A Formal Semantics for UML Statecharts

Alexandre David   Johann Deneux   Julien d'Orso

Dept. of Computer Systems
P.O. Box 325
S-751 05 Uppsala, Sweden
{adavid,johannd,juldor}@docs.uu.se

Abstract

The UML language is a large set of notations and rules to describe different aspects of a system. It provides a set of diagrams to view the system from different angles: use case diagrams, class diagrams, statecharts diagrams, and deployment diagrams are some of them. In this report we are interested in the statecharts diagrams that describe dynamic behaviours. We give a formal semantics for a large subset of these statecharts, in particular we focus on the action language semantics. Our subset and semantics are very close to the one supported by the tool Rhapsody.

1 Introduction

The UML language is a semi-formal description language adopted as a standard by the industry. Its description is formalized with OCL (Object Constraint Language) but its semantics are left imprecise. Our motivation for this work is to be able to generate code or to simulate a system modelled using UML. To do this, we need an execution semantics for those features of UML we want to use. The sheer size of UML makes it impossible to treat all aspects in one document. In this paper, we treat a large subset of the *Statecharts* diagrams.

We emphasize here the fact that UML is generally loosely specified. It is naturally also the case for the statecharts. Since we need a formal semantics to be able to execute an UML model, we have been led to make design choices where the UML specification allows variations. We will try to make these choices clear, and justify them.

Related Work

- D. Latella, I. Majzik, and M. Massink. "Towards a formal operational semantics of UML statechart diagrams". In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems, pages 331-347. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6. In this paper the authors set the basis to model-check UML statecharts. They map the statecharts to an intermediate format of extended hierarchical automata and then they define an operational semantics for these automata. Our work is similar to theirs concerning the formal model of the statecharts, but we give a detailed description of the action language that they don't treat.

- Stefania Gnesi, Diego Latella, Mieke Massink. "Modular semantics for a UML statechart diagrams kernal and its extension to multicharts and branching time model-checking". In The Journal of Logic and Algebraic Programming 51 (2002) page 43-75. This paper gives a formal semantics to a subset of the UML statecharts with extension to branching time logic. They prove the correctness of their semantics with respect to major UML semantics requirements and they use their model in the JACK verification environment. Our work differs by the focus on the action language here as well. We put more emphasis on the run-to-completion semantics and event dispatching.

- Michael von der Beeck. "A Comparison of Statechart Variants". In Formal techniques in Real-time and Fault-Tolerant Systems, LNCS 863, pages 128-148. This paper is a survey on different approaches to give semantics to UML statecharts.

- Michael von der Beeck "Formalization of UML-Statecharts". In LNCS 2185, UML2001, pages 406-421. In this paper the author gives a syntax and semantics definition for the UML statecharts, in particular for the entry/exit actions and the history mechanism. They do not give any action semantics and the event dispatching mechanism is not treated as precisely as we do it.

- Ivan Porres “Modeling and Analyzing Software Behavior in UML”. Thesis 2001 Åbo Academie. In this thesis the author gives a formal semantics to the statecharts diagrams to be able to translate it to Promela with the help of the vUML tool. The semantics given fits SPIN. We stay closer to Rhapsody. Furthermore they do not give any action semantics because they use Promela’s features.

In most work very much effort is invested in defining the statecharts in terms of labeled transition systems, hierarchical or not, with an intermediate language or to a dialect of automata. The action language and primitive operations are left to the underlying used language, e.g. Promela for SPIN, simple assignments for automata (when variables are even used). On the other extreme work on action language, such as the response to the OMG RFP “Action Semantics for UML” focuses exclusively on the action semantics. We contribute in this work by giving an action semantics with the automaton model.

Overview: This paper is divided into two main parts. The first part introduces statecharts and gives their syntax. The second part actually goes into specifying the semantics of such statecharts.

2 Syntax

In this section we present the syntax of the subset of UML statecharts we are considering. We give an informal and a formal description of the syntax.

2.1 Informal Syntax

A statechart is a hiearchical state machine associated to a UML object (a class instance) modeling its behaviour. The statechart of a particular object communicates with other objects and its environmnent via an interface. In this view, we consider our statechart as an open system with the interface describing which events and calls are accepted and sent. We consider such a class to describe the syntax (and semantics in the next section) of the statechart.

An interface is the set of events, operation calls, and primitive calls being accepted by a statechart. The events may be generated by the environment or sent by another object. The operation calls come from other objects (through the environment). And the primitive calls come from the object itself, or some other entity.

A statechart is a hierarchical state machine composed of nested And , Or , and basic states. And states are composed of Or states. If an And state is active, all its sub Or states are active. Or states are composed of (And /Or) states. If an Or state is active, only one of its sub states is active. A basic state is a state with no nested states. It can be later refined as an And or an Or state.

Or states must have special initial states and may have history states. The initial state or “default entry” has only one outgoing edge to the state that shall be entered. The history state is marked with a H for shallow history and H* for deep history. It points to the state to take if the Or state was never taken before.

Edges connect states. An edge has possibly several source and target states. The source and target may be an Or , And , or a basic state. When entering an Or composite state the state border of the Or state may be crossed and the target state may be situated on a different hierarchical level than the source state. The same applies for And states and exits of states. Figure 1 shows these different constructs.

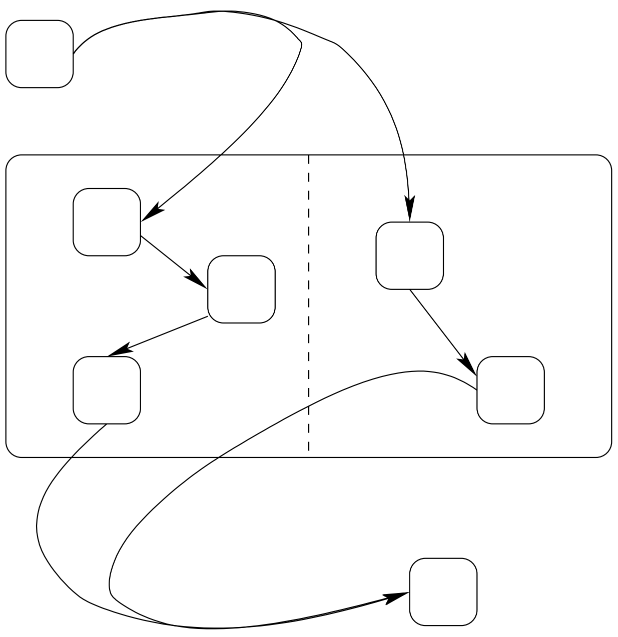


Figure 1: Hierarchical states, fork, and join representations.

Edges can be labeled with event triggering, guard, and action expressions. An event expression is composed of the name of an event and its parameters. A guard expression is a boolean expression. An action expression is a sequence of instructions, calls, event sending.

Variables are the attributes of the object and are global to the statecharts. We make an abstraction on the precise type of these variables and the associated typed value. The types are compiler/machine dependent and we are not concerned with these issues. We will consider that a variable has a particular value, it can be tested and modified by functions and operators.

Primitive operations are non-blocking operations executed immediately. They are specified using programs. These operations belong to the object. Entry and exit actions can be associated to locations. They are executed when a location is entered or left.

2.2 Formal Syntax

2.3 Statechart

A statechart is a tuple $\langle L, L_0, I, \delta \rangle$ where

- $L = \langle S, \lambda, \pi, \nu \rangle$ is a tree structure representing the locations.
  $S \subseteq \mathbb{N}^*$ is the set of nodes in the tree (ϵ denotes the root).
  $\lambda : S \rightarrow \{\textsc{And}, \textsc{Or}, \textsc{ShallowHist}, \textsc{DeepHist}\}$ is a mapping from the set of nodes to labels giving the type of the node.
  ν is a mapping from nodes to sets of variables. $\nu(l)$ stands for the subset of local variables of a particular node l. At the root level $\nu(\epsilon)$ is the set of attributes of the object we are modeling.
  π is the priority mapping: $\pi : S \rightarrow \{\textsc{Normal}, \textsc{Microstep}\}$.

  For $l \in S$ we let $\textsc{And}(l)$ denote that $\lambda(l) = \textsc{And}$. Similarly, we introduce $\textsc{Or}(l)$ (respectively $\textsc{ShallowHist}(l)$) to denote that $\lambda(l) = \textsc{Or}$ (resp. $\lambda(l) = \textsc{Or}$).

  S has a tree structure: if $n.k \in S$ then (1) $\forall j : 0 \leq j < k, n \bullet j \in S$, and (2) $n \in S$.

- L_0 is a partial mapping from locations to boolean expressions. It denotes the potential initial substates for each OR-state. The actual initial state is dynamically chosen by evaluating the boolean expressions. When evaluated, exactly one boolean expression must be true.

- I is the interface: $I = E \cup O \cup P$ where E is the set of events, O the set of operation calls, and P the set of primitive calls. A message is a tuple $\langle n, s, d, t, P \rangle$ where n is the name, s the sender object, d the destination object, t the type $t \in \{\textsc{Event}, \textsc{Opcall}, \textsc{Pricall}, \textsc{Reply}\}$, P the set of parameters. The type $\textsc{Reply}$ is used for the response of a call. We define the predicate $\textsc{Reply}(e) \overset{def}{=} e.t = \textsc{Reply}$ for a message e. Similarly we define the predicates $\textsc{Event}$, $\textsc{Opcall}$, $\textsc{Pricall}$.

- δ is the edge relation: $\delta \subseteq 2^S \times Command \times 2^S$ where $Command \subseteq (\{\varnothing\} \cup \mathbf{timeouts} \cup O \cup E) \times Guard \times Action$. *Guard* is a boolean expression without side effects or calls. *Action* is defined by our action language. *TypeEdge* is a function from edges to $\{MESSAGE, TIMEOUT, \varnothing\}$

- $\mathcal{P}$ is a mapping from the set of primitive operations to programs. A program is a sequence of instructions.

- $\mathcal{E}$ is a mapping from locations to programs. It denotes the entry actions associated to each location.

- $\mathcal{X}$ is a mapping from locations to programs. It denotes the exit actions associated to each location.

We define the notation $l^\bullet$ to denote the set of all descendants of a node l.

$$l^\bullet = \{l.w \in S | w \in \mathbb{N}^*\} \setminus \{l\}$$

We also define the following notations

- $sons(l) = \{l.x \in S | x \in \mathbb{N}\}$

- $parent(l) = k | \exists x \in \mathbb{N} | k.x = l$
- $ancestors(l) = parent(l) \cup ancestors(parent(l))$

Furthermore, each ORlocation l always has one shallow history node connector and one deep history connector. They are noted respectively $\mathcal{H}(l)$ and $\mathcal{H}^*(l)$.

The action language syntax is defined in the following sub-section.

2.4 syntax

We begin by giving an overview of the features of the language:

Action:	Sequential composition
	Variable declaration
	Assignment
	Operation call
	Reply
	Conditional statement
	Event emitting
	Loop

Here are more details about the grammar:

- Variable declaration

  ```
  var VARIABLE NAME;
  var LIST OF VARIABLES;
  ```

  Note that the variables are not typed. We see no obstacle to using typing, as present in C++ or Java, for example. However, developping the full semantics for a complex language would be beyond the scope of this paper.

- Assignment

  ```
  VARIABLE NAME := VALUE;
  (LIST OF VARIABLE NAMES) := (LIST OF VALUES)
  ```

  The second form executes a concurrent assignment. The variables on the left hand side must all be different.
  A `VALUE` is defined as follows:

VALUE :=	OPERATION CALL
	VALUE'
VALUE' :=	(VALUE')
	VARIABLE
	VALUE' OPERATOR VALUE'

We do not specify what OPERATOR is, as it would depend on the type, which we do not handle. Possible operators could be arithmetics operators, string concatenations...
VALUE' is introduced to forbid expressions where several operation calls are performed. Operation calls must be made separately, and stored in temporary variables. This choice is made to simplify the task of giving semantics. However, it is not difficult to allow the user to use complex expressions (involving several operation calls) and provide a compiler that would generate the simple language we describe.

- Operation call

  ```
  A := OBJECT->METHOD(LIST OF PARAMETERS)
  ```

  There are two different kinds of operations:

  - Triggered operations: A request is sent to OBJECT. The request is stored, and OBJECT will handle it as soon as it can, which may be never. It is not possible for an object to call one of its own triggered operation calls. This would put it in a deadlocked situation.
  - Primitive operation calls: These requests cannot block, and are processed immediately.

- Event emitting

  ```
  send(OBJECT, EVENT, LIST OF PARAMETERS)
  ```

  An event is sent to a specific object.

- Reply

  ```
  reply VALUE
  ```

  This intruction stores VALUE as the value to return. The value is actually returned when the next stable state is reached when dealing with a triggered operation call, or at the end of the function for a primitive operation.

- Conditional statement

  ```
  if (VALUE) then THEN else ELSE
  ```

  The interpretation of the if statement is similar to those of C and C++. A non-null value is considered true.

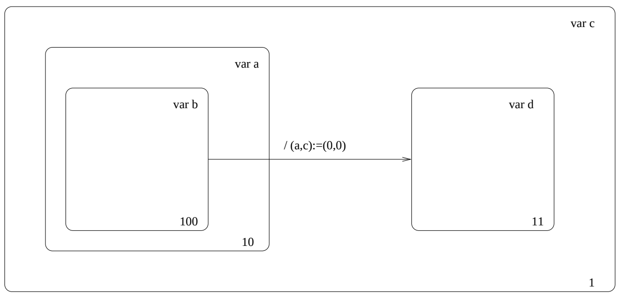
- Loop

  ```
  while (VALUE) BODY
  ```

  BODY is executed as long as VALUE is not zero.

2.5 Scope of variables

Actions on an edge can access all variables seen by the source location(s) and all variables visible from the destination(s). Each location sees all variables of its ancestors. It does not see its own variables.
Another possible choice would have been to allow actions to access variables of the common ancestors of all the destination and source locations. However, this would have prevented us to do the action preprocessing as described in subsection 4.1.
The figure below shows all variables accessible to some edges.



We are now finished with the description of the language. The next paragraphs deal with the semantics of this language.

3 Semantics

In this section we give the formal semantics of UML statecharts. We give first the tuple definition of a configuration, the edge action pre-processing, the run-to-completion, the event dispatching, and the time semantics.

3.1 Configuration

A configuration is a tuple $\langle \sigma, v, q_e, q_o, H, X, D, \mathit{calling} \rangle$ where

- $\sigma \subseteq L$ is the location representation.
- v is the variable valuation: $v : \bigcup_{l \in L} \nu(l) \rightarrow VALUES$. $VALUES$ is the union of all the possible values taken by any datatype.
- q_e is the event queue. The queue is a FIFO containing events.
- q_o is the operation call queue. The queue is a FIFO containing operation calls.
- H is the history mapping. To each Or node is associated the last visited substate. Initially, each history node is associated to $\perp$. It is not valid to enter an history node when its value is $\perp$.

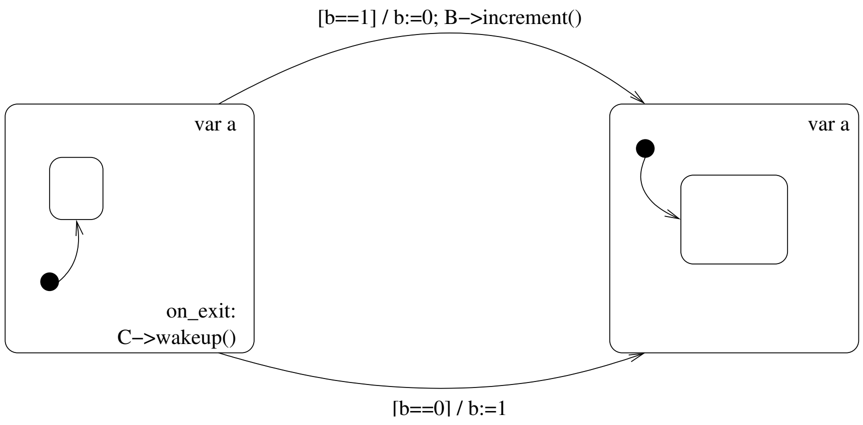
(PB: Another solution: assign a “state expression” as initial value. It corresponds to the initial states, and is evaluated when necessary).

- X is the object’s clock.
- D is the deadline mapping: $D : \delta \rightarrow \mathbb{N} \cup \{+\infty\}$.
- *calling* is the calling flag: *calling* is true whenever an operation call is in progress.

Concerning the clock, we will let time pass in discrete quantities at specified moments. We let time pass in discrete quantities because we define a semantics in the context of real-time systems driven by scheduling cycles. When time passes a cycle ends and a tick occurs.

4 Semantics for the action language

Each edge can carry several actions. They are executed when edges are taken. For example, if b is null, an operation call is performed when leaving the left state, then b is set to 1.



4.1 Semi-statecharts

UML statecharts differ from other commonly used finite state machines in the way that whole programs can be executed on each transition. We chose to transform UML statecharts into simpler statecharts, where each transition can only perform one simple action. We need to split transitions to avoid remaining blocked between two locations. It is possible to be blocked when a triggered operation call is performed.
We define a function translating edge actions to semi-statecharts. Semi-statecharts $\mathcal{S}$ are defined as follows:

$\mathcal{S} = \langle L, \nu, \pi, \delta \rangle$
L is the set of states organised in a tree structure.
ν is the set of variables for each node.

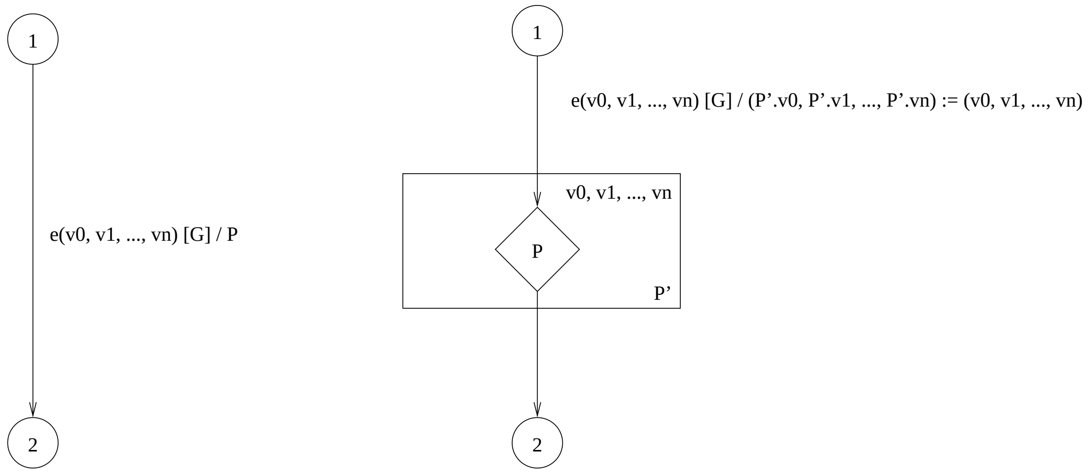
π is the priorities of nodes.

$$\delta \subseteq \left(\begin{array}{ll} & (\{(input, \phi)\} \times BasicActions \times L) \\ \cup & (L \times BasicActions \times L) \\ \cup & (L \times BasicActions \times \{(output, \phi)\}) \end{array} \right)$$

$BasicActions = EmitEvent | Assignment | Reply$

Edges can originate from a special place, *input*, and point to another special place, *output*. Semi-statecharts must be connected to existing places in order to become part of the original statechart. These special places can be seen as connection slots.

The preprocessing step considers every edge in the original UML statechart, and generates egdes and nodes such that each edge is labelled with at most one instruction:

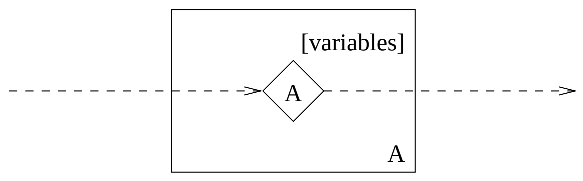

1
e(v0, v1, ..., vn) [G] / P
2
1
e(v0, v1, ..., vn) [G] / (P'.v0, P'.v1, ..., P'.vn) := (v0, v1, ..., vn)
v0, v1, ..., vn
P
P'
2

4.2 Translation from action to semi-statecharts

Below are shown the translations from actions to semi-statecharts, represented graphically. Outgoing dotted arrows are arrows to the special *output* node. Incoming dotted arrows originate from the *input* node. Diamonds are semi-statecharts.

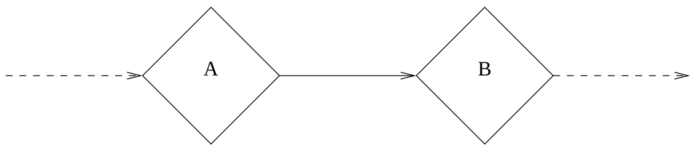
- Blocks

  `{[variables declarations] A}`



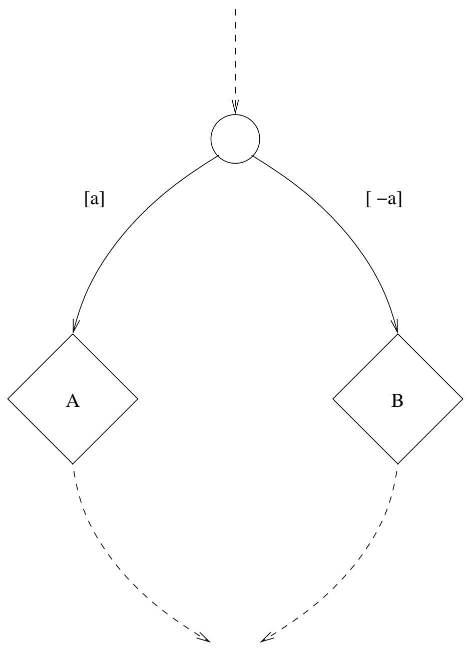
- Sequential compositions

```
A;B
```



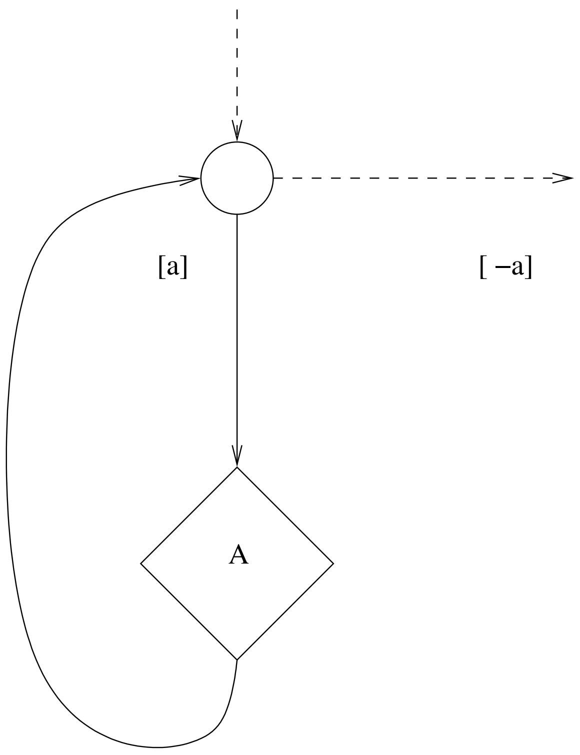
- Conditional statements

```
if (a) then A else B
```



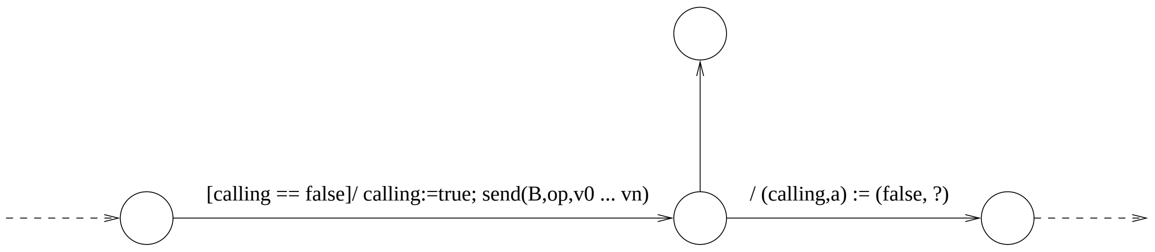
- Loops

```
while (a) A
```



- Triggered Operation Calls

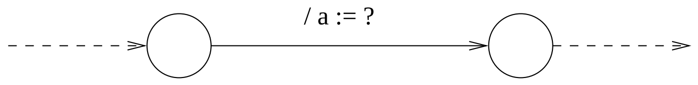
  `a:=B->op(v0, ..., vn)`


[calling == false]/ calling:=true; send(B,op,v0 ... vn)
/ (calling,a) := (false, ?)

  The question mark "?" in this figure indicates that the variable `a` is set to a random value upon reception of the reply. This cannot be helped as the semantics we give is a simulation semantics for a single object in an open system. Therefore, we cannot tell what is the actual value returned by the callee.
  One object can only have one thread of execution. An object waiting for a triggered operation call to complete must hence stop its execution. This is the role of the "calling" flag. Thanks to the choice of interleaving of MICROSTEP edges, there is no race condition on "calling".

- Primitive Operation Calls

```
a:=B->op(v0, ..., vn)
```



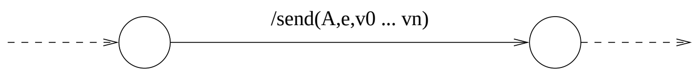
This case is similar to the previous one. We cannot tell what is the actual value returned by the callee, unless the caller called one of its own primitive operations. In that case, the variable `a` is assigned a known value. Other variables may as well be modified, as primitive operations may have side effects. Therefore, assuming v is the valuation of variables before the execution of the call, the valuation v' after execution is:

$$v' = v''_{[a \mapsto p']}$$

With: $\mathcal{P}[\![B]\!](v, (v0, ..., vn)) = (v'', p')$
For an explanation of this notation, see the denotational semantics for primitive operations in 4.4.

- Event emiting

```
send(A, e, v0 ... vn)
```



Primitive operations are not specified as statecharts by the user, rather as programs.

4.3 Primitive operations

Primitive operations have the following properties:

- Their execution must allways terminate.
- The execution is immediate.

Each primitive operation is specified as a program expressed in the action language. It is not preprocessed to generate statecharts. Primitive operations are not allowed to send or receive operation calls or events. Therefore, they can be given semantics using the traditional method: the semantics of a primitive operation is a function from variables valuation to variables valuation.

4.4 Denotational semantics of actions of primitive operations

The language supports arithmetics expressions, boolean expressions, commands, functions calls with side-effects. Three different types of variables exist:

- Attributes. They live as long as their object. Any primitive operation can access them. Hereafter, v denotes the valuation of attributes. $v = \nu(\epsilon)$ (ϵ is the root node).
- Local variables. They are created when entering a primitive operation, and destroyed upon exit. Their scope is limited to their primitive operation. u denotes the valuation of local variables.
- Parameters. They are similar to local variables, but cannot be modified. They can be seen as read-only local variables. They are initialised to some value upon calling, while local variables are not initialised. Their valuation is p.

We need the following following semantic functions:

- $\mathcal{A}[\![a]\!] : \{v\} \times \{u\} \times \{p\} \to$ typeof(A), a being an expression.
- $\mathcal{C}[\![c]\!] : \{v\}\times\{u\}\times\{p\}\times VALUES \to \{v\}\times\{u\}\{p\}\times VALUES$, c being an instruction (an assignement, for example). An instruction may set the return value of a function, hence the need for the $VALUES$ component returned.

$\mathcal{C}[\![]\!]$ is defined as follows:

- Assignment
  $\mathcal{C}[\![\text{a := A}]\!](v, u, p, r) = (v', u', p, r)$
  with $(v', u') = \begin{cases} (v, u_{[a\mapsto \mathcal{A}[\![A]\!](v,u,p)]}) & \text{if a is a local variable} \\ (v_{[a\mapsto \mathcal{A}[\![A]\!](v,u,p)]}, u) & \text{else} \end{cases}$
- Conditional statement

  $$\mathcal{C}[\![\text{if (A) then B else C}]\!]((v, u, p, r)) = \begin{cases} \mathcal{C}[\![B]\!](v, u, p, r) & \text{if } \mathcal{A}[\![A]\!](v, u, p) \\ \mathcal{C}[\![C]\!](v, u, p, r) & \text{else} \end{cases}$$

- Loops
  $\mathcal{C}[\![\text{while (A) B}]\!] = fix(\Gamma)$
  where $fix(\Gamma)$ is the Least Fixed Point of the function Γ defined as follows:
  $\Gamma(\phi)(v, u, p, r) = \begin{cases} (v, u, p, r) & \text{if } \neg\mathcal{A}[\![A]\!](v, u, p) \\ \phi \circ \mathcal{C}[\![B]\!](v, u, p, r) & \text{else} \end{cases}$
- Primitive operation calls (to self)
  $\mathcal{C}[\![\text{a := func(b,c,d)}]\!](v, u, p, r) = (v', u', p, R)$
  with:
  - $\mathcal{C}[\![\text{body(func)}]\!](v, u_0, p_0, R) = (v'', u'_0, p_0, R')$

– $(v', u') = \begin{cases} (v'', u_{[a \mapsto R']}) & \text{if a is a local variable} \\ (v''_{[a \mapsto R']}, u) & \text{else} \end{cases}$

u_0 denotes the valuation of the variables local to func.
Note: this definition is valid only if no recursive call may happen. In the case of recursive calls, the definition should be expressed using fixed points, as done in the while-loop case.

- Primitive operation calls (to someone else)
  $\mathcal{C}[\![\text{B-¿func(a,b,c)}]\!](v, u, p, r) = (v', u', p, r)$ There is absolutely no constraint on v' and u'. As the other object may call some of our own primitive operations, the valuation of variables may change in an unpredictable way. Similarly, we cannot predict the return value of the primitive operation call to the other object.

- Replies
  $\mathcal{C}[\![\text{reply(A)}]\!](v, u, p, r) = (v, u, p, \mathcal{A}[\![A]\!](v, u, p))$

- Sequential composition
  $\mathcal{C}[\![\text{A;B}]\!](v, u, p, r) = \mathcal{C}[\![A]\!] \circ \mathcal{C}[\![B]\!](v, u, p, r)$

- The rest is left unspecified, as it is obvious.

5 Stable States and Run-to-completion

5.1 Stable States

Intuitively, a *stable state* is a configuration of an object where no further transition is possible without dispatching an event or operation call.

More formally, let us define the following function:

$$\mathit{enabledSet}(\sigma) = \{e \in \delta | e = \langle l_1, \mathit{cmd}, l_2 \rangle \Rightarrow l_1 \in \sigma \wedge \mathit{cmd.guard} = \mathit{true}\}$$

Then a configuration $c = \langle \sigma, v, q_e, q_o, H, X, D, \mathit{calling} \rangle$ is stable if and only if $\mathit{enabledSet}(\sigma) = \varnothing \wedge \neg \mathit{calling}$.

In the rest of the document, we will use the notation $c \not\rightarrow$ to denote that c is stable.

5.2 Run-to-completion

A *run-to-completion* is a sequence of transitions between two stable configurations containing exactly two stable configutaions. Intuitively, this means that once an operation call or event has been dispatched, the system evolves on its own until no more transitions can be taken. Then, the dispatcher has to be called once again. Note that at the end of each run-to-completion, we make time pass by a fixed amount of time (cf. section 6.5 for details) and send replies to triggered operation calls.

6 The Dispatcher

The role of the dispatcher is to find an operation call or event that can be accepted in the current configuration of the object.

We remember that operation calls are given to the object by the environment asynchronously. When such an operation call arrives, it needs to be stored (cf. the section about the environment: 7).

The dispatcher works on two separate First In First Out (FIFO) queues dedicated to events and operation calls respectively.

In the dispatching process, we give a higher priority to operation calls than to events. I.e. we try to dispatch an operation call, and if it's not possible, then we try to dispatch an event.

6.1 Dispatching Operation Calls

As described earlier, the basis for dispatching operation calls is a FIFO queue. When trying to dispatch an operation call, we consider operation calls in the queue in the order they were put in (from the oldest to the youngest). Operation calls that cannot be served immediately are kept in the queue for later processing. Indeed, operation calls may not be discarded, even if they cannot be served at a given time. Our method ensures that calls are served in the order they appear whenever possible. And delayed calls naturally get deblocked as soon as possible.

6.2 Dispatching Events

Basically, we treat events in the same way as operation calls. The only difference is that we can distinguish beween events that can be discarded, and those that should be kept for later consideration. Two policies are possible here: Either we consider that some events are important and must never be discarded, or we consider that the relevance of an event depends on the state of the statechart. The first option leads to discard or keep events on a per-event basis, whether the second one leads to take this decision depending on the location. Each location would then have a "deferrable events" set. We chose the first choice, but it seems it may not be the best choice. We may change this point later.

6.3 Dispatch Function

Let the object be in a configuration $c = \langle \sigma, v, q_e, q_o, H, X, D, \mathit{calling} \rangle$. We define the function $\mathit{dispatch}(\sigma, q_e, q_o)$, which returns the event or operation call to be dispatched in the current configuration, as follows:

$$\mathit{dispatch}(\sigma, q_e, q_o) =$$

$$\begin{cases} op & \text{if } head(q_o) = op \text{ and } canAccept(\sigma, op), \\ dispatch(\sigma, q_e, next(q_o)) & \text{if } head(q_o) = op \text{ and } \neg canAccept(\sigma, op), \\ e & \text{if } q_o \text{ is empty and } head(q_e) = e \text{ and} \\ & canAccept(\sigma, e), \\ dispatch(\sigma, next(q_e), q_o) & \text{if } q_o \text{ is empty and } head(q_e) = e \text{ and} \\ & \neg canAccept(\sigma, e), \\ nil & \text{if } q_e \text{ and } q_o \text{ are empty.} \end{cases}$$

Note that the *dispatch*() function does not modify the event and operation call queues. We define the following functions, which return respectively, the new event queue, and the new operation calls queue:

$$newEventQueue(q_e) =$$

$$\begin{cases} newEventQueue(next(q_e)) & \text{if } head(q_e) \neq dispatch() \text{ and} \\ & head(q_e)\text{is not deferrable,} \\ head(q_e) \bullet newEventQueue(next(q_e)) & \text{if } head(q_e) \neq dispatch() \text{ and} \\ & head(q_e)\text{is deferrable,} \\ next(q_e) & \text{if} head(q_e) = dispatch(). \end{cases}$$

$$newOpCallQueue(q_o) =$$

$$\begin{cases} head(q_o) \bullet newOpCallQueue(next(q_o)) & \text{if} head(q_o) \neq dispatch(), \\ next(q_o) & \text{if} head(q_o) = dispatch(). \end{cases}$$

6.4 Transitions

When a transition is taken (between two configurations) it may involve several edges. Edges after the preprocessing stage have one simple action exactly (*skip* used for empty action). We need to define when an edge t is enabled, identify its source and destination states, and define the transition relation itself.

An edge t is enabled in a configuration c, noted *isEnabled*(c, t), if all of the propositions below are true:

(1) its guard expression is true

(2) either (i) t is triggered by the right event if any, or (ii) a timeout has occured

(3) there is no higher priority edge enabled that could preempt t

(4) the state priority allows t to be taken

(5) all of its source states are active

ThThe guard expression g of the edge t (omitted for brevity) is evaluated over the current valuation v of the variables. $g(v)$ denotes its truth value.

An edge is said triggered if it is triggered by an event or the object clock is greater or equal to the transition deadline. Event dispatching is treated in subsection 6. The event trigger may be in a special case the response of a call. An event trigger is taken into account only at the beginning of a run-to-completion step. A run-to-completion step always and only begins from a stable state. For a configuration c, the triggered predicate of an edge t with e its event trigger is

$$\mathit{triggered}(t) \overset{\mathit{def}}{=} \begin{cases} (c \not\rightarrow \wedge dispatch(c) = e) & \text{if } \mathit{TypeEdge}(t) = \text{MESSAGE} \\ (X \geq D(t)) & \text{if } \mathit{TypeEdge}(t) = \text{TIMEOUT} \end{cases}$$

An edge t' has higher priority than an edge t if one of its sources is located in a substate of a source of t. Formally, we define the priority partial order over edges: t' has higher or equal priority than t, noted $t' \geq t$ iff $\exists l' \in \mathit{source}(t') \exists l \in \mathit{source}(t) | l' \in l^{\bullet} \wedge \forall l \in \mathit{source}(t) l \notin l'^{\bullet}$. If neither $t' \geq t$ or $t \geq t'$ then the edges are not comparable. We define the strict partial order $t' > t \overset{\mathit{def}}{=} t' \geq t \wedge \neg t \geq t'$.

The state priority allows the edge to be taken if there is no higher priority active state. This means higher priority states have to be left first. The order of priority is: *NORMAL* $<$ *MICROSTEP*. The edge is allowed, noted $\mathit{allowed}(t)$, iff

$$max_{l \in source(t) \wedge l \in \sigma} \pi(l) >= max_{l' \notin source(t) \wedge l' \in \sigma} \pi(l')$$

The edge t must have all of its source states active is defined by $\mathit{active}(t) = \forall l \in \mathit{source}(t) l \in \sigma$.

We can now give the expression to evaluate if an edge is enabled in a given configuration $\langle \sigma, v, q_e, q_o, H, X, D, \mathit{calling} \rangle$:

$$\mathit{isEnabled}(t) \overset{\mathit{def}}{=} g(v) \wedge \mathit{triggered}(t) \wedge \mathit{allowed}(t) \wedge \mathit{active}(t) \wedge \forall t' \neq t \neg (t' > t \wedge \mathit{isEnabled}(t'))$$

This definition is recursive and valid since at every evaluation the evaluation is done at a deeper hierarchy (from the $t' > t$). The hierarchy depth being finite the evaluation is finite and converges.

Taking a transition may involve several edges though these edges are not conflicting ones. Two edges t and t' are in conflict iff $\mathit{source}(t) \cap \mathit{source}(t') \neq \varnothing$. If several edges are in conflict only one of them is choosen. The edges taken are those of a maximal enabled edges set $S_k = \{t_1, \ldots, t_k\}$ with no conflict and normal priority, or a singleton of one of the highest priority edges.

The transition relation is then for $c_1 = \langle \sigma, v, q_e, q_o, H, X, D, \mathit{calling} \rangle$ and $c_2 = \langle \sigma', v', q'_e, q'_o, H', X', D', R', \mathit{calling}' \rangle$: $c_1 \xrightarrow{S_k} c_2$ with:

- $\sigma' = \sigma \setminus \bigcup_{1 \leq i \leq k} \mathit{exits}(\mathit{source}(t_i)) \cup \bigcup_{1 \leq i \leq k} \mathit{entries}(\mathit{dest}(t_i))$.
- $v' = a_{\alpha(1)} \circ \cdots \circ a_{\alpha(k)}(v)$ where α is one permutation (any one), and $a_i(v)$ the result of the action of the edge t_i on the valuation v.

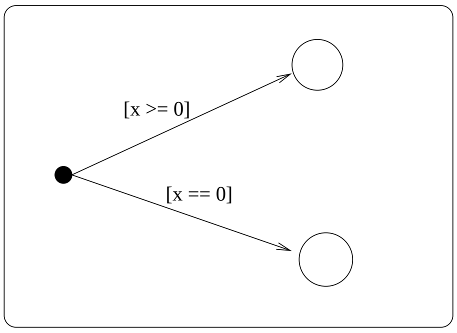
- $q'_e = newEventQueue(q_e)$ if $c_1 \not\rightarrow$ else $q'_e = q_e$.
- $q'_o = newOpCallQueue(q_o)$ if $c_1 \not\rightarrow$ else $q'_o = q_o$.
- H' is the history update: $H' := H$ with the update $\forall l | OR(l) \wedge \sigma(l) l : H'(l) = k | k \in sons(l) \wedge \sigma(k)$.
- $X' = X + 1$ if $c_2 \not\rightarrow$ else $X' = X$.
- $D' = D$ with the update $\forall t \in S_k : \forall t' | source(t') \cap dest(t) \neq \varnothing : D(t') = X + tm(t') + 1$. where $tm(t')$ is the timeout value of the transition t'. The $+1$ term is needed because of the delay transition that takes place at the end of the run-to-completion step. The other transitions are not tested, so we do not need to update them.

The update of σ uses the functions *entries* and *exits*. These functions compute the right set of locations to enter and to leave. The common parent state of the sources and destinations of a transition t is $P = prefix(source(t) \cup dest(t))$ where *prefix* is the greatest common (string) prefix of the locations. The locations to be exited for this transition t are defined by $exits(t) \overset{def}{=} P^{\bullet}$.

Concerning the entries, we need to enter all the substates of an AND state, we need to follow the init and history states if OR states are entered. We define the *entries*(t) function of a transition t as follows: $entries(t) = \{l | isEntered(c, l)\}$
Here is the definition of the *isEntered*:

$$isEntered(c, l) \overset{def}{=} \begin{array}{ll} & \bullet \text{ if } useHistory(parent(l)) \\ & H(parent(l)) = l \\ & \bullet \text{ else} \\ & \exists t \in S_k | l \in dest(t) \\ \vee & (isEntered(c, parent(l)) \wedge AND(parent(l))) \\ \vee & (isEntered(c, parent(l)) \wedge OR(parent(l)) \wedge init(c, l)) \end{array}$$

with $init(c, l)$ being a predicate true if the initial state of the parent of l points to l and if its guard is true in the configuration c. Note that the initial state of $parent(l)$ must be unique. The following case for example is not correct:



The predicate *useHistory* is defined below:

$$useHistory(c,l) \overset{def}{=} \begin{array}{ll} & \exists t \in S_k | \mathcal{H}(l) \in dest(t) \\ \vee & \exists t \in S_k | \exists a \in ancestors(l) | \mathcal{H}^*(l) \in dest(t) \end{array}$$

6.5 Time

To keep the fact that a run-to-completion between two stable configurations corresponds to a scheduling cycle, time passes with one unit after each run-to-completion. This is done in the transition rule, but if no transition is enabled we have to perform an idle transition to let time pass to enable timeouts. The idle transition is defined by: if $c \not\to \wedge \forall t \in \delta \neg isEnabled(t)$ then $c \to \langle \sigma, v, q_e, q_o, H, X+1, D, R, false \rangle$.

7 Environment

7.1 General

We recall that the definitions given previously only define individual objects. If we only provide a semantics to those individual objects, we get an open system which cannot be simulated *per se*, since there is no way to express the fact that some events or operation calls should be passed on to their respective destination object by the environment. This is the case for example if one models a machine with a button. To be able to make a simulation, one must consider a user that will provide an input to the machine.

The semantics we give in the next sections will consider this *environment* to be a *black box.*

7.2 Transition Rules

The environment can give operation calls or events to an object at any point in time. So the following rules are actually valid at all times.

Let $c1 = \langle \sigma, v, q_e, q_o, H, X, D, calling \rangle 1$ and $c2 = \langle \sigma, v, q_e, q_o, H, X, D, calling \rangle 2$ be two configurations of our object. Then the following transitions are valid:

1. $c1 \to c2$ iff for some event $e \in E$ we have

$$\sigma^1 = \sigma^2 \wedge v^1 = v^2 \wedge q_e^1 \bullet e = q_e^2 \wedge q_o^1 = q_o^2 \wedge H^1 = H^2$$

2. $c1 \to c2$ iff for some operation call $op \in O$ we have

$$\sigma^1 = \sigma^2 \wedge v^1 = v^2 \wedge q_e^1 = q_e^2 \wedge q_o^1 \bullet op = q_o^2 \wedge H^1 = H^2$$

8 Issues

In this work we had to choose a number of compromises concerning the action and statecharts semantics. In particular a number of issues remain. The model does not allow time to pass when the system is locked waiting for a reply. The environment is still loosely specified and it is supposed to behave relatively well with the system interface. We support only one object running, thus describing only one statecharts running. Concurrency between different statecharts, time uniformity, and run-to-completion definition between several statecharts are some of the major issues. We do not support in this work entry and exit actions for forks and joins.

9 Conclusion and future work

The semantics presented in this paper aim at simulating an object described with an UML statechart. They do not intend to be used for automated verification. It would certainly be useful to integrate certain aspects we ignored in order to create a verification semantics. For exemple, all attributes and operations are public in our model. However, knowing restrictions on the access to attributes or methods could be used to prove certain properties. Similarly, the knowledge that a primitive operation cannot modify its owning object can be important. Therefore, handling a const qualifyer, in the way C++ does, could be interesting.

As a future work, it would be better to define deferrable events on a per-state basis instead of per-statechart. The description would be more fine-grained. Furthermore the issues could be addressed, in particular the concurrency of statecharts. This would give a better description to generate code for concurrent systems.