Parallelism/Concurrency specification within UML

Sébastien Gérard (CEA-LIST: Sebastien.Gerard@cea.fr)

Ileana Ober (Telelogic:Ileana.Ober@telelogic.com)

The concurrency is an important issue to tackle when modelling real-time applications which are intrinsically concurrent. According to the definitions of *real-time* given in (CNRS 1988) and (Stankovic 88), *real-time* implies on one hand that the response are waited at a precise moment (neither before nor after) and on other hand, that such systems are coupled with the real world, therefore highly concurrent.

This document overviews the support for concurrency modelling provided by UML. It describes first the main concept of active object and second a (non exhaustive) list of issues pertaining to concurrency mechanism interactions.

1 ACTIVE OBJECT CONCEPT

The integration of concurrency issues within object-oriented environments generated a large amount of research. In the case of object-oriented languages, most concurrency issues are related to the concept of *active object* (Atkinson 91), (Guerraoui 95) et (Tripathi, Oosten et al. 99). Concrete examples of concurrent object-oriented languages are Act++ , Hybrid (Nierstratz 1987), ABCL (Yonezawa, Shibayama et al. 87), Argus (Liskov 1988), PRAL-RT (Fouquier and Terrier 95), RTGOL (Sourrouille and Lecoeuche 95), TOM , etc.

1.1 UML 1.4 definition

When it comes to concurrency, UML also offers the concept of *active object*, which is an instance of an *active class*.

According to the UML definition, classes may be either *active* or *passive*. The meta-class ***Class*** ([1a] pp. 2-26) has the attribute ***Class::isActive***, whose semantics ([1a] pp. 21) is:

> ***Class::isActive*** specifies whether an ***Object*** of the ***Class*** maintains its own thread of control and runs concurrently with other active ***Objects***. If false, then the ***Operations*** run in the address space and under the control of the active ***Object*** that controls the caller.

All instances of an active class are active objects, therefore the concurrency property is defined in UML as being a class-level property. Active objects own their execution threads and have mechanism for *controlling the incoming invocations* that protect the object's integrity against concurrent invocations.

Here, the *thread of control* represents an abstract notion of control and not an operating system thread. Some relationships between the two may exist, and can be specified in UML, for instance using Component Diagrams. However for the rest of the document we will only focus on the abstract notion of control thread.

The internal concurrency of active objects outcomes from:

- *state machine specification*: concurrent states of the state machine can be perceived as concurrent threads of control;

- *operation executions*: concurrent invocations of a same operation may be executed at a same time, leading to concurrent executions of the operation specification;

- *action specification*: according to the new action semantics definition [ad/01-03-01], the actions contained in an action set may be executed concurrently, unless explicit or causal dependencies constrain their sequencing.

Therefore, it is reasonable to consider that an active object has at least one (possibly many) execution threads.

On the notation point of view, the active property of a class or an object is shown thanks to the heavy border of its rectangular symbol that models it. Otherwise (regular border line), the class/object is passive (see Figure 1).

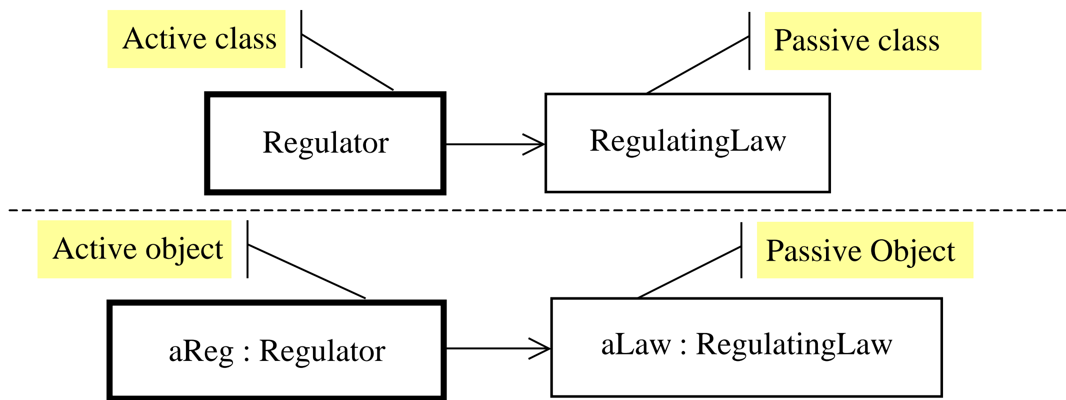

Figure 1 - Active and passive class modelling in UML.

Older versions of UML [UML 1.1], defined two stereotypes, ***« processus »*** and ***« thread »***, to specify how a resource owned by an active class is treated. The first stereotype specified that the active object owns a resource of type heavy-weight flow of control and runs in its own address space, which is not shared with other active objects. The second stereotype specifies that the active object may run with some other active objects in the same address space. These stereotypes disappeared from the standard UML, still, if needed, they could be added at user model level.

The *isActive* attribute of a class is intended to assist the specification of concurrency. However, in order to treat concurrency correctly it is important that the various mechanisms for specifying concurrency work properly together.

The discussion that follows is looking in details at the role and the impact of the *isActive* attribute on the other concurrency mechanisms in UML.

1.2 UML mechanisms for specifying concurrency

In UML there are three levels of concurrency description:

1. *system level* ensured by the concurrency between objects (several objects may run concurrently at a time);
2. *object level* where an object may perform various things at a time (various operation executions, and the state machine execution may co-exist);
3. *operation level*, which can be described by a state machine with concurrent states, or by an internally concurrent action.

UML provides four mechanisms for specifying concurrency:

- the *isActive* meta-attribute of the *Class* meta-class;
- the *concurrency* meta-attribute of the *Operation* meta-class, who is supposed to address primarily passive classes, but it can be used also for active classes;
- the state machines;
- the actions.

At run-time, these mechanisms should work together correctly in order to ensure the correct behaviour of a model.

In this section we will go into the details of these mechanisms and see how they are defined in UML.

As we have seen in the previous section, the *isActive* meta-attribute of the class serves to specify whether the class instances will be active or passive objects. From the concurrency point of view, it specifies whether the instances of this class will control or not the invocation of its operations, and will have (at least one) thread of execution.

The meta-model class *Operation* has an attribute *concurrency* that "specifies the semantics of concurrent calls to the same passive instance" ([UML 1.4] pp. 2-42). The attribute can have one of the values: *sequential*, *guarded*, *concurrent*. The operations of a class are divided into three categories, according to the value of the *concurrency* attribute:

- from the *Sequential* category, only one operation may be active at a time (for a certain object), but the system does not protect them by default, so if two calls are made simultaneously to two such operations, the integrity of the object cannot be guaranteed. However, specifying such a property may be useful when designing the system rather than when implementing it, because it may be not easy and safe to achieve it in practice;

- from the *Guarded* operations, only one is allowed to execute at a time (for a certain object), but this constraint is guaranteed by the object itself. This means that when an operation from the Guarded category is executed, the object will block all the other calls for Guarded operations. How this is done, and whether blocking other operations means that their calling threads are locked or that they return an unavailability response, it is not mentioned in the UML semantics. Guarded operations in UML should not be confused with methods with guard conditions that exist in many other concurrent object models;

- any number of operations from the *Concurrent* category may be active at a time and the object designer must guarantee that the object's integrity is not destroyed by this.

The third mechanism offered by the UML for specifying concurrency details is represented by the behaviour specification using state machines. State machines specify the behaviour of *ModelElements*. Usually, state machines describe either the body of an operation – when the state machine is attached to an operation, or the general behaviour of an object - when it is the state machine of the class. The state machine of the class can be used both to describe the behaviour of the class, and to control the calls addressed to the object the state machine runs into. The states of state machines may be composite or simple. The *CompositeState* meta-class owns a meta attribute called *isConcurrency*. If true, this means that the composite state is directly decomposed into orthogonal regions, the so-called AND-States. The presence of AND-states in the behaviour description of a *ModelElement* whose is "usually associated with concurrent execution". Therefore concurrent states offer means to express concurrent execution.

Another mean of specifying parallelism is offered by the actions. These one can appear:

?? as description of the behaviour of an operation;

?? on state machine transitions, or as entry/exit/do actions in states.

The UML definition of actions (not adopted yet) focuses on allowing the highest level of parallelism possible. In the virtue of this principle, all the actions in a group action are assumed to run in parallel, unless they are implicitly sequentialized by some data dependency or explicitly sequentialized by the designer:

?? implicitly (when there is a data dependency between two actions):

Ex. x := op()+1; - the variable x is assigned only after the operation call and the expression execution.

?? explicitly (when needed by the algorithm):

Ex. UpdateSubscriberList(); SendHolidayMessage(subscrList);

2 INTERACTION BETWEEN VARIOUS MECHANISMS FOR SPECIFYING CONCURRENCY

The mechanisms for specifying concurrency are not orthogonal. Any combination of them is not possible. In the following sections we will underline some aspects of the interaction between the various concurrency mechanisms UML.

2.1 Interaction between "concurrency" and "isActive"

According to UML, the *concurrency* attribute of an operation applies usually to passive Instances. Active objects "usually (although not required)" [UML 1.4 pp.2-52] do not use this mechanism for managing incoming calls, and have their operations *sequential*.

The consequence for passive objects is that callers do have some control over their invocations to operations of passive objects, which is not usual for passive objects, which are supposed not to have any control over concurrent invocations.

UML states no other restriction on the possible combination between various mechanisms for concurrency. However, concurrency specification may imply other side effects on the behaviour of an application depending on other aspects of its modelling. The following sub-sections describes some of the possible interactions between concurrency specification and other modelling properties.

2.2 Interaction between "concurrency" and behaviour specification mechanisms

UML offers basically two means to specify behaviour: state machines and actions (sequence diagrams, pre-/post- conditions, etc. only characterise the behaviour without really specifying it). There is no restriction between the concurrency of an operation and the way its body is described. Indeed, the two aspects are disconnected, as they refer to different moments. The *concurrency* meta-attribute describes the conditions under which an operation gets called, whereas the behaviour specification matters only once the method call is received.

2.3 Interaction between class state machine and "isActive"

UML allows both passive and active objects to own state machines without any constraint. However there are some problems in the connection between passive objects and class state machines.

First, a passive object, by not having its own thread of control, does not have a scope for executing the state machine. As a consequence the state machine will not run continuously as in the case of active objects, instead it will only be able to be executed while the passive object can run on some other object's thread of control.

Following this model, while a passive object's state machine has the control (via the resource of the caller object which has to be directly or indirectly an active object) the incoming message queue may be filled with requests. Therefore instead of executing the request for which it has received the control, the passive object may be busy with some other activities.

The previous section was an example of state machine use for passive object. But it is just a suggestion and regarding to this point a lot of issues are still open. Indeed, a passive object has no thread of control. So it is not able to manage itself its queue. On the above example, it uses the control thread of the calling object. But we could choose that passive object cannot have a queue. To conclude, behaviour specification of passive objects raises some questions:

?? Where are the messages directed towards a passive object stored?

?? If there is a queue, how it is managed?

?? On which execution thread does it execute, etc.

It is out of the scope of the current document to solve this problem, however we underline the fact that the combination of the two concurrency features may give place to restriction according to the real needs. These restrictions could go from the radical case or restricting state machines for passive objects as in (Ober and Stan 1999) to allowing constrained forms of them (Gérard, Voros et al. 00).

2.4 Interaction between the "concurrency" and the class state machine

As the *concurrency* attribute applies mostly to passive objects, the interaction between the concurrency meta-attribute and the class state machine should be done in the context of the interaction between passive objects and class state machines.

In the case of active objects owning a state machine and operations with relevant *concurrency* property, two may be in one of the two situations:

- the state machine and the concurrency property are *completely independent*: in this case the state machine runs normally, in parallel, the object may receive operation invocations. The operation invocations will depend exclusively on the *concurrency* attribute of the operation. In this case there should be a clear priority policy, between the state machine and the operation calls, especially when a same call event may in the same time fire a state transition, and generate a new operation execution.

- the state machine actually *ensures the implementation* of the *concurrency* attribute values. This means that the states and transitions of the state machine serve to model the semantics of the operations concurrency attributes. In this case the state machine could be either automatically generated so that it only reflects the semantics of the concurrency attribute, or the tool automatically changes the statemachine designed by the user, to also embed the semantics of the *concurrency* attribute.

2.5 Interaction between the class state machine and other means of behaviour specification

The interaction between state machines and other means of specifying behaviour yields some problems. Given an UML class, the behaviour of its instances can be specified entirely or partially by its statechart.

According to the standard, state machines can be used in two different ways. In one case, the state machine may specify complete behaviour of its context, typically a class. In that case requesters send requests to the owner of a state machine, and the state machine receiving an

event determines what the effect will be, by attaching actions to transitions, from which complete specifications of operations can be derived.

In the second case, the state machine may be used as a protocol specification, showing the order in which operations may be invoked on a type. Transitions are triggered by call events, and their actions invoke the desired operation. The protocol state machine does not specify actions that describe the behaviour of the operation itself, but shows a change of state determining which operation can be invoked next.

This could mean that the behaviour of an object is *completely controlled* by its state machine if it has one. This means that all the requests sent to the object pass through its state-machine, whether they are *signals* or *method calls*. State machines in UML respond to ***Events*** that may be:

- ***SignalEvents*** – the arrival of a signal sent by another object, which is treated asynchronously, only by the state machine;
- ***TimeEvents*** or ***ChangeEvents*** – which are also treated *only* by the state machine;
- ***CallEvents*** – whose semantics is: "A call event is the reception of a request to invoke an operation. The expected result is the execution of the operation." ((OMG 00) pp. 2-132).

The state machine processes events in a sequential way so that all events are queued and processed following a given policy that the user has to define (FIFO, priority based, earliest deadline based, …), even if they are *method calls*. The semantics of classes and operations and that of state machines interfere dramatically in UML, without many clarifications.

In the case of both active and passive objects, the semantics of a ***CallEvent*** receipt is that the state-machine can decide whether to respond to an operation call by invoking the corresponding method, by performing some *other actions* or by doing nothing. The problem is that UML does not define a notation to distinguish between the case in which the response to a call event is the invocation of the corresponding method and the case in which the response is another one (perhaps none). In fact, from the excerpt cited above, it is unclear whether the methods of an object are executed by its state machine or if the state machine is manipulated by the methods.

Additionally, if you define a class with an operation *op* and a statechart that does not mention *op* at all, then *op* will never be executed, since the state machine does not know about its existence. The fact that you can write an *op* operation inside a class and then you can alter the response to *op* requests in the statechart, thus distributing the specification of *op*, is a rather dangerous characteristic of the UML model, which may lead to dead-lock in the system caused by the callers that wait for return from operations never executed

As we mentioned, the expected result to a ***CallEvent*** is the execution of the operation, but it is not *mandatory* that one.

In conclusion, the interaction between state machines and other means of specifying behaviour should be carefully analysed in order to avoid possible pitfalls.

2.6 Conclusion on the interaction specification and its impact on the global model

In general, if a model is expected to deal with concurrency (and this is the most probable case) the various mechanisms for specifying concurrency should be carefully specified, as well as the relationship between them.

References

ad/01-03-01	OMG Document ad/01-03-01. "Action Semantics Final Submission" March 26, 2001
Atkinson 91	C. Atkinson, "Object-oriented reuse, concurrency and distribution : an Ada-based approach", Addison-Wesley Pub., 1991
CNRS 1988	CNRS, "Le temps-réel", in TSI - Technique et Science Informatiques, Vol.7, n°5, pp.493-500, 1988
Fouquier and Terrier 95	G. Fouquier, F. Terrier, "Introducing priorities into a C++ based actor language for multithread machines", Fifteenth International Conference on Technology of Object Oriented Languages and Systems (TOOLS PACIFIC'94), Melbourne, Australia, November, 1994
Gérard, Voros et al. 00	S. Gérard, N. S. Voros, C. Koulamas, and F. Terrier, "Efficient System Modeling of Complex Real-time Industrial Networks Using The *ACCORD/UML* Methodology," presented at DIPES'2000, Paderborn University, Germany, 00
Guerraoui 95	R. Guerraoui, "Les langages concurrents à objets", in Technique et Science Informatiques (TSI), Vol. 14, n°8, 1995
Liskov 1988	B. Liskov, "Distributed Programming in Argus", in Communication of ACM, Vol. 31, n°3, pp. 300-312, 1988.
Nierstratz 1987	O. M. Nierstratz, "Active Objects in Hybrid", OOPSLA'87, pp. 243-253, 1987
Ober and Stan 1999	I. Ober, I. Stan "On the concurrent object model of UML". EuroPar'99, Toulouse, Sept. 1999, LNCS 1685, p. 1377-1384
Stankovic 88	J. A. Stankovic, "Misconceptions about real-time : a serious problem for next-generation systems", in IEEE Computer, Vol. 21, n°10, pp. 10-19, 1988.
Tripathi, Oosten et al. 99	A. Tripathi, J. V. Oosten, R. Miller, "Object-Oriented Concurrent Programming Languages and Systems", in JOOP, Vol. novembre/decembre, pp. 22-29, 1999
UML 1.4	Unified Modelling Language Specification, v. 1.4, Object Management Group, 1999
Yonezawa, Shibayama et al. 87	A. Yonezawa, E. Shibayama, T. Takada, H. Honda "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1", in *Object-Oriented Concurrent Programming,* pp. 55-89, The MIT Press, Cambridge Massachusetts, 1987