A CASL Formal Definition of UML Active Classes and Associated State Machines

G. Reggio(1) – E. Astesiano(1) – C. Choppy(2) – H. Hussmann(3)

(1) DISI Università di Genova - Italy

(2) LIPN, Institut Galilee - Universite Paris XIII, France

(3) Department of Computer Science, Dresden University of Technology - Germany

Abstract

We consider the problem of precisely defining UML active classes with an associated state chart. We are convinced that the first step to make UML precise is to find an underlying formal model for the systems modelled by UML. We argue that labelled transition systems are a sensible choice; indeed they have worked quite successfully for languages as Ada and Java. Moreover, we think that this modelization will help to understand the UML constructs and to improve their use in practice. Here we present the labelled transition system associated with an active class using the algebraic specification language CASL.

The task of making precise this fragment of UML raises many questions about both the "precise" meaning of some constructs and the soundness of some allowed combination of constructs.

1 Introduction

The Unified Modeling Language (UML) [10] 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.

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 [4], 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. That would allow for instance: – to specify user-defined data types that just have values but do not behave like objects; – to use algebraic axioms as a constraints. These new constraints may cover also behavioural aspects, because there exist extensions of algebraic languages that are able to cover with them [3], whereas OCL does not seem to have any support for concurrency. 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 understanding (formally defining) the most relevant UML features. In this paper we present the results of such formalization work, 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 or idealizations). We are not going to consider a small OO language with a UML syntax, but just what it is presented in the official OMG documentation [11] (shortly UML 1.3 from now on). [1]

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 [2].

Lightweight formalization By "lightweight" we mean made by using the most simple formal tools and techniques: precisely labelled transition systems algebraically specified using a small subset of the specification language CASL (conditional specification with initial semantics).

Integrated with the formalization of the other fragments of UML In contrast to many other papers on UML semantics, we more or less ignore the issue of class diagram semantics here and concentrate on the state machines[2]. However, 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. For this reason, we are using an algebraic approach to the semantics of state machines, since it is well known that class diagrams can be mapped relatively easily onto algebraic specifications. The algebraic semantics described here enables an algebraic access to the semantics also of active, not only of passive, classes. Usually, an active class is statically

[1] To be precise version 1.3 it is not still officially approved, but it is the latest.

[2] 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.

described in the class diagram and dynamically described in an associated statechart diagram. As an example of this intention, see, in Sect. 11, how we can extend without problem our formalization of active objects to consider constraints.

The formalization of active classes and state machines has lead to perform a thorough analysis of them 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 is clearly formulated but the allowed usages seem problematic from a methodological point of view). To stress this aspect and to help the reader we have used the mark pattern **PROBLEM** to highlight them.

In Sect. 2 we define the subset of the active classes with state machines that we consider. Then in Sect. 3 and 9 we introduce the used formal techniques (labelled transition systems and algebraic specifications), and present step after step how we have built the lts modelling the objects of an active class with an associated state machine. Due to lack of room part of the definition is reported in Appendix **??**, while the complete formal model (rather short and simple) is in [7].

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!

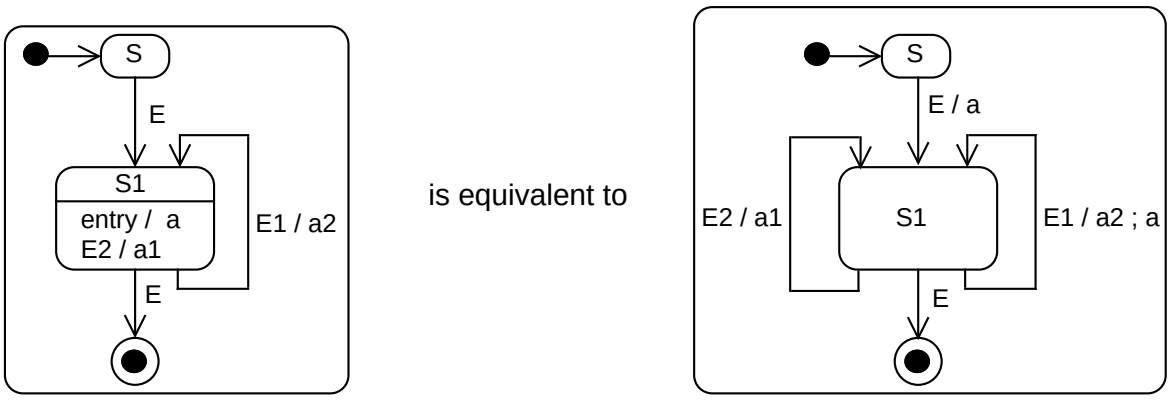
Shortcuts We do not consider the following UML state machine constructs, because they can be replaced by equivalent combinations of other constructs.

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., [10] p. 266, but semantically they are not relevant; indeed we can eliminate them by adding such actions to all transitions entering/ leaving such state.

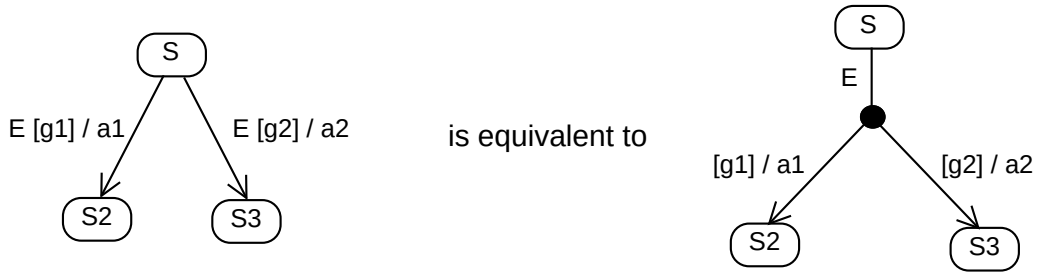
Internal transitions An internal transition differs from a transition whose source and target state coincide only for what concerns entry/exit actions. Because we have dropped entry/ exit actions, we can drop also internal transitions.

The following picture shows, on an example, how to eliminate entry actions and internal transitions.

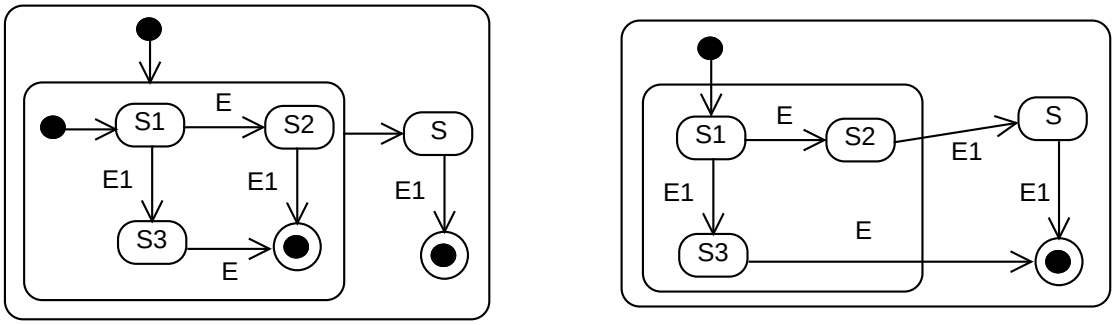


Different transitions leaving the same state having the same event trigger We can replace these transitions with compound transitions using junction states. The latter presentation seems better from a methodological point of view, and makes the semantics

clearer (what to do when dispatching an event is explained in UML 1.3 by considering the transitions with the same trigger leaving a state all together). The picture below shows on an example of this simplification.



Multiple initial/final states We assume that there is always one unique initial state at the top level (used to determine the initial situation of the class objects) and one unique final state at the top level (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.



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

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 could be modelled without much trouble.

- 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

A *labelled 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,
- where two identically decorated subtrees with the same root are considered as a unique subtree,
- and such that 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 that here 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 "sequential process", indeed also concurrent processes, which are processes having cooperating components that are in turn other processes (concurrent or not), can be modelled through particular lts, named *structured lts*. A structured lts is obtained by composing other lts describing such components, say *clts*; its states are built by the states of *clts*, and its transitions are determined by composing those of *clts*.

An lts may be formally specified by using the algebraic specification language CASL-LTL [6] an extension of CASL [5] for the specification of processes based on lts's with a specification of the following form. Recall that extension in this case means that CASL is a subset of CASL-LTL and that any CASL specification is also a CASL-LTL specification.

```
spec LTS =
    ..... then
free {
    dsort State
    .....
    axioms
    .....
} end
```

The CASL-LTL construct "**dsort** *State*" declares a sort *State*, for the states of the lts, and implicitly also a sort for the labels, *Label_State*, and a transition predicate

$\_\_ \dashrightarrow \_\_ : State \times Label\_State \times State$.
Thus each element s of sort *State* in a model M (an algebra or first-order structure) of the above specification LTS corresponds to a process modelled by a transition tree with initial state s determined by the lts $(State^M, Label\_State^M, \_\_ \dashrightarrow \_\_^M)$ [3].

The specification LTS extends (CASL-LTL keyword **then**) some specifications of some basic data used to define the states and the labels. The construct **free** requires, instead, that the specification has an initial semantics. Moreover, the axioms of such specification must 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), to guarantee the existence of the initial semantics.

Assume to have a given active class ACL with a given 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. The instances of ACL, called using a UML terminology *active objects*, are just processes and we model them by using an lts. We build such lts, named in the following L, and specify it algebraically with the specification L-SPEC, following the steps below, which will be reported in the following sections.

1. check whether L is simple or structured
2. if L is structured, then determine its components and the lts's modelling them
3. determine the grain of the transitions of L
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-LTL, by means of conditional axioms).

The constraints attached either to ACL or to SM are treated apart in Sect. 11, 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.

4 Basic Data Type Specifications

We report in this section the CASL specifications of the basic data types that will be used in the specification of L.

- FINITESET is the parametric specification of finite sets with the usual operations and predicates

[3] Given a Σ algebra A, and a sort s of Σ, s^A denotes the interpretation of s in A; similarly for the operation and predicates of Σ.

- LIST is the parametric specification of lists with the usual operations and predicates
- BAG is the parametric specification of finite bags with the usual operations and predicates
- FINITEMAP is the parametric specification, with arguments S and T, of finite maps from elements of sort s into elements of sort t, with the usual operations and predicates

The above parametric specifications are provided by the CASL standard libraries [8].

- VALUE is a specification of the UML values (datavalues and object identities)
- TIME is a specification of the time values
- NAME is a specification of the UML names (those used to identify attributes, classes, operations, signals, ...), see UML 1.3 p. 2-75
- IDENT is a specification of the identities of the UML objects
- EXP is a specification of the UML expressions (i.e., the elements of the meta class Expression, see UML 1.3 p. 2-75)

We do not report a detailed definition of the above specifications, because such data are not precisely defined in the UML metamodel [11].

- ACTION is a specification of the UML actions see UML 1.3 p. 2-86; as an example of uninterpreted action we consider the assignment, while the dots stands for further uninterpreted actions.

spec ACTION =
  IDENT **and** NAME **and** LIST[EXP] **then**
  free type *Action* ::=
    scall(*Ident*, *Name*, *List*[*Expression*]) | *acall*(*Ident*, *Name*, *List*[*Expression*]) |
    send(*Name*, *List*[*Expression*]) | *create*(*Name*) | *destroy*(*Ident*) |
    __._ = _(*Ident*; *Name*; *Expression*) | ...

- EVENT is a specification of the events

spec EVENT =
  NAME **and** IDENT **and** LIST[VALUE] **and** EXP **then**
  free type *Event* ::=
    call(*Ident*; *Name*; *List*[*Value*]) | *signal*(*Name*; *List*[*Value*]) |
  %% operation call and signal events
    destroy(*Ident*) |
  %% special events to handle object destruction
    after __(*Expression*) | *after* __ *since* __(*Expression*; *Name*) |
  %% time events
    change __(*Expression*)
  %% change events
end

5 Is L simple or structured?

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

PROBLEM Does an active object correspond to a single thread of control (running concurrently with the others), 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. We have thus two possibilities, consider L to be

- a structured lts, where each component correspond to a thread, but in this case we should define several L, one for each choice of the number of the existing threads;
- a simple lts, where each of its transitions corresponds to one of the possible existing threads that executes part of its activity, in this case a unique L will be sufficient.

Here, we take the second choice.

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.

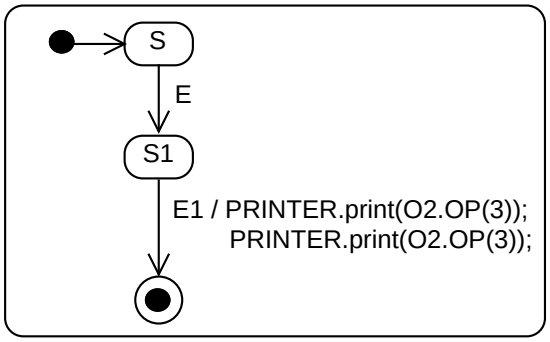


Figure 1: A simple State Machine

6 Determining the granularity of the L-transitions

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

PROBLEM By looking at UML 1.3 we see that there are two possibilities corresponding to two 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 groups of transitions of SM, are 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.

7 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 [10] 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

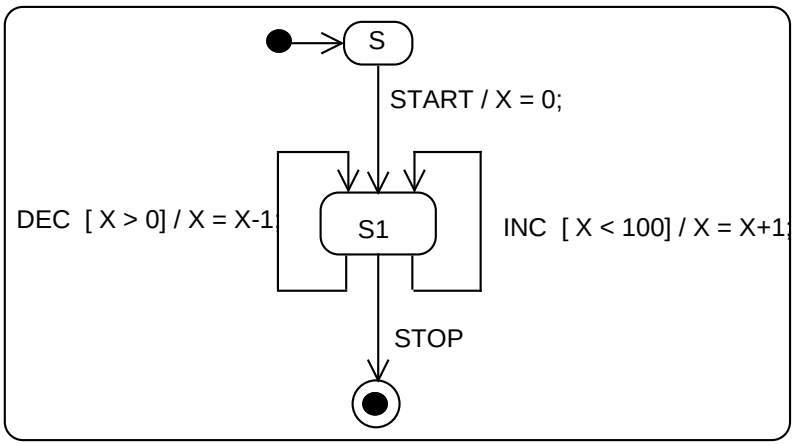
However, UML 1.3 does not consider explicitly the interactions marked by (+), which do not correspond to exchange a stimulus, and does not say anything about when they can be performed (e.g., they are not considered by the state machines).

Object creation and destruction correspond to stimuli (and so are considered in sequence and interaction diagrams) but do not correspond to events, thus they do not appear in the state machine.

PROBLEM When may an object be destroyed?

A way to settle this point is to make "to be destroyed" an event, which thus can be dispatched when the machine is not in a run-to-completion-step and can 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 updates 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 OR while OR 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 UML 1.3 p. 2-136, because another object updates its attributes?
In the following example, another object may perform OR.X = OR.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 the problem is about active classes, but notice that there may be similar problems also for passive classes, since the concurrency properties of operations do not consider updates.

In this paper also for this point we consider the most general choice, and thus we take into account all possible interactions listed before.

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. This choice is sound because the time is received at any step.

spec INPUT =
IDENT **and** NAME **and** LIST[VALUE] **then**
free type *Input* ::=
I_scall(*Ident*; *Name*; *List*[*Value*]) | *I_acall*(*Ident*; *Name*; *List*[*Value*]) |

$I\_return(Ident; Name; Value) \mid I\_send(Ident; Name; List[Value]) \mid$
$I\_destroy(Ident; Name) \mid$
$I\_read(Ident; Name; Value) \mid I\_update(Ident; Name, Value)$

spec OUTPUT =
IDENT **and** NAME **and** LIST[VALUE] **then**
free type $Output ::=$
$O\_scall(Ident; Name; List[Value]) \mid O\_acall(Ident; Name; List[Value]) \mid$
$O\_return(Ident; Name; Value) \mid O\_send(Ident; Name; List[Value]) \mid$
$O\_create(Ident; Name) \mid O\_destroy(Ident; Name) \mid$
$O\_read(Ident; Name; Value) \mid O\_update(Ident; Name, Value)$

spec L-LABEL =
TIME **and** FINITESET[INPUT] **and** FINITESET[OUTPUT] **then**
free type $Label\_State ::= \langle \_, \_, \_ \rangle (FinSet[Input]; Time; FinSet[Output])$

8 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 (names of the) states (of the state machine SM) that are active in such situation;
- whether the (threads of the) object is (are) in some run-to-completion steps, and in such case which are the states that will become active at the end of such steps, 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 at least such information; successively, when defining the transitions we discovered that, to detect event occurrences, we need also to know:

- some information, named *history* in the following, on the past behaviour of the object, precisely the times when the various states of the state machine SM became active;
- the previous values of the expressions appearing in the change events of the state machine SM.

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

spec *L*-STATE =
IDENT **and** CONFIGURATION **and** ATTRIBUTES **and** EVENT_QUEUE **and**
HISTORY **and** CHANGEINFO **then**
free type *State* ::=
$\_\_ : \langle \_\ \_\ \_\ \_\ \_ \rangle (Ident; Configuration; Attributes; Event\_Queue; History; ChangeInfo) \mid$
$\_\_ : terminated(Ident)$

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

A configuration contains the set of the states of the state machine 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. Recall that the states of a state machine are identified by their names, as specified by the specification NAME.

spec CONFIGURATION =
FINITESET[NAME] **and** ACTION **then**
free {
type *Configuration* ::=
$null \mid active(\_)\&\_(Name; Configuration) \mid rtc\_step(\_,\_)\&\_(Action; Name; Configuration)$
preds $executing : Configuration$
%% checks whether the object is executing at least a run-to-completion step
$not\_frozen : Configuration$
%% checks whether the object is not fully frozen executing run-to-completion steps
$to\_execute : Configuration \times Action \times Name$
%% checks whether in a configuration there is an action to be executed going into a state of
%% the state machine
ops $active\_states : Configuration \to FinSet[Name]$
%% returns the set of the states of the state machine that are active in such configuration
$run : Configuration \times Name \times Action \times Name \to Configuration$
%% records the start of a run-to-completion step, going from an active state into another
%% state performing an action
$execute : Configuration \times Name \times Action \to Configuration$
%% records that an action reaching some state has been executed
axioms
$\neg\ executing(null)$
$executing(conf) \ \Rightarrow\ executing(active(S)\&conf)$
$executing(rtc\_step(act, S)\&conf)$
$not\_frozen(null)$
$not\_frozen(conf) \ \Rightarrow\ not\_frozen(rtc\_step(act, S)\&conf)$
$not\_frozen(active(S)\&conf)$
$\neg\ to\_execute(null, act, S)$
$to\_execute(conf, act, S) \ \Rightarrow\ to\_execute(active(S')\&conf, act, S)$
$to\_execute(rtc\_step(act, S)\&conf, act, S)$
$to\_execute(conf, act, S) \ \Rightarrow\ to\_execute(rtc\_step(act', S')\&conf, act, S)$
.....
} end

spec ATTRIBUTES =
FINITEMAP[NAME][VALUE] **with** $FiniteMap \mapsto Attributes$ **then**

free {
op $\_\_[\_\_/\_]$: $Attributes \times Value \times Name \to Attributes$ %% sets the value of an attribute
vars $v : Value$; $x : Name$; $attrs : Attributes$;
- $attrs[v/x] = add(x, v, attrs)$

} end

spec EVENT_QUEUE =
BAG[EVENT] **and** (BAG[EVENT] **with** $Bag[Event] \mapsto Event\_Queue$) **then**
free {
preds $no\_dispatchable\_event : Event\_Queue$
%% checks whether there is no event in the queue that may be selected for dispatching
$dispatchable : Event \times Event\_Queue$
%% checks whether an event in the queue may be selected for dispatching
ops $put : Bag[Event] \times Event\_Queue \to Event\_Queue$
%% adds some events to the queue
$remove : Event \times Event\_Queue \to Event\_Queue$
%% removes an event from the queue
vars
- $no\_dispatchable\_event(empty)$
- $dispatchable(ev, e\_queue) \Leftrightarrow ev\ elemOf\ e\_queue$
- $put(empty, e\_queue) = e\_queue$
- $put(ev + evs, e\_queue) = put(evs, ev + e\_queue)$
- $remove(ev, empty) = empty$
- $remove(ev, ev + e\_queue) = remove(ev, e\_queue)$
- $\neg\ ev = ev' \Rightarrow remove(ev', ev + e\_queue) = ev + remove(ev', e\_queue)$

} end

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.

Our formal model allows to easily consider various policies for handling the event queue, indeed it is sufficient to change the definitions of the operations of specification EVENT_QUEUE.

For example, some precedences over the dispatching of different events may be formalized by *dispatchable* (if $dispatchable(ev, e\_queue)$ and ev has not been deferred, then there are no deferred events in e_queue, corresponds to the case when deferred events take precedence over the others).

Here, we make the most general choice, thus the event queue is just a multiset (bag) of events.

spec HISTORY =
FINITESET[NAME] **then**
free {
type $History ::= \Lambda \mid \langle \_, \_ \rangle \ \& \ \_(FinSet[Name]; Time; History)$
op $entered\_state : History \times Name \to Time$
%% returns the time at whom a state (of the state machine) became active
axioms
$S \in Sset \ \wedge \ \neg\, S \in Sset' \ \Rightarrow$
$entered\_state(\langle Sset, t \rangle \ \& \ \langle Sset', t' \rangle \ \& \ history, S) = t$
$S \in Sset \ \wedge \ S \in Sset' \ \Rightarrow$
$entered\_state(\langle Sset, t \rangle \ \& \ \langle Sset', t' \rangle \ \& \ history, S) = entered\_state(\langle Sset', t' \rangle \ \& \ history, S)$
$\neg\, S \in Sset \ \Rightarrow$
$entered\_state(\langle Sset, t \rangle \ \& \ history, S) = entered\_state(history, S)$
$entered\_state(\Lambda, S) = 0$
} end

spec CHANGEINFO =
FINITEMAP[EXP][VALUE] **with** $FiniteMap \mapsto ChangeInfo$ **then**
free {
op $\_\_[\_\_/\_\_] : ChangeInfo \times Expression \times Value \to ChangeInfo$
%% sets the value of an expression appearing in a change event of the state machine
vars $v : Value$; $exp : Expression$; $chinf : ChangeInfo$;
• $chinf[v/exp] = add(exp, v, chinf)$
} end

9 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 be destroyed by dispatching a special event,
4. or to receive some inputs, to have the attributes read from other objects and to raise the time and change events.

Moreover, (4) may be also performed simultaneously to (1), (2) and (3), because we cannot delay the reception of inputs, the access of the attributes from other objects, and the raising of time and change events.

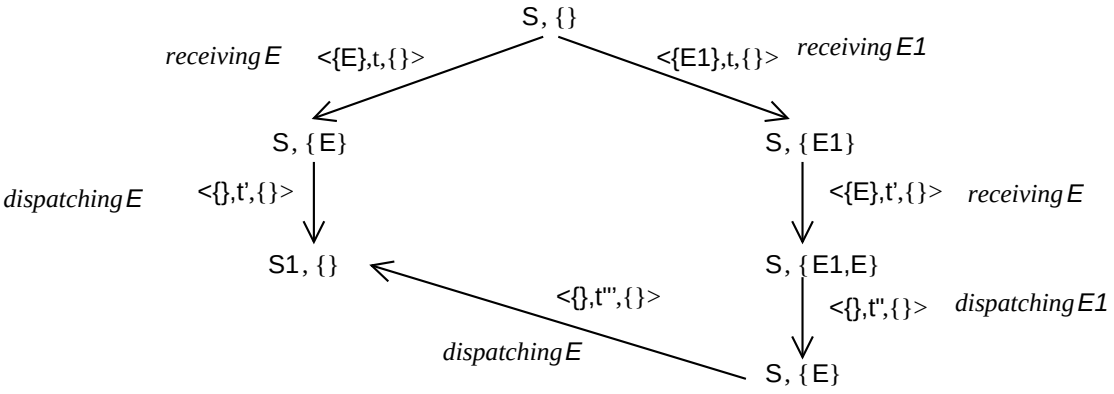


Figure 2: A fragment of a transition tree

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 partly report in Fig. 2 the transition tree associated with the simple state machine of Fig. 1 (to simplify the picture we only report the configuration and the event queue of each L-state); there 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

spec L-SPEC =
  L-LABEL **and** L-STATE **then**
free {
  dsort *State*
  %% recall that because *State* is a dynamic sort, we have also the implicitly declared
  %% transition predicate $\_\_ \Longrightarrow \_\_ : State \times Label\_State \times State$
  axioms
    .....
    $cond \Rightarrow s \xrightarrow{l} s'$
    .....
} end

In the following subsections we give the axioms corresponding to the four cases above. To master complexity and to improve readability and to offer a modular set for easily handling different interpretations of the official (but informal) semantics of UML and of its various versions, we use in such axioms several auxiliary operations, whose name is written in sans serif font, which are reported in Sect. 10.

Notice that while the specifications of the states and of the labels are the same for any class ACL and any state machine SM, the axioms of the above specification differ instead for each pair.

9.1 Dispatching an Event

Comment

If the active object is not fully frozen executing run-to-completion steps (checked by *not_frozen*), an event *ev* is ready to be dispatched (checked by *dispatchable*), then there

is an L-transition, with the label made by the received inputs ins, the time t, and the sent outputs $outs$ (all of them corresponding to have an attribute read, that is checked by Read_Attributes), where the history has been extended with the current states, the received inputs have modified the attribute status, as described by Updates, and the events generated by the inputs received from outside (Events_Of(ins)) plus the raised time and change events (defined by TimeEvs and ChangeEvs) have been added to the event queue.

Notice that Updates returns a description of how to modify the attributes, called here *effect*, and not directly the new attribute status; that is needed to allow to compose the attribute transformations.

Dispatch$(ev, conf, e_queue, ins, attrs, id) = conf', e_queue'$ means that dispatching event ev in the configuration $conf$ changes it to $conf'$ and changes e_queue to e_queue'.

- $not\_frozen(conf) \wedge dispatchable(ev, e\_queue) \wedge$ Read_Attributes$(outs, attrs) \wedge$
  Dispatch$(ev, conf, e_queue, ins, attrs, id) = conf', e_queue' \wedge$
  ChangeEvs$(ins, attrs, id, chinf) = ch\text{-}evs, chinf' \Rightarrow$
  $id : \langle conf, attrs, e\_queue, history, chinf\rangle \xrightarrow{\langle ins, t, outs\rangle} id : \langle attrs', conf', e\_queue'', history', chinf'\rangle$

where $attrs' = apply($Updates$(ins), attrs)$
$history' = \langle active\_states(conf), t\rangle \,\&\, history$
$e_queue'' = put($TimeEvs$(attrs, id, history, t) \cup ch\text{-}evs \cup$ Events_Of$(ins), e_queue')$

9.2 Executing an action

Comment

If the active object is executing at least a run-to-completion step (checked by *executing*), then there is an L-transition, with the label resulting from the received inputs ins, the time t, and the sent outputs (those corresponding to have an attribute read $outs$ [checked by Read_Attributes] and those to the events generated by the executed action to be propagated outside out_evs), where the attributes may be updated due to executed action and to the received inputs (as described by Updates), the history has been extended with the current states, and the events generated by the inputs received from outside (Events_Of(ins)) plus the raised time and change events (defined by TimeEvs and ChangeEvs) have been added to the event queue.

Exec$(conf, ins, attrs, id) = conf', eff', loc_evs, outs$ means that the active object id with configuration $conf$ may execute an action changing its configuration to $conf'$, updating its attributes as described by eff' and producing the set of local events loc_evs and the set of outputs $outs$.

- $executing(conf) \wedge$ Read_Attributes$(outs, attrs) \wedge$
  Exec$(conf, ins, attrs, id) = conf', eff, loc_evs, outs \wedge eff'' \; is_comp_of(eff,$ Updates$(ins)) \wedge$
  ChangeEvs$(ins, attrs, id, chinf) = ch\text{-}evs, chinf' \Rightarrow$
  $id : \langle conf, attrs, e\_queue, history, chinf\rangle \xrightarrow{\langle ins, t, outs'\rangle} id : \langle attrs', conf', e\_queue', history', chinf'\rangle$

where $outs' = outs \cup out\_evs$

$attrs' = apply(eff'', attrs)$

$history' = \langle active\_states(conf), t\rangle \;\&\; history$

$e\_queue' = put(\mathsf{TimeEvs}(attrs, id, history, t) \cup ch\text{-}evs \cup \mathsf{Events\_Of}(ins) \cup loc\_evs, e\_queue)$

9.3 Being Destroyed

Comment

We consider a destruction request as an event and assume that an active object cannot be destroyed while it is frozen in a run-to-completion step.

- $not\_frozen(conf) \;\wedge\; dispatchable(destroy(id), e\_queue) \;\Rightarrow$

  $id : \langle conf, attrs, e\_queue, history, chinf\rangle \xrightarrow{\langle ins, t, \{\}\rangle} id : terminated$

Recall that the predicate *dispatchable* is taking care of the precedence on dispatching events; so if we assume that $destroy(id)$ takes precedence over any other event, then whenever $destroy(id)$ is in the queue $dispatchable(ev, e\_queue)$ for any ev s.t. $ev \neq destroy(id)$.

9.4 Receiving Some Inputs, Having the Attributes Read and Raising the Time and Change Events

Comment

If the active object is not executing any run-to-completion step (checked by $\neg\, executing$), the event queue is empty (checked by $no\_dispatchable\_event$), then there is an L-transition, with the label resulting from the set of the received inputs ins, the time t, and the sent outputs $outs$ (all of them corresponding to have an attribute read, checked by Read_Attributes), where the history has been extended with the current states, the received inputs have modified the attribute status, as described by Updates, and the events generated by the inputs received from outside ($\mathsf{Events\_Of}(ins)$) plus the raised time and change events (defined by TimeEvs and ChangeEvs) have been added to the event queue.

- $(\neg\, executing(conf)) \;\wedge\; no\_dispatchable\_event(e\_queue) \;\wedge\; \mathsf{Read\_Attributes}(outs, attrs) \;\wedge$
  $\mathsf{ChangeEvs}(ins, attrs, id, chinf) = ch\text{-}evs, chinf' \;\Rightarrow$

  $id : \langle conf, attrs, e\_queue, history, chinf\rangle \xrightarrow{\langle ins, t, outs\rangle} id : \langle attrs', conf, e\_queue', history', chinf'\rangle$

where $attrs' = apply(\mathsf{Updates}(ins), attrs)$

$history' = \langle active\_states(conf), t\rangle \;\&\; history$

$e\_queue' = put(\mathsf{TimeEvs}(attrs, id, history, t) \cup ch\text{-}evs \cup \mathsf{Events\_Of}(ins), e\_queue)$

PROBLEM The condition requiring the emptiness of the event queue may be discussed; although the dispatch-run-to-completion cycle presented in UML 1.3 seems to suggest that if there is an event in the queue it should be dispatched. Different choices may be easily formalized by changing the definition of *no_dispatchable_event* in the specification EVENT_QUEUE.

10 Auxiliary Datatypes, Operations and Predicates

• Read_Attributes

pred Read_Attributes : $FinSet[Output] \times Attributes$

Read_Attributes$(outs, attrs)$ holds iff the outputs in $outs$ corresponding to outputting the values of some attributes are in accord with their values as defined by $attrs$.

It is defined by the axioms

Read_Attributes$(\{\}, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_scall(id, x, vl)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_acall(id, x, vl)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_return(id, x, v)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_send(id, x, vl)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_create(id, x)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_destroy(id, x)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \Rightarrow$
Read_Attributes$(\{O_update(id, x, v)\} \cup outs, attrs)$

Read_Attributes$(outs, attrs) \wedge evaluate(x, attrs) = v \Rightarrow$
Read_Attributes$(\{O_read(id, x, v)\} \cup outs, attrs)$

• Effect

The *effects* are descriptions of updatings of the attribute values that can be composed and applied. We need them to give the semantics to some constituents of the state machines.

spec Effect =
Attributes **then**
free {
type $Effect ::= no\_change \mid [\_ \to \_](Name; Value) \mid \_\&\_(Effect; Effect)$
%% *no_change* correspond to updating no attributes
%% $[x \to v]$ corresponds to update the attribute x with the value v
ops $\_\&\_ : Effect \times Effect \to Effect$ **assoc**
$apply : Attributes \times Effect \to Attributes$
%% modifies the attributes as requested by an effect
pred $\_\ is\_comp\_of(\_, \_) : Effect \times Effect \times Effect$
%% holds if an effect is a possible composition of two other effects
axioms
$apply(attrs, no\_change) = attrs$
$eff\ is\_comp\_of(eff, no\_change)$
$eff\ is\_comp\_of(no\_change, eff)$
.....
} end

• Updates

op Updates : $FinSet[Input] \rightarrow Effect$

Updates$(ins) = eff$ means that eff describes how to modify the attributes due to the inputs corresponding to updates included in ins. It is defined by the axioms

Updates$(\{\}) = no_change$
Updates$(\{I_scall(id, x, vl)\} \cup ins) =$ Updates(ins)
Updates$(\{I_acall(id, x, vl)\} \cup ins) =$ Updates(ins)
Updates$(\{I_return(id, x, v)\} \cup ins) =$ Updates(ins)
Updates$(\{I_send(id, x, vl)\} \cup ins) =$ Updates(ins)
Updates$(\{I_destroy(id)\} \cup ins) =$ Updates(ins)
Updates$(\{I_read(id, x, v)\} \cup ins) =$ Updates(ins)
Updates$(\{I_update(id, x, v)\} \cup ins) = [x \rightarrow v]$ & Updates(ins)

• Events_Of

op Events_Of : $FinSet[Input] \rightarrow Bag[Event]$

Events_Of$(ins) = evs$ means that evs is the set of events generated by the inputs in ins. It is defined by the axioms

Events_Of$(\{\}) = empty$
Events_Of$(\{I_scall(id, x, vl)\} \cup ins) = call(id, x, vl) +$ Events_Of(ins)
Events_Of$(\{I_acall(id, x, vl)\} \cup ins) = call(id, x, vl) +$ Events_Of(ins)
Events_Of$(\{I_return(id, x, v)\} \cup ins) =$ Events_Of(ins)
Events_Of$(\{I_send(id, x, vl)\} \cup ins) = signal(id, x, vl) +$ Events_Of(ins)
Events_Of$(\{I_destroy(id)\} \cup ins) = destroy(id) +$ Events_Of(ins)
Events_Of$(\{I_read(id, x, v)\} \cup ins) =$ Events_Of(ins)
Events_Of$(\{I_update(id, x, v)\} \cup ins) =$ Events_Of(ins)

• TimeEvs

op TimeEvs : $FinSet[Input] \times Attributes \times Ident \times History \times Time \rightarrow Bag[Event]$

Assume that the time events appearing in the state machine SM are $TE_1, \ldots, TE_n$.

TimeEvs$(ins, attrs, id, history, t) =$

$tev_1(ins, attrs, id, history, t) \cup \ldots \cup tev_n(ins, attrs, id, history, t)$ where for $i = 1, \ldots, n$ tev_i is an operation corresponding to TE_i defined by cases on the form of TE_i

op $tev_i : FinSet[Input] \times Attributes \times Ident \times History \times Time \rightarrow Bag[Event]$

– $TE_i =$ `after` exp (absolute time event)

This event occurs when the difference between the actual time and the time the object entered the source state of the transition in which appear the event (S_i) is equal to the value of exp.

Eval$(exp, ins, attrs, id) = (t - entered_state(history, S_i)) \Rightarrow$

$tev_i(ins, attrs, id, history, t) = \{TE_i\}$

Eval$(exp, ins, attrs, id) \neq (t - entered_state(history, S_i)) \Rightarrow$

$tev_i(ins, attrs, id, history, t) = \{\}$

– TE_i = `after` exp `since` S (relative time event)

This event occurs when the difference between the actual time and the time the object entered state S is equal to the value of exp.

$\mathsf{Eval}(exp, ins, attrs, id) = (t - entered\_state(history, S)) \Rightarrow$

$tev_i(ins, attrs, id, history, t) = \{TE_i\}$

$\mathsf{Eval}(exp, ins, attrs, id) \neq (t - entered\_state(history, S)) \Rightarrow$

$tev_i(ins, attrs, id, history, t) = \{\}$

PROBLEM May operation calls to other objects appear within the expressions of time events (similarly for the change events)?
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 active object).
We assume no, and this is the reason for the above quite simple functionality of TimeEvs.

• ChangeEvs

op ChangeEvs : $FinSet[Input] \times Attributes \times Ident \times ChangeInfo \rightarrow Bag[Event] \times ChangeInfo$

Assume that the change events appearing in the state machine SM are $CE_1, \ldots, CE_n$.

$\mathsf{ChangeEvs}(ins, attrs, id, chinf) = ch\text{-}evs_1 \cup \ldots \cup ch\text{-}evs_n, chinf_n$ where

$cev_1(ins, attrs, id, chinf, chinf) = ch\text{-}evs_1, chinf_1$

.....

$cev_2(ins, attrs, id, chinf, chinf_{n_1}) = ch\text{-}evs_n, chinf_n$

where for $i = 1, \ldots, n$ cev_i is an operation corresponding to CE_i defined as follows.

op $cev_i : FinSet[Input] \times Attributes \times Ident \times ChangeInfo \times ChangeInfo \rightarrow Bag[Event] \times ChangeInfo$

Assume CE_i = `change` $bexp$. This event occurs when the value of $bexp$ is true in the current state and was false in the previous state.

$\mathsf{Eval}(bexp, ins, attrs, id) = True \ \wedge \ evaluate(bexp, chinf) = False \ \Rightarrow$
$cev_i(ins, attrs, id, chinf, chinf') = \{CE_i\}, chinf'[True/bexp]$

$\mathsf{Eval}(bexp, ins, attrs, id) = True \ \wedge \ evaluate(bexp, chinf) = True \ \Rightarrow$
$cev_i(ins, attrs, id, chinf, chinf') = \{\}, chinf'$

$\mathsf{Eval}(bexp, ins, attrs, id) = False \ \Rightarrow \ cev_i(ins, attrs, id, chinf, chinf') = \{\}, chinf'[False/bexp]$

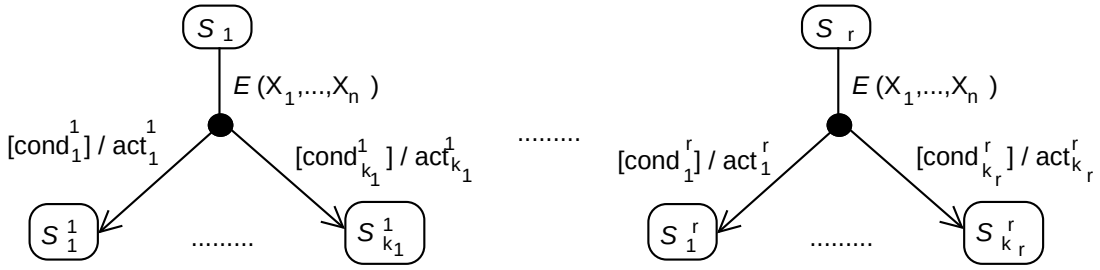
• Dispatch

op Dispatch : $Event \times Configuration \times Event\_Queue \times FinSet[Input] \times Attributes \times Ident \rightarrow Configuration \times Event\_Queue$

$\mathsf{Dispatch}(ev, conf, e\_queue, ins, attrs, id) = conf', e\_queue'$ means that dispatching event ev in the configuration $conf$ changes it to $conf'$ and changes e_queue to e_queue'.

It is defined by cases.

Some transitions triggered by the event Assume that in the state machine SM there are the following branched transitions triggered by E starting from the states belonging to $Sset = \{S_1, \ldots, S_r\}$; and that there are no transitions triggered by E starting from the states belonging to $Sset'$.



Note that inheritance of transitions between nested states and overriding may be handled here by deciding which transitions to consider.

- If $Sset \cup Sset'$ is the set of the active states, and $cond^i_{q_i}$ 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 $act^i_{q_i}$ $(i = 1, \ldots, h)$ 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 $cond^i_{q_i}$ does not hold for $i = h+1, \ldots, r$, $q = k^i_1, \ldots, k^i_{r_i}$.

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

  $\mathsf{Dispatch}(E(p_1, \ldots, p_n), conf, e\_queue, ins, attrs, id) =$
  $conf', remove(E(p_1, \ldots, p_n), e\_queue)$

  where

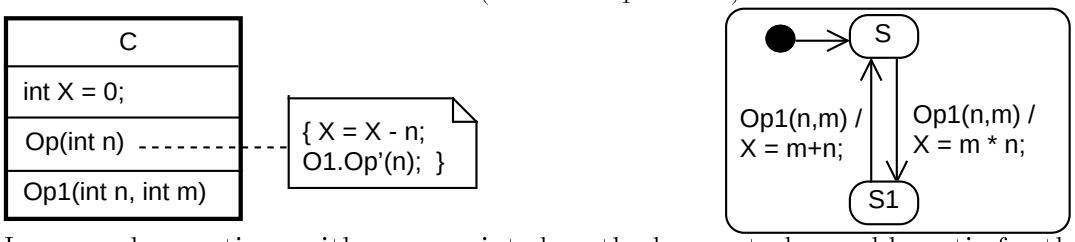
  $conf' = run(\ldots run(conf, S_i, act^1_{q_1}[p_j/x_j], S^1_{q_1}) \ldots, act^h_{q_h}[p_j/x_j], S^h_{q_h})$ $active\_states$ and run are operations of the specification CONFIGURATION returning respectively the active states of the state machine and recording the start of a run-to-completion step, going 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 the one that holds are semantically relevant.
Here we have assumed that the conditions 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 [10] p. 369 and other UML sources ([9]).

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 to 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.

- If the active states are $Sset \cup Sset'$, E is not deferred in any of the states in $Sset$, and for $i = 1, \ldots, r$, $q = 1, \ldots, k_i$ $cond_q^i$ does not hold, then such event is removed from the queue.

  $active\_states(conf) = Sset \cup Sset' \ \wedge$

  $\wedge \ {}_{i=1}^{r} \ \wedge \ {}_{q=1}^{k_i} \mathsf{Eval}(cond_q^i[p_j/x_j], ins, attrs, id) = False \ \Rightarrow$

  $\mathsf{Dispatch}(EV(p_1, \ldots, p_n), conf, e\_queue, ins, attrs, id) = conf, remove(E(p_1, \ldots, p_n), e\_queue)$

- If the active states are $Sset \cup Sset'$, E is deferred in some of the states in $Sset$, and for $i = 1, \ldots, r$, $q = 1, \ldots, k_i$ $cond_q^i$ does not hold, then such event is left in the queue.

  $active\_states(conf) = Sset \cup Sset' \ \wedge$

  $\wedge \ {}_{i=1}^{r} \ \wedge \ {}_{q=1}^{k_i} \mathsf{Eval}(cond_q^i[p_j/x_j], ins, attrs, id) = False \ \Rightarrow$

  $\mathsf{Dispatch}(E(p_1, \ldots, p_n), conf, e\_queue, ins, attrs, id) = conf, e\_queue$

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.

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

No transitions triggered by a nondeferred event Assume that in the state machine SM there are no transitions starting from the states belonging to $Sset$ triggered by E and that E is not deferred in any element of $Sset$.

If the active states are $Sset$, then event $E(p_1, \ldots, p_n)$ is just removed from the event queue.

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

• Exec

op Exec : $Configuration \times FinSet[Input] \times Attributes \times Ident \rightarrow$
$Configuration \times Effect \times Bag[Event] \times FinSet[Output]$

$\mathsf{Exec}(conf, ins, attrs, id) = conf', eff, loc\_evs, outs$ means that the object id with configuration $conf$ may execute an action changing its configuration to $conf'$, updating its attributes as described by eff and producing the set of local events loc_evs and the set of outputs $outs$.

Exec is defined by cases on the form of the action to be executed.

synchronous operation call

- Self calling
  $to\_execute(conf, scall(id, Op, exp_1, \ldots, exp_n), S) \wedge \overset{n}{i=1} \mathsf{Eval}(exp_i, ins, attrs, id) = v_i \Rightarrow$
  $\mathsf{Exec}(conf, ins, attrs, id) =$
  $execute(conf, scall(id, Op, exp_1, \ldots, exp_n), S), no\_change, \{call(id, Op, v_1 \ldots v_n)\}, \{\}$
- Calling another object
  $to\_execute(conf, scall(id', Op, exp_1, \ldots, exp_n), S) \wedge id' \neq id \wedge$
  $\wedge \overset{n}{i=1} \mathsf{Eval}(exp_i, ins, attrs, id) = v_i \Rightarrow$
  $\mathsf{Exec}(conf, ins, attrs, id) =$
  $execute(conf, scall(id', Op, exp_1, \ldots, exp_n), S), no\_change, \{\}, \{O\_scall(id', Op, v_1 \ldots v_n)\}$

asynchronous operation call similarly to the above case

sending a signal

- Sig is a signal for whom ACL has a reception (self sending). Notice that Sig is sent also outside because it must be received also by the other objects of the same class and by the objects of other classes having a reception for it.
  $to\_execute(conf, send(Sig, exp_1, \ldots, exp_n), S) \wedge \overset{n}{i=1} \mathsf{Eval}(exp_i, ins, attrs, id) = v_i \Rightarrow$
  $\mathsf{Exec}(conf, ins, attrs, id) =$
  $execute(conf, send(Sig, exp_1, \ldots, exp_n), S), no\_change,$
  $\{signal(Sig, v_1, \ldots, v_n)\}, \{O\_send(Sig, v_1, \ldots, v_n)\}$

– *Sig* is a signal for whom ACL has not a reception (sending outside)
$to\_execute(conf, send(Sig, exp_1, \ldots, exp_n), S) \ \wedge \ {}^{n}_{i=1}\mathsf{Eval}(exp_i, ins, attrs, id) = v_i \ \Rightarrow$
$\mathsf{Exec}(conf, ins, attrs, id) =$
$execute(conf, send(Sig, exp_1, \ldots, exp_n), S), no\_change, \{\}, \{O\_send(Sig, v_1, \ldots, v_n)\}$

destroy

– Self destruction
$to\_execute(conf, destroy(id), S) \ \Rightarrow$
$\mathsf{Exec}(conf, ins, attrs, id) = execute(conf, destroy(id), S), no\_change, \{destroy(id)\}, \{\}$

– Destroying another object
$to\_execute(conf, destroy(id'), S) \ \wedge \ id' \neq id \ \Rightarrow$
$\mathsf{Exec}(conf, ins, attrs, id) = execute(conf, destroy(id'), S), no\_change, \{\}, \{O\_destroy(id')\}$

creation of a new object

$to\_execute(conf, create(x), S) \ \Rightarrow$
$\mathsf{Exec}(conf, ins, attrs, id) = execute(conf, create(x), S), no\_change, \{\}, \{O\_create x\}$

assignment to one of its attributes

$to\_execute(conf, id.x = exp, S) \ \wedge \ \mathsf{Eval}(exp, ins, attrs, id) = v \ \Rightarrow$
$\mathsf{Exec}(conf, ins, attrs, id) = execute(conf, id.x = exp, S), [x \rightarrow v], \{\}, \{\}$

assignment to an attribute of another object

$to\_execute(conf, id'.x = exp, S) \ \wedge \ id \neq id' \ \wedge \ \mathsf{Eval}(exp, ins, attrs, id) = v \ \Rightarrow$
$\mathsf{Exec}(conf, ins, attrs, id) = execute(conf, id'.x = exp, S), no\_change, \{\}, \{O\_update(id', x, v)\}$

As said before, recall that we consider assignment just to give an example of uninterpreted action.

• Eval

op Eval : $Expression \times FinSet[Input] \times Attributes \times Ident \rightarrow Value$

$\mathsf{Eval}(exp, ins, attrs, id) = v$ means that the value of exp calculated using $attrs$ and ins (that contains the inputs corresponding to read the values of the attributes of other objects appearing in exp) by the object with identity id is v.

– attribute of the object
$\mathsf{Eval}(id.x, ins, attrs, id) = evaluate(x, attrs)$

– attribute of another object
$id' \neq id \ \wedge \ I\_read(id', x, v) \in ins \ \Rightarrow \ \mathsf{Eval}(id'.x, ins, attrs, id) = v$

– data operator
$\wedge \ {}^{n}_{i=1}\mathsf{Eval}(exp_i, ins, attrs, id) = v_i \ \Rightarrow$
$\mathsf{Eval}(OP(exp_1, \ldots, exp_n), ins, attrs, id) = op(v_1, \ldots, v_n)$

11 Constraints

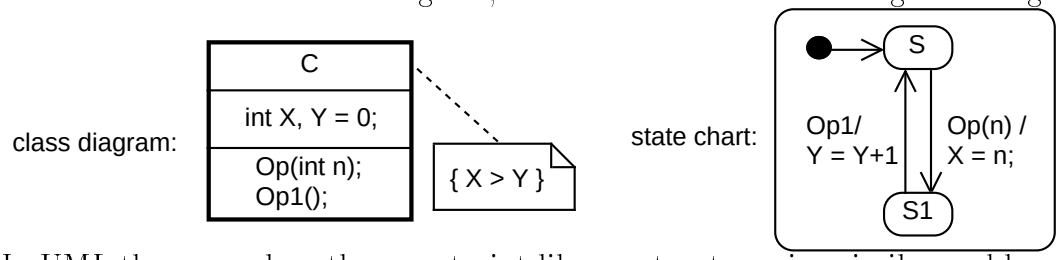
Constraints may be attached to any element of a UML model. For what concerns the fragment of UML considered in this paper we have constraints attached to the class icon (e.g., invariants) and to the operations (e.g., pre-post conditions), in the class diagram, and attached to the state machine (e.g., invariants). The language for expressing the constrains is not fixed in UML (also if 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 meta-model, 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 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 formulae of CASL allow to express a rich set of relevant properties, recall that the underlying logic of CASL is many sorted first-order logic, and that CASL extensions with temporal logic combinators, based on [3] are under development. Moreover the constraints expressed using OCL may be translated in CASL without too many problems.

Assume we have the constraints C_1, ..., C_n attached either to ACL or to SM, then the UML model containing ACL and SM is well formed iff for $i = 1, \ldots, n$, $L \models \Phi_i$, 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 state machine violating them is given elsewhere.



In UML there are also other constraint-like constructs posing similar problems; as the query qualification for operation (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 precise methodology for using these constraints, making precise their role and when to use them in the development process.

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 International Conference, Edimburgh, 1986*, pages 239–248. University Press, Cambridge, 1986.

[2] 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.

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

[4] 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 Lecture Notes in Computer Science, pages 115–137, Berlin, 1997. Springer Verlag.

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

[6] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.

[7] 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.

[8] M. Roggenbach and T. Mossakovski. Basic Data Types in CASL . CoFI Note L-12. Technical report, 1999. `http://www.brics.dk/Projects/CoFI/Notes/L-12/` .

[9] J. Rumbaugh. Some questions relating to actions and their parameter, and relating to signals. Private communication, 1999.

[10] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.

[11] UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.

Contents