An Extension of UML for Modelling the nonPurely-Reactive Behaviour of Active Objects

G. Reggio and E.Astesiano

DISI - Università di Genova, Italy
reggio,astes@disi.unige.it
http://www.disi.unige.it/person/ReggioG/

Abstract. Modelling nonpurely-reactive systems, such as agents and autonomous processes, does not find a direct support in the UML notation as it stands, and it is questionable whether it is possible and sensible to provide it in the form of a lightweight extension via stereotypes.
Thus we propose to extend the UML notation with a new category of diagrams, "behaviour diagrams", which are, in our opinion, complementary to statecharts for nonpurely-reactive processes (active objects). The proposed diagrams, to be used at the design level, also enforce localization/encapsulation at the visual level. Together with motivating and presenting the notation, we also discuss the various possibilities for presenting its semantics in a palatable way, depending on the reader.

1 Introduction

The Unified Modeling Language (UML), see [21]and [18], is an industry standard visual notation widely used for specifying software systems. UML incorporates in an integrated way several notations for object-oriented modelling, covering different aspects of a system also at different points of its development process.

Concerning the modelling of the dynamic behaviour of a system UML offers many different notations, as sequence, collaboration, activity, and statechart diagrams. Furthermore, we can attach constraints to the various parts of a UML description, and thus implicitly constraining the system behaviour, and define methods for the class operations, thus modelling parts of such behaviour.

Because of the richness and the abundance of the notations provided by UML, someone may object to the introduction of further notations. But UML is intended to be extensible, adaptable, and modifiable. From the preface of the proceedings of the <<UML>>'99 conference [6]:

> *Flexibility is needed if the UML is to be used in a variety of application domains. Tailoring of UML syntax and adaptation of UML semantics to system domains is highly desiderable. Incorporating domain-specific concepts into the language will yield modelling languages that more effectively support system development in these domains. Tailoring may involve determining a subset of the UML that is applicable to the domain, extending or modifying existing language elements, or defining new language elements.*

Furthermore, UML itself provides the mechanisms, summarized in [12], to define its variants. UML Real Time [20] is an example of a variant for modelling embedded real-time systems.

In this paper we consider the case of using UML for modelling systems containing processes whose behaviour is not purely-reactive, and this is a relevant field including many widespread applications. Consider, for example,

- agent systems
- distributed systems made by autonomous processes, as a special case we have processes that cooperate by using tuple spaces (as those supported by the JavaSpaces^{TM} technology, see [10]). The basic paradigm for describing their behaviour is absolutely nonreactive, indeed they go on by adding tuples to the spaces, and by reading or taking tuples matching some patterns from the same spaces, and so they act and test the external environment (the tuples) and do not react to its changes (tuple spaces are fully passive)
- (mobile) processes travelling over a net doing maintenance and monitoring activity. A concrete instance is the following process monitoring the dead links of a web site.

  It downloads the pages of the web site following some order one after another, thus for each page it checks whether it contains some dead links; if the answer is positive, then it replaces such links with some text; moreover if the dead links are more than two, it informs the web master. At each moment it can receive the order to follow for visiting the pages by the webmaster.

We have analysed UML and think that it does not offer direct means to describe the nonpurely-reactive behaviour of active objects (objects of active classes), which is the UML word for processes. Here, we sketchy present the result of our analysis.

Sequence, and similarly collaboration diagrams, allow to present possible sequences of stimuli[1] exchanged among some of the objects of the system.

Activity diagrams are used to describe causal relationships among internally-generated actions in classes instances or in an operation implementation, that is, following [21] p. 3.151, procedural flow of control.

Constraints attached to a class or to the whole model[2] (as invariants), operations and to signal receptions (as pre-post conditions) may indirectly constrain the dynamic behaviour of the systems or of some of the composing active objects.

Methods attached to operations may only describe parts of the behaviour of the system or of some of its components.

Thus, the constructs listed above cannot be used to model the behaviour of active objects, whereas statecharts[3] are the canonical candidates for this task. However, they are strongly biased towards reactivity, and below we try to corroborate this statement.

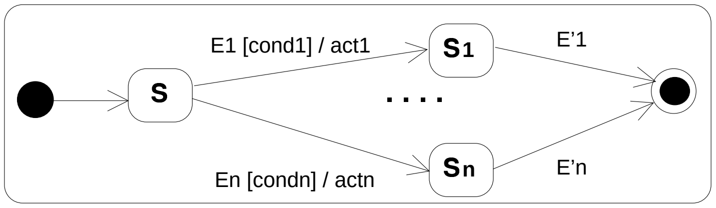
[1] In the UML terminology, a stimulus is a communication between two objects.
[2] In the UML terminology, a model is the description/specification of a system.
[3] Statecharts are the diagrams that visually present the state machines.

UML statecharts have been originated by Harel statecharts [7], though they have many new advanced features and their semantics for what concerns the treatment of the received events is different.

The UML documentation explicitly states that statecharts are for reactivity.

A statechart diagram can be used to describe the behavior of a model element such as an object or an interaction. Specifically, it describes possible sequences of states and actions through which the element can proceed during its lifetime as a result of reacting to discrete events (e.g., signals, operation invocations). [21] p. 3-131
State machine may be used to describe user interface, device controllers, and other reactive subsystems. [18] p. 30
A class .... It may have a state machine that specifies its reactive behaviour - that is, its response to the reception of events. [18] p. 186

The purely reactive nature of statecharts is shown by the following example.



An active object whose behaviour is described by such chart will never move in the case the context is not generating an event corresponding to one of E1, . . . , En, for example an empty environment.

Moreover, there is another aspect of statecharts that is often unnoticed and whose possible negative effect is understimated. The statecharts visually model only when and to what the active objects react, while the content of the reaction is presented textually. Indeed, each transition visually models a possible way of the object to react to some external stimulus, but the activity corresponding to the reaction is not depicted; however, in some cases it may be quite complex and consisting of several atomic interactions with many other objects. Consider, for example, the statechart in Fig. 1, there the reaction to event E consists of calling several operations of several objects, and of creating and destroying other objects, but the diagram is not suggesting it. For a common, i.e., not very skilled, user the result may well be a misunderstanding of the semantic implications of what s/he depicts and sees, since apparently everything should be clear from the visual notation.

Thus, we propose in this paper to extend UML by introducing a new kind of diagrams for modelling in general the behaviour of active objects, named "behaviour diagrams"[4]. The basic ideas behind are the following.

[4] In the UML terminology (see [12]) this would be a heavyweight extension, because such diagrams would be defined by introducing a new syntactic category and not as a stereotype (a variation) of an existing one. However, the question whether these diagrams can be obtained by the existing ones via stereotypes is still unanswered.

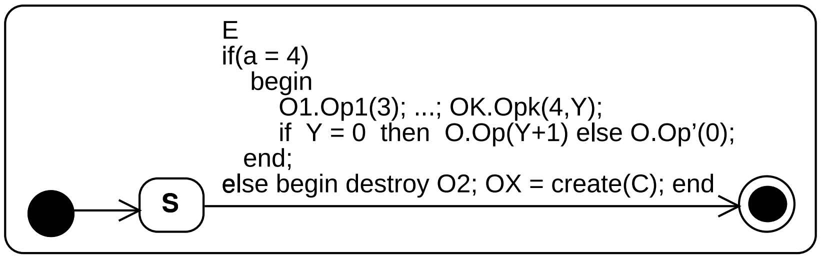

E
if(a = 4)
begin
O1.Op1(3); ...; OK.Opk(4,Y);
if Y = 0 then O.Op(Y+1) else O.Op'(0);
end;
else begin destroy O2; OX = create(C); end
S

Fig. 1. A very complex reaction

- We keep the state-transition paradigm for modelling behaviour as for statecharts.
- However, differently from statecharts,
  - the transitions correspond to perform "atomic interactions" (atomic activities made by the object in cooperation) with the external environment, as to receive/send a call, to receive/send a signal, to create/destroy an object[5].
  - we enforce localization: the attributes of the object modelled by behaviour diagrams cannot be visible outside the object (except than in the diagram) and the operations appearing in the diagram cannot have an associated method behavior.

Thus the fundamental difference with statecharts is double: describe (possible autonomous) atomic interactions instead of just reactions; make visible on the diagram all the interactions with the external environment. For those who are acquainted with CCS [11] and CSP [8], the first difference would roughly correspond to allow on the transitions also the output operations (denoted by !) and not only the input ones (denoted by ?).

The paper is organized as follows.

We present in Sect. 2 in a detailed way the kernel of behaviour diagrams (that is the simplest diagrams) to introduce the new "interactive paradigm"; in Sect. 3 we shortly present some of the advanced features; in Sect. 4 we discuss some sensible ways to present the semantics of the behaviour diagrams; finally in the conclusions we put our work into perspective with other work and current UML.

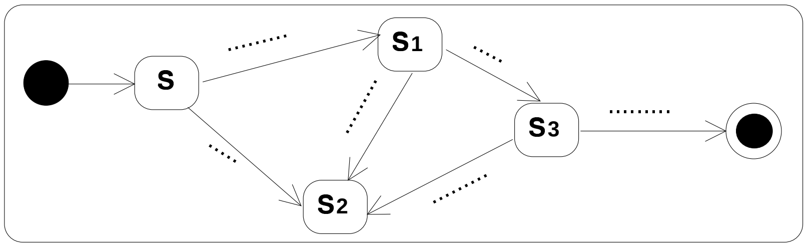
[5] Notice that in the UML terminology *interaction* is used for something different: "*A collaboration of objects interacts to accomplish a purpose (such as performing an Operation) by exchanging Stimuli. These may include both sending Signals and invocations of Operations, as well as more implicit interaction through conditions and time events. A specific pattern of communication exchanges to accomplish a specific purpose is called an interaction.*" [21] p. 3-117

2 Behaviour Diagrams: the Kernel

2.1 The notation

The behaviour notation is based, similarly to the UML statecharts, on the state-transition paradigm; thus a behaviour diagram in its simplest form is just a graph with nodes decorated by a name (the states), with oriented arcs among nodes (the transitions). The fundamental difference with the statecharts is the form of the inscriptions decorating the transitions and their meaning. In a statechart transitions are triggered by events raised by the external environment and in some special cases by the past activities of the object itself; whereas in behaviour diagrams transitions correspond to atomic interactions with the external environment.

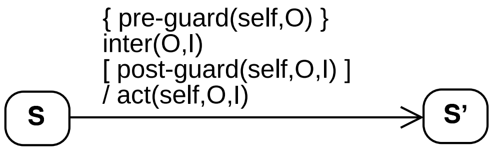
Below is an example of a behaviour diagram belonging to the kernel. Notice that we consider only the case of a behaviour diagram whose context is an active class[6], say AC.



The modelling paradigm is quite simple. During the life of an active object of class AC exactly one state is "active", and only the transitions from an active state are eligible for execution. When the object is created exactly one state will be active, the one marked by ●⟶; clearly it is required to be unique. When the final state, the one marked by ◉, becomes active, the object activity ends.

The modelled behaviour consists of executing eligible transitions, as a run-to-completion-steps, one after another; in the meantime events (only call, signal and destroy) may be added to the event queue as soon as the corresponding call (or signal, or destroy indication) arrives.

Now we present the generic form of (the inscription decorating) the transitions and describe which eligible transitions will be picked up for execution and what does it mean to execute a transition, i.e., the run-to-completion-step for behaviour diagrams.



where

[6] In the UML terminology, the context is the element whose behaviour is modelled by the diagram.

- pre-guard, the *static guard*, is a boolean expression (written using OCL[7]).
- inter, the *atomic interaction*, is defined by the fragment of the metamodel in the UML metamodel[8] in Fig. 2 may be either a simple action requiring an interaction with some other object, or an event that is the result of an interaction by some other object or null that is no interaction with any object, i.e., an activity purely internal to the object. Input atomic interactions may have parameters, denoted in the above picture by I, that are just variables, output atomic interaction may have arguments, denoted in the above picture by O, that are just ground expressions, whereas the null atomic interaction has neither parameters nor arguments.
- post-guard, the *dynamic guard*, is a boolean expression (written using OCL).
- act, the *action*, is an UML action.

The intuitive meaning of a transition of a behaviour diagram is that the object may perform (whenever possible) some interaction with some other object possibly followed by some local activity when some guard conditions are satisfied.

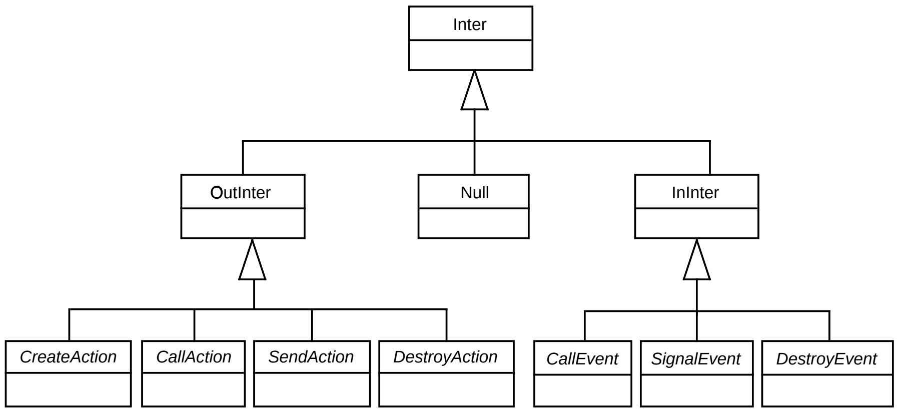
In this case the "Well-Formedness Rules" are quite important and are as follows

- O, pre-guard and post-guard cannot have any side effect, as already required by statecharts
- O, pre-guard, post-guard and action cannot contain references to other objects neither to associations (roughly, it must be possible to evaluate/execute them without accessing anything outside the context object). For example, this means that, differently from statecharts, any call to other objects cannot appear in the action part
- I, the parameters of input atomic interactions, may appear in post-guard and action
- any operation of class AC appearing in the behavior diagram cannot have an associated method, while any one not appearing in such diagram should have an associated method and should be visible only inside the class and the behaviour diagram
- all the attributes of the context active class are visible only inside the class itself and the behaviour diagram, thus no other objects may access or modify them indirectly, they can do that only explicitly by calling operations or sending signals to the object.
- there cannot be two transitions with the same atomic interaction part (this is not restrictive, because of the presence of the null atomic interaction).

Notice that when we speak of objects, we do not consider instances of UML datatypes.

[7] The Object Constraint Language to specify constraints and other expressions attached to UML models, see [21][chapter 7].

[8] The abstract syntax of UML is given by means of an object-oriented description, a class diagrams, called *metamodel*, whose classes corresponds to the abstract syntactic categories.



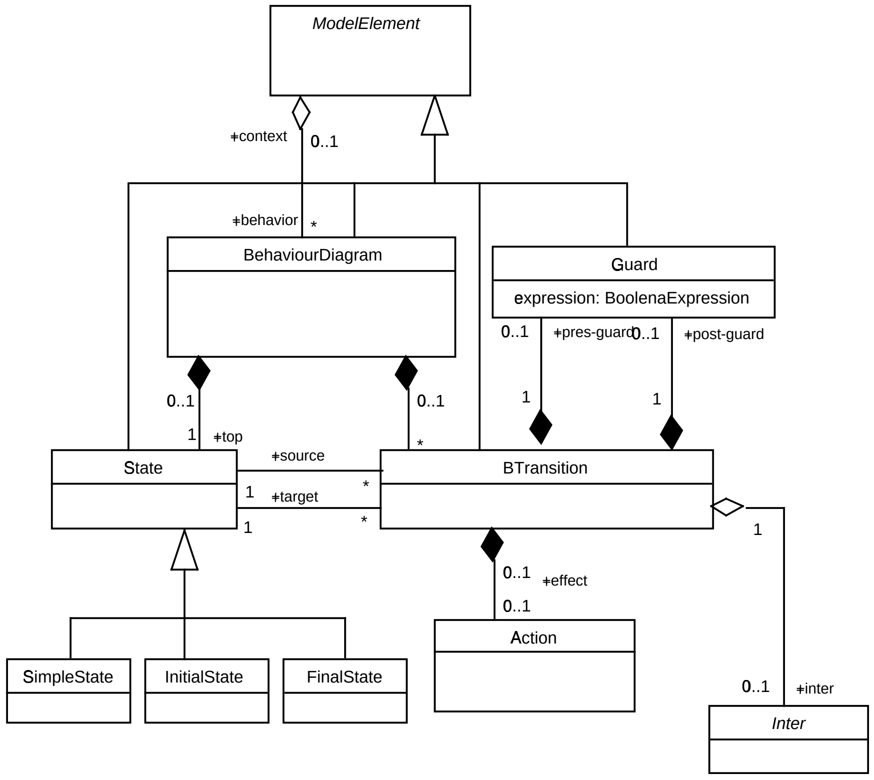


Fig. 2. The metamodel definition of behaviour diagrams (kernel)

The precise meaning of a behaviour diagram transition is given by describing a run-to-completion step (RTC) as for statechart.

We call a transition of a kernel behaviour diagram *eligible* whenever its source state is active; this definition will be more complex for the case of behaviour diagrams not belonging to the kernel, but it can always be given by using just the static structure of the diagram and the information of which states are active.

1. Picks up an eligible transition s.t. its pre-guard holds.
2. Execute inter (that can either be successful or fail). The meaning of executing an atomic interaction is defined by cases.
   call-action execute the call action; in the kernel we consider only asynchronous calls, thus it cannot fail
   call-event if there is matching call in the event queue, then take it instantiating the parameters, otherwise fail
   send-action execute the send action; sending signal is always asynchronous, thus it cannot fail
   signal-event if there is matching signal in the event queue, then take it instantiating the parameters, otherwise fail
   null inter do nothing, clearly it cannot fail
   create-action execute the create action; it cannot fail
   destroy-action execute the destroy action, that is considered as a special kind of call-action; so it results in putting a destroy event in the queue of the argument object; thus it cannot fail
   destroy-event if there is a destroy event in the event queue, then take it, otherwise fail
3. If the execution of inter is not failed and the post-guard holds, then act will be executed, and, denoting respectively by **S** and **S'** the source and the target states of the transition, at the end **S'** will be active and **S'** will be inactive (except if it is **S**), else do nothing.
4. The RTC is finished.

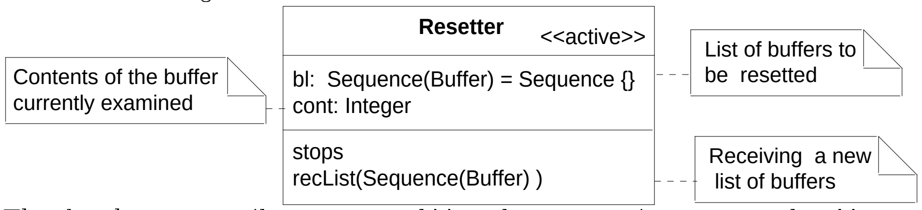
The algorithm for picking up the transitions to be executed is as follows: all transitions leaving a state are totally ordered in some way not further specified, then they are picked up one after another following such order; if all of them fail, the procedure starts again possibly ordering the transitions in some other way. As a consequence we have that the nondeterministic capabilities corresponding to the transitions leaving the same state will be tried in a quite fair way (it cannot happen that the object goes on forever trying failing transitions while another one can be executed); in the next section we will see another mechanism for allowing the user to explicitly decide the order in which to pick up transitions.

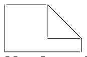
2.2 An Example: the distributed buffer resetter

In this section we show a simple example of the use of behaviour diagrams to model a nonpurely-reactive process, the *distributed buffer resetter*. This example is quite simple but is paradigmatic of autonomous agents doing monitoring and maintenance over distributed systems or Internet.

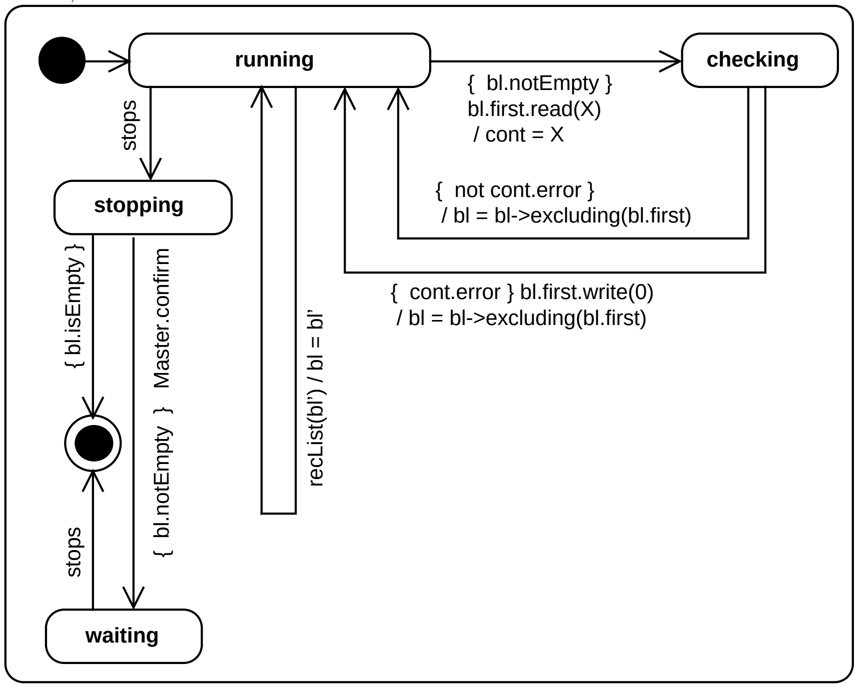
– It accesses some buffers one after another following some given order; thus for each buffer if its contents is "wrong", then it resets it.
– At each moment, it can receive from some manager the list of the buffers to reset, and it can be stopped.

Using UML we model the buffer resetter using an active class with an associated behaviour diagram.


Resetter
<<active>>
bl: Sequence(Buffer) = Sequence {}
cont: Integer
stops
recList(Sequence(Buffer))
Contents of the buffer currently examined
List of buffers to be resetted
Receiving a new list of buffers

The class has two attributes cont and bl, and two operations stops and recList. The text enclosed by  is an UML comment.

We assume that all the OCL types, for example, Sequence and Integer are UML datatypes, i.e., special classes whose instances have no identity (they are pure values) and whose operations never change their states (they are pure functions).



The resetter in the running state has three possible moves:

– when bl is not empty, it may access the first buffer of the list getting its content; if such content is or not an error (checked by the operation error) it resets such buffer by writing 0

– it may receive a new list of buffers to be resetted by accepting a call of its operation recList
– it may receive a request to stop (by a call of its operation stops); in such cases it passes in the state **stopping**. If bl is empty, it terminates by a transition into the final state, whose atomic interaction is the null one, otherwise it asks for a confirmation to its master by calling its operation confirm. Then, if it receives it as a new call of the stops operation, it terminates.

Notice that, as a *Presentation Option* we have that it is possible to omit the null atomic interactions (as in the transition from state **waiting** to **running** and in the one from **stopping** to the final state).

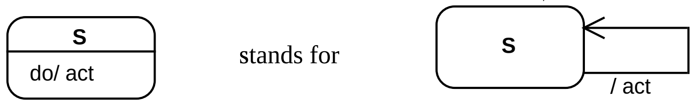
3 Behaviour Diagrams: Advanced Features

In the previous section we have presented the kernel of the behaviour diagram notation, showing its main differences with respect to UML basic statecharts. Designing behaviour diagrams we followed the idea that their differences with statecharts should be confined to the kernel part, while the advanced features should be, as much as possible, the same, but trying to avoid some problematic aspects of statecharts, as those found in the formally based analysis of [16, 15]. Here, for lack of room we cannot consider all such features and so we briefly present some of the most relevant. In Fig. 3 we present a new behaviour diagram for the buffer resetter using some of the advanced features.

Sequential and concurrent hierarchical states and completion transitions as for statecharts. In Fig. 3 we present a new resetter that can also be destroyed in an uncontrolled way, by modifying the corresponding behaviour diagram by adding a hierarchical state.

Internal transition, entry and exit actions as for statecharts.

State activity (do activity) We define a nonproblematic form of do activity for behaviour diagrams as a derived construct, in the following way (that is possible due to the presence of the null atomic interaction).



Compound transitions UML is quite complex and truly unclear for what concerns the compound transitions, state vertex and pseudo states (see the metamodel at [21] p. 2.130); indeed many different features are introduced by vary combining such ingredients. We think that only three of these features are relevant for behaviour diagrams and that may be introduced quite simply as follows.

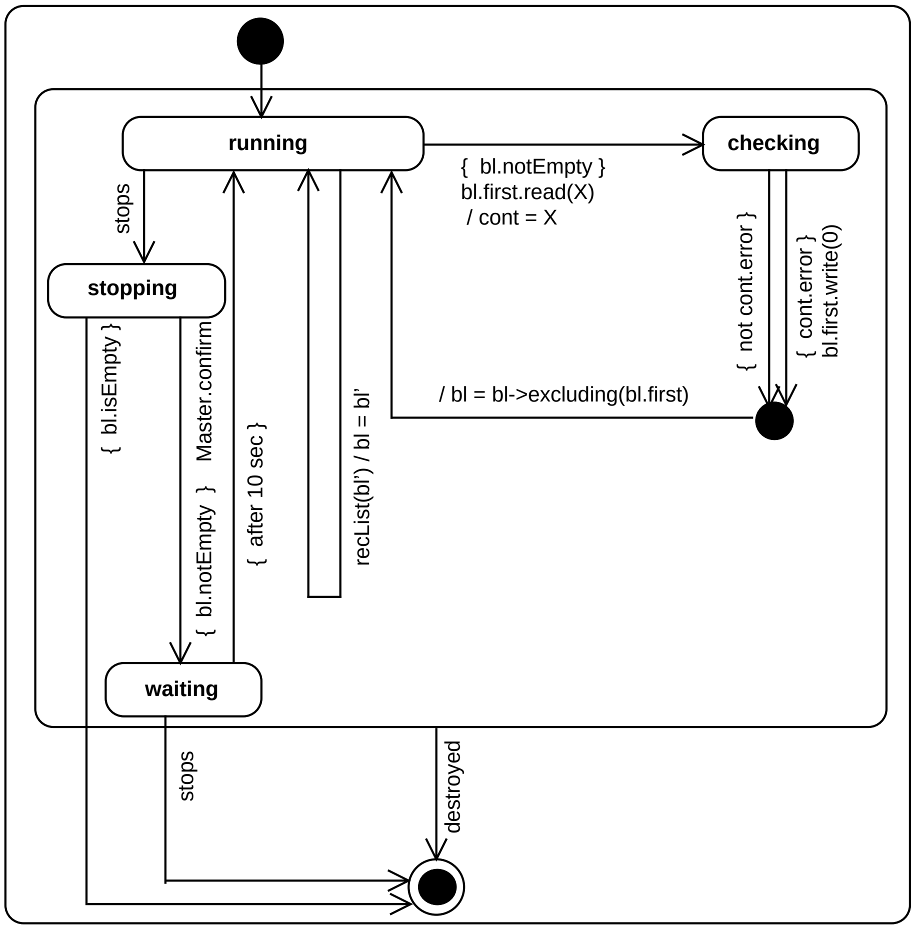


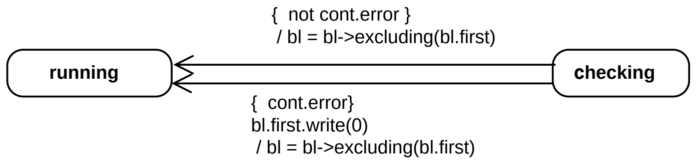
Fig. 3. A buffer resetter (advanced notation)

Factorizing transitions into segments It is useful to visually present a transition by joining many *transition segments* (that are not transitions) by a special symbol, the *junction* visually presented by ●.

Technically, a *transition segment* is an arc either between two junctions (we prefer not to call them states) or a junction and a state or a state and a junction decorated with a partial transition inscription (e.g., just a pre or a post-guard, an atomic interaction, an action, or a pre-guard and an atomic interaction, ...). The meaning of junctions and segments is simply given by some replacement rules: a junction may be eliminated by connecting any incoming arcs with any outgoing arcs and decorating the resulting arc with the combination of the two inscriptions (clearly, not all combination of segments are correct, e.g., a guard cannot follow an action).

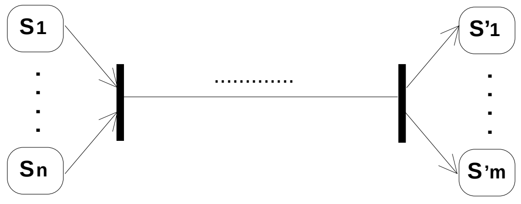
The factorization of transitions improves the readability of the diagrams, by splitting in pieces complex transitions and by avoiding to depict many times the same part of inscription.

In Fig. 3 we have factorized the following two transitions

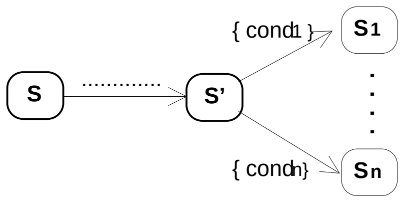


to avoid to write twice the segment containing the assignment action. Visually this helps to grasp that in any case the first buffer is eliminated from the sequence.

Transitions with multiple source/target states Transitions with multiple source/target states can be easily added to behaviour diagrams by changing in the metamodel the arity of the source and target associations of transitions. Visually they will be represented in the following way.



Dynamic choice A UML dynamic choice point allows to split a transitions in several outgoing fragments, each with its own guard; such guards are evaluated dynamically, that is after the preceding part of the run to completion step has been executed (recall the normal guards are instead evaluated before deciding to take the transition). The presence of the postguards and of the null interaction allows to realize this construct in behaviour diagrams as derived.



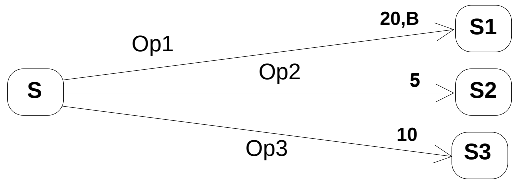
Time events Time events are the way of statecharts to react to some conditions related to the time, as the reaching of some time or the elapsing of some delay. It is not sensible to see these time-related conditions as special atomic interactions, whereas in our opinion is convenient to consider them as special conditions that may appear in pre-guards. Technically, we introduce them as new OCL constructs: an operation returning the actual time and another one returning the time elapsed since when the currently state has become active. See in Fig. 3 an example of use of these new conditions in the transition from **waiting** to **running**.

Synchronous calls In UML operation calls may be either synchronous or asynchronous; in the kernel part of behaviour diagrams we have considered only asynchronous calls. However, full behaviour diagrams should allow also atomic interactions corresponding to synchronous calls. The problem is an old one and consists of how handling nondeterministic choice among many possibilities of interactions, where some of them are synchronous, and thus blocking, and others asynchronous, and thus nonblocking. If a blocking interaction has been picked up, should the object remain blocked, perhaps also forever, when another transition may be executed? The behaviour diagrams of JTN (see [3]) solved this problem by restricting the allowed combinations of synchronous and asynchronous interactions and by ordering them, precisely defining when the object should remain blocked and when not. The behaviour diagram variant of [17] solved it by abstractly handling asynchronous interactions by requiring the simultaneous execution of the two matching interactions by the two partners of the communication; however this cannot be proposed for UML also because it is not consistent with any form of distribution of the objects.

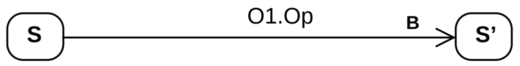
Our idea in handling this point is to mark the transitions as blocking or not; a blocking transition will never fail and so once picked up the object will wait, also forever, till it will be executed.

Furthermore, we think that it may be useful in some cases to attach to the transitions a priority value (just a natural number) and thus the RTC step will consider the transitions following such order till to find one whose execution is not failing.

Consider the following example:



if there is a call of Op2 in the queue, the object passes in state **S2**, otherwise if there is a call of Op3, the object passes in state **S3**, otherwise it waits until there will be a call of Op1, and in such case it will pass in **S1**. If we drop the blocking decoration from the transition in **S1**, then it will go on forever by looking for a call in the event queue following the order Op2, Op3, Op1.



instead shows a blocking call to some object, that is a synchronous call.

Finally let us note that two features of statecharts, namely *Deferrable* and *Change Events*, lose their meanings for behaviour diagrams. The first because there is no more the event dispatching. Indeed, the object activity is driven by the object thread, which picks up events from the queue only when it is able to handle them.

The second, which is a way to react to some boolean expression becoming true, because the value of the object attributes cannot change implicitly, since no other object may indirectly access them, and the expression controlled by the change event should not contain references to other objects, as already strongly suggested by the UML specification [21].

4 Semantics of Behaviour Diagrams

Up to now, in this paper, we have presented the pragmatic of the behaviour diagrams trying to motivate them and to show what they are good for; now we consider their "semantics", providing an outline. The pair "UML - semantics" has been quite debated, e.g., has been the topic of a long series of workshops at OOPSLA, ECOOP and <<UML>> conferences and has even raised philosophic discussions in many occasions. Behaviour diagrams have a semantics, which is precise/rigorous/sound/well-founded (we have no semantic variation points).

However, as often happens in practice, the relevant point is how to present such semantics, in a way that is well accepted by a wide UML audience. We think that there are three different "Presentation options", each one particularly apt for a different kind of person, precisely for the

- UML practitioner (who reads at most the reference book and the specification)
- formalist (who believes in the role of precise formal definition of any used notation)
- metamodel lover (who thinks that the only viable way to propose a precise semantics of UML is using UML itself)

4.1 UML practitioner semantics

This is exactly the kind of "informal" semantics given in the UML specification document [21]. To complete??? it in the spirit and in the form of [21], we need to give

– an extension of the metamodel with the new metaclass *BehaviourDiagram* (see Fig. 2) with the appropriate well-formedness conditions precisely presented using OCL
– a new chapter of UML specification ([21]) chapter on semantics similar to that of the statecharts, but about behaviour diagrams and based on our RTC step. Perhaps the only difference may be a new organization distinguishing the basic constructs defined directly and the derived ones, defined by giving the corresponding combinations of basic constructs (see e.g., the state activity defined in Sect. 3)
– no other modifications are needed, except the extension of OCL with the new time related operations, because the newly introduced diagrams concern only the behaviour of active objects.

An interesting question arising at this point is "do we really need a new metaclass for behaviour diagrams ?" or can they be obtained as a special subcase or as a stereotype of some already existing model element ? This is quite common in UML; for example, activity diagrams are a specialization of statecharts, and sequence diagrams are a variant of collaborations. At the moment we have not an answer; this point needs to be further investigated.

4.2 Formalist semantics

It is possible to give a formal semantics to behaviour diagrams by using labelled transition systems presented by means of an appropriate specification language, for example CASL-LTL (see [14]) similarly to the lightweight semantics for statecharts of [16].

The task in this case is simpler and easier because the definition of behaviour diagrams is more precise than that of the statecharts, and because many problematic points of statecharts have been avoided.

Here we cannot present such the semantics, but it can be found in [13].

We would like to point out that the existence of this formal semantics of behaviour diagrams will guarantee the quality of its informal presentation of Sect. 4.1; indeed by defining it we have already made a throughly analysis of such diagrams, preventing the presence of ambiguities, problematic points and so on, or just of features whose semantics may be precisely formalized but that it is too complex to be effectively used.

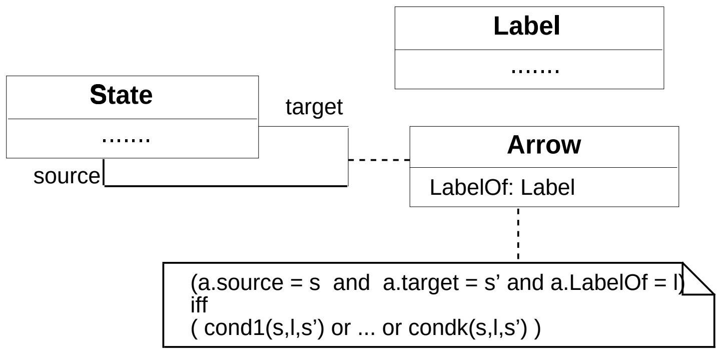
4.3 Metamodel lover semantics

It has been argued are persons that think that the only viable way to propose a precise semantics of UML that can be effectively used by real people is to present it by using UML itself (see e.g., the pUML group[9]; this viewpoint has been strongly supported at the <<UML 2000>> conference in the invited talk by Steve Cook [2].

[9] `http://www.cs.york.ac.uk/puml/index.html`

Naively this could be interpreted in a purely "translational way", that is UML is translated into UML1 (a subset of UML), that in turn is translated into UML2 and so till UML* (consisting more or less of passive class definitions plus OCL) to whom a direct semantics will be given. Now, that approach is instead more clearly understood in a different way; we give a semantics to UML by associating to its parts some meanings in some domains, and we describe both domains and associations with UML itself. Recall that the abstract syntax of UML is already presented by using UML itself, and this is the so-called metamodel.

The key point of the formal semantics of Sect. 4.2 is the use of a labelled transition system to model the active object behaviour, whereas the use of CASL-LTL is not mandatory; indeed such semantics may be presented also using, for example, a plain mathematical notation for sets, functions and and predicates. Thus there is no problem in using UML itself to present such labelled transition system. States and labels could be presented as classes, and the transition predicate by an association class[10], and its definition by conditional rules can be given by an OCL constraint, as we can see in the following picture.


Label
....
State
....
target
source
Arrow
LabelOf: Label
(a.source = s and a.target = s' and a.LabelOf = l)
iff
(cond1(s,l,s') or ... or condk(s,l,s'))

The proposed definition is technically ok, but perhaps not so convenient from the notational point of view.

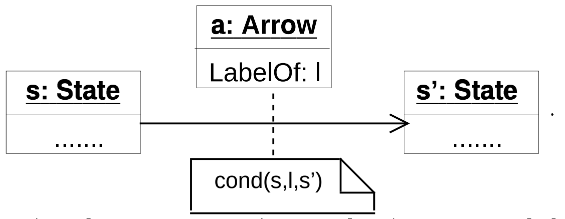
We can propose some appropriate "Presentation Option" in the pure UML style.

For example, the OCL constraint may be equivalently presented in the following ways:

$$\boxed{\textbf{if } \mathit{cond1}(s,l,s') \textbf{ then } s \xrightarrow{l} s' \ \ldots\ \textbf{if } \mathit{condk}(s,l,s') \textbf{ then } s \xrightarrow{l} s'}$$

or more visually by a set of object diagrams of the form

[10] *An association class is an association that is also a class. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not any of the classifiers.* [21]



Using the UML extension mechanism we can define a set of coordinate stereotypes as a toolkit for specifying labelled transition systems, with associated nice notations (one of package for the lts, one of association class for the transition predicate and so on).

Now let us consider the problem of the "circularity" of such definition. UML is defined in terms of itself, and also the semantics of the passive class diagrams is quite problematic (e.g., aggregation [5]). But when proposing the <<LTS>> stereotype we can also impose precise static restrictions, prohibiting the use of some of the problematic features, and moreover the definition of a UML stereotype may include also the refinement and the clarification of the semantics [12]. Furthermore we can also give directly a formal semantics to thus subset.

A similar approach to metamodelling for UML has been proposed in [4], where they use a variant of collaboration diagrams to visually depict SOS-like rules for defining the operational semantics.

5 Conclusions

We have presented in brief a new kind of diagrams, called behaviour diagrams, for modelling the behaviour of autonomous/active/proactive/interactive /.../ active processes (using the UML terminology instances of active classes, or active objects) in the case they are not purely reactive.

From a methodological point of view behaviour diagrams should be used only associated with active classes and in alternative to statecharts, when modelling active objects, depending on the nature of their behaviour.

Behaviour diagrams are quite well incorporated in UML since in their definition we have tried to depart from the UML style and philosophy as less as possible. Clearly we have tried to avoid some of the problematic points of original UML statecharts, that have been found during our previous analysis work (see [16]). Among these points we would like to mention

- incomplete, ambiguous or methodological questionable semantics (the latter is the case of a feature with precise semantics but easy to use in a wrong way, for example the absence of any encapsulation of active classes)
- a quite complicated abstract syntax (the metamodel), where the attempt to use very abstract syntactic categories (metaclasses) generalizing many different cases results in something really complicated; that makes hard to relate the semantics to the syntax (cfr. "transitions", the basic syntactic ingredient, and "compound transitions", the basic semantic ingredient, in

statecharts) and perhaps makes unnecessarily rich the set of statically correct UML models[11]

At the moment we do not know whether behaviour diagrams may be recovered from standard UML [21]by an appropriate use of stereotypes on existing model elements; our attempts have been unsuccessful and we look for further investigation. If the answer would be positive, that will be considered by someone as a nice proof of the good properties of UML is extreme flexibility and richness and adequacy, ... But we should start to think if that is really positive, for users, for tool developers, and so on.

The novelty of behaviour diagrams w.r.t. statecharts is the meaning of transition: in statechart a transition corresponds to a possible reaction to something happened in the system (inside or outside the object) whereas in behaviour diagram a transition corresponds to a possible atomic interaction of the object with the external environment.

We would like to point out that the proposed extension is quite modular w.r.t. the UML structure, the main reason is that the new diagram concerns only the behaviour of active objects, and so no modifications of other parts are needed. Only, to allow time guards on transitions we should introduce new time conditions into OCL. Extensions in this direction of time features are required also by others, for example to introduce a delay action.

Behaviour diagrams have a quite precise semantics that is expressed following different presentation options depending on the prospective readers: normal UML practitioner, formalist, and metamodel lover; but such semantics is precise/ rigorous/sound/well-founded/... because the difference is only in the presentation, and so it has always the "good"properties of a formal one.

Behaviour diagrams should be an alternative to statecharts for modelling the behaviour of active objects, and that should be their main use. They could also be used for modelling the behaviour of a method of an active or passive class, while we do not know if they can be used for use cases, while surely they cannot be used for modelling the behaviour of passive classes.

It is also interesting to analyse the relationships between the notation of the statecharts and that of the behaviour diagrams at the level of the expressive power. We can prove that under the encapsulation conditions of behaviour diagrams, statecharts can be expressed in terms of those.

Finally we would like to cite some of the many current works to extend UML for cope with particular kinds of dynamic systems.

For example, [9], presents an extension of UML for distribution that incorporates in UML a particular technique for developing distributed systems (meta object protocol) with proper newly introduced notations. Instead, [1] propose a UML profile, just lightweight extension, obtained by a few stereotypes for modelling distributed systems made by autonomous processes interacting via tuple spaces.

The extension proposed here is quite different from UML RT [20] with respect to two aspects; first of all the latter is under the keywords "complex event-

[11] The UML terminology for specifications.

driven real-time systems", where the former uses "autonomous, active, proactive, ... processes, agents"; the other point is that the latter is a set of coordinated notations supporting a particular methodology, inspired by ROOM [19]),while the former is just one new notation.

References

1. E. Astesiano and G.Reggio. UML-SPACES: A UML Profile for Distributed Systems Coordinated Via Tuple Spaces. Technical Report DISI–TR–00–20, DISI, Università di Genova, Italy, 2000.
2. S. Cook. The UML Family: Profiles, Prefaces and Packages. In S. Kent A. Evans and B. Selic, editors, *Proc. UML'2000*, LNCS. Springer Verlag, Berlin, 2000.
3. E. Coscia and G. Reggio. JTN: A Java-targeted graphic formal notation for reactive and concurrent systems. In Finance J.-P., editor, *Proc. FASE 99 - Fundamental Approaches to Software Engineering*, number 1577 in LNCS. Springer Verlag, Berlin, 1999.
4. G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modelling: A Graphical Approach to the Operational Semantics of Behavioural Diagrams in UML. In S. Kent A. Evans and B. Selic, editors, *Proc. UML'2000*, LNCS. Springer Verlag, Berlin, 2000.
5. A. Evans, R. France, G. Genilloud, B. Henderson-Sellers, and P. Stevens. Is it feasible to construct a semantics for all of UML?: Aggregation. In S.Kent, A.Evans, and B. Rumpe, editors, *ECOOP'99 Workshop Reader: UML Semantics FAQ*. Springer Verlag, Berlin, 1999.
6. R. France and B. Rumpe. <<UML>>'99 preface. In R France and B. Rumpe, editors, *Proc. UML'99*, LNCS. Springer Verlag, Berlin, 1999.
7. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3), 1987.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, 1985.
9. J.S. Lee, T.H. Kim, G.S. Yoon, J.E. Hong, S.D. Cha, and D.H. Bae. Developing Distributed Software Systems by Incorporating Meta-Object Protocol (*di*MOP) with Unified modelling Language (UML). In *Proc. 4th International Symposium on Autonomous Decemtralizzed Systems*, 1999.
10. Sun Microsystems. JavaSpaces Specification. Technical report, Sun, 1999.
11. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
12. OMG. White paper on the Profile Mechanism – Version 1.0. `http://uml.shl.com/u2wg/default.htm`, 1999.
13. G. Reggio and E. Astesiano. UML Behaviour Diagram Specification. Technical Report DISI–TR–00–28, DISI, Università di Genova, Italy, 2000. In preparation.
14. 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/ReggioEtAl99a.ps`.
15. 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`.

16. 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 LNCS. Springer Verlag, Berlin, 2000.
17. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in LNCS. Springer Verlag, Berlin, 1997.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.
19. B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
20. B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, ObjecTime Limited, 1998.
21. UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.