Defining a UMC Model

1 Introduction

A complete UMC model is constituted by a sequence of class and object declarations.
Classes are can model active or non-active objects. The initial system configuration is defined by a set of object instatiations.
A state machine (with its events queue) is associated only to the active objects of the system.
Non-active objects play the role of "interfaces" towards the outside of the system, and can only be the target of signals.
A system is constituted by fixed static set of active objects (no dynamic active object creation), and a system is a “closed” system, i.e. 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.

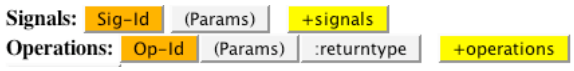
2 Syntax of UMC models

Classes

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

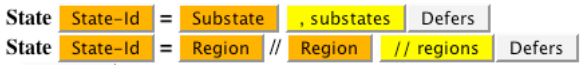
- the class name
- the list of events which trigger the transitions of the objects of the class (signals or call operations)
- the list of attributes (variables) local to the objects of the class
- the structure of the states of the class (nodes of the statechart diagram)
- the transitions of the objects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

The events handled by the class are distinguished between asychronous signals and synchronous call operations; the seconds can also have a return type.



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

The structural definition of the states of a statechart consists in a sequence of state definitions which starts from the definition of the top level state. The definition of outer states must precede the definition of its nested substates. The top level state of a statechart must be a composite sequential state.
A composite sequential state is defined by a list of substates (which can be composite sequential states, parallel states or simple states). The first substate of a composite state is assumed to be its default initial substate. 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.
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.
A state definition can also define the set of events deferred while active.

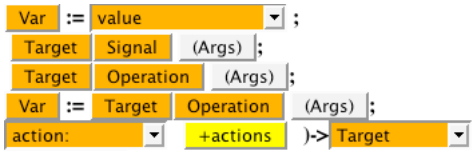


A Defers cause defines the list of events (matching those already declared as Signals or Operations) to de deferred.

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 chart variables.



The Actions part can be a sequence of simple actions. Each simple action can be an assignment of an expression to a local attribute, the sending of a signal to a target object, the calling of a synchronous operation on a target object, or an assignment of a synchronous function call to a local attribute.
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 destination specification the meaning is that the signal is sent to the events queue of the specified destination chart ("the term "self" can be used to denote the same object; if no destination is specified, then "self" is implicitly assumed).

Examples

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

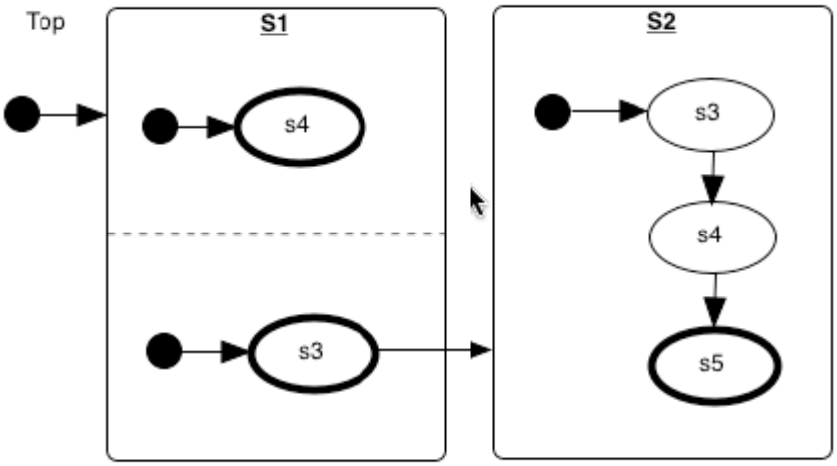


Figure 1: A first sample statechart diagram

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

Figure 2 A first sample textual representation of a statechart diagram

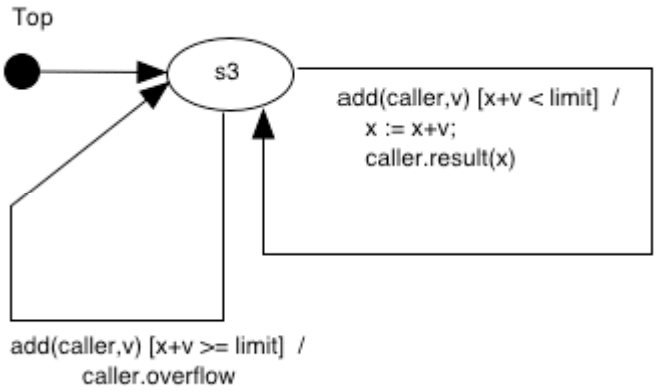


Figure 3: A second sample statechart diagram

```
Class Counter is
Signals: add(caller:obj, v:int)
Vars:  x:int, limit:int
State Top =  s3
Transitions:
  s3 -(add(caller,v) [x+v <limit] /
          x:=x+v; caller.result(x) )-> s3
  s3 -(add(caller,v) [x+v >=limit] /
         caller.overflow )-> s3
end
```

Figure 4 A second sample textual representation of a statechart diagram

Objects

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).

Example:

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

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

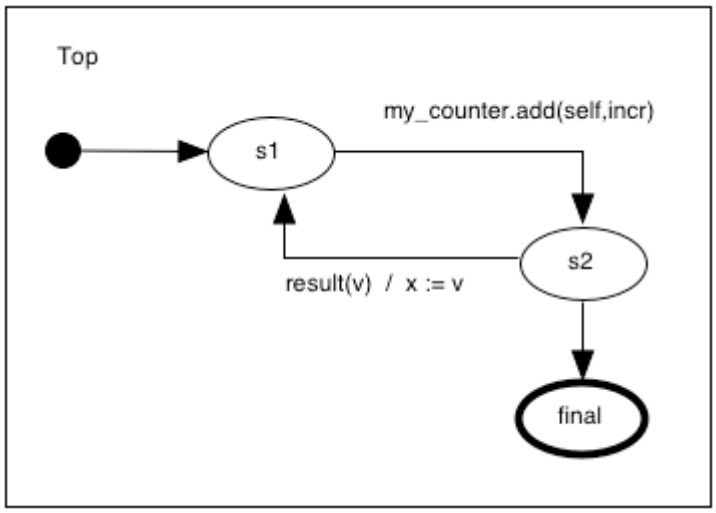


Figure 5: A third sample statechart diagram

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

Figure 6 A second sample textual representation of a statechart diagram

Expressions and Identifiers

An integer expression is either an integer literal, the name of an integer attribute or parameter, or a parantesized integer expression, or an composite expression built over one of the integer operands "+", "-" ,"*", "/", "mod" .

An object expression is either "self" or "null" or the name of object attribute or parameter.

A boolean expression is either an equality comparison between two object expressions (obj_expr1 = obj_expr2, obj_expr1 /= obj_expr2), or 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 ...) .

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

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

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 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 chart events queue, and whose guards evaluate to true in the current configuration. The resulting set is called the set of enabled transitions (w.r.t. active states, trigger, guards).

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

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

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

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

- modifying the events queue of the state machine by adding the signals specified by the sequence of sequences of actions, in their order.

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[1].

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) 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 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.

3) The propagation of signals inside a state machine and among state machines is considered instantaneous, and loss free (this is an aspect intentionally left as unspecified by the UML standard); the communication is direct and one-to-one (no broadcasts).

---

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

4) 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) The relative priority of a join transition is always well defined and statically fixed.[2]

[2] 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".