# An Algebraic Semantics of UML Supporting its Multiview Approach

G. Reggio – M. Cerioli – E. Astesiano
DISI Università di Genova - Italy

**Abstract**

We aim at using algebraic techniques, and in particular an extension, CASL-LTL of the CASL basic language in order to produce a formal semantics of the UML. Contrary to most cases, this task is far from trivial. Indeed, the UML notation is complex, including a lot of heterogeneous notations for different aspects of a system, possibly described in different phases of the development process. Moreover, its informal description is incomplete and ambiguous, not only because it uses the natural language, but also because the UML allows the so called *semantics variation points*, that are constructs having a *list* of possible semantics, instead of just one.

## Introduction

UML is the object-oriented notation for modelling software systems recently proposed as a standard by the OMG (Object Management Group), see UML Revision Task Force (1999) and Rumbaugh et al. (1999).

A UML model consists of a bunch of diagrams of different kinds, expressing properties on different aspects of a system. In the following we will call UML-*systems* the "real world" systems modeled by using the UML (some instances are information systems, software systems, business organizations). Thus a UML model plays the role of a specification, but in a more pragmatic context.

Another analogy that we can establish between UML models and specifications is the fact that the meaning of each diagram (kind) can be given in isolation, as well as the semantics of each axiom, and its effect on the description of the overall UML-system is to rule out some elements from the universe of all possible systems (semantic models). Indeed, both in the case of a UML model and of a collection of axioms, each individual part (one diagram or one axiom) describes a point of view of the overall systems.

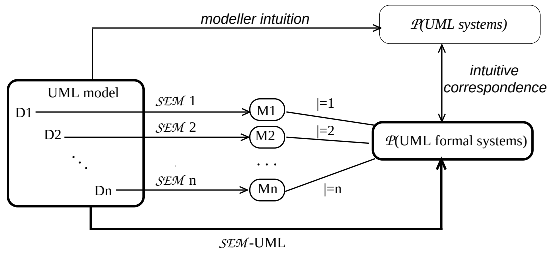


Figure 1:

Therefore, our understanding of the optimal form of a semantics for the UML is illustrated in Fig. 1, where the UML *formal systems* are the formal counterparts of the UML-systems.

We have a box representing a UML model, collecting some diagrams of different kinds, and its overall semantics, represented by the arrow labelled by $\mathcal{SEM}$-UML, is a class of UML formal systems. But, each diagram in the model has its own semantics (denoted by the indexed $\mathcal{SEM}$), that is a class of appropriate structures, as well, and these structures are imposing constraints on the overall UML formal systems, represented by lines labelled by the indexed $\models$. A sort of commutativity on the diagram has to hold, that is the overall semantics must be the class of the UML formal systems satisfying all the constraints imposed by the individual semantics. Moreover, the formal semantics must be a rigorous representation of the expected "intuitive semantics", described by the UML standard, version 1.3 (UML Revision Task Force (1999), shortly written from now on UML 1.3).

Several attempts at formalizing the UML are currently under development, but most of them are taking into account only a part of the UML, with no provision for an integration of the individual diagram semantics toward a formal semantics of the overall UML models; see the book [France and Rumpe (1999)] and the report on a recent workshop on the topic of the UML semantics [S.Kent et al. (1999)] also for more references). The only exception known to us is the

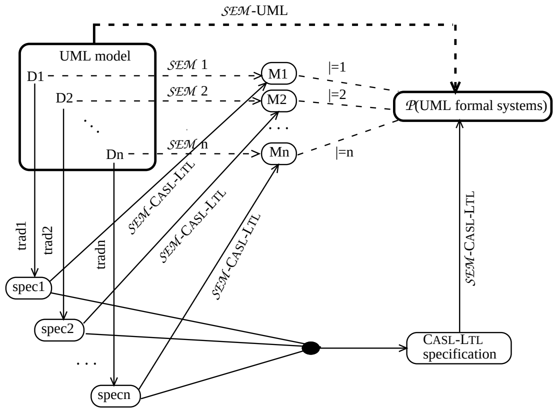


Figure 2:

attempt at describing the semantics of the UML within the UML itself (the *meta-model* approach) as advocated by the pUML group (see the site `http://www.cs.york.ac.uk/puml/`). But even in this case it is difficult to recognize the nature of the semantics of the individual diagrams, as the semantics is given as a sequence of translations into more and more restricted core languages, that are subsets of the UML, and only the smallest is directly given a semantics.

Our approach, accordingly with the previous discussion, is an attempt at formalizing UML models as a whole, while simultaneously giving also a formalization of each kind of diagram in an integrated way. In Fig. 2, we graphically summarize our proposal.

From a technical viewpoint, we proceed in two steps: first, we determine the needed semantic structures (the $M_i$ and the UML formal systems in Fig. 2) through an analysis of the document UML 1.3, and formally describe them as algebraic structures. Then, we translate the diagrams into CASL-LTL specifications (represented by the downward arrows), whose formal semantics gives, by composition, the semantics of each diagram in the UML model (represented by the dotted horizontal arrows). We will use CASL-LTL as metalanguage to describe the semantics of UML models. CASL-LTL is an extension of the algebraic specification language CASL developed as central part of the `CoFI` initiative[1]

In Sect. 2 and Sect. 3 we will sketch two particularly significant cases, concerning the translation of class diagrams and state machines, illustrating our techniques on a running example which is a fragment of the invoice system (see Allemand et al. (1998)).

[1] See the site `http://www.brics.dk/Projects/CoFI`.

Moreover, in the lower part of the diagram, the CASL-LTL specifications representing the individual diagrams are combined (in a non-trivial way) into an overall specification, whose semantics is (has to be) compatible with the constraints imposed by the individual diagrams and provides a semantics for the overall UML model. This combination is graphically represented by a bullet, and in Sect. 4 we will summarize the current version.

We are currently working on filling the above schema, providing the semantic structures and the translations of the various diagrams into CASL-LTL. This activity is performed as part of the `CoFI` initiative, within the `CoFI`-reactive task group.

# 1 UML Formal Systems

Roughly speaking, there are two aspects of a system that we are able to describe using the UML: the *structure* of the system, that is which are the components and which are their capabilities, and the *activity* of the system, that is the evolution of its components along the time and the interactions of the system with the external world (e.g., with the users). Since the handling of the time in UML, also of the real time, does not require to consider systems evolving in a continuous way, we have to describe a *discrete* sequence of moves, each one of them representing one step of the system evolution.

Therefore, we will use *generalized structured labelled transition systems* (shortly *glts*) as UML formal systems, representing the evolution steps as transitions. Moreover, the labels of the transitions capture interactions with the external world, and the structured states, as sources and targets of the transitions, provide a formal counterpart to the system structure.

Let us shortly describe, in the next subsection, generalized structured labelled transition systems; in Sect. 4, we will formally present which glts are used to model UML systems.

## 1.1 Generalized Labelled Transition Systems

A *generalized labelled transition system* (shortly *glts*) is a 4-tuple $(ST, LAB, INFO, \rightarrow)$, where $ST$, $LAB$, and $INFO$ are sets, and

$$\rightarrow \subseteq INFO \times ST \times LAB \times ST$$

is the *transition relation*. A 4-tuple $(i, s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $i : s \xrightarrow{l} s'$. Classical labelled transition systems are the glts where the set *INFO* has just one element.

Given a glts 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 $i : s \xrightarrow{l} s'$ holds, then it has a node $n'$ decorated with $s'$ and an arc decorated with $i$ and $l$ from $n$ to $n'$.

We model a process P with a transition tree determined by a glts $(ST, LAB, INFO, \rightarrow)$ 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 moving from one situation to another.

It is important to note that here an arc (a transition) $i : s \xrightarrow{l} s'$ has the following meaning: P in the situation $s$ has the *capability* of moving 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, while $i$ is some additional information on the move, not concerning interactions.

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 glts, named *structured glts*. A structured glts is obtained by composing other glts describing such components, say *clts*; its states are built by the states of *clts*, and its transitions from some state $s$ are determined by composing those of *clts* starting from the components of $s$.

So a concurrent process has components that are in turn processes, said in the following *active components*; but, in general, it has also *passive components* (think for example of a buffer and of a shared memory). Here passive means that such components may change their states only as result of some transition of some active component.

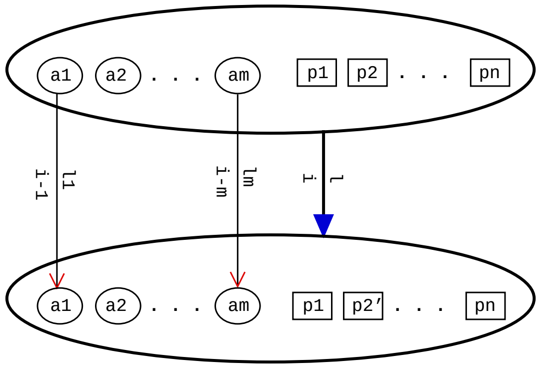


Figure 3: A generic transition of a structured glts

In Fig. 3 we show graphically a generic transition of a structured glts. In such case the system has $m$ active components (represented by ellipses) and $n$ passive components (represented by boxes); and the active components are modelled by a glts. In that transition two active components move (`a1` and `am`), and a passive component (`p2`) is modified as a consequence of that.

A glts may be formally specified by using the algebraic specification language CASL-LTL (see Reggio et al. (1999a)), an extension of CASL (see CoFI (1998)), for the specification of processes based on glts's. 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. Indeed, CASL-LTL extends CASL under two aspects: the logic is enriched by constructs from a branching-time CTL-style temporal logic, and it is possible to declare dynamic sorts as follow:

**dsort** *State* **label** *Label* **info** *Info*

This CASL-LTL construct declares the sorts *State*, *Label*, and *Info* for the states, the labels and the information of the glts, and implicitly also a transition predicate

$$\_ : \_ \xrightarrow{\_} \_ : Info \times State \times Label \times State.$$

Usually a specification of processes in CASL-LTL has the following form:

**spec** GLTS =
    ...... **then**
**free** {
    **dsort** *State* **label** *Label* **info** *Info*
    ......
    **axioms**
    ......
} **end**

Thus each element $s$ of sort *State* in a model $M$ (an algebra or first-order structure) of the above specification GLTS corresponds to a process modelled by a transition tree with initial state $s$ determined by the glts $(Info^M, State^M, Label^M, \_ : \_ \xrightarrow{\_} \_^M)$ [2].

The specification GLTS extends (CASL-LTL keyword **then**) some specifications of some basic data used to define the states, the labels and the information. 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 $i = 1, \ldots, n+1$, $\alpha_i$ is a positive atom (i.e., either a predicate application or an equation), to guarantee the existence of the initial semantics.

## 1.2 UML Formal Systems as glts

In the case of a structured glts corresponding to a UML formal system the components correspond to the class instances. Those of active classes are the active ones, and thus are in turn modelled by a (non-structured) labelled transition system, whereas those of passive classes are the passive components.

**Passive components** The passive objects are passive components.

We simply model passive components by giving their states and the operations for acting on them.

A state of a passive object should be characterized not only by its identity and the (current) values of its attributes, but also by the current calls of the various operations, to handle the 'concurrency' qualifier of operations.

The relevant operations are the creation of a new object (returning the initial state given some parameters), reading or updating the value of an attribute, checking if a call of some operation may start, and starting and ending a call of an operation

Moreover, we have also other passive components storing relevant information, for instance about the current state of associations.

[2] Given a $\Sigma$ algebra $A$, and a sort $s$ of $\Sigma$, $s^A$ denotes the interpretation of $s$ in $A$; similarly for the operation and predicates of $\Sigma$.

**Active components** The active objects are active components, which are modelled by generalized labelled transition systems. Let us here briefly sketch their relevant features.

Each state must at least contain the object identity, the values of the object attributes and the actual time. The labels are as those determined in the state machine part, i.e., they are triples consisting of a set of inputs (e.g., to receive signals or operation calls from other objects), a set of outputs (e.g., to update attributes of other objects, or to send signals to other objects) and the received time (received from some clock (see Rumbaugh et al. (1999) p. 475)). We have no restriction on the transitions.

**States** The states of the overall system will be sets of states of the components. Indeed, we do not need multisets, since we can always distinguish the components corresponding to objects, as their states include their identities, and those that do not correspond to objects, because there will be at most one component for each kind of information.

**Labels** A label of a UML formal system will contain a list of received/sent operation calls (also received/sent signals) with *external* users. The external users correspond, for instance, to the small human like pictures in the execution diagrams, or to the external interactions considered in the use cases.

**Transitions** A transition of a UML formal system is a combination of groups of transitions of the active components (recall that there is always at least one active component corresponding to the main) closed by "synchronization", i.e., s.t. each "send" should be together the corresponding "receive".

To accommodate an implicit semantic variation point, the system should in the end result in a free-parallel system (i.e., a system where at each step any group of active components that are capable of performing some transitions may move).

The matching pairs of sent-received information correspond to the stimuli (messages) considered by UML 1.3, precisely: exchange of an operation call; exchange of the return of an operation call (not considered here, as well as synchronous calls); exchange of a signal; creation/destruction of objects.

**Information** To be able to evaluate the satisfaction of an execution diagram by an UML formal system M we need to record for each transition of M which are the exchanged stimuli during such transition. Thus, the "Information" component of the generalized transitions will be a set of stimuli.

## 2 UML Class Diagrams

The UML class diagrams carry information about the static structure of the systems, that is about the typing, i.e., which classes are used to classify the components of the systems, the available attributes and operations of each class and the relationships (associations) among classes.

Classes and data types denote sets of values. In particular, those corresponding to classes are sets of object states, that are, roughly speaking, tuples of attribute current values and the identity of the object, so far as the class diagram is considered in isolation.

Analogously, associations, or more precisely association states, are represented by the sets of tuples (pairs) of the objects related by them.

A notion of *global state*, recording the currently living instances of classes, is needed, as well as the information about static semantics, that are accessible through the OCL.

Operations (but queries) describe how a (global) state is modified by their performances, possibly resulting into a value as well. Since several activities can take place concurrently, operations should be represented by non-deterministic functions. However, as we prefer to use first-order structures as models, we represent non-deterministic functions by predicates. Indeed, an operation relates its inputs (the explicit arguments and the implicit parameters, like the call recipient and the global state) to the possible outputs (the result, if any, and the modified state) and the actual result of an individual operation call is chosen, among the possible ones, that is among those related to that particular sequence of inputs, accordingly to the evolution of the other components of the system.

Let us not linger on the models, but directly sketch the algebraic specification describing them; its structure is represented in Fig. 4.

First of all, such specification has four parts.

Two of them, the *generic context part* and the *state part*, are generic, in the sense that they are common to all class diagrams. The *generic context part*, CONTEXT, includes, for instance, the sorts, operations and predicates used to deal with the class system and its typing. The *state part*, STATE, concerns the form of global and (generic) local states.

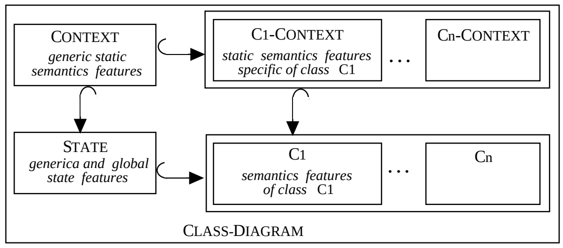


Figure 4:

The other two parts, the *class-context part* and the *class-semantics part* are specific of an individual class diagram and each of them is the sum of smaller specifications, one for each classifier in the class diagram. This structure is chosen in order to reflect as much as possible the structure of the UML model. Analogously to the generic parts, the *class-context part* for a classifier C, C-CONTEXT, introduces the information about the static semantics, like, for instance, the names of each class, of its attributes and operations, while the *class-semantics part*, C, introduces, for instance, the local states of each class or association.

### 2.1 Generic Parts

In the following, we intersperse fragments of the specification with explanations. The overall specifications include the lines proposed here and other analogous parts, that we omit for brevity.

We will use, that is, include in our specifications, some basic data types, adapted from the standard library for CASL, like INTEGER, with the (main) sort *Integer* or DATE, with the (main) sort *Date*, and some *parametric* data type, adapted from the standard library for CASL as well, like LIST[_] and SET[_], where _ is the place holder for the parameter.

**Context part** First of all, we introduce data types for dealing with values and types. The former collect the standard OCL values, like booleans or numbers, and, in particular, the identites of objects, *Ident*. The latter are required in order to translate some OCL constraints and include the names of the standard types as well as the names of classes, *Name*. Types will be used in the following to check the static semantics requirement for the correctness of an operation call. For this aim, in particular, we also need an operation verifying that a list of values matches the list of expected types.

**sorts** *Ident, Name*
**pred** $isSubType : Name \times Name$
$hasType : Ident \times Name$
**vars** $id : Ident;\ c, c' : Name;$
**axioms**
$isSubType(c, c') \wedge hasType(id, c) \Rightarrow hasType(id, c')$
**type** $Value ::=$ **sort** $Bool$ | **sort** $Ident$ | **sort** $Integer$ | **sort** $Date \ldots$
$Type ::= bool$ | **sort** $Name$ | $integer$ | $date \ldots$
**pred** $matches : List[Value] \times List[Type]$
**vars** $lv : List[Value];\ tv : List[Type];$
$v : Value;\ t : Type;$
$id : Ident;\ c : Name;$
**axioms**
$matches([], [])$
$\neg matches(v\ lv, [])$
$\neg matches([], t\ tv)$
$v \in Bool \Rightarrow$
$matches(v\ lv, t\ tv) \Leftrightarrow$
$matches(lv, tv) \wedge t = bool$
$v \in Ident \Rightarrow$
$matches(v\ lv, t\ tv) \Leftrightarrow$
$id = v\ as\ Ident \wedge$
$c = t\ as\ Name \wedge$
$matches(lv, tv) \wedge hasType(id, c)$
$v \in Integer \Rightarrow$
$matches(v\ lv, t\ tv) \Leftrightarrow$
$matches(lv, tv) \wedge t = integer$
$v \in Date \Rightarrow$
$matches(v\ lv, t\ tv) \Leftrightarrow$
$matches(lv, tv) \wedge t = date$

Other static information have to be recorded as well, for instance which names correspond to attributes, operations, classes, (binary) associations, expected types of the operation input and output, types of the attributes, etc. In the following, we will assume that the names used to represent attributes, operations and classes are all distinct.

**pred** $isaClass : Name$
$isAttr : Name \times Name$
**ops** $type : Name \rightarrow ?\ Type$
**pred** $isanOp : Name \times Name$
**ops** $argType : Name \rightarrow ?\ List[Type]$
$resType : Name \rightarrow ?\ List[Type]$
**pred** $isAnAssociation : Name$
$isBinAssociation : Name$
**vars** $n : Name;$
**axioms** $isBinAssociation(n) \Rightarrow isAnAssociation(n)$
**ops** $assType : Name \rightarrow ?\ List[Type]$
$LType, RType : Name \rightarrow ?\ Type$
**axioms** $isBinAssociation(n) \Rightarrow assType(n) = LType(n) :: RType(n)$

The treatment of other kind of classifiers, e.g., signals, that is completely analogous, is omitted.

**State part** Extending the CONTEXT specification, we introduce the constructs to deal with states.

The local states of objects (sort *Object*) provide information about the identity of the objects and the current values of their attributes. We can read and write attributes, with the usual properties about typing (e.g., we cannot assign to an attribute a value of an incompatible type, or updating an attribute is not affecting the others nor the object identity).

**sorts** *Object*
**ops** $getIdent : Object \rightarrow Ident$
$getAttr : Object \times Name \rightarrow ?\ Value$
$setAttr : Object \times Name \times Value \rightarrow ?\ Object$
**vars** $o, o' : Object;\ v : Value;$
$a, a' : Name;\ c : Name;$
**axioms**
$setAttr(o, a, v) = o' \Rightarrow getAttr(o', a) = v$
$setAttr(o, a, v) = o' \Rightarrow getIdent(o') = getIdent(o)$
$setAttr(o, a, v) = o' \wedge \neg a' = a \Rightarrow getAttr(o', a') = getAttr(o, a')$
$def\ setAttr(o, a, v) \Leftrightarrow (matches(v\ , type(a)\ ) \wedge isAttr(a, c) \wedge hasType(getIdent(o), c))$
$def\ getAttr(o, a) \Leftrightarrow isAttr(a, c) \wedge hasType(getIdent(o), c))$

The system state gives information about the living instances of the classes, that are needed, in particular, in order to describe possible side effects of operation calls.

**sorts** *State*
**pred** $knownIn : Ident \times State$
**ops** $localState : Ident \times State \rightarrow ?\ Object$
$allInstancesOf : Name \times State \rightarrow ?\ Set[Ident]$
**vars** $id : Ident;\ gs : State;\ c : Name;$
**axioms**
$def\ localState(id, gs) \Leftrightarrow knownIn(id, gs)$
$isin(id, allInstancesOf(c, gs)) \Leftrightarrow hasType(id, c)$

Each operation of a class represents a non deterministic (partial) function having as input the operation owner (that is implicit in object-oriented approaches), the global state (in order to use information on the other objects that can be reached by navigation) and the explicit parameter list; the output is the result (if any) and the new global state, possibly modified by some side effect of the operation call. We formalize operations by the predicate named *call* relating the name of each operation and the inputs to the possible outputs.

**pred** $call : Ident \times Name \times State \times$
$\quad List[Value] \times List[Value] \times State$
**vars** $gs\_in, gs\_out : State;$
$\quad lv\_in, lv\_out : List[Value];$
$\quad id : Ident;\ op : Name;\ c : Name;$
**axioms**
$$call(id, op, gs\_in, lv\_in, lv\_out, gs\_out) \Rightarrow matches(lv\_in, argType(op)) \wedge matches(lv\_out, resType(op)) \wedge (\exists\ c : isanOp(op, c) \wedge hasType(id, c))$$

The local states of the associations are represented by a sort; we will introduce a constant of sort *Name* to represent the name of the association in the specification of each actual association, so that we can retrieve its local state from the global state.

**sorts** $Association$
**ops** $AssocState : Name \times State \to ?\ Association$
**vars** $a : Name;\ gs : State;$
**axioms** $def\ localState(a, gs) \Leftrightarrow knownIn(a, gs)$

Since the case of binary associations is by far the most common in the UML, we are also providing a specialization of the association type for such case. In the following specification we add, for each association end, an operation yielding the objects that are in relation with a given object that will be used to represent the UML *navigation*.

**sorts** $BinAssocState < Association$
**ops**
$\quad LAssoc : Ident \times BinAssocState \to Set[Ident]$
$\quad RAssoc : Ident \times BinAssocState \to Set[Ident]$
**pred**
$\quad isInAssociation : Ident \times BinAssocState \times Ident$
**vars** $id, id' : Ident;\ as : BinAssocState;$

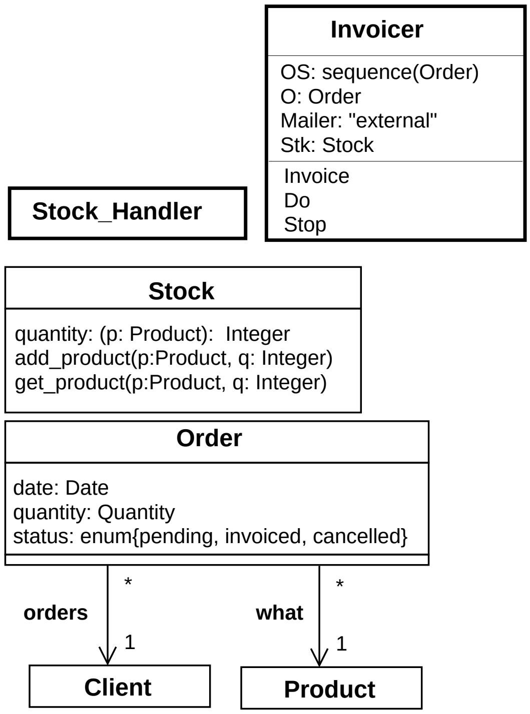


Figure 5: Class Diagram for the Invoice system

**axioms**
$$isin(id', RAssoc(as, id)) \Leftrightarrow isInAssociation(id, as, id')$$
$$isin(id, LAssoc(as, id')) \Leftrightarrow isInAssociation(id, as, id')$$
$$isInAssociation(id, as, id') \Rightarrow hasType(id, LType(id))$$
$$isInAssociation(id, as, id') \Rightarrow hasType(id', RType(id))$$

The complete specification of this part, which we have just partly sketched here, is called STATE.

## 2.2 Class specific parts

Let us start by illustrating our running example: a fragment of an invoice system, that we will use to illustrate more concretely how the specification corresponding to a class diagram looks like.

The class diagram of the invoice system is reported in Fig. 5.

We have some passive classes, recording information about clients, products (we do not detail these parts), current (and past) orders and stock of an e-commerce firm, and some active classes, managing the above described data and representing two kinds of "software" clerks: the stock handler, who put the newly arrived products in the stock and remove the correct amount of products to settle an order, and the invoicer, who processes orders and send invoices.

The parts of the specification strictly depending on the individual class diagram are hierarchically built starting from smaller specifications reflecting the structure of the class diagram as much as possible and in particular we aim at having a specification for each class and a specification for each association.

**Class Specific Context Part** Each class contributes to the context by the names introduced by the class. So, for instance, the specification corresponding to the class `Client` introduces only its name, as follows.

**spec** CLIENT-CONTEXT = CONTEXT
**then**
**ops** $Client :\to Name$
**axioms** $isaClass(Client)$

If there are attributes, like in the case of the class ORDER, then we add also the names of the attributes and their typing. We also have to provide a name for the enumeration type and to add it to the admissible values. Thus, we have to provide some clauses for the specification of operations and predicates having *Value* as input, for instance of the *matches* predicate.

**spec** ORDER-CONTEXT = CONTEXT
**then**
**type** $StatusType$ ::= $pending \mid invoiced \mid cancelled$
**sorts** $StatusType < Value$
**ops** $StatusType :\to Type$
**vars** $v : Value;\ t : Type;$
$lv : List[Value];\ tv : List[Type];$
**axioms**
$v \in StatusType \Rightarrow$
$matches(v\ lv, t\ tv) \Leftrightarrow$
$matches(lv, tv) \wedge t = StatusType$
**ops** $Order :\to Name$
$date :\to Name$
$quantity :\to Name$
$status :\to Name$
**axioms** $isaClass(Order)$
$isAttr(date, Order)$
$type(date) = date$
$isAttr(quantity, Order)$
$type(quantity) = Quantity$
$isAttr(status, Order)$
$type(status) = StatusType$

Analogously we deal with the case of operations. Since the type of some operation of the class `Stock` involves other classes, for instance `Product`, we declare the constant with that name, but we do not state that it is a class, as this information is not part of the class `Stock`. When in the end we will add this specification with that corresponding to the static context of `Product`, we will have the missing information.

**spec** STOCK-CONTEXT = CONTEXT
**then**
**ops** $Stock :\to Name$
$Product :\to Name$
$quantity :\to Name$
$addProduct :\to Name$
$getProduct :\to Name$
**axioms** $isaClass(Stock)$
$isanOp(quantity, Stock)$
$argType(quantity) = Product$
$resType(quantity) = integer$
$isanOp(addProduct, Stock)$
$argType(addProduct) = Product\ integer$
$resType(addProduct) = []$
$isanOp(getProduct, Stock)$
$argType(getProduct) = Product\ integer$
$resType(getProduct) = []$

Associations are dealt with in an analogous way with respect to classes.

Let us, for instance, consider the specification corresponding to the association **orders**.

**spec** ORDERS-CONTEXT =
CONTEXT **and** ORDER-CONTEXT **and** CLIENT-CONTEXT
**then**
**ops** $orders :\to Name$
**axioms** $isAnAssociation(orders)$
$LType(orders) = Order$
$RType(orders) = Client$

**Class semantics part** Each specification corresponding to a class introduces (at least) the sort of the local states of that class objects.

**spec** CLIENT = STATE **and** CLIENT-CONTEXT
**then**
**sorts** $Client < Object$
**vars** $o : Object;$
**axioms**
$hasType(getIdent(o), Client) \Leftrightarrow o \in Client$

Active classes are translated in a slightly different way. Indeed, their object sort is declared as *dynamic*, in order to provide means to describe, in the semantics of the overall system, their object evolution.

**spec** STOCKHANDLER0 = STATE
**then**

**dsort** *StockHandler*
  **label** *StockHandler-Label*
  **info** *StockHandler-Info*
**sorts** *StockHandler* $<$ *Object*
**vars** $o$ : *Object*;
**axioms**
  $hasType(getIdent(o), StockHandler) \Leftrightarrow$
    $o \in StockHandler$

Also in this part associations are dealt with in an analogous way with respect to classes. Let us, for instance, consider the specification corresponding to the association `orders`.

**spec** ORDERS = STATE **and** ORDER **and** CLIENT
**then**
**sorts** *Orders* $<$ *BinAssocState*
**vars** $gs$ : *State*; $o$ : *Ident*; $ord$ : *Orders*;
**axioms** $AssocState(orders, gs) \in Orders$
  $size(RAssoc(o, ord)) = 1$

Two points worth to keep in mind are that in CASL there is the principle "same-name same-thing" imposing that the realizations of sorts (functions) [predicates] with the same name in different parts of the same overall specification must coincide. Thus, for instance, the semantics of the basic parts that are imported by the specifications representing individual classes must be unique, so that we will have just one global state.

The second point is that the models of the overall specification are not required to be built from models of the individual subspecification, that is the structure of the specification is not reflected onto the architecture of the model. Therefore, the choice about the structuring of the information, for instance by layering the specifications representing individual classes, does not affect the semantics we are proposing for a class diagram, but only its *presentation*.

## 3 UML State Machines

In Reggio et al. (1999b) we present our complete formalization of UML state machines using CASL-LTL, and a short version has been presented in Reggio et al. (2000); here we briefly report the main ideas.

Assume to have a given active class ACL with a given associated state machine SM. The instances of ACL, called using a UML terminology *active objects*, are just processes and we model them by using a glts. We build such glts, named in the following *GLTS*, and specify it algebraically with the specification ACL-DYNAMIC. In Fig. 6 we report an example of a state machine, precisely the one associated with the active class Invoicer of the class diagram in Fig. 5.

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

We assume that all "static" information about the class ACL (and the others), like for instance which are the attributes/operations of ACL, which are their types, which are found in the class diagram, are given in the specification CONTEXT, defined before in Sect. 2.

Accordingly to the method described in Larosa and Reggio (1997) we determine the specification ACL-DYNAMIC by analyzing the nature of *GLTS* in several steps as follows.

**Is *GLTS* simple or structured?** To decide whether *GLTS* is simple or structured, we need to know whether 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 for what concerns this point. Indeed, somewhere it suggests that there is exactly one thread, otherwise, it assumes that there are many threads. 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 *GLTS* to be

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

Here, we take the second choice.

**Determining the granularity of the *GLTS*-transitions** We model an active object by means of a glts, that means that we model the behaviour of such object by splitting it into "atomic" pieces (the *GLTS*-transitions). Thus, to define *GLTS*, we must first determine the granularity of this splitting.

We assume that each *GLTS*-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.

**Determining the *GLTS*-Labels** The *GLTS*-labels (labels of the glts *GLTS*) describe the possible interactions/interchanges between the active objects of class ACL and their external environment (the other objects comprised in the model).

A *GLTS*-label, which formalizes the interactions with the external environment happening during an *GLTS*-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** ACL-LABEL =
  TIME **and** FINITESET[ACL-INPUT] **and**
  FINITESET[OUTPUT] **then**
  **free type** *Acl-Label* ::=
    $\langle \_, \_, \_ \rangle$(*FinSet*[*Acl-Input*];
      *Time*;
      *FinSet*[*Output*])

As a result of a careful scrutiny of UML 1.3 we can deduce that the basic ways the active objects interact with the other objects are the following, classified in "input" and "output":

**input:**

- to receive a (synchronous/asynchronous) operation call from another object
- to receive a signal from another object
- to be destroyed by another object
- to read an attribute of another object
- to have an attribute updated by another object

**output:**

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

**spec** ACL-INPUT =
  IDENT **and** NAME **and**
  LIST[VALUE] **and** CONTEXT **then**
**free** {
  **type** *Acl-Input* ::=
    *I_scall*(*Ident*; *Name*; *List*[*Value*])? |
    *I_acall*(*Ident*; *Name*; *List*[*Value*])? |
    *I_return*(*Ident*; *Name*; *Value*)? |
    *I_send*(*Ident*; *Name*; *List*[*Value*])? |
    *I_destroy*(*Ident*)? |
    *I_read*(*Ident*; *Name*; *Value*)? |
    *I_update*(*Ident*; *Name*, *Value*)?
  **axioms**
    $isaClass(id, ACL) \wedge isanOp(ACL, op) \wedge$
    $matches(argType(op), vl) \Rightarrow$
    $def\ (I\_scall(id, op, vl))$
    .......
}

**spec** OUTPUT =
  IDENT **and** NAME **and**
  LIST[VALUE] **and** CONTEXT **then**
**free** {
  **type** *Output* ::=
    *O_scall*(*Ident*; *Name*; *List*[*Value*])? |
    *O_acall*(*Ident*; *Name*; *List*[*Value*])? |
    *O_send*(*Ident*; *Name*; *List*[*Value*])? |
    *O_create*(*Ident*; *Name*)? |
    *O_destroy*(*Ident*; )? |
    *O_read*(*Ident*; *Name*; *Value*)? |
    *O_update*(*Ident*; *Name*, *Value*)?
  **axioms**
    .......
}

**Determining the *GLTS*-States** The *GLTS*-states (states of the glts *GLTS*) 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 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 *GLTS*-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 *GLTS*-states are thus specified by the following Casl-Ltl specification.

**spec** Acl-State =
  Ident **and** Configuration **and**
  Attributes **and** Acl-Event_Queue **and**
  History **and** ChangeInfo **then**
  **free type** *Acl* ::=
    $\_\_ : \langle \_\_ \_\_ \_\_ \_\_ \_\_ \rangle (\mathit{Ident};$
      $\mathit{Configuration};$
      $\mathit{Attributes};$
      $\mathit{Acl\text{-}Event\_Queue};$
      $\mathit{History};$
      $\mathit{ChangeInfo}) \mid$
    $\_\_ : \mathit{terminated}(\mathit{Ident})$

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. Here we do not report the specifications of the various parts of the state (they can be found in Reggio et al. (1999b)).

**Determining the Information** The information on the transitions of *GLTS* describes which stimuli have been generated in such transitions, not detectable by looking at the transition labels, precisely start and end of self calls, self-sent signals, and dispatched events. For brevity, we do not report here Acl-Info, its specification (see Reggio et al. (1999b)).

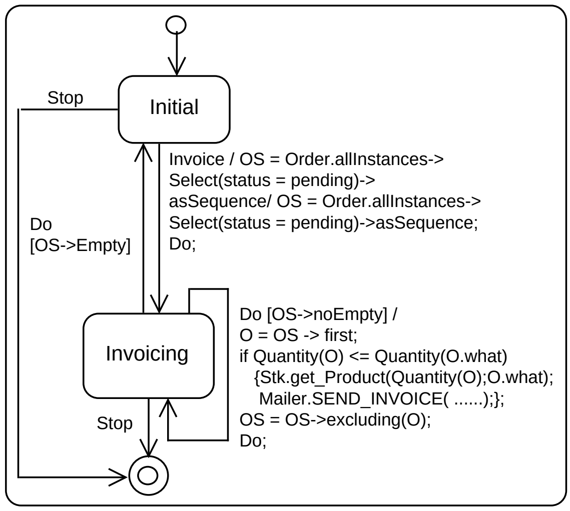


Figure 6: The State Machine Associated with Invoicer

**Determining the *GLTS*-Transitions** A *GLTS*-transition, i.e., a transition of the glts *GLTS*, corresponds to one of the following

1. to dispatch an event,
2. to execute an action,
3. to be destroyed by dispatching a special event,
4. 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.

It is important to notice that the *GLTS*-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. 7 the transition tree associated with the state machine of Fig. 6 (to simplify the picture we only report the configuration and the event queue of each *GLTS*-state). There it is possible to see that one state machine transition corresponds to a (unnecessarily contiguous) sequence of *GLTS*-transitions.

The *GLTS*-transitions are formally defined by the axioms of the following specification (recall that because *Acl* is a dynamic sort, we have also the implicitly declared transition predicate $\_\_ : \_\_ \xrightarrow{\_\_} \_\_ : \mathit{Acl\text{-}Info} \times \mathit{Acl} \times \mathit{Acl\text{-}Label} \times \mathit{Acl}$).

**spec** Acl-Dynamic =
  Acl-Label **and** Acl-State **and** Acl-Info **then**

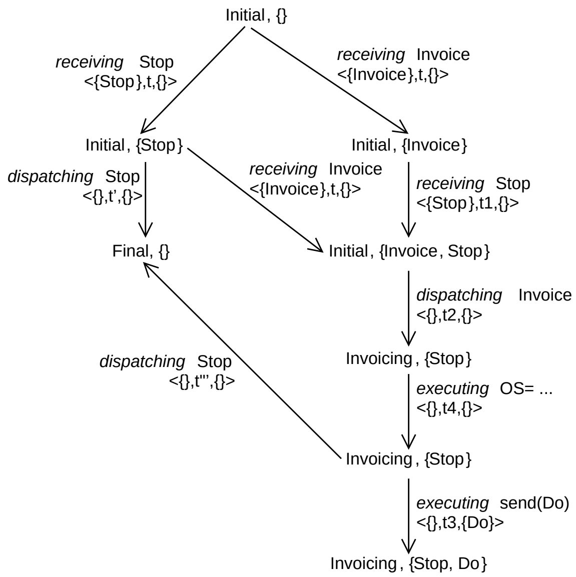


Figure 7: A fragment of a transition tree

**free** {
  **dsort** *Acl* **label** *Acl-Label* **info** *Acl-Info*
  **axioms**
    ......
    $cond \ \Rightarrow \ i : s \xrightarrow{l} s'$
    ......
} **end**

To master complexity, 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. Here, we report only the axiom defining the transitions corresponding to dispatch an event, for the other see Reggio et al. (1999b). .

**Dispatching an Event** 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*), and *outs* is a set of outputs corresponding to have an attribute read (checked by Read_Attributes_ACL), then *GLTS* has a transition, with the label made by the received inputs *ins*, the time *t*, and the sent outputs *outs*, 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.

$\mathsf{Dispatch}(ev, conf, e\_queue, ins, attrs, id) =$
$conf', e\_queue'$

means that dispatching event $ev$ in the configuration $conf$ changes it to $conf'$ and changes $e\_queue$ to $e\_queue'$.

$$\begin{array}{l}
not_frozen(conf) \ \wedge \\
dispatchable(ev, e_queue) \ \wedge \\
\mathsf{Read_Attributes_ACL}(outs, attrs) \ \wedge \\
\mathsf{Dispatch}(ev, conf, e_queue, ins, attrs, id) = \\
conf', e_queue' \quad \wedge \\
\mathsf{ChangeEvs}(ins, attrs, id, chinf) = \\
ch\text{-}evs, chinf' \quad \Rightarrow \\
Dispatched(ev) : \longrightarrow \\
id : \langle conf, attrs, e_queue, history, chinf \rangle \xrightarrow{\langle ins, t, outs \rangle} \\
id \quad : \quad \langle attrs', conf', e_queue'', history', chinf' \rangle
\end{array}$$

where

$$\begin{array}{l}
attrs' = apply(\mathsf{Updates}(ins), attrs) \\
history' = \langle active_states(conf), t \rangle \ \& \ history \\
e_queue'' = \\
put(\mathsf{TimeEvs}(attrs, id, history, t) \cup ch\text{-}evs \cup \\
\mathsf{Events_Of}(ins), e_queue')
\end{array}$$

## 4 Semantics of UML models

The semantics of a whole UML model is finally given by the combination of the specifications giving the semantics of the single views determined by the various diagrams that compose the model.

Summarizing we have

- the "static view", i.e., the class diagram (for simplicity we have assumed to have a unique class diagram), which defined a specification named CLASS_DIAGRAM (see Sect. 2) providing both context information and a model for the objects and the associations among them.
- some local dynamic views, i.e., some state machines associated with active classes, say $\mathsf{ACL}_1, \ldots, \mathsf{ACL}_p$, which defined some specifications named ACL-DYNAMIC$_1$, ..., ACL-DYNAMIC$_p$ giving the glts's describing the behaviour of the instances of such classes.

For each active class in the model without an associated state machine, say $\underline{\mathsf{ACL}}_1, \ldots, \underline{\mathsf{ACL}}_q$, we define the specifications of the glt's describing the behaviour of their instances.

**spec** $\underline{\text{ACL}}$-DYNAMIC =
ACL-LABEL **and** ACL-STATE **and** ACL-INFO **then**
**dsort** $\underline{Acl}$ **label** $\underline{Acl}$*-Label* **info** $\underline{Acl}$*-Info*

where the specifications of the labels, of the states, and of the information (ACL-LABEL, ACL-STATE,ACL-INFO) are defined as in Sect. 3. Here we have no axioms, because in the UML model there are no views describing the behaviour of the instances of this class.

The specification STIMULUS of the stimuli is the the trivial translation into CASL of the UML stimuli (see UML 1.3 p. 2-86) and hence we do not report it.

The specification SYSTEM-LABEL of the labels of the system can only be given by analyzing those diagrams where the interactions with the external environment. Since we are not presenting here those kind of diagrams (e.g., sequence and use case) we also omit this specification.

In the following specification SYSTEM we introduce the sort of the overall system state, that is a set of objects and associations, extending the specifications corresponding to class diagrams and to active classes.

CLASS_DIAGRAM **and**
ACL-DYNAMIC$_1$ **and** ... **and** ACL-DYNAMIC$_p$ **and**
$\underline{\text{ACL}}$-DYNAMIC$_1$ **and** ... **and** $\underline{\text{ACL}}$-DYNAMIC$_q$ **and**
SYSTEM-LABEL **and** FINITESET[STIMULUS] **then**
**dsort** *State* **label** *System-Label*
**info** *FinSet*[*Stimulus*]
**type** *State* ::=
$\Lambda \mid \_||\_(Object; State) \mid \_||\_(Association; State)$

Then we have to introduce some axioms in order to put together the information given by the different parts. For instance, we give an axiom stating that the initial and final states of a sequence of transitions corresponding to an operation call must be related by that operation already in the specification corresponding to the class diagram.

$$\begin{array}{l} i : s \xrightarrow{l} s' \wedge Start\_Call(i, id, op, vl) \wedge \\ in\_one\_case(s', \\ \{I.\neg End\_Call(I, id, op, vl)\}\ until \\ (\{X.End\_Call(I, id, op, vl)\} \wedge [\ X \bullet X = s_{\mathrm{F}}\ ])) \\ \Rightarrow \\ call(id, op, s, s_{\mathrm{F}}) \end{array}$$

The intuitive meaning of this axiom that requires the respect of the constraints on the operations put in the class diagram is as follows.

**If** the system in state $s$ performs a transition going into state $s'$ corresponding to start an operation call (checked on the associated information by the predicate *Start_Call*) and from state $s'$ the system may perform a sequence of transitions such that the call is ended when reaching state $s_{\mathrm{F}}$,

**then** the predicate characterizing the execution of the called operation $op$ (defined in the specification CLASS_DIAGRAM) must hold on $s$ and $s_{\mathrm{F}}$.

Though most of the identifications needed are already given by the "same-name same-thing" principle of CASL, we also need axioms stating that an identity is known in a system state iff it corresponds to a component of the system, that can be easily inductively defined or axioms stating that for active classes, getting the identity corresponds to selecting the identity component of their state tuples. There are several of such axioms, that, though quite trivial to be expressed would produce a long boring list.

Much more interesting is the following axiom, that describes how the transitions of the overall system are built starting from the transitions of the individual active components, using several auxiliary functions, that we omit, as they are not interesting.

$$\begin{array}{l} \wedge_{i=1}^{n}\ inf_i : a_i \xrightarrow{l_i} a_i' \wedge \\ Ok\_Labels(l_1 \ldots l_n l) \wedge \\ Read\_Attributes(l_1 \ldots l_n l, o_1 \ldots o_k p\_ss) \wedge \\ Modified(l_1 \ldots l_n l) = \{id_1, \ldots, id_k\} \wedge \\ \wedge_{j=1}^{k}\ getIdent(o_j) = id_j \wedge \\ Changes(l_1 \ldots l_n l, \{o_1, \ldots, o_k\}) = \{o_1', \ldots, o_k'\} \wedge \\ Created(l_1 \ldots l_n l) = \{\overline{o}_1, \ldots, \overline{o}_m\} \Rightarrow \\ Stimuli\_Of(l_1 \ldots l_n l, inf_1, \ldots, inf_n) : \\ a_1 || \ldots || a_n || o_1 || \ldots || o_k || p\_ss || a\_ss \xrightarrow{l} \\ a_1' || \ldots || a_n' || o_1' || \ldots || o_k' || \overline{o}_1, \ldots, \overline{o}_m || p\_ss || a\_ss \end{array}$$

The intuitive meaning of the above axiom is as follows. If the following conditions are satisfied:

- a group of active objects ($a_1$, ..., $a_n$) may perform some transitions, accordingly to the specification of their active classes;
- the inputs and the outputs appearing on their labels, and on the label $l$ of the resulting transition (that represents the interactions of the overall system with he environment during such move) are pairwise matched (i.e., any thing sent is received by the addressee, including the external environment and only sent things are received); this is checked by the predicate *Ok_Labels*;

- the inputs corresponding to read the attributes of passive objects are in accord with their actual values; this is checked by the predicate *Read_Attributes*;
- the identities of the passive objects whose state is modified in this transitions are $id_1, \ldots, id_k$ (determined by the operation *Modified*);
- $o_1, \ldots, o_k$ are their states;
- such states are changed in $o'_1, \ldots, o'_k$ by the moving active objects (determined by the operation *Changes*);
- some new objects $\overline{o}_1, \ldots, \overline{o}_m$ are created (determined by the operation *Created*);

then the whole system consisting, besides the active and passive components described so far, by some other set $a\_ss$ ($p\_ss$) of active (passive) components, may perform a transition labelled by $l$, where the states of the active objects are changed as described by their transitions, and whose information is given by stimuli determined either by the exchanged inputs or by the local information of the moving active objects.

It is interesting to note that there is no guarantee that the specification of the overall system is consistent. Indeed, if, for instance, the constraints imposed by the class diagrams are not met by the behaviour described by the state machines, then the UML model corresponds to no systems and this is shown by the fact that the overall specification is inconsistent.

## 5 Future work and conclusion

The kinds of diagram considered so far are the class diagrams, and the statechart diagrams.

We are currently working on the sequence diagrams. Each sequence diagram is modelled by an *event structure*, where events are sets of UML stimuli, that is, a partial order on sets of UML stimuli. A UML formal system satisfies such an event structure if there is a chain in the partial order that is a path in the transition tree.

Some other kinds of diagrams that we have partly analyzed, and that we conjecture can be added to our schema without major problems, are

- the collaboration diagrams, as they are rather similar to the sequence diagram;
- the activity diagrams, as they are a specialization of the statechart diagrams.

Moreover, we still have to take into account the deployment diagrams, though we do not foresee particular problems for their formalization within our framework, while we are doubtful about the possibility of giving a formal semantics to the use case diagrams, because they are, roughly speaking, too close to natural language descriptions.

We translate diagram annotations as well, currently using the OCL constraints, but we are in some sense parametric w.r.t.such annotations, so that we could easily substitute any other constraint language for OCL.

The mechanisms for self-extension provided by the UML, like stereotypes, are still to be taken into account.

We have presented a formalization of a few kinds of UML diagrams. The main difference with other approaches is that ours is supporting the multiview philosophy, in the sense that each part of a UML model has a proper formalization which is integrated with those of the other parts.

The most interesting result, besides the obvious advancement towards a complete precise UML, is, from our point of view, the clarification of UML achieved by translating it in algebraic specifications.

For instance, even from this small part concerning just two kinds of diagrams, it results clear that the separation of concerns apparently achieved by using class diagrams to describe the system structure and state machines to capture the system dynamics only reaches so far. Indeed, the class diagram imposes restrictions on the dynamic behaviour of the objects, through the constraints on the operations and on the classes. Vice versa, a state machine cannot be considered in isolation, as we need to know if it is associated to an active or to a passive class and which are the operations/signals/attributes of such class and of the other classes.

Moreover, we have also found that we need to know how a UML model interacts with its external environment, in order to describe the labels of the overall system. For instance, in our example the objects of class Invoicer call an operation SEND_INVOICE (see Fig. 6) of some entity belonging to the external environment. We had to introduce the stereotype <<external>> in order to annotate that the object kept in the Mailer attribute corresponds to something external to the system and, for that reason, the calls to SEND_INVOICE have to appear on the label of the system transition.

We expect that furthering our investigation to other kinds of diagrams more relationships among the parts of a UML model will be exposed, deepening our understanding of the UML.

## References

Allemand, M., Attiogbe, C., and Habrias, H., editors (1998). *Proc. of Int. Workshop "Comparing Specification Techniques: What Questions Are Prompted by Ones Particular Method of Specification". March 1998, Nantes (France).* IRIN - Universite de Nantes.

France, R. and Rumpe, B., editors (1999). *UML'99 - The Unified Modelling Language.* Number 1723 in Lecture Notes in Conputer Science. Springer Verlag.

Larosa, M. and Reggio, G. (1997). A Graphic Notation for Formal Specifications of Dynamic Systems. Technical Report DISI–TR–97–3, DISI – Università di Genova, Italy. Full Version.

The CoFI Task Group on Language Design (1998). CASL Summary. Version 1.0. Technical report. Available on `http://www.brics.dk/Projects/CoFI/`.

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

Reggio, G., Astesiano, E., Choppy, C., and Hussmann, H. (1999b). A CASL Formal Definition of UML Active Classes and Associated State Machines. Technical Report DISI-TR-99-16, DISI – Università di Genova, Italy. Revised March 2000. Available at `ftp://ftp.disi.unige.it/person/ReggioG/` as `Reggio99b.ps`.

Reggio, G., Astesiano, E., Choppy, C., and Hussmann, H. (2000). Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science. Springer Verlag, Berlin.

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

S.Kent, A.Evans, and Rumpe, B. (1999). UML Semantics FAQ . In Moreira, A. and Demeyer, S., editors, *ECOOP'99 Workshop Reader*, Lecture Notes in Conputer Science. Springer Verlag, Berlin.

UML Revision Task Force (1999). *OMG UML Specification.* Available at `http://uml.shl.com`.