Towards a Rigorous Semantics of UML Supporting its Multiview Approach*

G. Reggio – M. Cerioli – E. Astesiano

DISI - Università di Genova (Italy)
email: {reggio,cerioli,astes}@disi.unige.it

Abstract. We discuss the nature of the semantics of the UML. Contrary to the case of most languages, this task is far from trivial. Indeed, not only the UML notation is complex and its informal description is incomplete and ambiguous, but we also have the UML *multiview* aspect to take into account. We propose a general schema of the semantics of the UML, where the different kinds of diagrams within a UML model are given individual semantics and then such semantics are composed to get the semantics on the overall model. Moreover, we fill part of such a schema, by using the algebraic language CASL as a metalanguage to describe the semantics of class diagrams, state machines and the complete UML formal systems.

Introduction

UML is the object-oriented notation for modelling software systems recently proposed as a standard by the OMG (Object Management Group), see e.g., [12, 14]. The semantics of the UML, which has been given informally in the original documentation, is a subject of hot debate and of lot of efforts, much encouraged by the OMG itself. Not only users and tool developers badly need a precise semantics, but the outcome of the clarification about semantics is seen of great value for the new version of UML to come.

A most notable effort, which likely preludes to a proposal for semantics to the OMG, is a very recent feasibility study in [3] for an OO meta modelling approach. There are serious reasons why that kind of approach seems the right one for an official semantic definition, if any; in particular the need of seeing UML as an evolving family of evolving languages (that requires a very modular approach) and the request of a notation well-known to the UML users (in this case the meta language would be part of UML).

What we present here is not a proposal for a standard definition and is partly orthogonal and partly complementary to the above direction of work; we even believe, as we will argue later on, that it provides insights and techniques that can be used to fill some scenes in the big fresco of [3].

Essentially our contribution is twofold:

* Partially supported by Murst - Programma di Ricerca Scientifica di Rilevante Interesse Nazionale Saladin and by CoFI WG, ESPRIT Working Group 29432.

- we propose a general schema of what a semantics for UML should provide from a logical viewpoint, combining the local view of a single diagram with the general view of an overall UML model;
- we give hints on how to fill some relevant parts of that schema, especially showing how the dynamic aspects fit in.

In this work we adopt rather classical techniques, like modelling processes as labelled transition systems as in CCS et similia, and algebraic specification techniques, supported by a recently proposed family of languages, the `CoFI` family[1] [6]. In particular, since we have to model also the behavioural aspects, we use CASL-LTL [8], which is an extension of CASL, the basic `CoFI` language, especially designed for specifying behaviours and concurrent systems.

We are convinced that explaining the semantics of the UML in terms of well-known techniques helps the understanding of the UML; indeed, as we have shown already in [10], this analysis can easily lead to discover ambiguities and inconsistencies, as well as to highlight peculiarities.

Apart from the work already mentioned [3], several attempts at formalizing the UML have been and are currently under development. Usually they deal with 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, because their final aim is the application of existing techniques and tools to the UML (see [4, 13] and the report on a recent workshop on the topic of the UML semantics for the behavioural part [11], also for more references).

In Sect. 1 we will discuss our understanding of the nature of the expected semantics for the UML. In Sect. 2 and 3 we will sketch the semantics of class diagrams and state machines, illustrating our techniques on a running example which is a fragment of an invoice system. Then, in Sect. 4 we will summarize the non-trivial combination of the CASL-LTL specifications representing the semantics of individual diagrams, to get the overall semantics for a UML model.

We want to emphasize that the main contributions of the work presented here are not bound to the particular metalanguage that we used, namely CASL-LTL; what is essential is the overall structure of the semantics and the use of labelled transition systems for modelling behaviour. Still the nice structuring features of CASL-LTL have much facilitated our investigation.

1 A View of the Overall Semantics

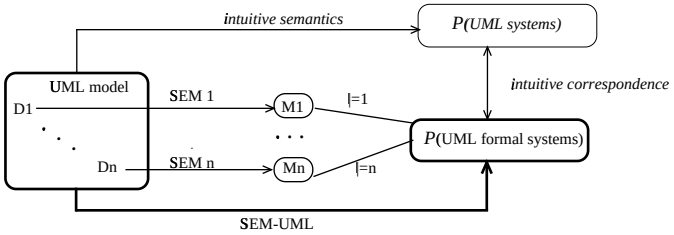
A UML model consists of a bunch of diagrams of different kinds, expressing properties on different aspects of a system. In the following we will call UML-*systems* the "real world" systems modeled by using the UML (some instances are information systems, software systems, business organizations). Thus 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) is informally described

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

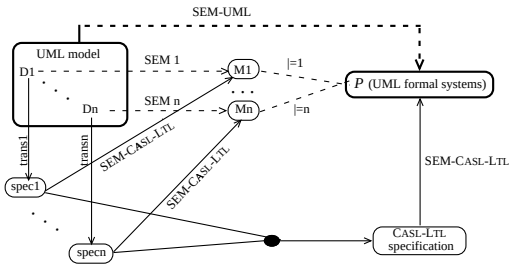
by [14] in isolation, as well as the semantics of each axiom in logic, and its effect on the description of the overall UML-system is to rule out some elements from the universe of all possible systems (semantic models). Indeed, both in the case of a UML model and of a collection of axioms, each individual part (one diagram or one axiom) describes a point of view of the overall system.

Therefore, our understanding of what should be a general schema for a semantics of the UML is illustrated in the following picture, 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 SEM-UML, is a class of UML formal systems. But, each diagram in the model has its own semantics (denoted by the indexed 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, [14].

As a technical solution, illustrated in the following picture

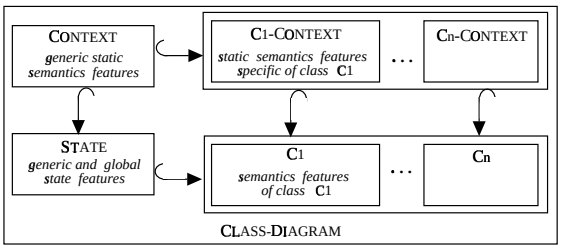


we propose to translate the diagrams (possibly of different kinds) into a common metalanguage, whose formal semantics gives, by composition, the semantics of each diagram in the UML model. Such semantics must correspond to that informally described in [14] and hence the translations are driven by a careful analysis of such document. Moreover, the metalanguage has to provide constructs to compose the translations of the diagrams in order to get an expression representing the overall UML model. As in the case of the individual diagrams, the (metalanguage) semantics of such expression gives a rigorous semantics to the UML model and hence the composition as well has to be driven by [14]. Because

of the need of integrating the formalization of both the static and the dynamic parts of UML, we have found convenient to use as metalanguage an extension of the CASL basic language [6], namely CASL-LTL [8], especially devised to model formally also the dynamic behaviour of the objects.

2 UML Class Diagrams

Let us sketch the algebraic specification describing a given class diagram; its structure is graphically shown below and reflects the logical organization of the information given by the class diagram itself.



Indeed, 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 concepts of object-oriented modeling. The *state part*, STATE, concerns the form of global and (generic) local states.

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, for each classifier C, C-CONTEXT introduces the information about the static semantics (e.g., names of the class, of its attributes and operations), while C introduces information about the semantics (e.g., local states of classes or associations).

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 generic 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 a structured specification are not required to be built from models of its subspecifications, 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*.

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.

In our specifications, we will use a few standard data types, whose obvious specifications are omitted. Some of them are *parametric*, like for instance LIST[_], where _ is the place holder for the parameter.

2.1 Generic Parts

Context part (Context) First of all, we introduce data types[2] for dealing with UML 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*.

sorts *Ident*, *Name*
preds $isSubType : Name \times Name$
$hasType : Ident \times Name$
axioms $\forall id : Ident; c, c' : Name \bullet isSubType(c, c') \wedge hasType(id, c) \Rightarrow hasType(id, c')$
type $Value ::=$ **sort** $Bool \mid$ **sort** $Date \mid$ **sort** $Integer \mid$ **sort** $Ident \ldots$
$Type ::= bool \mid date \mid integer \mid$ **sort** $Name \ldots$

Other static information have to be recorded as well; for instance we need predicates recording which names[3] correspond to classes (**preds** $isClass : Name$) or associations; attributes/operations (**preds** $isOp : Name \times Name$) of a class; types of the operation input and output (**ops** $argType, resType : Name \rightarrow ? List[Type]$); types of the attributes, or of left and right ends of binary associations, etc. Such operations are defined as partial; thus they accept any name as input, but are undefined on those that are not operations/associations. For instance, *argType* will be undefined, if applied to those names not corresponding to an operation. The treatment of other kind of classifiers, e.g., signals, that is analogous, is omitted.

State part (State) Extending the CONTEXT specification, we introduce the constructs to deal with the global states (**sorts** *State*) of the system and the local states of the objects (**sorts** *Object*).

The global states provide information about living instances of the classes, with tools to check if an object exists (**preds** $knownIn : Ident \times State$), to get the local state of an object (**ops** $localState : Ident \times State \rightarrow ? Object$), etc.

The local states give information about the identity of the objects and the current values of their attributes. We have operations to get the object identities (**ops** $getId : Object \rightarrow Ident$) and to read (**ops** $rdAttr : Object \times Name \rightarrow ? Value$) and write (**ops** $wrAttr : Object \times Name \times Value \rightarrow ? Object$) attributes, as well, with the usual properties about typing (e.g., we cannot assign to an attribute

[2] The datatype construct in CASL, like **types** $::= \ldots$ **sorts** $s' \ldots c(\ldots) \ldots$, is a compact notation to declare a sort s stating at the same time that it includes all the elements of s' and that its elements may be as well built by a constructor c.

[3] In the following, we will assume that the names used to represent attributes, operations and classes are all distinct.

a value of an incompatible type; also updating an attribute is not affecting the others nor the object identity), axiomatically stated.

Each operation of a class represents a nondeterministic (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 is expected, else we will use the empty list) and the new global state, possibly modified by some side effect of the operation call. We formalize the operations by the predicate named *call* relating the name of each operation and the inputs to the possible outputs.

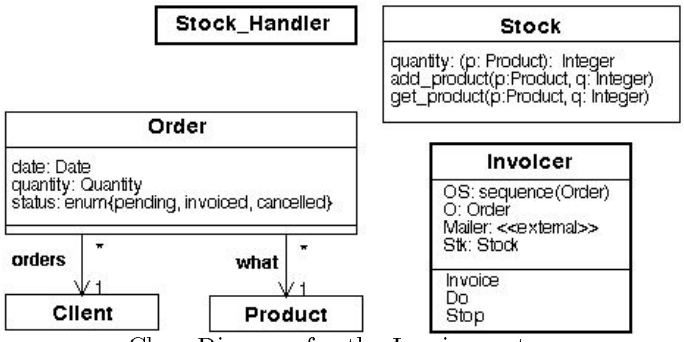
preds $call : Ident \times Name \times State \times List[Value] \times List[Value] \times State$

This is one major difference w.r.t. other translations of the UML into logic/algebraic languages, where operations most naturally become functions, that is due to the need of integrating the semantics of class diagrams given in isolation with the sematics of the overall UML model, including concurrent features.

The local states of the associations are represented by a sort; since the case of binary associations is by far the most common in the UML, we are also providing a specialization (**sorts** $BinAssocState < Association$) [4] of the association type for such case, having, for each association end, an operation (**ops** $LAssoc, RAssoc : Ident \times BinAssocState \to Set[Ident]$), yielding the objects that are in relation with a given object that will be used to represent the UML *navigation*.

2.2 Class specific parts

To illustrate more concretely how the specification corresponding to a class diagram looks like, we will use as running example a fragment of an invoice system.



Class Diagram for the Invoice system

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 two active classes, managing such data and representing two kinds of "software" clerks: the *stock handler*, who put the newly arrived products in the stock and removes the correct amount of products to settle an

[4] Subtyping in CASL is denoted by $<$ and a declaration $s < s'$ implicitly introduces a predicate $\_ \in s$, checking if a term of sort s' is interpreted on a value of type s.

order, and the *invoicer*, who processes orders and sends invoices. Finally, there are associations relating each order respectively to the ordering client and the ordered product. Notice that 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.

Class Specific Context Part (C-Context) Each class contributes to the context with the names introduced by the class. So, for instance, the specification for the class `Client` introduces only its name as a constant of sort *Name*, while that for the class `Order` (`Stock`) introduces the names of the attributes (operations) and their typing as well. As the type of some operation of the class `Stock` involves another class, `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 put together this specification with that corresponding to the static context of `Product`, we will have the missing information.

spec STOCK-CONTEXT = CONTEXT **then**
ops $Stock, Prod, quantity, addProd, getProd :\rightarrow Name$
axioms
$isClass(Stock)$
$isOp(quantity, Stock) \wedge argType(quantity) = Prod \wedge resType(quantity) = integer...$
We deal with associations in an analogous way with respect to classes.

Class semantics part (C) Each specification corresponding to a class introduces (at least) the sort of the local states of that class objects. Moreover, if we have OCL constraints, then they are translated into axioms.

spec CLIENT = STATE **and** CLIENT-CONTEXT **then**
sorts $Client < Object$
axioms $\forall o : Object; \bullet hasType(getId(o), Client) \Leftrightarrow o \in Client$
To translate active classes we use CASL-LTL, an extension of CASL, where sorts may be declared *dynamic*. As we will see in the next section, in this way we have the means to describe the behaviour of their elements. Apart from declaring their object sort as dynamic, active classes are translated like passive ones.

spec STOCKHANDLER0 = STATE **and** STOCKHANDLER-CONTEXT **then**
dsort $StockHandler$ **label** $StockHandler - Label$ **info** $StockHandler - Info$
sorts $StockHandler < Object$
axioms $\forall o : Object; \bullet hasType(getId(o), StockHandler) \Leftrightarrow o \in StockHandler$
Also in this part associations are dealt with in an analogous way w.r.t. classes.

The overall specification giving the semantics of the class diagram, called CLASSDIAGRAM, is the union (CASL-LTL keyword **and**) of the specifications representing the individual classes and relations.

3 UML State Machines

In [9] we present our complete formalization of UML state machines associated with active classes using CASL-LTL; a short version has been presented in [10]; here we briefly report the main ideas.

3.1 How to Represent Dynamic Behaviour

Since the handling of the time in the 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. Thus, taking advantage of a well established theory and technology (see, e.g., [5, 7]), we model an active object A with a *transition tree*. The nodes in the tree represent the intermediate (interesting) situations of the life of A, and the arcs of the tree the possibilities, or better the *capabilities*, of A of moving from one situation to another. Moreover, each arc is labelled by all the relevant information about the move. In standard transition systems, the label is unstructured; here, for methodological reasons, we prefer to use pairs as decorations, where the first component, called *label*, represents the interaction with the external environment and a second one, called *info*, is some additional information on the move, not concerning interactions.

To describe transition trees, we use *generalized labelled transition systems* (shortly *glts*). A *glts* $(STATE, LABEL, INFO, \rightarrow)$ consists of the sets $STATE$, $LABEL$, and $INFO$, and of the *transition relation* $\rightarrow \subseteq INFO \times STATE \times LABEL \times STATE$. Classical labelled transition systems are the glts where the set $INFO$ has just one element.

Each $(i, s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $i : s \xrightarrow{l} s'$.

The transition tree for an active object A described by a glts is a tree whose nodes are labelled by states (the root by the initial state of A) and whose arcs are decorated by labels and informations. Moreover, there is an arc decorated by l and i between two nodes decorated respectively by s and s' iff $i : s \xrightarrow{l} s'$. Finally, in a transition tree the order of the branches is not considered, and two identically decorated subtrees with the same root are considered as a unique one.

A glts may be formally specified by using the algebraic specification language CASL-LTL (see [8]), an extension of CASL (see [6]) designed for the specification of processes based on glts's, where we have a construct for the declaration of dynamic sorts:

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

introducing the sorts *State*, *Label*, and *Info* for the states, the labels and the information of the glts, and implicitly also the transition predicate:

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

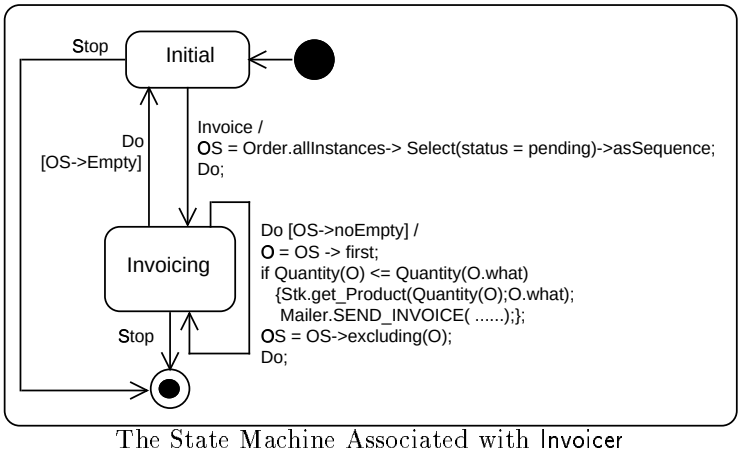
Each model M (an algebra or first-order structure) of a specification including such construct corresponds to the glts $(Info^M, State^M, Label^M, \rightarrow^M)$[5].

In most cases we will be interested in a *minimal* glts, where the only transitions are those explicitly required by the axioms. This is, for instance, the case when translating active classes, because their activity is required to be fully described by the corresponding state machine. To achieve this result we restrict the specification models to the (minimal) *initial* ones[6] (CASL keyword **free**).

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

[6] If any, otherwise the model set becomes empty.

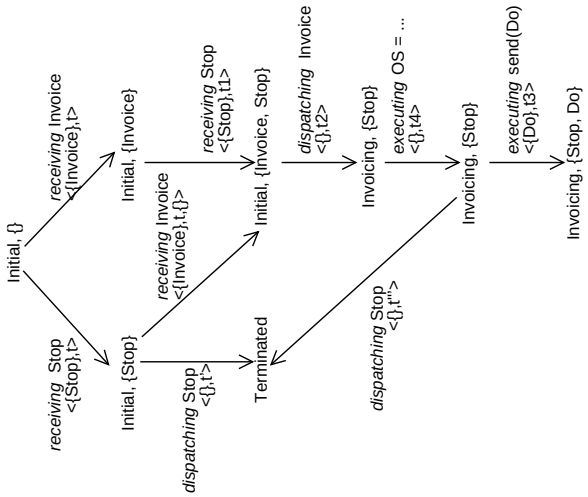
Active classes as glts Assume to have a given active class ACL with a given associated state machine SM. For instance, let us consider the following state machine, associated with the active class Invoicer of our running example.



The State Machine Associated with Invoicer

The instances of ACL, called *active objects* in the UML terminology, are just processes and we model them by using a glts. We algebraically specify such glts, named in the following *GLTS*, with the specification ACL-DYNAMIC.

It is important to notice that the states and the transitions of the state machine SM are different in nature from, and are not in a bijective correspondence to, those of the corresponding *GLTS*. Indeed, one SM-transition corresponds to a sequence of *GLTS*-transitions and the *GLTS*-states are more detailed than those of the SM. To clarify the relationship, see a small fragment of the transition tree associated with the state machine for the Invoicer class (to simplify the picture we only report the configuration and the event queue of each *GLTS*-state).



A fragment of a transition tree

Each *GLTS*-transition corresponds to performing a part of a state machine transition. Then the atomicity of the transitions of SM (run-to-completion assumption) required by [14] 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.

We translate SM into the following specification ACL-DYNAMIC:

spec ACL-DYNAMIC =
LABEL **and** ACL-STATE **and** INFO **then**
free {
dsort *Acl* **label** *Label* **info** *Info*
axioms
. . . . . .
} end

Following the UML philosophy, we assume that all "static" information about the class ACL (and the others), like for instance the attributes and operations, are found in the class diagram and hence in the specification CONTEXT, defined in Sect. 2.

As in the case of class diagrams, a large part of the specification of the *GLTS* representing a state machine is generic, that is, it is common to all state machines. Actually, the only point where the features of an individual state machine play a role is in the definition of the function associating each set of states of the state machine with the set of the transitions starting from it, discussed at the end of the section.

To illustrate the modularity of the approach, we report the axiom defining the transitions corresponding to dispatch an event; for the others see [9].

The intuitive meaning of the axiom is that, if

- an active object is not fully frozen executing run-to-completion steps,
- an event *ev* may be dispatched,
- *rds* is a set of correct interactions for the class ACL corresponding to read the attributes of other objects (checked by *Ok_Read_*ACL),

then *GLTS* has a transition,

- with the label made by *rds* and the received time *t*,
- where *ev* has been dispatched and the history has been extended with the current states.

$$not\_frozen(conf) \wedge dispatchable(ev, e\_queue) \wedge Ok\_Read\_\mathsf{ACL}(rds) \wedge$$
$$\mathsf{Dispatch}(ev, conf, e\_queue, rds, attrs, id) = conf', e\_queue' \Rightarrow$$
$$id : \langle conf, attrs, e\_queue, history, chinf\rangle \xrightarrow{\langle rds, t\rangle} id : \langle attrs, conf', e\_queue', history', chinf\rangle$$

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

In such axiom we use the auxiliary operation Dispatch:
$\mathsf{Dispatch}(ev, conf, e\_queue, rds, attrs, id) = conf', e\_queue'$ holds whenever the object *id* with configuration *conf* and event queue *e_queue* using *rds* and *attrs* may dispatch the event *ev* changing its configuration to *conf'* and its event queue to *e_queue'*.

Notice that the axiom is independent from the particular state machine to be translated. However, the (omitted) specification of the function Dispatch is based on the function **Trans** associating with each set of states (of the state machine) S the set of the transitions having S as source, and its specification is different for each state machine. For instance, **Trans** for the state machine for the Invoicer class associates two transitions to $\{Initial\}$, three to $\{Invoicing\}$ and none to the final state.

4 Semantics of UML Models

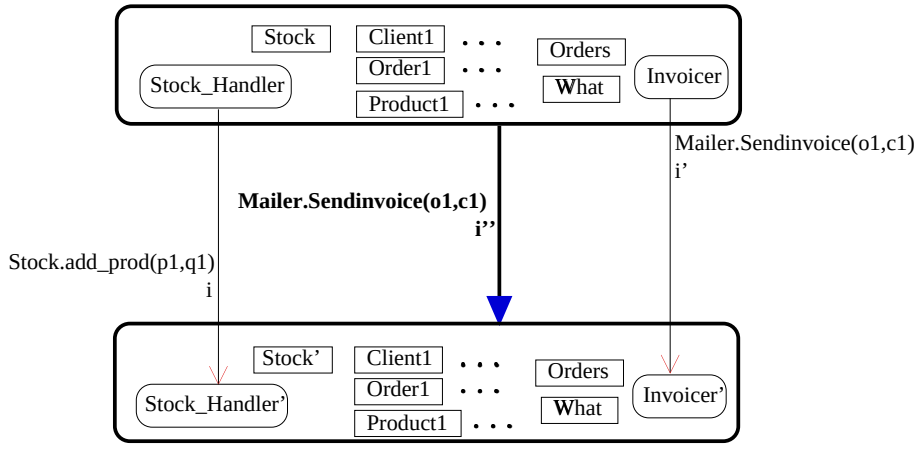
The overall semantics of a UML model is given by a set of what we have called (see Sect. 1) UML formal systems. First we briefly illustrate them also with the help of our example and then we outline their formal specification.

4.1 UML Formal Systems

A UML model describes the *structure* of a system, that is which are the components and which are their capabilities, and the *behaviour* of the system itself, 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).

Therefore, its semantics is a set of acceptable structured processes that we represent again by means of *GLTS*, and that we call UML *formal systems*. But, since reducing the concurrency among the components of a system to interleaving appears to be reductive, we want to use *structured GLTS*, where the transitions correspond to really concurrent executions of sets of transitions of the system subcomponents. The sorts *State*, *Info* and *Label* of a *GLTS* can be endowed with operations (and predicates) and be defined, or built as datatypes starting from other (dynamic) sorts. Thus, we can easily impose a state of the structured *GLTS* to be a set of states of its subcomponents and its transitions and labels to be defined in terms of those of its active parts.

For instance, a possible transition of a UML formal system corresponding to our running example is the following:



Both in the source and in the target state we have two active components (represented by rounded shapes), the Stock_Handler and the Invoicer and a number of passive components (represented by rectangles): some instances of Client and Product, the unique instance of Stock and the state of the associations orders and what. In this picture we consider the parallel execution of two activities, graphically represented by the thin lines connecting the involved active components of the source and target state:

- the Stock_Handler adds to the Stock some quantity *q1* of the product *p1*; the effect of this action is to change the state both of the Stock_Handler and of the Stock;
- the Invoicer sends to the client *c1* an invoice for the order *o1*, by calling a method of some external *mailer*; thus, the effect of this action is to change the state of the Invoicer and to communicate with the external world through the label of the transition.

The parallel composition of these actions is described by a transition, graphically represented by the thick arrow connecting source and target structured states. Notice that, since the labels of the structured system carry only information about the interactions with the external world, the label of this transition is taking into account only the call to the mailer. But the resulting state of the system is modified because the internal states of all the objects involved in the move are (possibly) modified.

4.2 The Specification of a UML Formal System

The semantics of a whole UML model is given by the combination of the specifications providing the semantics of the single views determined by the various diagrams that compose the model. Therefore, we start by extending the specifications produced by the class diagram(s) and the state machines in order to describe the glts modeling the system.

The states of the overall system are sets of states of the components. Indeed, we do not need multisets, since objects are distinguished by their identities.
type *State* ::= Λ | **sort** *Object* | **sort** *Association* | _||_(*State*; *State*)
op _||_ : *State* × *State* → *State* **assoc**, **comm**, **idem**, **unit**: Λ
The specification of the labels of the system (System-Label), based on the interactions with the external environment, is omitted. The informations of the generalized transitions will be a set of stimuli. Indeed, to be able to evaluate the satisfaction of a sequence diagram by a UML formal system M we need to record for each transition of M which are the stimuli exchanged during such transition. The specification Stimulus, describing the stimuli, that is the the trivial translation into Casl of the UML stimuli (see [14] p. 2-86) is omitted as well.

In a context extending the specifications Class_Diagram, Acl-Dynamic$_1$, ..., Acl-Dynamic$_p$, System-Label and FiniteSet[Stimulus], we can describe the dynamics of the elements of sort *State*
dsort *State* **label** *System-Label* **info** *FinSet*[*Stimulus*]

It is worth noting that to state the behavioural axioms we need some temporal logic combinators available in CASL-LTL that we have no space to illustrate here. The expressive power of such temporal logic will be also crucial for the translation of sequence diagrams, though they are not discussed here.

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 such axioms, that, though quite trivial to be expressed, would produce a long boring list.

Much more interesting are the axioms describing the transitions of the overall system. For instance, we can state that we can add to the source and the target of a transition any number of components that do not participate into the transition: **axiom** $inf : s \xrightarrow{l} s' \Rightarrow inf : s||s_0 \xrightarrow{l} s'||s_0$.

Let us now show how to compose the transitions of the individual active components, using several auxiliary functions, whose axiomatization we omit for lack of room. Assume that we have, as part of a state s, a group of active objects a_i which may perform some transitions $inf_i : a_i \xrightarrow{l_i} a_i'$, accordingly to the specification of their active classes. Then, we can compose such transitions into a transition of s only if the interactions appearing on the l_i's, and on the label l of the resulting transition 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*) and the interactions corresponding to read the attributes of passive objects are in agreement with their actual values (this is checked by the predicate *Read_Attributes*). If these conditions are satisfied, then the system can move into a new state where the active objects are evolved into the a_i''s, the passive components of s are updated and some new components may be possibly added.

$$
\begin{array}{l}
\wedge_{i=1}^{n}\ inf_i : a_i \xrightarrow{l_i} a_i' \wedge \\
Ok\_Labels(l_1 \ldots l_n l) \wedge \\
Read\_Attributes(l_1 \ldots l_n l, pss) \wedge \\
Updated(l_1 \ldots l_n l, pss) = pss' \wedge \\
Created(l_1 \ldots l_n l) = \overline{oss} \Rightarrow \\
Stimuli\_Of(l_1 \ldots l_n l, inf_1, \ldots, inf_n) : a_1||\ldots||a_n||pss \xrightarrow{l} a_1'||\ldots||a_n'||pss'||\overline{oss}
\end{array}
$$

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 (i.e., it has no models).

5 Conclusion

Our work stems from the belief, supported by concrete experience, that analysing the UML with different techniques helps the understanding, especially when an

official formal definition of its semantics is lacking and many proposal for improvements and extensions are under way. Indeed, this kind of analysis is explicitly encouraged by some members of the OMG involved in the definition of UML (Bran Selic during the discussion at the <<UML>> 2000 Workshop "Dynamic Behaviour in UML Models: Semantic Questions", York, October 2000). Moreover, we have already shown in [10] how this kind of analysis can lead to spot problematic points and offer various alternatives for a precise definition.

With respect to the proposal of a standard definition by an OO metamodelling approach, like the one advocated by [3], our work is complementary, in two senses. First of all, it provides a basic technique for modelling dynamic behaviour, which is based on the twenty years long and successful experience of CCS and the like (labelled transition systems), already used for the full formal definition of Ada ([1]). That technique is adequate for modelling all dynamic features of UML; for example by this technique we are currently working on sequence diagrams; 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, being rather similar to the sequence diagrams;
- the activity diagrams, as they are a specialization of the statechart diagrams.

Thus it would be interesting to explore the possibility of adopting this technique as a basis in the so called dynamic-core sketched in [3], which is admittedly rather preliminary. For a proposal in that direction see [2]. A point that we want to stress again is that our approach is supporting the multiview philosophy, in the sense that each part of a UML model has a proper formalization that is integrated with those of the other parts. Moreover, we are considering full UML and this is quite important, because it is often the case that the semantics given for a restricted part of UML in isolation is useless when trying to uplift it to the full UML.

For instance, even from this partial work 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 is limited. 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 whether it is associated with an active or with 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 (see Sect. 4.2).

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 and paving the way for better new versions and a standard complete definition of its semantics.

References

1. E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca. The Draft Formal Definition of Ada. Deliverable, CEC MAP project: The Draft Formal Definition of ANSI/STD 1815A Ada, 1986.
2. E. Astesiano and G. Reggio. A Proposal of a Dynamic Core for UML Metamodelling with MML. Technical Report DISI-TR-01-1, DISI – Università di Genova, Italy, 2001. `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio01a.pdf`.
3. 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.
4. R. France and B. Rumpe, editors. *<<UML>>'99 - The Unified Modelling Language*. Number 1723 in Lecture Notes in Conputer Science. Springer Verlag, 1999.
5. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
6. The CoFI Task Group on Language Design. CASL Summary. Version 1.0. Technical report, 1998. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
7. G. Plotkin. An Operational Semantics for CSP. In D. Bjorner, editor, *Proc. IFIP TC 2-Working conference: Formal description of programming concepts*. North-Holland, Amsterdam, 1983.
8. 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`.
9. 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`.
10. 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 - Fundamental Approaches to Software Engineering*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.
11. 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.
12. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.
13. S.Kent, A.Evans, and B. Rumpe. UML Semantics FAQ . In A. Moreira and S. Demeyer, editors, *ECOOP'99 Workshop Reader*. Springer Verlag, Berlin, 1999.
14. UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.