Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002

Metamodelling Behavioural Aspects: the Case of the UML State Machines

G. Reggio
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso 35, 16146 Genova, Italy
reggio@disi.unige.it

ABSTRACT: The "object-oriented meta-modeling" seems currently to be one of the most promising approach to the "precise" definition of UML. Essentially this means using a kernel object-oriented visual notation to define UML. This has proved itself to be an intuitive way of defining the abstract syntax of UML. For what concerns the "static part", the initial work of the pUML group seems to confirm that the approach is workable, whereas no convincing proposal have been presented to cover the dynamic aspects, as active classes, cooperation/communications among active objects, state charts, sequence diagrams, and so on.

We think that to conveniently and precisely define such aspects, we need an underlying formal model of the dynamic behaviour of the active objects, and we think, supported by our past experience, that labelled transition systems are a good one. We thus propose how to use a kernel visual notation to define labelled transition systems in an object-oriented way. Furthermore, we present also a new kind of diagrams, quite similar to state charts, LTD (Labelled Transition Diagrams) to visually present such definitions.

As an example, we work out the meta-model of the state machines starting from our formal definition of their semantics based on labelled transition systems.

I. Introduction

UML [25] is the object-oriented visual notation for modelling software systems recently proposed as a standard by the OMG (Object Management Group[1]). The semantics of UML, which has been given only informally in the original documentation, is a subject of hot debate and a of lot of efforts, much encouraged by the OMG itself.

In the literature there are many attempts to give a formal semantics to UML, or better to a subset of it; for references see the proceedings of the last UML conferences [9], [2], [10]. UML presents special features making this task non-trivial. The behavioural parts of UML, as active classes, state machines, sequences and collaboration diagrams, pose most of the problems, see e.g., [26], [23], and have been the topics of many workshops at ECOOP, OOPSLA and UML conferences in the last years, see, e.g., [21], [5], [1]. If we just consider the state machines, then we can build a long list of papers trying to give their semantics using a variety of techniques and notations, see, also for references, [9], [2], [10]. However, in some cases only a very restricted subset of the state machines is considered; and in other cases there is no idea on how to integrate the semantics of the behavioural view given by a state machine with the views of a system given by the other diagrams that constitute a UML model.

The formal methods group in Genova, of which I am part, has worked on this topic by giving a formal semantics to state machines associated with active classes [19], [18]. This work has been useful, because we have been able to discover many, also quite subtle, problems in the intended meaning of state machines. It seems that their new revised version in the forthcoming UML 2.0 will fix some of them. We have also studied how to integrate such semantics of the state machines with those of the other views [20]. In those papers we adopted rather classical techniques, like modelling processes as labelled transition systems (shortly *lts* from now on) as in CCS et similia, and algebraic specification techniques, supported by a recently proposed family of languages [24], [17].

Unfortunately our semantics, though treating adequately the various features of the UML, are rather hard to read and to understand for people that have not a background in formal methods. The reason is that the algebraic specification languages are based on logic, require a lot of notational overhead (so specifications are quite long), and are not visual. Furthermore, still more difficult for such people is to modify the formal definition of UML when they have to design a UML variant for some particular application (and this is more or less the standard way to use UML).

Other formalizations of parts of UML suffer the same problems; in many cases, the notations are more exotic, the underlying formal models are more complex, and the resulting specifications longer than the algebraic one based on lts.

Recently a new different approach to define the notations used in the software development process appeared, that is OO visual metamodelling, and has been used also for UML.

A *model* is a collection of artifacts assembled during the

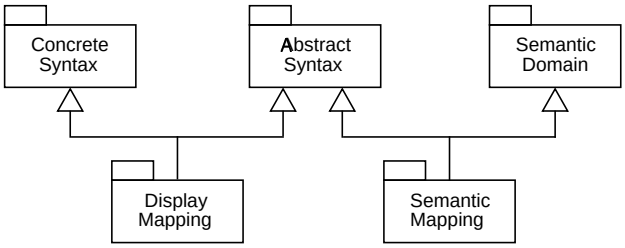
[1] http://www.omg.org/

modelling of a software system (e.g., a UML model). A *metamodel* is a model of the information that can be expressed during modelling (e.g., the definition of UML [25] by OMG, that is of the UML models). A *meta-metamodel* is the language in which the metamodel(s) can be expressed. Following the OMG approach, a metamodel of a notation (e.g., UML) is roughly the OO visual definition of its abstract syntax, given using MOF [15]. Since MOF is essentially a small subset of UML, the metamodels, as that of the UML, could be easily understood by the people using UML, because they are written using more or less the same notation.

Unfortunately, these definitions of notations are not rigorous at all; indeed, the semantics of MOF, and also that of the metamodelled notation (e.g., UML) are given only by natural language text, showing many ambiguities, incompleteness, and inconsistencies, see, e.g., [23].

To fix this and many other points of the OMG-MOF approach, "precise metamodelling" has been proposed, initially within the pUML group (Precise UML[2]), then designated as MMF (Meta Modelling Facility) and very recently supported by the "2U Consortium"[3], with the aim to use it for a precise definition of the next version of UML (UML 2.0).

Precise metamodelling [7] means that "the metamodel should concerns not only the abstract syntax of the notation, but also its semantics and the presentations of its models". A metamodel presented using a kernel object oriented language, should have the form below



where boxes and hollow arrows denote packages and package specializations respectively.

Moreover, always to guarantee "precision", the precise metamodelling proposal includes a formal semantics of the notation used to present the metamodels [6]. Up to now, as it results from the available documentation [7], [14], [3], such proposal does not cover the dynamic/behavioural aspects of a notation.

In this paper, we introduce a way to handle notations with behavioural dynamic aspects within a comprehensive approach to metamodelling that we are developing, [11], by defining a metamodel for UML state machines. Our approach, "Extreme metamodelling", offers a visual OO notation for presenting metamodels (the meta-metamodel), GML, that we briefly introduce in Section II; its precise definition by means of a metamodel is in [11].

We have already given a formal semantics of the UML state machines in [19], [18] using labelled transition systems and an algebraic notation CASL [24], [17]; thus, here we do not discuss all subtleties and open problems of their semantics. Therefore, we have already found some appropriate semantic domains for active classes and state machines. An instance of an active class is modelled by a labelled transition system, and a class by the set of all lts's modelling its instances. A state machine associated with an active class AC defines which are the lts's giving the semantics of AC.

Thus, to metamodel state machines we need to define lts's in an OO fashion by using GML. In the past, we have found many ways of visually presenting formal definitions of lts's [22], [16], [8], [13] by means of variants of graphs whose arcs are templates for sets of transitions. So, we introduce in GML a new construct, **LTS**, to define lts's in an OO way, and a new kind of diagrams, **LTD** (Labelled Transition Diagram) to visually define their transitions. Such constructs are presented in Section III. The state machine metamodel is presented in Section IV; due to space limit this is a short version, the corresponding complete one including all details is in [12].

In the conclusions Section V we discuss the result of this experiment in metamodelling state machines.

II. GML: The Meta-Metamodel

GML is the notation offered by the Extreme Metamodelling approach [11] for defining the metamodels of notations used in the software development (the metametamodel) . It is an elementary object-oriented visual notation roughly corresponding to a subset of UML.

GML is similar to MOF and to MML (the metametamodels of OMG and 2U respectively). As MML it is defined both in the standard way (by giving its syntax and its formal semantics) and by a metamodel expressed using itself, see [12]. Differently from it, and from MOF, it uses a standard (functional-infix) notation for expressions and formulae, instead of the strange concrete syntax of OCL (the language used for expressions and constraints in UML).

GML is quite elementary, but it has a straightforward mechanism for extending itself based on macro-expansion.

A. The GML Notation

A GML model, see Figure 1 (UML State Machines Semantic Domain) for an example, consists of a set of
classes characterized by a name, some attributes and some operations; the states of their instances (objects) are not updateable, and so the operations are just functions ;
associations characterized by a name and the two ends (two classes, they are always binary); they are navigable in both senses;
subtype assertions stating that a class is a subtype of another one.

[2] http://www.cs.york.ac.uk/puml/
[3] http://www.2uworks.org/index.html

Moreover, it is possible to restrict the instantiations of the various elements of a model (also the model itself) by stating properties (named "constraints") that they must satisfy. A constraint attached to
the model restricts the possible instantiations of its classes and associations (model invariant);
a class restricts the possible values of its attributes (class invariant);
an operation restricts the returned values; these constraints are pairs pre-post conditions, whose meaning is as follows: when the pre-condition holds, then the returned value satisfies the post condition. Notice that these constraints, differently from UML, do not impose restrictions on when the operation may be called but only on the results;
an association restricts the sets of its links (e.g., a multiplicity assumption).
All constraints are expressed by first-order logic formulae.

GML provides also a rich set of predefined datatypes, including the usual basic types (boolean, integer, string, ...), those needed to evaluate the navigation expressions (sets and sequences), plus some special types related to the OO features. The latter are similar to those present in OCL and help express many constraints frequently used in metamodels. Among them, we have the most general supertype, the type of all objects, the type of the "types", and so on. Such special types are equipped with the corresponding operations; e.g., for checking if an object has a given type or if a type is a subtype of another, for returning all instances of a given type, et cetera.

The visual notation is the standard UML like one. We use boxes with three compartments for classes, lines for associations, arrows for marking navigation versus, and attached notes for the constraints (alternatively they can be reported apart in a purely textual form). Hooked arrows represent subtype assertions. Expressions, and so boolean expressions (i.e., formulae), are presented using the usual common functional or infix syntax.

B. GML Formal Semantics

It is easy to give a complete formal semantics to GML; moreover, because it is extremely simple there is no need of complex mathematical theories and techniques, as those used in [6].

We give a formal meaning to a GML model following these points.

- The meaning of an object is the sets of all its states; a state of an object is characterized by its identity, its type, the values of its attributes and the values returned by the various operations, when applied to it.
- The meaning of a class is the set of all the meanings of its instantiations (objects).
- The meaning of an association is the set of all its states, that are the sets of its links (pairs of object identities).
- The meaning of all subtype assumptions is the subtype relationship (a set of pairs of values of type "type").
- The meaning of a model is the set of all the possible objects communities that it defines (a set of objects and links that can exist simultaneously). In this case, a community is simply represented by a set of object and association states.
- The meaning of a datatype is standard, i.e., a bunch of sets and functions over them.
- The constraints allows to determine which are the admissible meanings for the various constructs; e.g., a constraint on a class restricts the set of the possible states of its objects, one on a model reduces the set of the object communities it defines and so on.

The above ideas may be formally presented using any appropriate notation; for example CASL [24] following [20], or just a plain mathematical notation sufficient to present sets and functions.

C. GML Macro-Based Extension Mechanism

GML is really simple, you can say also poor, for example there is no provision either for specialization (it has just subtyping), or for packages. Thus, it is poorer than MOF [15] and of MML [6]. But, to overcome this limit it has a simple extension mechanism based on macro expansion.

An extension GML* of GML can be defined in the following way.

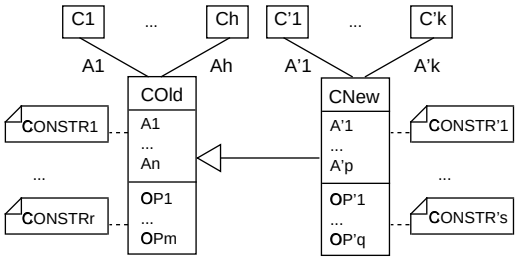
Define GML* at a syntactic level, by giving the abstract syntax of the new constructs, together the new static correctness conditions, and their visual presentation (and this part is mandatory). Then, you give a translation of any (clearly statically correct) model fragment corresponding to an instance of the new constructs into a model fragment of GML. Thus, any model defined by using GML* may be transformed into a corresponding model defined by using GML, by replacing any instance of the new constructs with its macro expansion.

Such translation must be easy to explain and to present, and quite intuitive. An informal rule to check if the translation is simple enough is "you should be able to intuitively present it by showing how to translate a generic instance of the new constructs by using two generic visual diagrams".

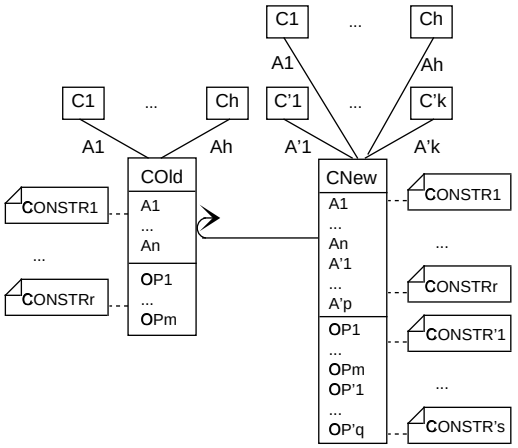
Examples of new constructs for GML, which may be easily defined in this way, are multiplicity constraints for associations, aggregated objects, class specialization, packages, package specialization, and so on. Below we give the complete definition of class specialization.

Class specialization means to define a class starting from an old one by giving new attributes, operations, and constraints on the new class and operations, but also on the old operations. Old attributes and operations cannot be overridden (the new ones must have new names). Clearly, the new constraints may be inconsistent with the old ones, but this is not different from defining by scratches a model with inconsistent constraints.

A class diagram fragment including specialization as



stands for



III. Defining Labelled Transition Systems with GML

We recall that a *labelled transition system* (shortly *lts*) is a triple $(State, Label, Transition)$, where *State* and *Label* are two sets, and $Transition \subseteq State \times Label \times State$ is the *transition relation*. A triple $(s, l, s') \in Transition$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$.

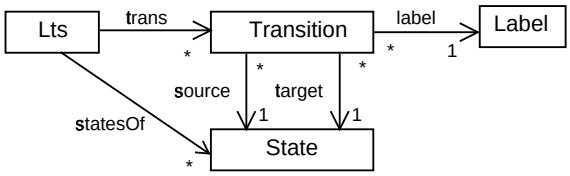
If an lts models a dynamic system *S*, then a transition $s \xrightarrow{l} s'$ has the following meaning: *S* in the situation s has the *capability* of passing into the situation s' by performing a transition, where the label l represents the interaction with the environment during such a move; thus 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.

We add to GML two constructs to help to define an lts: **LTS** to define which are its states and labels, and **LTD**, a new kind of diagrams, to visually define its transitions.

A. *LTS (Labelled Transition System)*

Visually an instance of the construct **LTS** is a package marked by *lts* in the small box containing the package name. We require that such package contains one class marked by *state* and another one marked by *label* in the name compartment (the two marks may be put also on the same class) and that it does not contain any class named either Lts or Transition.

A correct **LTS** construct L, where the classes marked by *state* and *label* are respectively State and Label, stands for a package containing all elements of L plus



and the following constraint

```
context lts: Lts
  for all tr in lts.trans
    tr.source in lts.statesOf and
    tr.target in lts.statesOf
```

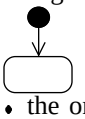
It is easy to see that each instance lts of class Lts corresponds to a standard lts

(lts.statesOf, { tr.lab | tr in lts.trans }, lts.trans).

B. *LTD (Labelled Transition Diagram)*

The construct **LTS** allows to define which are the classes corresponding to the states and labels of an lts; moreover it introduces a class Transition, whose elements correspond to the transitons of the lts itself. By adding constraints we can precisely define the set of the transitions of the lts. However, large sets of constraints on the transitions may be not very readable and are complicate to write. So here we present a visual way to define the transitions of an lts by means of a "Labelled Transition Diagram" (**LTD**).

An **LTD** must be associated with an **LTS** package, say L, and is an oriented graph where:

- the nodes are rounded boxes decorated by strings (elements of the GML type String); exactly one node must be labelled with the string Initial and any number with the string Final. The nodes marked by Initial may be shown as  and those marked by Final as  .
- the oriented arcs going from one node (not labelled by Final) to another one (also the same) are decorated by

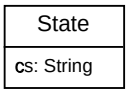
  [cond1] lab / cond2

where

– lab, cond1 and cond2 are three GML expressions, possible with free variables, lab having type Label and the others having type Bool,

– at most one free variable of type State may appear in cond1 and in lab (and it is named S1), at most two free variables of type State may appear in cond2 and are named S1 and S2.

The **LTD** construct corresponds to enrich the states of the lts L with a "control state" (an attribute of type String) and with a set of constraints defining the transitions of the lts itself. Precisely, an instance of **LTD** is translated into a specialization of the package L containing



(that means to add exactly the attribute cs: String to class State) and the constraints

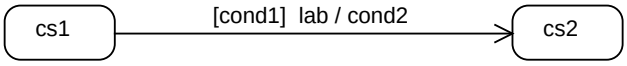
- context S: State
  S.cs = str1 or . . . or S.cs = strk

where str1, . . . , strk are all the strings appearing on the nodes of the **LTD**.

- context lts: Lts
  for all tr: Transition
  tr in lts.trans iff (condT1 or . . . or condTn)

where T1, . . . , Tn are all transitions of the **LTD**, and for i = 1, . . . , n condTi is the condition derived by the transition Ti defined as follows.

A transition of the form



corresponds to the condition

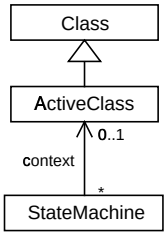
there exit x1:T1, . . . , xh:Th such that
tr.source.cs = cs1 and
cond1[tr.source/S1] and
tr.label = lab[tr.source/S1] and
tr.target.cs = cs2 and
cond2[tr.source/S1,tr.target/S2]

where the free variables, different from S1 and S2, appearing in cond1, lab and cond2 are x1:T1, . . . , xh: Th, and a[b/X] denotes a where all free occurrences of X have been replaced by b.

IV. Metamodeling UML State Machines

A. UML State Machines Abstract Syntax

In the case of the UML state machines the abstract syntax has been already given in [25] and we cannot change it, because it is the official one. We just assume that the abstract syntax package (UML AS) contains the following fragment. Notice, that however, there are no problems in using GML to define it.



B. UML State Machines Semantic Domain

In [19] when giving a formal semantics to UML state machines we have consider almost all constructs, we just left out only few ones not very relevant (i.e., because they are equivalent to combinations of other constructs, or because they are small variations of others, and so whose semantics can be given without effort by looking at that of the similar construct). Here due to space limits, we do not consider timed, change and deferred events and signal sending and receiving; however they can be introduced later without too much troubles. It just requires adding some attributes to the class state, some new kinds of labels and some transitions to the **LTD**.

The lts, SMLts, modelling the UML active objects with associated state machine is defined by first using **LTS**, and after by defining its transitions by means of **LTD**.

As explained in [25] the activity of an active object specified by a state machine consist of performing a run-to-completion step (shortly r-t-c-s) after the other. An r-t-c-s consists of dispatching an event (i.e., to get it from the event queue), then if the state machine has a transition with such event as trigger and whose condition holds, then the action part of such transition will be executed till it ends; otherwise the r-t-c-s is just terminated. After that the object is ready to dispatch another event, if any, starting another r-t-c-s. At any moment, either when engaged in a r-t-c-s or not, some attributes of the object may be read and updated by other objects (in UML they are not encapsulated), and some calls of its operations may be received, resulting in events which are put in the event queue.

In Figure 1, we present the instance of the construct **LTS** used for the UML state machines. For lack of space, the details of some classes and the constraints are omitted, they can be found in [12]. There we use the following two new GML constructs.

black arrow (strong specialization) it is a variant of specialization additionally requiring that any element of the super type belongs to one and only one subtype .

datatype A datatype class is presented by putting in the left upper corner of the class icon a capital D enclosed by a square box (D). It requires that the class has always exactly a unique instance for any possible choice of the values of its attributes. Moreover, if the name of the class is Cla, for each choice of values of the attributes, say v_1, . . . , v_n, the unique instance determined by such value may be denoted by Cla(v_1, . . . , v_n). If the class is defined by strong specialization, instead of a unique constructor, we have many constructors, one for each subtype.

In the same picture the classes *SMSState* (control states of the state machines) and *Action* are defined in the package *UML AS* containing the definition of the abstract syntax of the UML state machines.

Since the **LTD** defining the transitions of SMLts cannot be put on just one page, we first present its schematic form in Figure 2, and after give the precise presentations of all its transitions.

To present in a convenient way the transitions of this **LTD**, we introduce a shortcut for a very common kind of conditions on their target. We write

S2.A1 = exp1; . . . ; S2.An = expn;

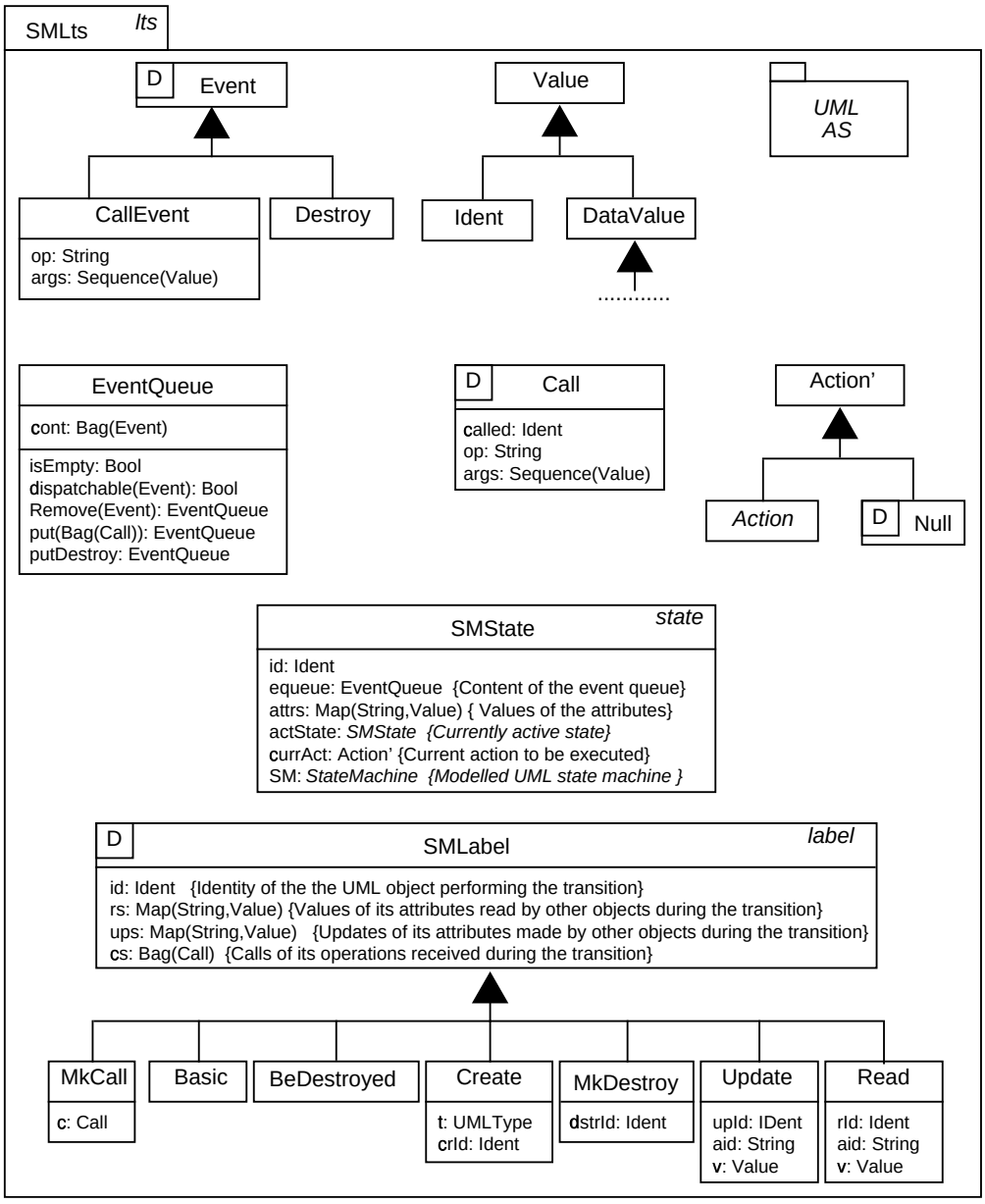


Fig. 1. UML State Machines Semantic Domain

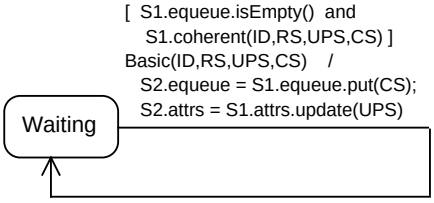
for a condition of the form

S2.A1 = exp1 and . . . and S2.An = expn and S2.An+1 = S1.An+1 and . . . and S2.Am = S1.Am

where A1, . . . , An, An+1, . . . , Am are all the attributes of the class State, different from cs.

Here we give just some samples of the transitions of the **LTD** the remaining ones are in [12]. We need some additional operations to define them; similarly to [25] they are defined textually near to the point where they are used; for lack of space we do not report their precise definitions that can be found in [12].

Have some attributes read or updated and receive some calls



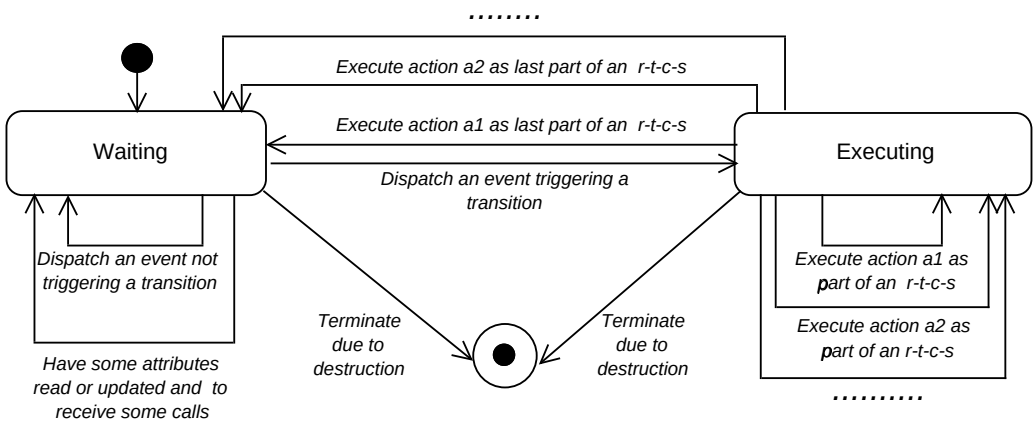
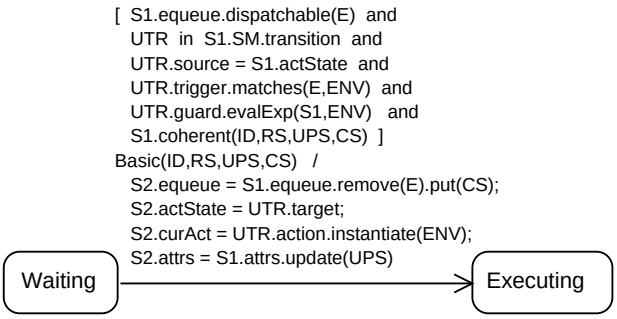


Fig. 2. **LTD** defining the transitions of SMLts

- context SMState::
  coherent(ID:Ident,RS:Map(String,Value),
  UPS:Map(String,Value),CS:Bag(Call)):Bool

returns True iff ID (identity), RS (read values of attributes), UPS (updates of attributes) and CS (set of operation calls) are correct for the UML active object of which self is a state

Dispatch an event triggering a transition



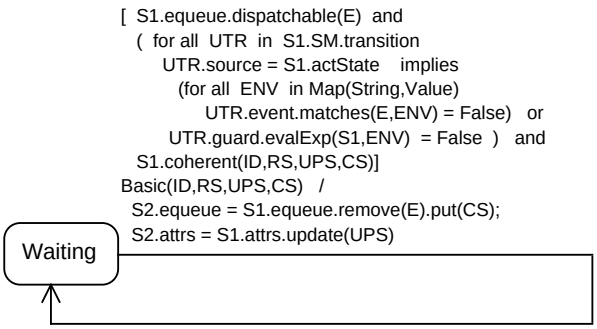
- context *Event*::
  matches(E:Event,ENV:Map(String,Value)): Bool

returns True iff E matches self when its free variables are instantiated using the environment ENV

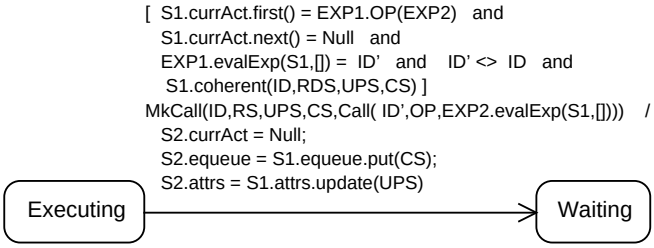
- context *Expression*::
  evalExp(S:SMState,ENV:Map(String,Value)): Value

evaluates self over the state S instantiating its free variables using the environment ENV

Dispatch an event not triggering a transition



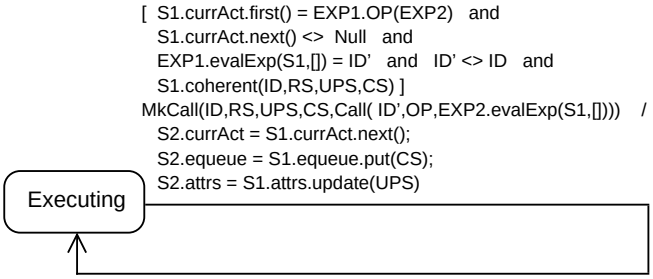
Call an operation of another object as last part of a r-t-c-s



- context Action'':: first(next): Action'

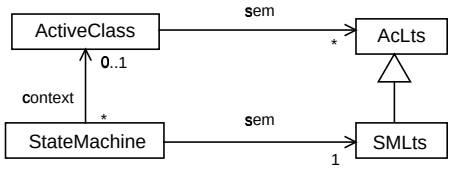
returns the atomic action which must executed as first step of self (the action that must be executed after the first atomic one has been executed)

Call an operation of another object as a part of a r-t-c-s



C. UML State Machines Semantics

The package defining the UML state machines semantics consists of the following class diagram



and of the following constrains.

```
context sm: StateMachine
  sm.sem = lts implies
    for all S in lts.statesOf S.SM = sm
```

The semantics of a state machine associated with an active class is an lts of the kind defined in the previous section. We have reported in each state of the lts the original state machine just to have at hand which are its transitions that define those of the lts.

The semantics of a UML active class is a set of lts's (belonging to AcLts) which are a generalization of those in SMLts (the labels are the same but their states have only the attributes attrs and Id and their transitions are whatever). Their precise definition is in [12].

If an active class has an associated state machine, then its semantics is fully determined by such machine, as stated by the following constraint.

```
context AC: ActiveClass
  AC.context <> {} implies
    for all SM in AC.context
      AC.sem = SM.sem[4]
```

V. Conclusions

The result of the experiment in metamodelling UML state machines presented in this paper is, in our opinion, positive. First, we have shown that the "precise metamodelling" of [7] can be used also for the behavioural aspects of a software engineering notation as UML.

This precise metamodel has all the good properties of our previous formal algebraic definition of state machines, which was covering almost any aspect of such constructs, and which can be integrated with the precise definition of the other UML diagrams. The object-oriented visual notation GML, which we have used, has proven to be adequate to present all the needed structures without too much effort, thanks to its macro-extension facility. But the state machine metamodel can be shown without a long preliminary presentation of the used notation; it is visual, much shorter and extremely more readable than the algebraic one. Furthermore, the OO features of GML are useful to update and to extend such definition, whenever the UML state machines are updated or extended.

The GML extension (**LTS** + **LTD**), defined here for the state machines, can be reused many times; for instance, to give the precise metamodel of the whole UML following [20] (the meaning of a whole UML model is again a set of lts's, clearly different ones), or to give the metamodel of other behavioural aware notations as JTN [8], a visual notation for Java targeted designs.

The GML constructs for lts's can be specialized to help describe more specialized lts's for particular uses in metamodelling. For example, we can define structured lts to describe groups of dynamic entities cooperating together (e.g., the UML active/passive objects in a model, in turn define by other lts's), and generalized lts (i.e., structured lts's where each transition is equipped with a description of the moves made by the composing entities during it) used, e.g., in [20] to describe UML sequence diagrams. Special visual notations may be introduced to describe how the component entities of a structured lts cooperate among them, for example, as the cooperation diagrams of [22].

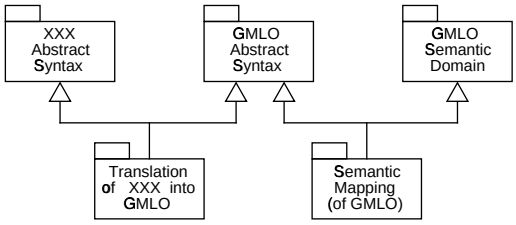
Here we have presented a precise metamodel of state machine where the semantic domains have been considered at a quite conceptual level; their elements are values of some kinds abstractly denoting the various elements of the notation, following the classical denotational semantics style. Such values have been presented using an object-oriented notation, but the object-oriented flavour is mild: almost all classes defining the semantic values are datatypes (then, their states are not updateable).

However, there is another way to describe the semantic domain in an object-oriented way that is as follows.

We define GMLO by extending GML with classical objects with updateable states, mechanisms to define their methods, active objects and mechanisms to define their behaviours, mechanisms to define which are the objects composing a system, and how they cooperate among them. Clearly, such mechanisms should be quite general and simple, but powerful.

Then, the semantic mappings package for a notation XXX will associate the constructs of XXX with particular instances of GMLO constructs; e.g., a UML model is associated with a GMLO model, a UML active class with a GMLO active class, a UML state machine with a GMLO Labelled Transition Diagram (assuming that the behaviour of active objects in GMLO is defined by an LTD), and so on. In this case, the semantic domains package for XXX is just the abstract syntax package of GMLO; but, because GMLO has a semantics given directly and so its own semantic Domain package, then indirectly this is also the semantic domain package of XXX. In some sense, the semantics given in this way is a kind of translation of XXX into an elementary OO language.

[4] UML allows to associate many state machines with a unique active class, but this should be prohibited, since a state machine fully determines the lives of the class objects; so we assume that AC.context contains at most one element.



A similar style, but in a setting based on algebraic specification and process calculi, has been used successfully to formally define the Ada programming language [4].

We plan to investigate whether this "translation-based" style of metamodelling may present some advantages with respect the one followed here for the UML state machines.

REFERENCES

[1] UML 2001 Workshop: Concurrency Issues in UML . Web site at `http://wooddes.intranet.gr/uml2001/Home.htm`, 2001.

[2] S. Kent A. Evans and B. Selic, editors. *Proc. UML'2000*. Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[3] J.M. Alvarez, T. Clark, A. Evans, and P. Sammut. *An Action Semantics for MML*. In M. Gogolla and C. Kobryn, editors, *Proc. UML'2001*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

[4] 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*. University Press, Cambridge, 1986.

[5] J.-M. Bruel, J. Lilius, A. Moreira, and R.B. France. *Defining Precise Semantics for UML*. In J. Malefant, S. Moisan, and A. Moreira, editors, *ECOOP 2000 Workshop Reader*, number 1964 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[6] T. Clark, A. Evans, and S. Kent. *The Meta-Modeling Language Calculus: A Foundation Semantics for UML*. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

[7] T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. *A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach - Version 1.0. (September 2000)*. Available at `http://www.cs.york.ac.uk/puml/mmf.pdf`., 2000.

[8] E. Coscia and G. Reggio. *JTN: A Java-targeted graphic formal notation for reactive and concurrent systems*. In Finance J.-P., editor, *Proc. FASE 99*, number 1577 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.

[9] R. France and B. Rumpe, editors. *Proc. UML'1999*. Number 1723 in Lecture Notes in Computer Science. Springer Verlag, 1999.

[10] M. Gogolla and C. Kobryn, editors. *Proc. UML'2001*. Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

[11] G.Reggio. *An "Extreme" Approach to Metamodelling*. Technical report, DISI, Università di Genova, Italy, 2002. In preparation.

[12] G.Reggio. *Metamodelling Behavioural Aspects: the Case of the UML State Machines: Complete Version*. Technical Report DISI–TR–02–3, DISI, Università di Genova, Italy, 2002. Available at `ftp://ftp.disi.unige.it/person/ReggioG/Reggio02b.pdf`.

[13] G.Reggio and E. Astesiano. *A Proposal of a Dynamic Core for UML Metamodelling with MML*. Technical Report DISI–TR–01–1, DISI, Università di Genova, Italy, 2001. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioAstesiano01a.pdf`.

[14] A. Kleppe and J. Warmer. *Unification of Static and Dynamic Semantics of UML*. Technical report, Klass Objecten, The Netherlands, 2001. Available at `http://www.cs.york.ac.uk/puml/mmf/KleppeWarmer.pdf`.

[15] OMG. *Meta Object Facility (MOF 1.3) Specification*. `http://www.omg.org/cgi-bin/doc?formal/00-04-03.pdf`, 2000.

[16] G. Reggio and E. Astesiano. *An Extension of UML for Modelling the non Purely-Reactive Behaviour of Active Objects*. Technical Report DISI–TR–00–28, DISI, Università di Genova, Italy, 2000. Available at `ftp://ftp.disi.unige.it/person/ReggioAstesiano00b.pdf`.

[17] 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`.

[18] 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. Revised March 2000. Available at `ftp://ftp.disi.unige.it/person/ReggioG/Reggio99b.ps`.

[19] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. *Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach*. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[20] G. Reggio, M. Cerioli, and E. Astesiano. *Towards a Rigorous Semantics of UML Supporting its Multiview Approach*. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

[21] G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa (editors). *Dynamic Behaviour in UML Models: Semantic Questions*. Technical report, Ludwig-Maximilian University, Munich (Germany), 2000.

[22] G. Reggio and M. Larosa. *A Graphic Notation for Formal Specifications of Dynamic Systems*. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.

[23] G. Reggio and R. Wieringa. *Thirty one Problems in the Semantics of UML 1.3 Dynamics*. OOPSLA'99 workshop "Rigorous Modelling and Analysis of the UML: Challenges and Limitations", 1999.

[24] The CoFI Task Group on Language Design. CASL *The Common Algebraic Specification Language Summary. Version 1.0*. Technical report, 1999. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[25] UML Revision Task Force. *OMG UML Specification 1.3*, 1999. Available at `http://www.rational.com/media/uml/post.pdf`.

[26] R. Wieringa, E. Astesiano, G. Reggio, A. Le Guennec, H. Hussman, K. van den Berg, and P. van den Broek. *Is it feasible to construct a semantics for all of UML?: Dynamic behaviour and concurrency*. In S.Kent, A.Evans, and B. Rumpe, editors, *ECOOP'99 Workshop Reader: UML Semantics FAQ*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.