDLING_BOARDING -( landing_request(P) /
      P.landing_delayed )-> HANDLING_BOARDING
  HANDLING_BOARDING -( checkin(D,T)  /
      T.checkin_closed )-> HANDLING_BOARDING
  HANDLING_BOARDING -( boarding_done /
      MyPlane.allow_takeoff )-> HANDLING_TAKEOFF
  HANDLING_TAKEOFF -( landing_request(P) /
      P.landing_delayed )-> HANDLING_TAKEOFF
  HANDLING_TAKEOFF -( checkin(D,T)  /
      T.checkin_closed )-> HANDLING_TAKEOFF
  HANDLING_TAKEOFF -( takeoff_done /
      MyPlane := null )-> HANDLING_LANDING
  HANDLING_LANDING -( checkin(D,T)  /
      T.checkin_closed )-> HANDLING_LANDING
  HANDLING_LANDING -( landing_request(P) /
      P.allow_landing )-> HANDLING_ARRIVALS
  HANDLING_ARRIVALS -( landing_request(P) /
      P.landing_delayed )-> HANDLING_ARRIVALS
  HANDLING_ARRIVALS -( checkin(D,T)  /
      T.checkin_closed )-> HANDLING_ARRIVALS
  HANDLING_ARRIVALS -( landing_done(P) /
      MyPlane := P )-> HANDLING_CHECKIN
```

Figure 18 : The textual representation of Airport statechart

```
Class Plane
Vars: T1:obj, MyDest:obj, atLoc:obj
Events: allow_boarding(T:obj,D:obj), allow_takeoff, takeback_tray,
        allow_landing, landing_delayed
State Top = BOARDING, LEAVING, FLYING, LANDING
 BOARDING -( allow_boarding(T,D) /
     T1 := T; MyDest := D;
     T1.onboard(Self);
     atLoc.boarding_done )-> LEAVING
LEAVING -( allow_takeoff /
     atLoc.takeoff_done;
     atLoc := null;
     T1.take_tray )-> FLYING
FLYING -(takeback_tray /
     MyDest.landing_request(Self) )-> LANDING
LANDING -( landing_delayed /
     MyDest.landing_request(Self) )-> LANDING
LANDING -( allow_landing /
     MyDest.landing_done(Self);
     atLoc := MyDest;
     T1.deboard   )-> BOARDING
```

Figure 19 : The textual representation of Plane statechart

```
Class Passenger
Vars: atLoc:obj, Destination:obj
Events: checkin_ok, checkin_closed, onboard(P:obj), take_tray, deboard
State Top = STARTING, TRYING_CHECKIN, BOARDING, FLYING, DEPLANING, FINAL

STARTING -( - / atLoc.checkin(Destination,Self) )-> TRYING_CHECKIN
TRYING_CHECKIN -( checkin_closed /
                  atLoc.checkin(Destination,Self) )-> TRYING_CHECKIN
TRYING_CHECKIN -( checkin_ok ) -> BOARDING
BOARDING -( onboard(P) /  atLoc := P )-> FLYING
FLYING -( take_tray /
            OUT.eating(Self); atLoc.takeback_tray )-> DEPLANING
DEPLANING -( deboard / atLoc := Destination )-> FINAL
```

Figure 20 : The textual representation of Passenger statechart

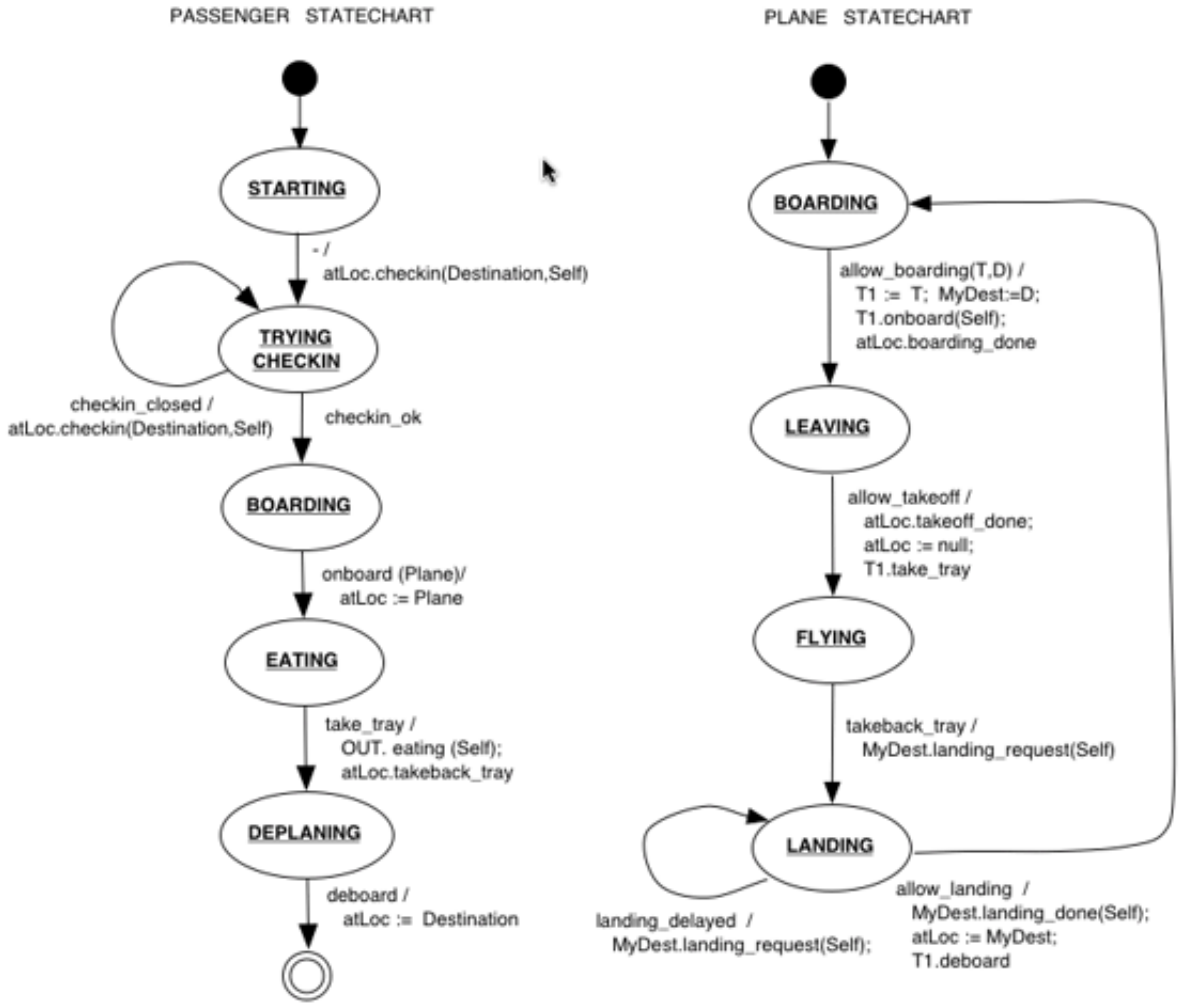

Figure 21 : The Plane and Passenger class statechart

The initial deployment of the system is defined by the following Object declarations

```
          OBJECT  CLASS          NITIAL VALUE FOR ATTRIBUTES
Object Airport1 : Airport (MyLink => Airport2, MyPlane => Plane1)
Object Airport2 : Airport (MyLink => Airport1)
Object Traveler1: Passenger (atLoc => Airport1, Destination => Airport2)
Object Traveler2: Passenger (atLoc => Airport2, Destination => Airport1)
Object Plane1 : Plane (atLoc => Airport1)
```

An example of property which can be verified is the following:

It is always true that `Traveller1` performs an "eating" operation only while flying on Plane1. This property can be written in mu-ACTL as:

```
AG ((EX{eating(Traveler1)} true) ->
    (ASSERT(Traveler1.atLoc=Plane1) & ASSERT(Plane1.atLoc=null))
```

7 Related Works and Conclusions

Linear-time model checking of UML Statechart Diagrams is addressed in [19], [10] and [24]. In [11] a simple (branching time) model-checking approach to the formal verification of UML Statechart Diagrams was presented exploiting the “classical” model checking facilities provided by the AMC model checker available in JACK. We are currently aware of three available tools for model checking UML systems described as sets of communicating state machines. HUGO [22,26] and *v*UML [22] take the approach of translating the model into the Promela language using SPIN [15] as the underlying verification engine. We have not had direct experience with these tools, but clearly in this case the properties to be verified need to be mapped into LTL logic. While *v*UML is restricted to deadlock checking, HUGO is mainly intended to verify whether certain specified collaborations are indeed feasible for a set of UML state machines. In both cases, the UML coverage of the tools is wider than ours because it includes UML call operations, history states, and internal state activities. A timed version of HUGO (called HUGO/RT [17]) has also been developed, which maps into the UPPAAL verification engine, instead than into SPIN.
A third interesting approach is that one adopted in the ongoing UMLAUT [16,28] project. In this case an UML execution engine has been developed, adopting the Open Caesar standard interface of the CADP environment. In this way all the CADP [8] verifications tools (including the “on the fly” Evaluator tool [24]) can be applied also to this new engine.

As far as we now, FMC and UMC are the only "on the fly" tools supporting full m-calculus (SPIN uses LTL, CADP Evaluator the alternation free m-calculus).
The fact of being able to state and check also structural properties of system configurations (state attributes and predicates) and not just events, opens the door to the modelling and verification of several structural properties of parallel systems, like topologic issues, state invariants, and mobility issues.

The approach adopted in UMC seems promising but there is still a lot work to do. Certainly the UMC coverage must be extended to include at least call operations, events deferring, and state internal activity. Moreover the semantic / logic issues still need to be assessed (i.e. precisely which kind of property do we want to verify, and which kind of optimisations do they allow to be implicitly performed by the tool).

8 Known limitations, bug reports, feedback, availability, acknowledgements

UML Issues

UMC does not currently support all the features of UML1.4 state machines.
In particular, UMC suffers the following limitations:

Events: Only asynchronous signals are supported. Call events, time events, change events, events deferring are not supported.

States: Internal transitions, Enter / Exit/ Do activities, are not supported. History states, Sync states, Choice pseudo-states are not supported.

Transitions: Initial default transitions do not have actions, static and dynamic choice transitions are not supported. Completion transitions cannot appear in more than one region of concurrent state.

Other: Sub-machines are not supported. Actions can only be simple assignments and sending of signals.
The only data type for variables and signal parameters is constituted by 32 bits integers. Boolean and Integer expressions have some simplification.

Some of the missing features might be added in future versions of UMC. We do not see any intrinsic difficulty in modelling also the currently omitted aspects, even if for the purposes of the umc project their support is probably not essential.

WWW Issues

Because of a problem in the Graphviz package in the evolutions map ("Explore the Model" button), when the image must be scaled down to fit into the maximum width (i.e. not shown at 100% scale), the correspondence of the nodes with respect to the clickable area is sometimes lost.
For some unknown reasons, Internet explorer sometime produces a "network error" when trying an operation.
Repeating the operation a second time usually works.

UMC Issues

Performance is definitely not a target for the current version of the tool. We are currently more interested in experimenting with user friendliness and easiness of verification. Many strong optimisations could be done.

Availability

The UMC basic tool is constituted by a single program written in Ada and its binary code (easily portable and compilable for many platforms) is freely available. For source code distribution please contact the author.
The WWW interface is currently accessible from http://matrix.iei.pi.cnr.it/FMT/Tools (there is no guarantee that it will remain accessible in the future). At the present time some work is still needed in order to make it easily portable to other systems).

Acknowledgements

The XMItoUMC translation has been developed in tcl-tk by Marco Fabbri.
The airport example has been developed in the context of the AGILE[19] project.
Useful feedback and suggestions have come also from the QUACK[20] and PRIDE[21] projects, and from many colleagues among which Stefania Gnesi, Sandro Fantechi, Diego Latella and Mieke Massink. Parts of this report have been extracted from [10].

Feedback

Please submit bug-reports, comments and suggestions to franco.mazzanti@isti.cnr.it.

8 References

1. M. von der Beeck, Formalization of UML-Statecharts, UML 2001 Confrence, LNCS 2185, Springer-Verlag, pp. 406-421, 2001.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, Symbolic Model Checking without BDDs, TACAS'99, LNCS 1579, Springer-Verlag 1999.
3. A. Bouali, S. Gnesi, S. Larosa. The integration Project for the JACK Environment, Bulletin of the EATCS, n.54, pp. 207-223, 1994.
4. E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic Verification of Finite--State Concurrent Systems Using Temporal Logic Specification, ACM Transaction on Programming Languages and Systems, 8, pp. 244-263 (1986).
5. R. De Nicola, F. W. Vaandrager. Action versus State based Logics for Transition Systems, Proceedings Ecole de Printemps on Semantics of Concurrency, LNCS 469, pp. 407-419, 1990.
6. 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.
7. J-C Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R Mateescu, and Sighineanu M., CAPD: a Protocol Validation and Verification Toolbox. In CAV '96, LNCS 1102, Springer-Verlag 1996, see also http://www.inrialpes.fr/vasy/cadp/.
8. S. Gnesi, D. Latella, and M. Massink. Model checking UML statechart diagrams using JACK. In A. Williams, editor, Fourth IEEE International High-Assurance Systems Engineering Symposium, pages 46--55. IEEE Computer Society Press, 1999.
9. 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).
10. S. Gnesi and F. Mazzanti, On the Fly model checkling of communicating UML State Machines (submitted for publication)..
11. M. Hennessy, R. Milner. Algebraic Laws for Nondeterminism and Concurrency, JACM 32:137-161, 1985.
12. G.J. Holzmann, The SPIN Model Checer, IEEE TSE 23 (1997), pp. 279-295

---

[19] AGILE project IST-2001-32747 by the proactive Initiative on Global Computing
[20] QUACK, A Platform for the Quality of New Generation Integrated Embedded Systems (Pr. MURST 40%)
[21] PRIDE, ambiente di PRogettazione Integrato per sistemi DEpendable(Italian Space Agency, 2002)

13. W.M. Ho and Le Guennec A. Pennaneac'h F. Jézéquel, J.-M. -- Umlaut: an extendible UML transformation framework, INRIA-RR-3775, 1999. http://www.inria.fr/RRRT/RR-3775.html.
14. A. Knapp, S. Merz, and C. Rauh, Model Checking Timed UML State Machines and Collaborations, FTRTFT 2002: 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems. Springer LNCS, to appear, 2002.
15. D. Kozen. Results on the Propositional m-calculus, Theoretical Computer Science, 27:333-354, 1983.
16. Jacobson, I., Booch, G., Rumbaugh J. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.
17. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. Technical Report CNUCE-B4-1999-008, Consiglio Nazionale delle Ricerche, Istituto CNUCE, 1999.
18. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. IFIP TC6/WG6.1 FMOODS '99, pages 331--347. Kluwer Academic Publishers, 1999.
19. J. Lilus, I. Porres Paltor, vUML: a Tool for Verifying UML Models,14th IEEE International Conference on Automated Software Engineering, (ASE'99), pp.255-258 1999.
20. http://www.pst.informatik.uni-muenchen.de/projekte/hugo/
21. R. Mateescu, M. Sighireanu: Efficient On-the-fly Model-Checking for regular Alternation-Free m-Calculus. Science of Computer Programming 46(3) 2003.
22. OMG Unified Modeling Language Specification,Version 1.4 beta R1, November 2000, http://www.omg.org/technology/documents/formal/uml.htm).
23. T. Schäfer., A. Knapp and S. Merz, Model Checking UML State Machines and Collaborations, Proc. Wsh. Software Model Checking, 55(3) Electronic Notes in Theoretical Computer Science, Paris 2001
24. UML All pUrposes Transformer, http://www.irisa.fr/pampa/UMLAUT
25. R. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class and behavioral diagrams. ICSE'98 Workshop on Precise Semantics for Software Modeling Techniques, 1998.

Appendix A Grammar Meta-Notation

Meta-Notation used in grammar rules: Non_terminal symbols in the lexical grammar are shown in italic. The symbol | means choice, [ ] means zero or one occurrences , { } means zero or more occurrences .
In the current version of UMC boolean and integer expressions over integer values and variable names are restricted to a very simple form.

Basic Grammar Items:

```
<id>       ::= <letter> { <letter> | <digit> | <underscore> }
<composite_name> ::= <id>{.<id> }
<int_literal> ::=  <digit>{<digit>}

<bool_expr>  ::=
      <simplebool_expr>
      <intbool_expr >          |
   ~  <intbool_expr >          |
      <intbool_expr >   &  <intbool_expr >   |
      <intbool_expr >   |  <intbool_expr >

<intbool_expr >   ::=
    <int_expr>  <   <int_expr>
    <int_expr>  >   <int_expr>
    <int_expr>  =   <int_expr>
    <int_expr>  !=  <int_expr>
    <int_expr>  <=  <int_expr>
    <int_expr>  >=  <int_expr>

<int_expr> ::=
    <simple_intexpr>   |
  <simple_intexpr>  +  <simple_intexpr>   |
  <simple_intexpr>  -  <simple_intexpr>   |
  <simple_intexpr>  *  <simple_intexpr>   |
  <simple_intexpr>  /  <simple_intexpr>   |
  <simple_intexpr>  mod  <simple_intexpr>

<obj_expr> ::=  <id> | self | null
<simplebool_expr> ::= <id> | true | false
<simple_intexpr> ::=     <id>   |    <int_literal>
<simple_expr> ::= <simple_intexpr> | <obj_expr> | <simplebool_expr>
<expr> ::=  <int_expr> | <bool_expr> | <obj_expr>
```

Appendix B UMC Model Grammar

```
SYSTEM ::=   { CLASS |  OBJECT }

CLASS::=
     Class  <class-id>
  [  Events: EVENTS_LIST  ]
  [  Vars:   VARS_LIST    ]
     STATES_STRUCTURE
  [  Transitions:
      TRANSITIONS_LIST    ]

VARS_LIST   ::= <VAR_DECL> {, <VAR_DECL> }
EVENTS_LIST ::= <EVENT_DECL>  {, <EVENT_DECL>  }
<VAR_DECL>    ::=  <var-id> [: <type>] [ = <simple_expr> ]
<EVENT_DECL>   ::=  <event-id> [ ( <PARAM_LIST>  ) ]
<PARAM_LIST> ::= <PARAM_DECL>  {, <PARAM_DECL>  }
<PARAM_DECL>  ::=  <param-id> [: <type>]
<type> ::=  int | obj | bool

STATES_STRUCTURE  ::=  TOP_STATE { SUBSTATES }

TOP_STATE ::=  COMPOSITE_SEQUENTIAL_STATE

SUBSTATES  ::=  COMPOSITE_SEQUENTIAL_STATE   |
                COMPOSITE_PARALLEL_STATE

COMPOSITE_SEQUENTIAL_STATE   ::=
    State <composite_name>  =  <substate-id>{ , <substate-id>}

COMPOSITE_PARALLEL_STATE   ::=
   State <composite_name>  =
       <substate-id> //  <substate-id> {  // <substate-id> }

TRANSITIONS_LIST ::=   TRANSITION  { TRANSITION }

TRANSITION  ::=
  [LABEL:] SOURCE  -->  TARGET     |
  [LABEL:] SOURCE -( TRIGGER [GUARD] [/ ACTIONS ])-> TARGET

SOURCE ::=
  <composite_name> | (<composite_name> {,<composite_name> })

TARGET::=
  <composite_name> | (<composite_name> {,<composite_name> })

LABEL      ::=  <label-id>
TRIGGER    ::=   -  | <EVENT-DECL>
GUARD      ::=  [ <bool_expr> ]
ACTIONS    ::=  ACTION { ;  ACTION }
ACTION     ::=  SIGNAL  |    ASSIGNMENT
SIGNAL     ::=  [ DESTINATION .  ]  <EVENT_INSTANCE>
DESTINATION ::=  <obj-id>
ASSIGNMENT ::=  <var-id> := <expr>

<EVENT_INSTANCE> ::=   <event-id> [ ( <expr> {, <expr>} ) ]

OBJECT ::=  Object <id> : <class-id> [ ( <initial values> ) ]

<initial values> ::=  <initial-value> [, <initial values>]
<initial-value>  ::=  <var-id> => <simple_intexpr>
```

Appendix C mu-ACTL+ Logic Grammar

```
FORMULA ::=            STATEPREDICATE     |   . . .

STATEPREDICATE ::=
   true  | false | ASSERT( <state_assertion> )
```

The only kind of assertions allowed are predicated of the form var=value, where var is the name of some chart variable, and value is an integer value.

```
<state_assertion> ::=  [<object-id>.]<var-id> = <static_value> |
                       [<object-id>.]<var-id> > <static_value> |
                       [<object-id>.]<var-id> < <static_value> |

<static_value> ::= <int_literal> | true | false | null | <object-id>

< FORMULA> ::=
      ~ FORMULA  |
        FORMULA  &  FORMULA  |
        FORMULA  |  FORMULA    |
        FORMULA  ->  FORMULA   |
        ( FORMULA  )   |
        EX {  <action-predicate> }  FORMULA    |
        AX {  <action-predicate> }  FORMULA    |
        max <fixpointid>: FORMULA  |
        min <fixpointid>: FORMULA  |
        <fixpointid>  |
        EF  FORMULA   |
        AF  FORMULA   |
        EG  FORMULA   |
        AG  FORMULA   |
        E[ FORMULA U FORMULA ]  |
        A[ FORMULA U FORMULA ]  |
        E[ FORMULA { <action-predicate> }  U FORMULA ]  |
        A[ FORMULA { <action-predicate> }  U FORMULA ]  |
        E[ FORMULA { <action-predicate> } U <action-predicate> FORMULA ]  |
        A[ FORMULA { <action-predicate> } U <action-predicate> FORMULA ]  |
        < <action-predicate> > FORMULA   |
        [ <act> ] FORMULA

<action-predicate> ::=    <action-expression>  |
      ~ <action-predicate > |
        <action-predicate>  & <action-predicate> |
        <action-predicate> | <action-predicate>   |
...     ( <action-predicate> )

 < action-expression >  ::=
     true | false | tau | [<obj_id>.]<event_id>[<event_args>]

<event_args> ::=  (<simple_intexpr>[,<simple_intexpr>])
```

Shortcuts:
The following shortcuts are defined to improve the compatibility with ACTL/CTL expressions, or with the logics supported by FMC.

```
FORMULA  ::=
        ET FORMULA  | -- this is a shortcut for EX{ tau} FORMULA
        AT FORMULA  | -- this is a shortcut for.AX{ tau} FOMULA
        EX FORMULA  | -- this is a shortcut for.EX{ true | tau} FORMULA
        AX FORMULA  | -- this is a shortcut for.AX{ true | tau} FORMULA
        FINAL       | -- this is a shortcut for ~ EX{ true | tau}
```

Appendix D Overview of the Dynamic semantics of UMC models

In the UML-1.4 [26]standard definition there is a first attempt to assign a reasonably defined dynamic semantics (i.e. the possible behaviours) to the state machine associated with a statechart. The basic concept used in the standard to define the possible evolutions of a the state machine configuration is the concept of "run to completion" step .
UMC follows these standard indications, with a few simplifications due to the set of UML features not yet supported by UMC. From a logic point of view , the possible evolutions of from a given state machine configuration can be discovered by performing the following substeps :

a) Dealing with active states, triggers and guards: It is identified the set of transitions whose source states are active in the current configuration, whose trigger satisfies the current top event (if any) of the chart events queue, and whose guards evaluate to true in the current configuration. The resulting set is called the set of enabled transitions (w.r.t. active states, trigger, guards).

b) Dealing with priorities: According to the relative priority between transitions (which is a partial ordering), we find a maximal subset of the transitions identified at the previous step so that:
- there are no two transition inside the set, of which one has a priority lower than the priority of the other.
- there are no transitions inside the set with a priority which is lower than the priority of any other transition outside the set.

c) Dealing with conflicts: Given the set of maximum priority enabled transitions (some of which might be executed in parallel) we must find all its maximal subsets, such that no two transitions in the subset are in conflict (two transitions are in conflict if the intersections of the set of states they exit is not empty.
Notice that if a statechart has no parallel substates then each of these subsets will contain exactly one transition. These subsets represent a set of concurrently fireable transition.

d) Dealing with serialisation: For each subset identified at the previous step, if the subset contains more than one transition, we generate the set of all the possible sequences of transitions deriving from all the possible serialisations of the transitions in the subset.
Each such sequence of transitions defines a possible evolution of the given machine configuration.

e) Computing the target configuration: The final state-machine configuration resulting after this evolution if obtained by:
- removing the top event (if any) from state machine event queue.
- modifying the values of the state machine variables as specified by the sequence of sequences of actions as requested by the firing transitions.
- modifying the events queue of the state machine by adding the signals specified by the sequence of sequences of actions, in their order.

UMC allow to describe a system as a list of state-machines communicating through the asynchronous exchange of signals. In the current version of UMC the overall semantics for this set of state machine is simply an interleaving semantics (i.e. any single state-machine step is allowed to become a full system transition).

Notice that the set of possible evolutions of an initial model are, in general, not finite.
In fact, even if we consider only limited integer types (which is a reasonable assumption), we can still have infinitely growing queues of events. The following is an example of very simple model presenting an infinite behaviour:

```
Chart Main
Events: a
State Top =  s1
s1 -( a / a;a )-> s1
```

When coming to give a formal framework to the above informal description of a run-to-completion step, and when coming to model the parallel evolution of state machines, some aspects which are not precisely and univoquely defined by the UML standard (often intentionally) have to be in some way fixed (other approaches for the definition of a formal semantics of UML Statecharts are presented in [1,21,29].
With respect to this, UMC makes certain assumptions which, even if compatible with the UML standard, are not necessarily the only possible choice.

1) The whole sequence of actions constituting the actions part of statechart transition, is supposed to be executed as an indivisible atomic activity, i.e. two parallel statechart transitions, fireable together in the current state-machine configuration, cannot interfere one with the other, but they are executed in a sequential way (in any order).[22]

2) Given a model constituted by more than one state machine, a system evolution is constituted by any single evolution of any single state machine. I.e. state-machine evolutions are considered atomic and indivisible.[23]

3) The propagation of signals inside a state machine and among state machines is considered instantaneous, and loss free. [24]

4) The events queue associated with a state machine handles its events in a FIFO way.[25]

5) The relative priority of a join transition is always well defined and statically fixed.[26]

---

[22] This assuption is harder to defend when synchronous call operations are taken into account.
[23] This assuption is harder to defend when synchronous call operations are taken into account.
[24] This is an aspect intentionally left as unspecified by the standard.
[25] This is an aspect intentionally left as unspecified by the standard.
[26] This assumption is related to an ambiguity of the UML definition of priority of join transitions, in which all the sources have the same "depth". In this cases the priority being defined as that of the "deepest source" leaves some open space to multiple interprations when there is not a unique "deepest source".