Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML

Werner Damm[1], Bernhard Josko[1], Amir Pnueli[2], and Angelika Votintseva[1]

[1] OFFIS, Oldenburg, Germany
{damm,josko,votintseva}@offis.de
[2] The Weizmann Institute of Science, Rehovot, Israel
amir@wisdom.weizmann.ac.il

Abstract. We define a subset *krtUML* of UML which is rich enough to express all behavioural modelling entities of UML used for real-time applications, covering such aspects as active objects, dynamic object creation and destruction, dynamically changing communication topologies in inter-object communication, asynchronous signal based communication, synchronous communication using operation calls, and shared memory communication through global attributes. We define a formal interleaving semantics for this kernel language by associating with each model $M \in krtUML$ a symbolic transition system $STS(M)$. We outline how to compile industrial real-time UML models making use of generalisation hierarchies, weak- and strong aggregation, and hierarchical state-machines into *krtUML*, and propose modelling guidelines for real-time applications of UML. This work provides the semantical foundation for formal verification of real-time UML models described in the companion paper [11].

1 Introduction

The establishment of a real-time profile for UML [25], the proposal for a UML action language [24], and the installation of a special interest group shared between INCOSE and OMG to develop a profile for UML addressing specification of real-time systems at the system-level all reflect the pressure put on standardisation bodies to give a rigorous foundation to the increasing level of usage of UML to develop hard real-time systems.

Its increased use also for safety critical applications mandates the need to complement these modelling oriented activities with an agreement on the formal semantics of the employed modelling constructs, as a prerequisite for rigorous

This research was partially supported by the German Research Council (DFG) within the priority program Integration of Specification Techniques with Engineering Applications under grant DA 206/7-3 and by the Information Society DG of the European Commission within the project IST-2001-33522 OMEGA (Correct Development of Real-Time Embedded Systems).

formal analysis methods, such as formal verification of compliance to requirements. This need has been perceived by the research community, leading to a substantial body of formalisation of various subsets of UML, discussed in detail in Section 5 of this paper. The precise UML group has in a series of papers [5, 6, 7] been proposing a meta-modelling based approach, which however lacks the capability to address dynamics aspects at the level of detail required for formal verification. Approaches based on translation into existing formalisms, such as e.g the π-calculus [26, 27], ASMs [23], CASL [30], Object-Z [18] fall short of covering the rich range of behavioural modelling constructs covered in this paper. Closest to our work addressing the intricacies of understanding active objects are [29, 30].

Our approach takes into account functional aspects of real-time systems, considering discrete time model with two levels of granularity. In this paper we focus our investigation on the semantic foundation of such critical features of real-time applications as concurrency (including the specification of the time points for interferences) and two types of inter-object communication — synchronous, via operation calls, and asynchronous, via signal event emission. The described approach benefits from numerous discussions with industrial users employing UML tools for the development of real-time systems, e.g. the partners of the IST projects Omega [1] and AIT-Wooddes [2]. The IST project Omega has developed an agreed specification *rtUML* of those modelling concepts from UML required to support industrial users in their application development (Deliverable IST/33522/WP1.1/D1.1.1, [10]), subsuming such concepts as inheritance, polymorphism, weak and strong aggregation, hierarchical state machines, rich action language, active, passive, and reactive objects, etc., taking into account detailed issues such as navigability, visibility, changeability and ordering of association end-points, and allowing unbounded multiplicity of these.

We propose a two-stage approach to give a formal semantics to *rtUML*: A precompilation step translates *rtUML* models into a sufficiently compact sublanguage *krtUML*, eliminating the need at the kernel level to address the various facets of associations, inheritance, polymorphism, and hierarchical state-machines. We then give a formal semantics of *krtUML*, using the formalism of symbolic transition systems [22]. In this semantic framework, the state-space of the transition system is given by valuations of a set of typed system variables, and initial states and the transition relation are defined using first-order predicate logic. We show how to capture a complete snapshot of the dynamic execution state of a UML model, using unbounded arrays of object configurations to maintain the current status of all objects, and a pending request table modelling the status of all submitted, but not yet served operation calls. Object configurations include information on the valuation of the object's attributes, the state-configuration of it's state-machine, as well as the pending events collected in an event-queue.

Due to space restrictions, this paper focusses on the definition and formal semantics of *krtUML*, and only sketches some ideas of the precompilation phase.

[1] IST-2001-33522, `http://www-omega.imag.fr/index.php`
[2] IST-1999-10069, `http://wooddes.intranet.gr`

We refer the reader to [10] for a full description of this step, as well as for the specification of *rtUML*.

The paper is organized as follows. Section 2 gives a formal definition of the constituents of a *krtUML* model. Section 3, the heart of this paper, develops the STS-based semantics, motivating and introducing in consecutive sections the system variables spanning the state-space of the transition systems, and the transition relation itself. Section 4 highlights aspects of the pre-compilation step, addressing inheritance and aggregation. Section 5 discusses related work.

2 The krtUML Language

In developing *krtUML*, we strived to maintain in purified form those ingredients of UML relating to the interaction of active objects. Intuitively, an active object (i.e., an instance of an active class) is like an event-driven task, which processes its incoming requests in a first-in-first-out fashion. It comes equipped with a dispatcher, which picks the top-level event for the event-queue, and dispatches it for processing to either its own state-machine, or to one of the passive reactive objects associated with this active object, inducing a so-called run-to-completion step. We generalize this concept in Section 4 by proposing to group one active object and a collection of passive server objects into what we call *components*. [3] Within a component, all passive objects delegate event-handling to the one active object of the component; pre-compilation will capture this delegation relation by allowing to refer through *my_ac* to the active object responsible for event-handling of a passive object. While the semantical model is rich enough to support communication through shared attributes, operation calls, and signals, we restrict our communication model so that all inter-component communications are purely asynchronous, i.e. via signal events.

Our kernel language thus still caters for the difference between active and passive objects. All objects are assumed to be reactive, that is their behaviour can be made dependent on the current state of the system. We support so-called triggered operations, i.e. operation calls, whose return value depends on the current state of the system, as distinguished from what we call primitive operations, the body of which is defined by a program in the supported action language. Since primitive operations only involve services of an object within the same component, pre-compilation can eliminate all calls to primitive operations by inlining (assuming, that the call-depth of primitive operations is bounded). In contrast, for triggered operations the willingness of the object to accept a particular operation call in a given state is expressed within the state-machine, by labeling transitions emerging from the state with the operation name as triggering guard, in the same style as the willingness of the object to react to a given signal event is specified by using this signal as triggering guard. Reflecting the wish to make the return value of triggered operations dependent on the object state,

[3] In this paper, we use the notion of components which is a restriction of the more general concept from the standard UML. Namely, we consider only a kind of components containing exactly one active object.

its "body" is "spread out" over the state-machine itself: the acceptance of a call will induce a run-to-completion step, hence the transition-labels passed during this run-to-completion step determine the response for this particular invocation of the triggered operation. Pre-compilation will have flattened the hierarchical state-machines of *rtUML* into the flat state-machines considered in our kernel language. It will also have split compound transition annotations, hence within the kernel language, only atomic actions and triggering guards (signal/operation names possibly with conditions) are allowed as labels of transitions.

We now elaborate on the formal definition of *krtUML* models. Note that the different ingredients are mutually dependent, hence we collect them in one formal definition.

Definition 1 (*krtUML* model). *A krtUML model*

$$M = (T, F, Sig, <, C, c_{root}, A)$$

consists of the following elements:

- $T \supseteq \{\texttt{void}, \mathbb{B}, \mathbb{N}\}$*: A set of* basic types *comprising at least booleans and natural numbers.*
- *F: A set of typed* predefined primitive functions*.*
- *Sig: A finite set of* signals*. Every instance of a signal is called* signal event*, or* event *for brevity.*
- $< \subset Sig \times Sig$*: A* generalisation relation *on signals, i.e. the transitive closure* $<^+$ *is irreflexive, where* $ev_1 < ev_2$ *denotes that* ev_2 *is a generalisation of* ev_1*. In the following, we use* $\leq$ *to denote the reflexive transitive closure of* $<$*.*
- *C: A finite, non-empty set of* classes*. A class*

  $$c = (c.isActive, c.attr, c.ops, c.sm)$$

  consists of:
  - *c.isActive: A predicate. Class* $c \in C$ *is called* active *iff* $c.isActive = true$*.*
  - *c.attr: A finite set of typed* attributes*, which may not be of type* `void`*.*
  - *c.ops: A finite set of typed* triggered operations*.*
  - *c.sm: A* c-state-machine *as explained in (v) below in terms of c-actions over c-expressions.*

  Each class contains two specific implicit attributes *(introduced by the pre-processing):* $self \in c.attr$ *keeping the reference to the object itself, and* $my\_ac \in c.attr$ *specifying the event-handling object associated with class c.*
- $c_{root} \in C$*: The class of the* root object *(serving to specify system initialisation as defined in Definition 7).*
- $A \subset C$*: A subset of active classes called* actors *and used to denote external objects (part of the environment).*

krtUML allows for some set of base types T, as well as a set F of functions operating on them, including, in particular, booleans and natural numbers together with all logical and arithmetical operators. Signals as well as operations may have parameters of well-defined types. Note that we support explicitly generalisation hierarchies on signals (while generalisation hierarchies on objects are eliminated during pre-compilation). We now elaborate on the elements of *krtUML* model defined so far, and start by defining the supported types. Here we clear distinguish between base types and reference types (visible on the UML level), as well as a third category of types catering for implicit attributes representing association end-points, which typically hold a number of references depending on their multiplicity. By choosing to type these uniformly with functions from the naturals to classes, we cater for unbounded multiplicity. Operationally, we hence view such implicit attributes as unbounded arrays, with each index pointing to an associated object of a given class, or containing a nil-pointer.

Definition 1 (Continued)

(i) **Typing:** *A krtUML model M defines the set of types*

$$T(M) =_{df} T \cup T_C \cup T_{as}$$

where $T_C =_{df} \{T_c \mid c \in C\}$ is the set of reference types *and $T_{as} =_{df} \{\mathbb{N} \to T_c \mid c \in C\}$ the set of* association types*, which will be used to represent all kinds of associations described in [10] (i.e., composition, aggregation and neighbour).*

For each type $\tau \in T(M)$, we assume existence of a designated element $\mathsf{nil}_\tau \in \tau$ as a default value.

We use 'type' to denote the type of attributes, functions etc. as follows:

- *For each class $c \in C$ and each attribute $a \in c.attr$, type$(a) \in T(M)$ denotes the type of $a \in c.attr$,*
  where type$(self) = T_c \in T_C$ and type$(c.my_ac) \in T_C$.
- *For each class $c \in C$ and each triggered operation $op \in c.ops$, $type_{par}(op) = T_1 \times \cdots \times T_n$ denotes the parameter type where $T_i \in T(M)$ is the type of the i-th parameter and $type_r(op) \in T(M)$ denotes the type of the* reply value *($type_r(op) =$ `void` if op does not yield a return value). The type of op is defined as type$(op) = type_{par}(op) \to type_r(op)$.*
- *For each $f \in F$, $type_{par}(f) = T_1 \times \cdots \times T_n$ denotes the parameter type where $T_i \in T(M)$ is the type of the i-th parameter and $type_r(f)$ denotes the value type of f. The type of f is type$(f) = type_{par}(f) \to type_r(f)$.*
- *For each $ev \in Sig$, $type_{par}(ev) = T_1 \times \cdots \times T_n$ denotes the parameter type of ev where $T_i \in T(M)$ is the type of the i-th parameter.*

We next introduce the expression language, supporting navigation expressions, accessing objects through association end-points, and closing this under application of base-type functions (including equality and boolean operations). Expressions are terms defined in the scope of a class that can be used in transition

guards orprimitive actions of this class.

Definition 1 (Continued)

(ii) **Expressions:** *For a class $c \in C$, a* c-expression *'expr' is defined inductively as follows:*

- Navigation expression: $expr ::= r.a$,
  where $r \in c.attr$ *with* $type(r) = T_{c_0} \in T_C$ *and* $a \in c_0.attr$. *We set* $type(expr) =_{df} type(a)$. *Note, that we only consider "flat" navigation expressions in krtUML, where* r *can also refer to the object itself (if* $r = self$*).*
- Association access: $expr ::= expr_1[expr_2]$,
  where $expr_1$ *and* $expr_2$ *are* c-expressions $type(expr_1) = (\mathbb{N} \to T_{c'}) \in T_{as}$ *and* $type(expr_2) \in \mathbb{N}$. *We set* $type(expr) =_{df} T_{c'}$.
- Function application: $expr ::= f(expr_1, \ldots, expr_n)$,
  where $expr_1, \ldots, expr_n$ *are* c-expressions, $f \in F$, *and* $type(expr_i)$ *matches the type of the* i*-th parameter of* f, $0 < i \leq n$. *We define* $type(expr) = type_r(f)$.

In the following definition of c-guards, c-actions and c-state-machines, '$expr$', '$expr_1$', and '$expr_2$' denote c-expressions.

Guards can be just boolean expressions, or express the willingness to accept a signal event or an operation call, possibly conjoined with a boolean condition.

Definition 1 (Continued)

(iii) **Guards:** *For a class $c \in C$, a* triggering guard *to be used in the state-machine of class $c \in C$,* c-guard *for short, is one of the following:*

- Signal trigger*:* $ev[expr]$, *where* $ev \in Sig$ *and* $type(expr) = \mathbb{B}$.
- Call trigger*:* $op[expr]$, *where* $op \in c.ops$ *and* $type(expr) = \mathbb{B}$.
- Condition*:* $[expr]$, *where* $type(expr) = \mathbb{B}$.

We support a rich action language, allowing for object-creation and destruction, operation calls, event emission, and assignments of attributes and association end-points. The expression passed in an object creation is intended to pass the identity of the active-object responsible for event-handling. Reply actions serve to define the return value of a triggered operation.

Definition 1 (Continued)

(iv) **Actions:** *A* (primitive) action *to be used in the state-machine of class $c \in C$,* c-action *for short, is one of the following:*

- Object creation: $r.a := create_{c'}(expr)$,
  with $r \in c.attr$, $type(r) = T_{c_0} \in T_C$, $a \in c_0.attr$ *and* $type(a) = T_{c'} \in T_C$, *and* $type(expr) = type(c'.my\_ac)$.

- Object creation (into association place): $r.a[expr_1] := create_{c'}(expr_2)$, *with* $r \in c.attr$, $type(r) = T_{c_0} \in T_C$, $a \in c_0.attr$, $type(a) = (\mathbb{N} \to T_{c'}) \in T_{as}$, $type(expr_1) = \mathbb{N}$, *and* $type(expr_2) = type(c'.my\_ac)$.
- Attribute assignment: $r.a := expr$, *with* $r \in c.attr$, $type(r) = T_{c_0} \in T_C$, $a \in c_0.attr$, *and* $type(a) = type(expr)$.
- Association place assignment: $r.a[expr_1] := expr_2$, *with* $r \in c.attr$, $type(r) = T_{c_0} \in T_C$, $a \in c_0.attr$, $type(expr_1) = \mathbb{N}$, $type(a) = (\mathbb{N} \to T_{c'} \in T_{as})$, *and* $type(expr_2) = T_{c'}$.
- Event emission: $r.send(ev, expr_1, \ldots, expr_n)$, *with* $r \in c.attr$ *and* $type(r) \in T_C$, $ev \in Sig$, *and* $(\times_{i=0}^{n} type(expr_i)) = type_{par}(ev)$.
- Operation call (ignoring reply value): $r.call(op, expr_1, \ldots, expr_n)$, *with* $r \in c.attr$, $type(r) \in T_C$, $op \in type(r).ops$, *and* $(\times_{i=0}^{n} type(expr_i)) = type_{par}(op)$.
- Operation call (assigning value): $r.a := r'.call(op, expr_1, \ldots, expr_n)$, *with* $r \in c.attr$, $type(r) = T_{c_0} \in T_C$, $a \in c_0.attr$, *and* $r' \in c.attr$, $type(r') \in T_C$, $op \in type(r').ops$, *and* $(\times_{i=0}^{n} type(expr_i)) = type_{par}(op)$, *and* $type(a) = type_r(op)$.
- Operation call (assigning value into association place): $r.a[expr_0] := r'.call(op, expr_1, \ldots, expr_n)$, *with* $r \in c.attr$, $type(r) = T_{c_0} \in T_C$, $a \in c_0.attr$, *and* $r' \in c.attr$, $type(r') \in T_C$, $op \in type(r').ops$, *and* $(\times_{i=0}^{n} type(expr_i)) = type_{par}(op)$, *and* $type(a) = (\mathbb{N} \to c') \in T_{as}$, $type(expr_0) = \mathbb{N}$, *and* $type_r(op) = c'$.
- Setting reply value: $reply_\tau(expr)$, *with* $\tau \in T \cup T_C$ *and* $type(expr) = \tau$.
- Object destruction: $destroy(expr)$, *with* $type(expr) \in T_C$.

Triggering Guards and Actions appear as decorations of transitions of the state-machine of a class. We assume a designated destruction state. Pre-compilation will extend the user-defined state-machine by pre-fixing the initial state with a sequence of transitions modelling constructor actions, while the destruction state is the unique entry point into a section added by pre-compilation modelling destructor-code. Pre-compilation also transfers hierarchical statecharts into flat state-machines, each extended by a destruction state having no incoming transitions.

Definition 1 (Continued)

(v) **State-machines:** *A c-state-machine for a class* $c \in T_C$ *is a tuple*

$$c.sm = (c.Q, c.q_0, c.q_x, c.tr)$$

where

- $c.Q$ *is a finite set of* states.
- $c.q_0 \in c.Q$ *is the* initial state.

- $c.q_x \in c.Q$ *is the* destruction state*, which is used to mark the beginning of the destructor's actions.*
- $c.tr \subseteq c.Q \times (\{\gamma \mid \gamma$ *is a c-guard or c-action*$\}) \times c.Q$ *is the* transition relation. ■

We will use *krtUML* to denote the set of all *krtUML* models.

Note that on the *krtUML* level, there is intentionally no inheritance relation on classes, since for each class $c \in C$, inheritance is explained by the introduction of *uplink*- and *downlink*-neighbour associations $uplink_{c'}, downlink \in c.attr$ for each superclass c' of c in the preprocessing step. The uplink-association is used to model static polymorphism, whereas downlink-association allows to capture dynamic one.

Further note that association access is restricted to accessing a single index, i.e. on the *krtUML* level, there are no operations like iteration over associations or adding references. We assume that such operations are also explained in terms of primitive actions by the preprocessing.

The identification of actors is not considered necessary from a semantical point of view, since actors should be treated as every other active class.

But the information whether an object is an actor instance can be exploited in formal verification: these objects need not necessarily be encoded like ordinary objects but can be interpreted as an assumption about environment behaviour, i.e. the expected sequences of input stimuli.

In the following, we assume that the preprocessing step as outlined in Subsection 4.1 establishes the following set of requirements regarding the sets of attributes and the state-machines of a *krtUML* model, which we rely on in Section 3 when explaining the semantics.

(i) All attribute and triggered operation names of all classes are pairwise different, for example *qualified* by a class name like $c::a$, and all states of all state-machines are pairwise different.
(ii) For each class $c \in C$, $c.attr$ contains the attribute $c::my\_ac$ to store the reference to the responsible active object such that $c::my\_ac$ is of type $T_{c'}$ and $c'.isActive = true$.
(iii) Values of the implicit attributes $c::self$ and $c::my\_ac$ are assigned once at the initialization of the corresponding object and do not change during the life-time of the object.
(iv) For each triggered operation $op \in c.ops$, $c \in C$, there are attributes $c::op_{p_i} \in c.attr$, $1 \leq i \leq n$ to hold local copies of the parameters, typed s.t. $(c::op_{p_1}, \ldots, c::op_{p_n}) = type_{par}(op)$
(v) For each $ev \in Sig$ which $c \in C$ is *willing to receive*, i.e. there is a transition $(q, ev[expr], q') \in c.tr$, there are attributes $c::ev_{p_i} \in c.attr$, $1 \leq i \leq n$ to hold local copies of the signal parameters, typed s.t $(c::ev_{p_1}, \ldots, c::ev_{p_n}) = type_{par}(ev)$.

3 *krtUML* Semantics

We will give the semantics of *krtUML* in terms of symbolic transition systems, proposed in [22] under the name Synchronous Transition Systems. Separate subsections derive from types of *krtUML* models the type structure of related symbolic transition systems, and introduce the system variables required to represent a snapshot in the dynamic execution of a *krtUML* model. We then elaborate the way snapshots can evolve, defining for each of the possible cases a transition predicate. Finally, we define the predicate characterizing initial snapshots, and collect all pieces of the transition relation into a full predicative definition of the transition relation, leading to a definition of the symbolic transition system associated with *krtUML* model.

3.1 Symbolic Transition Systems

We first introduce the semantic model of symbolic transition systems, which allow for a purely syntactical description of a transition system by first-order logic predicates over a set of typed system variables.

Definition 2 (STS). *A* symbolic transition system *(STS)* $S = (V, \Theta, \rho)$ *consists of* V*, a finite set of typed* system variables*,* Θ*, a first-order predicate over variables in* V *characterizing the initial states, and* ρ*, a* transition predicate*, that is a first-order predicate over* V, V'*, referring to both primed and unprimed versions of the system variables (their current and next states).* ■

An STS *induces* a transition system on the set of interpretations of its variables as follows.

Definition 3 (Runs of an STS). *Let* $S = (V, \Theta, \rho)$ *be an STS and* $\mathcal{T}$ *the set of types of variables in* V*. Let* $\mathcal{D}_\tau$ *be a semantic domain for each* $\tau \in \mathcal{T}$*.*

(i) A snapshot

$$s : V \to \bigcup_{\tau \in \mathcal{T}} \mathcal{D}_\tau$$

of S *is a type-consistent interpretation of* V*, assigning to each variable* $v \in V$ *a value* $s(v)$ *over its domain.* Σ *denotes the set of snapshots of* S*.*

(ii) A snapshot $s \in \Sigma$ *inductively defines the value* $[\![expr]\!](s)$ *for first-order predicates 'expr' over* V *and the value* $[\![expr]\!](s, s')$ *for first-order predicates 'expr' over* V, V'*, where* s *provides the interpretation of unprimed and* s' *the interpretation of primed variables in 'expr'.*

(iii) A snapshot $s \in \Sigma$ *is called* initial*, iff* $[\![\Theta]\!](s) = true$*.*

(iv) Let $s, s' \in \Sigma$ *be snapshots of* S*. Snapshot* s' *is called* S-successor *of* s*, iff* $[\![\rho]\!](s, s') = true$*.*

(v) A computation*, or* run*, of* S *is an infinite sequence of snapshots* $r = s_0\, s_1\, s_2 \ldots$*, satisfying the following requirements:*

- Initiation: s_0 *is initial.*
- Consecution: *Snapshot* s_{j+1} *is an* S*-successor of* s_j*, for each* $j \in \mathbb{N}_0$.

(vi) The set of all computations of S *is denoted as* $runs(S)$*. We use* $r(i)$ *to denote the* i*-th snapshot of a run* $r \in runs(S)$ *and*

$$r/i =_{df} r(i)\, r(i+1)\, r(i+2) \ldots$$

to denote the infinite suffix starting at $r(i)$*,* $i \in \mathbb{N}_0$. ■

3.2 System Variables for the *krtUML* Semantics

We motivate our choice of types and system variables using snapshots related to the Automated Rail Car System described in [15], a model of autonomous rail-bound cars which transport passengers between terminals and which adhere to a simple arrive- and departure protocol to allocate and de-allocate platforms inside the terminal. We refer the reader to [15] for details.

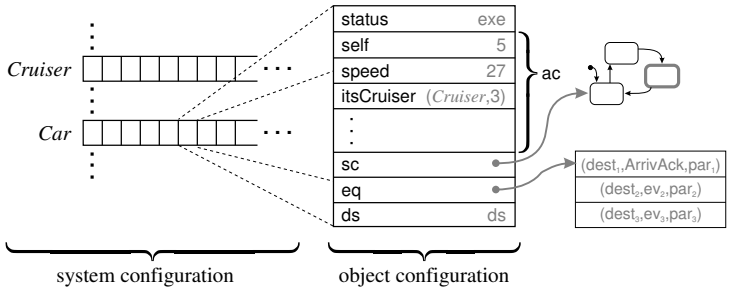


Fig. 1. System Configuration: A variable of type $\mathcal{T}_{sconf}$ contains one object configuration for every object identifier in O_C. The example of an object configuration *oconf* for the object $(\mathsf{Car}, 5)$ is shown enlarged.

Figure 1 depicts the way, how an object-configuration is captured. It shows enlarged the entry of an object of class *Car*, currently executing. The current state-machine configuration is illustrated by a state-machine, where in fact only the current state is stored. An object onfiguration not only gives the current valuation of all its attributes as well as its current state configuration, but also maintains the current object status (elaborated below), the event-queue (for active objects only), and a dispatcher status (for active objects only) used to enforce a single thread of control within the objects grouped into one component. The semantic entity representing a single class is a (potentially unbounded) array of object configurations, with each entry corresponding to a single instance of this class. The object status reflects the life-cycle of an object (see Figure 2). Prior to creation, objects are perceived as being dormant. Creation of a new object instance will pick a dormant index of the corresponding class, and awake the object to realities of life. During life, objects become suspended when waiting for completion of an operation call, and idle, (except for the special case

discussed below) when becoming stable, i.e. when a run-to-completion step terminates. This happens when reaching a state, where all outgoing transitions are either guarded by signal triggers (of the form *ev*[*expr*]) or call triggers (of the form *op*[*expr*]), or conditions (of the form [*expr*]) which are evaluated to false. In the particular case of accepting destruction, the object status will switch to dying, remaining in this status until its last run-to-completion step induced from the objects' destructor is finally completed. From then on, the object status will remain dead. Note, that destruction of an aggregate object (w.r.t. the composition association, defined in *rtUML*) induces destruction of all its parts, hence dying may be a long and painful process. Our semantics thus allows to observe nastities such as sending events to dying objects, as well as detecting dangling references.

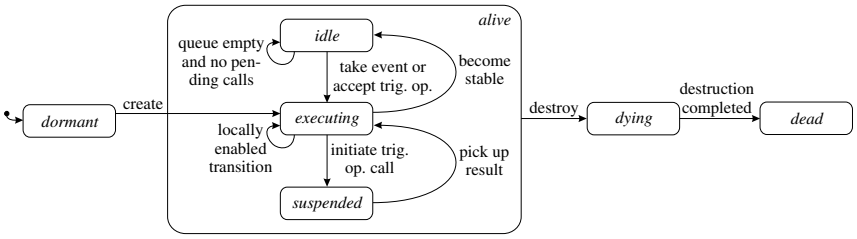


Fig. 2. Object life-cycle.

For the rest of the current section, let $M = (T, F, Sig, <, C, c_{root}, A)$ be a *krtUML* model.

We now define for the semantic types employed in the definition of the associated symbolic transition system, as well as the semantic domain of all semantic types. The type-system of semantic types subsumes all types of the *krtUML* model.

Definition 4 (Object Reference Types and Domains). *For each basic type $\tau \in T$, we assume the existence of a corresponding semantic type $\mathcal{T}_\tau$ with domain $\mathcal{D}_\tau$.*

For each type $T_c \in T_C$, we denote by O_c or $\mathcal{T}_{T_c}$ the corresponding semantic type and choose $\mathcal{D}_{O_c} =_{df} \{c\} \times \mathbb{N}$ as its domain. We call O_C with domain $\mathcal{D}_{O_C} =_{df} \bigcup_{c \in C} \mathcal{D}_{O_c}$, the object reference type resp. domain*. For each object type O_c, we assume existence of a designated element $\mathsf{nil}_c \in \mathcal{D}_{O_c}$ to serve as a default value.*

For each each association type $\tau = (\mathbb{N} \to T_c) \in T_{as}$, $\mathcal{D}_\tau =_{df} (\mathbb{N} \to \mathcal{D}_{O_c})$ is the domain of $\mathcal{T}_\tau$. ■

We now define the semantic type of system configurations and its associated domain, by first defining the semantic type of object configurations.

Definition 5 (Object and System Configuration).

(i) An object configuration $oconf = (status, ac, sc, eq, ds)$ *consists of the following elements:*

- *An* object status *'status' of type* $\mathcal{T}_{objstatus}$ *with associated semantic domain*

$$\mathcal{D}_{\mathcal{T}_{objstatus}} =_{df} \{dormant, idle, executing, suspended, dying, dead\}.$$

- *An* object attribute configuration *'ac' of type* $\mathcal{T}_{ac} =_{df} \bigcup_{c \in C} (c.attr \rightarrow \mathcal{T}_{T(M)})$.
- *An* object state-machine configuration *'sc' of type* $\mathcal{T}_{sc}$ *with associated semantic domain* $\mathcal{D}_{\mathcal{T}_{sc}} =_{df} \bigcup_{c \in C} c.Q$.
- *The* event queue *eq of type* $\mathcal{T}_{eq} =_{df} \mathcal{T}^*_{eqe}$, *i.e. a sequence of entries* $(dest, ev, par)$ *of type* $\mathcal{T}_{eqe} =_{df} O_C \times Sig \times \bigcup_{ev \in Sig} \mathcal{T}_{type_{par}(ev)}$.

  For an event-queue entry, 'dest' denotes the destination, *'ev' the* event type *(i.e. signal name), and 'par' the* event parameters. *We will use* ε *to denote empty event queue.*
- *A* dispatch reference *ds of type* $\mathcal{T}_{ds} =_{df} O_C$, *i.e. a reference to some object of any class which is used to denote the object currently processing an event.*

Thus the type of an object configuration *of* M *is*

$$\mathcal{T}_{oconf}(M) =_{df} \mathcal{T}_{objstatus} \times \mathcal{T}_{ac} \times \mathcal{T}_{sc} \times \mathcal{T}_{eq.} \times \mathcal{T}_{ds}$$

(ii) The type of a system configuration *is* $\mathcal{T}_{sconf}(M) =_{df} O_C \rightarrow \mathcal{T}_{oconf}(M)$.

(iii) We will call a component *to be a set* $Cm(o) = \{o' | o'.my\_ac = o\}$ *of objects assigned to the same event dispatcher* o. ■

The symbolic transition system uses the variable $sconf : \mathcal{T}_{sconf}$ to maintain the object-configuration of all objects of M. Note that, in general, the assignment of an event dispatcher to a reactive object can be user defined. In [10], a default assignment is given derived from the composition association.

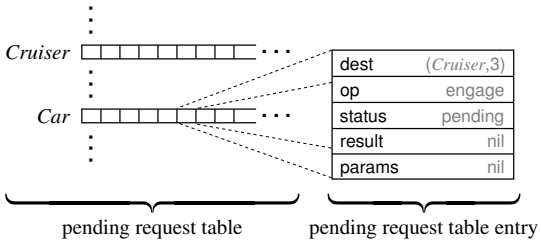


Fig. 3. Pending Request Table: The pending request table is a system variable of type $\mathcal{T}_{prt}$. It contains one entry for every object identifier in O_C.

We collect the status of all pending operation calls within a pending request table. An example on Figure 3 shows enlarged the entry for calls from an object of class *Car*. Currently the call of triggered operation *engage* for a *Cruiser* is pending. Here we exploit the fact, that all objects become suspended on calling an operation. We can thus maintain the status of all operation calls in a table indexed by sender objects resp. actors. Each entry in the pending request table maintains the identity of the receiver, the name of the requested operation, the list of parameters, a result-field, and status information. The life-cycle of an entry in the pending request table is depicted in Figure 4. Whenever the object owning the entry emits a new operation call, the status of the entry switches to pending. It will remain in this status until the receiving object is willing to serve the call, which causes the status to switch to busy. Once the run-to-completion step induced from accepting the call is terminated, the result of the call is entered into the result field of the entry, and its status changes to completed. This will allow the calling object to pick up the result and resume computation, causing the status of the entry to become unused.

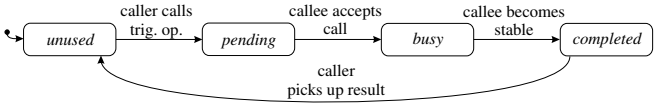


Fig. 4. Life-cycle of a triggered operation call.

Definition 6 (Pending Request Table).

(i) A pending request table entry $opreq = (dest, op, status, result, params)$ *maintains:*

- *The* receiver of a triggered operation call *'dest' of type* $\mathcal{T}_{dest}$ *with associated semantic domain* $\mathcal{D}_{\mathcal{T}_{dest}} =_{df} O_C$.
- *The* triggered operation identifier *'op' of type* $\mathcal{T}_{op}$ *with associated semantic domain* $\mathcal{D}_{\mathcal{T}_{op}} =_{df} \bigcup_{c \in C} c.ops$.
- *The* triggered operation status *'status' of type* $\mathcal{T}_{opstatus}$ *with semantic domain*
$$\mathcal{D}_{\mathcal{T}_{opstatus}} =_{df} \{unused, pending, busy, completed\}.$$
- *The* result *(or* reply*) 'result' of type* $\mathcal{T}_{res}$ *with associated semantic domain*
$$\mathcal{D}_{\mathcal{T}_{res}} =_{df} \bigcup_{\substack{c \in C \\ op \in c.ops}} type_r(op).$$
- *The* parameters *'params' of type* $\mathcal{T}_{par}$ *with associated semantic domain*
$$\mathcal{D}_{\mathcal{T}_{par}} =_{df} \bigcup_{\substack{c \in C \\ op \in c.ops}} type_{par}(op).$$

Thus the type of a pending request table entry is

$$\mathcal{T}_{opreq}(M) =_{df} \mathcal{T}_{dest} \times \mathcal{T}_{op} \times \mathcal{T}_{opstatus} \times \mathcal{T}_{res} \times \mathcal{T}_{par}.$$

(ii) The type of the pending request table *is* $\mathcal{T}_{prt}(M) =_{df} O_C \rightarrow \mathcal{T}_{opreq}(M)$. ■

The symbolic transition system uses the variable $prt : \mathcal{T}_{prt}$ to maintain the operation requests of all objects of M.

Furthermore, we need a boolean flag *sysfail*, which is used to indicate an undefined state of the system if e.g., it is tried to read the attribute of object reference nil or if the type of the reply action does not match the type of the currently processed triggered operation. Performing some arithmetic computations can also raise this flag in failure situations (e.g., division by 0). Initially, *sysfail* is set to *false* and it remains set, once it changed to *true*.

3.3 The Transition Predicate

Intuitively, there is a transition between two snapshots s, s' if there exists exactly one object $o \in O_C$ whose configuration changes by one of the following reasons:

- Object o is idle and an event is dispatched to it by its active object or an event with destination o is discarded since it is not enabled in o's state-machine. (Coarse-granularity flow of control is kept by elements *ds* of active objects' configurations.)
- Object o is idle and accepts a triggered operation call. (Fine-granularity flow of control is kept by elements *dest* of the pending request table.)
- Object o is executing or dying, unstable, and takes a transition of its state-machine and thereby executing an action. (No changes in the flow of control.)
- Object o is suspended and picks up the result of a triggered operation call which has been completed by the callee. (Fine-granularity flow of control kept by *dest* in *prt*.)

The system may remain in snapshot s if no object is *executing* (implying that pending request table is empty) and all event queues are empty.

In the following, we formalize each of the above conditions separately as first-order logic predicates which are then used to construct the transition predicate of the semantics $S(M)$.

For brevity, we use the following abbreviations for $o \in O_C$ in the definitions of predicates over *sconf* and *prt*:

- $o.status \equiv_{df} sconf(o).status$ and analogously for *sc*, *ds*, *eq*.
- $o.a \equiv_{df} sconf(o)(a)$, i.e. the value of attribute a.
- $o.a.b \equiv_{df} sconf(sconf(o)(a))(b)$, for attributes b of reference type.
- For an event or operation parameter tuple e, we use $o.ev'_p := e$ to denote simultaneous assignment of the i-th components of e to their corresponding attributes ev_{p_i} in o.

A primed abbreviation indicates that the primed system variable is to be used, for example $o.a' \equiv sconf'(o).a$.

For an event-queue $q = e_1\, e_2 \ldots e_n \in \mathcal{D}_{\mathcal{T}_{eq}}$ we introduce the following abbreviations:

- $head(q) =_{df} e_1$ denotes the first entry of the queue if $q \neq \varepsilon$.
- $tail(q) =_{df} e_2 \ldots e_n$ denotes q with the first entry removed and ε if $n < 2$.
- $enqueue(e, q) =_{df} q\, e$ denotes the result of appending entry $e : \mathcal{T}_{eqe}$ to q.

For brevity we assume that boolean expressions *expr* are evaluated to $\bot$ if it is for example tried to read an attribute via a reference with value nil.

Note that in the following incremental definition of the transition predicate, we use an assigment symbol ":=" which has to be processed as explicated in Definition 7 to yield the final transition predicate. Informally, this symbol indicates that there is no difference between the current and next states of the system variables other than specified explicitly in the sequence of the ":="-expressions (or their constituents). We will use $\oplus$ to denote logical XOR-operator: $a \oplus b =_{df} (a \vee b) \wedge \neg(a \wedge b)$.

We first define for each object $o \in O_c$ the predicate $stable(o)$ in the current system configuration as follows: [4]

$$
\begin{aligned}
stable(o) =_{df}\ & \forall\ (q, \gamma_s, q') \in c.tr : q = o.sc \implies \\
& ((\gamma_s \equiv \text{“}ev[expr_1]\text{”} \wedge sysfail' := (sysfail \vee expr_1 = \bot)) \\
& \vee (\gamma_s \equiv \text{“}op[expr_2]\text{”} \wedge sysfail' := (sysfail \vee expr_2 = \bot)) \\
& \vee (\gamma_s \equiv \text{“}[expr_3]\text{”} \wedge \neg expr_3 \wedge sysfail' := (sysfail \vee expr_3 = \bot)))
\end{aligned}
$$

Getting an Event. Formally, an event ev_1 with destination o can be dispatched to o from the event queue of its active object, if no other object of o's active object is currently processing an event and if a transition (q, γ, q') guarded by a superclass ev of ev_1 is enabled, i.e. originating at the current state q (cf. Figure 5):

$$
\begin{aligned}
\rho_{get\_event} =_{df}\ & \gamma \equiv \text{“}ev[expr]\text{”} \wedge o.my\_ac.ds = \mathsf{nil} \\
& \wedge\ expr = true \wedge sysfail' := (sysfail \vee expr = \bot) \\
& \wedge\ o.my\_ac.eq \neq \varepsilon \wedge head(o.my\_ac.eq).dest = o \\
& \wedge\ o.my\_ac.eq' := tail(o.my\_ac.eq) \\
& \wedge\ (\exists\ ev_1 \in Sig : \\
& \qquad \wedge\ head(o.my\_ac.eq).ev = ev_1 \wedge ev_1 \leq ev \\
& \qquad \wedge\ (\neg stable(o)' \\
& \qquad\qquad \implies (o.my\_ac.ds' := o \wedge o.status' := executing)) \\
& \qquad \wedge\ o.ev'_p := head(o.my\_ac.eq).par)
\end{aligned}
$$

[4] Here and later on: $\gamma \equiv$ "$ev[expr]$" ($\gamma \equiv$ "$op[expr]$" or $\gamma \equiv$ "$[expr]$") means that the label γ of the current transition (q, γ, q') is of the form $ev[expr]$ ($op[expr]$ or $[expr]$, resp.), i.e. a signal trigger (a call trigger or a condition, resp., cf. Definition 1 (iii)).

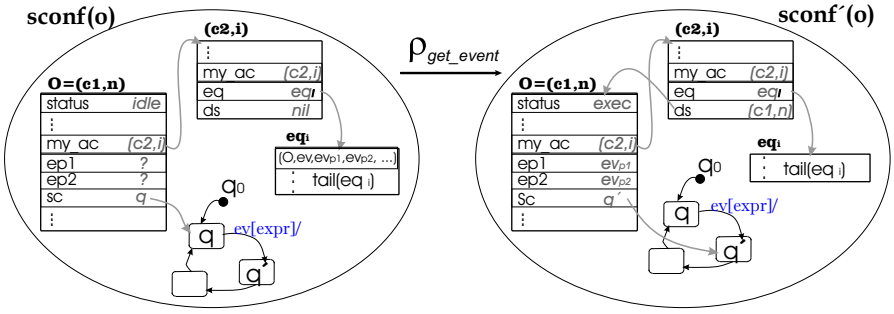


Fig. 5. Transition relation: ρ_{get_event}.

Element *o.my_ac.ds*, when not equal to nil, locks its component for processing a signal event. It can be released (and the component can start to process the following event, i.e. new run-to-completion step) only when all computations within the component are completed.

Note that we exploit the fact, that the syntactic category of boolean expression used in the definition of *krtUML* models is subsumed in the expression language of the first-order logic used to define transition predicates. In particular the above defined abbreviations apply to expressions of transition predicates thus providing the intended relation to *sconf*.

Accepting a Triggered Operation. Object o can accept a triggered operation call op if a transition (q, γ, q') with guard op is enabled in the current state q and some object o_1 has called op of o:

$$\begin{aligned}\rho_{accept\_op} =_{df}\ \gamma \equiv\ & \text{“}op[expr]\text{”} \wedge expr = true \wedge sysfail' := (sysfail \vee expr = \bot) \\ & \wedge (\exists\ o_1 \in O_C :\ prt(o_1).dest = o \wedge prt(o_1).op = op \\ & \quad \wedge prt(o_1).status = pending \\ & \quad \wedge (\neg stable(o)' \\ & \qquad \Longrightarrow\ prt(o_1).status' := busy \wedge o.status' := executing) \\ & \quad \wedge (stable(o)' \implies prt(o_1).status' := completed) \\ & \quad \wedge prt(o_1).result' := \mathsf{nil} \wedge o.op'_p := prt(o_1).op_p)\end{aligned}$$

Note that an object can call a trigger operation only from an object of the same component because of the restrictions on the inter-component communication. Thus, $o.my\_ac.ds' = o.my\_ac.ds = o_1$ during the execution of operations within one RTC-step (the change of the control between objects at this level of communication is captured by $prt(o).dest$ with $prt(o).status$).

Skipping Guards. Object o can take a transition guarded with a boolean expression only, if the expression evaluates to *true*:

$$\rho_{skip\_guard} =_{df} \gamma \equiv \text{“}[expr]\text{”} \wedge expr = true \wedge sysfail' := (sysfail \vee expr = \bot)$$

Discarding Events. If there is an event for object o in the queue of o's active object but o is not willing to accept it, i.e. if no transition with a matching signal is enabled, then the event is simply discarded:

$$\begin{aligned}\rho_{discard\_event} =_{df}\ & o.my\_ac.ds = \mathsf{nil} \\ & \wedge\, o.my\_ac.eq \neq \varepsilon \wedge head(o.my\_ac.eq).dest = o \\ & \wedge\, o.my\_ac.eq' := tail(o.my\_ac.eq) \\ & \wedge\, (\forall\ (q, ev_1[expr], q') \in c.tr : \\ & \qquad (expr = false \vee ev_1 \not\leq head(o.my\_ac.eq).ev) \\ & \qquad \wedge\, sysfail' := (sysfail \vee expr = \bot))\end{aligned}$$

Note that triggered operation calls are not discarded, but remain until the callee accepts the call.

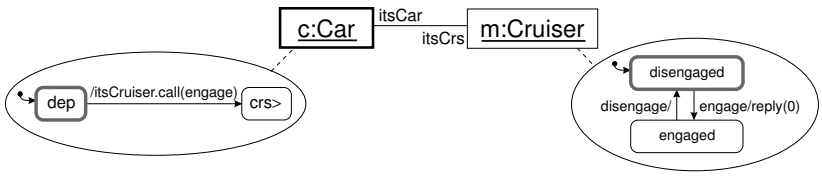
Executing Actions. Object o can execute an action if the current transition (q, γ, q') is enabled and annotated with the action. Below we distinguish two types of actions — operation calls and other actions — treating them in two separated subformulas. These subformulas will be combined with different contexts — conditions on their performance — in the final transition predicate.
An assignment action simply assigns a value to the destination. An event-sending action causes a new event to be appended to the queue of the destination's active object. A reply action causes the parameter value to be written into the reply field of the pending request table at o_1 if o processes the call from another object o_1; otherwise system failure is indicated. A destroy action causes the destination's state-machine configuration to be changed s.t. q_x is the current state and the status is "dying". Then the subsequent steps will execute the actions of the destructor. Killing a dying or dead object causes a system failure:

$$\begin{aligned}\rho_{non\_op\_action} =_{df}\quad & (\gamma \equiv \text{“}r.a := expr\text{”} \wedge (expr \neq \bot \implies o.r.a' := expr) \\ & \wedge\, (sysfail' := (sysfail \vee o.r = \mathsf{nil} \vee expr = \bot))) \\ & \vee (\gamma \equiv \text{“}r.send(ev, expr_1, \ldots, expr_n)\text{”} \\ & \wedge\, sysfail' := (sysfail \vee o.r = \mathsf{nil} \vee \bigvee_{i=0}^{n} expr_i = \bot) \\ & \wedge\, o.r.my\_ac.eq' \\ & \qquad := enqueue(o.r.my\_ac.eq, (o.r, ev, (expr_1, \ldots, expr_n)))) \\ & \vee (\gamma \equiv \text{“}reply_\tau(expr)\text{”} \\ & \wedge\, [(\exists\ o_1 \in O_C : \\ & \qquad prt(o_1).dest = o \wedge prt(o_1).status = busy \\ & \qquad\quad \implies prt(o_1).result' := expr \\ & \qquad\qquad \wedge\, sysfail' := (sysfail \vee \tau \neq type_r(o_1) \vee expr = \bot)) \\ & \quad \oplus\, sysfail' := true])\end{aligned}$$

$$
\begin{array}{l}
\vee\ (\gamma \equiv \text{“}destroy(expr)\text{”} \\
\quad \wedge\ [(expr \neq \bot \wedge \exists\ o_1 \in O_C : o_1 = expr \neq \mathsf{nil} \\
\qquad \wedge\ o_1.my\_ac = o.my\_ac \\
\qquad \wedge\ (o_1.status' \notin \{dormant, dying, dead\} \\
\qquad \implies [o_1.sc' := q_x \\
\qquad\qquad \wedge\ (\neg stable(o)' \implies o_1.status' := dying) \\
\qquad\qquad \wedge\ (stable(o)' \implies o_1.status' := dead)))] \\
\quad \oplus\ sysfail' := true])
\end{array}
$$

An operation call action suspends object o and configures o's entry of the pending request table s.t. it denotes the callee, the called triggered operation, and the parameters. Initially the status of the operation is "*pending*". A creation action is handled like a triggered operation since the caller should block until an object of the desired class is readily created with all inherited parts and all aggregated parts:

$$
\begin{array}{ll}
\rho_{init\_opcall\_or\_create} =_{df} & (\gamma \equiv \text{“}r.call(op, expr_1, \ldots, expr_n)\text{”} \\
& \quad \vee\ \gamma \equiv \text{“}r_1.a := r.call(op, expr_1, \ldots, expr_n)\text{”}) \\
& \wedge\ o.r.my\_ac = o.my\_ac \\
& \wedge\ sysfail' := (sysfail \vee o.r = \mathsf{nil} \vee \bigvee_{i=1}^{n} expr_i = \bot \\
& \quad \vee\ o.r.my\_ac \neq o.my\_ac) \\
& \wedge\ (o.status' := suspended \\
& \quad \wedge\ prt(o).dest' := o.r \wedge prt(o).op' := op \\
& \quad \wedge\ prt(o).status' := pending \\
& \quad \wedge\ prt(o).result' := \mathsf{nil} \\
& \quad \wedge\ prt(o).op'_p := (expr_1, \ldots, expr_n)) \\
& \vee(\gamma \equiv \text{“}r.a := create_{c_1}(expr)\text{”} \wedge\ o.my\_ac = expr \wedge \\
& \quad sysfail' := (sysfail \vee expr = \bot\ \vee o.my\_ac \neq expr \vee \\
& \quad (expr = o_1 \in O_C \wedge o_1.status \in \{dormant, dying, dead\})) \\
& \wedge\ (\exists\ o_1 \in O_{c_1} \setminus \{\mathsf{nil}_{c_1}\} : \\
& \qquad o_1.status = dormant \wedge o_1.status' := idle \\
& \qquad \wedge\ (\neg c_1.isActive \implies o_1.my\_ac' := expr) \\
& \qquad \wedge\ (c_1.isActive \implies o_1.my\_ac' := o_1) \\
& \qquad \wedge\ o.r.a' := o_1 \wedge o.status' := suspended \\
& \qquad \wedge\ prt(o).dest' := o_1 \wedge prt(o).op' := create_{c_1} \\
& \qquad \wedge\ prt(o).status' := pending \\
& \qquad \wedge\ prt(o).result' := \mathsf{nil} \wedge prt(o).params' := \mathsf{nil})
\end{array}
$$



	c.status	c.sc	m.status	m.sc
sconf':	suspended	dep	idle	disengaged
sconf'':	suspended	dep	executing	disengaged
sconf''':	suspended	dep	idle	engaged
sconf'''':	executing	crs	idle	engaged

	dest	op	status	ret	par
prt[c]':	m	engage	pending	nil	nil
prt[c]'':	m	engage	busy	nil	nil
prt[c]''':	m	engage	completed	0	nil
prt[c]'''':	nil	nil	nil	nil	nil

Fig. 6. Triggered Operation Call. Changing status of the caller c, callee m, and the called operation *engage* in the pending request table between the beginning (unprimed variables, not depicted here) and the end (at time t'''') of the operation call: the values of selected elements of *sconf* and *prt*.

Becoming Stable. If object o becomes stable, some bookkeeping takes place. If o was processing an event, the dispatch reference of its active object is reset. If o was executing a triggered operation, the pending request table status is set to "*completed*" to event completion to the caller. In both cases, o becomes idle. If o is executing the run-to-completion step starting at q_x, then it becomes dead:

$$\begin{aligned}\rho_{becoming\_stable} =_{df}\ & (stable(o)' \implies [o.my\_ac.ds = o \\ & \qquad \implies (o.my\_ac.ds' := \mathsf{nil} \wedge o.status' := idle)] \\ & \wedge\ [\forall\, o_1 \in O_C : \\ & \qquad prt(o_1).dest = o \wedge prt(o_1).status = busy \\ & \qquad\quad \implies (prt(o_1).status' := completed \\ & \qquad\qquad \wedge\, o.status' := idle)]) \\ & \wedge (o.status = dying \implies o.status' := dead)\end{aligned}$$

Picking Up a Result. Object o can pick up the result of a previous triggered operation call if the callee has set the status of o's pending request table entry to "*completed*":

$$\begin{aligned}\rho_{pick\_up\_result} =_{df}\ & prt(o).status = completed \implies prt'(o) := \mathsf{nil} \\ & \wedge (\neg stable(o)' \implies o.status' := executing) \\ & \wedge ((\gamma \equiv \text{“}r_1.a := r_0.call(op, expr_1, \ldots, expr_n)\text{”} \\ & \qquad \wedge \neg o.r_1 = \mathsf{nil}) \implies o.r_1.a' := prt(o).result) \\ & \wedge (sysfail' := (sysfail \vee o.r_1 = \mathsf{nil})\end{aligned}$$

The complete execution of an example of a triggered operation *engage*() is illustrated in Figure 6. The first row of the tables show the relevant part of

the system configuration at time t', just after c has entered the call into the pending request table. Note that c has not yet taken the transition, it remains in its previous state. The second row shows time t'', just after the *Cruiser* m has accepted the call. At time t''', m has just completed its run-to-completion step, i.e. written the result, changed the operation's status to "completed", and become idle. This is an indicator for c to pick up the result at time t'''', i.e. read the reply value from the table, clear the table entry, and now take the transition. c is executing and continues its run-to-completion step, assuming that c does not become stable.

3.4 The STS Semantics of a *krtUML* Model.

Putting all specifications of different kinds of transitions together we define the semantics of *krtUML* as a symbolic transition system over the three system variables (from Subsection 3.2) with the initial condition and combined transition relation specified in the following definition.

Definition 7 (*krtUML* Model Semantics). *Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a krtUML model. The* semantics of M *is the STS*

$$STS(M) = (V, \Theta, \rho), \text{ where}$$

System Variables: $V =_{df} \{sconf : \mathcal{T}_{sconf}(M), prt : \mathcal{T}_{prt}(M), sysfail : \mathbb{B}\}$.

Initial condition: *Initially a single object of class c_{root} exists and has status "executing". All other objects are dormant, and all attributes have default values:*

$$\begin{aligned}\Theta =_{df}\ & \exists\ o_0 \in O_{c_{root}} \setminus \{\mathsf{nil}_{c_{root}}\} : \\ & (o_0.status = executing \wedge o_0.ds = o_0 \\ & \wedge o_0.sc = c_{root}.q_0 \wedge o_0.eq = \varepsilon \\ & \wedge (\forall\ o_1 = (c_1, n_1) \in O_C \setminus \{o_0\} : \\ & \quad o_1.status = dormant \wedge o_1.sc = c_1.q_0 \\ & \quad \wedge o_1.ds = \mathsf{nil} \wedge o_1.eq = \varepsilon)) \\ \wedge\ & \forall\ o = (c, n) \in O_C : o.c::self = o \\ & \quad \wedge (\forall\ a \in c.attr : o.a = \mathsf{nil}_{type(a)})\end{aligned}$$

The unique single object of class c_{root} which is alive at the beginning of a run r is called the root object *of r.*

Transition relation: *The intermediate predicate ρ_0 composes the above introduced subpredicates as follows:*

$$\begin{aligned}\rho_0 =_{df}\ & \forall\ o \in O_C : o.status \neq executing \wedge o.eq = \varepsilon \\ & \vee \big(\neg sysfail \wedge \exists\ o = (c, n) \in O_C\ \exists\ (q, \gamma, q') \in c.tr :\end{aligned}$$

$$
\begin{aligned}
o.sc = q \wedge \big(o.sc' := q' \wedge (\\
&[o.status = idle \wedge (\rho_{get\_event} \oplus \rho_{accept\_op})]\\
&\vee [(o.status = executing \vee o.status = dying)\\
&\quad \wedge (\rho_{skip\_guard} \vee \rho_{non\_op\_action})]\\
&\vee [o.status = suspended \wedge \rho_{pick\_up\_result}])\\
&\wedge \rho_{becoming\_stable}\\
&\vee o.sc' := o.sc \wedge ([o.status = idle \wedge \rho_{discard\_event}]\\
&\vee [o.status = executing \wedge \rho_{init\_opcall\_or\_create}])))
\end{aligned}
$$

The final transition relation ρ is obtained from ρ_0 by adding a frame axiom *which requires that only those places of s are allowed to change in the transition to s', which get new values by an assignment ":=" in ρ_0, and changing the assignments to "=".*
The semantics of a krtUML model M is given as the set $runs(STS(M))$ of all computations in M. ■

It is easy to see that ρ effectively restricts activity to at most one object, resulting in an interleaving of actions from different objects.

The following consequence from Definition 3 and Definition 7 formalises the main properties of the described *krtUML* semantics.

Consequence: Let M be a *krtUML* model, $r \in runs(STS(M))$, ($sconf$, prt, $sysfail$) $\in r$, and $o \neq \bar{o} \in O_C$. Then

(i) (Level of the Computation Concurrency)
$sconf(o).status = sconf(\bar{o}).status = executing \iff o \in Cm(o_1)$, $\bar{o} \in Cm(o_2) \wedge o_1 \neq o_2$ — only objects from different components can be executing at the same time.

(ii) (Component Interference Points)
$sconf(o).my\_ac.ds = \mathsf{nil} \iff (\forall \hat{o} \in O_C : sconf(\hat{o}).my\_ac = sconf(o).my\_ac \Rightarrow sconf(\hat{o}).status \in \{idle, dead, dying\})$. An object o can accept an event only if other objects from its component are not currently executing. ■

4 Assessing the Expressiveness of *krtUML*

In this section we indicate how to reduce richer UML models in *rtUML* as supported in the IST project Omega [10] to the *krtUML* subset defined in Section 2. Besides, we explain the choice of the design decision behind the formal semantics.

4.1 Translating *rtUML* to *krtUML*

UML defines *associations* and *association end-points* to capture relations between classes. Semantically, association-end points maintain pointers to objects

accessible through this association end-point (subject to restrictions on visibility and navigability). Our precompilation introduces these as what we call *implicit attributes*, and translates code invoked when creating compound objects for establishing links employing a set of *implicit operations* such as '*add_to_association_end*'. Note the introduction of the special type T_{as} provided for such attributes in the *krtUML* model. In particular, pre-compilation will create implicit attributes for maintaining knowledge about all (possibly dynamically created) component objects [5] of a strong aggregation (also called composition); it will include calls for creation of component objects with bounded multiplicity in the constructor code of the aggregate object; it will contain calls for destroying every existing component object within the destructor code of the aggregate object.

Regarding generalisation of objects, we create private instances of all predecessors in the generalisation hierarchy (much as the creation of a compound objects induces creation of its components) to keep all operations and statecharts overwritten in the specialised objects. This allows to capture both static and dynamic polymorphisms using implicit attributes *uplink* and *downlink* to navigate across these instances to find e.g. the definition of operations matching a call. We do not require any restrictions on the statechart inheritance: a sub-class might have state-machine overwritten independently from that of the corresponding super-class. All private copies maintain their own object-configuration, hence e.g. accepting a triggered operation will only change the state-configuration of that state-machine corresponding to the object offering the operation in the generalisation hierarchy. The statechart inheritance described in [10] can be considered as a particular case.

Another precompilation step transfers hierarchical UML statecharts from *rtUML* to flat state-machines of *krtUML* without changing its behavior. The states in a flattened state-machine correspond to state configurations (a set of states) from the original statechart extended with the history function (keeping information for the history connectors). Besides, for the statechart of each reactive class c we add the following kinds of auxiliary states:

- One or several "*creation*" states $q_0, \ldots, q_n$ $(n \geq 0)$, where q_0 has the outgoing transition guarded by triggered operation $create_c$ and followed by the constructor code, ending with the initial state of the original (hierarchical) statechart. Only the statechart of the root-class does not contain any triggered operation at its "creation" transitions.
- A "*destruction*" state q_x with outgoing transitions containing the destructor code. Then every state in the flattened state-machine containing termination vertice (from the original statechart) has an outgoing transition to some auxiliary state without triggering guard and with action *destroy*(*self*);
- Several "*internal*" states necessary to split complex transitions, e.g. transitions containing non-primitive actions.

[5] Note the difference between a component object, specified by the composition association as a "part" of an aggregate object and used at the *rtUML* level, and the notion of component as a group of one active and several passive objects, used at the *krtUML* level

4.2 The Choice of *rtUML* Communication Scheme

Certain transformations in the pre-compilation steps are based on modelling assumptions. In this paper we only elaborate on the concept of *components* as introduced in Definition 5 (iii). When targeting distributed system-implementations of real-time systems, synchronous operation calls clearly cannot be used for component communication. Indeed, any estimation of worst-case execution time would have to cater for a waiting delay until the receiving component is able to accept a call, which itself may be blocked while awaiting serving of an operation call by yet a third component. We thus assume a modeling style, where inter-component communication is restricted to signal-based communication. To exploit this, we allow the grouping of objects into *components*; within a component, no restrictions are placed as to inter-object communication. Based on the pragmatics of active objects in UML, we mandate, that each such component-group contains exactly one active object, and allow to include an arbitrary number of passive objects in the group. Active objects are assumed to be reactive, and reactive passive objects are required to delegate their event-handling to the one active object within the group.

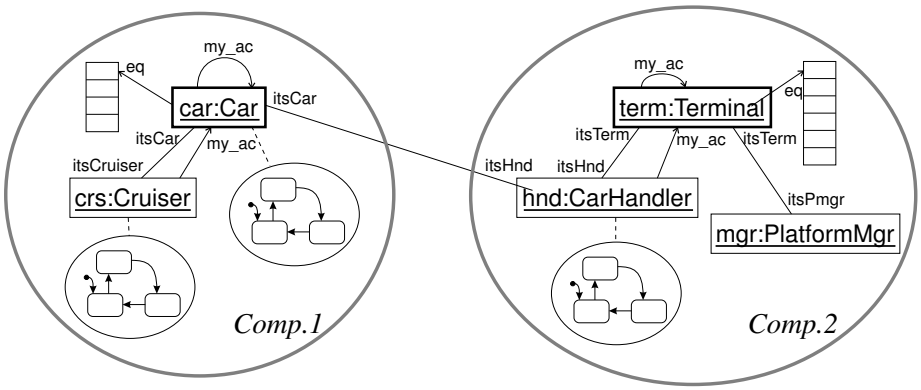


Fig. 7. UML Components: A snapshot of a model part shows active objects *car* and *term* (with their event queues) and passive objects *crs*, *hnd*, and *mgr*. Reactive objects *car*, *crs*, and *hnd* are denoted by associated schematic state-machines. Active objects *car* and *term* designate their components $Comp.1$ and $Comp.2$, respectively.

Figures 7 and 8 illustrate the concepts of components and intercomponent communication using the Automated Rail Cars System example from [15]. The graphical representation of a snapshot of a model on Figure 7 shows objects on the *krtUML* level. Each reactive object has a link to an active object via *my_ac* which is assumed to be constant for the object's lifetime. Objects referring to the same active object form a component. Figure 7 shows two components with a single link across a component-boundary. All event-handling is delegated to the component's active object, which keeps all events in its event queue. When the event has reached the top of the queue, the active object may decide to take

the event from the queue and dispatch it to the destination. This is indicated in Figure 8 by light-gray arrows. The semantics in Section 3 is explained from the perspective of the destination.

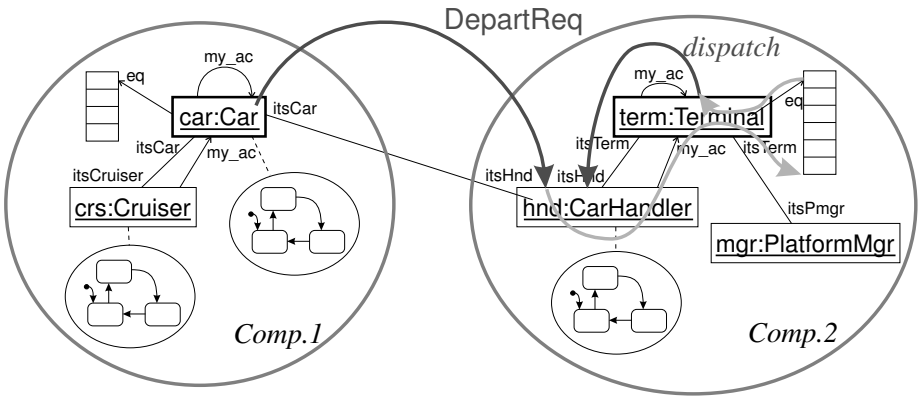


Fig. 8. Event communication: Sending an event of *DepartReq* from *car* to *hnd* in fact enters the event into the event queue of *term*, which is the active object associated with *hnd*.

The semantics enforces, that at most a single thread of control is active within one component. We feel, that deviating from this modelling paradigm, and in particular allowing multiple threads to execute within one object could easily cause modelling errors not acceptable for hard real-time applications.

5 Related Works

All attempts to define UML semantics can be classified into different orthogonal dimensions.

One direction in the semantics classification is the level of UML coverage. Many people have been trying to build the semantics of individual diagrams of the UML – e.g., [20, 4] etc. on state-machines, [12] on collaboration diagrams, [13, 16] etc. on class diagrams, [28] on use cases, [3] on activity diagrams – or just to give formal foundations for action language (e.g., [24, 2]). Because all diagrams are only views on one and the same model, the attempts to give semantics for separated UML diagrams fail in producing the right semantics for the entire UML. In our approach a symbolic transition system represents both static and dynamic aspects. The combination of statics and dynamics is also given in [29] which considers the problem of defining active classes with associated state-machines. It gives a very fine interleaving semantics for state-machines in terms of transition systems. In difference to our approach, the authors do not give precise semantics for state-machines, for event queue handling, consider a limited inheritance, and they treat only flat UML state-machines without action semantics.

Another coverage level relates to the problems with possible concurrency as well as aspects of objects communication, which have been uncovered in [29] and not addressed in the original UML 1.x documents itself. Such open problems are typical for so called *loose semantics* introduced in [16], where the aspects of concurrency and object communication are not fixed to some design decision, but cover different implementations. Such loose semantics is not suitable for formal verification. Our paper tries to overcome this problem by fixing one detailed semantics as an example of the feasibility of UML semantic for verification purposes. From other side, there are a number of UML modelling and/or verification tools implementing precise semantics by translating UML models to programming language or model checker internal formats ([17, 9, 1, 21]). These tools have different limitations on the supported UML features and do not provide formal description of the implemented semantics or it is just technical translations.

H. Hußmann [16] proposes the third dimension for the classification of attempts towards the UML formal semantics, dividing approaches into the following groups:

1) Naive set-theoretic approach. M. Richters and M. Gogolla [31] have suggested to use simple set-theoretic interpretation for UML class diagrams. In this approach, the semantics of a class diagram is described as a set of hypergraphs, corresponding to a configuration of objects. This approach has the low level of abstraction, where the concepts of UML itself are more abstract than the formal semantics given to it. This kind of semantics is mostly used for the formal definition of OCL constraints within UML models. We do not consider OCL in our approach.

2) Metamodelling semantics. This group of approaches is based on the application of a "bootstrapping" principle [7], where the semantics of UML is described using a small subset of UML as a core based on static semantics only. The approach of the pUML group to the UML semantics is given in [6, 5, 2]. Essentially, an algebraic specification is used to describe legal (local) snapshots of the system without treating actions, whereas our approach gives a formal semantics for dynamic behaviour taking into account primitive action semantics as well. The study of A. Kleppe and J. Warmer [19] is based on the pUML OO meta modelling approach. In addition, it takes into account that static and dynamic viewpoints on the system can not be separated. But the formal semantics for state-machines is not really defined, the set of primitive actions is very restrictive, and the transporting mechanism for signal inter-object communication is not specified. In our approach, we give a formal semantics for actual state-machines (not their unfolding into actions) with a larger set of primitive actions. We also resolved open problems with concurrency.

3) Translation semantics. An approach, which tries to keep the right abstraction level, defines translation from UML class diagrams to traditional specification languages (Z [14], Object-Z [18], CASL [30] etc.). For example, G. Reggio et al. [30] proposed a general schema of the UML semantics by using an extension of the algebraic language CASL for describing individual diagrams (class diagrams and state-machines) and then their semantics are composed to get the

semantics of the overall model. Also other UML diagram types have been translated to formal notations, e.g., using Abstract State Machines ([4, 3, 23, 8]). E. Börger et al. [4] defined the dynamic semantics of UML in terms of ASM extended by new construct to cover UML state-machine features. The model covers the event-handling and the run-to-completion step, and formalises object interaction by combining control and data flow features. However, the authors did not give a complete solution to solve transition conflicts and it is not clear how firable transitions are selected. The semantics implemented by UML-tool vendors via code generation or model simulation can be also classified to this group of approaches (among other: [17, 9, 1]).

Differently from these approaches, our study provides one formalism (STS) for both static and dynamic semantics, which also contains action language.

4) Combination of the approaches mentioned above. An example of combination of several approaches can be found in [23]. In this research, static semantics is defined using meta-modelling mechanism of UML, the execution semantics is expressed as ASM programs. The study covers all features contained in the class diagrams, and in the body of the operations (quite thorough set of action types). The aspects of inter-object communications were not really covered and the semantics of UML statecharts was not addressed, although it can be accompanied by the complementary papers [3] and [4]. But these articles consider state-machines separated from the rest of UML, whereas our approach provides one semantics for class diagrams and statecharts.

6 Conclusion

With respect to the approaches sketched above, the main novelty of our approach is that it resolves uncovered problems with concurrency and object communication by giving a formal semantics for a chosen concrete decision. W. Damm and B. Westphal [11] have shown that this semantics can be used for formal verification. In our approach we allow that both active and passive objects can be reactive, thus considering event communication between all objects. We also capture two different kinds of inter-object communication – synchronous (via triggered operation calls) and asynchronous (via signal events).

Thus, we have provided the semantical foundation for a rich sublanguage of UML which is expressive enough to deal with industrial UML models for real-time applications. Our partners from Verimag have proposed extensions of the semantical model focussed on real-time, in particular taking into account the need to support annotations for real-time scheduling. Ongoing work within Omega builds on the semantical foundation layed down in this paper to develop a verification environment for real-time UML.

7 Acknowledgements

We gratefully acknowledge the contribution of our Omega partners in finetuning the semantics.

References

[1] Telelogic AB. Telelogic Tau, 2003. http://www.telelogic.com/products/tau/index.cfm.

[2] J.M. Alvarez, T. Clark, A. Evans, and P. Sammut. An Action Semantics for MML. In *Proc. UML 2001*, 2001. http://www.cs.york.ac.uk/puml/mmf/AlvarezUML2001.pdf.

[3] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. AMAST 2000*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.

[4] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich, Ph.W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Proceedings*, volume 1912 of *LNCS*, pages 223–241. Springer-Verlag, 2000. DBLP, http://dblp.uni-trier.de.

[5] T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In *Proc. FASE 2001*, pages 17–31, 2001. www.dcs.kcl.ac.uk/staff/tony/docs/MMLCalculus.ps.

[6] T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages Using a Precise OO Meta-Modelling Approach, version 1.0, September 2000. Available from http://www.puml.org.

[7] T. Clark, A. Evans, S. Kent, and P. Sammut. The MMF Approach to Engineering Object-Oriented Design Languages. In *Proc. Workshop on Language Descriptions, Tools and Applications (LDTA2001)*, 2001. Available via http://www.puml.org.

[8] K. Compton, J. Huggins, and W. Shen. A Semantic Model for the State Machine in the UML. In G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors, *Dynamic Behaviour in UML Models: Semantic Questions, Workshop Proceedings, UML 2000 Workshop*, Bericht 0006, pages 25–31. Ludwig-Maximilians-Universitt Mnchen, Institut fr Informatik, October 2000. http://www.kettering.edu/∼jhuggins/papers/uml2000.ps.

[9] Rational Software Corporation. Rational Rose Family, 2003. http://www.rational.com/products/rose/index.jsp.

[10] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. A Formal Semantics for a UML Kernel Language. Omega Technical report, part 1 of the deliverable D1.1.2, Project IST-2001-33522 OMEGA, January 2003. Available from http://www-omega.imag.fr/doc/d1000009_6/D112_KL.pdf.

[11] W. Damm and B. Westphal. Live and Let Die: LSC-based Verification of UML-Models. In *Proceedings FMCO'02*, 2003. (to appear).

[12] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceed. of the 3d International Conference on the UML 2000*, October 2000.

[13] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In *The Unifed Modeling Language: the first international workshop, June 1998*. Springer-Verlag, 1999.

[14] A.S. Evans and A.N. Clark. Foundations of the Unified Modeling Language. In *2nd Northern Formal Methods Workshop, Ilkley, electronic Workshops in Computing*. Springer-Verlag, 1998. http://www.cs.york.ac.uk/puml/papers/nfmw97.ps.

[15] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[16] H. Hußmann. Loose Semantics for UML, OCL. In *Proceedings 6th World Conference on Integrated Design and Process Technology (IDPT 2002)*. Society for Design and Process Science, June 2002.

[17] I-Logix Inc. Rhapsody, 2002. http://www.ilogix.com/products/rhapsody/index.cfm.

[18] S.-K. Kim and D. Carrington. Formalizing the UML Class Diagrams Using Object-Z. In France and Rumpe, editors, *Proc. UML'99*, volume 1723 of *LNCS*, pages 83–98. Springer-Verlag, 1999.

[19] A. Kleppe and J. Warmer. Unification of Static and Dynamic Semantics of UML, 2001. http://www.klasse.nl/english/uml/unification-report.pdf.

[20] G. Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statcharts. In *Proceed. of the 3d International Conference on the UML 2000, University of York*, October 2000.

[21] J. Lilius and I.P. Paltor. *vUML: a Tool for Verifying UML Models*. Turku Centre for Computer Science, Abo Akademi University, Finland. Technical Report.

[22] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[23] I. Ober. *Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics, PhD Thesis*. Institut National Polytechnique de Toulouse, France, April 2001.

[24] Object Management Group. *UML 1.4 with Action Semantics, Final Adopted Specification, ptc/02-01-09*, January 2002. Available from http://www.kc.com/as_site/home.html.

[25] Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification, ptc/02-03-02 OMG Adopted Specification*, March 2002. Available from http://cgi.omg.org/docs/ptc/02-03-02.pdf.

[26] G. Övergaard. Formal Specification of Object-Oriented Meta-Modelling. In T.Maibaum, editor, *Proceedings Fundamental Approaches to Software Engineering, FASE*, number 1783 in LNCS. Springer-Verlag, 2000.

[27] G. Övergaard. Using the BOOM Framework for Formal Specification of the UML. In *Proceedings of Defining Precise Semantics for UML*, 2000.

[28] G. Övergaard and K. Palmkvist. A Formal Approach to Use Cases and Their Relationships. In *UML 1998*, 1998.

[29] G. Reggio, E. Astesiano, C. Choppy, and H. Hußmann. Analyzing UML Active Classes and Associated State Machines - A Lightweight Formal Approach. In *FEAS 2000*, 2000. ftp://ftp.disi.unige.it/pub/person/ReggioG/Reggio99a.ps.

[30] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting Its Multiview Approach. In *FASE 2001*, 2001. ftp://ftp.disi.unige.it/pub/person/CerioliM/FASE2001.pdf.

[31] M. Richters and M.Gogolla. On Formalizing the UML Object Constraint Language OCL. In T.-W. Ling, S. Ram, and M.L. Lee, editors, *Proc. 17th International Conference Conceptual Modelling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer-Verlag, 1998.