Consiglio Nazionale delle Ricerche

Technical Report

Designing UML Models with UMC

Franco Mazzanti

2009-TR-043

ISTI

Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"

Pisa

Designing UML Models with UMC

(ref. UMC V3.6 build p - April 2009)

Franco Mazzanti

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

1 Introduction

The goal of the UMC project under development at ISTI 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 [1,2,3,4].

- 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 [5,6] 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, together with distribution and mobility aspects.

This experimentation is carried out through the actual development of a sevral verification tools (FMC,UMC,CMC), specifically tailored to the aims of the project. In this paper we will describe how UMC can be used to generate system models according to the UML paradigm, models which can later be explored in their evolutions, abstracted, minimized and checked w.r.t. formal temporal properties (expressed in UCTL).
We anticipate that the immediate purpose of the UMC project is definitively not that one of building a commercial 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 features, and not yet on the quantatitive optimizations of them (in terms 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. Also in terms of future plans, the emphasis is more towards the experimentation of new features than towards to the freezing and optimization of the current release.
In the case of UMC, 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 - by a set of objects instantiations, and by a set of abstraction rules.

In Section 2 we describe in more detail the main syntactic model components, namely classes, objects, basic types, expressions and abstraction rules; in Section 3 we describe an overview of semantics of the modelled UML behaviour. In Section 4 we describe the structure and semantics of the observation rules. In Section 5 we show a BNF grammar for the umc language, and in Section 6 we show some examples.

We anticipate that the language of UMC defined for the encoding of an UML system actually suffers a problem of schizophrenia.. In fact, from one side "UMC" should ideally be seen as a target language of a translation from other higher levels (maybe graphical) design/modelling languages like Java/UML (indeed Java has many important notions like private/public methods, strong type checking , ... and all these aspects would require an high degree of checking which should be perfomed statically at the Java or UML definition level and not at the UMC level). On the other side, since currently there is no such translator from higher level Java/UML, in practice what happens is that users directly uses UMC for writing their model specification; for this reason some essential checks and syntactic suger have been introduced into UMC, but not as much as if UMC were considered a real specification language. As a consequence, the overall shape of the UMC language is definitely not as clean as it should be if it were thought to be precisely either an internal encoding language (like java bytecode) an high level self-standing user-centered design language.

2 The structure of UMC models

A complete umc model description is given by providing:

- a set of class declarations
- a set of object instatiations
- a set of abstraction rules.

Hence, the template of a umc model is the following:

```
Class classname_1 is
   ...
end  classname_1 ;
   ...
Class classname_n is
   ...
end classname_n ;

Objects
  objname_1:  classname_1 ... ;
  objname_n:  classname_n  ... ;
   ....

Abstractions {
   Action ... -> ...
   State  ... -> ...
   ...
}
```

Class declarations represent a template for the set of active or non-active objects of the system. In the case of active objects the states and transitions associated to class are 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 in the end constituted by static set of objects (no dynamic object creation), and it must be an “input closed” system, i.e. the input/stimulating source must modelled as an active object interacting with the rest of the system (e.g. modelling a service).
At least one active object must be defined.
The abstraction rules do not play any role from the point of view of the ground behavioural semantics of the model under analysis; they define instead what we are interested to observe w.r.t. the overall system behaviour. From this point of view their role is essential for constructing trace minimizations of the graph illustrating all the possible system evolutions, and for verifying system properties through model checking.

Class declarations

Classes define the structure and dynamic behaviour of the objects which compose the system.
Classes are defined by class declarations which introduce:

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

The template of class declaration is shown below:

```
Class classname is

 Signals
  ...                    -- list of asychronous events accepted by the class objects

 Operations
  ...                    -- list of synchronous operation calls accepted

 Vars
  ...                    -- list of local, private, attributes of the class objects

 State  ... = ...
 State  ... = ...        -- the structure of statechart nodes and subnodes
 State  ... = ...

  ... -> ... {...}
  ... -> ... {...}       -- the definition of the edges of the statechart
  ... -> ... {...}

end  classname;
```

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

There is a predefined non-active `Token` Class whose objects play the role of static literal names.

Basic Types, Literal Values, Expressions

There are three main basic types, namely `int` (with default value `0`, `bool` (with default value `False`), and `obj` (with default value `null`).
Of these types there is also the corresponding vectorial version `int[]`, `bool[]`, `obj[]` (all with default value "`[]`" denoting the empty vector).
Class names can also be used as specialized object type names in place of the more generic `obj`.

The operators over integer values are just the basic binary "`+`" (plus), "`-`" (minus)[1], "`*`" (times), "`/`" (integer division) and "`mod`" (modulus) operators. Composite arithmetic expressions can be written using paranthesis.("`(`" "`)`").

The operators over boolean values are just the basic "`and`", "`or`" and "`not`" operators.
The basic boolean literals are "`True`" and "`False`". There is also a special name "`emptyqueue`" which denotes the value "`True`" if the queue of object evaluting it is currently empty, and denotes the "`False`"otherwise.
The relational operators "`=, /=, >, <, >=, <=`" allow to compare two integer values returning a boolean value[2]. The relational operators "`=`" and "`/=`" allow to compare for equality any two element of the same type (hence also vectors and class objects).

There are no operators for `obj` or classname objects. The special literals "`self`" and "`this`" both denote the executing object when evaluated at runtime. "`null`" denotes no object. The global names of the objects consituting the system declared in the `Objects`: section can be used as literal values of obj or class type.

The expression "`[]`" denotes the empty vector, the expression "`[` *value1* `,` ... `,` *value_n* `]`" is used to denote array literals, and the operator "+" can be used to join two arrays (e.g. `[1]+[2] = [1,2]`). The "`.head`" selector applied to a vector variable name returns the first element of the vector, and the "`.tail`" selector returns the remaining part of the vector one the first element has been removed (e.g.if `v1=[1,2]` and `v2=[3,4]`, `[v1.head] + v2.tail = [1,4]`).
The selector "`.length`" returns the number of elements of a vector variable.
The expression vectorvar[i] allows to select the i-th+1 element of the vector variable (e.g. if `v1=[3,4,5]`, `v1[0] = 3, v1[1] = 4, v2[2] = 5`). The selection of the i-th element of a vector contains less than i elements returns the default value for the vector elements type.
The i-th+1 element of a vector variable can be changed by assigning to the selected component (e.g. `v1[1] := 4`). It is an error to try to perform an assignent to the i-th element of a vector which contains less than i elements. Errors of this kind do not interrupt the tool execution but just the single evolution path sending a "`Runtime_Error`" signal to the predefined `ERR` object.

Event declarations

The `Signals` and `Operations` sections define the set of asynchronous or synchronous events accepted by the class objects.

---

[1] notice that currently there is no unary minus.
[2] the relational operator "<" is also extended to cover all kind of types, e.g. objects or vectors, in an implementation dependent way. This is done to reduce the generation of Runtime_Error signals which may terminate the specific execution trace. Specification relaying on this implementation dependent aspect are however considered not well written.

Signal declarations may have the template:

```
-- signal with no parameters
signal_name;

-- signal with n parameters
signal_name (arg_1, ... , arg_n: type_n);
```

Operation declarations may have the template:

```
-- operation with no parameters and no result value
op_name;

-- operation with n parameters and result type.
op_name (arg_1, ... , arg_n: type_n): result_type;
```

The typing of a signal or operation parameter or operation result is not strictly required, however it is definitevely recommended to allow some degree of static checking and to enable a more meanininful display of the runtime value of the parameters exchanged messeges (which otherwise are displayed just using the internal integer encoding of their values).
All the parameters of signals and operations have an implicit *in* mode (in the sense that assignments to them are not propagated back to the sender/caller).

Local Variable declarations

The `Vars` section introduces the list of attributes of the objects of the class. Notice that all these attributes are local to the object (not shared among all objects of the class) and private (not accessible by other objects). For these reasons they are often called as "local variables". Indeed they have the role of local, private variables of the class objects, with the exception of the `Priority` and `RANDOMQUEUE` cases in which play a rather different role (as described in Section3).

Vars declarations may have the template:

```
-- untyped, initialized by default, multiple local variable
varname1; varname2; varname3;

-- expicitly typed local variable
varname: int;

-- explicilty typed, expliciltly initialized, local array [3]
varname1: int[] := [1,2,3];
```

As for the case of event parameters, the typing of a local variable is not strictly required even if it is definitevely recommended to allow some degree of static checking and to enable a more meanininful display of the runtime value of the object attributes (which otherwise are displayed just using the internal integer encoding of their values).

The local variable "`Priority: int;`" when declared as first variable of the `Vars:` section has a special meaning w.r.t. system scheduling issues (this occurs only when all active classes have such variable defined). The local variable "`RANDOMQUEUE`" when declared immediately after the

[3] currently multi-var declarations cannot have explit initial values.

"`Priority`" variable (or as first variable of the `Vars:` section of a class declaration) has the meaning of specifying that the events queue for the objects of that class should behave as RANDOM queue and not as a FIFO queue, as it happens by default. These two aspects are presented in more detail in Section 3.

States Structure Definition

The active dynamic behaviour of the objects of a class is described as an UML statechart diagram by a set of rules (`State` ... rules) wich describe the set of nodes which constitute the statechart, and by a set or rules (`Transitions:` ...) which define the edges of the statechart.

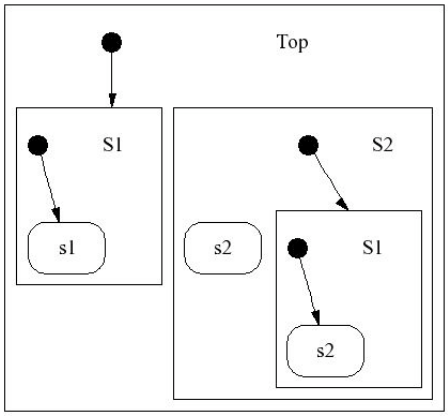
The definition of the nodes of a statechart diagram starts from the definition of the top level state (usually called "Top", but just as a convention)
The definition of nested substates be must preceded the definition of the outer substates.
The states can be classified in *simple-state*, *composite-sequential-stat*e and *composite-parallel-state, initial-pseudostate*, *final-state*..

A composite sequential state acts as a container for a nested statechart diagram.
The template for its declaration is the following:

`State` *parentstate* `=` *substate_1*`,` *substate_2*`, ... ,` *substate_n*

Where *parentstate* univoquely denotes either the outmost top-level state or an already defined nested substate. The names *substate_1*`,` *substate_2*`, ... ,` *substate_n* denote the substates of the *parentstate* state. If a substate represents a nested composite (poarallel or sequential) state, we need to expliclty declare it as such later. If a substate denotes a simple state no further explicit declaration is required.

```
State Top = S1, S2
State S2 = S1, s2
State Top.S1 = s1
State S2.S1 = s2
```



Usually, as a convention, composite state are written with initial letter in uppercase, while simple states are written with initial letter in lowercase. Notice that disambiguation of *parentstate* names is achieved by prefixing the actual state name with a sufficient sequence of the names of the outer states in which it is included.

The first substate of a composite state is assumed to be its default initial substate unless an explicit "initial" substate if defined. The name "initial" denotes the default initial pseudostate (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.

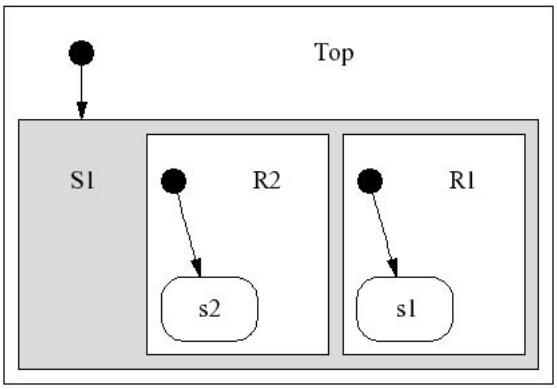
Formally, according the UML2.0 defintion, a composite sequential state actually includes an implicit, anonymous internal region which in turn includes all the required substates; such implicit region is never explicitly defined or shown by umc.

A composite parallel state is defined in UML2.0 as a composite state which contains more than one region. In this case the number and names of the constituting regions must be explictly defined. For each such region, moreover, we need to subsequently given an explicitly definition as we do for a composite sequential state.
The template for the declaration of a composite parallel state is the following:

`State` *parentstate* `=` *region_1* `/` *region_2* `/ ... /` *region_n*

In the following example S1 is a parallel state, R1 and R2 are its two regions, s1 and s2 are simple states.

```
State Top = S1
State S1 = R1 / R2
State R1 = s1
State R2 = s2
```



Deferrring states
A state definition can also define the set of events deferred while being active.
A Defers declaration allows to specify efines the list of events (matching those already declared as Signals or Operations) deferred by a state, which can be either a simple or composite sequentai state. The template for a deferring declaration is the following:

`State` *statename* `Defers` *event_1* `,` ... `,` *event_n*`(`*arg1*`,...,`*argn*`)`

Notice that the events appearing in the deferrring declaration must have exaclty the event name and possibly sequence of parameters, as in their corresponding definition; the type of parameters can be, on the contrary, omitted[4].

Transitions Definition

The `Transitions` section of the class declaration allows to define the edges of the statechart modelling the behaviour of the objects of the class
This sections contains a sequence of transition definitions, which in general have the template:

source `->` *target* `{` *trigger* `[` *guard* `]` `/` *actions* `}`

.

[4] and thery are actually ignored if provided.

The *source* and *target* can either be a single name univoquely identifying a state, or a list of such names (i.e. `(`*name_1*`,...,` *name_n*`)`). A transition with multiple sources is called a *join* transition. A transition with multiple targets is called a *fork* transition. In both cases the multiple sources or target must belong to different regions of the same parallel state, while the corresponding target or source must be outside of the previously mentioned parallel state. In the case of *join* transitions, the first state in the source list is required to be "*the most transitively nested source state*" in the sense of UML(); in fact the first state univoquely determinates the priority of the transition.

The *trigger* and *guard* of a transition define the conditions under which the *source* state (which miust be active) is exited and the *target* state entered.
The *trigger* of a transition is constituterd by an event name and possibly its sequence of formal parameters as defined in the Signals or Operations Section. As for events appearing in deferred declarations the type of parameters can be, on the contrary, omitted. The expression "`-`" denotes the absence of trigger. This means that the transition is what UML calls a "completion transition".
If the trigger denotes an operation call the identity of the calling object can be accessed through the implicit "`_caller`" parameter of the operatio; this value is also implicilty used by a "`return`" action to notify the completion of the invoked behavior.
The *guard* component is optional. If present is should be a boolean expression involving trigger parametrs and /or local variables.
The *actions* part defines a sequence (possibly empty) of basic action which are executes when the transition is selected. The main examples of actions are the sending of a signal, the call of an operation, an assignment to a variable.
The detailed semantics of how transitions are selected and fired according to the UML semantics is rather complex and is described in Section 3.

Examples:

```
s1 -> s2      -- a very simple completion transition with no guards nor effects

s1 -> s2
   { myopcall(x,y)[_caller=obj1 and x>0] /
     v1:=x;
     returb;   -- a full transition with trigger, guards and actions
   }
```

Actions

UMC supports several kind of actions: assignments, sending of events, conditionals and finite loops.
Assignments have the form:

```
varname := right_side;              -- assignment to local variable
varname[index]:= right_side;        -- assignment to a component of vector
```

The *right_side* of an assignment can be an expression or an operation_call.
The *varname* must be the name of a local variable, or the name of the of the parameters of the transition trigger, or the name of a local transition variable.

It is in fact possible to declare temporary variables inside the list of actions of a transition. In this case the scope of the transition variable is the rest of the action sequence. Transition variables can appear in the same places as local variables in the remaining part of the action sequence of the

tramnsition. The declaration of a transition variable has the same form as the declaration of a local variable. An example is shown below:

```
s1 -> s2 {-/ tmp:bool:= b1 and b2; b3:= tmp or b3 }
```

Asynchronous sending of signals is another important kind of action. It has the form:

target_object`.`*signal_name*`(`*expr_1*`,`...`,`*expr_n*`);`

The *target_object* must be a name denoting one of the objects which constitute the system (hence either a variable name,or a parameter name or a global static object name).
The *signal_name* must be the name of a signal event declared in the `Signals:` section of the class of the target object[5]. The number of expressions following the signal name must match the number of parameters of the signal profile[6].
The execution of this action causes the enqueuing of the corresponding signal message in the event queue of the target object. If no target_object is specified, then "self" is implicitly assumed.
If the target:_object does not denote one of the objects which constitute the system an error occurs. which interrupt the particular evolution path sending a "`Runtime_Error`" signal to the predefined `ERR` object.

Synchonous calling of an object operation is achieved through the operation_call action which has the form:

target_object`.`*operation_name*`(`*expr_1*`,`...`,`*expr_n*`);` *-- plain operation call*

varname `:=` *target_object*`.`*operation_name*`(`*expr_1*`,`...`,`*expr_n*`);` *-- function call*

Like the sending of an asynchrnous message, the sending of a synchronous operation_call message involves the identification of the target_object, the operation_name, and possibly a list of arguments. The *operation_name* must be the name of a operation event declared in the `Operations:` section of the class of the target object. The number of expressions following the operation name must match the number of parameters of the signal profile. The parameters always have an "in" mode and, even if assigned by the target object, do not convey back any modified value. The value returned by the operation call can be retrived by assigning it to a variable inside and assignement. When a call-operation is accepted by an object, i.e. when one of the transition triggered by the operation event is fired and the corresponding sequence of action is executed, the execution of a `return(`*return_expression*`)` action causes the sending to the caller of an operation_return message. If the operation declaration has no return type, and the execution of the sequence of actions of a transition is completed without the execution of an explicit return, then a final implicit return action is executed.
Examples:

-- operation call with no return value
`c1 -> c2 { - / server.write_operation(123)}` -- caller side

-- operation execution with implicit return action

---

[5] the checking performed on this kind of constraint is not very precise, because in general the class of the target object is not statically detectable if untyped "obj" vars are used. A weaker check is instead performed, i.e. that the signal sent is actually declared by some class.
[6] no type checking is actually performed on the correspondence between actual value and formal parameters type.

```
s1 -> s2 {write_operation(val) / var := val}          -- called side

-- operation call execution explicit return action
s1 -> s2 {write_operation(val) / var := val; return} -- called side

-- function call
c1 -> c2 { - / var := server.read_var_operation}      -- caller side

-- function call call execution explicit return action
s1 -> s2 {read_var_operation / return(var)}           -- called side
```

Composite actions are essentially conditionals and finite loops.
The form of a conditional is one of the following:

```
if condition then { actions-list } else { actions-list };
if condition then { actions-list };
```

Where condition is a boolean expresssion, and the actions-lists are sequences of actions.

The form of a finite loop is:

```
for iterator in min_expre .. max_expr{ actions-list };
```

Where *iterator* acts as a transition variable initialized with *min_expr* and incremented at the end of each cycle until it excedees *max_expr*. The *actions-list* is the sequence of actions which is executed at each cycle. Both *min_expr* and *max_expr* are evaluated before the beginning of the cycle.

Objects Declarations

Once the needed classes are have declared, we can define the actually deployed system as a set of object instances. This is done inside the `Objects:` section of the model definition. Each object instance is declared by an object declaration which has the following form:

```
object_name: class_name                    -- an object declaration with initiaizations
        (obj_attribute_1 => initial_value_1,
                ...,
          obj_attribute_n => initial_value_n);

object_name: class_name;                   -- an object declaration without initiaizations
```

Each object declaration introduces the object name, the name of its class, and possibly any specific initial values for some its attributes.
These initial values can be literals or names of other objects (possibly also objects which will be declared later in this section).
The object names introduced in these declaration act a gloabl literal names and are fully visible also inside all the class decalations.

Abstraction Rules

Abstraction rules are introduced inside the specific

```
Abstractions {
   ...
}
```

section of the model specification. As already hinted, they do not play any role in the definition of the ground dynamic behaviour of system. They have instead all to do with respect to what we want to observe of the system evolutions. Hence they become essential when we want to state properties to be verified over the system, when we want to generate minimized abstract views of all the possible system behaviours, or when we want to explore and simulate the system execution.
For this reason their are not described here, but in Section 4, after the presentation of the dynamic semantics of UMC models.

Other syntactic issues

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

Static type checking of the model is performed only in a limited way.
If some ot the vars or parameter types are not specified, sometimes an attempt is made to statically/dynamically infer them. A system design containing structural type violations is considered erroneous and it not guaranteed that such errors are detected either statically or dynamically. Notice that a model might violate exploit the absence of explicit static typing rules (e.g. defining and using a polimorphic queue) without actually generating any structural type violation.

Line comments start with "-- " or "//" and end at the end of the line
/* ... */ can be used to encapsulate possibly multiline comments.

Several syntactic altarnatives are supported (though not encouraged) just to simplify the encoding from other languages. In particular:

"and" inside boolean expression can be substituted by "&" or "&&"
"or" inside boolean expressiocan be substituted by "|" or "||"
"not" inside boolean expressio can be substituted by "!"
"/=" inside boolean expressio can be substituted by "!="
"=" inside boolean expressio can be substituted by "=="
":=" in assignments can be substituted by "="
"=>" in object instantiations can be substituted by "=" or "->"

The full adherence to the umc grammar is not always strictly enforced when the semantics of the code remains clear (e.g. sometimes the "`:`" following a keyword can be omitted, as the class name after the "`end`" keyword, or as the "`;`" following the last action of an action-lists).

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 Terminate / History / Deep-History / Synch states, state Entry / Exit / Do activities, Dynamic Choice / Static Choice transitions, Change and Time events. With respect to UML2.0 we currenty do not support state refinements, inheritance in events structure, substatemachines and the possibility of having multiple triggers for a transition.

We believe, however, that most of these features can still be quite easily encoded in our framework. In UMC composite states always have a default initial state (which is the first substate of the list), while in UML2.0 the legality and semantics of a composite state without any default initial substate is a semantic variation point. None of the above limitations is intrinsic to the tool, and further versions of the prototype are likely to overcome them

3 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 pictures remains essentially the same also in UML2.0. 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) Dealing with active states, triggers and guards: It is identified the set of transitions whose source states are active in the current configuration, whose trigger satisfies the current top event (if any) of the 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).

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

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

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

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

The above steps defines the possible effects of starting a nre "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 step will be 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. In our model we suppose that all completion events are dispatched all toghether in a unique step[7].

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 of vectorial data elements. The following is an example of very simple model presenting an infinite behaviour:

```
Chart Main is
  Signals: a
  State Top =  s1
  Transitions:
   s1 -> s1 { - / self.a; }
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) 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) Given a model constituted by more than one state machine, a single 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 synchronoes calls are involved.
If priorities are defined (i.e. all active classes have as first variable of the "`Vars:`" list a variable named "`Priority`" and with type "`int`"), then only the evolving objects of the highest priority value are considered for defining the possible system evolutions.

3) The propagation of messages 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).

[7] From this poi of view the UML semantics is not very clear.

4) The events queue associated with a state machine is by default a FIFO way (this is an aspect intentionally left as unspecified by the UML standard). If a class declaration has as first variable declaration (or as second variable declaration, immediately following a Priority declaration) an untyped entity named "RANDOMQUEUE" then the events queues for the objects of that class are handled according to a RANDOM policy (i.e. any enqued event is elegible for being dispatched, independendetly from its position in the queue).

E.g.
```
Vars:
        Priority:int :=2;
        RANDOMQUEUE;          -- objects queue are RANDOM, not FIFO
        ....
```

5) 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.[8]

6) The return signal from an operation/function call is sent 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. . If no return statement is execution by the run-to-completion step triggered by a call no return event is generated and the caller deadlocks.

4 Abstraction Rules

As already said in Section 2, abstractions do not play any role in the definition of the ground dynamic behaviour of system since their role is to define what we want to observe of the system evolutions. Abstracion rules are of two kind: `Action`: rules and `State:` rules.
The `Action:` rules allow to define which *events* occurring during the firing of a transition, we want to observe. The `State:` rules allow to defined which structural aspect of a sytem state we want to observe.

action abstractions

The general form of an action abstraction rule is the following:

`Action:` *source_obj*`:`*target_obj*`.`*event*`(`*arg_1*`,...,`*arg_n*`) ->` *main_label*`(`*flag*`,...,` *flag*`)`

Where source_obj and target_obj must either be an object name lietral, event must be either a signal or operation name, or another predefined eventname, arg_1 ... arg_n must be literal values.
The mail_label and the associated flags are just free identifiers.
The Source_obj: and target_obj. prefixes can be omitted, as can be omitted the list of flags associated to the main_label. Hence a minimal Action rule could just be:

---

[8] 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".

`Action:` *event* `->` *main_label*

Let us now see in more detail the meaning of these rules.

static pattern matching

The rule:

```
Action: reset -> resetting_request
```

States quite generically, that whenever during a system evolution a "`reset`" message happens to be sent, independently from the absence of presence of parameters, and independently from their number, then that evolution is labelled with the abstract label "`resetting_request`". While the rule:

```
Action: reset() -> resetting_request
```

States more specifically, that whenever during a system evolution a "`reset`" message with no parameters happens to be sent, then that evolution is labelled with the abstract label "`resetting_request`".
The rule:

```
Action: reset(soft) -> soft_resetting_request
```

States that whenever during a system evolution a "`reset`" message with exaclty one parameter whose values is equal to the (token) literal "soft" happens to be sent, then that evolution is labelled with the abstract label "`softresetting_request`".
The following rule:

```
Action: obj1:reset -> obj1_asking_reset_request
```

States instead that whenever during a system evolution a "`reset`" message happens to be sent by object `obj1` to any other object, then that evolution is labelled with the abstract label "`obj1_asking_reset_request`".

The following rule:

```
Action: obj2.reset -> resetting_request_for_obj2
```

States instead that whenever during a system evolution a "`reset`" message happens to be sent by some object to object `obj2`, then that evolution is labelled with the abstract label "`resetting_request_for_obj2`".

The following rule:

```
Action: obj1:obj2.reset -> obj1_sending_reset_request_to_obj2
```

States instead that whenever during a system evolution a "`reset`" message happens to be sent by object `obj1` to object `obj2`, then that evolution is labelled with the abstract label "`obj1_sending_reset_request_to_obj2`".

generic matching

Any literal or name if the left side of the rule can be replaced by an "*" symbol, meaning that any dynamic value generared at runtime will satisfy the rule for that component.

E.g.

```
Action: reset(*,0) -> resetting_request
```

The above rule establishes that whenever during a system evolution the signal "`reset`" is sent by some object to another, and that signal has exactly two arguments, and the second of them is equal to the numeric literal "`0`" then that evolution is labelled with the abstract label "`resetting_request`". Notice instead that

pattern matching with variables ans substutitions

Sometimes it is useful to convey in the abstract label of an evolution some dynamic information information extracted from a specific occurred event. This can be done using $variables at the place of names or literal inside the left side of rules,using pattern matching with the actually occurring event to assign values to them and using and variable substitutions to assign then inside the abstract labels at the right side of the rule. Let us consider for example, the rule:

```
Action: $targetobj.reset -> kill_request($targetobj)
```

The above rules states that whenever during a system evolution a "`reset`" message happens to be sent to some (any) object, then that evolution is labelled with the abstract label "`killed`" having as flag precisely the name of the object to which the message is sent.
Let us see another example:

```
Action: $src.*.reset(*,$arg) -> killedby($src,$arg)
```

The above rules states that whenever during a system evolution a "`reset`" message with exactly two parametrs happens to be sent by some object, then that evolution is labelled with the abstract label "`killedby`" having as flag precisely the name of the object which sent the message and the second parameter of the message itself.

There is also a special variable notation for identifying a list of values, which ha the form "$*".
The effect of using this variable is as exemplified below:

```
Action: reset($*) -> kill_request($*)
```

The above rules states that whenever during a system evolution a "`reset`" message happens to be sent, then that evolution is labelled with the abstract label "`resetting_request`" followed by a sequence of flags which correspond exactly the the sequence of values (if any) used in the message.
Another possible use of this variable-list notation might be as below:

.

```
Action: reset(*,$*) -> kill_request($*)
```

The above rules states that whenever during a system evolution a "`reset`" message happens to be sent, then that evolution is labelled with the abstract label "`resetting_request`" followed by a sequence of flags which correspond exactly the the sequence of values (if any) following the first one (which is omitted) used in the message.

special events: accept, lostevent, Runtime_Errror, assign

The events shown so far correspond to effect of sending a message when a signal is sent of an operation or fumnction is called. These indeed are the events which directly correspond the the basic action of sending a signal or calling an operation. There are other "events", however that might be important to observe.
For example we might be interested in observing the fact that a certain message is removed from the events queue of an objects, and either discarded because it does not trigger any transition, or used to fire a set of enabled transitions. This fact can be observed using the "system defined" event names "accept" and "lostevent". Here follow some examples.

.

```
Action: $obj1:accept(reset,$*) -> starting_reset($obj,$*)
```

In the above case whenever during a system evolution a "`reset`" message happens to removed from the events queue and used (as trigger) to fire some transition, then that evolution is labelled with the abstract label "`starting_reset`" and flagged with the name of the executing object and the parameters (if any) of the reset message.
In the case of the accept pseudo-event at list the event name must be provided (i.e. it cannot appear without any argument). The genric rule:

.

```
Action: accept($e,$*) -> $e($*)
```

allows to observe (i.,e. labled the abstract evelutions) with the triggers (if present) of all the transitions fired by the system.

When an event is instead removed by the events queue of an object and simply discarded because in that object state there is no enabled transition which might fire (this is called a "stuttering" evolution), that can be observed using the "lostevent" pseudo event. as shown below.

```
Action: lostevent($e,$*) -> discarded_message($e,$*)
```

This kind of pseudo event is particularly useful because often its occurrence is the sign of the presence of design errors in the specification (e.g. a message is sent to a worng object, or at a wrong time).

.We have already sain that under certain circumstances an error message is generated and sent to the default ERR object (e.g. when a message is sent to a non existing object, or when an assignment is made to a nonexisting component of a vector). All these events can be captured by expliclty using the name of this error event which is "Runtime_Error" as shown below:

```
Action: $obj:Runtime_Error -> Design_Error($obj)
```

Currently the Runtime:_Error signal has no parameters.

Finally, we might be interested in observing that fact that a certain assign action is being executed by some object. Thic can be done using the predenfined "assign" pseudo event which has three parameters: the name of the variable being assigned, the index its component being assigned, and the value being assigned. The fact the a specific object `obj1` executes the assignament
`x[i]:=` *somevalue*`;` can than be observed by the following rule:

```
Action: obj1:assign(x,$i,$v) -> obj_changing(x,$v)
```

where `$i` will match the index of the component being assigned, and `$v` the assigned value.

state abstractions

The form of a state abstraction rule is the following:

```
State:
      ground_state_predicate_pattern and
            ...
      ground_state_predicate_pattern   -> main_label(flag,...,flag)
```

In the case of state abstractions the left side of an abstraction rule is allowed to be a conjunction of ground state predicate patterns. A ground state predicate pattern can be either a predicate on the active status of a substate of some object, or e relation involving the current values the local variables of some object.
E.g. The following rule:

```
State: inState(obj2.Top.s1) -> obj2_in_state_s1
```

States that whenever in a system configuration the object "obj2" is in the substate "`Top.s1`" then that configuration is labelled with the abstract label `obj2_in_state_s1.`
No pattern matching valiables are allowed inside this kind of ground state predicate.

The other kind of ground state predicate may have the form:

object_name`.`*attribute* *relop* *object_nam*`.`*attribute*
object_name`.`*attribute* *relop* *literal_value*

Where *object_name* is the literal name of one of the objects which constitute the system.
attribute is the name of one the local variables declared for the object class,
relop is one of the relational operators which are used in relational expression (e.g. "`=`", "`>`"),
and a *literal_value* can be a number, a boolean literal, or an object literal (e.g "`null`" or the identifier of some static object). Notice that currently vector literals are not allowed as literal values and compoenents of vectorial attributed are not allowed to be mentioned in State abstraction rules (this limitation may be overcome in futures releases).
For example, the following rule:

```
State:  obj2.speed < 30  -> obj2_slow
```

States that whenever in a system configuration the object "obj2" has the local variable "`speed`" holding a value which is smaller thehn `30`, then that configuration is labelled with the abstract label `obj2_slow.`
If the relational operator is the equality operator, then the right side of the comparison is allowed to be pattern matching variable as shown by the following example;

```
State:  obj2.speed = $v    ->  speed($v)
```

There are two other special cases of relations which can be used as ground state predicate.
One is when the special name "`maxqueuesize`" is used in the left side of the relation. E.g.

```
State:  maxqueuesize > 10    ->  Unfair_or_Diverging_Path
```

maxqueuesize denotes the maximum length of the event queues of the objects which consititute the system. Observing this system property may sometime be useful to observe and check for the boundness of the model.
A similar secial name is the "`queuesize`" identifier which can be used as if it were an object attribute, and which denotes the current lentgh of an object queue. E.g.

```
State:  obj2.queuesize > 10  ->  Unfair_or_Diverging_Path
```

default abstraction rules

If no `Abstraction {...}` section is provided in model specification the following one is assumed by default:

```
Abstractions {
  Action: $1($*) -> $1($*)
}
```

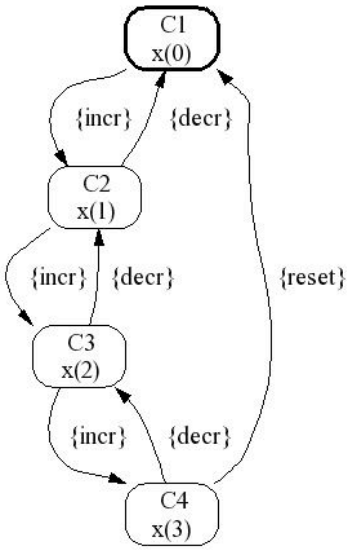
rule handling

All rules of the `Abstractions` section are taken into consideration and applied to the current state or evolution. As a result a null or multiple labelling may result, according to the number of rules which successfully match the current state structure or current set of evolution events.
By applying the abstraction rules, all the possible evolutions of an UMC system can be abstractly rrepresented as a bilabelled transition system (or directed graph), in which nodes and edges are labelled with sets of abstract labels, according the underlying ground configuaration structure or ground events occurring during the evolutions, and according to the abstraction rules specified.
We show below the specificiation of a very small system and its abstract evolutions doubly labelled transition sysyem.

```
Class Counter is
Vars x:int;
State Top = s1
Transitions:
 s1 -> s1
  {- [x<3]/x:=x+1;OUT.incr}
 s1 -> s1
  {- [x>0]/x:=x-1;OUT.decr}
 s1 -> s1
  {- [x>2]/x := 0;OUT.reset}
end Counter;

Objects:  OO: Counter;

Abstractions {
  State OO.x=$1 -> x($1)
  Action $1 -> $1
}
```



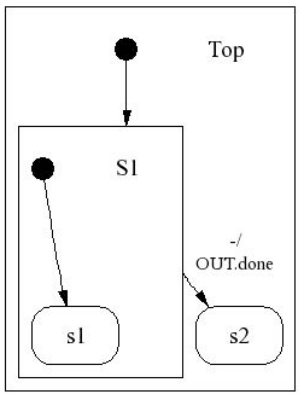
5 Some Examples

5.1 Completion Transitions

Edges which do not have an explicit event as trigger in UML are supposed to denote “completion transitions”, i.e. transitions which can occur when source state is active and it has completed all its internal activity. If the source state is a simple state it has no internal activity hence the outgoing enabled triggerless transitions can immediately be fired at the next step.
if the source state is a composite state it is necessary that all its regions are completed, i.e. all their active substates are either composite and completed, or the `final` pseudostate.
Notice that UMC does not directly support state internal “Do” activities, hence they cannot play any role in defining the completed status of a state.
For example, in the following case:

```
Class Deadlock is
State Top = S1, s2
State S1 = s1
 S1 -> s2 {- /OUT.done}
end Deadlock ;


Objects:  OO: Deadlock ;
```

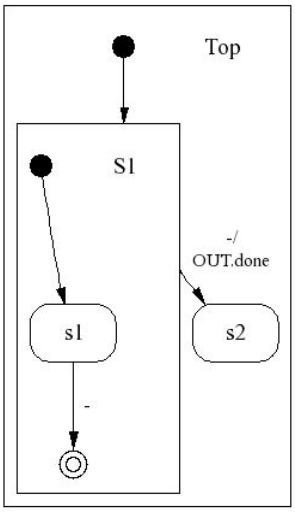


The system would not have any evolution since the S1 composite state is not ufficially “completed”.

In the following case, instead,

```
Class Evolving is
State Top = S1, s2
State S1 = s1, final
 s1 -> final
 S1 -> s2 {- /OUT.done}
end Evolving ;


Objects:  OO: Evolving ;
```



The system can evolve perfoming two steps: a
first step after which S1 becomes completed, and a
second step in which the system ,moves into the s2 state generating the done signal.

5.2 Recursive Operation calls

The semantics of an operation call is that one of suspending the execution of a run-to-completion step until a return event is received from the called object. This implies that no recursive operation calls can be performed by one or more object because in this case they would deadlock as soon as an operation call is performed targeting an object already suspended in an outer operation call.

The simplest case of deadlock will obviously occur when an object tries to call an operation of "self".

5.3 Parallel Operation Calls

We have already seen that when several transitions can be fired in parallel (because belonging to different concurrent regions) they are actually fired in any order but sequentially, as part of the same, unique, run-to-completion step. This fact, together with the fact that an operation call suspends the execution of the whole state machine until the return event is received, has a deep influence in the case of (apparently) parallel operation calls. A failure of the called object to correctly dispatch (and return to) one of the called operations in fact would suspend the initial state-machine preventing further evolutions of it even if, apparently, some region might be able to proceed in its execution flow. This is well shown by the following example:

```
Class ParallelClient is
Vars
  theserver: SequentialServer ;
State Top =  S1, final
State S1 = R1 / R2
State R1 = c1, c2
State R2 = c3, c4
  c1 -> c2
    { -/ theserver.add(10)}
  c3 -> c4
    { -/ theserver.sub(3)}
  S1 -> final
    { - / OUT.done}
end ParallelClient

Class SequentialServer is
Vars
  total: int := 0;
Operations
  add(x:int);
  sub(y:int);
State Top = s1,s2
  s1 -> s2
    {add(x) /
     total := total + x;
     return}
  s2 -> s1
    {sub(x) /
     total := total - x;
     return}
end SequentialServer

Objects
  Server: SequentialServer
  Client: ParallelClient
          (theserver => Server)
```

In the above case the Client object apparently issues in parallel two call operations (add and sub), and when both are completed sends the signal "done" to the standard OUT object.

Of the two possible system evolutions one of them actually terminate with the "done" signal being sent to OUT, while the other execution (that one in which "sub" is attempted before "add") shows a deadlock for the system (with no operation calls being successfully executed). Notice that in this particular case the adiition in the SequentialServer class of a "Defers" clause for its operations would solve the problem allowing the server to handle the two request also in the reverse order.

6 References

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

[2] *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).*

[3] *S. Gnesi, F. Mazzanti:* "A Model Checking Verification Environment for UML Statecharts" *XLIII Congresso Annuale AICA, Udine 5-7 Ottobre 2005*

[4] *Maurice H. ter Beek, Stefania Gnesi, Franco Mazzanti*: CMC-UMC: A Framework for the Verification of Abstract Service-Oriented Properties *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09), Honolulu, Hawaii, USA, ACM Press, 2009, 1844 - 1850*

[5] *Jacobson, I., Booch, G., Rumbaugh J.* "The Unified Modeling Language Reference Manual." *Addison-Wesley, 1999.*

[6] *OMG* Unified Modeling Language Specification,Version 1.4 beta R1, November 2000, *http://www.omg.org/technology/documents/formal/uml.htm).*