A Semantics of Communicating Reactive Objects with Timing*

Jozef Hooman and Mark van der Zwaag

Department of Computing Science
University of Nijmegen, The Netherlands
hooman@cs.kun.nl, mbz@cs.kun.nl

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 timed 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. 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 list the decisions taken in the proposed semantics. The resulting semantics is 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 timed 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. In this paper we concentrate on the modelling of reactive systems [HP85] using class diagrams and flat state machines.

We shall refer to the set of UML notations that our semantics covers, as our kernel language. It turned out that in defining a formal semantics for this (rather small) language, a large number of questions arose; these 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. The resulting semantics is defined as a labelled transition system in the typed higher-order logic of the interactive theorem prover

* This work has been supported by EU-project IST 33522 OMEGA "Correct Development of Real-Time Embedded Systems". (See http://www-omega.imag.fr/)

PVS [ORS92,PVS], which led to the identification of a number of further subtleties.

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 via the XMI representation.[1] The aim is further to propose a methodology for the software development process. For all of this work, the formal semantics forms the unifying basis.

Representing the semantics in PVS proved to be a useful means to both a further and rigorous formalization, and in many ways to a higher level of abstraction. Next to that, the PVS representation of the semantics plays an important role in the set of tools that is developed in the Omega project, since it enables formal reasoning and mechanized proof checking on concrete UML models. Current work includes the verification of case studies and the automatic translation of the XMI representation to PVS.

Strongly related to our work is the formalization of active classes and associated state machines [RACH00] by Reggio et al. 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 [HG97]. 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 [HK00] 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.

In Section 2 we discuss the main characteristics of our semantics, and in Section 3 we present the kernel language in which the UML model must be represented. In the subsequent sections we present the semantics. For some key parts we have included the PVS declarations. The complete listing of the PVS theories can be found at `http://www.cs.kun.nl/~mbz/sempvs.html`.

2 Concepts and Decisions

Starting point of the definition of a precise meaning of the Omega kernel language is an early deliverable of the Omega project [DJVP03]. We mention the main

[1] See `http://www.omg.org/technology/documents/modeling\_spec\_catalog.htm` for the latest UML-related specifications.

characteristics and discuss a number of questions that had to be answered to make the semantics more precise and to improve it.

2.1 Kernel Language

Conventional class diagrams are used, with value attributes and reference attributes (which refer to an object), and operations. Associations between classes are represented by reference attributes. Other relations include inheritance and composition (strong aggregation).

The behavior of reactive classes may be specified by hierarchical state machines, which can be flattened as shown in [DJVP03]. Here we assume, for uniformity, that all classes have a flat state machine (possibly empty). A state machine consists of a number of transitions between locations (states). A transitions may have a boolean guard (assumed to be free of side-effects) and can be triggered by a signal or an operation call. There is an action part, here a nonempty list of actions, which is executed when the transition is taken.

2.2 Time

First of all, the time domain is the set of nonnegative real numbers. As in timed automata [AD94], timing constraints are expressed in state machines using local clocks: an object may reset its clocks, and express conditions on clock values in the guard of a state machine transition. Moreover, as in UPPAAL,[2] invariants may be imposed on locations, so that, for example, it can be expressed that an object must leave some location within a certain time. Thus we provide general, low-level constructions for the expression of timing constraints. More high-level syntax for UML with real-time, as developed in the Omega project, can be found in [GO03].

The passing of time is modelled by a global delay action that increases the global time and adjusts all local clocks accordingly. Then, when we define execution traces of the system, we exclude traces that visit states that violate the invariant. We also exclude so-called Zeno traces, that is, traces in which the progress of time is limited to a certain bound.

If a state machine transition is taken by some object, then no time passes between the triggering of the transition (a synchronization with the caller in case of a call event, or the acceptance of a signal in case of a signal event), and the evaluation of the guard, both of which do not take time. With the triggering, the state machine location of the executing object changes from the source to the target location. The execution of an action is an atomic step of the system that does not take time. Passing of time and interleaving with other system activity is allowed before and in between the execution of the actions of the transition. During the execution of the actions the object remains in the target location.

We added the modelling of timing as an orthogonal feature to the untimed semantics: the untimed semantics is restored simply by forbidding the delay actions.

[2] See `http://www.uppaal.com`.

2.3 Activity Groups

A class can be active or passive, leading to active or passive objects at run-time. Each passive object has exactly one corresponding active object.

The main idea is that objects are running asynchronously, which is modelled by interleaving the transitions of all objects. A dynamic assignment of *control* to objects restricts the concurrency of the system; only an object that has control is allowed to execute a state machine transition. The set of objects that are currently alive is partitioned into *activity groups*, that are centered around active objects. Every object has exactly one active parent, and 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. Control changes when performing a call inside the same group, otherwise an object may only loose 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 the term activity group instead.

2.4 Run To Completion

We define a *run-to-completion* semantics, as already defined by the ROOM methodology [SGW94]. That is, the execution following a signal or operation call is continued until a stable state is reached. An object is *stable* if it is ready to execute (i.e., not suspended) and no untriggered transition can be taken, that is, further execution is only possible after receiving a signal or operation call.

2.5 Operation Calls

Operation calls are executed synchronously (i.e. the call action may be performed only if the callee is ready to accept it, and the caller is blocked until the call returns), whereas signal-based communication is asynchronous (the signal is placed in a queue at the receiver; the sender may continue). We consider two types of operations: *triggered* operations, which may occur as a trigger on a transition and are implemented as a part of the callee's state machine (so that it leads to the execution of other actions, possibly including operation calls, and a return action); and *primitive* operations, or *methods*, which are implemented by a piece of code and cannot be used as a trigger. For simplicity, we assume the code does not include operation calls.

In the case of *triggered* operations, we separate the triggering of a transition from its execution, so that a call action of a triggered operation by the caller synchronizes with the triggering of a transition in the callee's state machine. This is then taken to be one step of the system.

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 operation synchronization requires that the caller has control and

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

With the call, the caller becomes suspended.

Execution of a *return* action returns a value (changing an attribute at the caller) and has the effect that the caller is no longer suspended, but it need not lead to a control change: if callee and caller are in the same activity group, control returns to the caller when the callee becomes stable.

We do not allow re-entrance of objects for triggered operations, i.e., when executing a triggered operation it is not possible to accept new calls to triggered operations.

Next consider *primitive* operations; a primitive operation has a method, which is an expression that can be evaluated in a local state of the callee; for simplicity, the resulting value is returned instantaneously at the moment of invocation.

Question is when primitive operations can be executed, e.g. like triggered operations only in stable states of non-suspended objects? Observe that it is quite common that an object may want to call its own primitive operations. But when performing the call the object becomes suspended. Hence it is desirable to be able to call primitive operations of suspended objects. To generalize this, we allow the acceptance of primitive operations in *any* state, i.e., the callee may be unstable or suspended.

A similar problem has been described in [TS03], discussing recursive calls and callbacks. They do not distinguish triggered and primitive operations, but introduce two kinds of state diagrams. The main idea behind our solution is that it is not desirable to model all behavior by state machines; they are only used to model the main reactive behavior, i.e. the interaction between objects, and other operations are more conveniently expressed in some action language (or specified more abstractly using OCL).

2.6 Object Creation

An object can create a new object to which it will be able to refer. We take a highly simplified view: the initial attribute values of the new object are completely determined by its class—not by its creator; we do not model entry scripts, and there is no recursion (as may be used to model aggregation).

2.7 Generalization

The usual questions about inheritance apply here (see, for instance [HK00]). The main point is to which extent behavior of a super-class is inherited by a subclass. Conforming to the current use of inheritance in industrial applications, we allow that a subclass redefines the behavior of an inherited operation. Then the question remains whether the definition of a triggered operation is inherited by a subclass when it does not (re)define it. 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 super-class.

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 preprocessing. We also record for each object the corresponding class, thus obtaining conventional polymorphism.

3 Kernel Language

First, the types *Attribute*, *Class*, *Clock*, *Location*, *Method*, *Operation*, *Reference*, *SignalName*, and *Value* are parameters of the PVS theories for the kernel language and the semantics: the UML model under study is required to supply instantiations for these (abstract) types.

The data expressions used in the action language and in method bodies, and the guards of state machine transitions are modelled as mapping from valuations to an interpretation, see Figure 1.

A *valuation* offers an interpretation for expressions: it consists of a mapping of attributes to data values, a mapping of clocks to time elements, and a mapping of references to objects. A valuation represents a local state of an object.

Let *Object* (object identifiers) be an uninterpreted type, and let the type *Time* be the set of nonnegative real numbers.

```
AttributeValuation: TYPE+ = [Attribute -> Value]
ReferenceValuation: TYPE+ = [Reference -> Object]
    ClockValuation: TYPE+ = [Clock -> Time]

         Valuation: TYPE+ = [# aval: AttributeValuation,
                               rval: ReferenceValuation,
                               cval: ClockValuation #]

        Expression: TYPE+ = [Valuation -> Value]
             Guard: TYPE+ = [Valuation -> bool]
```

Fig. 1. Valuations and Expressions

3.1 State Machine Transitions

For state machine actions we distinguish the following forms.

- *call*(*a*, *ref*, *op*, *exp*): A call of the operation *op*, with as parameter the value of the data expression *exp*, to the object referred to by *ref*. When a return value is received, it is assigned to the attribute *a*.
- *return*(*result*): Return the value of the *result* expression.
- *emitSignal*(*ref*, *signame*, *exp*): The emitting of a signal named *signame*, with as parameter the value of the data expression *exp*, to the object referred to by *ref*.
- *assign*(*a*, *exp*): A local assignment action assigns a the value of the data expression *exp* to the attribute *a*.
- *methodCall*(*a*, *ref*, *meth*): A call to the object referred to by *ref* of the method *meth*. The result is assigned to the attribute *a*.
- *create*(*ref*, *c*): Create an object of class *c*, and use *ref* as a reference to this new object.
- *skip*: Do nothing.
- *reset*(x): Reset the clock x to zero.

A trigger event is of one of the following forms:

- *callEvent*(*op*, *a*): For the triggering by a call of the operation *op*. The parameter of the call is assigned to the attribute *a*.
- *signalEvent*(*signame*, *a*): For the triggering by a signal *signame*. The parameter of the signal is assigned to the attribute *a*.

The value *none* is used to model untriggered transitions.

A state machine transition is defined as a record with fields for its *source* and *target* location, *guard*, *trigger*, and *actions*. Also, a transition has a field for (the state machine of) the *class* that it belongs to. The *actions* field contains a nonempty list of state machine actions.

3.2 Input From The Model

Besides the instantiations for the basic types, the UML model under study is required to provide the following input to the semantics.

- A function *method* of type $\mathit{Method} \times \mathit{Class} \to \mathit{Expression}$ that gives the method body of primitive operations as a data expression (recall that we do not allow operation calls in method bodies; the return value must be computed locally by the callee).
- A set $\mathit{transitions} \subseteq \mathit{Transition}$ containing all transitions of the state machines for the classes of the model. A transition is linked to its state machine by its class field.
- A predicate *active* on classes that defines which classes are active.
- The root class *rootClass*, which must be active.

- A function *initialLocation* of type $Class \rightarrow Location$ that assigns an initial state machine location to classes (exactly one for each class).
- A function *initialAttrVal* of type $Class \rightarrow AttributeValuation$ that assigns initial values to attributes.
- An invariant $inv \subseteq Class \times Location \times Valuation$.

4 States

First, we distinguish four values for the *status* of an object.

- An object is *dormant* until it is created.
- An object is *free* if it is not processing a triggered operation call.
- The object status is *processingCall*(*caller*, *a*) if the object has accepted a call from the object *caller*, for which it has not returned the result yet. The result value must be assigned to the attribute *a* of the caller.
- Finally, the status is *completingCall*(*caller*) if the caller is from the same activity group and the object has, having executed the return action, to *complete* the call, i.e., run to completion/become stable, before it returns control to the caller.

An *object state* contains all relevant local informant concerning an object; it is a record with the following fields.

- A valuation *val* that interprets expressions, see Section 3.
- The *class* that the object belongs to.
- The *status* of the object.
- A list *alist* of state machine actions that are to be executed by the object.
- A boolean *suspended* indicating whether the object is suspended.
- The object's current state machine location *loc*.
- The object's signal queue *sq*.
- The active object *aobj* that is the leader of the object's activity group.
- The object identity *control* of the control object of the object's activity group. This value is used only if the object is active.

A (global) *state* is a record with the following two fields (that are sometimes called system variables): a mapping F of type $Object \rightarrow ObjectState$ that provides the current state of objects, and the global time *time*.

We assume that all system behavior starts in the initial state, in which there is a single nondormant root object, see the definition of *runs* in Section 6.

5 Labelled Transition System

We define a labelled transition system (LTS) for the input model. The states of this LTS are as defined above. In Figure 2 we give the definition of the global

transition relation *steps*. At this abstract level the definition can be presented in a single rule:

$$\frac{condition(s, l)}{s \xrightarrow{l} effect(s, l)}$$

The condition (boolean) and effect (yielding a state) functions are defined by case distinction on the label l of the step. A label is the visible part of the execution of either (1) a state machine action, or (2) the triggering of a transition (either untriggered, signal-triggered, or call-triggered), or (3) a global time delay. If at some point it is desirable that certain information is not visible in the label, then we can abstract from that information (by renaming).

Note. Only the passing of time is not linked to a certain executing object.

5.1 Preliminaries

Some auxiliary definitions concerning stability and control.

- An object is *alive*, if it is not dormant.
- An object is *ready*, if it is alive, has an empty action list, and is not suspended.
- An object is *executing*, if it is alive, has a nonempty action list, and is not suspended.
- The control object of object p in state s, notation $co(s, p)$ is the value of the control field of its active parent in state s.
- An object is *in control* (in some state), if it is equal to its control object.
- A transition is *locally enabled* for an object in some state, if it belongs to the objects state machine, its source is also the object's current location, and the boolean guard is true.
- An object is *stable* in some state, if it is *ready*, it is not currently processing a call, and all locally enabled transitions have a trigger (other than *none*).
- Two objects belong to the *same activity group* if they have the same active parent.

Passing of Control The passing of control within an activity group is modelled as a side-effect of the other steps that the system may perform: note that there is no label defined for it.

If an object is to start executing, then it must either have or be able to obtain the control within its activity group. Defined in Figure 3 are the condition and effect functions for the passing of control to an object p in global state s. It may be that the object p already has control, in which case the resulting state is equal to s; except in the particular case that p has reached a stable of after completing a call for a group member. In this case the control may be returned to the caller. However, if the caller happens to be stable, then the control may be passed back to (or, effectively remain with) p.

If an other object is in control, then that object must be stable. If the control object has just finished a call, then either p is the caller, or another (third) object is the caller, which then must be stable as well.

```
Label: DATATYPE
 BEGIN
  action(action: Action, actor: Object): action?
  localTriggering(t: (transitions), actor: Object): localTriggering?
  acceptSignal(t: (transitions), n: nat, actor: Object): acceptSignal?
  acceptCall(t: (transitions), caller, actor: Object): acceptCall?
  delay(dt: DelayTime): delay?
 END Label

l: VAR Label

condition(s, l): bool =
  CASES l
   OF
    action(act, p): actionCondition(act, s, p),
    localTriggering(t, p): localTriggeringCondition(t, s, p),
    acceptCall(t, caller, p): acceptCallCondition(t, caller, s, p),
    acceptSignal(t, n, p): acceptSignalCondition(t, n, s, p),
    delay(dt): true
  ENDCASES

effect(s, l): State =
  CASES l
   OF
    action(act, p): actionEffect(act, s, p),
    localTriggering(t, p): localTriggeringEffect(t, s, p),
    acceptCall(t, caller, p): acceptCallEffect(t, caller, s, p),
    acceptSignal(t, n, p): acceptSignalEffect(t, n, s, p),
    delay(dt): delayEffect(dt, s)
  ENDCASES

Step: TYPE+ = [# current, next: State, label : Label #]

steps: setof[Step] = LAMBDA (step: Step):
  condition(step`current, step`label) AND
   step`next = effect(step`current, step`label)
```

Fig. 2. The global transition relation

```
s: VAR State; p: VAR Object

takeControlCondition(s, p): bool =
  IF inControl?(s, p)
  THEN (stable?(s‘F(p)) AND completingCall?(s‘F(p)‘status))
        IMPLIES stable?(s‘F(caller(s‘F(p)‘status)))
  ELSE LET co = co(s, p) IN
      (stable?(s‘F(co))
    AND
      (completingCall?(s‘F(co)‘status) IMPLIES
           ( p = caller(s‘F(co)‘status)
            OR
             stable?(s‘F(caller(s‘F(co)‘status))))))
   ENDIF

takeControlEffect(s, p): State =
   IF inControl?(s, p)
   THEN
     (IF stable?(s‘F(p)) AND completingCall?(s‘F(p)‘status)
      THEN s WITH [(F)(caller(s‘F(p)‘status))(suspended):= FALSE]
       ELSE s ENDIF)
   ELSE LET co = co(s, p) IN
    (IF stable?(s‘F(co)) AND completingCall?(s‘F(co)‘status)
      THEN s WITH [(F)(caller(s‘F(co)‘status))(suspended):= FALSE,
                   (F)(co)(status):= free]
      ELSE s ENDIF) WITH [(F)(s‘F(p)‘aobj)(control):= p]
   ENDIF
```

Fig. 3. Passing of Control

5.2 Semantics of Action Execution

We describe how the execution of a state machine action changes a state. The action is required to be the first element of the executing object's action list; it is removed from that list after execution.

return(*exp*) An executing object that is processing a call for some *caller* may execute a return action with result expression *exp*. It is required that the caller is alive. The result is that the caller becomes un-suspended, that the value of the result expression is assigned to the designated attribute for the caller. If the caller belong to the same group, then the callee becomes *free*, otherwise its status becomes *completingCall*: it must then run to completion (become stable), after which it returns the control to the caller.

assign(*a*, *exp*) The executing object assigns the current value of the expression *exp* to its attribute *a*.

emitSignal(*ref*, *sn*, *exp*) A new signal with signal name *sn* and the value of the expression *exp* is inserted in the signal queue of the receiving object.

methodCall(*a*, *ref*, *meth*) The result of a call of a method (primitive operation) is returned instantaneously as the value of the operation's method in the local state of the callee (the object that is referred to by *ref*). The result is assigned to the attribute *a* at the caller. The callee must be alive (non-dormant).

skip Nothing changes.

create(*ref*, *c*) The executing object can create a new object of class *c*, for which it will have the reference *ref*, by initializing an (arbitrary) dormant object. The new object is initialized as follows: its class is *c*; it is free and not suspended; its signal queue and action list are empty; it has the initial valuation and state machine location and valuation associated with its class; it is its own active parent if *c* is an active class, and otherwise it inherits its creator's active parent.

reset(*x*) Resetting a clock *x* means changing the value of *x* to zero for the executing object (like a local assignment).

Note. The semantics of the *call* action is given below; it is part of the synchronization with the callee object.

5.3 Triggering of a State Machine Transition

The semantics of the triggering of a state machine transition for an object *p* depends on the kind of its trigger.

Untriggered A transition can be triggered *locally* for an object, if it is untriggered. It is required that the object is *ready* (i.e., is alive, has an empty action list, and is not suspended) and can *take control*, and that the untriggered transition is locally enabled. The result is that the transition is taken (the target location becomes the current location of the object, and the action list of the transition is copied in the object's action list field) and that the object obtains control.

Call Event The triggering of a transition with a call event trigger involves a synchronization with the caller of the operation: it is required that there is a caller object that is *executing* and whose first action is a call to the object p that matches the trigger event on the transition.

```
acceptCallCondition(t, caller, s, p): bool =
   stable?(s‘F(p)) AND executing?(s‘F(caller))
  AND
   ((NOT samegroup?(s‘F(p), s‘F(caller)))
           IMPLIES takeControlCondition(s, p))
  AND
   LET action = car(s‘F(caller)‘alist) IN
    call?(action)
  AND
   callTriggersTransition?(action, t, s, caller, p)

acceptCallEffect(t, caller, s, p): State =
  IF executing?(s‘F(caller)) AND
       call?(car(s‘F(caller)‘alist)) AND
        callEvent?(t‘trigger)
  THEN
   LET action = car(s‘F(caller)‘alist) IN
    takeControlEffect(basicTriggeringEffect(t, s, p), p)
      WITH
       [(F)(p)(status):= processingCall(caller, a(action)),
        (F)(p)(val)(aval)(a(t‘trigger)):= exp(action)(s‘F(caller)‘val),
        (F)(caller)(alist):= cdr(s‘F(caller)‘alist),
        (F)(caller)(suspended):= TRUE ]
  ELSE s ENDIF
```

Fig. 4. Acceptance of a call.

Furthermore, p is required to be stable, and if the caller belongs to another group, then p must be able to take control. Finally, the transition must be locally enabled for p (after the assignment of the value of the parameter of the call to the attribute specified in the trigger expression). See the definition in Figure 4.

Result of the synchronization is that the callee is triggered (gets control, changes location to the target of transition, and copies the action list of the transition); the caller becomes suspended (and it looses control if the callee belongs to the same activity group); the status of the callee becomes *processingCall*(*caller*, a), where a is the attribute the result must be assigned to; and the value of the parameter of the call is assigned to the designated attribute.

Signal Event In Figure 5 we define how a transition t is triggered for object p by a signal. Let n be the position of the triggering signal in p's signal queue. It is required that this signal is the first element of the queue that triggers a transition. We further require that the object is stable and that it can take control.

The result is that the transition is triggered, and that the signal queue is *cleaned up*: we remove the triggering signal and all the preceding signals that are not *deferrable* in the current location.

```
t: VAR (transitions); n: VAR nat; s: VAR State; p: VAR Object

acceptSignalCondition(t, n, s, p): bool = LET os = s`F(p) IN
   stable?(os) AND takeControlCondition(s, p)
  AND
   nonempty?(os`sq)
  AND n < length(os`sq)
  AND
   signalTriggersTransition?(nth(os`sq, n), t, os)
  AND
   FORALL (m: below[n]): NOT EXISTS (t1: (transitions)):
    signalTriggersTransition?(nth(os`sq, m), t1, os)

acceptSignalEffect(t, n, s, p): State =
  IF nonempty?(s`F(p)`sq) AND signalEvent?(t`trigger)
   THEN takeControlEffect(basicTriggeringEffect(t, s, p), p)
      WITH [(F)(p)(sq):=
              cleanUp(s`F(p)`sq, n, s`F(p)`class, s`F(p)`loc)]
  ELSE s ENDIF
```

Fig. 5. Acceptance of a signal.

5.4 Passing of Time

Delaying with time dt (with $dt > 0$) means that the delay time is added to the global time and that all local clocks are delayed accordingly. PVS definition in Figure 6.

6 Execution Traces

Above we defined a labelled transition system for the input model. For verification purposes, we define the behavior of a system as a set of infinite sequences

```
u: VAR Time; dt: VAR DelayTime

delayClocks(val: Valuation, u): Valuation =
   val WITH [(cval):=  LAMBDA (x: Clock): val‘cval(x) + u]

delayEffect(dt, s): State =
  s WITH [(time):= s‘time + dt,
          (F):= LAMBDA p:
            s‘F(p) WITH [(val):= delayClocks(s‘F(p)‘val, dt)]]
```

Fig. 6. Passing of time.

of steps that satisfy some further requirements. Such sequences

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \cdots$$

of steps are called execution traces, or *runs*, of the system. Correctness properties of systems can be expressed in terms of these runs.

First, we define a run as any sequence of *steps*, see Figure 7. Then, we define the *runs* of the system as a set of these runs by imposing a number of restrictions.

- A run is *non-Zeno* if for every state of the run, and for every delay time u, it is possible to proceed to a state taking more than u time.
- The invariant *inv* is satisfied for some run if it holds for all its states, *but also* for all intermediate states that are reached by the passing of time (the delay time of non-delay actions is defined as zero).
  The invariant *inv* is satisfied for some global state s and some delay time u, if the invariant holds for all objects that are *ready* in s, but only after a time delay of u.
- The first state of a run must be the *initial state*: in this state the global time is 0, and the object *root* is the only object alive. This *root* object belongs to the root class, is free, not suspended, in control, and in the initial location of its state machine; its signal queue and actions list are empty; and it has the initial valuation associated with its class.

The set *runs* is defined as the set of runs that satisfy these criteria; see Figure 7.

Note. Execution traces that lead to a time deadlock (a state where no further actions, including *delay*, can be performed without violating the invariant) are excluded from the set of runs. However, we may be interested in such finite behavior. A trick to define infinite runs for finite executions, is to extend the *steps* relation as follows: in the case of time deadlock an *escape* action may be performed, to an *empty* state, that is, a state where all objects are dormant.

```
Run: TYPE = sequence[(steps)]

i, j: VAR nat; u: VAR Time; s: VAR State; r: VAR Run; l: VAR Label

nonZeno(r): bool = FORALL i, u: EXISTS j:
  r(i+j)‘current‘time > r(i)‘current‘time + u

delayTime(l): Time =
 CASES label OF delay(u): u ELSE 0 ENDCASES

invOK(s, u): bool =
  FORALL p: ready?(s‘F(p)) IMPLIES
   inv(s‘F(p)‘class, s‘F(p)‘loc, delayClocks(s‘F(p)‘val, u))

invOK(r): bool =
  FORALL i, u: u <= delayTime(r(i)‘label)
   IMPLIES
    invOK(r(i)‘current, u)

runs: setof[Run] = LAMBDA r: (FORALL i: r(i)‘next = r(i+1)‘current)
 AND
  r(0)‘current = initialState AND nonZeno(r) AND invOK(r)
```

Fig. 7. Definition of runs.

From such a state the only further activity is the passing of time, while the invariant is always (trivially) satisfied. Thus an empty state models a delayable deadlock.

7 Concluding Remarks

We have presented a formal operational semantics for a core UML language for modelling real-time reactive systems, with a focus on the communication between reactive objects whose behavior is described by state machines. Objects belong to an activity group and may communicate by means of asynchronous signals or synchronous operations. 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.

Moreover, by representing the semantics in the specification language of the tool PVS, we detected a number of errors in earlier versions of the semantics that were described on paper only. E.g., already the type-checking capabilities of PVS revealed a number of inconsistencies.

In current work we are experimenting with the interactive theorem proving capabilities of PVS to verify reactive systems. We are also developing a first prototype of a tool that translates XMI output of UML-based CASE tools to PVS. To enable compositional verification, we also intend to define an equivalent denotational, and hence compositional, semantics. Other work concerns the extension of the current semantics with signal priorities, exceptions, and interrupts.

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

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[DJVP03] 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.

[GO03] S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In *Proceedings of SDL 2003 Forum*, LNCS, 2003.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, pages 31–42, 1997.

[HK00] 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.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[ORS92] 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.

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

[RACH00] 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.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[TS03] J. Tenzer and P. Stevens. Modelling recursive calls with UML state diagrams. In *Proc. FASE 2003 - Fundamental Approaches to Software Engineering*, pages 135–149. LNCS 2621, Springer-Verlag, 2003.