# Thirty one Problems in the Semantics of UML 1.3 Dynamics

G. Reggio* R.J. Wieringa†

September 14, 1999

## 1 Introduction

In this discussion paper we list a number of problems we found with the current dynamic semantics as presented in the defining document of UML: the UML specification version 1.3 [2] (shortly UML 1.3 from now on). For some of these problem, we suggest solutions. The intention is to contribute to a further improvement of the dynamic semantics of the UML. All problems were encountered during the attempt to formalize UML 1.3 semantics or during the attempt to explain the semantics to computer science students.

We distinguish the following kinds of problems

Incompleteness UML 1.3 is silent about this issue.

Inconsistency UML 1.3 says mutually inconsistent things at different places.

Ambiguity UML 1.3 says something that can be interpreted in different ways. In this case, it cannot be even determined whether or not the specification contains mutually inconsistent statements.

Danger UML 1.3 says something about the issue that is precise and consistent with all other statements in the document. Nevertheless, we consider the UML 1.3 statement on this issue dangerous, because for example it makes it easy for the UML user to write a wrong model.

We discuss examples of each of these kinds of problems in the following sections. All page numbers below refer to UML 1.3 [2]. This paper does not consider collaborations, use cases, and activity graphs. These will be the subject of future work.

## 2 Communication

Two important kinds of communication actions that an object can perform are the *call action* and the *signal send action* (UML 1.3 p. 2-85). These actions are directed at one or more receivers. Calls actions may be synchronous (the caller must wait for the receiver(s) to have finished processing the call) or asynchronous (the caller continues without waiting for the receiver to have processed the call). Signal send actions, on the other hand, are always asynchronous. A call or send action creates a stimulus, that may or may not be received by the receiver(s). If received, then the receipt is a *call event* or *signal event*, respectively. An operation call can only be received by objects for which this operation is declared.

*Dipartimento di Informatica e Scienze dell'Informazione Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy. email: `reggio @ disi.unige.it`. `http://www.disi.unige.it/person/ReggioG/`

†Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands. email: `roelw@cs.utwente.nl`. `http://www.cs.utwente.nl/~roelw`.

Similarly, a signal event can only be received by an object for which a *signal reception* is declared. Reception of a call event results in the execution of a method, defined in the static structure diagram, or in the firing of a transition, defined in a statechart for the class of the object. (See UML 1.3 p. 3-138 for the the fact that operations may trigger state machine transitions.) Reception of a signal always results in firing a transition.

This raises the following issues.

1. (Incompleteness) A signal need not be associated with a classifier. What happens if you send one that is not associated to a classifier? Presumably, the receiver must be computed at the moment of sending, as is done for for exceptions (which are signals). But this is nowhere stated explicitly.

2. (Incompleteness) But if the set of receivers must be computed, how is this done? Without further information, the only way to do this is that all objects for which the reception of this signal is declared, will receive the signal. This is unattractive, and as an alternative, the model may simply be viewed as incorrect: Signals must only send to objects with a corresponding reception. (We doubt whether this property is statically checkable.) UML 1.3 is silent on this issue.

3. (Incompleteness) What happens if an asynchronous operation has arguments whose kind is either *out*, *inout* or *return*? The sender continues its operation without waiting for a return value, but this value must nevertheless be sent.

4. (Danger) It is very hard to see a difference between an asynchronous operation implemented by a state transition, and a signal for which a reception in an object is declared. With the introduction in UML 1.3 of asynchronous operations that can be declared for objects, the only difference between signals and operations is that a signal need not to be allocated to an object. The distinction becomes even smaller if we would require that such asynchronous operations cannot have an associated method (this also would eliminate problem 26). Signals are used to specify exceptions (which are signals for which the receivers are computed dynamically). But it might have been better to consider exceptions as a different kind of communication altogether and avoid the complexity of having two different kinds of communication constructs, operation calls and signal sendings, with very similar semantics but different terminology.

# 3 Active and Passive Objects

Information about the distinction between active and passive objects is very scarce in UML 1.3. The only explanations of the difference can be found on UML 1.3 p. 2-149, 2-150 and 3-122. From these pages, we can learn that an active object has its own thread of control, whereas a passive object has not: A passive object can only perform work in the thread of an active object. Note that the concept of a thread of control is not defined in UML 1.3. Using the concept of a thread known from operating systems theory, but without diving into the details of registers, address spaces and program counters, we can best view a thread of control as a token that signifies the capability to do a computation. Only objects that have threads can perform computations. An active object owns a thread and therefore has the capability to do computations. When an object having a thread calls an operation (method) of a passive object, it executes the code provided by that object. Finally, each thread has an event queue, that holds events (e.g. call events and signal events) that the object received while it was doing something else.

## 3.1 Concurrent states

5. (Ambiguity) The state machine of an object may contain and-states. An and-state $S$ has concurrent substates , say $S1$, $S2$, etc. UML 1.3 suggests that if an object is livelocked into an infinite loop in one of its concurrent states, say $S1$, then another thread may nevertheless enter a parallel state,

say $S2$, and trigger a transition there. If the object is passive, this can be understood: There is a thread, that originated from an active object, that is locked in an infinite loop in state $S1$. Another thread, that is not locked in a loop, may then enter $S2$ and do something there. But what if the object is active? If an active object is in state $S$, it is in states $S1$ and $S2$ simultaneously. A thread can only do one thing at a time. If it is deadlocked in $S1$, it cannot do anything in $S2$. Or perhaps it may still interleave actions performed in $S1$ with those performed in $S2$?

6. (Inconsistency) But perhaps an active object starts a concurrent thread for every concurrent state it enters. UML 1.3 p. 2-168 suggests this. But this seems to oppose UML 1.3 p. 2-150, which suggests that triggered and enabled transitions in concurrent states are executed in one thread and in that case, a thread does not start a concurrent thread when a concurrent state is entered.

7. (Incompleteness) If an active object starts concurrent threads for concurrent states, the question arises how many event queues such an object has. Or if there is one queue, how the threads of the object synchronize their dispatching mechanisms.

### 3.2 Concurrency semantics of operations

An object may receive several operation calls simultaneously. The UML allows the definition of several concurrency semantics for each operation. One of these is the *sequential semantics*, which does not guarantee the integrity of the object if several calls to this operation occur simultaneously; another is the *concurrent semantics*, which guarantees the integrity of the object under any number of simultaneous calls of the operation.

8. (Ambiguity) But consider two operations of an object, one of which is sequential and the other concurrent, that both update the same variable. What happens if both are called simultaneously? Can the concurrent operation still maintain its guarantee if the sequential operation messes up the object? UML 1.3 is not clear about this issue. This suggests that the concurrency semantics should not be a property of each operation, but of the object as a whole, that applies to all its operations.

9. (Inconsistency) This resolution is confirmed by one remark in the UML documentation that seems to suggest that a passive object could be implemented by a monitor (UML 1.3 p. 2-149); but nevertheless, concurrency semantics is declared for operations rather than for the complete object.

### 3.3 State machine for passive objects

As a preliminary remark, we assume that passive objects can have state machines. There is no information on this in UML 1.3, but we see no reason to prohibit it. This hunch is supported by the fact that the UML reference manual [1, pages 20 and 132] also mentions state machines for passive objects. But since UML 1.3 completely does not consider this case as all, we have a list of incompletenesses:

10. (Incompleteness) A passive object can only perform activity in the thread of an active object. Suppose a passive object performs some work in thread $T$. So what happens if at that moment, the passive object receives a signal? Is the signal added to the queue of $T$? This problem would be solved if we also define an event queue for passive objects. But such a queue would introduce a new problem, see the next point.

11. (Incompleteness) What happens if a passive object $O$ receives a *call event* (rather than a signal, as above) from a thread $T_1$ when it is already running in another thread $T_2$? In this case, more options exist than above.

- Suppose the operation call is added to the queue of $T_2$. Then, by the time $T_2$ comes around to processing this event, it must wake up $O$, or pull it out of its work in another thread. This is not an attractive option.
- So does the call event bypass the queue of the active object? But then: suppose that the call event has sequential semantics, and that it is currently executing an operation with concurrent semantics. Processing the call event may mess up the result of the concurrent operation. This leads us again to the proposal to make the concurrency semantics a property of passive objects rather than of operations.
- If we define a separate event queue for the passive object, as suggested above, then the issue is raised what happens if the passive object is currently running in a thread with its own event queue. Suppose the passive object finished its work. Does it return control to the owner of the thread or does it use the thread's resources to process the next event in its own queue?

A way out of this mess could be to explicitly allow state machines for passive objects but to prohibit certain constructions, e.g. no signal, no actions, and only operation calls.

## 3.4 Activity States

In a stable state, activities may still be running (UML 1.3 p. 2-150). This means that in a stable state, attributes of the object may nevertheless be changing.

12. (Ambiguity) If a passive object has no thread of its own, then what happens when it enters a basic activity state? Does that activity run in the calling thread, interleaved with other activities in that thread? UML 1.3 p. 2-144 says that when an activity in a state is started, it starts running in its own thread . That is a strange thing for a passive object to do. We would propose that a state machine associated with a passive object cannot have basic activity states.

13. (Incompleteness) If a passive object $O$ is running in thread $T$ (of some active object) and enters a non-basic activity state $S$, then it will start some activity. There are two ways to do this:

    - If the activity runs in $T$, it will run until $S$ is left. As long as this activity is running, $T$ may update $O$ even when $O$ is "idle" or when it is running in some other thread.
    - If on the other hand, the activity spawns its own thread, we get the situation that there is a thread whose only activity is to update a passive object. That would make the passive object active after all.

    To avoid these unattractive situations, we propose that a passive object cannot have activity states.

14. (Incompleteness) By contrast, a basic state of an active object may contain activity. When does this activity execute? If, after a step, the object has an event in its queue, it will immediately proceed processing the next event. But then the activity has no time to run. Again, the problem does not exist if each activity starts its own thread.

15. (Incompleteness) What happens if two concurrent basic states of an active object are both running activities? Can they update the same attributes? But then activities should have sequential, guarded or concurrency semantics just as operations of passive objects have.

16. (Incompleteness) Suppose a state machine enters a hierarchy of nested states, for all of which activities are defined. Then either there is one thread that interleaves all these activities or they all start their own thread. Again, UML 1.3 is silent on this issue.

# 4 Events

## 4.1 Event queue

Each active object runs in its own thread, and each thread has an event queue (UML 1.3 p. 2-149), that holds events that are received while the object was busy doing something else. When an nevent in the queue is removed for processing, it is said to be *dispatched*.

12. (Danger) The event "queue", however, is not a queue: Entries can be removed from it in any order and the analyst must not make any assumptions about this. This is misleading: We propose to call it an *event set*. (All events held in the "queue" are different, even if they consist of calls of the same operation.)

13. (Incompleteness) If the "queue" is really a set, we must make a fairness assumption about it: Any event entered in it is dispatched after finite time.

14. (Incompleteness) We assume that an active object can receive synchronous and asynchronous call events. This is not stated, nor is it prohibited, by UML 1.3.

15. (Incompleteness) But this raises the next issue: What happens if an *active* object receives a synchronous operation call event? Does it bypass the event queue? In that case, the operation would need a concurrency semantics, just as for passive objects. We propose the other option: the call event is added to the event queue of the active object. Together with the fairness assumption mentioned above, this will guarantee that the caller does not have to wait forever.

## 4.2 Deferred events

16. (Incompleteness) An event deferred in a state of a state machine may be dispatched by another, concurrent state of thast state machine. Is this what is intended? (This problem is related to those in section 3.1)

17. (Inconsistency) Do previously deferred events take precedence over the nondeferred ones when dispatching ? UML 1.3 p. 3-160 seems to suggest so.

## 4.3 Change events

18. (Incompleteness) In addition to call events and signal events, there are *change events*, which consist of the fact that some attribute changed its value. A state machine transition may be triggered by a change event. Can change events be deferred? There is no information about this in UML 1.3.

19. (Danger) UML 1.3 says that a change event in one object may refer to the states of all the other objects in the system. This means that the modeled system becomes a completely nondistributed system, where a move in one component may be triggered by a move in any other component of the system. (This problem was considered in the reference manual **??**, which says that the controlled expression should be built using only local attributes). Suggestion: restrict the form of the expressions used to build the change events.

## 4.4 Timed events

20. (Incompleteness) How does an active object process timeouts? If the timeout occurs during a step, does it send a timeout event to its event queue, to be processed some time later? Or are timeouts processed in a Statemate-like manner, e.g., when an event has been consumed, the active object

looks to see which timeouts have passed since the start of the previous step? The fairness assumption for event "queues" mentioned earlier, cannot make a special provision for timeout events. If we would require timeout events to be processed before other events, then this could lead to unfair behavior for these other events.

21. (Incompleteness) In addition to an event queue, an active object has a set of *deferred events.* These are events for which the statechart of the active object has declared that response to them is to be deferred to a later state. Can timeouts be deferred? We would propose not, but there is no information about this in UML 1.3.

# 5 Behavior Specification in Static Structure Diagrams

## 5.1 Constraints

Any UML model element may be associated with a constraint, that expresses some property of it. The semantics of constraints is sufficiently clear in UML 1.3 (see UML 1.3 p. 2-29 and 2-30). A model that does not satisfy its constraints is "wrong". However, there are problems when the constrained element has also a behavior that is precisely defined elsewhere in the model.

22. (Incompleteness) A constraint on a class (as an invariant on the values of its attributes) may be inconsistent with the associated state machine.

23. (Incompleteness) A constraint on an operation (as a pre-post condition) may be inconsistent with the effects of the transitions triggered by its calls in the associated state machine. Suggestion: Add the restriction (a constraint in the metamodel) that constraints on some elements are not allowed whenever the behavior of such elements is defined elsewhere in the model. For example:

    - an active class with an associated state machine cannot have a constraint in the class diagram
    - an operation appearing in a transition of the associated state machine cannot have constraints in the class diagram

Some constructs of UML 1.3 that pose problems that are specializations of this general problem are treated next.

## 5.2 Operations

24. (Inconsistency) A query operation is an operation that does not update its owning object when called. However, the operation may be specified by a method that nevertheless modifies the state. An obvious fix to this inconsistency is to require the method to be read-only with respect to the owning object. This cannot be statically checked, but we can deal with it in a way similar to what has been done in the Ada semantics definition, namely by calling models that violate this constraint "erroneous" and leaving tools free to handle these errors in whatever way.

25. (Incompleteness) However, this addition is not enough, for the operation can also (even additionally) be specified by a transition in the state machine of the object; and even if no update actions are triggered by the transition, the transition causes a change of state and this itself is an update. We suggest that a query operation should not be defined by a state transition.

26. (Incompleteness) UML 1.3 nowhere prohibits an operation to be implemented by both a method and a state transition. This may cause inconsistent object behavior.

27. (Incompleteness) Since an object can have several statecharts, an operation may even be implemented by several state transitions. (Any model element can have several state machine, see UML 1.3 p. 2-130.)

28. (Inconsistency) We may also have an object with a state machine and some operations that are defined by a method and that do not trigger a transition in the state machine. This is inconsistent with the statement in UML 1.3 that a state machine describes all behavior of an object.

### 5.3 Signals

29. (Incompleteness) A signal reception in a class may have a "specification" (see the specification attribute or signal in the metamodel) expressing some property about the effect of receiving such signal in such class. This specification can be inconsistent with the effects of the transitions triggered by such signal in the state machine(s) associated with the class (UML 1.3 p. 2-92, 2-93).

## 6 Miscellaneous

30. (Inconsistency) UML 1.3 p. 2-102 mention assignments actions, but these are omitted from the metamodel on UML 1.3 p. 2-85.

31. (Incompleteness) An internal transition triggered by event e conflicts with an outgoing transition of the same state triggered by the same event e. In Statemate, the outgoing transition has priority over the internal one. But how is the priority defined in the UML? UML 1.3 p. 2-151 is silent on this.

## 7 Conclusions

If a language specification is incomplete, vague and inconsistent, it is impossible to explain to students who must use the language in their assignments. The presence of dangerous constructs makes it even harder to explain the language. One of the benefits of formalization is that it uncovers these issues and allows us to search for resolutions and alternatives that do not suffer these problems. We hope that our analysis will contribute the achievement of this goal.

## References

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

[2] UML Revision Task Force. *OMG UML Specification*. Object Management Group, june 1999. http://uml.shl.com.