International Journal on Software Tools for Technology (2006) 8(2): 97–112
DOI 10.1007/s10009-005-0207-8

SPECIAL SECTION ON SPECIFICATION AND VALIDATION OF MODELS OF REAL TIME AND EMBEDDED SYSTEMS WITH UML

Jozef Hooman · Mark B. van der Zwaag

A semantics of communicating reactive objects with timing

Published online: 23 September 2005

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, i.e., 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, list a number of questions that arose, and discuss the decisions taken. The resulting semantics has been defined in the typed logic of the interactive theorem prover PVS, thus enabling formal verification based on this semantics.

Keywords Formal semantics · UML · Real-time · Formal methods · Theorem proving

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 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.

Sources for the formal Omega semantics are the UML standard [21] (version 1.3 at the start of the project, version 2.0 at the moment of writing) and the execution mechanism of UML-based CASE tools from the real-time domain, such as Rhapsody [14] and Rose RealTime [12]. These CASE tools implicitly define a UML semantics by generating code. Starting point in Omega was an operational semantics [6], which was especially inspired by the execution mechanism of Rhapsody. This semantics is also closely related to the implementation of untimed model checking by project partner OFFIS [5]. The semantics has also been implemented in Verimag's IF tool [13] which additionally allows timed model checking. Although model-checking is attractive, due to the minimal amount of required user interactions, it imposes rather strong limitations on the models. Hence, in the Omega project we also consider complementary techniques such as theorem proving that require a large amount of user interaction but have less restrictions on the models that can be verified.

The aim of this paper is to define an abstract semantics which is suitable for interactive theorem proving and which clarifies a number of semantic questions and decisions. Based on previous experience and own expertise, we decided to define the semantics in the typed higher-order logic

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/.

J. Hooman (✉)
Embedded Systems Institute, Eindhoven, The Netherlands
E-mail: hooman@cs.ru.nl

J. Hooman · M. B. van der Zwaag
Department of Computing Science, Radboud University Nijmegen, The Netherlands
E-mail: mbz@cs.ru.nl

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

of the interactive theorem-prover PVS [16, 18]. The rigorous formalization in PVS by itself revealed a number of ambiguities in earlier versions of the Omega semantics. For instance, questions arose concerning the passing of control, the dispatching of signal events, and the synchronization of operation calls. In this paper, we identify these issues and present the decisions taken to resolve them. Moreover, we show how a basic untimed semantics can be extended with a continuous notion of time in an orthogonal way. Also the addition of threads of control can be done in a rather modular way.

1.1 Related work

Strongly related to our work is the formalization of active classes and associated state machines by Reggio et al. [19]. They define a labeled 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 [9]. 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 [10] 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. For example, in [5] there is only a notion of a discrete step. Our timing extension is based on classical timed automata [2], but unlike e.g. Uppaal [22], we do not use invariants but an urgency predicate which is a restricted version of the timed automata with deadlines of [4]. A more high-level syntax for UML with real-time annotations, as developed in the Omega project, can be found in [8]. They can be translated into our basic timing framework.

1.2 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 enables formal reasoning and mechanized proof checking on concrete UML models. The idea is depicted in Fig. 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 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. This semantics defines the meaning of an UML model as a set of runs, denoting the snapshots of the system configuration during execution.

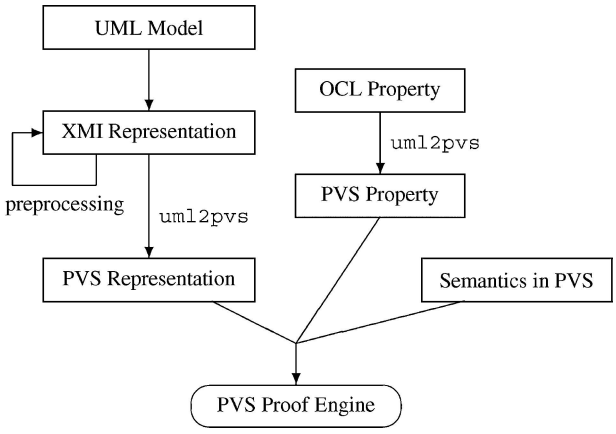


Fig. 1 Verification of UML models in PVS

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 can be expressed in PVS, including safety and liveness properties. By means of the interactive proof checker of PVS it can be proved 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 re-occurring patterns in the proofs. Within Omega, the TLPVS package is used to experiment with powerful strategies for the proof of temporal properties [17]. The results of this work are reported elsewhere [3].

1.3 Overview

In our semantics, time is modeled as an orthogonal aspect, and it turned out that especially for the untimed part there were a number of questions 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.

In Sect. 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 Sect. 3. Section 4 shows how the untimed kernel language has been represented in PVS. All PVS theories are available online [11]. Section 5 explains in detail how the formal

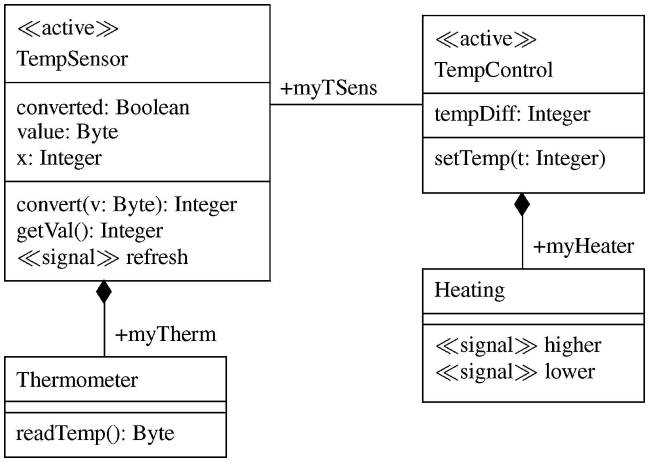


Fig. 2 Example class diagram

semantics is defined in PVS. The introduction of real-time can be found in Sect. 6, where the untimed semantics is extended to deal with timed systems. A notion of threads of control, which is realized using the concept of so-called activity groups, is added in Sect. 7. Concluding remarks can be found in Sect. 8.

2 Kernel language

In the Omega kernel language, classes are declared by conventional class diagrams with attributes, operations, and signals. Consider, for example, the class diagram in Fig. 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 external signal event *setTemp*(*t*) which is a request to keep the temperature at level *t*. The heater responds to the signal events *higher* and *lower*. The sensor responds to signal event *refresh* by reading the temperature value from its thermometer. In general, emitted signals are stored as signal events into the event pool of the receiving object, where they are dispatched and possibly trigger transitions (this is discussed in more detail later).

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, here called *activity group* (discussed in Sect. 7). 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 a part of a controller. Typically, the passive parts of an active object belong to the same activity group. So in the example of Fig. 2, each (passive) *Thermometer* object is in the same activity group as an (active) *TempSensor* object.

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 are called *triggered operations* because they can trigger a transition. Moreover, the state machine specifies the response to signal events.

In our formalization, state machines are assumed to be flat, that is, to consist of transitions between states; 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, see also [5]. 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 state and no transitions.

A state machine transition is written as

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

where s is the source and s' is the target state, 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 state s may reach state s' 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 event. 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 either a list of primitive actions or a call to triggered operation; the special treatment of triggered operation calls will be explained in Sect. 3. The list may be empty, denoting skip. For the untimed part of UML, we consider the following five primitive actions:

Assignment: $a := exp$
Assign the current value of expression exp to attribute a.

Object Creation: $r := new\ c$
Create an object of class c and let reference r refer to it.

Signal Emission: $r!sig\ (exp)$
Insert a signal event with name *sig* and the value of expression exp in the event pool of the object referred to by r.

Primitive Operation Call: $a := r.m$
Call method (primitive operation) m of the object to which r refers (the callee) and store the result in attribute a of the caller.

Return: *return exp*
Return the value of expression *exp* to the caller (if any, otherwise skip).

Finally, we have the nonprimitive call of a triggered operation:

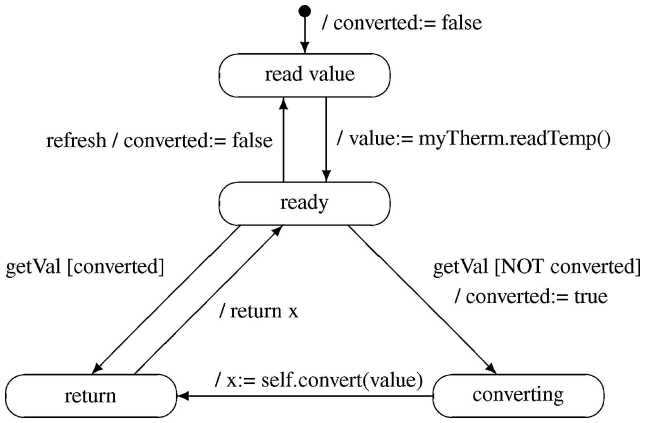


Fig. 3 State machine for the *TempSensor* class

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

Note that we focus on the fundamental concepts of the semantics and only consider one type of operations with one input parameter and a result value.

As an example, Fig. 3 shows the state machine for class *TempSensor* of Fig. 2. The incoming arrow with the black circle at the top indicates the initial state; 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 state *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 event is put in an event pool at the receiver side. Signals events are selected from this pool and then either trigger a transition or are discarded if they do not trigger a transition in the current state. An exception are signals that are declared to be *deferrable* in the current state; they are not discarded and remain in the event pool.

Although not shown in Fig. 2, the kernel language also allows the generalization relation between classes. This leads to the usual questions concerning inheritance (see, for instance [10]). 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.

Finally, we mention that the execution mechanism of UML is based on the *run-to-completion* paradigm, as already defined by the ROOM methodology [20], that is, an object may only execute transitions that are triggered by an operation call or a signal if it is stable. An object is *stable* if, in its current state, there are no outgoing untriggered transitions for which the guard is satisfied. Thus a stable object can only proceed by accepting a call or a signal event. For instance, in Fig. 3 only state *ready* is stable. Observe that the run-to-completion mechanism reduces the number of interleavings and the amount of interference between objects.

3 Semantic choices

Our first attempt to formalize the informal meaning of UML diagrams, as sketched in the previous section, revealed a number of questions to be answered and, consequently, a number of decisions had to be taken. The first question concerns the treatment of triggered operations.

> Should a call to a triggered operation lead to a call event in the event pool, or should it be put in a separate pool or table, or should there be a more direct synchronization between caller and callee?

In the original Omega semantics, as presented in [5], triggered operation calls are put into a separate pending request table. This approach turns out to be convenient for model checking components relative to assumptions about its environment, but it is somewhat biased towards the implementation of the model checker.

In this paper we take a more abstract view and use explicit synchronization between caller and callee. That is, the caller has to wait until the callee is ready to accept the call (the caller must be stable and have a transition triggered by the operation in its current state). Such a direct synchronous semantics for operations is also proposed in [9], with the argument that it allows a close control of sequencing and the possibility to model tight object synchronization. It also avoids the overhead of the event pool and can be efficiently implemented in an object-oriented programming language.

An alternative, which seems more in line with the current UML version, is to put the call events also in the event pool. This, however, might easily lead to callers that are blocked forever when such events are discarded. Additional verification effort is needed to show that such situations do not occur in a particular design.

The next decision concerns the concurrency model; the concurrent execution of the objects is modeled by interleaving the steps of each object. An execution of a set of

concurrent objects is represented by a sequence of configurations, where each configuration models the state of affairs at a point during execution and each pair of successive configurations 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, 6] there are small "bookkeeping" steps, e.g., to discard signals from the event pool. We also experimented with several possibilities, e.g., a version in which each action of the action list on a transition forms a step and also discarding an event from the event pool is a separate 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. The only exception is a call of a triggered operation, when 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.

Observe that the enabledness of a transition with a list of primitive actions depends only on the trigger event and the guard, whereas an action part that consists of a triggered operation call adds additional constraints to the enabledness condition.

Next we discuss the treatment of signal events, which corresponds to an explicit variation point in the description of UML.

How are signal events stored, selected, and discarded?

Questions are, for instance, whether we have a separate input event pool for each object or for a set of objects, how to select events from this pool, how to put deferrable signals that do not trigger a transition back into the pool, etc.

To obtain relatively simple, predictable behavior, we made the following decisions. Each object has its own input event pool which acts as a first-in-first-out queue (henceforth, the pool is called "event queue"). An emitted signal is placed into this queue at the receiver. When an object is stable, the first event in the queue which triggers a transition may be selected, removed from the queue and—in the same step—the triggered transition is executed. Moreover, during this step all preceding events in the queue (which do not trigger a transition in the current configuration) are discarded, except if they have been made *deferrable* for the current state. The deferrable preceding events maintain their order in the queue. This is formalized in Sect. 5.6 and compared with [15].

Finally, we list three other semantic questions which required a decision.

Is it possible for an object to read or write attributes of another object?

To reduce the amount of interference between objects, any sharing of attributes is disallowed. Attributes are encapsulated in objects and can be modified externally only via operations.

Is it possible for an object to accept a new operation call when it is still busy answering a preceding call?

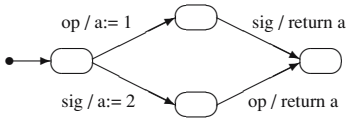
In the Omega semantics, re-entrance is not allowed. This is motivated by the aim to reduce the amount of interference between objects and to keep the semantics as simple as possible. Hence, an object can only accept a triggered operation call if it is stable (the run-to-completion assumption mentioned in the previous section) and if it is not already been 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. Note that when the callee accepts the call, the caller becomes suspended (blocked) and is not able to perform any action. It remains suspended until the reception of the result value.

When can an object accept primitive operations?

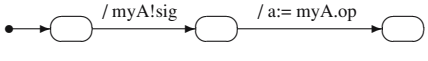
Primitive operations (also called methods) are implemented by a piece of code and cannot be used as a trigger. For simplicity, we require here that the method body does not contain operation calls. Moreover, we assume that the result can be computed atomically and the complete execution, including the method call, the computation, and the return of the result, is modeled as one step in the semantics. Such primitive operations are often useful to compute some piece of internal functionality of an object during a more complex computation, e.g., while answering a call. So, often one would like to call own primitive operations of an object. Hence, we allow the acceptance of primitive operations in *any* state, so in particular also when the object is unstable or processing a triggered operation.

3.1 Example

We present a small example to illustrate some consequences of the decisions described above. Consider an object of class *A* with the following state machine:

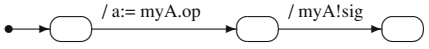


An object of class *B* has the following state machine:



where *myA* refers to the object of class *A*. The concurrent behavior of these two objects has two traces ending in the final configuration, i.e., the configuration where both objects are in the rightmost state. 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 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 Kernel language in PVS

We show how the syntax of the basic Omega kernel language has been implemented in PVS. In general, PVS specifications are expressed in a typed higher-order logic, and they can be organized into parameterized theories. To define the syntax, a theory `KL` has been defined with eight parameters that represent types for attributes, classes, states, methods, object identifiers, operations, references, signals, and a value domain, respectively. The + behind `TYPE` denotes that the type is not empty.

```
KL[Att, Class, State, Meth,
   Obj, Op, Ref, Sig, Dom: TYPE+]: THEORY
BEGIN
```

It is possible in PVS to define an explicit syntax for expressions and boolean guards and later, in the semantics, define an interpretation using a valuation which assigns values to attributes and object identifiers to references. This, however, is not very convenient to use and would distract the attention from the main topic of this paper. Hence, we already define a valuation here by type `Val`, which is a record (denoted by `[#` and `#]`) with two fields: function `AVal` which maps attributes to values and `RVal` which maps references to object identifiers. Such a valuation is part of the global snapshot of the system and will change in time. Expressions and guards are then simply functions from a valuation to values and booleans, respectively.

```
AVal: TYPE+ = [Att -> Dom]
RVal: TYPE+ = [Ref -> Obj]
Val: TYPE+  =
   [# aval: AVal, rval: RVal #]

Exp : TYPE+ = [Val -> Dom]
Guard: TYPE+= [Val -> bool]
```

Next we define the syntax of the actions. Type `PrimAct` contains five primitive actions. For instance, signal emission *r*!*sig*(*exp*) is represented by `emitSignal(r,sig,exp)`. The `DATATYPE` construct implies a large number of implicit properties for the type, e.g., it ensures that all actions are different, that there are no other primitive actions, and it also provides mechanisms for induction and recursion on this type. It also defines so-called recognizers `assign?`, `create?`, etc., that can be used to check if a particular action is an assignment, create action, etc.

In a similar way, general actions on transitions are defined by type `Act`, consisting of a call or a list of primitive actions (type `list` is predefined in PVS). `skip` is introduced as an abbreviation for the empty list.

```
PrimAct : DATATYPE
 BEGIN
  assign(a: Att, exp: Exp): assign?
  create(ref: Ref, class: Class): create?
  emitSignal(ref: Ref,
             signame: Sig,
             exp: Exp): emitSignal?
  meth(a: Att, ref: Ref, meth: Meth): meth?
  return(exp: Exp): return?
 END PrimAct

Act: DATATYPE
 BEGIN
  call(a: Att,
       ref: Ref, op: Op, exp: Exp): call?
  alist(alist: list[PrimAct]): alist?
 END Act

skip: Act = alist(null)
```

To define transitions, we first define three types of triggers (no triggering call or signal, a signal trigger, or a call trigger). The basic type of a transition is a record with six fields. Note that we also record the class to which a transition belongs. An alternative is to have a separate function which assigns a set of transitions to each class; then—without further restrictions—a transition may belong to several classes or to no class at all. Finally, type `Trans` of transitions requires that a transition with a call trigger should not have a call in the action part. Note that `t`trigger` selects the `trigger` field of transition `t` (an alternative notation would be `trigger(t)` and recognizer `callTrig?` checks the subtype of the trigger.

```
Trigger: DATATYPE
 BEGIN
  noTrig: noTrig?
  sigTrig(signame: Sig, param: Att): sigTrig?
  callTrig(op: Op, param: Att): callTrig?
 END Trigger

BasicTrans: TYPE =
 [# source: State,
    trigger: Trigger,
    guard: Guard,
    action: Act,
    target: State,
    class: Class      #]

Trans: TYPE =
  {t: BasicTrans | callTrig?(t`trigger)
                   IMPLIES NOT call?(t`action)}
END KL
```

5 Basic semantics in PVS

We present the basic semantics of the kernel language and how it is expressed in the language of PVS. Timing and activity groups are not included; this will be added in Sects. 6 and 7, respectively. Also deferred events are not included in the basic version, in Sect. 5.6 it is shown how they can be added.

Given a concrete UML model, the semantics is defined in a general PVS theory `Semantics`, where the essential characteristics of the concrete model are represented by a number of parameters. First of all, there are the same nine parameters as the kernel language theory `KL` and this theory is imported (this is needed because subsequent parameters defined in the definitions in `KL`). Next we have five parameters: `class` to express for each object to which class it belongs, `initVal` and `initState` to define the initial valuation and state for each class, `trans` to represent the set of concrete transitions, and `meth` which defines for each method name and class the result by means of an expression. After having defined the parameters, we first define parameterized signals as a record with a signal name and a value from the value domain.

```
Semantics[
  Att, Class, State, Meth, Obj, Op,
  Ref, Sig, Dom: TYPE+,
  (IMPORTING KL[Att, Class, State, Meth,
                Obj, Op, Ref, Sig, Dom])
  class: [Obj -> Class],
  initVal: [Class -> Val],
  initState : [Class -> State],
  trans: setof[Trans],
  meth: [Meth, Class -> Exp] ] : THEORY
BEGIN

Signal: TYPE = [# signame: Sig, d: Dom #]
```

The general idea is that an execution of the UML model is represented by a *run* (i.e., an execution trace) which is a sequence of the form $c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow \cdots$ where the c_i are *configurations*, representing a snapshot of the system during execution and each pair of successive configurations represents a step of a state machine of one of the objects.

The configurations are defined in Sect. 5.1, including the definition of stability. The semantics of primitive actions, triggered operation calls, and signal triggers are described in the Sects. 5.2, 5.3, and 5.4, respectively. In Sect. 5.5, this is used to define the execution of state machine transitions and system runs. The extension to deferred events is presented in Sect. 5.6

5.1 Configurations

By means of an abstract datatype in PVS, we define the *status* of an object. There are three possibilities:

- `dormant`, when the object has not yet been created.
- `free`, when it is created and free to answer operation calls (i.e., it is not processing a triggered operation call).
- `busy(caller, attr)`, when it is busy processing an operation; if it has accepted a call for which it has not returned the result yet. `caller` is the identity of the caller and `attr` is the attribute of the caller to which the result must be assigned.

```
Status: DATATYPE
 BEGIN
  dormant: dormant?
  free: free?
  busy(caller: Obj, attr: Att): busy?
 END Status
```

To represent the event queue, we first define a simple general theory for a queue with elements of type `T`.

```
Queue[T: TYPE]: THEORY
BEGIN
Queue: TYPE = list[T]
elt  : VAR T
q    : VAR Queue

insert(elt, q): Queue =
    append(q, cons(elt, null))

nonempty?(q): bool = cons?(q)
neq: VAR (nonempty?)

first(neq): T     = car(neq)
rest(neq) : Queue = cdr(neq)
END Queue
```

A *local object configuration*, represented by `ObjectConfig` in PVS, contains all relevant local information concerning an object. It is a record with five fields:

- `val`: a *valuation* that assigns values to attributes and references.
- `state`: the current state of the state machine of the object.
- `eventq`: the event queue of the object.
- `status`: the status of the object.
- `suspended`: a boolean indicating whether the object is currently suspended, i.e., it is waiting for return after having done a call. Note this cannot be included in the status, because an object may become suspended when it is free or busy and in the last case (then it is calling another object during the processing of a call to an own triggered operation), the parameters of status busy are needed when it is no longer suspended.

It should be noted in PVS that we first import theory `Queue` with parameter `Signal`, before defining the object configuration. Next we define a (global) *configuration* as a mapping that assigns an object configuration to each object.

```
IMPORTING Queue[Signal]

ObjectConfig: TYPE+ =
  [# val: Val,
     state: State,
     eventq: Queue,
     status: Status,
     suspended: bool
   #]

Config: TYPE+ = [Obj -> ObjectConfig]
```

For convenience, we list a large number of variables that will be used in subsequent definitions. Note that the set of transitions `trans` is turned into a type by the `( ..)` brackets.

```
cl  : VAR Class;    obj : VAR Obj
sn  : VAR Sig;      sig : VAR Signal
eq  : VAR Queue;    d   : VAR Dom
pact:  VAR PrimAct; act : VAR Act
a   : VAR Att;      ref : VAR Ref
meth: VAR Meth;     op  : VAR Op
exp : VAR Exp;
n, i, j: VAR nat;   t, t1, t2 : VAR (trans);
c, c1, c2, current, next: VAR Config;
```

Finally, we introduce a few auxiliary definitions concerning stability:

– An object is *ready* in some configuration, if it is nondormant and not suspended.

```
ready?(obj, c): bool =
  NOT dormant?(c(obj)`status) AND
  NOT c(obj)`suspended
```

– A transition is *locally enabled* for an object in some configuration, if the object is ready, the transition belongs to the class of the the object, its source is the object's current state, and the boolean guard is satisfied.

```
locallyEnabled?(obj, t, c): bool =
   ready?(obj, c) AND
   t`class = class(obj) AND
   t`source = c(obj)`state AND
   t`guard(c(obj)`val)
```

– An object is *stable* in some state, if it is *ready* and all locally enabled transitions have a trigger.

```
stable?(obj, c): bool =
 ready?(obj, c) AND
 NOT EXISTS t:
     locallyEnabled?(obj, t, c) AND
     noTrig?(t`trigger)
```

Observe that enabledness of a transition does not only depend on its guard and trigger; the action part may also 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 nonprimitive actions.

5.2 Semantics of primitive actions

To define the effect of executing a list of primitive actions, we first define the effect of executing a single primitive action in a particular configuration, yielding a new configuration.

The effect of an assignment is a new configuration where the value of expression `exp` is assigned to attribute `a` in the attribute valuation (field `aval` of the `val` field) of the object `obj`.

```
assignEffect(obj, a, exp, c) : Config =
   c WITH [(obj)(val)(aval)(a) :=
                        exp(c(obj)`val)]
```

To define object creation, we assume that in any configuration there exists a dormant object of a particular class. In PVS, this is expressed by the axiom with name `NewObjExists` which states that there exists at least one function which delivers such an object in any state, i.e., type `NewObjFunction` is not empty. Next we declare a constant `newObj` of this type. We also define an initial object configuration here; it is used not at object creation but, as we will show later, it is assigned to all objects except the root at initialization.

```
NewObjFunction : TYPE =
 { nobj : [Config, Class -> Obj] |
    FORALL c, cl :
      dormant?(c(nobj(c, cl))`status) AND
      class(nobj(c, cl)) = cl }

NewObjExists : AXIOM
     EXISTS (nobj : NewObjFunction) : TRUE

newObj: NewObjFunction

initObjConf(cl): ObjectConfig =
 (# state:= initState(cl),
    eventq:= null,
    suspended:= FALSE,
    status:= dormant,
    val:= initVal(cl)
  #)
```

At object creation, simply the new object is assigned to the reference and its status is changed to `free`.

```
createEffect(obj, ref, cl, c) : Config =
 LET newobj = newObj(c, cl)
 IN c WITH [(obj)(val)(rval)(ref):= newobj,
            (newobj)(status):= free]
```

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).

To define the effect of a signal emission, the receiving object `rcv` is obtained by evaluating the reference `ref` in the current configuration (recall that `c(obj)` is the local configuration of object `obj`). The signal event is constructed from the signal name and the value of the expression `exp` and then inserted in the signal queue of the receiver.

```
emitEffect(obj, ref, sn, exp, c) : Config =
 LET rcv = c(obj)`val`rval(ref),
     newsig: Signal =
      (# signame:= sn, d:= exp(c(obj)`val) #)
 IN c WITH [(rcv)(eventq):=
              insert(newsig, c(rcv)`eventq)]
```

For the effect of primitive operation, i.e., a method call, the value of the reference `ref` yields the callee. Then the attribute `a` of the caller `obj` gets the result of executing the method `meth` of the class corresponding to the callee.

```
methEffect(obj, a, ref, meth, c) : Config =
 LET callee = c(obj)`val`rval(ref)
 IN c WITH [(obj)(val)(aval)(a):=
    meth(meth, class(callee))(c(callee)`val)]
```

Note that we do not require that the callee must be non-dormant, but we could show as a general property that references never refer to dormant objects.

A return statement has no effect if the object `obj` executing it is not busy. Otherwise, we obtain the caller and its result attribute from the status of `obj` and use this to assign the result value to this attribute and make the caller no longer suspended. Moreover, the status of `obj` switches to `free`.

```
returnEffect(obj, exp, c) : Config =
  IF busy?(c(obj)`status)
  THEN
   LET caller = caller(c(obj)`status),
       attr   = attr(c(obj)`status)
   IN c WITH
        [(obj)(status):= free,
         (caller)(val)(aval)(attr):=
                              exp(c(obj)`val),
         (caller)(suspended):= FALSE
        ]
  ELSE c ENDIF
```

Finally, we define the effect of executing primitive action `pact` by object `obj` in configuration `c` by case distinction. The effect of a list of primitive actions is defined recursively. To show that this is well-defined, PVS requires a measure function; here we can simply take the length of the list because this decreases for the recursive use of the definition.

```
primActEffect(obj, pact, c): Config =
 CASES pact
 OF
 assign(a, exp): assignEffect(obj,a,exp,c),
 create(ref, cl): createEffect(obj,ref,cl,c),
 emitSignal(ref,sn,exp):
            emitEffect(obj,ref,sn,exp,c),
 meth(a, ref, meth):
            methEffect(obj,a,ref,meth,c),
 return(exp): returnEffect(obj,exp,c)
 ENDCASES

alist: VAR list[PrimAct]

primActListEffect(obj, alist, c):
  RECURSIVE  Config =
   CASES alist
    OF
     null: c,
     cons(pact, rest):
      primActListEffect(obj, rest,
           primActEffect(obj, pact, c))
   ENDCASES
  MEASURE length(alist)
```

5.3 Semantics triggered operation calls

A call to a triggered operation `op` can only be executed if there exists a transition `t1` at the callee (this will be expressed later) and the following conditions hold, as expressed by the boolean `callCondition`:

- the callee must be stable and free,
- `t1` must be triggered by a call to `op` and,
- `t1` is locally enabled.

```
callCondition(obj, t1, ref, op, c): bool =
   LET callee = c(obj)`val`rval(ref) IN
     stable?(callee, c) AND
     free?(c(callee)`status) AND
     callTrig?(t1`trigger) AND
     op(t1`trigger) = op AND
     locallyEnabled?(callee, t1, c)
```

Note that by our restriction on transitions, the action part of `t1` is a list of primitive actions and hence no further conditions on `t1` are needed. This is also used in the definition of the effect of a call; assuming `t1` has a call trigger, we obtain the effect of the action list of `t1` in an intermediate configuration where `suspended` is true for the caller, the formal parameter of the operation trigger on `t1` is replaced by the value of the actual parameter (expression `exp`), the status of the callee becomes busy and the new state of `t1` is the target state of `t1`.

```
callEffect(obj, t1, ref, exp, a, c): Config =
 LET callee = c(obj)`val`rval(ref) IN
 IF callTrig?(t1`trigger)
 THEN
 primActListEffect(
  callee,
  alist(t1`action),
  c WITH
   [(obj)(suspended):= TRUE,
    (callee)(val)(aval)(param(t1`trigger)):=
                              exp(c(obj)`val),
    (callee)(status):= busy(obj, a),
    (callee)(state):= t1`target ])
 ELSE c
 ENDIF
```

5.4 Semantics signal triggers

In this section we describe the condition and the effect of a signal trigger. First we define when a transition `t` of object `obj` is triggered by signal `sig` in configuration `c`: `t` must be locally enabled, triggered by the signal and if the action part is a call, then the call condition defined in the previous section must hold.

```
sigTrigEnabled(obj, t, sig, c): bool =
 locallyEnabled?(obj, t, c) AND
 sigTrig?(t`trigger) AND
 signame(t`trigger) = signame(sig) AND
 (call?(t`action) IMPLIES
   EXISTS t1 :
      callCondition(obj, t1,
             ref(t`action), op(t`action), c))
```

Next we define the condition which expresses that the signal event at position `n` in the event queue triggers transition `t` and no previous event in the queue triggers an enabled transition.

```
sigTrigCondition(obj, t, n, c): bool =
 LET eq = c(obj)`eventq
 IN stable?(obj, c) AND
```

```
    nonempty?(eq) AND n < length(eq) AND
    sigTrigEnabled(obj, t, nth(eq, n), c) AND
    FORALL (m: below[n]):
     NOT EXISTS t2:
      sigTrigEnabled(obj,t2,nth(eq, m), c)
```

To define the effect of executing a signal trigger, we first define an operation `cleanUp` on event queues which removes all events before the `n`th event.

```
cleanUp(eq, n): RECURSIVE Queue =
  CASES eq
   OF
    null: null,
    cons(sig, rest):
      IF n > 0 THEN cleanUp(rest, n - 1)
      ELSE rest ENDIF
   ENDCASES
  MEASURE length(eq)
```

This leads to the effect of taking the signal trigger part of transition `t` by the `n`th event in the event queue. Note that the formal parameter obtains the value of the triggering event.

```
sigTrigEffect(obj, t, n, c): Config =
 LET eq = c(obj)`eventq IN
 IF sigTrig?(t`trigger) AND n < length(eq)
 THEN
  c WITH
    [(obj)(val)(aval)(param(t`trigger)):=
                                  d(nth(eq, n)),
     (obj)(eventq):= cleanUp(eq, n)]
 ELSE c ENDIF
```

5.5 State machine transitions and runs

Finally, we define the meaning of state machine transitions, leading to the runs of the system. The general idea is that a step from a current to a next configuration corresponds to the execution of a transition `t`. It is defined by means of two intermediate configurations `c1` and `c2`: `c1` is the configuration after evaluating the guard of the trigger and `c2` the one after executing the action part (details are presented below).

```
Step(current,next): bool =
  EXISTS obj, t, c1, c2 :
     GuardTrigger(obj, t, current, c1) AND
     ActionPart(obj, t, c1, c2) AND
     StateChange(obj, t, c2, next)
```

The definition for the guard and the trigger considers only a signal trigger, because a transition triggered by an operation is executed together with the call action.

```
GuardTrigger(obj, t, c1, c2) : bool =
   locallyEnabled?(obj, t, c1) AND
   IF sigTrig?(t`trigger)
    THEN EXISTS n :
         sigTrigCondition(obj, t, n, c1) AND
         c2 = sigTrigEffect(obj, t, n, c1)
    ELSE c2 = c1
    ENDIF
```

The action part considers the two cases for the action part which is either a list of primitive actions or a call of a triggered operation.

```
ActionPart(obj, t, c1, c2) : bool =
 CASES t`action
 OF
  call(a, ref, op, exp):
   EXISTS t1 :
    callCondition(obj, t1, ref, op, c1) AND
    c2 = callEffect(obj, t1, ref, exp, a, c1),
  alist(alist):
      c2 = primActListEffect(obj,alist, c1)
 ENDCASES
```

Finally, we define the next state.

```
StateChange(obj, t, c1, c2) : bool =
   c2 = c1 WITH [(obj)(state):= t`target]
```

The transitive closure of the step relation leads to the set of *runs* of the form $c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow \cdots$, representing all possible executions of the UML model.

To define the initial configuration, we define a root object and assign to each object the initial local configuration defined in Sect. 5.2 and only assign status free to the root object (so all others are dormant).

```
root: Obj

initConfig: Config =
  (LAMBDA obj: initObjConf(class(obj)))
               WITH [(root)(status):= free]
```

Then a run is a sequence of configurations where the first one is the initial configuration and each pair of successive configurations satisfies the step relation.

```
r : VAR sequence[Config]

Run: TYPE =
   { r | r(0) = initConfig AND
         FORALL i: Step(r(i), r(i+1)) }
```

5.6 Deferred events

Deferred event can be added easily to the basic semantics given above. Assume given a function `deferred?` which specified, for each class and each state, the set of deferred signal events. Then only the `cleanUp` operation has to be changed. Given parameter `n`, it now removes all events before position `n` from the queue, except those that are deferred in the current state; they are not discarded and remain in the signal queue.

```
deferred?: [Class, State -> setof[Sig] ]

cleanUp(obj, eq, n, c): RECURSIVE Queue =
 CASES eq
 OF
  null: null,
  cons(sig, rest):
  IF n > 0 THEN
   IF member(signame(sig),
                deferred?(class(obj),
                          c(obj)`state))
   THEN cons(sig, cleanUp(obj, rest, n-1, c))
   ELSE cleanUp(obj, rest, n-1, c) ENDIF
  ELSE rest ENDIF
 ENDCASES
 MEASURE length(eq)
```

Note that we decided that the preceding deferred events maintain their order in the queue.

The work of Lilius and Paltor [15] on model-checking state machines describes a more operational approach to deal with deferred events. There events in the queue are considered one-by-one and special care has to be taken to avoid livelock. Our approach is more declarative and avoids this by considering the complete queue in the definitions.

6 Adding time

In this section, the semantics is extended with a continuous notion of time. As in timed automata [2], 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 modeled 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. Hence, system behavior is again defined by a sequence of steps 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 [22] clock invariants on states 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 [4]. 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 semantics, whereas with invariants the condition also depends on the local configurations of objects.

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

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

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

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

If this transition is nonurgent (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).

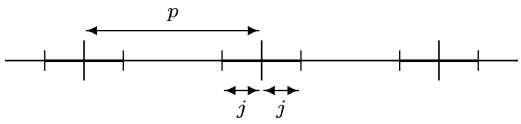


Fig. 4 Periodic messages with jitter

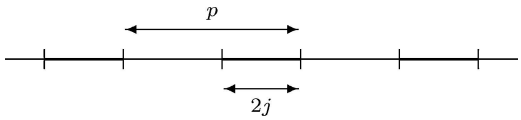


Fig. 5 Modeling of period plus jitter

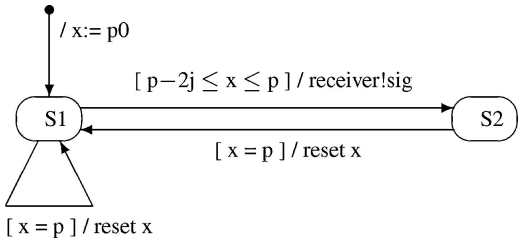


Fig. 6 Sender state machine. All transitions except the signal emission transition from $S1$ to $S2$ are urgent

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 p be the period of the cycle and let j be the jitter, see Fig. 4. Every period there is a time interval of length $2j$ during which the signal can be emitted. For a convenient modeling of the sender we shift our perspective on the period; a period starts at the end of the emission period, see Fig. 5.

The sender state machine is depicted in Fig. 6. Initially the value of the local clock x is p_0, with $p_0 \leq p$. The signal may be sent during the time interval from $p - 2j$ to p on the clock x. This transition is nonurgent, which means that time may progress while it is enabled. If the signal has not been sent at time p, time may not progress further because of the urgent time-out transition (the self-transition on state $S1$ with guard $x = p$). 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 period starts. If the emission does take place, the sender must wait in the state $S2$ for the end of the cycle, from which it urgently reenters the sending state $S1$.

The receiver state machine is shown in Fig. 7. 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 periods.

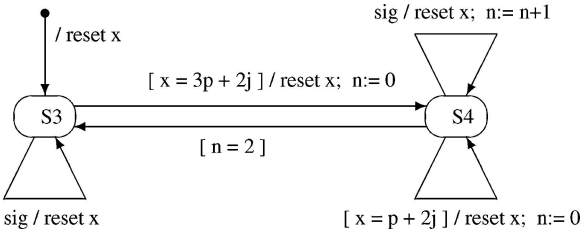


Fig. 7 Receiver state machine. All transitions are urgent

We model this with an urgent time-out transition which is taken if no signal is received for $3p + 2j$ time. The receiver stays in the error state $S4$ until it has received a signal in two consecutive periods; it then decides that it it safe to return to the operating state $S3$. A counter n is used to count these signals.

6.2 Timed semantics in PVS

In the section we show how the theories for the untimed semantics can be extended to deal with timing. First, we define a timed kernel language which has one additional type `Clock`.

```
TKL[Att, Class, Clock, State, Meth,
    Obj, Op, Ref, Sig, Dom: TYPE+]: THEORY
```

Let `Time` be the non-negative real number and we also use a delay time which must be positive (these subtypes of the real numbers are predefined in PVS). Moreover, valuations are extended with a mapping which assigns time values to clocks.

```
DelayTime: TYPE+ = posreal
Time     : TYPE+ = nonneg_real

CVal: TYPE+ = [Clock -> Time]

Val: TYPE+ =
   [# aval: AVal, rval: RVal, cval: CVal #]
```

The last syntactic extension is the addition of a reset to the primitive actions:

```
reset(clock: Clock): reset?
```

Next, we present the extensions to the theory `Semantics`, leading to theory `TSemantics` which also has a `Clock` parameter and additionally an `urgent` predicate on transitions.

```
TSemantics[
 Att, Class, Clock, State, Meth, Obj, Op,
 Ref, Sig, Dom: TYPE+,
 (IMPORTING TKL[Att, Class, Clock, State,
                Meth, Obj, Op, Ref, Sig, Dom])
 class: [Obj -> Class],
 initVal: [Class -> Val],
 initState : [Class -> State],
 trans: setof[Trans],
 meth: [Meth, Class -> Exp],
 urgent: pred[(trans)]    ] : THEORY
```

Of course, we add the effect of a reset action which resets the specified clock to 0.

```
clock : VAR Clock

resetEffect(obj, clock, c) : Config =
  c WITH [(obj)(val)(cval)(clock):= 0]
```

Next we define the condition and the effect of a delay step on the clocks. The effect of a delay `dt` on a configuration, as defined in Sect. 5.1 is simply the addition of `dt` to the values of all clocks.

```
u, dt: VAR DelayTime cval : VAR CVal

delayClocks(cval, dt): CVal =
  LAMBDA clock: cval(clock) + dt

delayEffect(c, dt): Config = LAMBDA obj:
  c(obj) WITH [(val)(cval):=
            delayClocks(c(obj)`val`cval, dt)]
```

Enabledness of a transition is defined using two abbreviations introduced in Sect. 5.5.

```
enabled?(t, c): bool =
    EXISTS obj, c2 :
     GuardTrigger(obj, t, c, c2) AND
     ActionPart(obj, t, c, c2)
```

Then the delay condition expresses that a delay step can be taken if no urgent transition can be taken after a smaller delay.

```
delayCondition(c, dt): bool =
  FORALL u: u < dt IMPLIES
   NOT EXISTS t:
    enabled?(t, delayEffect(c, u)) AND
    urgent(t)
```

Global configurations are extended with a global `time` field, to obtain a global notion of time that can be used to describe timing properties of a system.

```
TConfig: TYPE+ =
  [# config: Config, time: Time #]
```

The effect of a delay step then also includes updating the global notion of time. Finally, a (timed) step `TStep` is either an untimed `Step` of Sect. 5.5, where time does not progress, or a delay step.

```
tc1, tc2 : VAR TConfig

DelayStep(tc1,tc2) : bool =
   EXISTS dt :
    delayCondition(tc1`config, dt) AND
    tc2`time = tc1`time + dt

TStep(tc1,tc2): bool =
    (Step(tc1`config,tc2`config) AND
     tc1`time = tc2`time)
    OR
    DelayStep(tc1,tc2)
```

A run of the system is a sequence of timed configurations. As before, the initial configuration is the configuration in which there is a single nondormant root object; but

it can be redefined for a particular application. For example, in the example of Sect. 6.1, object creation is not modeled, and we would define the initial configuration as the configuration with two nondormant 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 p_0, and the initial value of x for the receiver would be 0.

As usual [1], we restrict the set of runs to the so-called *nonZeno* runs, disallowing Zeno behavior in which time never passes a certain bound. We formulate it as follows: a run is nonZeno 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.

```
r : VAR sequence[TConfig]

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

Run: TYPE =
 { r | r(0)`config = initConfig AND
       (FORALL i: TStep(r(i), r(i+1))) AND
       nonZeno(r) }
```

7 Activity groups; sharing control

In the semantics defined in Sect. 5, all objects are running asynchronously, which is modeled 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*. A class can be active or passive, and this leads to active or passive objects at run-time. Each activity group has exactly one active object and a number of passive objects. which are centered around active objects:

For example, in the system of Fig. 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, we can identify two activity groups, { *TempSensor*, *Thermometer* } and { *TempControl*, *Heating* }.

At any point of time, in every activity group exactly one object has the control. 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.

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? For instance, concerning triggered operations there are questions about the flow of the 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?

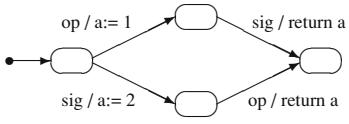
We decided that an object may only lose control if it is stable, except when a call is done to a triggered operation inside the same group. More precisely, a successful, synchronous call of a triggered operation requires that the caller has control and

- 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.

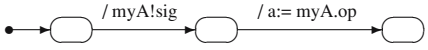
Execution of a return action need not lead to a control change; if callee and caller are in the same activity group, the callee must first become stable, before the control becomes available again for the caller.

Control is not needed to answer primitive operations; an object may reply to a calls of its primitive operations in any configuration.

To illustrate activity groups, we revisit the example from Sect. 3.1. For convenience, we repeat the state machines. For class *A*:



For class *B*:



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

7.1 Semantics of activity groups in PVS

To represent activity groups in PVS, theory `Semantics` is extended with two parameters: `active` is a predicate on classes to express which ones are active, and `rootClass` which defines an active root class. Moreover, we add an axiom to the theory to postulate the existence of at least one object `root` of type `rootClass`.

Type `Status` is extended with a new status `compl` which denotes that the object (the callee) is still completing a call after it already did a return but control could not change because the callee was not stable.

```
compl(caller: Obj): compl?
```

Note that when the callee becomes stable then control should go back to the caller if they are in the same group.

The object configurations, type `ObjectConfig`, are extended with two fields:

```
aobj     : Obj,
control  : Obj
```

Field aobj gives for passive objects the identity of the active object of its activity group; an active object is its own active parent. For active objects, field control provides the identity of the object that currently has the control within its group. This field is not used for passive object.

At initialization, a random object is assigned to these new fields and only at object creation they can be instantiated properly. Hence, the effect of object creation is modified such that a new active object becomes its own active parent and a passive object gets the same active object as its creator. Control is set to the newly created object, assuming that an active object starts having control (recall that the control field is not used for passive objects).

```
createEffect(obj, ref, c1, c) : Config =
 LET newobj = newObj(c, c1)
 IN c WITH [(obj)(val)(rval)(ref):= newobj,
            (newobj)(status):= free,
            (newobj)(aobj) :=
                IF active(newobj)
                THEN newobj
                ELSE c(obj)`aobj ENDIF,
            (newobj)(control) := newobj ]
```

The effect of a return statement is changed slightly; the new status becomes compl.

To formalize the change of control, first a few abbreviations. co(obj, c) is the object that has control in the activity group of obj. Moreover, we define when two objects are in the same activity group in a particular configuration.

```
co(obj, c): Obj =
  LET activeParent = c(obj)`aobj
  IN c(activeParent)`control

samegroup?(obj1, obj2,c): bool =
  (c(obj1)`aobj = c(obj2)`aobj)
```

The definition of a step is refined to include the change of control. After evaluating the guard and the trigger of a transition we have to ensure that the executing object has or gets control. This means that the object that has control must be stable.

```
GetControl(obj,c1,c2) : bool =
  stable?(co(obj, c1), c1) AND
  LET objaobj = c1(obj)`aobj
  IN c2 = c1 WITH
           [(objaobj)(control) := obj]
```

Furthermore, in the condition for a call we have to add the requirement that if the callee is in another group then it may get control because the object that has control is stable. Formally, we add:

```
 NOT samegroup?(obj,callee,c)
      IMPLIES stable?(co(callee, c), c)
```

The effect function of a call is extended with a clause where the control field of the active parent of the caller is set to the callee. Hence the callee gets control and if the caller and the callee are in the same activity group this means that the caller looses control.

As the last part of a step we add a check to see whether the executing object has status completing and became stable, because then control should return to the caller if caller and callee are in the same activity group.

```
CheckControlReturn(obj,c1,c2) : bool =
  IF compl?(c1(obj)`status) AND
     stable?(obj,c1)
  THEN
    LET caller = caller(c1(obj)`status),
        calleraobj = c1(caller)`aobj
    IN
    IF samegroup?(obj,caller,c1)
    THEN
       c2 = c1 WITH [(obj)(status):= free,
             (calleraobj)(control) := caller]
    ELSE c2 = c1 ENDIF
  ELSE c2 = c1 ENDIF
```

Using the additional definitions above, we obtain an updated definition of a step.

```
Step(current,next): bool =
  EXISTS obj, t, c1, c2, c3, c4 :
     GuardTrigger(obj, t, current, c1) AND
     GetControl(obj, c1, c2) AND
     ActionPart(obj, t, c2, c3) AND
     StateChange(obj, t, c3, c4) AND
     CheckControlReturn(obj, c4, next)
```

As in Sect. 5.5 the transitive closure of the step relation leads to the set of runs. 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).

8 Concluding remarks

We have presented a formal operational semantics of a subset of UML for modeling 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 modeled 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. For example, 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.

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 deferred events, 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.

8.1 Verification

Based on the semantics presented here, we verified within Omega a small example with an unbounded number of objects that are dynamically created (the so-called *Sieve* example, see, e.g., [7]). This example was modeled in UML using the Rhapsody tool [14]. 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 [17]. Moreover, a version of the NLR example described in Sect. 6.1 has been verified. More details about the verification can be found in [3].

In general, the untimed framework described in Sect. 5 is suitable to verify safety properties, i.e., properties that can be falsified by a finite part of a run. It can be used to verify that certain undesired situations will never occur. More difficult to verify are liveness properties which, for instance, may express that eventually some particular event will occur. To prove such properties, typically the user has to devise so-called well-founded ranking functions, whose values decrease until the desired event occurs. Moreover, the proof of such properties requires fairness assumptions, to be able to deduce that certain transitions will eventually be taken.

Based on experience with the TLPVS package, we have proposed the following definition [3]:

> For every activity group A, if there is always some transition of an object in A enabled, then A takes transitions infinitely often.

This definition rules out unfair runs in which an activity group always has at least one enabled transition but never a transition from this group is executed.

Observe that fairness is defined at the level of activity groups, but for the framework of Sect. 5 we assume all objects are active and hence obtain fairness at the object level. Moreover, it is not required that every transition that is continuously enabled will be taken, or that the same transition remain continuously enabled—it is sufficient to take any transition currently enabled for the activity group. This is motivated by the observation that switching control between objects in an activity group may cause transitions to become disabled. Finally, observe that in the timed framework of Sect. 6 liveness properties are usually derived from real-time safety properties which can be proved by means of the urgency predicate on transitions.

Verification experiments show that the close correspondence between semantic steps and the executed transitions is rather convenient. The most complicated part is the treatment of triggered operations which leads to two transitions that are executed simultaneously. It might be interesting to compare the current approach with alternatives where the call events are also stored in some event pool, e.g., in the same event queue as the signal events. Note, however, that such a more asynchronous treatment introduces more deadlock possibilities and will lead to more verification conditions.

Another topic of future work is the translation of the high-level timing annotations proposed within Omega [8] into our basic timing framework in PVS. We have to investigate which possibility is most suitable for interactive theorem proving. Moreover, we are working on the definition of an equivalent denotational, and hence compositional, semantics to enable compositional verification.

Acknowledgements We would like to thank the members of the Omega project for extensive discussions on the semantics issues presented here. The anonymous reviewers are gratefully acknowledged for many valuable comments and suggestions.

References

1. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Trans. Progr. Lang. Syst. **16**, 1543–1571 (1994)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
3. Arons, T., Hooman, J., Kugler, H., Pnueli, A., van der Zwaag, M.: Deductive verification of UML models in TLPVS. In: Proceedings of the UML 2004, LNCS 3273, pp. 335–349. Springer, Berlin Heidelberg New York (2004)
4. Bornot, S., Sifakis, J.: Relating time progress and deadlines in hybrid systems. In: Proceedings of the International Workshop, HART'97, Grenoble, LNCS 1201, pp. 286–300. Spinger, Berlin Heidelberg New York (1997)
5. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: Understanding UML: a formal semantics of concurrency and communication in real-time UML. In: Proceedings of the Symposium on Formal Methods for Objects and Components (FMCO 2002), LNCS 2852, pp. 71–98. Springer, Berlin Heidelberg New York (2003)
6. Damm, W., Josko, B., Votintseva, A., Pnueli, A.: 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)
7. de Boer, F.S.: A proof rule for process creation. In: Proceedings of the Third IFIP WG 2.2 Working Conference (Formal Description of Programming Concepts 3) (1987)
8. Graf, S., Ober, I.: A real-time profile for UML and how to adapt it to SDL. In: Proceedings of the SDL 2003: System Design, SDL Forum, LNCS 2708, pp. 55–76 (2003)
9. Harel, D., Gery, E.: Executable object modeling with statecharts. IEEE Comput. **30**, 31–42 (1997)
10. Harel, D., Kupfermann, O.: On the behavioral inheritance of state-based objects. In: Proceedings of the 34th International Conference on Component and Object Technology. IEEE Computer Society, Washington, DC (2000)
11. Hooman, J., van der Zwaag, M.B.: UML semantics in PVS. http://www.cs.ru.nl/~hooman/STTTpvs.html
12. IBM/Rational. Rose RealTime. http://www.ibm.com/
13. IF: http://www-verimag.imag.fr/~async/IF/
14. Ilogix. Rhapsody. http://www.ilogix.com/

15. Lilius, J., Paltor, I.P.: Formalising UML state machines for model checking. In: Proceedings of the UML 1999, LNCS 1723, pp. 430–444. Springer, Berlin Heidelberg New York (1999)
16. Owre, S., Rushby, J., Shankar, N.: PVS: a prototype verification system. In: Proceedings of the 11th Conference on Automated Deduction. Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer, Berlin Heidelberg New York (1992)
17. Pnueli, A., Arons, T.: TLPVS: a PVS-based LTL verification system. In: Proceedings of the Symposium on Verification: (Theory and Practice), LNCS 2772, pp. 598–625. Springer, Berlin Heidelberg New York (2003)
18. PVS: http://pvs.csl.sri.com/
19. Reggio, G., Astesiano, E., Choppy, C., Hußmann, H.: Analysing UML active classes and associated statecharts—a lightweight formal approach. In: Proceedings of the FASE 2000–Fundamental Approaches to Software Engineering, LNCS 1783, pp. 127–146. Springer, Berlin Heidelberg New York (2000)
20. Selic, B., Gullekson, G., Ward, P.T.: Real-time object-oriented modeling. Wiley, New York (1994)
21. UML: Documentation of the unified modeling language (uml). Available from the Object Management Group (OMG), http://www.uml.org/
22. Uppaal: http://www.uppaal.com/