Modeling the Dynamics of UML State Machines

Egon Börger[1], Alessandra Cavarra[2], and Elvinia Riccobene[2]

[1] Dipartimento di Informatica - Università di Pisa - C.so Italia, 40 - 50125 Pisa
boerger@di.unipi.it (currently visiting Microsoft Research, Redmond)
[2] Dipartimento di Matematica e Informatica - Università di Catania -
V.le A. Doria, 6 - 95125 Catania
{cavarra, riccobene}@dmi.unict.it

Abstract. We define the dynamic semantics of UML State Machines which integrate statecharts with the UML object model. The use of ASMs allows us (a) to rigorously model the event driven *run to completion* scheme, including the sequential execution of entry/exit actions (along the structure of state nesting) and the concurrent execution of internal activities; (b) to formalize the object interaction, by combining control and data flow features in a seamless way; and (c) to provide a precise but nevertheless provably most general computational meaning to the UML terms of atomic and durative actions/activities. We borrow some features from the rigorous description of UML Activity Diagrams by ASMs in [7].

1 Introduction

The Unified Modeling Language [2, 5, 20] is a standardized notation based on a set of diagrams to describe the structure and the behavior of a software system. In [5] it is stated that "UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and *semantics* of that building block" although the official document [4] for the UML semantics only gives an unambiguous textual definition of the syntax for UML notations and leaves the behavioral content of various UML constructs largely open. The necessity to develop the UML as a precise (i.e. well defined) modeling language is widely felt [10, 9, 19] and the *pUML* (*precise UML*) group has been created to achieve this goal [18].

In this paper we analyze one of the principal diagram types which are used in UML for the description of dynamical system behavior, namely statechart or state diagrams, and provide a rigorous definition of their dynamics. Many papers on the semantics of statecharts [16, 21, 11, 17] exist in the literature, in particular in relation to their implementation in STATEMATE [15] and RHAPSODY [14]. Nevertheless, the debate is still ongoing on what exactly should be considered as the authoritative definition of UML State Machines which integrate statecharts with the UML object model. One major difficulty here concerns the mechanisms for object interaction [14, 19].

ASMs [1, 12] provide a technique to solve such specification problems and to clarify the relevant issues. In this paper, we propose an ASM model that (a) rigorously defines the UML event handling scheme in a way which makes all its "semantic variation points" explicit, including the event deferring and the event completion mechanism; (b) encapsulates the run to completion step in *two* simple rules (**Transition Selection** and **Generate Completion Events**) where the peculiarities relative to entry/exit or transition actions and sequential, concurrent or history states are dealt with in a modular way; (c) integrates smoothly the state machine control structure with the data flow; (d) clarifies various difficulties concerning the scheduling scheme for internal ongoing (really concurrent) activities; (e) describes all the UML state machine features that break the thread-of-control; (f) provides a precise computational content to the UML terms of atomic and durative actions/activities, without loosing the intended generality of these concepts (see footnote 10), and allows one to clarify some dark but semantically relevant points in the UML documents on state machines.

We do not take any position on which UML concepts or understandings of them are reasonable or desirable. Through our definitions we however build a framework for rigorous description and analysis of logically consistent interpretations of the intuitions which underly UML concepts. In fact, exploiting the abstract nature of ASMs it is easy to adapt our definitions to changing requirements. We hope that this will contribute to their rational reconstruction, for the standardization, and to the comparison of different implementations. Our model can also serve as reference model for implementing tools for code generation, simulation and verification of UML models. This work can be viewed as a continuation of [7] where a rigorous semantics of UML activity diagrams has been provided.

The paper is organized as follows. Section 2 introduces the basic concepts underlying UML statechart diagrams and ASMs. The ASM model for the behavioral meaning of these diagrams is defined in section 3. In section 4, the semantical equivalence between some state machine building blocks is discussed. In section 5 we compare our model with related work and show that it satisfies the UML meta-model requirements for state machines.

2 Basic concepts

In this section we sketch the basic concepts underlying UML state machines and ASMs and review the notation.

2.1 UML statechart diagrams

Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. Statecharts were invented by David Harel [15, 16], the semantics and the notation of UML statecharts are substantially those of Harel's statecharts with adaptations to the object-oriented context [3].

Statechart diagrams focus on the event-ordered behavior of an object, a feature which is specially useful in modeling reactive systems. A statechart diagram shows the event triggered flow of control due to transitions which lead from state to state, i.e. it describes the possible sequences of states and actions through which a model element can go during its lifetime as a result of reacting to discrete events. A state reflects a situation in the life of an object during which this object satisfies some condition, performs some action, or waits for some event. According to the UML meta-model [4], states can belong to one of the following categories: *simple states*, *composite states* (sequential, concurrent, submachine), *final*, and *pseudostates* (initial, history, stub, junction, synch).

Transitions are viewed in UML as relationships between two states indicating that an object in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain conditions are satisfied [3]. UML statecharts include *internal*, *external* and *completion* transitions.

The semantics of event processing in UML state machines is based on the *run to completion* (rtc) assumption: events are processed one at a time and when the machine is in a stable configuration, i.e. a new event is processed only when all the consequences of the previous event have been exhausted. Therefore, an event is never processed when the state machine is in some intermediate, unstable situation.

Events may be specified by a state as being possibly deferred. They are actually deferred if, when occurring, they do not trigger any transition. This will last until a state is reached where they are no more deferred or where they trigger a transition.

2.2 Abstract State Machines

ASMs are transition systems, their states are multi-sorted first-order structures, i.e. sets with relations and functions, where for technical convenience relations are considered as characteristic boolean-valued functions. The transition relation is specified by rules describing the modification of the functions from one state to the next, namely in the form of guarded updates ("rules")

$$\textbf{if } \textit{Condition} \textbf{ then } \textit{Updates}$$

where *Updates* is a set of function updates $f(t_1, \ldots, t_n) := t$, which are simultaneously executed when *Condition* is true.

We use *multi-agent ASMs* [12, 1] to model the concurrent substates and the internal activities which may appear in a UML statechart diagram. A multi-agent ASM is given by a set of (sequential) agents, each executing a program consisting of ASM rules. Their distributed runs are defined in [12].

Since ASMs offer the most general notion of state, namely structures of arbitrary data and operations which can be tailored to any desired level of abstraction, this allows us on the one side to reflect in a simple and coherent way the integration of control and data structures, resulting from mapping statecharts to the UML object model. In fact, machine transitions are described by ASM rules where the actions become updates of data (function values for given arguments). On the other side also the interaction between objects is naturally reflected by the notion of state of multi-agent (distributed) ASMs.

For the constructs of sequentialization, iteration and submachine of sequential ASMs we use the definitions which have been given in [8]. They provide the concept of "stable" state needed to guarantee that the event triggered sequential exit from and entry into nested diagrams is not interrupted by a too early occurrence of a next event.

3 ASM model for UML statechart diagrams

In this section we model the event governed *run to completion step* in statechart diagrams. We first introduce the signature of UML statecharts and then define the execution rules.

Statechart diagrams are made up of (control) states[1] and transitions, belonging to the abstract sets *STATE* and *TRANSITION*.

3.1 Control State

The set *STATE* is partitioned into simple, composite (with substructure), and pseudo states. Composite states are partitioned into sequential and concurrent ones. Pseudostates are partitioned into initial and history states.
Simple states (1) are of the form $state(entry, exit, do(A), defer)$[2], where the parameters *entry*/*exit* denote actions that have to be performed as soon as the state is entered/exited, $do(A)$ denotes the internal ongoing activity A that must be executed as long as the state is active, and *defer* is a set of events that are candidate to be retained in that state.
Sequential composite states (2) are of the form $state(entry, exit, do(A), defer, init, final, history)$, where *entry*/*exit*, $do(A)$ and *defer* have the same meaning as for simple states, *init* denotes the initial state of the submachine state associated to this composite state, *final* denotes its final state, and *history* its associated history state (see below for details on initial, final and history states). Sequential composite states contain one or more substates, exactly one of which is required to be active when the composite state is active.
Concurrent composite states (3) are of the form $state(entry, exit, do(A), defer, concurrentComp)$, where *entry*/*exit*, $do(A)$ and *defer* are as above, and *concurrentComp* yields the set of the concurrent (sequential) substates[3] composing the state. When a concurrent state is active, all of its subcomponents are active.

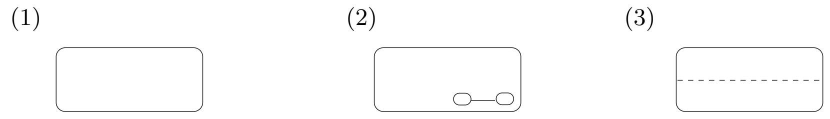
According to [4], an event that is deferred in a composite state is automatically deferred in all its directly or transitively nested substates. For reasons of

[1] This notion of control state, deriving from the finite state machine notion of "internal" state, is only a tiny fraction of the overall system state which is reflected by the ASM notion of state as structure, i.e. domains (of to be instantiated objects) with operations and relations.

[2] Simple and composite states may have a *name*, i.e. a string denoting uniquely the state. We omit the name parameter in the signature of such states as it is not relevant for our model.

[3] Each substate is sequential because it must enclose an initial and a final state.

simplicity, but without loss of generality, we assume that the defer set of each state explicitly contains all the inherited events to be deferred.



Initial states • indicate where to start by default when the enclosing (composite sequential) state is invoked. A **History state**, associated to a sequential composite state say S, is a pseudostate that can be of two types: *shallow history* Ⓗ and *deep history* Ⓗ*. The shallow history state records, upon exiting S, only the most recent active state directly contained in S and restores the recorded state when the history state is invoked. The deep history state records the most recent active hierarchical configuration of S, and restores this configuration when the history state is invoked. To keep track of the configuration, we use a dynamic function

$$memory : STATE \longrightarrow STATE^*$$

that is initialized to the empty sequence for each state which has never been accessed. To guarantee the correct entering order, we handle *memory* as a LIFO list. In case of shallow history state *memory* contains at most one state.
Final states ⦿ are special states whose activation indicates that the enclosing state is complete.

We denote by *SimpleState, SequentialState, ConcurrentState, PseudoState, FinalState* the characteristic functions of the corresponding subsets of *STATE*.

Any state which is enclosed within a composite state is called a *substate* of that state. In particular, it is called *direct substate* when it is not contained in any other state; otherwise it is referred to as a *transitively nested substate*. The nesting structure of statechart diagrams is encoded by the following static functions:

- *UpState*: $STATE \longrightarrow STATE \cup \{undef\}$, such that $UpState(s) = t$ iff s is a direct substate of a compound state t.
- *DownState*: $STATE \times STATE \longrightarrow BOOL$, such that $DownState(t, s) = true$ iff s is direct substate of a compound state t.
- $UpChain : STATE \times STATE \longrightarrow STATE^*$,
  $UpChain(s,t) = [S_1, \ldots, S_n]$ where $n > 1$ &
  $$S_1 = s \ \&\ S_n = t \ \&\ \forall i = 2 \ldots n,\ UpState(S_{i-1}) = S_i$$
- $DownChain : STATE \times STATE \longrightarrow STATE^*$,
  $DownChain(s,t) = [S_1, \ldots, S_n]$ where $n > 1$ &
  $$S_1 = s \ \&\ S_n = t \ \&\ \forall i = 1 \ldots n-1,\ DownState(S_i, S_{i+1})$$

  Upchain and *DownChain* yield empty sequences on each pair of not nested states. We write $Up/DownChain(s_1, \widehat{s_2})$ to indicate the right open sequence $Up/DownChain(s_1, \widehat{s_2}) = [T_1, \ldots, T_n[$, if it exists. Notice that $Up/DownChain(s, \widehat{s}) = []$.

3.2 Transitions

The set *TRANSITION* is partitioned into internal and external transitions.
External transitions are of form *trans*(*source,target,event,guard,action*), where *source*/*target* represent the source/target states of the transition, *event* denotes the triggering event which may enable the transition to fire, *guard* is a boolean expression that is evaluated as soon as the event occurs (if it evaluates to false the transition does not fire), *action* is an action that is executed at the time the transition fires.
Internal transitions are of the form *trans*(*source,event,guard,action*), where all the parameters have the same meaning as for the external transitions. Internal transitions have a source state but no target state because the active state does not change when they fire, and no exit or entry actions are executed. We distinguish between external and internal transitions using a predicate *internal* on *TRANSITION*.
Statechart diagrams include also **completion transitions**, namely transitions with an implicit "completion event" indicating the completion of the state the transition leaves. We can handle completion transitions as special trigger transitions, labeled by *completionEvent*(S), where S is the source state, and assume that all transitions in a statechart diagram are labeled with an event. The only transitions outgoing pseudostates are completion transitions [20].

For each type of state and transition parameter, we use a (static) function *param* which applied to the related states or transitions yields the corresponding parameter. For example *entry*(*state*) yields the entry action associated to *state*, *source*(*trans*) the source state of the transition *trans*, etc. We often suppress parameters notationally.

3.3 Agents

Let $AGENT$ be the set of agents which move through the statechart diagram, each executing what is required for its *currently active state*. A state becomes active when it is entered as result of some transition, and becomes inactive if it is exited as result of a transition. "When dealing with composite and concurrent states, the simple term current state can be quite confusing. In a hierarchical state machine more then one state can be active at once. If the control is on a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active" [4]. Therefore, to maintain what in UML is called the current configuration of active states, we introduce a dynamic function

$$\mathit{currState} : AGENT \to \mathcal{P}(STATE)$$

whose updates follow the control flow of the given statechart diagram. The function $\mathit{deepest} : AGENT \longrightarrow STATE$ yields the last (innermost) state reached by an agent.

The agents execute UML statechart diagrams, i.e. they all use the same program (or ASM *Rule*). As a consequence, in the formulation of these rules below, we use the 0-ary function *Self* which is interpreted by each agent a as a.

When a new agent is created to perform a concurrent subcomputation (defined by one of the substates in a concurrent composite state), it is linked to the *parent* agent by the dynamic function

$$parent : AGENT \to AGENT \cup \{undef\}$$

We assume that this function yields *undef* for the main agent who is not part of any concurrent flow. The active subagents of an agent a are collected in the set $SubAgent(a) = \{a' \in AGENT \mid parent(a') = a\}$

At the beginning of the computation, we require that there is a unique agent, positioned on the initial state of the *top state*, and whose program consists of the rules **Transition Selection** and **Generate Completion Event** described below.

3.4 Event handling

In UML it is assumed that a state machine processes one event at a time and finishes all the consequences of that event before processing another event [5, 20]. "An event is *received* when it is placed on the event queue of its target. An event is *dispatched* when it is dequeued from the event queue and delivered to the state machine for processing. At this point, it is referred as the *current event.* Finally, it is *consumed* when event processing is complete. A consumed event is no longer available for processing" [4].

We therefore assume that one event is processed at a time. Since the particular event enqueuing and dispatching mechanisms are deliberately not furthermore specified in UML, we model them here explicitly as semantic variation points and therefore use a monitored predicate *dispatched* indicating which event is dequeued to be processed. At any moment, the only transitions that are eligible to fire when an event e occurs are the ones departing from an active state (i.e. whose source state belongs to *currState*) whose associated guard evaluates to true[4]. This is expressed by the following condition

$$enabled(t,e) \equiv event(t) = e \ \& \ guard(t) \ \& \ source(t) \in currState$$

It is possible for more than one transition to be enabled by the same event, but UML allows only those transitions to be fired simultaneously which occur in concurrent substates [4]. In all the other cases, the enabled transitions are said to be in *conflict* with each other. One can distinguish three types of conflict situations: (1) an internal transition in an active state conflicts with a transition outgoing from that state, (2) two or more transitions originating from the same source in an active state are enabled by e, and (3) two or more transitions with different source states but belonging to the same active state are enabled by the occurrence of e. In UML the selection among conflicting transitions is constrained only for case (3) by giving priority to the innermost enabled transition. We now formalize this priority for (3), whereas in the cases (1) and (2) we reflect the choice between different scheduling mechanisms as a semantic variation point, namely by using abstract selection functions; see the **Transition Selection** rule below.

[4] If no guard is associated to a transition t, we assume $guard(t) = true$.

Let $\mathit{enabled}(e) = \{t \in \mathit{TRANSITION} \mid \mathit{enabled}(t, e)\}$ be the set of all transitions enabled by e. We define an equivalence relation $\sim$ on $\mathit{enabled}(e)$ as follows: $\forall\, t_1,\ t_2\ \in \mathit{enabled}(e),\ t_1 \sim t_2$ iff $\ \mathit{source}(t_1) = \mathit{source}(t_2)$.
The nesting of states induces the total order relation[5] $\leq$ on the quotient set $\mathit{enabled}(e)/\sim$, defined as $[t_1] \leq [t_2]$ iff $\mathit{source}(t_1)$ is a direct or a transitively nested substate of $\mathit{source}(t_2)$.
Let $\mathit{FirableTrans}(e)$ be the minimum equivalence class in $\mathit{enabled}(e)/\sim$. It reflects the UML requirement that among transitions enabled by the same event and with different source states, priority is given to an innermost one. The choice among those innermost ones is left open as semantic variation point (see the **choose** construct in the Transition Selection rule).

If a dispatched event does not trigger any transition in the current state, it is lost unless it occurs in the deferred set of the deepest active state. This is formalized by the following predicate *deferrable* on *EVENT*:

$$\mathit{deferrable}(e) = \mathit{true} \Leftrightarrow \mathit{enabled}(e) = \emptyset\ \&\ e \in \mathit{defer}(\mathit{deepest})$$

As suggested in [20], to store deferred events we associate to each agent a list[6] of events *deferQueue* that is dynamically updated during the computation (see rule Transition Selection). We can therefore define $\mathit{deferred}(e)$ to mean $e \in \mathit{deferQueue}$.

We call a deferred event *releasable* when it becomes ready to be consumed, i.e. when it can trigger a transition in the current state configuration

$$\mathit{releasable}(e) = \mathit{true} \Leftrightarrow \mathit{deferred}(e)\ \&\ \mathit{enabled}(e) \neq \emptyset$$

3.5 Statechart diagram main rules

In this subsection we define the ASM rules for the execution of statecharts, i.e. we specify the sequences of states that an object goes through, and of the actions it takes, in response to events which occur during its lifetime [20].

Apparently, UML leaves it unspecified how to choose between dispatched and releasable events. We reflect this by using a selection function which, at any moment, chooses either a dispatched event triggering a transition, or an event that has been deferred. A dispatched event, if *deferrable*, has to be inserted into the *deferQueue*. A releasable event, when chosen for execution, has to be deleted from *deferQueue*[7]. This implies that when choosing an event which is

[5] Observe that $\leq$ is total since all the source states of the transitions in *enabled* belong to *currState* and therefore they are nested.

[6] Apparently, this list is meant to be a set, leaving the exact ordering of elements open as a semantic variation point. A similar remark applies also to other lists occurring in the UML texts.

[7] If upon execution of transition *trans*, a deferred event $e \in \mathit{defer}(\mathit{source}(\mathit{trans}))$ does not belong to $\mathit{defer}(\mathit{target}(\mathit{trans}))$, then it must be deleted from *deferQueue*, as specified as part of the *enterState* macro below.

simultaneously *dispatched* and *releasable*, that event will be deleted from the deferred events.[8]

We define in the next section the exact meaning of the state machine execution of a transition, namely by a parameterized macro *stateMachineExecution*. This leads us to the following main rule for selecting the machine transition to be executed next.

```
Rule Transition Selection
choose $e$ : $dispatched(e) \vee releasable(e)$
        choose $trans$ in $FirableTrans(e)$
                $stateMachineExecution(trans)$
        if $deferrable(e)$ then $insert(e, deferQueue)$
        if $releasable(e)$ then $delete(e, deferQueue)$
```

The rule for selecting and executing a transition fires simultaneously, at each "run to completion step", with a rule to generate completion events.

Completion events are generated when an active state satisfies the *completion condition* [4]. They trigger a transition outgoing such states. An active state is considered completed if one of the following cases occurs: (1) it is an active pseudostate, (2) it is a sequential composite state with active final state, (3) the state internal activity terminates while the state is still active, or (4) it is a concurrent composite state and all its direct substates have reached their final state. We formalize this by the predicate

$$\begin{aligned} completed(S) = true \iff\ & PseudoState(S)\ or \\ & (SequentialState(S)\ \&\ final(S) \in currState)\ or \\ & terminated(A(S))\ or \\ & (ConcurrentState(S)\ \& \\ & \quad \forall S_i \in concurrentComp(S)\ \forall a_i \in SubAgent(Self) \\ & \qquad final(S_i) \in currState(a_i)) \end{aligned}$$

where *terminated*$(A(S))$ is a derived predicate that holds if and only if the *run* of the ASM $A(S)$, which formalizes the internal activity of S, reaches a final state.

Each time the completion condition evaluates to true for an active state S that is not a direct substate of a concurrent state[9] a completion event is generated. This is expressed by the rule **Generate Completion Event** that is executed simultaneously for each state $S \in currState$.

[8] Should another interpretation be intended, we would probably change the guard *"if releasable*(e)*"* in the Transition selection rule to e.g. *"if releasable*(e) *& not dispatched*(e)*"*.

[9] This restriction reflects that in UML no direct substate of a concurrent state can generate a transition event. Such substates are required to be sequential composite states.

```
*Rule* **Generate Completion Event**
**do forall** $S \in \mathit{currState}$
          **if** $\mathit{completed}(S)$ & $\neg$ $\mathit{ConcurrentState}(\mathit{UpState}(S))$
          **then** $\mathit{generate}(\mathit{completionEvent}(S))$
```

Although the order of event dequeuing is not defined, it is explicitly required that completion events must be dispatched before any other queued events [4]. We reflect this requirement as a constraint on the monitored predicate *dispatched*. The above two rules, which fire simultaneously at each *run to completion step*, define the top level behavior of UML state machines. It remains to define in more detail the meaning of the macros appearing in those rules.

The UML requirement that an object is not allowed to remain in a pseudostate, but has to immediately move to a normal state [20], cannot be guaranteed by the rules themselves, but has to be imposed as an integrity constraint on the permissible runs.

3.6 The rule macros

We define now the subrule *stateMachineExecution* where parameterization by transitions allows us to modularize the definition for the different types of transitions and the involved states.

State machine execution If an internal transition is triggered, then the corresponding action is executed (there is no change of state and no exit or entry actions must be performed). Otherwise, if an external transition is triggered, we must determine the correct sequence of exit and entry actions to be executed according to the transition source and target state. Transitions outgoing from composite states are inherited from their substates so that a state may be exited because a transition fires that departs from some of its enclosing states. If a transition crosses several state boundaries, several exit and entry actions may be executed in the given order. To this purpose, we seek the innermost composite state that encloses both the source and the target state, i.e. their *least common ancestor*. Then the following actions are executed sequentially: (a) the exit actions of the source state and of any enclosing state up to, but not including, the least common ancestor, innermost first (see macro *exitState*); (b) the action on the transition; (c) the entry actions of the target state and of any enclosing state up to, but not including, the least common ancestor, outermost first (see macro *entryState*); finally (d) the “nature” of the target state is checked and the corresponding operations are performed.

The sequentialization and iteration constructs defined for ASMs in [8] provide the combination of black box – atomic step – view and the white box – durative – view which is needed here to guarantee that when the two ASM rules defined above are executed, all the updates which occur in the macros defined below are performed before the next event is dispatched or becomes releasable. This behavior is reflected by the parameterized macro *stateMachineExecution* (which constitutes the body of the Transition Selection Rule). The macros appearing in this rule are described below.

```
$stateMachineExecution(trans) \equiv$
**if** $internal(trans)$ **then** $action(trans)$
**else seq**
        $exitState(source(trans), ToS)$
        $action(trans)$
        $enterState(FromS, target(trans))$
        **case** $target(trans)$
            $SequentialState$: $enterInitialState(target(trans))$
            $ConcurrentState$: $startConcurrComput(target(trans))$
            $HistoryState$: $restoreConfig(target(trans))$
        **endcase**
$where\ anc = lca(source(trans), target(trans))$
      $ToS = directSubState(anc, UpChain(source(trans), anc))$
      $FromS = directSubState(anc, DownChain(anc, target(trans)))$
```

and *directSubState*: $STATE \times STATE^* \longrightarrow STATE$ is defined by $directSubState(s,L) = s'$ iff $s' \in L$ & $UpState(s') = s$, i.e. s' is the only direct substate of s belonging to the list L.

It remains to define the macros for exiting and entering states, and for the additional actions for sequential, concurrent and history states.

Exiting states If a transition that crosses the boundary of a composite state fires, we must distinguish two cases to perform the exits from nested states in an order which respects the hierarchical structure (see macro *exitState* below):

1. The agent is not inside a concurrent flow (i.e. $parent(Self) = undef$). If the agent is (1) not parent of concurrent subagents or (2) it is parent of concurrent subagents but each subagent already performed its exit actions, then for each state from the source state up to, but excluding, the source/target least common ancestor state (see the stateMachineExecution rule above), innermost first, it sequentially (a) stops the internal ongoing activities, (b) performs the exit actions , and (c) removes those states from the agent's current state. Moreover, it (d) updates the history (if defined and provided that the final state has not been reached), memorizing in it all the states it is exiting in case of deep history, or only its direct active substate in case of shallow history, and (e) updates *deferQueue* by deleting all those events which are no more deferred (see macro *sequentialExit* which uses a macro *abortInternalActivity* defined below). In case (2) the agent must furthermore update its *deferQueue* to retain all deferred events of its own but none of those processed by its subagents. Finally it disconnects all its deactivated subagents (see the corresponding macro defined below).
2. The agent is inside a concurrent flow (i.e. $parent(Self) \neq undef$). We have to consider the two cases, whether the trigger event is relevant for all the subagents running in parallel within the concurrent state or not. To this purpose we check that the transition source state belongs to the active state of both the agent and its parent (when a subagent is created, it inherits

its parent's current state, therefore at any time the *currState* of the parent agent is a subset of its subagents' *currState*). In this case, each subagent performs the same sequentialExit macro as in the first case, i.e. starting from its deepest state up to, but excluding, its parent's deepest state, it sequentially (a) stops the internal ongoing activities, (b) performs the exit actions and (c) removes those states from the agent's current state. Moreover, it (d) updates the history (if defined and provided that the final state has not been reached) memorizing in it all the states it is exiting in case of deep history, or only its direct active substate in case of shallow history, and (e) updates *deferQueue* by deleting all those events which are no more deferred (see macro *sequentialExit*). Finally, the agent is deactivated, meaning that its rule is set to undef and its current state to the empty set (see macro *deactivate*).
Now consider the case that the transition source state belongs to the active state of at least one but not to all subagents of an agent. Then the event is relevant only for this subagent, and this agent performs the sequential exit as in case 1.

```
$exitState(s,t) \equiv$ if $parent(Self) = undef$
    then if $SubAgent(Self) = \emptyset$
        then $sequentialExit(s,t)$
        elseif $noActiveSubAgents$
        then seq $deferQueue(Self) := defer(deepest(Self)) \cap \bigcap_{a_i \in SubAgent(Self)} deferQueue(a_i)$
            $sequentialExit(s,t)$
            $disconnectSubAgents$
    if $parent(Self) \neq undef$
    then if $s \in currState(Self)$ &
            $s \in currState(parent(Self))$
        then
            $sequentialExit(S, S')$
            $deactivate(Self)$
        else $sequentialExit(s,t)$

    where $S = deepest(Self)$
        $S' = deepest(parent(Self))$
        $noActiveSubAgents = \forall a_i \in SubAgent(Self) : currState(a_i) = \emptyset$
```

For the definition of the macro *sequentialExit* we use the macro *abortInternalActivity* which will be defined below. In defining *sequentialExit* we use a function $hist(s, S)$ whose value depends on whether S is a deep history state or not. $hist(s, S)$ yields $UpChain(s,\widehat{S})$ for deep history, $directSubState(S, UpChain(s,S))$ for shallow history.

```
$sequentialExit(s,t) \equiv$ loop through $S \in UpChain(s,t)$
        seq
            $abortInternalActivity(S)$
            $exit(S)$
            $currState := remove(S, currState)$
        endseq
        if $history(S) \neq undef$ & $final(S) \notin currState$
        then $memory(history(S)) := hist(s, S)$
    endloop
```

```
$deactivate(a) \equiv Rule(a) := undef$
    $currState(a) := \emptyset$
```

```
$disconnectSubAgents \equiv$ do forall $a_i \in SubAgent(Self)$
    $parent(a_i) = undef$
```

Entering states A transition may have a target state nested at any depth in a composite state. Therefore, any state enclosing the target one up to, but excluding, the least common ancestor will be entered in sequence, outermost first. Entering a state means that (a) the state is activated, i.e. inserted in *currState*, (b) its entry action is performed, and (c) the state internal activity (if any) is started. This is realized by the macro *enterState* for which we use the macro *startActivity* defined below. The agent's *deferQueue* is updated by deleting all those events which are no more deferred in the target state.

```
$enterState(s,t) \equiv$ loop through $S \in DownChain(s,t)$
        seq
            $currState := insert(S, currState)$
            $entry(S)$
            $startActivity(S)$
        endseq
        $deferQueue := deferQueue \cap defer(S)$
    endloop
```

Internal activities When a state is active, its internal activity (if any) is required to be executed. Apparently, internal activities are intended as concurrent and [4] imposes no particular scheduling conditions for them. We model this by creating a new *worker* agent whose job is to execute the activity of its associated state. The *worker* agent is created when the state is entered and after its entry action has been executed. It receives as program the ASM $A(S)$ formalizing the state activity.

```
$startActivity(S) \equiv$ extend $AGENT$ with $a$
    $Rule(a) := A(S)$
    $worker(S) := a$
```

Using an ASM as rigorous replacement for the intuitive notion of "internal UML activity", we obtain a mathematically rigorous definition without loosing generality[10]. In addition we make the UML notion of "ongoing" activity precise by defining it as steps of an ASM in a multi-agent distributed ASM run.

If an activity is aborted prior to its termination as result of the firing of an outgoing transition, then before leaving the state its associated worker agent is deactivated since its job is terminated. This is performed by the following macro which is used for defining *sequentialExit*.

```
$abortInternalActivity(S) \equiv Rule(worker(S)) := undef$
                                 $worker(S) := undef$
```

Sequential composite states A transition drawn to the boundary of a sequential composite state is equivalent to a transition to its initial pseudostate [4]. Therefore, when a composite sequential state is the target state of a triggered transition, the control passes to its initial state that is inserted in *currState*.

$$enterInitialState(S) \equiv currState := insert(init(S), currState)$$

History states If a transition incoming to a history state within a composite state fires, the configuration of active states stored in its *memory* is restored. Therefore, each state in the history is activated, i.e. it is inserted in *currState*, its entry action is performed, its activity is executed, and the state is removed from the history. The right entering order is guaranteed by the LIFO structure of *memory*.

Observe that when a state is entered for the first time or its most recently active state prior to its exit was the final state, its history (if any) must be empty [4]. This is guaranteed in our model since we initialize each history state memory to the empty sequence, delete it after using it, and store nothing in it when its enclosing state is exited by a final state.

```
$restoreConfig(H) \equiv$ loop through $S \in memory(H)$
                              seq
                                  $currState := insert(S, currState)$
                                  $entry(S)$
                                  $startActivity(S)$
                                  $memory(H) := delete(S, memory(H))$
                              endseq
                              $deferQueue := deferQueue \cap defer(S)$
                         endloop
```

Concurrent composite states If a transition incoming to a concurrent composite state fires, the flow of control is split into two or more flows of control. The currently active agent creates a new agent a_i for each concurrent component S_i. All the subagents inherit their parent's program to execute statechart diagrams,

[10] That no generality is lost derives from Gurevich's proof of the ASM thesis in [13].

its *currState* configuration and the parent's list of active deferred events. As a transition drawn to the boundary of a concurrent composite state is equivalent to a transition to any of its concurrent components and consequently to the component initial state, each agent a_i activates the component S_i and its associated initial state.

```
startConcurrComput(S) ≡
            let $S_1, \ldots, S_n = concurrentComp(S)$
                extend $AGENT$ with $a_1, \ldots, a_n$
                    do forall $1 \leq i \leq n$
                        $parent(a_i) := Self$
                        $deferQueue(a_i) := deferQueue(Self)$
                        $Rule(a_i) := Rule(Self)$
                        $currState(a_i) := insert(\{S_i, init(S_i)\}, currState)$
```

The parent agent will stand idle waiting for the termination of its subagents' computation. This is enforced by the definition of when a concurrent state is completed to trigger the completion event which may enable the transition exiting the concurrent state. The running subagents can finish their job either because of a completion event generated by their parent[11] or by the firing of an explicit event labeling a transition outgoing their enclosing state. In our model the substates' exit action and internal activity abortion are performed by the *exitState* macro, in a synchronized fashion. Other choices are easily defined modeling our rules appropriately. The UML documents seem not to mention this semantically relevant issue at all.

Remark. In a concurrent compound state S', a transition *trans*(e,g,a) outgoing from a state S in a concurrent component and incoming to a state S'' sibling of S' (see Fig. 1.a), can be viewed as split into two transitions (see Fig. 1.b): a transition *trans*$(e,g,Send\ exitEvent(S))$ from S to S', where $Send\ exitEvent(S)$ is an event generation action[12], and a transition *trans*$(exitEvent(S),true,a)$ from S' to S''. To guarantee the expected semantics of UML statecharts, we impose,

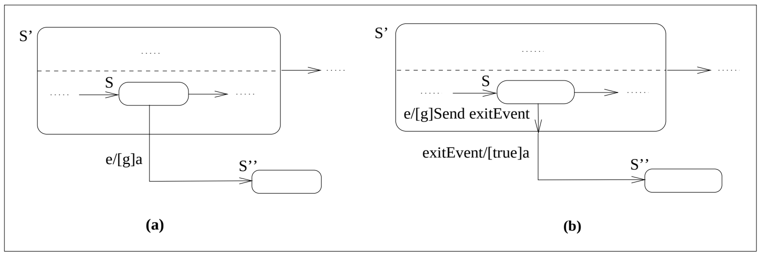


Fig. 1.

[11] In this case they all must have reached their final state.
[12] According to [4] an action labeling a transition may consist in sending a signal.

as an integrity constraint on the permissible runs, that the event *exitEvent*(S) must be dispatched before any other event (see the event handling mechanism in section 3.4).

4 Semantical equivalence among building blocks

UML statecharts encompass for notational convenience some constructs which can be defined in terms of basic constructs. Not to overload our model, we decided to include only the basic notations and to sketch here how to replace the remaining constructs by combinations of basic constructs.

Fork-Join pseudostates *Fork* and *join* pseudostates split and merge transitions arriving at, emanating from, concurrent states. A transition to the boundary of a concurrent compound state is equivalent to a transition to each of its direct substates (and therefore to their initial states), and a transition from the boundary of a concurrent compound state is equivalent to a transition from the final states of each of its substates. Therefore, the fork (resp. join) semantics can be obtained by allowing only incoming (resp. outgoing) transitions that terminate on (resp. depart from) the boundary of concurrent states, and imposing that each concurrent substate must enclose an initial and a final state.

Junction pseudostate *Junction* states are used only to chain together multiple transitions – this is known as *merge* –, or to split an incoming transition into multiple outgoing transitions labeled with different guard conditions – this is known as *conditional branch* [4].

Submachine states UML statecharts provide also *submachine states*, a syntactical convenience to facilitate reuse and modularity [4]. A submachine state is only a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. According to the UML metamodel, a submachine state is of the form $state(entry, exit, do(A), include(S'))$. One can assume that each occurrence of a submachine state is substituted by the sequential[13] composite state defined by *entry, exit,* $do(A)$*, S'*. Moreover, we identify transitions directly incoming to, respectively outgoing from, the submachine state with transitions directly incoming to, respectively outgoing from, the resulting sequential composite state.

Stub states A stub state is nothing else then an alias for an entry point to or an exit point from a state s in S' of a submachine state $state(entry, exit, do(A), include(S'))$.

Additional constructs *Synch states* are used to synchronize the execution of concurrent substates. Their semantics can be given by slightly modifying the above formalization of concurrent states.

[13] Submachine states are never concurrent [4].

5 Conclusion and related work

In this section we discuss some ambiguities in the official semantics of UML [4, 5, 20] which are resolved in the ASM model. We also show how UML requirements for state machines are satisfied by our model.

The state machine execution is formalized through the macro *stateMachine-Execution* (invoked by the rule Transition Selection) that reflects the scheme of a generic control machine. These ASM statecharts generalize the *Mealy ASMs* defined in [6].

Our model reflects all the characteristics of the state machines metamodel in [4] and adds to its structural, static definition the underlying control flow semantics. A subtle question regards the execution of ongoing state activities. What does happen when an internal transition occurs? Does the activity interrupt and then restart from the same computation point, or does it never interrupt? The way we model internal activities guarantees the second, to our understanding reasonable, alternative. However, our model can be easily adapted to formalize other behaviors.

By replacing the undefined UML terms of "action" and "activity" with (possibly structured, in the sense of [8]) "ASM rule", we provide a precise mathematical content to these terms without loosing the generality intended by the designers of UML (see in this connection Gurevich's proof of the ASM thesis [13]). Our model also provides a precise meaning of the vague UML term "ongoing internal activity", namely as execution of an ASM in a multi-agent distributed run as defined in [12]. The sequentialization, iteration and submachine constructs defined for ASMs in [8] clarify in what sense sequences of nested exit and entry actions can be guaranteed to be executed in one "run to completion step", as postulated by the UML documents, namely before the next event may trigger the next "step". Our model also makes some semantically relevant features[14] explicit which seem not to have been considered in the official UML documents.

Several semantics for statecharts have been proposed in the literature [21]. Most of these are concerned with modeling Harel's statecharts, whose semantics is rather different from UML state machines (e.g. in the event handling policy). Although our model can be adapted to grasp such differences, our intent is to define the UML state machine semantics up to the degree of precision one can reach without compromising the desired freedom of the so called "semantic variation points".

Differently from the formalization of UML state machines in [11, 19], our model reflects the original structure of machines as described in the UML documents, without imposing any graphical transformation or flattening of diagrams. [11] uses graph rewriting techniques to transform UML state machines into a "normal form" machine, without considering the execution of actions and activities. The model in [19] leaves out some state machines features, and some are covered by means of semantical equivalences which, however, do not always

[14] E.g. whether abortion of internal activities and exit actions of concurrent agents should be synchronized or not.

respect the UML metamodel constraints (see [4], pp. 2-126). For instance, entry/exit actions in a state are replaced by attaching such actions respectively to the state incoming/outgoing transitions, whereas the metamodel asserts that the multiplicity of `Action` in `Transition` is 0..1, that is no or exactly one action may label a transition.

In [17] the operational behavior of UML state machine constructs is described using pseudo-code in a way which in many places includes specific implementation decisions (mostly without stating them), whereas we tried to let the intended semantic variation points of UML stand out explicitly as such.

References

1. Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.
2. *Rational Software Corporation, Unified Modeling Language UML, version 1.3*, 1999.
3. *UML 1.3 Notation*, 1999. (Published as part of [2]).
4. *UML 1.3 Semantics*, 1999. (Published as part of [2]).
5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
6. E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and W. Stephan and P. Traverso and M. Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
7. E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *AMAST2000*, volume 1816 of *LNCS*, pages 293–308. Springer Verlag, May 2000.
8. E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In *Gurevich Festschrift CSL 2000*, 2000. (To appear).
9. A. S. Evans, J-M. Bruel, K. Lano R. France, and B. Rumpe. Making UML Precise. In *In OOPSLA'98 Workshop on Formalizing UML. Why and How?*, October 1998.
10. R. B. France, A. S. Evans, K. C. Lano, and B. Rumpe. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, Accepted for publication, 1998.
11. Martin Gogolla and Francesco Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.
12. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
13. Y. Gurevich. Sequential Abstract State Machines capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1, 2000. (To appear).
14. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer, IEEE Computer Society*, 30(7):31–42, 1997.
15. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Soft. Eng. method*, 5(4):293–333, 1996.
16. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

17. Ivan Paltor and Johan Lilius. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.
18. The precise UML group. `http://www.cs.york.ac.uk/puml/`.
19. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In *FASE 2000 - Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, 2000. (To appear).
20. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
21. M. von der Beek. A Comparison of Statechart Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 526. Lecture Notes in Computer Science, 1994.

To appear in: International Workshop on Abstract State Machines ASM'2000 (Eds. Y.Gurevich, Philipp Kutter, Martin Odersky, Lothar Thiele). Springer LNCS, 2000.