On Formalizing UML State Machines Using ASMs

Egon Börger[a] Alessandra Cavarra[b] Elvinia Riccobene[c]

[a]Dipartimento di Informatica Università di Pisa, Italy

[b]Oxford University Computing Laboratory, Wolfson Building Parks Road, Oxford, OX1 3QD, UK

[c]Dipartimento di Matematica e Informatica Univerisità di Catania, Italy

1. INTRODUCTION

UML (Unified Modeling Language [1]) is a standard notation based on a set of diagrams to describe the structure and the behavior of software systems. In [2] the authors claim 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". However, the UML semantics document [1] only gives an unambiguous textual definition of the syntax for the 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 [11,10,24] and the *pUML* (*precise UML*) group has been created to achieve this goal [23]. With this paper we want to survey previous work [3–5] done to provide a precise definition of some basic concepts of UML which are related to the dynamics of systems. Technical details are left out for lack of space but they are duly referenced. Our goal here is to *discuss* a framework for the rigorous description and analysis of different possible logically consistent interpretations of the intuitions which underly UML concepts: how (a) it faithfully reflects the original intuitions as far as possible; (b) it makes the so-called *semantic variations points* explicit; (c) clarifies some dark but semantically relevant points in the UML documents. This includes the event deferring and completion mechanism, the meaning of atomic and durative actions, concurrent internal activities and conflict situations which may arise through the concurrent behavior of active objects.

We develop this framework for one of the principal diagram types which are used in UML for the description of system behavior, namely statecharts or UML state machines.

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")

if *Condition* **then** *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* [14] to model the concurrent substates and the internal activities which may appear in a UML state machine. A multi-agent ASM is given by a set of (sequential) agents, each executing a program consisting of ASM rules. Multi-agent runs are defined in [14].

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 state machines 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 [6]. 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. UML STATE DIAGRAMS

State diagrams focus on the event-ordered behavior of an object, a feature which is specially useful in modeling reactive systems.

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. *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 [1]. Accordingly, we introduce the abstract sets $STATE$ and $TRANSITION$ (more details can be found in [3,4]).

3.1. Handling State Diagrams Control

To formalize the control in a state machine, we introduce a set of $AGENT$s which move through the 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. "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" [1]. To maintain the current configuration of active states, we introduce a dynamic function $currState : AGENT \rightarrow \mathcal{P}(STATE)$ whose updates follow the control flow of the given state machine. The function $deepest : AGENT \rightarrow STATE$ yields the last (innermost) state reached by an agent.

The agents execute UML state machines, i.e. all use the same 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 $parent : AGENT \rightarrow AGENT \cup \{undef\}$.The active subagents of an agent a are collected in the set $SubAgent(a) = \{a' \in AGENT \mid parent(a') = a\}$. The function $descendents : AGENT \rightarrow \mathcal{P}(AGENT)$ yields the set of all the active subagents of an agent at any depth of concurrency.

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** (described below) for selecting and executing a transition, and **Generate Completion Event** (see [3,4] for a complete explanation) to handle completion events. These two rules define the *top level behavior* of UML state diagrams.

3.2. Event Handling

"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" [1]. Since the particular event enqueuing and dispatching mechanisms are deliberately not furthermore specified in UML, we model them explicitly as semantic variation points and therefore use a monitored predicate *dispatched* indicating which event is dequeued to be processed.

The *run to completion assumption* requires that a state diagram processes one event at a time and finishes all the consequences of that event before processing another event [1]. We reflect this requirement by a constraint on *dispatched*.

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. belonging to *currState*) and whose associated guard evaluates to true. This is expressed by the condition $enabled(t, e, a) \equiv event(t) = e$ & $guard(t)$ & $source(t) \in currState(a)$.

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 [1]. In all the other cases, the enabled transitions are said to be in *conflict* with each other. A number of conflicting situations are reported in the UML official documentation. In general, these conflicts are left open as semantic variation points. In UML the selection among conflicting transitions is constrained only when two or more transitions with different source states, but belonging to the same active state configuration, are enabled by the occurrence of an event. In this case priority is given to the innermost enabled transition.

Let $enabled(e)$ be the set of all transitions (simultaneously) enabled by e. On $enabled(e)$ we define an equivalence relation: $\forall\ t_1, t_2 \in enabled(e),\ t_1 \sim t_2$ iff $source(t_1) = source(t_2)$. The nesting of states induces the total order relation $\leq$ on the quotient set $enabled(e)/\sim$, defined as $[t_1]_a \leq [t_2]_b,\ a, b \in AGENT$, iff ($a = b$ & $source(t_1)$ is a direct or a transitively nested substate of $source(t_2)$) $\vee$ ($a \in descendants(b)$ & source$(t_1) \in currState(a)$ & source$(t_2) \in currState(b)$ & $source(t_1)$ is a direct or a transitively nested substate of $concurrentComp(source(t_2))$.

Let $FirableTrans(e)$ be the minimum equivalence class in $enabled(e)/\sim$. 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).

Remark. More conflict situations, not reported in the UML official documentation and mainly concerning transitions in concurrent substates, have been discovered by simulating our model in [7]. For a complete description of conflicting situations we refer the reader to [4].

Deferred Events. 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. An event deferred in a composite state is automatically deferred in all its directly or transitively nested substates [1].

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*: $deferrable(e) = true \Leftrightarrow enabled(e) = \emptyset$ & $e \in defer(deepest)$.

To store deferred events we associate to each agent a list *deferQueue* of events that is dynamically updated during the computation. We can therefore define $deferred(e)$ to mean $e \in deferQueue$. We call a deferred event *releasable* when it is ready to be consumed, i.e. when it can trigger a transition in the current state: $releasable(e) = true \Leftrightarrow deferred(e)$ & $enabled(e) \neq \emptyset$.

4. STATE DIAGRAMS BEHAVIOR

To describe the dynamic semantics of state diagrams by ASM rules, we proceed in a modular and top down fashion. At the top level, we describe the run-to-completion (rtc) step which is the passage between two state configurations, and then we define in more details the sequence of steps a machine performs in changing state.

The *Run To Completion* Step. A rtc step of a state machine consists in choosing an event with an enabled transition and firing the enabled transition. Apparently, UML leaves it unspecified how to choose between dispatched and releasable events. We reflect this by using a selection function which, at any step, 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*. This implies that when choosing an event which is simultaneously *dispatched* and *releasable*, that event will be deleted from the deferred events.

The submachine *stateMachineExecution* is responsible for the execution of transitions. Its pa-

rameterization by transitions allows us to modularize the definition for the different types of transitions and the involved states.

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

State machine execution. If an internal transition is triggered, then the corresponding action is executed (no exit or entry actions are 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 *ToS*, innermost first; (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 *FromS*, outermost first; finally (d) the "nature" of the target state is checked and the corresponding operations are performed. (Details on the macros can be found in [3,4].)

```
stateMachineExecution(trans) ≡
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
```

The ASM constructs for sequentialization and iteration defined in [6] provide the combination of black box (atomic step) view and the white box (durative) view which is needed in the definition of the parameterized macro *stateMachineExecution* to guarantee that when the ASM rule is executed, all the updates which occur in the called macros are performed before the next event is dispatched or becomes releasable.

Transitions to/from Concurrent Substates If a transition drawn from a concurrent state boundary fires, any active states in all its subregions are properly exited according to the *stateMachineExecution* macro. However, transitions may be drawn directly from states within a concurrent state region at any nesting depth to outside states. All exit actions are performed for any states that are exited on any transition [1].

In case a triggered transition overcomes the boundary of a concurrent sub-region, the *run-to-completion* step has to be performed by the ancestor agent lying in the transition's source/target least common ancestor state. The *stateMachineExecution* performed by this agent guarantees the correct semantics. In order to guarantee the reaction of the correct agent to the firing of a transition, we have redefined the predicate $enabled(t, e, a)$. (See [5] for details).

A transition drawn to a concurrent state boundary indicates a transition to the initial pseudostate of each of its concurrent substates. However, transitions may be drawn directly to states within a concurrent composite state region at any nesting depth. All entry actions are performed for any states that are entered on any transition. The transition target state is entered, whereas any other concurrent regions start with their default initial pseudostate.

To formalize the complex entering mechanism when the triggered transition overcomes the boundary of a concurrent region, we have refined the entering state procedure, as given by the macro *enterState*, in order to guarantee that the following actions are executed sequentially: (a) *entering* the sequence of nested states crossed by the transition from the direct substate of the transition's source/target least common ancestor state, to the first concurrent state of the chain; (b) *starting* the concurrent computation of as many

new sub-agents as the number of concurrent subregions of the concurrent state; all the newly created agents are set on the initial states of their associated subcomponents, except the agent lying in the subregion crossed by the transition, which must (c) *proceed the entering action* through the down chain of nested states till the next concurrent nested state. Details on the *enterState* macro refinement can be found in [5].

5. DISCUSSION

In this section we discuss some ambiguities in the official semantics of UML state machines [1] which are resolved in the ASM model and show how the UML requirements for statecharts are satisfied by our model.

The state machine execution is formalized through the macro *stateMachineExecution* that reflects the scheme of a generic control machine. The resulting ASM statecharts reflect all the characteristics of the state machines metamodel in [1] and add to its structural, static definition the underlying control flow semantics. It answers subtle questions related to the execution of ongoing state activities. E.g. 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 informal UML terms of "action" and "activity" with "ASM rule" in its structured form [6], we provide a precise mathematical content to these terms without loosing the generality intended by the designers of UML. In particular, a precise meaning of the informal UML term "ongoing internal activity" is obtained, namely as execution of an ASM in a multi-agent distributed run as defined in [14]. The sequentialization, iteration and ASM submachine constructs defined in [6] 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 explicit some semantically relevant features—e.g. whether abortion of internal activities and exit actions of concurrent agents should be synchronized or not—which have not been considered in the official UML specification.

Many papers on the semantics of statecharts exist in the literature (see next section), 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. Unless necessary to avoid inconsistencies, we do not take any position on which of these understandings of UML state machine concepts is reasonable or desirable but build a framework which shows the freedom offered to implementors by the semantic variations points, read: different possible logically consistent interpretations of the intuitions which underly the UML concepts.

Our specific goals concerning UML state machines are to: (a) accurately define the UML event handling scheme making its semantic variation points explicit, including the event-deferring and the event-triggered run-to-completion mechanisms; (b) encapsulate 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) clarify various difficulties concerning the scheduling scheme for internal ongoing (really concurrent) activities, including an analysis of the conflict situations which may arise through the concurrent behavior of active objects; (d) describe all the UML state machine features that break the thread-of-control, including transitions from and to concurrent states in the context of event deferring and run-to-completion. Note that this involves the sequential execution, respectively the termination, of entry and exit actions, in the order prescribed by the diagram structure as well as starting or terminating all the subcomputations encountered at the boundaries of concurrent substates between source and target state; (e) integrate smoothly the state machine control structure with the data flow; (f) provide a precise computational content to the UML terms of

atomic and durative actions/activities, without losing the intended generality of these concepts.

Our model can also serve as reference model for implementing tools for code generation, simulation and verification of UML models. A. Cavarra has extended the ASM interpreter AsmGofer [26] to build a simulator for UML state machines that can be used for high-level simulation [8,7].

6. RELATED WORK

Several semantics for statecharts have been proposed in the literature. Differently from the formalization of UML state machines in [13,19, 24,27], our model reflects the original structure of machines as described in the UML documents, without imposing any graphical transformation or flattening of diagrams. [13] uses graph rewriting techniques to transform UML state machines into a "normal form", without considering the execution of actions and activities. A similar approach based on graph transformation is used in [19] where however history states, actions, and guards are not formalized. Also in the model in [24], which uses an algebraic specification approach, some state machines features are left out or covered by semantical equivalences. A partial definition of UML state machine semantics is given in [27] using Labelled Transition Systems. Latella et al. [21] exploit hierarchical automata as an intermediate language for formulating the semantics in terms of Kripke structures.

Lilius and Paltor [22] develop an operational semantics of UML statechart on the basis of a term-rewriting system and a predetermined static priority relation on transitions. This then serves as the theoretical basis of their vUML tool. Schäfer et al. propose in [25] a dynamic computation algorithm; they model states as individual processes communicating through channels. In order to determine firable transitions, the control is transferred from the top state process down to leaf state processes. Translations of UML models into the SPIN input language Promela for verification in SPIN are provided in [22,25,20,12] .

Compton et al. present in [9] an ASM model for statecharts in order to develop a tool for UML model properties verification. They built an extended statechart by adding a transition for every affected state of a firing transition. In [17,18] Jürjens provides an ASM semantics for a part of UML that allows one to use UML subsystems to group together several diagrams making use of the statechart semantics from [3]. Using ASMs, Jin et al. present in [16] a generic approach to integrating a visual language in a heterogeneous modeling and simulation environment; they use UML diagrams as an example for their approach.

REFERENCES

1. *OMG Unified Modeling Languages Specification, version 1.4*, 2001.
2. G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
3. E. Börger, A. Cavarra, E. Riccobene. Modeling the dynamics of UML state machines. LNCS 1912, pages 223–241, 2000.
4. E. Börger, A. Cavarra, E. Riccobene. A precise semantics of UML State Machines: making semantic variation points and ambiguities explicit. In *Proc. SFEDL02 - ETAPS 2002*, 2002.
5. E. Börger, A. Cavarra, E. Riccobene. Modeling the meaning of transitions from and to concurrent states in UML State Machines. In *SAC 2003*.
6. E. Börger, J. Schmid. Composition and submachine concepts for sequential ASMs. Number 1862 in LNCS, pages 41–60, 2000.
7. A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method*. PhD thesis, 2000.
8. A. Cavarra, E. Riccobene. Simulating uml statecharts. In Proc. *EUROCAST 2001*, 2001.
9. K. Compton, J. Huggins, W. Shen. A semantic model for the state machine in the Unified Modeling Language. In *Dynamic Behavior in UML Models: Semantic Questions UML 2000 workshop*, 2000.
10. A. S. Evans, J-M. Bruel, K. Lano R. France, B. Rumpe. Making UML precise. In *OOPSLA'98*, 1998.
11. R. B. France, A. S. Evans, K. C. Lano, B. Rumpe. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, Accepted for publication, 1998.
12. S. Gnesi, D. Latella, M. Massink. Model checking UML statechart diagrams using JACK. In

Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering. IEEE, 1999.
13. M. Gogolla, F. P. Presicce. State diagrams in UML: A formal semantics using graph transformations. In Proc. *PSMT'98*. TUM-I9803.
14. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
15. D. Harel, E. Gery. Executable object modeling with statecharts. *Computer, IEEE Computer Society*, 30(7):31–42, 1997.
16. Y. Jin, R. Esser, and J.W. Janneck. Describing the Syntax and Semantics of UML Statecharts in a Heterogeneous Modelling Environment. In *Diagrams 2002*, volume 2317. LNAI, 2002.
17. J. Jürjens. A UML statecharts semantics with message-passing. In *SAC2002*, ACM.
18. J. Jürjens. Formal Semantics for Interacting UML subsystems. In *FMOODS 2002*. Kluwer.
19. S. Kuske. A formal semantics of UML State Machines based on structured graph transformation. In Proc. *UML 2001*, volume 2185 of *LNCS*.
20. D. Latella, I. Majzik, M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *FAC99*, 11(6).
21. D. Latella, I. Majzik, M.Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS99*.
22. I. Paltor, J. Lilius. Formalising UML state machines for model checking. In Proc. *UML'99*, volume 1723 of *LNCS*. Springer.
23. The precise UML group. `http://www.cs.york.ac.uk/puml/`.
24. G. Reggio, E. Astesiano, C. Choppy, H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In Proc. *FASE 2000*, *LNCS* 1783.
25. T. Schfer, A. Knapp, S. Merz. Model Checking UML State Machines and Collaborations. *ENTCS*, 47:1–13, 2001.
26. J. Schmid. Executing ASM specifications with AsmGofer. `http://www.tydo.de/AsmGofer`.
27. M. von der Beeck. Formalization of UML-Statecharts. In Proc. *UML 2001*, volume 2185 of *LNCS*. Springer, 2001.