A Semantics of Communicating Reactive Objects with Timing*

Jozef Hooman[1,2] & Mark B. van der Zwaag[1]

[1]*Department of Computing Science, University of Nijmegen, The Netherlands*
[2]*Embedded Systems Institute, Eindhoven, The Netherlands*

January 15, 2004

Abstract

The aim of this work is to provide a formal foundation for the unambiguous description of real-time, reactive, embedded systems in UML. For this application domain, we define the meaning of basic class diagrams where the behavior of objects is described by state machines. These reactive objects may communicate by means of asynchronous signals and synchronous operation calls. The notion of a thread of control is captured by a so-called activity group, which is a set of objects which contains exactly one active object and where at most one object may be executing. Explicit timing is realized via local clocks and an urgency predicate on transitions. We define a formal semantics for this kernel language, based on the run-to-completion paradigm. We show that this combination of communication primitives and execution mechanism gives rise to a large number of questions and discuss the decisions taken in the proposed semantics. The resulting semantics has been defined in the typed logic of the interactive theorem prover PVS.

1 Introduction

We present a formal semantics for a system consisting of concurrent reactive objects, specified by a UML class diagram with state machines. This work is carried out in the context of the EU project Omega (Correct Development of Real-Time Embedded systems). This project aims to improve the quality of software for embedded systems by the use of formal techniques. In particular, the focus of the project is on real-time aspects of systems. The Omega project addresses techniques such as model-checking of timed and untimed models, interactive theorem proving to support compositional reasoning, refinement rules relating different levels of abstraction, and synthesis from specifications. The developed formal tools are connected to commercial UML tools

*This work has been supported by EU-project IST 33522 – OMEGA "Correct Development of Real-Time Embedded Systems". For more information, see `http://www-omega.imag.fr/`.

via the XMI representation.[1] Moreover, the aim is to propose a methodology for the software development process. The unifying basis of all this work is a formal semantics of a suitable subset of UML for embedded systems.

Starting point for the formal semantics used within Omega was an operational semantics [5], which was inspired by the execution mechanism of the CASE tool Rhapsody [10]. This semantics was also closely related to the implementation of untimed model checking by project partner OFFIS [4]. The aim of this paper is to define a more abstract semantics which is more suitable for interactive theorem proving and which clarifies a number of semantic questions and decisions.

To allow system verification by theorem proving, we tried to define the initial Omega semantics [5] in the typed higher-order logic of the interactive theorem-prover PVS [11, 13]. The rigorous formalization in PVS by itself revealed a number of ambiguities. Also, questions arose concerned, for example, the passing of control, the dispatching of signals, synchronization of operation calls, etc. In this paper, we identify these issues, and present the decisions taken to resolve them. Moreover, we show how a continuous notion of time can be added in an orthogonal way.

Related Work Strongly related to our work is the formalization of active classes and associated state machines by Reggio et al [14]. They define a labelled transition system using the algebraic specification language CASL, also leading to a number of related questions about the meaning of UML models. Their decision is usually to consider the most general case; for instance, an active object may correspond to an arbitrary number of threads and the event "queue" is a multiset of events. Our decisions are mainly based on feedback from industrial users, current UML-based CASE tools for real-time systems and the aim to enable formal verification of embedded systems. This leads to more specific choices, such as a single thread of control per object and a simple FIFO event queue.

Our kernel language is close to the core UML language described in [7]. The meaning of event generation, operation invocation, and composition is based on the Rhapsody tool and basically the same as our informal meaning. The basic outline of our semantic model is similar to that of [8] which uses an abstract request mechanism and no distinction between asynchronous events and synchronous communication. The focus of that paper is more on behavioral conformity for inheritance and various types of refinement.

Not present in the literature mentioned above is a continuous notion of time. E.g. in [4] there is only a notion of a discrete step. Our timing extension is based on classical timed automata [1], but unlike e.g. Uppaal [16] we do not use invariants but an urgency predicate which is a restricted version of the timed automata with deadlines of [3]. A more high-level syntax for UML with real-time annotations, as developed in the Omega project, can be found in [6]. They can be translated into our basic timing framework.

Theorem Proving in Omega The PVS representation of the semantics presented here plays an important role in a tool that is developed in the Omega project, since it

[1]Unfortunately, not all tools export XMI and currently there are slight differences in the generated XMI representation.

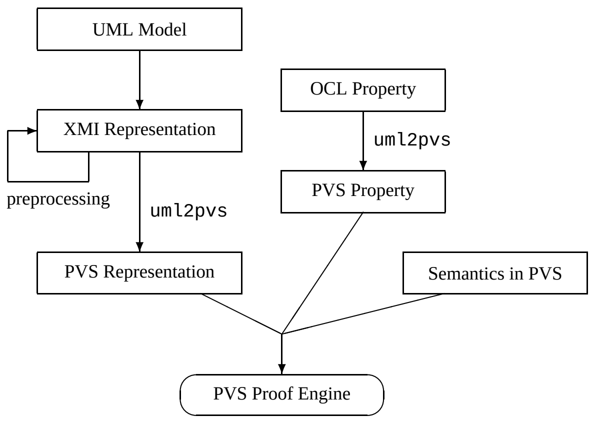


Figure 1: Verification of UML models in PVS.

enables formal reasoning and mechanized proof checking on concrete UML models. The idea is depicted in Figure 1. By means of a commercial CASE tool, a user constructs a concrete UML model, in the current version consisting of a class diagram and corresponding state machines. A textual representation of the model in XMI[2] should be provided by the UML-based CASE tool. Some automated preprocessing may be applied at this level, e.g., the flattening of state machines, and subsequently the representation of the model is translated to a representation of the model in PVS by the `uml2pvs` tool. The PVS representation of the model is combined with the general definition of the model-independent semantics, which assigns meaning to the model as a labelled transition system. This semantics defines the meaning of an UML model as a set of runs, denoting the snapshots of the system state during execution and the actions that cause the state change.

Properties of the UML-model may be expressed by the user in OCL, which has been extended with a notion of time. The `uml2pvs` tool translates these OCL constraints to PVS. As an alternative, the user may express properties directly in the higher-order logic of PVS. Basically, any property on runs that can be expressed in PVS, including safety and liveness properties. The interactive proof checker of PVS can be used to prove that the UML model, i.e., the set of its runs, satisfies certain properties. Proving properties is a complex task and requires quite some expertise, but the verifier may use certain *strategies* that can handle reoccurring patterns in the proofs. Within Omega, the TLPVS package is used to experiment with powerful strategies for the proof of temporal properties [12].

Overview In our semantics, time is modelled as an orthogonal aspect, and it turned out that especially for the untimed part, a large number of questions arose about the precise meaning of particular concepts. Hence, we first discuss the time-independent aspects of the semantics and later show how time can be added.

[2]See `http://www.omg.org/technology/documents/modeling_spec_catalog.htm` for the latest UML-related specifications.

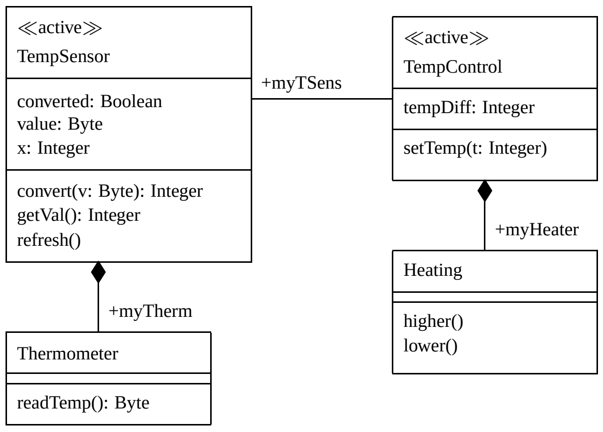


Figure 2: Example class diagram.

In Section 2 we start with the definition of an untimed kernel language, that is, the part of UML for which we give a semantics. The main semantic choices are discussed in Section 3. In Section 4 we explain in detail how we model the system behavior as a labelled transition system; this explanation follows the formal definition of the semantics in PVS which can be found at [9]. A notion of threads of control, which is realized using the concept of so-called activity groups, is added in Section 5. The introduction of real-time can be found in Section 6; we explain how a timed labelled transition system can be defined on top of the untimed semantics. We end with some concluding remarks in section 7.

2 Kernel Language

In the Omega kernel language, classes are declared by conventional class diagrams with attributes and operations. Consider, for example, the class diagram in Figure 2. This diagram models a temperature control system; a temperature controller (of class *TempControl*) contains a heater and has access to a sensor. The sensor can read the temperature from its thermometer.

The controller responds to the external signal $\mathit{setTemp}(t)$ which is a request to keep the temperature at level t. The heater responds to the signals *higher* and *lower*. The sensor responds to the signal *refresh* by reading the temperature value from its thermometer. The sensor has an operation *getVal* which returns the latest temperature value that has been read. This operation involves the conversion of this value to an integer in case this conversion was not performed before. The boolean attribute *converted* indicates whether the latest value has been converted (in which case it is assigned to the attribute x).

The controller and sensor classes are *active*, all others are *passive*. Similarly, the corresponding objects — the executing instances of classes — are either active or passive. Typically, active objects correspond to a thread of control, called *activity group* in

Section 5. Finally, the black diamonds on two of the associations represent a composition relation (strong aggregation). In our example, a thermometer is seen as a part of a sensor, and a heater is part of a controller. Often the passive parts of an active object belong to the same activity group. Associations between classes will be represented by reference attributes. For instance, class *TempControl* has an (implicit) attribute *myTSens* of type *TempSensor*.

For each class the behavior of its objects can be defined by means of a state machine and methods (program text) for its so-called *primitive operations*. Other operations are defined by means of the state machine and called *triggered operations* because they may trigger a transition. Moreover, the state machine specifies the response to signals.

In our formalization, state machines are assumed to be flat, that is, to consist of transitions between locations; there are no hierarchical states or pseudo-states. The flattening of state machines is part of the preprocessing of the model; tool support is developed in the Omega project. For uniformity, we assume that every class has a state machine, but this state machine may be empty, that is, consist of a single initial location and no transitions.

A state machine transition is written as

$$l \xrightarrow{e[g]/act} l',$$

where l is the source and l' is the target location, e is the trigger event, g is the boolean guard, and $/act$ is the action part of the transition. The idea is that an object in location l may reach location l' by the execution of the action part of the transition, provided the guard is satisfied and it is triggered by the trigger event.

A trigger event is either an operation call, or a signal. A transition may be untriggered. The guard is a boolean expression which can be evaluated locally by the object without side-effects (a *true* guard is often omitted). The action part of a transition is a list of basic actions. The list may be empty, denoting skip. In this paper, we consider the following five basic actions: assign a value or reference to an attribute, create an object and assign a reference to it to an attribute, send a signal to a particular object, call an operation of a particular object and assign a result value to an attribute, return from a call.

As an example, Figure 3 shows the state machine for class *TempSensor* of Figure 2. The incoming arrow with the black circle at the top indicates the initial location; the initial value for the attribute *converted* is *false*. The sensor first reads its thermometer by invocation of the operation *readTemp*; the result is assigned to the attribute *value*. In the location *ready* the sensor can either be triggered by the signal *refresh*, in which case it reads a new temperature value, or it can be triggered by a call of the operation *getVal*. The return value for this call must be an integer, which means that the sensor may have to convert the latest value that it has read. It can do so by calling its own method *convert*. The boolean attribute *converted* indicates whether the latest value has been converted; after conversion values are stored at attribute *x*.

Intuitively, all objects concurrently execute their state machine. When an object calls an operation it becomes blocked until the callee executes a corresponding return. Signals are sent asynchronously, i.e. the sender may continue immediately and the signal is put in a signal queue at the receiver side. Objects are selected from the signal

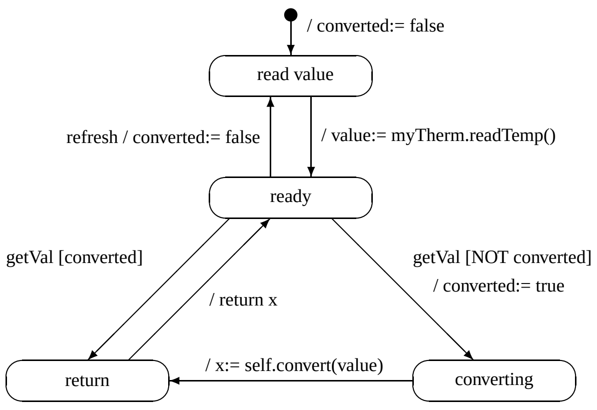


Figure 3: State machine for the *TempSensor* class.

queue and then either trigger a transition or are discarded if they do not trigger a transition in the current location. An exception are signals that are declared to be *deferrable* in the current location; they are not discarded and remain in the signal queue.

Although not shown in Figure 2, the kernel language also allows the generalization (inheritance) relation between classes. Concerning inheritance we face the usual questions (see, for instance [8]). The main point is to which extent behavior of a superclass is inherited by a subclass. Conforming to the typical use of inheritance in industrial applications, we allow that a subclass redefines the behavior of an inherited operation. We take the following decision: if a child class has a state machine, then it overrides the state machine of the parent completely; otherwise it inherits the state machine of the parent. In our formal semantics, we assume that all information about inherited attributes, operations, and state machines has been included in each class itself by some simple (automated) preprocessing. We also record for each object the corresponding class, thus obtaining conventional polymorphism.

3 Semantic Choices

Our first attempts to formalize the informal meaning of UML diagrams, as sketched in the previous section, revealed a number of questions to be answered and a number of decisions had to be taken.

The first decision concerns the concurrency model; the concurrent execution of the objects is modelled by interleaving the steps of each object. An execution of a system consisting of a set of concurrent objects is represented by a sequence of snapshots, modelling the state of affairs at a point during execution, where each pair of successive states represents some step of the system. This leads to the question:

What constitutes a step?

There are several choices for the granularity of steps. In related versions of the Omega semantics [5, 4] there are small “bookkeeping” steps, e.g. to discard signals from the queue. We also experimented with several possibilities, e.g., a version in which each action of the action list on a transition forms a step.

This detailed level of granularity, however, turned out to be quite cumbersome for interactive verification. Since the verifier intuitively would like to reason in terms of transitions, we decided to formulate the semantics in such a way that each step corresponds to the execution of a transition in a state machine of an object. In the case of triggered operations this means that the execution of the transition with the call action and the execution of the transition that is triggered by the call are combined into one step of the system.

Note that if the action part of the callee would again contain an operation call, this would lead to a cascading sequence of synchronizations. Since this greatly complicates the semantics, we disallow an operation call in the action part of a transition with a call trigger. To get additional simplifications, which is essential for successful verification, we make the restriction slightly stronger. Let the *primitive* actions be those actions which, by themselves, are always enabled, namely, local assignments and signal emission. Hence the three non-primitive actions are: operation calls, return, and object creation. We require that the action part of a transition is either a list of primitive actions or a single non-primitive action. Moreover, all transitions with a call trigger should have a primitive action part.

Observe that the enabledness of a transition with a primitive action part depends only on the trigger event and the guard, while non-primitive actions may contribute to the enabledness condition. See Section 4.2 for a description of the actions and their semantics.

Another important question concerns the set of possible interleavings.

When may an object accept a new operation call or a new signal?

An important decision is to adopt the *run-to-completion* semantics, as defined by the ROOM methodology [15], that is, when an object has been triggered by an operation call or a signal, it must become *stable* before it can accept a new event. An object is stable if, in its current location, it has no outgoing untriggered transitions for which the guard is satisfied. Thus a stable object can only proceed by accepting a call or a signal. For instance, in Figure 3 only state *ready* is stable.

The run-to-completion assumption is a reasonable and intuitive assumption which reduces the number of interleavings and the amount of interference between objects. This is also the motivation for the decision is to disallow re-entrance of triggered operations. In general, for the execution of an operation call the callee must be able to accept it. As mentioned above the callee must be stable. To disallow re-entrance, it may not already be processing a call; during the processing of the call (which is completed by the execution of a return action) no other calls to triggered operations are accepted by the callee. With the call, the caller becomes suspended (blocked). It remains suspended until the reception of the result value.

A related question concerns the acceptance of primitive operations. Primitive operations (also called methods) are implemented by a piece of code and cannot be used as a trigger. For simplicity, we require here that the method body does not contain operation calls. Moreover, we assume that the result can be computed atomically and the complete execution, including the method call, the computation, and the return of the result, is modelled as one step in the semantics.

Therefore, suspending the caller of a primitive operation is not needed. In order to allow an object to call its own methods, we allow the acceptance of primitive operations in *any* state, so in particular also when the object is unstable or processing a triggered operation.

The last main question we discuss here concerns the treatment of signals, which partly corresponds to an explicit variation point in the description of UML.

How are signals stored, selected, and discarded?

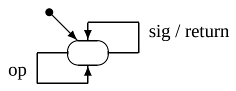
Questions are, for instance, whether we have signal queues for sets of objects or for single objects, how to select signals from the queue, how to put deferrable signals that do not trigger a transition back into the queue, etc.

To obtain relatively simple, predictable behavior, we made the following decisions. An emitted signal is placed in a first-in-first-out queue at the receiver. A signal can be accepted from this queue by the receiver if it triggers a transition. Moreover, to be accepted, it must be the first triggering signal in the queue. This acceptance is combined with the execution of the triggered transition into one step of the semantics. During this step all preceding signals in the queue (which do not trigger a transition in the current state) are discarded, except if they have been made *deferrable* for the current location. The deferrable preceding signals maintain their order in the queue.

3.1 Examples

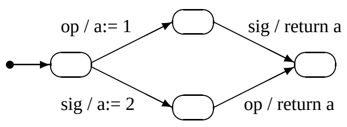
We present a few small examples to illustrate some consequences of the decisions described above.

As a first example, consider an object executing the following state machine:

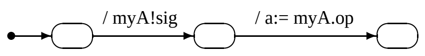


The object is always stable: both transitions are triggered. Initially, only the operation can be accepted; the return action is only enabled if the object is processing a call (the return action is non-primitive). After the acceptance of an operation, the acceptance of a new operation is not enabled: the object must finish the operation first. It finishes the operation by the return action, for which a signal is required. So, the object will alternate between the two transitions (provided that the environment provides the events).

As a second example, consider an object of class *A* with

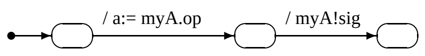


And an object of class *B* with



where *myA* refers to the object of class *A*. The concurrent behavior of these two objects has two traces ending in the final state, i.e., the state where both objects are in the rightmost location. One of these traces describes the case where the operation call has been accepted before the signal — in this case the final value of *a* is 1; in the other case the signal is accepted first and the final value is 2.

Alternatively, if we rewrite the state machine of class *B* to



then we see that a deadlock situation is reached after the operation call: the signal cannot be emitted by the object of class *B* because it is suspended (waiting for the call to return).

4 Semantics in PVS

In this section we give the general outline of the formal semantics, as it is defined in the language of the theorem prover PVS. The complete listing of the PVS theories can be found at [9]. Given a concrete UML model, we define an untimed labelled transition system (LTS). In PVS this is done by means of general theories, where the essential characteristics of the concrete model are represented by a number of parameters. Timing can be added on top of this LTS as is explained in Section 6.2.

The general idea is that an execution of the UML model is represented by a *run* (execution trace) of the LTS; a run is a sequence

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \xrightarrow{l_3} \cdots$$

of steps of the LTS, where the s_i are *states*, representing a snapshot of the system during execution and the l_i are *labels* representing the execution of a state machine transition for some object.

The states of the untimed LTS are defined in Section 4.1, including the definition of stability. Next we define in Section 4.2 the semantics of the action part of a state machine transition, i.e., its precondition and its effect. This is used in Section 4.3 to define the *steps*, that is, the global transition relation which forms the basis of runs.

4.1 States

Assume given a countably infinite set of object identifiers and a mapping from object identifiers to the set of class names.

A *local state* contains all relevant local information concerning an object; it consists of the following:

1. A *valuation* that assigns values to attributes and references.

2. The *status* of the object: an object can be
   - *dormant* if it is not yet created;
   - *free* if it is not processing a triggered operation call;
   - $busy(id, a)$ if it has accepted a call for which it has not returned the result yet; *id* is the identity of the caller, and *a* is the attribute the result must be assigned to.
3. The current state machine location of the object.
4. A boolean indicating whether the object is currently suspended.
5. The signal queue of the object.

A (global) *state* is a collection of local information: it is a mapping from object identifiers to local states.
A few auxiliary definitions concerning stability:

- An object is *ready* in some state, if it is non-dormant and not suspended.
- A transition is *locally enabled* for an object in some state, if it belongs to the object's state machine, its source is the object's current location, and the boolean guard is satisfied.
- An object is *stable* in some state, if it is *ready* and all locally enabled transitions have a trigger.

Observe that enabledness of a transition does not only depend on its guard and trigger, also the action part may impose conditions on enabledness. For example, a transition with a triggered operation call as action is enabled for an object only if it is locally enabled and the callee object is ready to accept the call. These additional conditions are defined in the next section as preconditions on non-primitive actions.

4.2 Action Semantics

We define the precondition and the effect of the action part of a transition. If the action part is a list of primitive actions, the *precondition* is trivial (i.e. *true*) and the *effect* of their execution is defined recursively: the elements of the list are executed one by one, from left to right, as follows.

Assignment: $a := exp$

> ***Effect:*** Assign the current value of the expression *exp* to attribute *a*.

Signal Emission: $r!sig(exp)$

> ***Effect:*** Insert a signal with signal name *sig* and the value of the expression *exp* in the signal queue of the object referred to by *r*.

The non-primitive actions have the following semantics:

Triggered Operation Call: $a := r.op(exp)$, where *op* is triggered operation of an object referred to by the reference *r*. The return value is assigned to the attribute *a*.

Precondition: (1) The callee must be stable; (2) the call must match the trigger of a transition *t* at the callee, and (3) *t* must be locally enabled for the callee (after the assignment of the value of *exp* to the attribute specified in the trigger expression of *t*).

Note that the action part of the triggered transition is primitive (by assumption) and therefore trivially enabled.

Effect: The callee is triggered (i.e., changes location to the target of *t* and assigns the value of *exp* to the designated attribute) and the (primitive) action part of *t* is executed. The caller becomes suspended, and the status of the callee becomes $busy(id, a)$, where *id* is the identity of the caller, and *a* is the attribute the result must be assigned to.

Primitive Operation Call: $a := r.m$, where *m* is a method (primitive operation) and *r* refers to the callee.

Precondition: The callee must be non-dormant.

Effect: The result value, which is defined as the value of the operation's method in the local state of the callee, is assigned to the attribute *a* at the caller.

Return: *return exp*, where *exp* is the result expression.

Precondition: The executing object must be processing a call, that is, its status value must be $busy(id, a)$, where *id* is the identity of the caller, and *a* is the attribute the result must be assigned to. Furthermore, it is required that the caller is non-dormant.

Effect: The result is that the caller becomes un-suspended, and that the value of *exp* is assigned to the attribute *a* of the caller. The status of the callee becomes *free*.

Object Creation: $r :=$ *new c*, where *c* is a class.

Precondition: The identity of a dormant object of class *c* is available to the executing object (the creator).

Effect: The new object is initialized as follows: it is free and not suspended; its signal queue is empty; and it has the initial valuation and state machine location associated with its class. The reference *r* will refer to the new object for the creator.

Note that in this simple version, the initialization of the new object is completely determined by its class—not by its creator. We have also defined more complex variants with entry scripts, and recursive creation of objects for hierarchical composition of objects (aggregation).

4.3 Semantics of State Machine Transitions

Finally, we define the meaning of state machine transitions, defining its precondition and effect, leading to the steps of our semantics. We make a case distinction on the kind of triggering of the transition. Since transitions triggered by an operation call are executed as part of the execution of the call by the caller object (see the action semantics of a triggered operation call in the previous section), it remains to consider an untriggered or a signal-triggered transition *t* of object *p*.

Untriggered: Transition *t* has no trigger event.

> ***Precondition:*** (1) The guard of *t* is satisfied, and (2) the action part of *t* is enabled (as described in Section 4.2).
>
> ***Effect:*** The next state is defined as the effect of the action part on the current state (as defined in Section 4.2), where the location of *p* is changed to the target location of *t*.

Signal-triggered: Transition *t* is triggered by a signal.

> ***Precondition:*** (1) *p* is stable; (2) the *n*-th signal *sig* from the signal queue of *p* triggers *t*, i.e., after the assignment of the parameter of *sig* to the designated attribute, the guard of *t* is satisfied and the action part of *t* is enabled; and (3) *sig* is the first triggering signal in the queue of *p*: all preceding signals may not trigger a transition of *p*.
>
> ***Effect:*** *p* changes location to the target of *t*, and executes the action part of *t* after assigning the parameter value to the designated attribute, and *cleaning up* the signal queue of *p*: *sig* is removed and also all the preceding signals that are not *deferrable* in the current location are removed. The deferrable preceding signals maintain their order in the queue.

Hence, a *step* $s \xrightarrow{l} s'$ corresponds to either an untriggered or a signal-triggered transition *t* where s satisfies the precondition of *t*. Note that that the action part of *t* may contain an operation call which has to synchronize with the trigger of another transition *t*′. Then s also satisfies the precondition of *t*′, and *t*′ is executed in the same step, i.e., s' is obtained from s by combining the effect predicates of *t* and *t*′.

The transitive closure of the step relation leads to the set of *runs* of the form

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \xrightarrow{l_3} \cdots$$

representing all possible executions of the UML model.

5 Activity Groups; Sharing Control

In the semantics defined in Section 4, all objects are running asynchronously, which is modelled by interleaving the transitions of all objects. In this section we add a dynamic assignment of *control* to objects which restricts the concurrency of the system; only an object that has control is allowed to execute a state machine transition. To achieve this, the set of objects is partitioned into *activity groups* which are centered around active objects: a class can be active or passive, and this leads to active or passive objects at run-time. For example, in the system of Figure 2, a *TempSensor* object is active because it belongs to an active class, while a *Thermometer* object is passive, because its class is not active. In this case, a *TempSensor* and a *Thermometer* object together constitute an activity group. Similarly, a *TempControl* and a *Heating* object form an activity group.

The partitioning is represented by assigning an active parent object to every object; an active object is its own active parent. At any point in time, in every activity group, exactly one object has the control; we refer to this object as the group's *control object*.

During execution the control within a group may shift from one object to another. The main question here is: when is it allowed to change control? We decided that control changes when performing a call inside the same group, otherwise an object may only lose control if it is stable.

This notion of activity groups is comparable to that of *threads of control*; an active object corresponds to a thread of control and at most one thread is active in each object. To avoid confusion with, e.g., Java-like threads, we decided to avoid the term "thread" and use "activity group" instead.

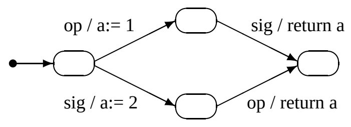
5.1 Operation Calls

Concerning triggered operations we now face questions about the flow of control and about when to pass the result back and when to change control; when the callee becomes stable or immediately when the result is available? We decided the following: a successful, synchronous call of a triggered operation requires that the caller has control and

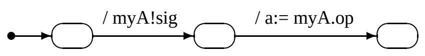
- if the callee belongs to the same activity group, then the control changes from caller to callee;
- if the callee is in another group, then the caller maintains the control in its group. The callee must either already have or take the control in its group.

Execution of a return action need not lead to a control change; if callee and caller are in the same activity group, the callee must first become stable, before the control becomes available again for the caller. An object does not need the control to answer *primitive* operations; it may answer these in any state.

We revisit the second example from Section 3.1. For convenience, we repeat the state machines. For class *A*:



For class *B*:



Consider an object *p* of class *A* and an object *q* of class *B*. Suppose the two objects share control, i.e., they are part of the same activity group. Let *q* start with the control. This time, there is only one trace leading to the final state, and it leads to the final value 1: directly after emitting the signal, *q* is unstable, and hence it cannot lose the control. The next step is the synchronous operation call during which the control is passed to *p*.

5.2 Semantics in PVS

In the local state of objects, see Section 4.1, we record the following additional information: (1) for passive objects, the identity of the active object that is leading the object's activity group; and (2) for active objects, the identity of the object that is currently in control within its group.

Moreover, the status value *busy*, indicating that the object is currently processing an operation call, is now refined into the following two values

- As before, an object is *busy* if it has not returned a result value yet.
- An object may still be *completing* the call after it has returned a value, if the caller belongs to the same group: then the object remains in control until it becomes stable.

In case the caller is from another group, the caller remains in control while it is suspended and the call is being processed by the callee. Then, after the return the callee becomes *free* and both objects can continue.

All *preconditions* are extended with the condition that the control must be available for the executing object. The control is available for an object if it either already has the control, or the control object is stable; but in case the control object is *completing* a call, the caller of the call gets priority in the assignment of the control. In this last case, an object can take the control if (1) it is the caller, or (2) it is not the caller, but the caller is stable. Part of the control change in this case is that the status of the control object becomes *free*. In the *effect* predicates we incorporate the result of the change of control.

Observe that there is no separate step for a change in control (we experimented with this in earlier versions). As a result, the set of runs of a UML model with sharing of control is a subset of the same model without activity groups (which can be obtained by making all classes active, so that all activity groups have exactly one member).

6 Adding Time

In this section, the semantics is extended with a continuous notion of time. As in timed automata [1], timing constraints are expressed in state machines using local clocks; an object may reset its clocks (like a local assignment), and express conditions on clock values in the guard of a state machine transition. We also introduce a global notion of time because it is convenient for specifications.

Our model of timing is an orthogonal feature to the untimed semantics; the passing of time is modelled by a global delay step that increases the global time and adjusts all local clocks accordingly. In this model, all other steps do not increase the clocks. We can therefore define the system behavior as a labelled transition system in which every step either corresponds to a step of the untimed semantics (execution of a state machine transition), or a time passing step.

The main question is how to ensure progress. For instance, in Uppaal [16] clock invariants on locations are used to block delay steps and to ensure that certain transitions will be taken. In our current semantics we decided not to use invariants, but an urgency predicate on transitions; this is a special case of the *timed automata with deadlines* of [3]. Our motivation for this choice is that the condition for the progress of time is less complicated if we work with urgency of transitions. When using invariants, time may progress if no invariants are violated; in our case time may not progress if urgent transitions are enabled. With urgency, the condition for the progress of time is defined in terms of the steps of the untimed transition system, whereas with invariants the condition also depends on the local states of objects.

A transition then is either urgent or non-urgent. Urgency can for example be used to model time-outs: if the transition

$$l \xrightarrow{[x=3]} l'$$

is urgent, or *eager*, then the location l must be left when the value of the local clock x is 3. As another example, take

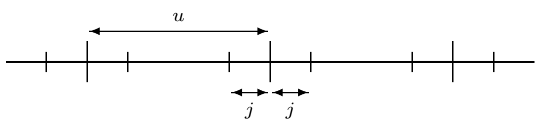
$$l \xrightarrow{[2\leq x\leq 4]} l'.$$

If this transition is non-urgent (also called *lazy*), then the transition may be taken at any time between 2 and 4 on the local clock x. It may happen that time passes past time 4, in which case the transition is not taken at all (it may be taken later, after the clock x has been reset).

6.1 Example

As an example we take a simplified part of a case study of the Omega project provided by the Dutch Aerospace Laboratory (NLR). We concentrate here on the transmission of signals between a sender and a receiver. The sender may fail to send a signal and the receiver should detect such failures.

The sender should periodically emit a signal to the receiver with some jitter. Let u be the cycle time, and let j be the jitter time:



Every cycle there is a time interval of length $v = 2j$ during which the signal can be emitted. For a convenient modelling of the sender we shift our perspective on the cycle time; a cycle starts at the end of the emission period.

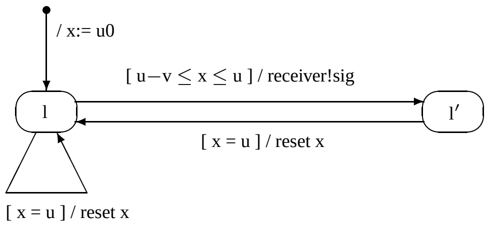


Figure 4: Sender state machine. All transitions except the signal emission transition from l to l' are urgent. Let u_0, with $u_0 \leq u$ be the local initial time.

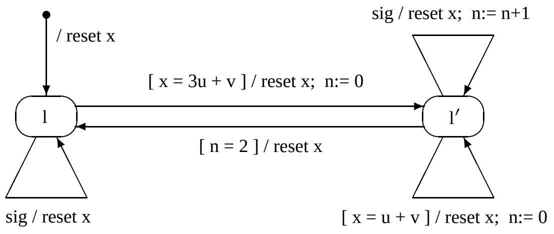
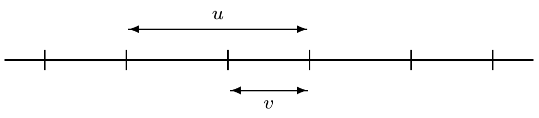


Figure 5: Receiver state machine. All transitions are urgent.



The sender state machine is depicted in Figure 4. Initially the value of the local clock x is u_0, with $u_0 \leq u$. The signal may be sent to the receiver during the time interval from $u - v$ to u on the clock x. This transition is non-urgent, which means that time may progress while it is enabled. If the signal has not been sent at time u, time may not progress further because of the urgent time-out transition (the self-transition on location l with guard $x = u$). At this point there is still the choice between the time-out and the emission. After the self-transition has been taken, and x has been reset, a new cycle starts. If the emission does take place, the sender must wait in the location l' for the end of the cycle, from which it urgently re-enters the sending location l.

The receiver state machine is shown in Figure 5. The receiver can accept the signals; assume for simplicity that it does not perform anything in response to the reception. We concentrate on the detection of a failure in the transmission. The receiver decides that there is some error if it has not received a signal from the sender for three consecutive cycles. We model this with an urgent time-out transition which is taken if no signal is received for $3u + v$ time. The receiver stays in the error location l' until it has received a signal in two consecutive cycles; it then decides that it it safe to return to the operating location l. A counter n is used to count these signals.

6.2 Semantics in PVS

In PVS, we define a timed labelled transition system on top of the untimed LTS defined in Section 4. The global states of this timed model are the global states of the untimed model, where we we assume that valuations interpret local clocks, extended with the value of the global clock. This leads to pairs of the form $\langle s, u \rangle$ where s is a global state of the untimed semantics and u is a non-negative real number representing the global time. As labels of steps, we may use the labels of the untimed semantics or a positive real number (to label a time step).

Then there are two types of steps: "untimed" steps and time passing steps. An "untimed" step

$$\langle s, u \rangle \xrightarrow{l} \langle s', u' \rangle$$

exists if and only if $u = u'$ and $s \xrightarrow{l} s'$ is a step of the untimed LTS. We interpret clock reset actions as local assignments on clocks. Observe that time does not increase during such steps.

A time passing step

$$\langle s, u \rangle \xrightarrow{v} \langle s', u' \rangle$$

for some $v > 0$, exists if and only if the following conditions are satisfied, where $\mathit{shift}_u(s)$ is the same as s except that all local clock values have been increased by u:

1. $s' = \mathit{shift}_v(s)$
2. $u' = u + v$
3. for all $v' < v$, there is no urgent state machine transition enabled in the state $\mathit{shift}_{v'}(s)$.

As before, given this timed LTS, we define the behavior of a system as a set of its infinite execution *runs*. Correctness properties of systems can be expressed in terms of these runs.

A *run* is a sequence of steps of the timed transition system (such that for every position in the sequence the next state is equal to the current state at the following position) which starts in the initial state at time zero. The initial state is the state in which there is a single non-dormant root object; but it can be redefined for a particular application. For example, in the example of Section 6.1, object creation is not modelled, and we would define the initial state as the state with two non-dormant objects, a sender and a receiver, with appropriate values for their attributes, references, and clocks. In particular, the initial value of clock x for the sender object would be u_0, and the initial value of x for the receiver would be 0.

Finally, we further restrict the set of runs to the so-called *non-Zeno* runs in which the progress of time is not limited to a certain bound. We formulate it as follows: a run is non-Zeno if for every position in the run, and for every delay time u, it is possible to proceed to a position where time has increased more than u.

7 Concluding Remarks

We have presented a formal operational semantics of a subset of UML for modelling real-time reactive systems, with a focus on the communication between reactive objects whose behavior is described by state machines. Objects may communicate by means of asynchronous signals or synchronous operations. Threads of control are modelled via active classes, and real-time is added via local clock variables and an urgency predicate on transitions.

By representing the semantics in the specification language of the tool PVS, we detected a number of errors in earlier versions of the semantics. E.g., already the type-checking capabilities of PVS revealed a number of inconsistencies. Although the main ideas about the intended semantics were rather clear, it turned out to be far from trivial to make this precise, and a large number of issues about inheritance, control, primitive and triggered operations, and signals had to be resolved.

We have tried to identify the design choices that had to be made. An important factor in the motivation for our decisions is our objective to use the semantics for theorem proving. This leads to some pragmatic decisions since for theorem proving it is essential that the semantics is as concise as possible.

Moreover, an important result is that we have been able to isolate timing and control sharing as orthogonal features to the semantics. This makes it easy to add or remove these features, depending on the application. Also other features such as deferrable signals, primitive operations, object creation, constructors, object destruction, etc., are relatively easy to add or remove. In this way, we can easily construct a minimal semantics for each application.

We have applied our techniques successfully on a first example (the so-called *Sieve* example, see, e.g., [2]), with an unbounded number of objects that are dynamically created. This example was modelled in UML using the Rhapsody tool [10]. The XMI representation of the model was translated to PVS by our `uml2pvs` tool. Consequently, the example was verified by proving its essential liveness and safety properties in PVS using the strategies of TLPVS [12]. To prove liveness properties, we extended the semantics with a general notion of weak fairness.

Currently, our verification framework is applied to real-time embedded systems provided by the industrial partners within the Omega project. Another topic is the translation of the high-level timing annotations proposed within Omega [6] into our basic timing framework in PVS. We have to investigate which possibility is most suitable for interactive theorem proving. Future work includes the definition of an equivalent denotational, and hence compositional, semantics to enable compositional verification.

Acknowledgement We would like to thank the members of the Omega project for extensive discussions on the semantics issues presented here.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] F. S. de Boer. A proof rule for process creation. In *Proceedings of the third IFIP WG 2.2 Working Conference (Formal Description of Programming Concepts 3)*, 1987.

[3] S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. In *International Workshop, HART'97, Grenoble*, LNCS 1201, pages 286–300. Spinger Verlag, March 1997.

[4] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Proceedings Symposium on Formal Methods for Objects and Components (FMCO 2002)*, pages 71–98. LNCS 2852, Springer-Verlag, 2003.

[5] W. Damm, B. Josko, A. Votintseva, and A. Pnueli. A formal semantics for a UML kernel language. Available via http://www-omega.imag.fr/ Part I of IST/33522/WP1.1/D1.1.2, Omega Deliverable, 2003.

[6] S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In *SDL Forum 2003, July 1-4, Stuttgart*, LNCS, 2003.

[7] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, pages 31–42, 1997.

[8] D. Harel and O. Kupfermann. On the behavioral inheritance of state-based objects. In *Proceedings, 34th Int. Conf. on Component and Object Technology*. IEEE Computer Society, 2000.

[9] J. Hooman and M.B. van der Zwaag. Definition of the semantics in PVS. `http://www.cs.kun.nl/˜mbz/sempvs.html`.

[10] Ilogix. Rhapsody development environment. `http://www.ilogix.com`.

[11] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[12] A. Pnueli and T.Arons. TLPVS: A PVS-based LTL verification system. In *Proceedings of the Intl. Symposium on Verification (Theory and Practice)*, to appear, June 2003.

[13] PVS. Information, documentation, download. Available from SRI Computer Science Laboratory, `http://pvs.csl.sri.com/`.

[14] G. Reggio, E. Astesiano, C. Choppy, and H. Husmann. Analysing UML active classes and associated statecharts - a lightweight formal approach. In *Proceedings FASE 2000 - Fundamental Approaches to Software Engineering*, LNCS 1783, pages 127–146, 2000.

[15] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[16] UPPAAL. `http://www.uppaal.com`.