UMC User Guide
(version 3.3)

Franco Mazzanti

July 2006

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

Abstract

In this report we present the prototypical UMC verification tool under development at ISTI.
UMC accept a system specification given in UML-like style as a collection of active objects, modelled by state-machines, and whose behavior is described through statecharts. On such systems UMC allows to verify properties specified in the mu-UCTL logic: a temporal logic which enriches the full mu-calculus with the more abstract (and weak) CTL/ACTL like temporal operators, and with a rich set of state propositions and ACTL action expressions.
Both the basic comand-line oriented tool (umc) and its more user-friendly web-based interface are presented. This web interface integrates also verification functionalities provided by the other environments (EST, FC2TOOLS) which allow system abstraction and minimization.

Table of Contents

1 Introduction

The goal of the UMC project [26,27] is to experiment in several 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 (and in particular in the statecharts technology) for the specification and verification of the dynamic behaviour of a system.

- We are interested in experimenting with several flavours of temporal logics which allow to take into consideration both the state oriented and the event oriented aspects of a system, toghether with distribution and mobility aspects.

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 commercal verification product (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.
So far the emphasis of the prototype development has been concentrated on the investigation of the desirable supported fetures, and not yet on the quantatitive optimizations of them (in tems of complexity, memory resources, performance, stability). As such the prototype in its current proof-of-concept status is good for education purposes and academic experimentations, but definitely not ready for an official public release or for use in a real industrial environment.

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-UCTL 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).

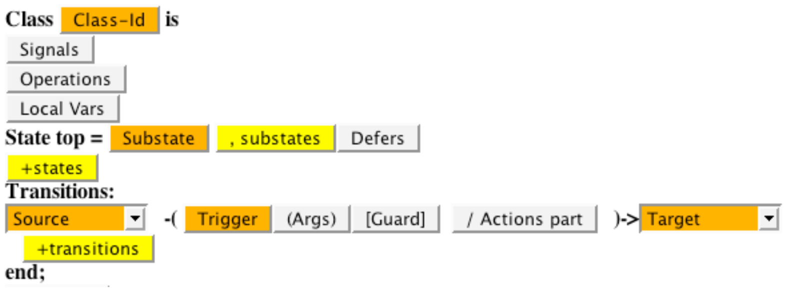
2 The structure and semantics of UMC models

A complete UMC model description is given by providing by a set of class definitions and a set of object instatiations. Class definitions represent a template for the set of active or non-active objects of the system. In the case of active objects a statechart diagram associated to class is used to describe the dynamic behaviour of the corresponding objects.

A state machine (with its events queue) is associated to each active object 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.
A system is constituted by fixed static set of objects (so far no dynamic object creation), and a system is an “input closed” system, i.e. the input source is modelled as an active object interacting with the rest of the system.
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 the following section we describe in more detail the two syntactic model components, namely classes and objects, in the subsequent one we describe an overview of their semantics.

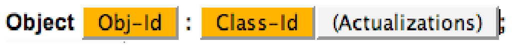
Classes and Objects

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 (signals or call operations)
- the list of private attributes (variables) local to the objects of the class
- the structure of the states of the class (nodes of the class statechart diagram)
- the transitions of the objects of the class (edges of the class statechart diagram)



In the case of non-active objects, the corresponding class declations can only define the list of accepted Signals and Operations (no Local Vars, States or Transitions can be introduced).

Once the needed classes are have their behavior 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).

Predefined Types

The parameters in the interface profile of the events supported by a class (signals and operations), the class local variables, and the class transition variables can be of one of the system predefined types. These types are the basic "int", "bool" and "obj" types, or a Class type denoted by a `Classname`, or the corresponding vector version of them ("`int[]`", "`bool[]`" and "`obj[]`" types, or a `Classname[]`.) The default value for an integer variable is "`0`", for a boolean variable is "`false`", for an object is "`null`", for vectors is the empty vector "`[]`".

Expressions and Identifiers

An integer expression is either an integer literal, the name of an integer attribute or parameter, or a parentesized integer expression, a vector length expression, a selection from an integer vector, or an composite expression built over one of the integer operands "`+`", "`-`" ,"`*`", "`/`", "`mod`" .

An object expression is either "`self`" or "`this`" or "`null`" or the name of object attribute or parameter (both "`self`" and "`this`" denote the executing object), or a selection from an object vector.

A boolean expression is either the boolean literal "`true`" or "`false`", or name of a `bool` attribute, a selection from an bool vector, an equality comparison between two object or vector expressions (`obj_expr1 = obj_expr2, obj_expr1 /= obj_expr2`), a relational expression between two integer expression (int_`expr1` relop `intexpr2`, where relop is `=`, `/=`, `>`, `<`, `>=`, `<=`), or a sequence of boolean expression all joined by an "`and`" operator or all joined by an "`or`" operator (`boolexpr1 and boolexpr2 and` ...) .

A vector expression is consituted by list of comma-separated items enclosed by brakets "`[obj1, obj2, obj3]`". The allowed operations over vector values are:
- the selection of an item of the vector "`vector[index]`" (where index ranges from 0 to vectorsize-1),
- the selection of the first element of the vector "`vector.head`" (equivalent to `vector[0]`),
- the selection of the rest of the vector once the first element has been skipped "`vector.tail`"
- the evaluation of the vector length "`vector.length`" (which is an integer value),
- the concatenation of two vectors "`vector1 + vector2`".

Identifiers are case sensitive and built over letters, digits and '_' (underscores).

Erroneous selection operations on incorrectly sized vectors (indexing, head) just return a default value for the type.

Signals and Operations

The events handled by the class are distinguished between asynchronous signals and synchronous call operations; they are correspondingly declared in the "**Signals**:" and

Operations:” sections of a class. Call operations can also have a return type which is one the predefined types.

```
Signals:    Sig-Id (Params) +signals
Operations: Op-Id (Params) :returntype +operations
```

Parameters of events can be explicitly typed, and they do have only an implicit “in” mode.

It is not statically enforced/checked the fact that an object should receive only events defined in its class interface, if such spurious undeclared events are sent, they will be discarded when dispatched.

Private Attributes

The attributes (local variables) defined for the objects of a class are defined in the “**Vars**:” section of a class. They can be explicitly typed and they can be provided with an initial (static) value. If no explicit initial value is specified, the default value for the type is used.

```
Var-Id :type := literal
```

Statechart Structure

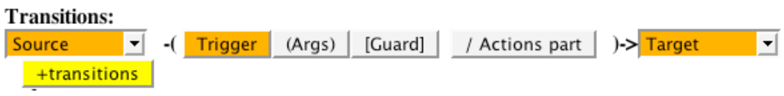
The statechart structure of a class is given by defining the nested structure of the states, and the set of transitions which connects them.
The nested structure of the states is defined by a sequence of composite state definitions which starts from the definition of the top level state. The definition 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, composite parallel states or simple states). The name “initial” denotes the default initial pseudostate of a composite sequential state (and must appear as first substate); if no “initial” pseudostate is explicitly provided, the first substate of the sequence is implicitly assumed as 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.

```
State State-Id = Substate , substates Defers
State State-Id = Region / Region / regions Defers
```

A composite state definition can also define the set of events deferred while active.
A Defers clause defines the list of events (matching the profile as declared in the Signals or Operations sections of the class) to de deferred.

```
Defers Sig-Id (Params) +defers
```

The set of transitions of the class statechart is defined in the "**Transitions:**" section of the class. The definition of a transition contains 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 (exactly 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. `source -> target` is a shortcut for `source -(-)-> target`.
The guard (if present) is a simple form of boolean expression (involving the object attributes).
If the trigger of a transition is a synchronous event (call operation) then an explicit "`return`" action must be executed by the object in the same transition if we want to notify the completion of the call. Moreover, during the execution of a transition triggered by a synchronous call an implicit "`_caller`" parameter of type obj is available, such parameter identifies the calling object and can be used to notify the completion of the operation (maybe also from subsequent transitions) by the explicit sending of `calling.return` signal (this allows the support of some kind of "delegations").

The Action Language

The Actions part of a transition is a sequence of simple or composite actions.

A simple action can be an *assignment* of an expression to a (local or transition) variable, the sending of an asynchronous *signal* to a target object, the calling of a synchronous *call* operation on a target object, an assignment of a synchronous *function call* to a (local or transition) variable, the declaration of a temporary *variable*, the *exit* from a loop, or the *return* from an operation call.

A signal is similar to an event declaration, but its arguments are constituted by value expressions instead that by formal parameters. A signal is preceded by a target specification: the meaning is that the signal is sent to the events queue of the specified destination object

The action of sending a signal or calling an operation specifies the target object,("the term "`self`" can be used to denote the sending object itself; if no destination is specified, then "`self`" is implicitly assumed), the signal being sent, and possibly the list of actual parameters.

A temporary transition variable declaration can appear inside an action sequence, with its scope extending till the end of the sequence. A transition variable declaration is similar to a local attribute declaration, but does not allow an explicit initialization.

A composite action can be a conditional statement or a loop statement.

A *conditional* statement has the form:

```
“if condition then {then-part}”
“if condition then {then-part} else { else-part}”
```

where `condition` if a boolean expression, and `then-part` and `else-part` are sequences of actions.

A *loop* has the form:

```
"for var in min ... max {loop-body}"
```

where `var` is a free (unused) identifier for the loop index, `min` and `max` are integer expressions, and `loop-body` is a sequence of statements. Inside the loop body the loop index cannot be target of assignments. Inside the loop-body can appear exit statements.

Other syntactic issues

In the current prototype type checking and many static analysis checks are not fully performed; i.e. the input specification is supposed to be mostly correct. This is due partly to the fact that the UMC modelling language is not seen as the primary design tool a user should use, but rather an intermediate language into which XMI models (or other models) are translated. However, in the absence of this higher lever translation phase some effort has been made in improving the direct usability of the UMC textual modelling language, and this includes a limited form of static checking.

In the definition of the UMC textual modelling language we have preferred not to associate it with a rigid syntactic structure, but to allow a great level of flexibility in order to simplify the possible translations and let the users to preserve as far as possible the concrete syntaxt to which they are used. This is particularly true for example for the action language, where we have tried not to adopt a fixed schema derived from the C language, the JAVA language or any other specific language. Several syntactic alternatives are therefore allowed:

Line comments can be introduced by prefixing them with “`--`“ or “//”:
The token "," as separator between Signals, Operations, Vars items and Defer events can be substituted by ";"
The token "and" in boolean expressions can be substituted by "&" or "&&"
The token "or" in boolean expressions can be substituted by "|" or "||"
The token "not" in boolean expressions can be substituted by "!"
The token "/=" in boolean expressions can be substituted by "!="
The token "=" in boolean expressions can be substituted by "=="
The token ":=" in assignments can be substituted by "="
The token "=>" in Object instantiations can be substituted by "=" or "->"

In the case of empty completion transitions the sequence `"-(-)->"` can be substituted by "->"

Another completely alternative syntactic structure (in Hugo style) for transitions is also:

```
      source  -> target  { trigger guard / actions }
```

In the definition of a Statepath (inside source and target of a transition) all the outer prefixes which are not necessary to univoquely identify a substate can be omitted.

e.g. "`Top.S1.R2.S3.s4`" can be substituted by "s4" if such "s4" is unique
e.g. "Top." can be always omitted

The keywords: `Vars Operations Signals Transitions` can be followed by ":"

The token "`end`" can be followed by the name of the class.

Unsupported features

With respect to UML 1.4 there are several statechart features which are not supported by the current release of UMC. The most relevant of these unsupported featues are History/Deep-History/Synch states, state Entry/Exit/Do activities, Dynamic Choice / Static Choice transitions, Change and Time events.
None of the above limitations is intrinsic to the tool, and further versions of the prototype are likely to overcome them

The UMC Grammar

{item} denotes 0 or more occurrences of the item
[item] denotes 0 or 1 occurrence of the item
"item" denotes a terminal character sequence
"item | item " denotes indicates alternative items

```
Model ::= { Class }  { Object }

Class ::= "class" ClassName "is"
             [ "Signals"   Signal, {"," Signal} ]
             [ "Operations"  Operation, {"," Operation} ]
             [ "Vars"  Attribute {"," Attribute}]
             [ "State" "top" "=" Composite
              {"State"  Statepath  "="  State } ]
             [ "Transitions"  {Transition} ]
           "end;" [ClassName]

Signal ::=  SignalName ["(" ParamName [":" TypeName]
                      {"," ParamName [":" TypeName] }
                  ")" ]

Operation ::=  OpName ["(" Name [":" TypeName]
                     {"," Name [":" TypeName] }
                      ")" ] [ ":" TypeName]

Attribute ::=   AttrName [":" TypeName] [":=" StaticExpr ] ";"

State  ::= Composite | Parallel

Composite ::=   StateName { "," StateName}
                  ["Defers" Defer {"," Defer} ]
```

```
Parallel ::=    Name { "/" Name}
                  ["Defers" Defer {"," Defer}]

StateName ::=  Name | “final” | “initial”

Defer ::= EventName ["(" ParamName {"," ParamName } ")"]

Transition ::=
    Statepaths "-(" Trigger[Guard] [“/” Actions] ")->" Statepaths

Statepaths ::=  Statepath  |   "(" Statepath {"," Statepath} ")"

Statepath ::=  ["top."] Name { “.” Name}

Trigger  ::=  "-" |  EventName ["(" ParamName {"," ParamName } ")"]

Guard ::=  "[" BoolBoolExpr "]"

Actions::=  [ Stm {";" Stm} ]

Object ::=  "Object"  ObjName ":" ClassName [ Initializations ]

Initializations ::= "(" AttrName "=>" StaticExpr
                     { "," AttrName "=>" StaticExpr } “)”

--    action statements

Stm ::=   Assignment |
          SignalSending |
          OperationCall |
          FunctionCall  |
          ConditionalStm |
          LoopStm       |
          VarDecl       |
          ReturnStm     |
          ExitStm

Assignment  ::=   TargetExpr ":=" Expr

SignalSending ::=  ObjExpr "." SignalName ["(" Expr {"," Expr} ")"]

OperationCall ::=  ObjExpr "." OpName ["(" Expr {"," Expr} ")"]

FunctionCall ::=
    TargetVar ":=" ObjExpr "." OpName ["(" Expr {"," Expr} ")"]

ConditionalStm ::=
     “if” BoolBoolExpr [ “then” ] “{“ Actions “}” [ “else {“ Actions “}” ]

LoopStm   ::=
    “for” LoopIndex “in” IntExpr “..” IntExpr “{“ Actions “}”

VarDecl  ::=    VarName ":" TypeName

ReturnStm   ::=   “return” ["(" Expr {"," Expr} ")"]

ExitStm   ::=    “exit”

TargetExpr   ::= AttrName [ Selection ] | VarName [Selection]

Selection  ::=    “[“ IntExpr “]”

--  names and expressions

Expr ::= “(“Expr“)” | BoolBoolExpr | IntExpr | ObjExpr | VectorExpr

BoolBoolExpr ::=  BoolExpr {"and"  BoolExpr}
                  | BoolExpr {"or"  BoolExpr}
                  | "not"  BoolExpr
                  | BoolExpr

BoolExpr ::=  "true" | "false"
              | AttrName [Selection]| VarName [Selection]
              | Expr “=” Expr    | Expr “/=” Expr
              | IntExpr relop IntExpr
```

```
ObjExpr ::=  "null" | AttrName [Selection]| VarName [Selection]
             | ObjName | "self" | "this"
IntExpr ::=   Number | AttrName [Selection] | VarName [Selection]
             | (Intexpr intop IntExpr ")" | VectorExpr ".head"
VectorExpr ::= "[]" | AttrName | VarName |
              VectorExpr "+" VectorExpr | VectorExpr ".tail"
StaticExpr  ::=  Number | ObjName | "null" | "self" | "this"
relop ::=     ">" |  ">=" |  "<"  | "<="
intop ::=  "+" | "-" | "*" | "/" | "mod"
TypeName ::=  "int"  | "bool"  | "obj" | ClassName
              "int[]"  | "bool[]"  | "obj[]" | ClassName"[]"
```

An Overview of the Dynamic semantics of UMC models

In the UML-1.4 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 a given state machine configuration can be discovered by performing the following substeps :

a)Selecting current trigger: The current event to be dispatched is selected from the object event queue (which might also a system generated "completion event").

b) 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 currently dispached event (if any) of the state machine 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). If the set is empty the run-to-completion step termininates here, with the removal of the dispatched event from the events queue (unless deferred).

c) Dealing with priorities: If the previous set is not empty, 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.

d) 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.

e) 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.

f) Computing the target configuration: The next state-machine configuration resulting after this evolution if obtained by:
- removing the dispatched 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.

The above steps defines the possible effects of starting a "run-to-completion" step. Once such a step is started it can atomically complete with the execution of all the involved actions or become suspended over some synchronous call operation. In this second case the local variables of the object are updated, and the generated events delivered to the system, just before the evolving object becomes suspended on the call operation. The run-to-completion step will be eventually resumed when a return signal is received from the called object.

Notice also that implicit "completion events" are generated when the activity of a state is terminated, and that these completion events have precedence over the other events possibly already enqueued in the object events queue.

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 is
Signals: a
State Top =  s1
Transitions:
  s1 -( a / self.a; self.a )-> s1
end
```

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.
With respect to this, UMC makes certain assumptions which, even if compatible with the UML standard, are not necessarily the only possible choice.

1) Atomicity of transition effects w.r.t. internal object concurrency
The whole sequence of actions constituting the actions part of statechart transition is supposed to be executed (with respect to the other concurrent transitions of the same object) 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). Notice that this does not mean that

statechart transition are atomic with respect to the system behavior, since its activity can contain synchronous call causing suspensions/resumptions of the activity.

2) Atomicity of object evolutions w.r.t. system evolutions
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 at system level when no synchronous calls are involved.

3) Reliable, point to point, and untimed communications
The propagation of signals inside a state machine and among state machines is considered instantaneous, and without duplication or losses of messages (this is an aspect intentionally left as unspecified by the UML standard); the communication is direct and one-to-one (no broadcasts).

4) FIFO event queues
The events queue associated with a state machine handles its events in a FIFO way (this is an aspect intentionally left as unspecified by the UML standard).

5) Well definedness of priorities
The relative priority of a join transition is always well defined (identitied with the priority of the first of its source states) and statically fixed.[1]

6) Completion of operation calls
The return signal from an operation/function call occurs when the "return" statement is executed inside the transition triggered by the operation-call. If more than one return statements are executed by the run-to-completion step triggered by a call the last return event overrides the previous events (this might eventually be an error syntactically and statically checked) . If no return statement is executed by the run-to-completion step triggered by a call then no return event is generated and the caller ramains suspended (until a return signal is eventually received).

7) Parallel completions events
In our model we suppose that all completion events generated by a run-to-completion step are dispatched all toghether in a unique step[2].

3 The structure and semantics of UCTL logics

The logics supported by UMC is essentially the full modal/propositional mu-calculus, extended with higher level CTL/ACTL-like operators, and structured action expressions.
Notice that if a system has an infinite number of states the termination of the evaluation of a formula is not guaranteed to succeed. In fact if such evaluation <u>requires</u> the analysis of an infinite number of states the evaluation itself never terminates. For example the formula: `"AG predicate"` will certainly fail to return a value if the system is infinite, while the other

[1] 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".
[2] From this poi of view the UML semantics is not very clear.

formula : "`EF predicate`" will return a defined value if and only if a configuration satisfying the predicate exists at some finite depth of the evolutions tree. In general, a definite result will be produced by the tool if and only if a finite example or counterexample for the formula does exist.

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.
A StatePredicate can also be relation between simple expressions involving attributes and valuers; such formulas hold for all the states in which the evaluation of the relation is true.

```
Examples: "ASSERT(obj.objattr.intattr > 0)"
          "ASSERT(obj3.x+1 > obj1.y+obj2.z)"
```

The allowed relational operators are "`<`", "`>`", "`=`", "`/=`", "`<=`", "`>=`".
The "simple expression" can be an object attribute, a numeric value, or a sum of them.
The obj prefix of the attribute can be omitted is the system is constituted by a single active object. E.g. "`(x <= y)`" is a valid state predicate for a single object system.
The special keyword "`queuesize`" acts like an object attribute and denotes the current length of the events queue of the object.
The keyword "`ASSERT`" can be omitted.

Boolean logical operators

Logic formulas can be composed with the usual boolean operators which have the classical meaning. The "`not`" operator has precedence over the others "`and`" "`or`" "`implies`" 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).
"`|`" can be used as a shortcurt for "`or`",
"`&`" can be used as a shortcurt for "`and`",
"`~`" can be used as a shortcurt for "`not`",
"`->`" can be used as a shortcurt for "`implies`".

Basic mu-calculus

The basic mu-calculus temporal operators are the box and diamond operators and the max and min fix point operators.

Diamond <>

The diamond operator `<action-expr> SUBFORMULA` allows to specify that a certain subformula should be true in at least one nextstate reachable in one step from the current state. If an `action-expr` is specified, the nextstate should be reachable with a (one-step) system evolution which satisfies the `action-expr`.

Box []

The box operator `[action-expr] SUBFORMULA` allows to specify that a certain subformula should be true in all the nextstates reachable in one step from the current state, if any. If the current state has no successors, the diamond is still satisfied. If an `action-expr` is specified, the subformula should be true in all the nextstates reachable with a (one-step) system evolution which satisfies the `action-expr ,` if any such nextstates exist.
`[action-expr] SUBFORMULA` is equivalent to `not <action-expr> not SUBFORMULA`

Max and Min FixPoints

The power of mu-calculus is achieved through the two fixed point operators max and min. The fixedpoint identifier can 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(true)  and
FixedPointBody(FixedPointBody(true))  and
FixedPointBody(FixedPointBody(FixedPointBody(true))) and
...  and
```

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

```
FixedPointBody(false) or
FixedPointBody(FixedPointBody(false))  or
FixedPointBody(FixedPointBody(FixedPointBody(false)))  or
 ... or
```

For example, the following formula: `max Z: <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 evolutions satisfying "a" which starts from s1).

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

```
max Z: FixpointBody(Z)                 is  equivalent to:
not min Z:  not FixpointBody(not Z)

min Z: FixpointBody(Z)                  is  equivalent to:
not max Z: not FixpointBody(not 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 "`min Z: not Z`", holds for any state because its semantics is given by:
" `false or not false or not not false or  ...`"
The above is equivalent to " `false or true or ...` " whose value is "`true`" .
It is likely that in future version of the tool the monotonicity of the formula will be required and checked.

The mu-calculus is an very powerful logic, and can be used to encode all the other CTL/ACTL like temporal constructs. However, because of its low level of abstraction, writing and understanding formulas written in pure-calculus can be a particularly hard task.
For this reason it is useful to explicitly define the other CTL/ACTL like modalities which allow a more natural expression of the system properties and a reasonably efficient way to

evaluate them (the full mu-calculus is known to have a complexity which is exponential w.r.t. the alternation depth of the formula).

While the CTL /ACTL modalites will be described later here we only show a few (from simple to complex) properties which unfortunately cannot be espressed in ACTL-like modalities, and hence need the explicit use of fixpoint operators.

"*There exist an infinite path whose evolution steps all satisfy the action expression "b"*
The above can be encoded as: `max Z: (<b> Z )`

There is a reachable livelock (i.e. a loop of `tau` *evolutions) somewhere.*
The above can be encoded as: `EF max Y: <tau> Y`

For all paths in which event "a" never occurs, eventually the predicate P will hold.
The above can be encoded as: `min Z: P or (not final and [not a] Z)`

There exist a path such that an evolution which satisfies the action expression "a" occurs infinitely often.
The above can be encoded as: `max Z: min W: ((<a> Z ) or (<not a> W) )`

There exist a path such that the state predicate "p" is satisfied infinitely often by its states sequence.
The above can be encoded as: `max Z: min W: ((p and <> Z ) or ((not p) and <> W))`

There exist a path from certain point of which the state predicate "p" holds infinitely often and state predicate "q" does no longer hold.
The above can be encoded as:

```
   EF  max Z:
         min W: ( (p and (not q) and <> Z)  or
                  ((not p) and (not q) and <> W) )
```

For all paths, if an evolution which satisfies the action expression "a" is <u>possible</u> an infinite number of times, then an evolution which satisfies "a" is <u>done</u> an infinite number of times.
(i.e. does not exist a path such that from e certain point a *is not done anymore but nevertheles "a" is infinitely possible.*
The above can be encoded as:

```
    not EF max Z:
           min W: ( ( (<a> true) and <not a> Z)  or
                    ( (not <a> true) and <not a> W ) )
```

Action Expressions

Basic Action Expressions

The basic form of action expression is eother the "`true`" or "`false`" predicate, or a basic action predicate about the actions performed by the system evloution.
In particular:
The action expression "`true`" is satisfied by all system evolutions.
The action expression "`false`" is not satisfied by any system evolution.
The action expression "`obj:`" is satisfied by any system evolution in which object "`obj`" is the one which evolves .

The actions expression "`targetobj.event_id`" is satisfied only by system evolution which generate the event event_id and send it to the object "`targetobj`".
The action expression "`event_id`" is satisfied only by those system evolutions which do generate an event "`event_id`", possibly with some parameters (notice that event_id might be just one of the events generated by the evolution).
The actions expression "`event_id(args)`" is satisfied only by system evolution which generate the event event_id `event_id` with the specified `args`, where `args` is a comma separated sequence of static values or "*" . E.g. the action expression "`callop(*,3)`" is satisfied by all evolution in which the signal or operation "`callop`" (which has two parameters) is being executed, and in with the second argument of the event has value "`3`".

Example1:
The actions expression "`sourceobj:targetobj.event_id(args)`" is satisfied only by system evolutions which object `sourceobj` generate a signal `event_id` with the specified `args`, being sent to object denoted by "`targetobj`".

Example2:
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;target!event_id(1,2,3);other_events`".

Boolean Compositions of Action Expressions
...
An `action expression` can be a boolean composition of other action expressions.
The action expression "`pred1 or pred2`" is satisfied by the system evolutions which satisfy either `pred1` or `pred2`.
The action expression "`pred1 and pred2`" is satisfied by the system evolutions which satisfy both `pred1` or `pred2`.
The action expression "`not pred1`" is satisfied by the system evolutions which do not satisfy the action predicate `pred1`.

Observable Events and Observation Modes
The concept of "observable event" is touched when we either make use of the "`tau`" action expression, or when the weak ACTL-like logical operators are used (box, diamod, until).
The action expression "`tau`" is satisfied only by the system evolutions which do not generate any observable event; these include both system evolutions which do not generate any event at all, and system evolutions which generate only events explicitly tagged as "non-observable" by the user . In fact, the user has the choice to select a particular "*observation mode*" in which the UMC is run, which affects the degree of observability of the generated model events. In particular:
When the "`white_box`" observation mode is selected, all events are considered observable (these are all change event caused by assignments, all signals, all operations, all returns.
When the "`gray_box`" observation mode is selected, are considered observable only signals and operation calls, but not the change events caused by assignments.
When the "`black_box`" observation mode is selected, are condidered observable only the signals (this includes all OUT and ERR signals) sent by the system to the outside (i.e. all signals not targeted to another active object of the model).
When the "`custom`" observation mode is selected, the user is required to explicitly list the events and the object attributes of interest, thus making observable only certain assignments, signals and operation calls.

When the "`selective`" observation mode is selected, the user is required to explicitly list the objects of interest, thus making observable only the signals and operation calls received and generated by them, and the assignments made by them.
When the "`interactions`" observation mode is selected, the user is required to explicitly list the objects of interest, thus making observable only the signals and operation calls exchanged between any two objects of the list.

Evolution predicates

Evolution predicates are a form of action expression which describes a relation beween the attributes (variables) of the source and target states (names ending with "`′`").
They have the form:
```
   <target_var>"′"   <relop>  <source_var_expr>
```
where `<relop>` is one of `"<", ">", "=", "/=", "<=", ">=".`

Examples are:
```
 "(x′ >= x)",  "(obj1.x′ /= obj1.obj2.x+1)",  "(obj1.obj2.x′ = obj1.y)"
```

ACTL-LIKE temporal operators

Next EX AX ET AT

The Exists Next operator "`EX {action-expr} 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.
It is exactly equivalent to the basic diamond operator "`<action-expr> SUBFORMULA`"

The Always Next operator "`AX {action-expr} 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).

"`AX {action-expr} subformula`" is equivalent in basic mu-calculus to:
```
    (<action-expr> true) and
      (not <action-expr> not subformula) and
        (not <not action-expr> true)
```

"`ET subformula`" is equivalent to "`EX {tau} subformula`"
"`AT subformula`" is equivalent to "`AX {tau} subformula`"
"`EX subformula`" is equivalent to "`EX {true} subformula`"
"`AX subformula`" is equivalent to "`AX {true} subformula`"

Beware: in the original ACTL definition the action expression `{true}` in the contex of the Next operators had the meaning of "any observable event", while here it has the meaning of "any event (either observable or not)".

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.

"`EF Formula`" is equivalent to its dual "`not AG not Formula`" and is equivalent to the fix point formula: "`min Z: (Formula or <>Z)`" .

"`AF Formula`" is equivalent to its dual "`not EG not Formula`" and is equivalent to the fix point formula: "`min Z: (Formula or AX Z)`"

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).

"`EG Formula`" is equivalent to its dual "`not AF not Formula`" and is equivalent to the fix point formula: "`min Z: (Formula and <> Z)`".

"`AG Formula`" is equivalent to its dual "`~EF ~Formula`" and is equivalent to the fix point formula: "`min Z: (Formula and AX Z)`".

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).
"`E[ Formula1 U Formula2 ]`" is equivalent to the fix point formula:
```
     "min Z: ( Formula2 or (Formula1 and <> 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).
"`A[ Formula1 U Formula2 ]`" is equivalent to the fix point formula:
```
     "min Z: ( Formula2 or (Formula1 and AX Z ))"
```

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 unobservable actions (satsfying `tau`) or actions satisfying `act`, and `Formula1` holds in all the intermediate states along the path from s1 to s2 (s1 included, s2 excluded).

"`E[ Formula1 {act} U Formula2]`" is equivalent to the fix point formula:
```
    "min Z: ( Formula2 or (Formula1 and <act or tau> Z ))"
```

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 unobservable actions (satsfying `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).

"`A[ Formula1 U Formula2 ]`" is equivalent to the fix point formula:

```
   "min Z: ( Formula2  or (Formula1 and AX {act or tau} Z ))"
```

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 unobservable evolutions (satisfying `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.

"`E[ Formula1 {act} U {act2} Formula2 ]`" is equivalent to the fix point formula:

```
   "min Z: Formula1 and ((<act2> Formula2) or <act1 or tau> Z})"
```

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 unobservable evolutions (satisfying `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.

" `A[ Formula1 {act1} U {act2} Formula2 ]`" is equivalent to the fix point formula:

```
 min Z: ( Formula1 and
          (not FINAL) and
          ([act2 and (not tau) and (not act1)] Formula2) and
          ([(act1 or tau) and (not act2)] Z )  and
          ([(act1 or tau) and act2] (Formula2 or Z) ) and
          ([not act1) and (not act2) and (not tau)] false)
```

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".

Weak Diamond <<act>>

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 (satisfying `tau`) or actions satisfying `act`.
`act` is not allowed to contain “`tau`” as subexpression.

Notice that in the original ACTL the syntax of this temporal operator was "`<act> Formula`" (now used for the basic mu-calculus diamond operator). From this point of view this operations consitutes a “backword incompatibility” between UMC and the original ACTL.

Weak Box [[act]]

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 unobservable evolutions (satisfying `tau`).
`act` is not allowed to contain "`tau`" as subexpression.

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

Notice that in the original ACTL the syntax of this temporal operator was "`[act] Formula`" (now used for the basic mu-calculus box operator). From this point of view this operations consitutes a "backword incompatibility" between UMC and the original ACTL.

The UCTL Grammar

{item} denotes 0 or more occurrences of the item
[item] denotes 0 or 1 occurrence of the item
"item" denotes a terminal character sequence
"item | item " denotes indicates alternative items

```
-- umc formulas

FORM ::=
    state-predicate
   | and-sequence
   | or-sequence
   | negation
   | implication
   | basic-mu-formula
   | actl-like-formula


-- state predicates

state-predicate ::=
   "true"
   | "false"
   | simple-expr
        [ "<", ">", "=", "/=", "<=", ">=" ] simple-expr

simple-expr ::=
    basic-expr
    | basic-expr "+" basic-expr

basic-expr ::=
    [obj"."]attr{"."attr}
    | value


-- boolean logical operators

and-sequence ::=  FORM "and" FORM { "and" FORM }
or-sequence ::=   FORM "or" FORM { "or" FORM }
negation ::=  "not" FORM
implication ::=  FORM "->" FORM
```

```
enclosing ::=  "(" FORM")"


--  basic mu-calculus

basic-mu-formula ::=
   maxfix  | minfix | mu-box | mu-diamond | fixid

maxfix  ::=  "max" VARID ":" FORM
minfix  ::=  "min" VARID ":" FORM
mu-box  ::=  "<" [action-expr] ">" FORM
mu-diamond  ::=  "[" [action-expr] " ]" FORM
fixid  ::=  VARID


--   action expressions

action-expr ::=
   "true"  | "false" | "tau",
  | action-expr  "and"  action-expr {"and" action-expr}
  | action-expr  "or"  action-expr  {"or" action-expr}
  | "not" action-expr
  | [source":"] [target"."] ( "*" | event [args] )
  | "("action-expr")"
  | "("obj"."]attr{"."attr}"'" relop  simple-expr ")"

relop ::=  "<" | ">" | "=" | "/=" | "<=" | ">="

--  CTL/ACTL like formulas

actl-like-formula ::= exists-next1 | forall-next1 |
                      exists-next2 | forall-next2 |
                      exists-future | forall-future |
                      exists-globally | forall-globally |
                      exists-until1   | forall-until1 |
                      exists-until2   | forall-until2 |
                      exists-until3   | forall-until3 |
                      weak-diamond   |  weak-box

exists-next1 ::= "EX" FORM

forall-next1 ::= "AX" FORM

exists-next2 ::= "EX" "{" action-expr "}" FORM

forall-next2 ::= "AX" "{" action-expr "}" FORM

exists-future ::= "EF" FORM

forall-future ::= "AF" FORM

exists-globally ::= "EG" FORM

forall-globally ::= "AG" FORM

exists-until1 ::= "E[" FORM1 "U" FORM2 "]"

forall-until1 ::= "A[" FORM1 "U" FORM2 "]"

exists-until2 ::= "E[" FORM1 "{" action-expr "}" "U" FORM2 "]"

forall-until2 ::= "A[" FORM1 "{" action-expr "}" "U" FORM2 "]"
```

```
exists-until3 ::=
       “E[“ FORM1 “{“ action-expr “}” “U” “{“ action-expr ”}” FORM2 “]”

forall-until3 ::=
       “A[“ FORM1 “{“ action-expr “}” “U” “{“ action-expr2 “}” FORM2 “]”

weak-diamond ::= “<<” action-expr “>>””

weak-box ::= “[[“ action-expr “]]” FORM
```

More Shortcuts

```
“FINAL” is a shortcut  for “not <true> true”
```

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 other 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 must fit a single line), it is immediately evaluated and further input is awaited (remember that the evalution might diverge in the case of infinite state systems).

```
|- 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. Alternatively the “tabwhy” command saves in the current directory a file explanation.tab containing an example/counterexample for the validity of the formula.

```
|- why
-------------------------------------------
, the formula:  EF ET true
   is FOUND_FALSE in State C1
because
 The formula:  ET true
   is FOUND_FALSE in State C1
|-
|- tabwhy
Explanation Model generated.
|-
```

Browsing into the current configuration detail

The "info" command allows 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=0
max_explanation_depth=100
max_evolutions_depth=10
showparams=TRUE
showtarget=TRUE
showstuttering=TRUE
observereturns=FALSE
observationmode=GRAY_BOX
showevolvingobject=TRUE
explicit_tau_requested=FALSE
|-
```

The `initial_lts_depth` parameter affects the way in which the on-the-fly evaluation of a formula proceeds. In particular it sets the initial depth limit at which a depth-first subcomputation should be truncated, in favour of other less deep subcomputations.
If no answer about the validity of a formula can be given with the curent lts_depth limit, the evaluation is restarted with a doubled lts_dept limit.

The `observationmode` parameters allow to define the user policy in terms of observability of events:
`set observationmode=BLACK_BOX` -- only outgoing signals are observed
`set observationmode=GRAY_BOX` -- all signal and operation events observed
`set observationmode=WHITE_BOX` -- all signal, operations, and assignments observed
`set observationmode=SELECTIVE:obj1,obj2,obj3`
 -- all events involving any of given objects, including local assignments, are observed
`set observationmode=INTERACTIVE:obj1,obj2,ojb3`
 -- all signals and operations exchanged between any two given objects are observed
`set observationmode=CUSTOM:obj1.x,op,obj3.y,sig`
 -- the evolutions of the given object attributes, and the exchanges of the given signals and operations are observed.

The remaining parameters affects the visualization of graphic output (in dot .format) representing the system evolutions tree. In particular:
The `max_explanation_depth` parameter constrains the maximal depth of the explanations steps visualized in consequence of a "`why`" or “`mapwhy`” command.

The `max_evolutions_depth` parameter constrains the maximal depth of the evolution steps visualized in consequence of a "`trace`" command without arguments, .or by the "`map`" command.
The `showstuttering` parameter defines whether or not to make visible stuttering transitions by adding fake "`ERR.lostevent`" signal in stuttering transitions (absence of stuttering is usually an interesting property of the system).
The `showparams` parameter selects whether or not the current value of the parameters of signals and calls are actually to be displayed in the evolutions tree generated by the "map" command.
The `showtarget` parameter selects whether or not we want to display the target object in signal and operations call when drawing the evolutions tree with the "map" command.
The `observereturns` parameter selects whether or not we want to draw the return signals generated when an operation call completes, when drawing the evolutions tree with the "map" command.
The `showevolvingobject` parameter selects whether or not we want to visualize the name of the evolving object inside the system evolutions , when drawing the evolutions tree with the "map" command.
The `explicit_tau_requested` parameter selects whether or not we want to label with "tau" the system evolutions without observable events, when drawing the evolutions tree with the "map" command.

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 (".").

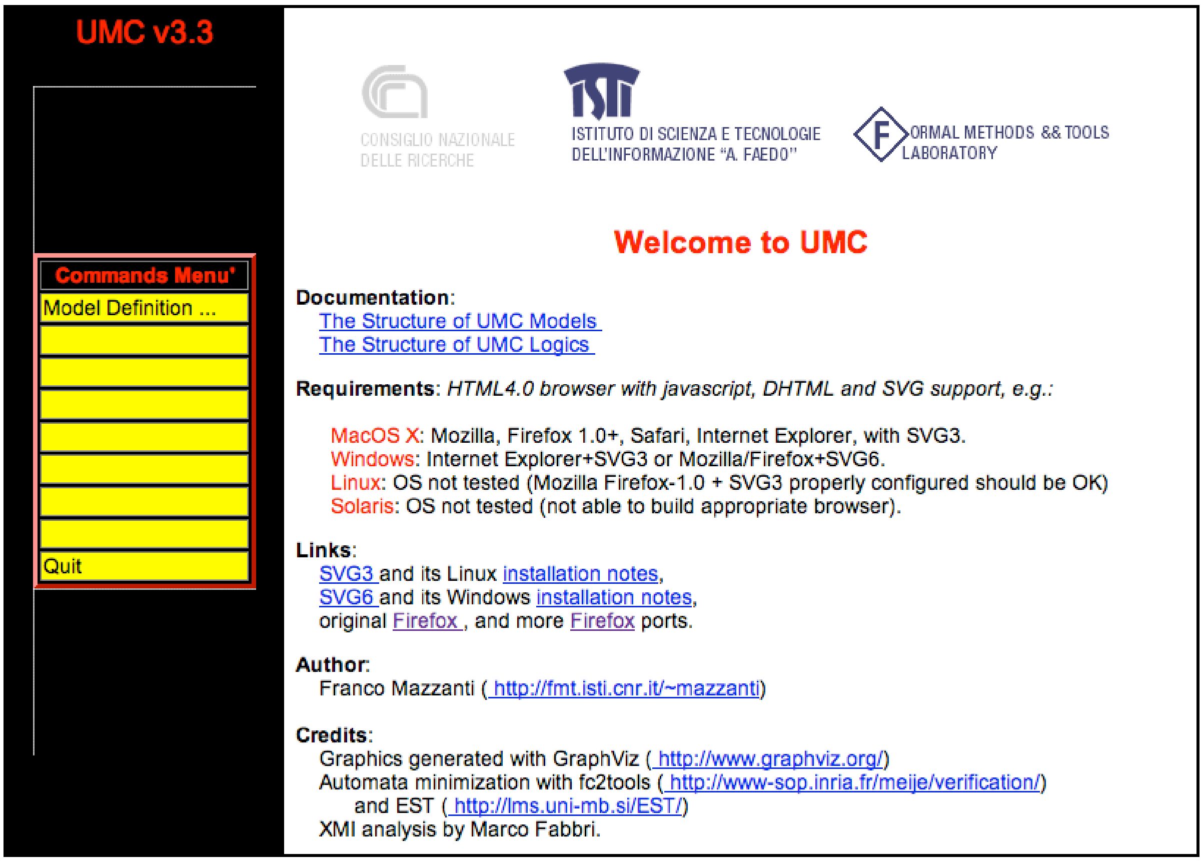
Command line options
Some of the already seen parameters of UMC can be set directly from the input comman line, and additional options are also available in this way.
The "-s" parameter disables the automatic attempt to evaluate a formula with double lts_delpth limit if the previous bound is found insufficient.
A number given on the command line establishes the current inital lts_depth limit as specified.
If a file name with the ".uctl" suffix is present in the input command line, then that file is supposed to contain the logic formula to be evaluated and the model checking is perfomed immediately and in a non-interactive way.

E.g. `umc model.umc formula.uctl -s 100000`

The model in file "model.umc" is loaded and the formula in "formula.uctl" is model-checked, starting with a maximum initial lts_depth of 100000, and without doubling the limit, but reporting the failure, if that limit is unsufficient.

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, svg and jpg images), the Ghostview package (for the translations of ps info pdf), the Fc2tools and EST 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 umc2tab and tofc2, totdot tools for the translation of a finite UMC model into specifically formatted finite LTS usable by other tools.


UMC v3.3
Commands Menu'
Model Definition ...
Quit
CONSIGLIO NAZIONALE DELLE RICERCHE
ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"
FORMAL METHODS && TOOLS LABORATORY

Welcome to UMC

Documentation:
The Structure of UMC Models
The Structure of UMC Logics

Requirements: *HTML4.0 browser with javascript, DHTML and SVG support, e.g.:*

MacOS X: Mozilla, Firefox 1.0+, Safari, Internet Explorer, with SVG3.
Windows: Internet Explorer+SVG3 or Mozilla/Firefox+SVG6.
Linux: OS not tested (Mozilla Firefox-1.0 + SVG3 properly configured should be OK)
Solaris: OS not tested (not able to build appropriate browser).

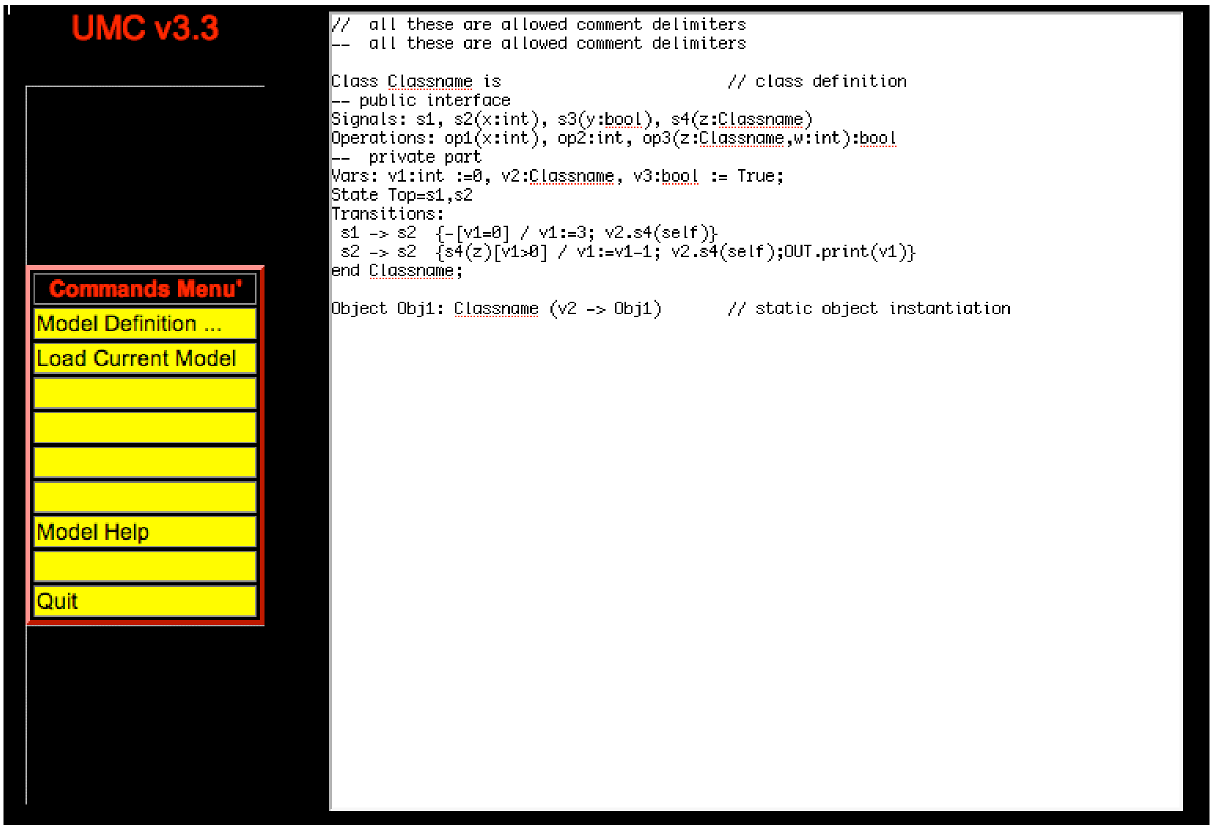
Links:
SVG3 and its Linux installation notes,
SVG6 and its Windows installation notes,
original Firefox , and more Firefox ports.

Author:
Franco Mazzanti (http://fmt.isti.cnr.it/~mazzanti)

Credits:
Graphics generated with GraphViz (http://www.graphviz.org/)
Automata minimization with fc2tools (http://www-sop.inria.fr/meije/verification/)
and EST (http://lms.uni-mb.si/EST/)
XMI analysis by Marco Fabbri.

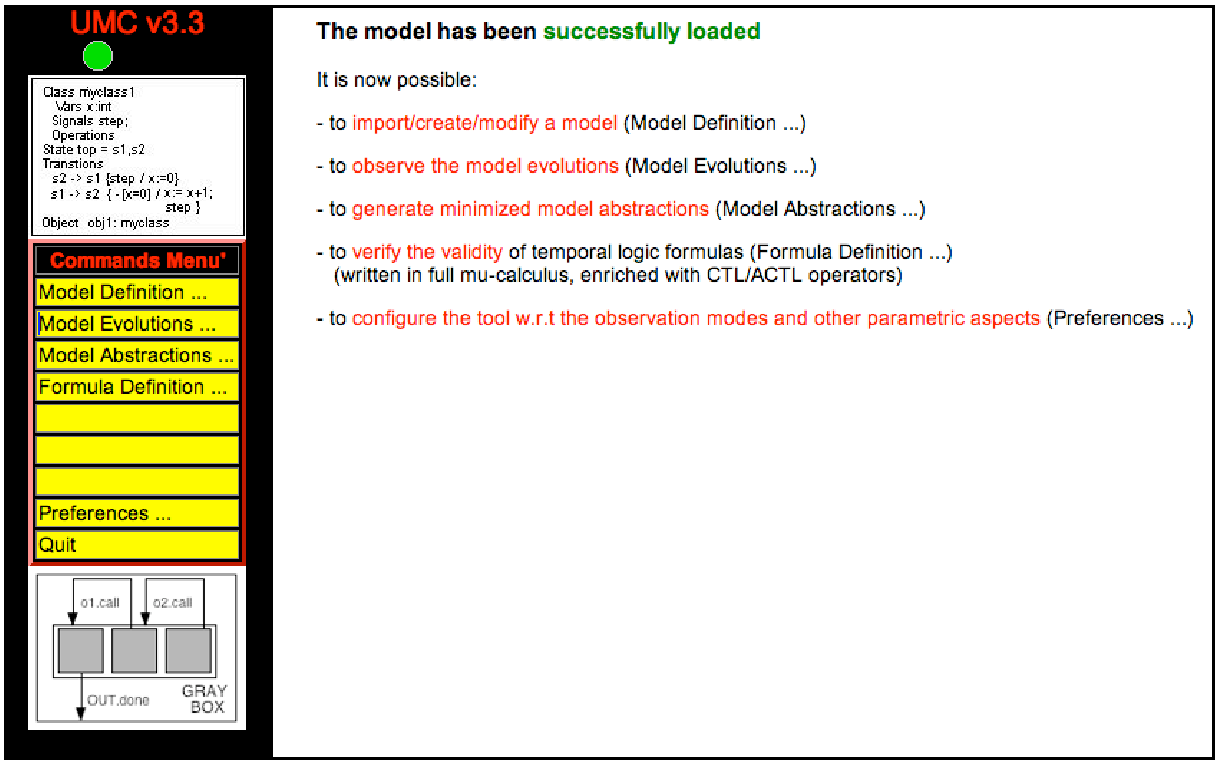
The UMC www page is constituted by two main sections:a left column containing a menu of commands, a set of status lights, and two clickable images for configuring the tool preferences and displaying the current model structure. and a right column used to display the I/O interactions (text editing, graph drawing) beteen the tool and the user.

The first step in performing a system anaysis is the definition of the model. A model can be inserted in its textual form, or reloaded from the server (in the case the tool session is for some reason closed and later restarted), or imported from a pre-existing local text file, or extracted from an existing XMI model (this functionality is currently very badly supported and generated code in an obsolete format, which requires further corrections by-hand).
Alternatively one of the predefined existing examples can be selected.



```
//  all these are allowed comment delimiters
--  all these are allowed comment delimiters

Class Classname is                         // class definition
-- public interface
Signals: s1, s2(x:int), s3(y:bool), s4(z:Classname)
Operations: op1(x:int), op2:int, op3(z:Classname,w:int):bool
--  private part
Vars: v1:int :=0, v2:Classname, v3:bool := True;
State Top=s1,s2
Transitions:
 s1 -> s2  {-[v1=0] / v1:=3; v2.s4(self)}
 s2 -> s2  {s4(z)[v1>0] / v1:=v1-1; v2.s4(self);OUT.print(v1)}
end Classname;

Object Obj1: Classname (v2 -> Obj1)       // static object instantiation
```

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



```
Class myclass1
   Vars x:int
  Signals step;
  Operations
State top = s1,s2
Transtions
  s2 -> s1 {step / x:=0}
  s1 -> s2  { - [x=0] / x:= x+1;
                        step }
Object  obj1: myclass
```

The model has been successfully loaded

It is now possible:

- to import/create/modify a model (Model Definition ...)

- to observe the model evolutions (Model Evolutions ...)

- to generate minimized model abstractions (Model Abstractions ...)

- to verify the validity of temporal logic formulas (Formula Definition ...)
  (written in full mu-calculus, enriched with CTL/ACTL operators)

- to configure the tool w.r.t the observation modes and other parametric aspects (Preferences ...)

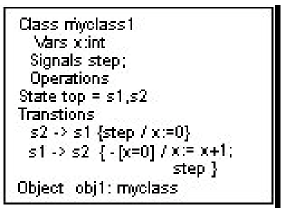
At this point the possible new activities are the following:

Observing the model statecharts
Observing the evolutions map
Abstracting the concrete model.
Editing and checking a formula
Customizing some user preferences

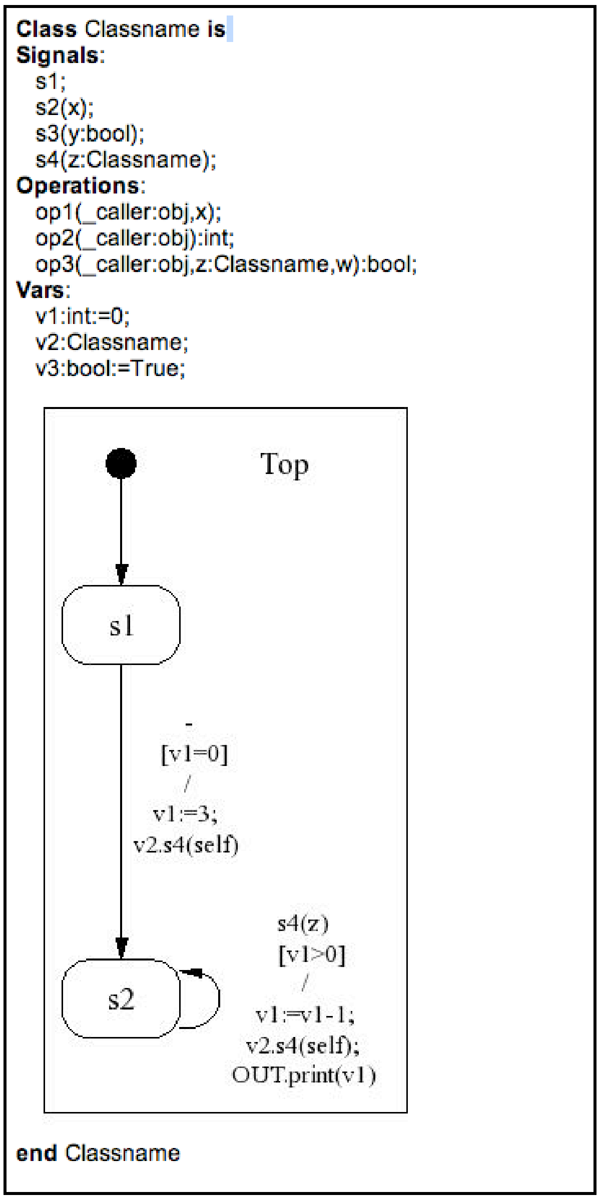
These activities are now described in more detail.

Observing the model statecharts

Clicking over the image shown here in the left frame on UMC a new pop-up window is opened which contains formatted description of the current classes consituting the model, where all the statecharts are shown in a graphical format.



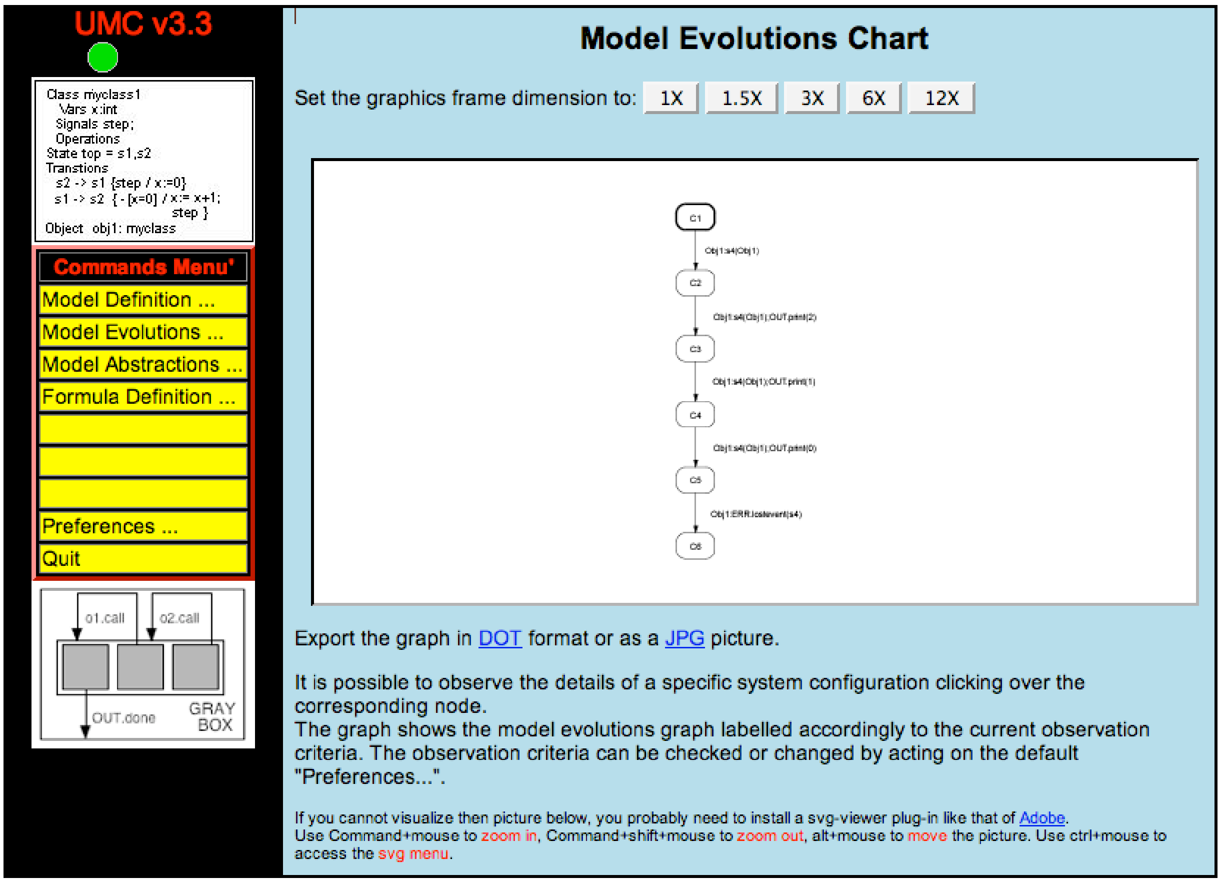
```
Class myclass1
  Vars x:int
  Signals step;
  Operations
State top = s1,s2
Transtions
  s2 -> s1 {step / x:=0}
  s1 -> s2  { - [x=0] / x:= x+1;
                              step }
Object  obj1: myclass
```



```
Class Classname is
Signals:
  s1;
  s2(x);
  s3(y:bool);
  s4(z:Classname);
Operations:
  op1(_caller:obj,x);
  op2(_caller:obj):int;
  op3(_caller:obj,z:Classname,w):bool;
Vars:
  v1:int:=0;
  v2:Classname;
  v3:bool:=True;

end Classname
```

Observing the evolutions map

Selecting the "`Model Evolutions`" button, in the right colums is shown a graphical map of the possible system evolutions starting from the current configuration. The map is shown in SVG format, hence an appropriate SVG viewer is needed (so far the free Abobe SVG viewer seems to be the best pluging usable for this purrpose). In addition, a JPG translation of the design can be requested, and the original textual code in the "dot" formal can be viewed.
The nodes in the graph represent the system configurations and the edges the possible system evolutions. In the design Edges are labelled with the (observed) signals generated by that evolution, event which are not considered observable with respect to the current observation criteria are not shown.
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.



```
Class myclass1
  Vars x:int
 Signals step;
 Operations
State top = s1,s2
Transtions
 s2 -> s1 {step / x:=0}
 s1 -> s2  { - [x=0] / x:= x+1;
                      step }
Object  obj1: myclass
```

Model Evolutions Chart

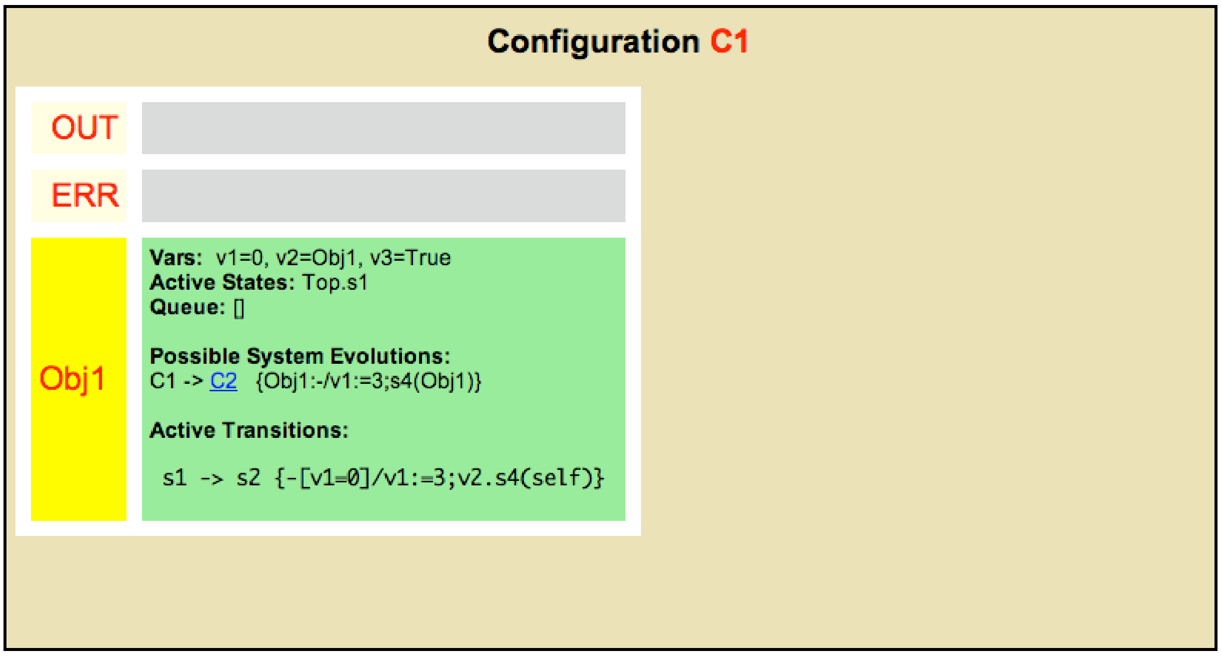
Set the graphics frame dimension to:

Export the graph in DOT format or as a JPG picture.

It is possible to observe the details of a specific system configuration clicking over the corresponding node.
The graph shows the model evolutions graph labelled accordingly to the current observation criteria. The observation criteria can be checked or changed by acting on the default "Preferences...".

If you cannot visualize then picture below, you probably need to install a svg-viewer plug-in like that of Adobe.
Use Command+mouse to zoom in, Command+shift+mouse to zoom out, alt+mouse to move the picture. Use ctrl+mouse to access the svg menu.

Clicking on one of the internal nodes, instead, a new window is opened (pop-up windows opening should be allowed) containing the detailed description of that configuration.


Configuration C1

OUT

ERR

Obj1

Vars: v1=0, v2=Obj1, v3=True
Active States: Top.s1
Queue: []

Possible System Evolutions:
C1 -> C2 {Obj1:-/v1:=3;s4(Obj1)}

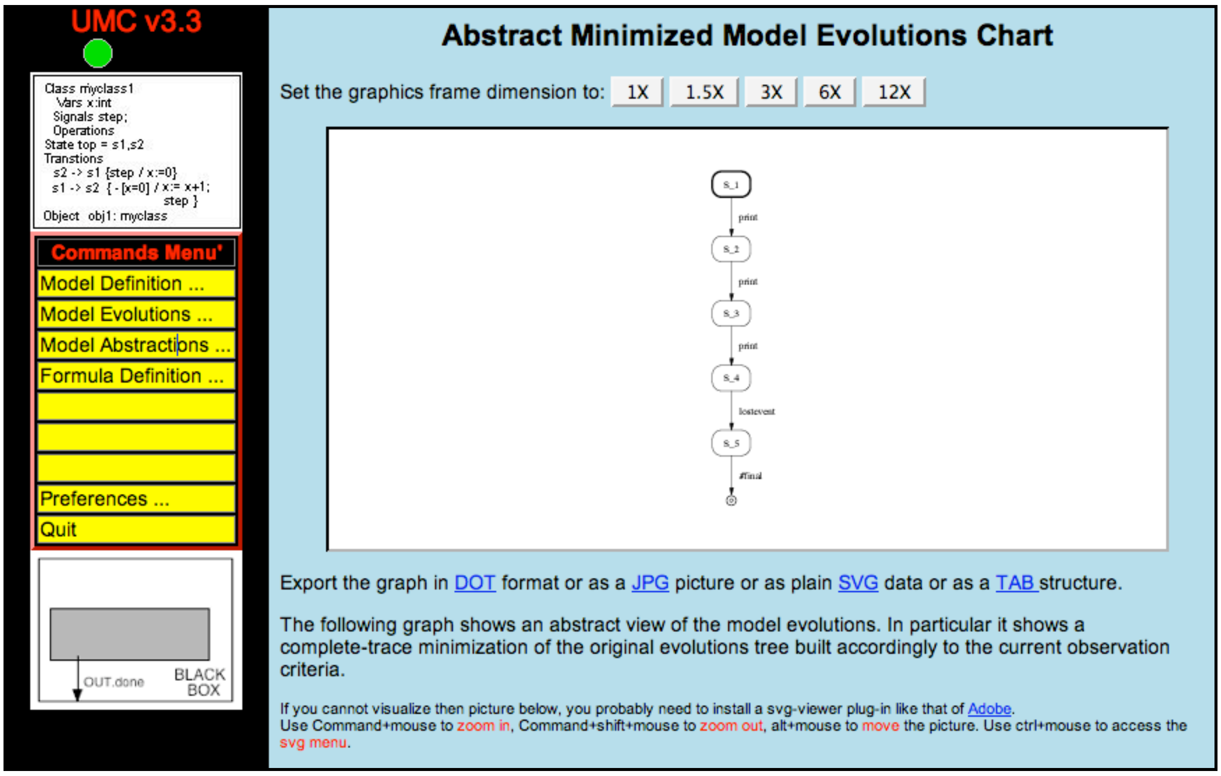
Active Transitions:

```
s1 -> s2 {-[v1=0]/v1:=3;v2.s4(self)}
```

Abstracting the concrete model.

Selecting the "Model Abstractions" button it is possible to visualize a minimized evolution graph representing the system behaviour. This functionality is useful when we are observing the system from an high level point of view, and we focus our attention on just a small aspect of the system behaviour. For example, whever we want to observe the evolution in time of just a single object attribute, or a few kind of interactions among the objects. In these cases the actual system transitions are likely to be constituted in the most part by “tau” transitions. Directly observing the concrete transition tree would not allow to grasp the correctness of the system behaviour. Observing instead a minimized version the system evolution graph may directly and intuitively reveal important correctness properties of the system.
The kind of minimization currently performed by UMC is the so-called “complete trace” minimization and is achieved by making use of the EST toolset.
Also in this case the map is shown in SVG format; in addition, a JPG translation of the design can be requested, the original textual code in the “dot” formal can be viewed and a representation of the lts in the “tab” format can be obtained.

We should note that abstract minimization of a system can only be achieved if the system is finite. In fact, the minimization algorithms are directly applied on an explicit representation of the all the system evolutions as a single automaton, and the generation of this automaton will never terminate is the system is infinite.
Moreover, the complexity of minimization is likely to be exponential in the case of full-trace minimization, and this tends to make minimizations practically unfeasable for systems of a finite but relevant size.



```
Class myclass1
  Vars x:int
  Signals step;
  Operations
State top = s1,s2
Transtions
  s2 -> s1 {step / x:=0}
  s1 -> s2  { - [x=0] / x:= x+1;
                        step }
Object  obj1: myclass
```

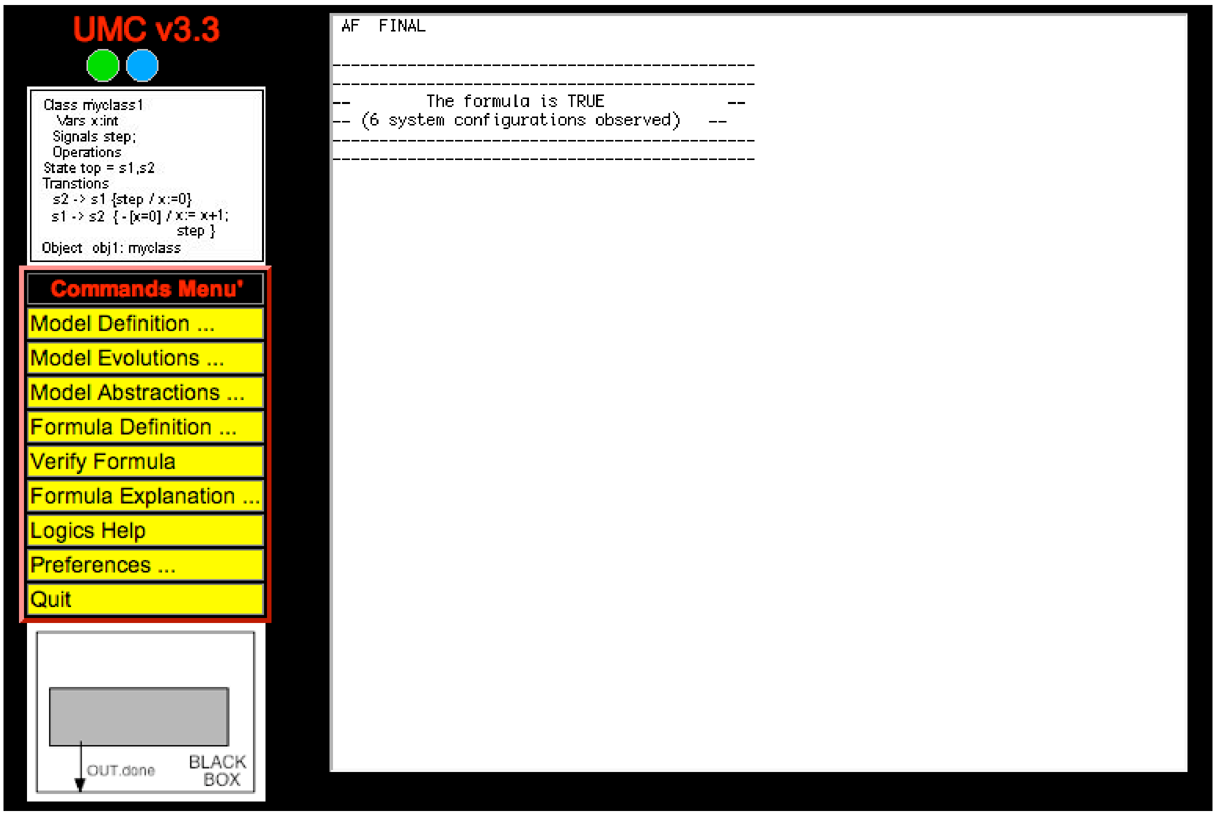
Abstract Minimized Model Evolutions Chart

Export the graph in DOT format or as a JPG picture or as plain SVG data or as a TAB structure.

The following graph shows an abstract view of the model evolutions. In particular it shows a complete-trace minimization of the original evolutions tree built accordingly to the current observation criteria.

If you cannot visualize then picture below, you probably need to install a svg-viewer plug-in like that of Adobe.
Use Command+mouse to zoom in, Command+shift+mouse to zoom out, alt+mouse to move the picture. Use ctrl+mouse to access the svg menu.

Editing and checking a formula



```
Class myclass1
  Vars x:int
  Signals step;
  Operations
State top = s1,s2
Transtions
  s2 -> s1 {step / x:=0}
  s1 -> s2  { - [x=0] / x:= x+1;
                        step }
Object  obj1: myclass
```

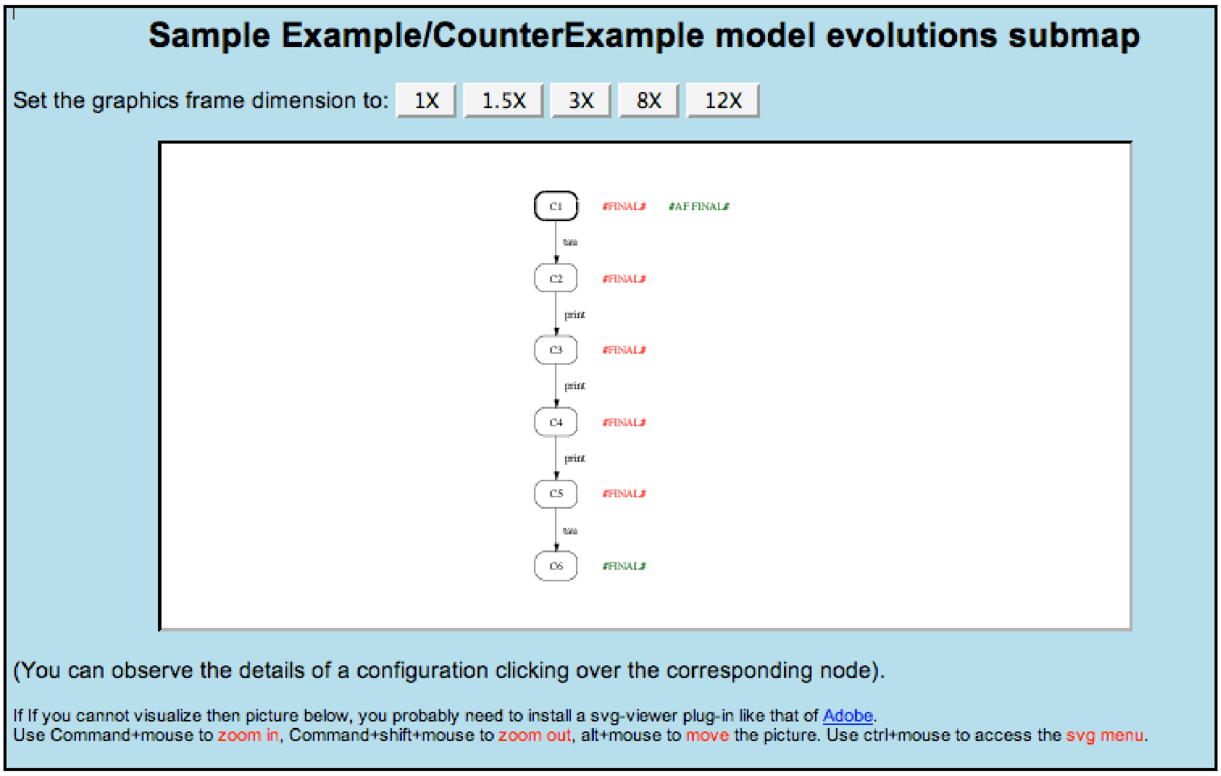
```
AF  FINAL

-----------------------------------------------
-----------------------------------------------
--          The formula is TRUE             --
-- (6 system configurations observed)    --
-----------------------------------------------
-----------------------------------------------
```

Selecting the "Formula Definition ..." button an editing area is shown on the right frame of UMC, were the formula to be checked can be edited.

Once the mu-UCTL formula has been written we start the evaluation selecting the "Verify formula" button. The evaluation is then started and its results shown in the same editing frame.

When the evaluation of a formula completes, if we want more details on how the result was achieved we can select the "Formula Explanation ..." button, which opens a pop-up window in which is shown a fragment of the model lts. which is sufficient to exemplify the validity of the formula. Nodes of the lts are labelled with the relevant formulas and their validity in that state (green=true, red=false).

Morevoer, clicking over a node of the example we can observe all the internal details of that system configuration (values of attributes, active transitions, possible configuration evolutions) which may help in understanding the validity of subformulas in that configuration state.


Sample Example/CounterExample model evolutions submap

Set the graphics frame dimension to: 1X 1.5X 3X 8X 12X

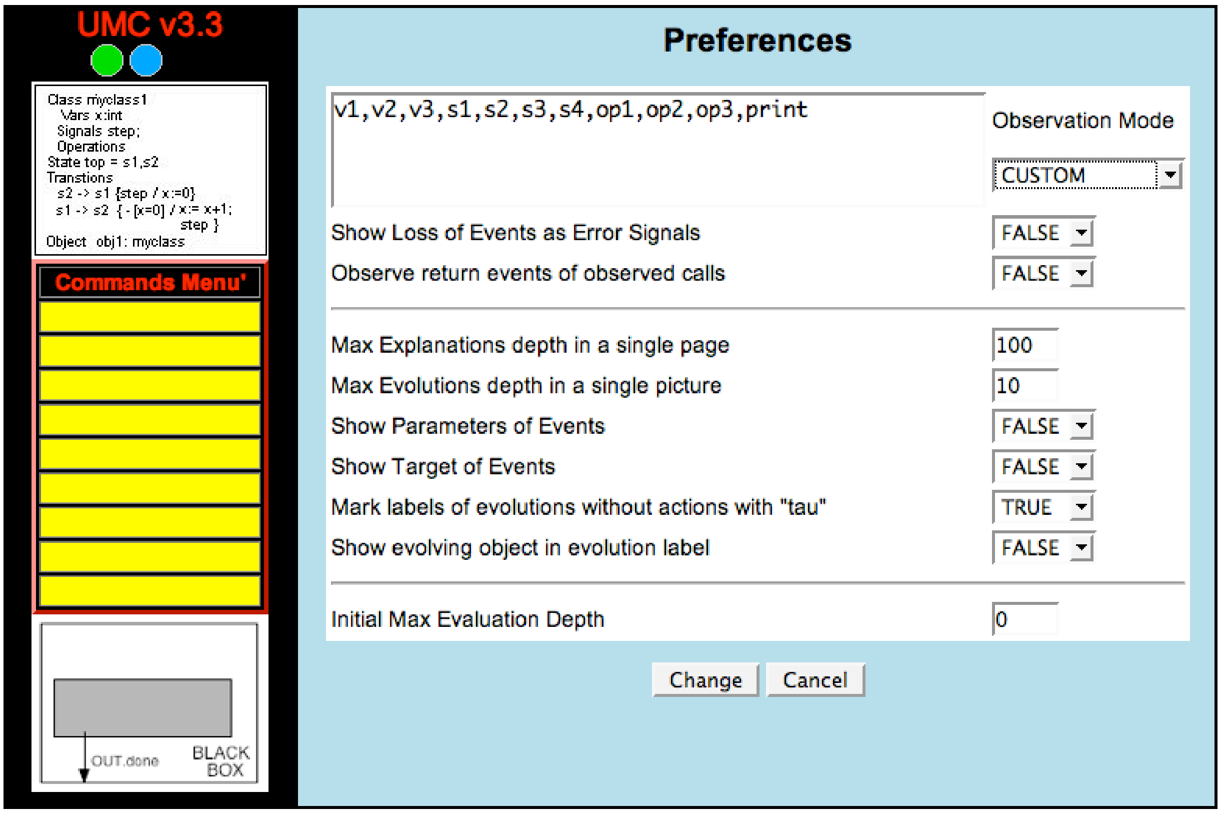
(You can observe the details of a configuration clicking over the corresponding node).

If If you cannot visualize then picture below, you probably need to install a svg-viewer plug-in like that of Adobe.
Use Command+mouse to zoom in, Command+shift+mouse to zoom out, alt+mouse to move the picture. Use ctrl+mouse to access the svg menu.

Customizing some user preferences

The behaviour of the underlying "umc" tool, and in particular the abstraction level / observations point of view, 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 4.1 under the title "Viewing and adjusting some tool preferences and paramters".
Here we only recall the “Observation Mode “parameters, which are those allowing to specify which events are we interested to observe, when drawing graphs or checking the system behavior. These “Observation Mode “parameters are constituted by a Select button through which we can select one of the predefined `BLACK_BOX`, `GRAY_BOX`, `WHITE_BOX`, `SELECTIVE`, `INTERACTIVE CUSTOM`, observation modes, and by a text area inside which we

can specify in the case of `SELECTIVE, INTERACTIVE CUSTOM` observation modes further information identifying the objects, the events and the attributes to which we are interested.
A picture representing the currently active “Observation Mode” is also shown on the left frame of the tool which, if clicked over, allows to observe of configure the preferences of the tool in the same way.



```
Class myclass1
  Vars x:int
  Signals step;
  Operations
State top = s1,s2
Transtions
  s2 -> s1 {step / x:=0}
  s1 -> s2  { - [x=0] / x:= x+1;
                         step }
Object  obj1: myclass
```

v1,v2,v3,s1,s2,s3,s4,op1,op2,op3,print

Observation Mode	CUSTOM
Show Loss of Events as Error Signals	FALSE
Observe return events of observed calls	FALSE
Max Explanations depth in a single page	100
Max Evolutions depth in a single picture	10
Show Parameters of Events	FALSE
Show Target of Events	FALSE
Mark labels of evolutions without actions with "tau"	TRUE
Show evolving object in evolution label	FALSE
Initial Max Evaluation Depth	0

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-UCTL 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-UCTL, and an exploration module which is support the interactive exploration of the system evolutions.

Our approach to the "on the fly" model checking of a mu-UCTL 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;
```

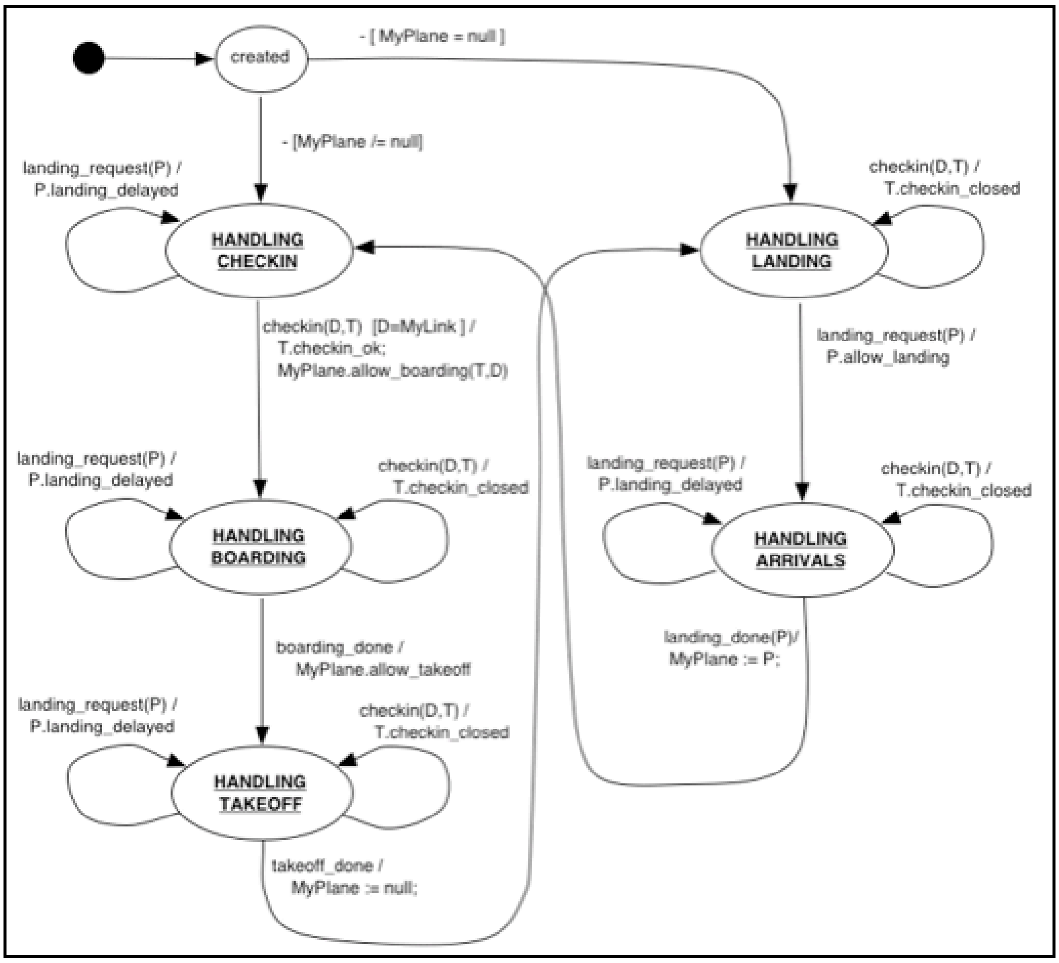
This approach seems promising when applied to UML state machines (or groups of communicating state machines) because it can easily be extended also to the case of potentially infinite state space, as it may happen for UML state machines. Indeed, a problem of the above evaluation schema is that, in case 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.

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

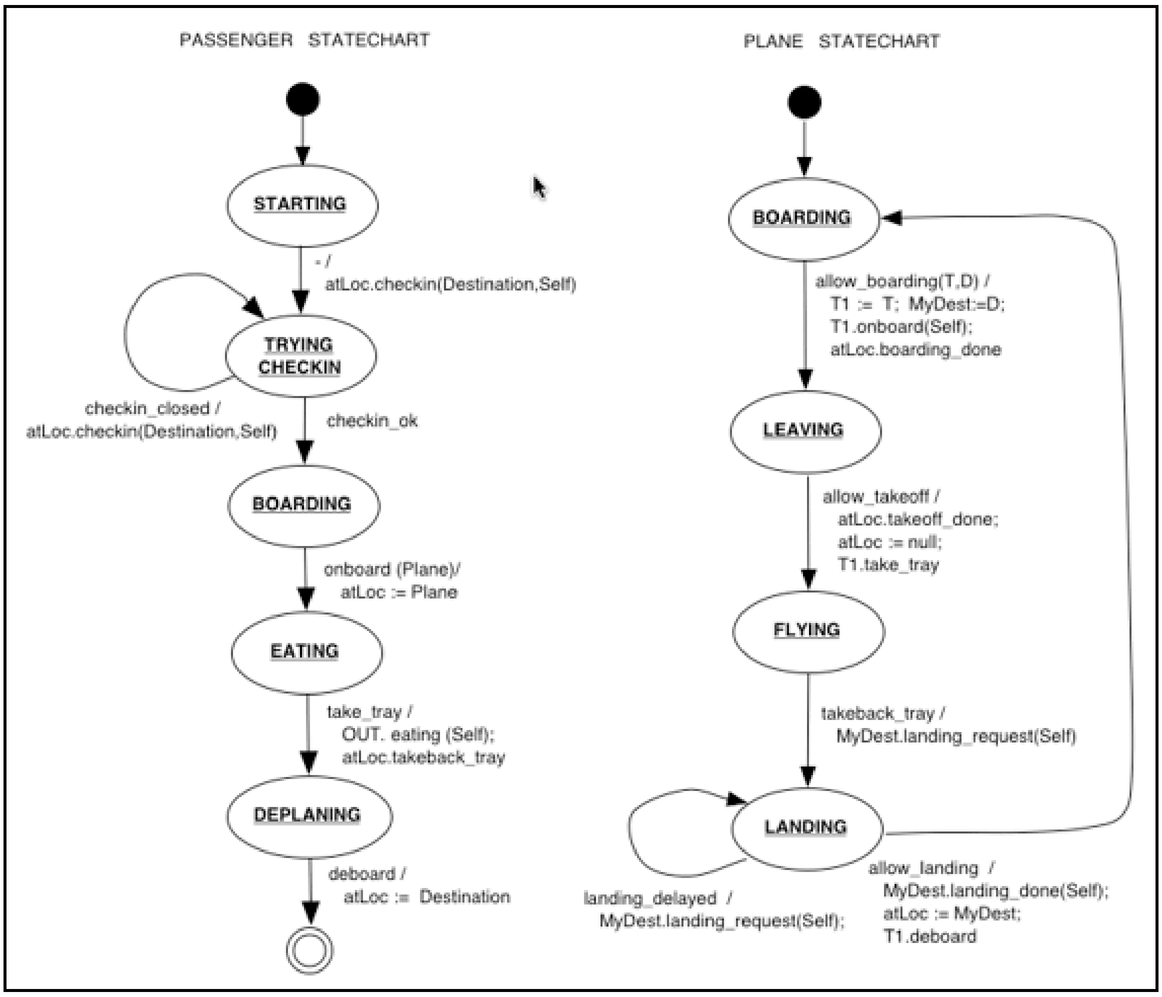
Let as consider as a toy example, a system constituted by two airports, two passengers (one at each airport), and a plane. The plane is supposed to carry exactly one passenger and flies (if it has passengers) between the two airports. Departing passengers try to check in at the airport and than board 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.


created
- [MyPlane = null]
- [MyPlane /= null]
landing_request(P) / P.landing_delayed
HANDLING CHECKIN
checkin(D,T) / T.checkin_closed
HANDLING LANDING
checkin(D,T) [D=MyLink] / T.checkin_ok; MyPlane.allow_boarding(T,D)
landing_request(P) / P.allow_landing
landing_request(P) / P.landing_delayed
checkin(D,T) / T.checkin_closed
HANDLING BOARDING
landing_request(P) / P.landing_delayed
checkin(D,T) / T.checkin_closed
HANDLING ARRIVALS
boarding_done / MyPlane.allow_takeoff
landing_done(P)/ MyPlane := P;
landing_request(P) / P.landing_delayed
checkin(D,T) / T.checkin_closed
HANDLING TAKEOFF
takeoff_done / MyPlane := null;

```
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
```

```
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
```

```
   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
```



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.

8 Known limitations, 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 and call opeartions are supported. Time events 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.

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.

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.isti.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[3] project.
Useful feedback and suggestions have come also from the QUACK[4] and PRIDE[5] 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].

[3] AGILE project IST-2001-32747 by the proactive Initiative on Global Computing
[4] QUACK, A Platform for the Quality of New Generation Integrated Embedded Systems (Pr. MURST 40%)

Feedback

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

9 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
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.

[5] PRIDE, ambiente di PRogettazione Integrato per sistemi DEpendable(Italian Space Agency, 2002)

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.
26 S. Gnesi, F. Mazzanti:"A Model Checking Verification Environment for UML Statecharts" XLIII Congresso Annuale AICA, Udine 5-7 Ottobre 2005
27 S. Gnesi, F. Mazzanti:"On the fly model checking of communicating UML State Machines" Second ACIS International Conference on Software Engineering Research Management and Applications (SERA2004) (Los Angeles, USA, 5-7 May 2004).