Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach*

G. Reggio[1], E. Astesiano[1], C. Choppy[2], and H. Hussmann[3]

[1] DISI, Università di Genova - Italy
[2] LIPN, Institut Galilée - Université Paris XIII, France
[3] Department of Computer Science, Dresden University of Technology - Germany

Abstract. We propose a precise definition of UML active classes through associated labelled transition systems using the algebraic specification language CASL. We are convinced that the first step to make UML precise is to find an underlying formal model for the systems modelled by UML, and we argue that labelled transition systems are a sensible choice. This modelization will help understanding the UML constructs and will improve their use in practice. One of our aims is, in the future, to use the powerful animation and verification tools available for algebraic specifications with UML specifications. We simplify the problem of the applicability of our semantics by restricting the state machine constructs considered. This restriction does not, however, narrow the UML subset in study because the restricted constructs can be replaced by equivalent combinations of other constructs. Because of some ambiguities in the UML official semantics, we discuss the several options at hand and choose, for each ambiguous case, the semantics that either makes more sense or that allows to simplify the problem the most.

1 Introduction

The Unified Modeling Language (UML) [13] is an industry standard language for specifying software systems. This language is unique and important for several reasons:

- UML is an amalgamation of several, in the past competing, notations for object-oriented modelling. For a scientific approach, it is an ideal vehicle to discuss fundamental issues in the context of a language used in industry.
- Compared to other pragmatic modelling notations in Software Engineering, UML is very precisely defined and contains large portions which are similar to a formal specification language, as the OCL language used for the constraints.

* Work supported by CoFI (ESPRIT Working Group 29432) and MURST – Sistemi Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software.

It is an important issue in Software Engineering to finally close the gap between pragmatic and formal notations and to apply methods and results from formal specification to the more formal parts of UML. This paper presents an approach contributing to this goal and has been carried out within the European "Common Framework Initiative" (CoFI) for the algebraic specification of software and systems, partially supported by the EU ESPRIT program. Within the CoFI initiative [8], which brings together research institutions from all over Europe, a specification language called "Common Algebraic Specification Language" (CASL) was developed which intends to set a standard unifying the various approaches to algebraic specification and specification of abstract data types. It is a goal of the CoFI group to closely integrate its work into the world of practical engineering. As far as specification languages are concerned, this means an integration with UML, which may form the basis for extensions or experimental alternatives to the use of OCL. This would allow for instance to specify user-defined data types that just have values but do not behave like objects, and/or to use algebraic axioms as constraints. These new constraints may cover also behavioural aspects, because there exist extensions of algebraic languages that are able to cope with them [4], whereas OCL does not seem to have any support for such aspects. The long-term perspective of this work is to build a bridge between UML specifications and the powerful animation and verification tools that are available for algebraic specifications.

To this end we need to precisely understand (and formally define) the most relevant UML features. In this paper we present the results of such formalization work for active objects with associated state machine [1], which was guided by the following ideas.

Real UML Our concern is the "real" UML (i.e., all, or better almost all, its features without simplifications and idealizations) as presented in the official OMG documentation [14] (shortly UML 1.3 from now on).

Based on an underlying model We are convinced that the first step to make UML precise is to find an underlying formal model for the systems modelled by UML, in the following called *UML-systems*.

Our conviction also comes from a similar experience the two first authors had, many years ago, when tackling the problem of the full formal definition of Ada, within the official EU project (see [1]). There too an underlying model was absolutely needed to clarify the many ambiguities and unanswered questions in the ANSI Manual.

We argue that labelled transition systems could be a sensible model choice; indeed, they were used quite successfully to model concurrent languages as Ada [1], but also a large part of Java [3].

Integrated with the formalization of the other fragments of UML The ultimate goal of this work is to have an approach by which it is easily possible to integrate semantically the most relevant diagram types. The underlying model

[1] Following UML terminology *state machine* is the abstract name of the construct, whereas *state chart* is the name of the corresponding diagram; here we always use the former.

plays a relevant role in that, because the UML diagrams of the various kinds either describe a part of such model or express some properties about it.

Lightweight formalization By "lightweight" we mean that we use the simplest formal tools and techniques: precisely labelled transition systems algebraically specified using a small subset of the specification language CASL (conditional specification with initial semantics). However, to further simplify the formalization CASL could be replaced by a simpler mathematical notation plus inductive definitions.

The formalization of active classes and state machines lead to perform a thorough analysis uncovering many problematic points. Indeed, the official informal semantics of UML, reported in UML 1.3, is in some points either incomplete, or ambiguous, or inconsistent or dangerous (i.e., the semantics of a part is clearly formulated but its allowed usage seem problematic from a methodological point of view). To stress this aspect and to help the reader we have used the mark pattern $\boxed{\textbf{PROBLEM} \text{ ....}}$ to highlight them.

Some of these cases are "semantic variation points", i.e., points where intentionally the semantics allows for multiple interpretations. In such cases we have formalized the most general choice, in the sense that any behaviour in our semantics is a behaviour of an admissible semantic variation and a behavior of an admissible semantic variation is a behaviour in our semantics.

The use of an algebraic specification language allows for abstract, modular and easily modifiable definitions, and that could make easy to modify our semantics in some point to take into account instead a particular variation.

In Sect. 2 we shortly introduce the UML part that we consider. Then in the following sections we introduce the used formal techniques (labelled transition systems and algebraic specifications), and present step after step how we built the labelled transition system modelling the objects of an active class with an associated state machine. Due to lack of room part of the definition and the complete formal model (rather short and simple) are in [11].

2 Introducing UML: Active Classes and State Machines

The UML defines a visual language consisting of several diagram types. These diagrams are strongly interrelated by a common abstract syntax and are also related in their semantics. The semantics of the diagrams is currently defined by informal text only.

The most important diagram types for the direct description of object-oriented software systems are the following:

– Class diagrams, defining the static structure of the software system, i.e., essentially the used classes, their attributes and operations, possible associations (relationships) between them, and the inheritance structure among them. Classes can be passive, in which case the objects are just data containers. For this paper, we are interested in active classes, where each object has its own thread(s) of control.

– Statechart diagrams (state machines), defining the dynamic behaviour of an individual object of a class over its lifetime. This diagram type is very similar to traditional Statecharts. However, UML has modified syntax and semantics according to its overall concepts.
– Interaction diagrams, illustrating the interaction among several objects when carrying out jointly some use case. Interaction diagrams can be drawn either as sequence diagrams or as collaboration diagrams, with almost identical semantics but different graphical representation.

A UML *state machine* is very similar to a classical finite state machine. It depicts states, drawn as rounded boxes carrying a name, and transitions between the states. A transition is decorated by the name of an event, possibly followed by a specification of some action (after a slash symbol). The starting point for the state machine is indicated by a solid black circle, an end point by a solid circle with a surrounding line.

The complexity of UML state machines compared to traditional finite state machines comes from several origins:

– The states are interpreted in the context of an object state, so it is possible to make reference, e.g., in action expressions, to object attributes.
– There are constructs for structuring state machines in hierarchies and even concurrently executed regions.
– There are many specialized constructs like entry actions, which are fired whenever a state is entered, or state history indicators.

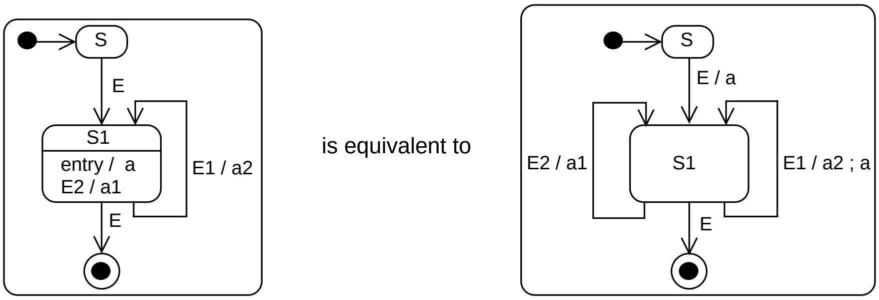
In order to simplify the semantical consideration in this paper, we assume the following restrictions of different kinds. Please note that none of these assumptions restricts the basic applicability of our semantics to full UML state machines!

We do not consider the following UML state machine constructs, because they can be replaced by equivalent combinations of other constructs.

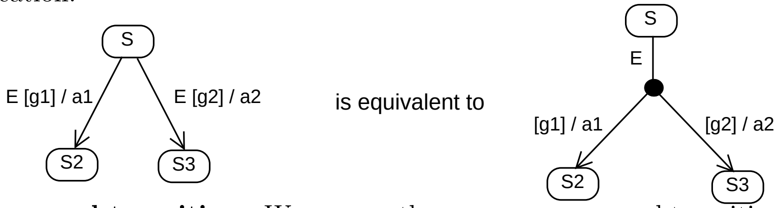
Submachines We can eliminate submachines by replacing each stub state with the corresponding submachine as UML 1.3 p. 2.137 states "It is a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state".

Entry and exit actions Entry and exit actions associated with a state are a kind of shortcut with methodological implication, see, e.g., [13] p. 266, but semantically they are not relevant; indeed we can eliminate them by adding such actions to all transitions entering/leaving such state.

Internal transitions An internal transition differs from a transition whose source and target state coincide only for what concerns entry/exit actions. Because we have dropped entry/exit actions, we can drop also internal transitions. The following picture shows, on an example, how to eliminate entry actions and internal transitions.

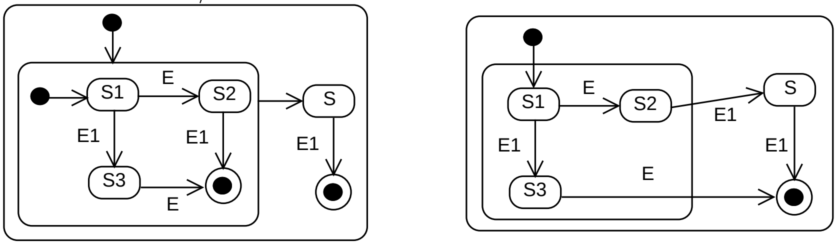


Different transitions leaving the same state having the same event trigger We can replace these transitions with compound transitions using junction states. The latter presentation seems better from a methodological point of view, because it groups together all transitions leaving a state with the same event trigger. The picture below shows on an example of this simplification.



Compound transitions We assume there are no compound transitions except the complex ones and compound transitions of length two (e.g., as those needed in the above case). Indeed compound transitions are used only for presentation reasons and can be replaced by sets of simpler transitions.

Multiple initial/final states We assume that there is always a unique initial state that is placed at the top level (determine the initial situation of the class objects) and a unique final state that is placed at the top level too (when it is active the object will perform a self destruction). The remaining initial/final states can be replaced by using complex transitions. The picture below shows an example of equivalent state machines, differing only in the number of initial/final states.



Terminate action in the state machine Indeed it can be equivalently replaced by a destroy action addressed to the self.

We do not consider the following features just to save space; indeed we think that they present at the semantic level problems posed by other considered constructs.

– Operations with return type and return actions
– Synch and history states
– Generalization on the signals (type hierarchy on signals)
– Activities in states (do-activities).

3 Modelling Active Objects with Labelled Transition Systems

3.1 Labelled Transition Systems

A *labeled transition system* (shortly *lts*) is a triple $(ST, LAB, \rightarrow)$, where ST and LAB are two sets, and $\rightarrow \subseteq ST \times LAB \times ST$ is the *transition relation.* A triple $(s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$.

Given an lts we can associate with each $s_0 \in ST$ the tree (*transition tree*) whose root is s_0, where the order of the branches is not considered, two identically decorated subtrees with the same root are considered as a unique subtree, and if it has a node n decorated with s and $s \xrightarrow{l} s'$, then it has a node n' decorated with s' and an arc decorated with l from n to n'.

We model a process P with a transition tree determined by an lts $(ST, LAB, \rightarrow)$ and an initial state $s_0 \in ST$; the nodes in the tree represent the intermediate (interesting) situations of the life of P, and the arcs of the tree the possibilities of P of passing from one situation to another. It is important to note here that an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: P in the situation s has the *capability* of passing into the situation s' by performing a transition, where the label l represents the interaction with the environment during such a move; thus l contains information on the conditions on the environment for the capability to become effective, and on the transformation of such environment induced by the execution of the transition.

Notice that here by process we do not mean only "sequential process", but also concurrent processes that have cooperating components (that are in turn other processes – concurrent or not), and that can be modelled through particular lts, named *structured lts.* A structured lts is obtained from from another lts describing such components, say *clts*; its states are built by the states of *clts*, and its transitions starting from a state S are determined by those of *clts* starting from the subcomponents of S.

An lts may be formally specified by using the algebraic specification language CASL (see [10]) with a specification of the following form:

spec LTS =
    STATE **and** LABEL **then**
free { **pred** $\_ \xrightarrow{\_} \_$: *State* | *Label* | *State*
    axioms ......
} end

whose axioms have the form $\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \alpha_{n+1}$, where for $i = 1, \ldots, n+1$, α_i is a positive atom (i.e., either a predicate application or an equation). The CASL construct **free** requires that the specification has an initial semantics [10]; if we forget about the algebraic apparatus, then this amount to use the conditional axioms as defining a set of inductive rules; hence the initial semantics corresponds to the inductively defined lts.

Assume we have an active class ACL with an associated state machine SM belonging to the subset of UML introduced in Sect. 2, and assume that ACL and SM are statically correct, as stated in UML 1.3. Here we present how we built the lts L modelling the objects of the class ACL, following the steps below, which will be detailed in the rest of the paper. It will be clear that our technique implies the possibility of defining a computable function associating with textual presentations of state machines the CASL specifications of the corresponding lts's.

1. determine whether L is simple or structured
2. determine the grain of the L-transitions
3. if L is structured, determine its components and the lts modelling them
4. determine the labels of L
5. determine the states of L
6. determine the transitions of L by means of conditional rules (in this case, because we are using CASL, by conditional axioms).

The constraints attached either to ACL or to SM are treated apart in Sect. 3.7, because they do not define a part of L, but just properties on it.

To avoid confusion between the states and the transitions of the state machine SM with those of the lts L, we will write from now on *L-states* and *L-transitions* when referring to those of L.

3.2 Is L Simple or Structured?

The first question is to decide whether L is simple or structured; in terms of UML semantics this means to answer the following question:

PROBLEM Does an active object correspond to a single thread of control (running concurrently with those corresponding to the other objects), or to several ones?

Unfortunately, UML 1.3 is rather ambiguous/inconsistent for what concerns this point. Indeed, somewhere it seems to suggest that there is exactly one thread, as in UML 1.3 p. 2-23, p. 2-149, p. 2-150:

> *It is possible to define state machine semantics by allowing the run-to-completion steps to be applied concurrently to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration.*

Otherwise, UML 1.3 seems to assume that there are many threads, as in p. 2-133, p. 2-144, p. 3-141:

> *A concurrent transition may have multiple source states and target states. It represents a synchronization and/or a splitting of control into concurrent threads without concurrent substates.*

and in p. 2-150:

> *An event instance can arrive at a state machine that is blocked in the middle of a run-to-completion step from some other object within the same thread, in a circular fashion. This event instance can be treated by orthogonal components of the state machine that are not frozen along transitions at that time.*

However, this seems to be what is called in UML a "semantic variation point", thus we consider the most general case, by assuming that an active object may correspond to whatever number of threads, and that such threads execute their activities in an interleaving way. Thus L may be a simple lts. Perhaps, a better way to fix this point is to introduce two stereotypes: *one-thread* and *many-threads*, to allow the user to decide the amount of parallelism inside an active object.

3.3 Determining the Granularity of the *L*-Transitions

Using lts means that we model the behaviour of processes by splitting it into "atomic" pieces (the L-transitions); so, to define L, we must first determine the granularity of this splitting.

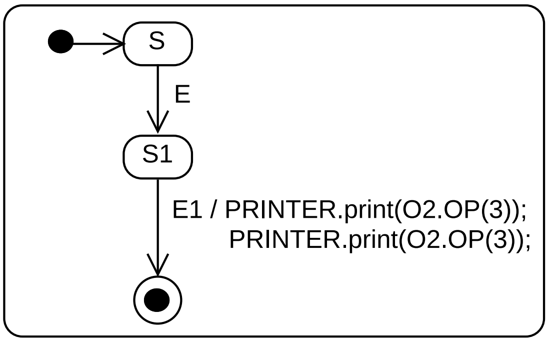


Fig. 1. A simple State Machine

PROBLEM By looking at UML 1.3 we see that there are two possibilities corresponding to different semantics.

1. each L-transition corresponds to performing all transitions of the state machine SM triggered by the occurrence of the same event starting from a set of active concurrent states. Because L-transitions are mutually exclusive, this choice corresponds to a semantics where L-transitions, and thus such group of transitions of SM, is implicitly understood as critical regions.
2. each L-transition corresponds to performing a part of a state machine transition; then the atomicity of the transitions of SM (run-to-completion condition) required by UML 1.3 will be guaranteed by the fact that, while executing the various parts of a transition triggered by an event, the involved threads cannot dispatch other events. In this case, also the parts of state machine transitions triggered by different events may be executed concurrently.

The example in Fig. 1, where we assume that there is another object O2 with an operation OP resulting in a printable value, shows an instance of this problem. Choice 1 corresponds to say that in any case pairs of identical values will be printed, whereas choice 2 allows for pairs of possibly different values (because the value returned by OP can be different in the two occasions due to the activity of O2.

Choice 2 seems to be what was intended by UML designers and so here we model it; however we could similarly also model choice 1.

3.4 Determining the *L*-Labels

The L-labels (labels of the lts L) describe the possible interactions/interchanges between the objects of the active class ACL and their external environment (the other objects comprised in the model). As a result of a careful scrutiny of UML 1.3 we can deduce that the basic ways the objects of an active class interact with the other objects are the following, and we distinguish them in "input" and "output":

input:
- to receive a signal from another object
- to receive an operation call from another object
- to read an attribute of another object (+)
- to have an attribute updated by another object (+)
- to be destroyed by another object (+)
- to receive from some clock the actual time (see [13] p. 475)

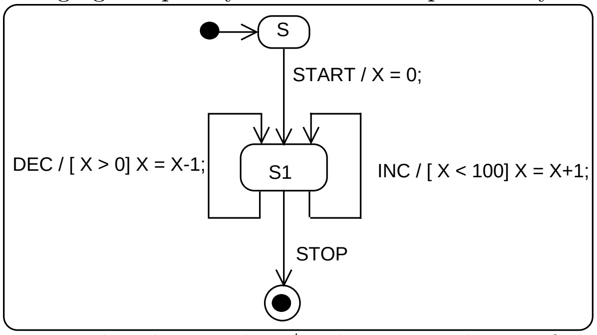
output:
- to send a signal to another object
- to call an operation of another object
- to update an attribute of another object (+)
- to have an attribute read by another object (+)
- to create/destroy another object

However, UML 1.3 does not consider explicitly the interactions marked by (+), which do not correspond to send or to receive events, and does not say anything about when they can be performed (e.g., they are not considered by the state machines).

PROBLEM When may an object be destroyed?

A way to settle this point is to make "to be destroyed" an event, which may be dispatched when the machine is not in a run-to-completion-step and may appear on the transitions the state machine.

PROBLEM The interactions corresponding to have an attribute read/ updated by other objects are problematic.
May an object have its attributes updated by some other object?
If the answer is yes, then when such update may take place? For example, is it allowed during a run-to-completion-step?
Are there any "mutual exclusion" properties on such updates? or may it happen that an object O1 updates an attribute A of O while O is updating it in a different way, or that O1 and O2 updates simultaneously A in two different ways?
May an active object have a behaviour not described by the associated state machine (see UML 1.3 p. 2-136), because another object updates its attributes?
In the following example, another object may perform O.X = O.X + 1000, changing completely the behaviour specified by this state machine.



Notice also that reading/updating attributes of the other active objects is an implicit communication mechanism, thus yielding a dependency between their behaviours that is not explicitly stated in UML 1.3.

A way to overcome this point may be to fully encapsulate the attributes within the operations, i.e., an attribute of a class may be read and updated only by the class operations. As a consequence, the expressions and the actions appearing in the associated state machine may use only the object attributes, and not those of other objects. Here we take this choice, because we think that any reasonable software engineering methodology will assume it.

Then an *L*-label, which formalizes the interactions happening during an *L*-transition, will be a triple consisting of a set of input interactions, the received time, and a set of output interactions.

3.5 Determining the *L*-States

The *L*-states (states of the lts *L*) describe the intermediate relevant situations in the life of the objects of class ACL.

On the basis of UML 1.3 we found that to decide what an object has to do in a given situation we surely need to know:

- the object identity;
- the set of the states (of the state machine SM) that are active in such situation;
- whether the threads of the object are in some run-to-completion steps, and in such case which are the states that will become active at the end of such step, each one accompanied by the actions to be performed to reach it;
- the values of object attributes;
- the status of the event queue.

Thus the *L*-states must contain such information; successively, when defining the transitions we discovered that, to handle change and time events, we need also to know

- some information, named *history* in the following, on the past behaviour of the object, precisely the previous values of the attributes and the times when the various states became active.

The *L*-states are thus specified by the following CASL specification

```
spec L-STATE =
    IDENT and CONFIGURATION and ATTRIBUTES and EVENT_QUEUE and HISTORY
    then free types
        State ::= Ident : ⟨Configuration, Attributes, History, Event_Queue⟩ |
                  Ident : terminated
        ......
```

where *Ident* : *terminated* are special elements representing terminated objects.

A configuration contains the set of the states that are active in a situation and of those states that will become active at the end of the current run-to-completion step (if any), the latter are accompanied by the actions to be performed to reach such states.

```
spec EVENT_QUEUE =
    SET[EVENT] then
    sort Event_Queue
    preds   no_dispatchable_event : Event_Queue
    %% checks whether there is no dispatchable event
                __ ∈ __ : Event × Event_Queue
    %% checks whether a given event in the queue may be selected for dispatching
    ops put : Bag[Event] | Event_Queue → Event_Queue
    %% adds some events to the queue
        remove : Event | Event_Queue → Event_Queue
    %% removes an event from the queue
        ......
```

PROBLEM UML 1.3 explicitly calls the above structure a queue, but it also clearly states that no order must be put on the queued events (UML 1.3 p. 2-144) and so the real structure should be a multiset. This choice of terminology is problematic, because it can induce a user to assume that some order on the received events will be preserved.
The fact that the event queue is just a bag causes other problems: an event may remain forever in the queue; time and change events may be dispatched disregarding the order in which happened (e.g., "`after` 10" dispatched before "`after` 5"); a change event is dispatched when its condition is false again; two signal or call events generated by the same state machine in some order are dispatched in the reverse order.

To fix this point, we can either change the name of the event queue in the UML documentation in something recalling its true semantics, or define a policy for deciding which event to dispatch first.

In a UML model we cannot assume anything on the order some events are received by an object (as the signal and operation calls); we conjecture that this was the motivation for avoiding to order the events in the queue. However, we think that it is better to have a mechanism ensuring that when two events are received in some order they will be dispatched in the same order, even if in many cases we do not know such order. Here, we make the most general choice, thus the event queue is just a multiset (bag) of events.

The specifications of the other components of the *L*-states are reported in [11].

3.6 Determining the *L*-Transitions

An *L*-transition, i.e., a transition of the lts *L*, corresponds to

1. either to dispatch an event,
2. or to execute an action,
3. or to receive some events; such events are either received from outside (signals and operation calls) or generated locally (self sent signals and operation calls, change and time events),
4. or to be destroyed by dispatching a special event.

Moreover, (3) may be also performed simultaneously with (1) and (2), because we cannot delay the reception of events.

It is important to notice that the *L*-transitions and the transitions of the state machine SM are different in nature and are not in a bijective correspondence. To clarify such relationship we report in Fig. 2 a fragment of the transition tree (see Sect. 3.1) associated with the simple state machine of Fig. 1 showing only the relevant parts of the states, where it is possible to see that one state machine transition corresponds to many *L*-transitions.

The *L*-transitions are formally defined by the axioms of an algebraic specification of the following form

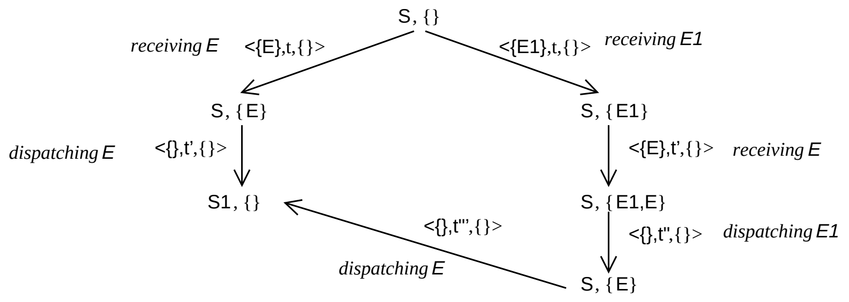


Fig. 2. A fragment of a transition tree

spec L-SPEC =
  L-LABEL **and** L-STATE **then**
free { **pred** $\_ \xrightarrow{\_} \_ : State \times Label \times State$
  axioms $\ldots\ldots cond \Rightarrow s \xrightarrow{l} s' \quad \ldots\ldots$

In the following subsections we give the axioms corresponding to the four cases above To master complexity and to improve readability we use several auxiliary operations in such axioms, whose name is written in sans serif font. Some relevant problems come to light when defining some of such operations, and thus we consider them explicitly in Sect. 4. The others are reported in [11].

Dispatching an Event If the active object is not fully frozen executing some run-to-completion steps (checked by *not_frozen*), an event *ev* is in the event queue ready to be dispatched (checked by *dispatchable*), then there is an *L*-transition, with the label made by the events received from outside *in_evs*, and the time *t*, where the history was extended, and all received events was put in the event queue, as described by Receive_Events.

Recall that Dispatch(*ev*, *conf*, *e_queue*) = *conf₁*, *e_queue₁* means that dispatching event *ev* in the configuration *conf* changes it to *conf₁* and changes *e_queue* to *e_queue₁*.

$$not\_frozen(conf) \ \wedge \ dispatchable(ev, e\_queue) \ \wedge$$
$$\mathsf{Dispatch}(ev, conf, e\_queue) = conf_1, e\_queue_1 \ \Rightarrow$$
$$id : \langle conf, attrs, history, e\_queue\rangle \xrightarrow{\langle in\_evs, t, \emptyset\rangle} id : \langle attrs, conf_1, history_1, e\_queue_2\rangle$$

where

$$e\_queue_2 = \mathsf{Receive\_Events}(e\_queue_1, in\_evs, attrs, history, t)$$
$$history_1 = \langle active\_states(conf), attrs, \rangle, t\rangle \ \& \ history$$

Executing an Action If the active object is executing at least a run-to-completion step, (checked by *frozen*), then there is an *L*-transition, with the label resulting from the events received from outside *in_evs*, the time *t*, and the set of events generated in the action to be propagated outside *out_evs*, where the attributes are updated due to executed action, the history was extended, and all received events was put in the event queue, as described by Receive_Events.

$\mathsf{Exec}(id, attrs, conf) = conf_1, attrs_1, out\_evs, loc\_evs$ means that the object id with configuration $conf$ executes an action changing its configuration to $conf_1$, updating its attributes to $attrs_1$ and producing the set of output events out_evs and the set of local events loc_evs.

$$frozen(conf) \ \wedge \ \mathsf{Exec}(id, attrs, conf) = conf_1, attrs_1, out\_evs, loc\_evs \ \Rightarrow$$
$$id : \langle conf, attrs, history, e\_queue\rangle \xrightarrow{\langle in\_evs, t, out\_evs\rangle} id : \langle attrs_1, conf_1, history_1, e\_queue_1\rangle$$

where
$e\_queue_1 = \mathsf{Receive\_Events}(e\_queue, in\_evs \cup loc\_evs, attrs, history, t)$
$history_1 = \langle active\_states(conf), attrs, \rangle, t\rangle \ \& \ history$

Receiving Some Events If the active object is not executing any run-to-completion step (checked by $\neg$ *frozen*), the event queue is empty (checked by *no_dispatchable_event*), then there is an *L*-transition, with the label resulting from the set of events received from outside *in_evs* and the time *t*, where the history was extended, and all received events was put in the event queue, as described by Receive_Events.

$$\neg\, frozen(conf) \ \wedge \ no\_dispatchable\_event(e\_queue) \ \Rightarrow$$
$$id : \langle conf, attrs, history, e\_queue\rangle \xrightarrow{\langle in\_evs, t, \emptyset\rangle} id : \langle attrs, conf, history_1, e\_queue_1\rangle$$
where
$e\_queue_1 = \mathsf{Receive\_Events}(e\_queue, in\_evs, attrs, history, t)$
$history_1 = \langle active\_states(conf), attrs, \rangle, t\rangle \ \& \ history$

Being Destroyed We consider a destruction request as an event and assume that an active object cannot be destroyed while it is fully frozen executing run-to-completion steps (checked by *not_frozen*).

$$not\_frozen(conf) \ \wedge \ dispatchable(destroy, e\_queue) \ \Rightarrow$$
$$id : \langle conf, attrs, history, e\_queue\rangle \xrightarrow{\langle in\_evs, t, \emptyset\rangle} id : terminated$$

3.7 Constraints

Constraints may be attached to any element of a UML model. For what concerns the fragment of UML considered in this paper we have constraints in the class diagram, attached to the class icon (e.g., invariants) and to the operations (e.g., pre-post conditions), and in the state machine (e.g., invariants). The language for expressing the constraints is not fixed in UML (though OCL is the most used), however the semantics of the constraints is precisely settled; indeed UML 1.3 p. 2-29,2-30 states:

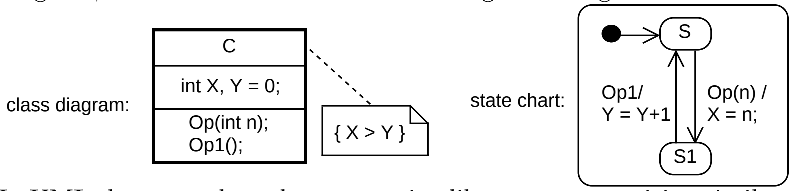
A constraint is a semantic condition or restriction expressed in text. In the metamodel, a Constraint is a BooleanExpression on an associated ModelElement(s) which must be true for the model to be well formed. . . . Note that a Constraint is an assertion, not an executable mechanism. It indicates a restriction that must be enforced by correct design of a system.

Such an idea of constraint may be easily formalized in our setting: the semantics of a constraint attached either to ACL or to SM is a property on L.

The CASL formulae allow to express a rich set of relevant properties (recall that the underlying logic of CASL is many sorted partial first-order logic), and CASL extensions with temporal logic combinators, based on [4] are under development. Moreover the constraints expressed using OCL may be translated into CASL without too many problems.

Assume we have the constraints $C_1, \dots, C_n$ attached either to ACL or to SM, then the UML model containing ACL and SM is well formed iff for $i = 1, \dots, n$, $L \models \Phi_i$, where Φ_i is the CASL formula corresponding to C_i. Techniques and tools developed for algebraic specification languages may help to verify the well-formedness of UML models.

PROBLEM The use of constraints without a proper discipline may be problematic, as in the following case, where the constraint attached to the icon of class C is an invariant that must hold always, and so must be respected also by the transitions triggered by the calls to OP and OP1. These inconsistencies may be hard to detect, because the problematic constraints are in the class diagram, while the state machine violating them is given elsewhere.



In UML there are also other constraint-like constructs raising similar problems; as the query qualification for operations (requiring that an operation does not modify the state of the object), or the "specifications" for signal receptions (expressing properties on the effects of receiving such signal). Also in these cases the behaviour described by the state machine may be in contrast with them.

We think that a way to settle those problems is to develop a clear methodology for using these constraints, making precise their role and their use them in the development process.

4 Auxiliary Functions

Receive_Events : *Event_Queue* × *Set*[*Event*] × *Attributes* × *History* × *Time* → *Event_Queue*

Receive_Events(*e_queue*, *evs*, *attrs*, *history*, *t*) = *e_queue*$_1$ means that *e_queue*$_1$ is *e_queue* updated by putting in it all received events: the signal and operation call events are given by a function parameter (*evs*), the time events are detected using *t* and *history* (by TimeOccur), and the change events are detected by using *attrs* and *history* (by ChangeOccur).

PROBLEM May operation calls to other objects appear within the expressions of change and time events? May such expressions have side effects ?
If the answer is yes, then we can have more hidden constraints on the mutual behaviour of objects (e.g., a synchronous operation call in the expression of a change event may block an object), and, due to the side effects, the behaviour of the whole system may be scarcely predictable. We assume no, and this is the reason for the above simple functionality of Receive_Events.

$\mathsf{Receive\_Events}(\mathit{e\_queue}, \mathit{evs}, \mathit{attrs}, \mathit{history}, t) =$

$\mathit{put}(\mathsf{TimeOccur}(t, \mathit{history}) \cup \mathsf{ChangeOccur}(\mathit{attrs}, \mathit{history}) \cup \mathit{evs}, \mathit{e\_queue})$

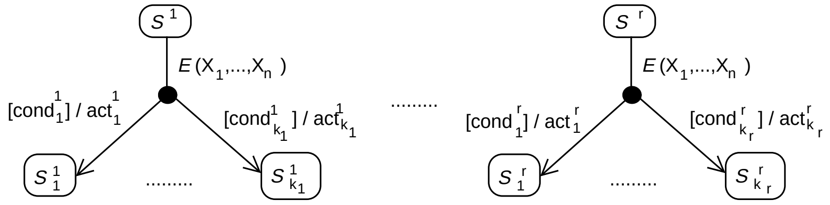
We report the definitions of TimeOccur and of ChangeOccur in [11].

Dispatch : *Event* $\times$ *Configuration* $\times$ *Event_Queue* $\longrightarrow$ *Configuration* $\times$ *Event_Queue*

$\mathsf{Dispatch}(\mathit{ev}, \mathit{conf}, \mathit{e\_queue}) = \mathit{conf}_1, \mathit{e\_queue}_1$ means that dispatching event *ev* in the configuration *conf* changes it to conf_1 and changes *e_queue* to $\mathit{e\_queue}_1$.

It is defined by cases, and here we report the most relevant ones, the others are reported in [11].

Some transitions triggered by an event Assume that in the state machine SM there are the following branched transitions triggered by E starting from the states belonging to *Sset*


S^1
$E(X_1,\ldots,X_n)$
$[\mathrm{cond}^1_1]$ / act^1_1
$[\mathrm{cond}^1_{k_1}]$ / $\mathrm{act}^1_{k_1}$
S^1_1
....
$S^1_{k_1}$
....
S^r
$E(X_1,\ldots,X_n)$
$[\mathrm{cond}^r_1]$ / act^r_1
$[\mathrm{cond}^r_{k_r}]$ / $\mathrm{act}^r_{k_r}$
S^r_1
....
$S^r_{k_r}$

If $\mathit{Sset} \cup \mathit{Sset'}$ is the set of the active states, and $\mathit{cond}^i_{q_i}$ holds for $i = 1, \ldots, h$ $(1 \leq h \leq r)$, then the active object may start a run-to-completion step going to perform the actions $\mathit{act}^i_{q_i}$ $(i = 1, \ldots, h)$ and to reach the states $S^i_{q_i}$ $(i = 1, \ldots, h)$ (the actual parameters of the event are substituted for its formal parameters within the actions to be performed). Notice that we do not require that $\mathit{cond}^i_{q_i}$ does not hold for $i = h + 1, \ldots, r$, $q = k^i_1, \ldots, k^i_{r_i}$.

$\mathit{active\_states}(\mathit{conf}) = \mathit{Sset} \cup \mathit{Sset'} \;\wedge\; {}^{h}_{i=1}\mathsf{Eval}(\mathit{cond}^i_{q_i}[p_j/x_j], \mathit{attrs}) = \mathit{True} \;\Rightarrow$

$\mathsf{Dispatch}(E(p_1, \ldots, p_n), \mathit{conf}, \mathit{e\_queue}) = \mathit{conf}_1, \mathit{remove}(E(p_1, \ldots, p_n), \mathit{e\_queue})$

where

$\mathit{conf}_1 = \mathit{run}(\ldots \mathit{run}(\mathit{conf}, S_i, \mathit{act}^1_{q_1}[p_j/x_j], S^1_{q_1}) \ldots, \mathit{act}^h_{q_h}[p_j/x_j], S^h_{q_h})$

active_states and *run* are operations of the specification Configuration returning respectively the active states of the state machine and recording the start of a run-to-completion step, going from an active state into another one performing a given action.

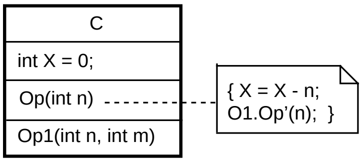
PROBLEM What to do if the conditions appearing on the transitions may have side effects, for example because they include operation calls ? In such case the order of evaluating the conditions and how many attempts have been done before to find those that hold are semantically relevant, and also in this case the behaviour of the whole system may become scarcely predictable.

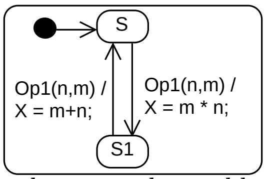
Here we assume that the conditions on the transitions have no side effects.

PROBLEM What to do when the dispatched event is an operation call for whom also a method has been defined in the class ACL?

The solution in this case is just to prohibit to have a method for the operation appearing in some transition, and it is supported, e.g., by [13] p. 369 and other UML sources ([12]).

PROBLEM A similar problem is posed by the case below: what will happen when someone calls method Op, whose body is described by the note attached to its name in the class diagram? On one hand, assuming that such call is never answered, may lead to produce wrong UML models, because the other classes assume that Op is an available service, since it appears in the class icon. On the other hand, answering it (perhaps only when the machine is not in a run-to-completion-step) seems in contrast with the role of the state machine (UML 1.3 p. 2-136).





In general operations with an associated method seem to be problematic for the active classes, and so it could be sensible to drop them.

No transitions triggered by a deferred event Assume that in the state machine SM there are no transitions starting from states belonging to *Sset* triggered by E and that E is deferred in some elements of *Sset*.

PROBLEM UML 1.3 does not say what to do when dispatching E in such case. The possible choices are:
– remove it from the event queue
– put it back in the event queue
– put it back in the event queue, but only for the states in which it was deferred.
Here we assume that it is deferred, and that after it will be available for any state; however, notice, that we could formally handle also the first cases, whereas to consider the last we need to assume that there are many threads in an object with the relative event queues (one for each active state).

If the active states are *Sset*, then the event $E(p_1, \ldots, p_n)$ is left in the event queue for future use.

$active\_states(conf) = Sset \Rightarrow$
$\mathsf{Dispatch}(E(p_1, \ldots, p_n), conf, e\_queue) = conf, e\_queue$

5 Conclusions and Related Work

The task of formalizing UML has been addressed using various available formal techniques, as we will discuss below. Most of these attempts are complementary, because they approach the task from different viewpoints and aims.

In this respect our work has several distinguishing features. First of all we are using a very elementary and well-known machinery, namely lts (which are at the basis of formalisms like CCS), and conditional algebraic specifications. As for Ada, [1], this simple model provides a powerful setting for revealing ambiguities, incompleteness and perhaps questionable features, both from the purely technical and the methodological point of view. For example, we have discussed several naturally arising questions concerning UML, such as "what is the behaviour of an object with several active subthreads?", and "does the run to completion execution of a state machine transition imply that a transition is a kind of critical region?"

Furthermore we have used a modular framework where the various possible interpretations of the aspects of UML (and of its many variants) may be made precise and the various problems exemplified and discussed also with users lacking a formal background (a lightweight formal method approach).

Within the "Precise UML" group and also outside, several proposals for formalizing UML or parts of them have been presented; we briefly report on some paradigmatic directions.

Some papers are addressing specific questions; for example [5] shows how to use graph rewriting techniques to transform UML state machines into another simplified machine (a kind of normal form); but, e.g., the execution of actions is not considered. That paper could be seen as providing a method for precisely handling the constructs not considered here, because can be derived from others.

Some other papers try to formalize UML by using a particular specification language. For example, as in [6], using Real-Time Action Logic, a form of real-time temporal logic. With respect to these approaches we put less emphasis on the specification language and more on the underlying model, because we think that in this setting it is easier to explain UML features (also the possible problems) and to derive a revised informal presentation. [7] also presents a formalization of the UML state machines using PROMELA, the specification language for the model checker SPIN.

The relevance of the underlying model for making precise UML has been considered in [2], where a different model, a kind of stream processing function, is used. But the main aim there is methodological: how a software engineering method can benefit from an integrative mathematical foundation. While one of the main results of our work could be a basis for a revised reference manual for UML.

Finally, we have not considered the large number of papers on the semantics of classical state machines (as those supported by the Statemate tool), because the semantics of the UML state machines is rather different; e.g., in the former the events are dispatched as soon as they are received, whereas they are enqueued (also forever) before being dispatched following some policy in UML (see [13] page 440).

We plan to go on analysing UML, to give a sound basis for our work in the CoFI project, trying to give an underlying formal (lightweight) model to the other constituents of a UML-system, i.e., to instances of passive classes and the system itself, and to consider the other kinds of diagrams, as class, sequence and collaboration.

The modular structure of our formalization of UML and the use of the algebraic specification CASL, allows us to easily modify the formalization of UML presented here to accommodate and to analyse particular choices for the variation points, and also the changes surely introduced by future versions.

This work could be also the basis for developing a "clean" profile for UML (see [9]), i.e., a variant of UML defined by extending it, using the various UML mechanisms as stereotypes, and by specializing its semantics, by means of natural language, without any of the problems presented here.

References

[1] E. Astesiano, A. Giovini, F. Mazzanti, G. Reggio, and E. Zucca. The Ada Challenge for New Formal Semantic Techniques. In *Ada: Managing the Transition, Proc. of the Ada-Europe Conference, Edimburgh, 1986*, pages 239–248. University Press, Cambridge, 1986.

[2] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.

[3] E. Coscia and G. Reggio. A Proposal for a Semantics of a Subset of Multi-Threaded Good Java Programs. Technical report, Imperial College - London, 1998.

[4] G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.

[5] M. Gogolla and F. Parisi-Presicce. State Diagrams in UML- A Formal Semantics using Graph Transformation. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *Proc. ICSE'98 Workshop on Precise Semantics of Modeling Techniques(PSMT'98)*, Technical Report TUM-I9803, 1998.

[6] K. Lano and J. Bicarregui. Formalising the UML in Structured Temporal Theories. In B. Rumpe H. Kilov, editor, *Proc. of Second ECOOP Workshop on Precise Behavioral Semantics, ECOOP'98*, Munich, Germany, 1998.

[7] J. Lillius and I Paltor. Formalising UML State Machines for Model Checking. In R France and B. Rumpe, editors, *Proc. UML'99*, LNCS. Springer Verlag, Berlin, 1999.

[8] P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in LNCS, pages 115–137, Berlin, 1997. Springer Verlag.

[9] OMG. White paper on the Profile Mechanism – Version 1.0. `http://uml.shl.com/u2wg/default.htm`, 1999.

[10] The CoFI Task Group on Language Design. CASL Summary. Version 1.0. Technical report, 1998. Available on `http://www.brics.dk/Projects/CoFI/Documents//Summary/`.

[11] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. A CASL Formal Definition of UML Active Classes and Associated State Machines. Technical Report DISI-TR-99-16, DISI – Università di Genova, Italy, 1999.
[12] J. Rumbaugh. Some questions relating to actions and their parameter, and relating to signals. Private communication, 1999.
[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.
[14] UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.