Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 55 (2005) 81–115
Science of Computer Programming
www.elsevier.com/locate/scico

A discrete-time UML semantics for concurrency and communication in safety-critical applications☆

Werner Damm[a,*], Bernhard Josko[a], Amir Pnueli[b], Angelika Votintseva[a]

[a]*OFFIS, Oldenburg, Germany*
[b]*The Weizmann Institute of Science, Rehovot, Israel*

Received 31 August 2003; received in revised form 15 April 2004; accepted 30 May 2004

Abstract

We define a subset *krtUML* of UML which is rich enough to express such modelling entities of UML, used in real-time applications, as active objects, dynamic object creation and destruction, dynamically changing communication topologies, combinations of synchronous and asynchronous communication, and shared memory usage through object 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 briefly outline how to compile models of industrial systems making use of generalisation hierarchies, weak and strong aggregation, and hierarchical state-machines into *krtUML*. The main aim of the paper is to provide an executable semantics for *krtUML* suitable for the formal verification of temporal model properties with existing model-checking tools.

☆ This research was partially supported by the Information Society DG of the European Commission within the project IST-2001-33522 OMEGA.
* Corresponding author.
E-mail addresses: damm@offis.de (W. Damm), josko@offis.de (B. Josko), amir@wisdom.weizmann.ac.il (A. Pnueli), votintseva@offis.de (A. Votintseva).

0167-6423/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.scico.2004.05.012

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 modelling constructs employed, as a prerequisite for rigorous 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. The precise UML group has in a series of papers [4–6] 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, e.g. the π-calculus [27,28], ASMs [23], CASL [32], Object-Z [17], fall short of covering the rich range of behavioural modelling constructs covered in this paper. Other approaches to the UML semantics are discussed in detail in Section 5 of this paper. Closest to our work addressing the intricacies of understanding active objects are [31,32].

Our approach takes into account functional aspects of real-time systems, considering a discrete-time model allowing us to define different levels of step granularity. In this paper, we focus our investigation on the semantic foundation of such critical features of real-time applications as concurrency and two types of inter-object communication — synchronous and asynchronous — including the specification of the time points for interferences. The proposed semantics, being executable and abstract enough to cover different choices for the final implementation and deployment (such as different execution times, scheduling strategy), is intended for the formal verification at earlier stages of the development process, such as preliminary and detailed design. Such "early" verification would allow us to find errors of possible further implementations already at the model level.

The approach described 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 [8], 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. This project also provides a real-time extension of the proposed semantics [13].

[1] IST-2001-33522, http://www-omega.imag.fr/index.php.

[2] IST-1999-10069, http://wooddes.intranet.gr.

We propose a two-stage approach to give a formal semantics to *rtUML*:
A pre-compilation step translates *rtUML* models into a sufficiently compact sublanguage *krtUML*, eliminating the need at the kernel level to address the various facets of associations, generalisation, and hierarchical state-machines. We then give a formal semantics of *krtUML*, using the formalism of symbolic transition systems [21]. 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 its state-machine, as well as the pending events collected in an event queue.

In this paper, we focus on the definition and formal semantics of *krtUML*, and only sketch some ideas of the pre-compilation phase, because most of the translation steps use standard compiler techniques. We refer the reader to [8] for a full description of these steps, as well as for the full specification of *rtUML*.

The paper is organised as follows. Section 2 outlines the aims for the semantics proposed in the paper and 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 class relations and the hierarchical state-machine. Section 5 discusses related work.

2. The *krtUML* language

Our kernel language caters for the difference between active and passive objects. We generalise this concept in Section 4 by proposing to group one active object and a collection of passive server objects into what we call *components*. Another class dichotomy, orthogonal to the "active–passive" hierarchy, considered in the paper is the difference between reactive and simple classes. All objects are assumed to have state-machines; that is, their behaviour can be made dependent on the current state of the system. Some state-machines can specify event receptions, which automatically implies a reactive behaviour of the corresponding class, i.e. its objects can react on the external stimuli. We do not require any restrictions on the combinations between active/passive and reactive/simple class notions.

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.

2.1. Basic notions

We first explain some UML related notions considered in the paper, as well as imposed problems, when resolving ambiguity of semantic variation points deliberately left in

the UML specifications. We use the notions of active class/object, thread (of control), concurrency, multiplicity, state-machine, association, composition, generalisation, multiple inheritance, dynamic classification, stimulus, signal, event, sender and receiver, method, parameter as they are defined in the UML 2.0 proposal [26].
In developing *krtUML*, we strived to maintain in purified form those ingredients of UML relating to the interaction of active objects.

Active classes are intended to be used to model threads — sequential executions — where all threads can run concurrently. Active classes provide means to sequentialise (independent) executions. Intuitively, an *active object* — 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 objects associated with this active object, inducing a so-called run-to-completion step.

Passive classes are those containing no scheduling (or sequentialisation) mechanisms. Their instances — *passive objects* — use such mechanisms from the assigned active objects. In other words, passive objects perform their services *on behalf* of the corresponding active ones.
Components. In this paper, we use the notion of a component which is a restriction of the more general concept from the standard UML. We will call a set of objects executing their services sequentially a component. This means that each component contains exactly one active object and possibly several passive ones associated with the active one. Within a component, all passive objects delegate event-handling to the one active object; pre-compilation will capture this delegation relation by allowing reference through *my_ac* to the active object responsible for event-handling of passive objects. We require static assignment of passive objects to active ones, such that an object can belong only to one component in its life-cycle.

A *Run-To-Completion (RTC) step* is a sequence of fired transitions in an object state-machine corresponding to the processing of a single event or operation call. An RTC step cannot be interrupted. Only RTC steps from different components can run concurrently (in our semantics, meaning all possible interleavings).

Semantic challenge. A problem for the semantic definition for concurrent executions, solved in the paper:

- on one hand, to take into account the different execution speeds within different components (executing concurrently and asynchronously),
- on the other hand, to find an abstraction from the actual execution durations (which can be different on different platforms),
- providing a semantics allowing telling about both state and run (or temporal) properties of complex systems.

Signals are specifications of asynchronous stimuli, whose reception is handled by state-machines. There can be several signal instances (called signal events) in a system at one point of time. Signals can be generalised, which means that if a state-machine can handle a reception of a generalised signal event, then it can also handle a more specialised event, but not vice versa.

Operations. 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 their methods into state-machine transitions (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 labelling transitions emerging from the state with the operation name as a triggering guard, in the same way as the willingness of the object to react to a given signal event is specified by using this signal as a triggering guard. Reflecting the wish to make the return value of triggered operations dependent on the object state, 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.

A general characteristic of reactive classes in UML is that they contain state-machines specifying reactions on the stimuli by changing their states. This reaction can also depend on the current state in the state-machine. In this article we propose a semantics, where executions are defined with respect to transitions of state-machines, where object creation and destruction are also explained in terms of (implicit) state-machine transitions. Therefore, in *krtUML* all classes have state-machines. We will define a slightly different notion of a reactive class to capture the proper reactive behaviour as follows.

A *reactive class* in *krtUML* is a class whose state-machine specifies event receptions or operation acceptance also after the initialisation phase, i.e. when the state-machine execution triggered by the creation operation is completed. Otherwise it is called a *simple class*.

We consider two types of the intra- and inter-object communication:

- Asynchronous — via signal event emission. The caller does not need any reply; therefore it proceeds further after the emission of a signal event. All emitted events need to be stored in additional repositories to be accepted later by callees.
- Synchronous — via operation calls. In *krtUML* we consider only triggered operations, which trigger state-machine transitions. A caller sends a request that it wants to synchronise with its callee (possibly to get a result of an operation) and becomes suspended. The callee may accept the call, if it enters the corresponding state.

Semantic challenge. A problem for the semantic definition of models with the combination of different kinds of communications, solved in the paper:

- on one hand, to distinguish semantically synchronous and asynchronous communications by treating them differently,
- on the other hand, to give a uniform state-machine-based semantics (also taking into account communication structure from class diagrams),
- providing a suitable granularity for the interference of object executions to capture properties of both synchronous and asynchronous communication schemes in complex systems.

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.

In the following (sub)sections we will give formal definitions of the above-mentioned notions with respect to *krtUML*. The notion of components will be also considered in Section 4.2 at a higher level of modelling formalism, called *rtUML*.

2.2. krtUML structure

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. Essentially a kernel model contains a set of classes and signals; signals can be ordered by the generalisation relation, with each class containing a state-machine, typed attributes, and operations implemented via the class state-machine. Some classes are distinguished as being active. We only consider here flat state-machines extended with object initialisation and object destruction phases. A designated root class serves later for the system initialisation.

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* from $c.attr$ 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 clearly 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

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

where $T_C \stackrel{df}{=} \{T_c \mid c \in C\}$ is the set of *reference types* and
$T_{as} \stackrel{df}{=} \{\mathbb{N} \to T_c \mid c \in C\}$ the set of *association types*, which will be used to represent all kinds of associations described in [8] (i.e., composition, aggregation, and neighbour).

For each type $\tau \in T(M)$, we assume the 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* ($type_r(op) = \texttt{void}$ 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 or primitive 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 $type(expr) \stackrel{df}{=} type(a)$. 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 $type(expr) \stackrel{df}{=} T_{c'}$.
- *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 values of triggered operations.

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 labels of transitions in the class state-machines. 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, having no incoming transitions, is the unique point of entry into a section added by pre-compilation modelling destructor code. Pre-compilation also transfers hierarchical state-machines into flat state-machines.

Definition 1 (*Continued*).

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

$$c.sm = (c.Q, c.q_0, c.q_x, c.tr), \text{ where:}$$

- $c.Q$ 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.
- $c.tr \subseteq c.Q \times (\{\gamma \mid \gamma \text{ is a } c\text{-guard or } c\text{-action}\}) \times c.Q$ is the *transition relation*. We require that there is the initial transition $(c.q_0, \gamma, q) \in c.tr$ with c-action $\gamma =$ "$create_c$".
- Class $c \in C$ is called *reactive* if there is a transition $(q, \gamma, q') \in c.tr$ such that $q \neq c.q_0$ and γ is in the form $ev[expr]$ or $op[expr]$ for some $ev \in Sig$ or $op \in c.ops \setminus \{create_c\}$. □

We will use *krtUML* to denote the set of all *krtUML* models.
An abstract example of a *krtUML* model with four classes is shown on Fig. 1.

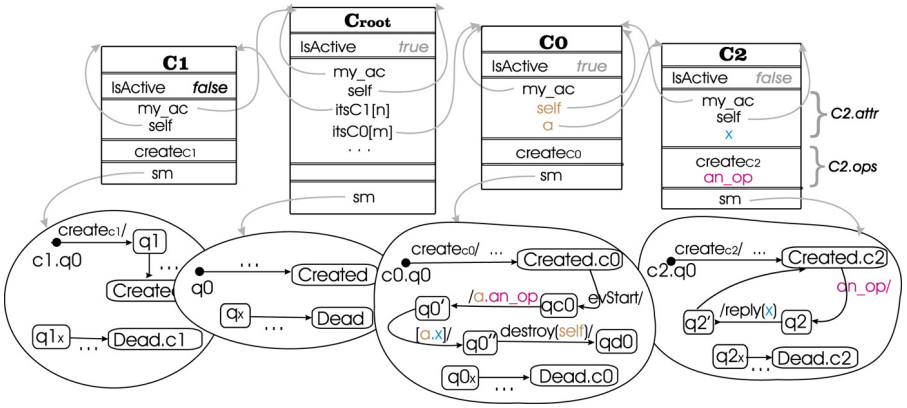


Fig. 1. Class examples. Classes C_{root} and $C0$ are active, whereas $C1$ and $C2$ are passive, i.e. perform their services within the sequences of C_{root} and $C0$ executions, respectively. Classes $C0$ and $C2$ are reactive, since they can react to stimuli after the initialisation phase. Classes C_{root} and $C1$ do not accept any stimuli other than creation.

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 implicit attributes $parent\_type_{c'}$ and *type_table* for each superclass c' of c in the preprocessing step described in Section 4.1. Association attributes $parent\_type_{c'}$ are used to keep the structure of the inheritance hierarchy, whereas *type_table* reflects the actual type of each object, which is available at each level of the dynamic classification (useful, e.g., for calls of abstract operations with a deferred implementation [22]).

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 classes. But the information on 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 Section 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* (as well as *type_table* and $parent\_type_c$) are assigned once at the initialisation of the corresponding object and do not change during the lifetime of the object.

(iv) For each triggered operation $op \in c.ops$, $c \in C$, there are attributes $c::op_{p_i} \in c.attr$, $1 \leq i \leq n$ for holding 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 $c::ev_{p_i} \in c.attr$, $1 \leq i \leq n$, for holding 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 [21] 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 in which snapshots can evolve, defining for each of the possible cases a transition predicate. Finally, we define the predicate characterising 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 allows 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 characterising 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',

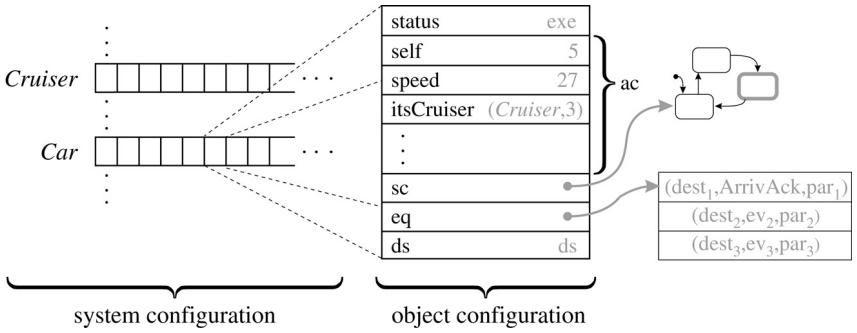


Fig. 2. System configuration. A variable of type $\mathcal{T}_{sconf}$ contains one object configuration for every object identifier in O_C. The example of an object configuration *oconf* for the object (*Car*, 5) is shown enlarged.

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.
- *Consecutiveness:* 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

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

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 [14], 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 [14] for details.

Fig. 2 depicts the way in which 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 configuration 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)

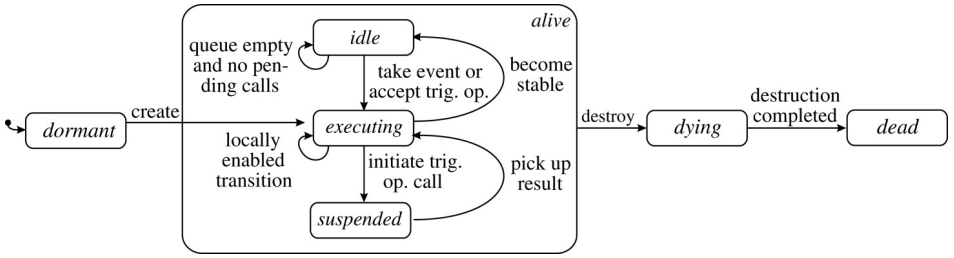


Fig. 3. The object life-cycle.

array of object configurations, with each entry corresponding to a single instance of this class.

The object status reflects the phase in the object life-cycle (see Fig. 3). 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 us to observe nastiness such as sending events to dying objects, as well as detecting dangling references.

For the rest of this 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 $\mathcal{D}_{O_c} \stackrel{df}{=} \{c\} \times \mathbb{N}$ as its domain. We call O_C with domain $\mathcal{D}_{O_C} \stackrel{df}{=} \bigcup_{c \in C} \mathcal{D}_{O_c}$, the *object reference type or domain*. For each object type O_c, we assume the existence of a designated element $\mathsf{nil}_c \in \mathcal{D}_{O_c}$ to serve as a default value.

For each association type $\tau = (\mathbb{N} \to T_c) \in T_{as}$, $\mathcal{D}_\tau \stackrel{df}{=} (\mathbb{N} \to \mathcal{D}_{O_c})$ 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
$$\mathcal{D}_{\mathcal{T}_{objstatus}} \stackrel{df}{=} \{dormant, idle, executing, suspended, dying, dead\}.$$
- An *object attribute configuration* '*ac*' of type $\mathcal{T}_{ac} \stackrel{df}{=} \bigcup_{c \in C} (c.attr \to \mathcal{T}_{T(M)})$.
- An *object state-machine configuration* '*sc*' of type $\mathcal{T}_{sc}$ with associated semantic domain $\mathcal{D}_{\mathcal{T}_{sc}} \stackrel{df}{=} \bigcup_{c \in C} c.Q$.
- The *event queue eq* of type $\mathcal{T}_{eq} \stackrel{df}{=} \mathcal{T}^*_{eqe}$, i.e. a sequence of entries $(dest, ev, par)$ of type $\mathcal{T}_{eqe} \stackrel{df}{=} O_C \times Sig \times \bigcup_{ev \in Sig} \mathcal{T}_{type_{par}(ev)}$.
  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 $\mathcal{T}_{ds} \stackrel{df}{=} O_C$, 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

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

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

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

(iv) We will call object $o \in O_c$ of class c an *active object* iff $c.isActive = true$ (i.e., c is an active class). Otherwise we call o a *passive object*. We also will write $o.isActive = true$ to specify that o is an active object and $o.isActive = false$ for passive ones.

(v) We will call object $o \in O_c$ of class c a *reactive object* iff c is a reactive class. Otherwise we call o a *simple object*. □

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 [8], a default assignment is given derived from the composition association.

We collect the status of all pending operation calls within a pending request table. An example in Fig. 4 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 or 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.

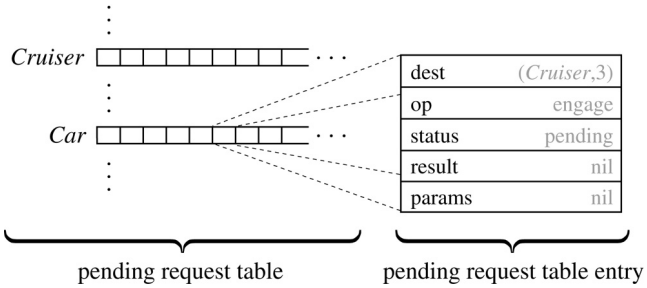


Fig. 4. The pending request table. The pending request table is a system variable of type $\mathcal{T}_{prt}$. It contains one entry for every object identifier in O_C.

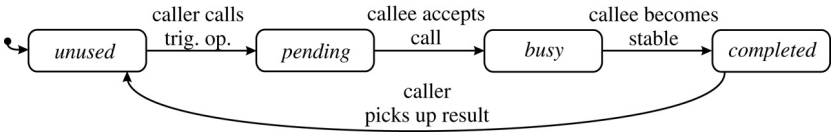


Fig. 5. The life-cycle of a triggered operation call.

The life-cycle of an entry in the pending request table is depicted in Fig. 5. 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.

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

Thus the type of a pending request table entry is

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

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

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

For each type τ considered, we assume the existence of a designated element $\mathsf{nil}_\tau \in \mathcal{D}_\tau$ to serve as a default, or undefined, value. Moreover, we assume that expressions *expr* are evaluated to $\perp$ in such situations as, for example, trying to read an attribute via a reference with value nil, or trying to execute division by 0 and other arithmetic exception situations. In other words, $[\![expr = \perp]\!](s) = true$ iff $[\![expr]\!](s) = \mathsf{nil}_\tau$ for $\tau = type(expr)$.

Furthermore, we need a boolean flag *sysfail*, which is used to indicate an undefined state of the system, e.g., if it tries to read an 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 has changed to *true*.

For brevity, we will use the following abbreviations for $o \in O_C$ in the rest of this section:

- $o.status \stackrel{df}{\equiv} sconf(o).status$ and analogously for *sc*, *ds*, *eq*.
- $o.a \stackrel{df}{\equiv} sconf(o)(a)$, i.e. the value of attribute a.
- $o.a.b \stackrel{df}{\equiv} sconf(sconf(o)(a))(b)$, 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 $q = e_1 \ldots e_n \in \mathcal{D}_{\mathcal{T}_{eq}}$ we introduce the following elements:

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

We will use logical XOR-operator for the following abbreviation: $a \oplus b \stackrel{df}{=} (a \vee b) \wedge \neg(a \wedge b)$.

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 for 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, which can be either simple (taking only one fine step with no changes in the flow of control) or delayed, waiting for the results from other objects.
- 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* and all event queues are empty. In the following, we formalise each of the above conditions separately as first-order logic predicates which are then used to construct the transition predicate of the semantics $STS(M)$.

Note that in the following incremental definition of the transition predicate, we use an assignment 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 first define for each object $o \in O_c$ the predicate $stable(o)$ in the current system configuration as follows:[3]

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

We will define the individual steps that an object can perform, thus defining the transitions locally to objects. Later, in Definition 7, the global transition predicate is combined out of these steps with additional conditions specifying a kind of "scheduling". Each such "partial" predicate, defined below for each kind of step, contains the following specifications:

(a) the state when the step can be performed: conditions on the current, i.e. unprimed, values of the system variables;
(b) changes in the values of object attributes or the pending request table induced by the transition;
(c) raising the failure flag if some values referred to are undefined.

[3] 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 $[expr]$, respectively), i.e. a signal trigger (a call trigger or a condition, respectively; cf. Definition 1 (iii)).

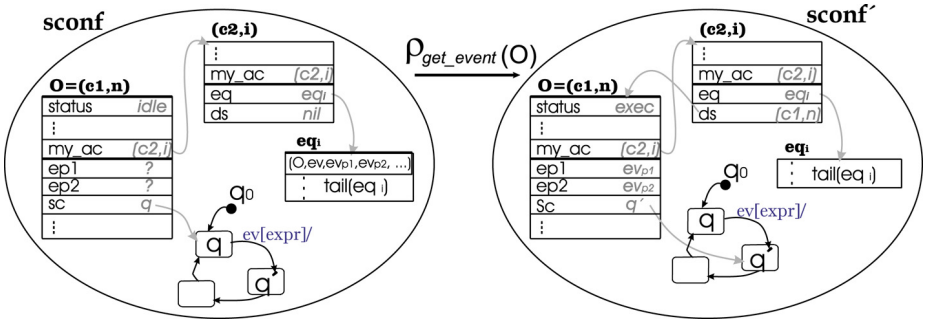


Fig. 6. The transition relation: ρ_{get_event}.

3.3.1. Getting an event

Intuitively, an event ev_1 with destination o can be dispatched to o from the head of the event queue of its active object if no other object in the same component is currently processing an event reception (specified by $o.my\_ac.ds = \mathsf{nil}$) and if there is a transition (q, γ, q') guarded by a superclass ev of ev_1 is enabled in the current state q (cf. Fig. 6):

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

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. a 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*.

3.3.2. Accepting a triggered operation

Object o can accept a triggered operation call op if a transition (q, γ, q') guarded by op is enabled in the current state q and some other object o_1 has called op from o (there is an entry point in the pending request table with this operation):

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

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 $prt(o).dest$ and $prt(o).status$).

3.3.3. Skipping guards

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

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

3.3.4. 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 (or its generalisation) is enabled, then the event is simply removed from the top of the queue:

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

Note that a discarded signal event can nevertheless trigger a transition, if the object is no longer in its stable state (the value of a guarding condition on a transition without signal or call trigger became true). Note also that triggered operation calls are not discarded, but remain until the callee accepts the call.

3.3.5. Executing simple actions

Object o can execute an action if the current transition (q, γ, q') is enabled and annotated with the action. We distinguish two types of action — simple (or non-operation) actions and operation calls (or synchronisation delays) — treating them in different ways.

These subformulas will be combined with different contexts — conditions on their performance — in the final transition predicate. There are four kinds of non-operation action:

$$\rho_{non\_op\_action}(o) \stackrel{df}{=} \rho_{assign}(o) \oplus \rho_{send\_event}(o) \oplus \rho_{reply}(o) \oplus \rho_{destroy}(o).$$

- An assignment action simply assigns a value to the destination attribute:

$$\begin{aligned}\rho_{assign}(o) \stackrel{df}{=}\ & \gamma \equiv \text{“}r.a := expr\text{”} \wedge (expr \neq \bot \implies o.r.a' := expr) \\ & \wedge sysfail' := (sysfail \vee o.r = \mathsf{nil} \vee expr = \bot).\end{aligned}$$

- An event-sending action causes a new event to be appended to the queue of the destination's active object:

$$\begin{aligned}\rho_{send\_event}(o) \stackrel{df}{=} & \gamma \equiv \text{“}r.send(ev, expr_1, \ldots, expr_n)\text{”} \\ & \wedge sysfail' := \left(sysfail \vee o.r = \mathsf{nil} \vee \bigvee_{i=0}^{n} expr_i = \bot\right) \\ & \wedge o.r.my\_ac.eq' := \\ & \quad enqueue(o.r.my\_ac.eq, (o.r, ev, (expr_1, \ldots, expr_n))).\end{aligned}$$

- 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:

$$\begin{aligned}\rho_{reply}(o) \stackrel{df}{=} & \gamma \equiv \text{“}reply_\tau(expr)\text{”} \\ & \wedge [(\exists\, o_1 \in O_C : \\ & \quad prt(o_1).dest = o \wedge prt(o_1).status = busy \\ & \quad \implies prt(o_1).result' := expr \\ & \wedge sysfail' := (sysfail \vee \tau \neq type_r(o_1) \vee expr = \bot)) \\ & \quad \oplus sysfail' := true].\end{aligned}$$

- A destroy action causes the destination's state-machine configuration to be changed, so 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:

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

3.3.6. Synchronisations via operation calls

Operation call actions differ from the just defined simple actions, because they are not treated atomically. An operation call suspends object o and configures o's entry of the pending request table, so that it denotes the callee, the called triggered operation, and the parameters. Initially the status of a called operation is "*pending*". Besides this, an additional check is performed to guarantee operation calls only from the same component (otherwise a run-time failure is observed).

$$\begin{aligned}\rho_{init\_opcall}(o) \stackrel{df}{=} &(\gamma \equiv \text{“}r.call(op, expr_1, \ldots, expr_n)\text{”}\\ &\quad \vee \gamma \equiv \text{“}r_1.a := r.call(op, expr_1, \ldots, expr_n)\text{”})\\ &\wedge o.r.my\_ac = o.my\_ac\\ &\wedge sysfail' := \Bigg(sysfail \vee o.r = \mathsf{nil} \vee \bigvee_{i=1}^{n} expr_i = \bot\\ &\quad \vee o.r.my\_ac \neq o.my\_ac\Bigg)\\ &\wedge o.status' := suspended \wedge prt(o).dest' := o.r\\ &\wedge prt(o).op' := op \wedge prt(o).status' := pending\\ &\wedge prt(o).result' := \mathsf{nil} \wedge prt(o).op'_p := (expr_1, \ldots, expr_n).\end{aligned}$$

3.3.7. Creating a new object

A creation action is handled like a triggered operation since the caller should be blocked until an object of the desired class is readily created with all inherited parts and all aggregated parts (possibly with attribute initialisation). This is modelled by implicit operations $create_c$ attended at the pre-compilation step to the initial transitions of all state-machines. A creation action looks for a dormant object, wakes it up, assigns it to an appropriate component (or creates it as a new component, if the created object is active), and then calls operation $create_c$ from the new object:

$$\begin{aligned}\rho_{call\_create}(o) \stackrel{df}{=} &\gamma \equiv \text{“}r.a := create_{c_1}(expr)\text{”} \wedge\ o.my\_ac = expr\\ &\wedge sysfail' := (sysfail \vee expr = \bot\ \vee o.my\_ac \neq expr\\ &\quad \vee (expr = o_1 \in O_C\\ &\qquad \wedge o_1.status \in \{dormant, dying, dead\}))\\ &\wedge (\exists\, o_1 \in O_{c_1} \setminus \{\mathsf{nil}_{c_1}\} :\\ &\qquad o_1.status = dormant \wedge o_1.status' := idle\\ &\qquad \wedge (\neg c_1.isActive \implies o_1.my\_ac' := expr)\\ &\qquad \wedge (c_1.isActive \implies o_1.my\_ac' := o_1)\\ &\qquad \wedge o.r.a' := o_1 \wedge o.status' := suspended\\ &\qquad \wedge prt(o).dest' := o_1 \wedge prt(o).op' := create_{c_1}\\ &\qquad \wedge prt(o).status' := pending\\ &\qquad \wedge prt(o).result' := \mathsf{nil} \wedge prt(o).params' := \mathsf{nil}).\end{aligned}$$

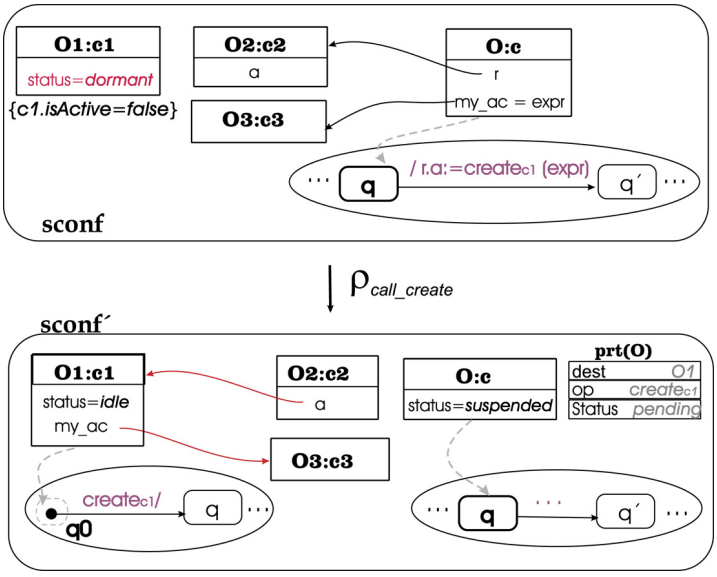


Fig. 7. The call of an object creation. Changing status of the caller O, callee $O1$, and inserting the called operation $create_{c1}$ into the pending request table: the values of selected elements from *sconf* and *prt*.

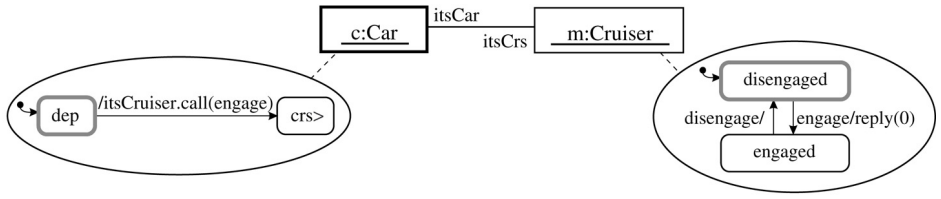
An example of some essential changes in the system configuration corresponding to a creation action is shown in Fig. 7. Here, only the creation of a passive object is shown. Note that newly created passive objects can be assigned only to the current component (defined in the corresponding attribute *my_ac*), within which the creation action has been called.

When a synchronisation took place and the callee completed the required operation, additional bookkeeping was needed at the end of the operation execution to raise the flag "the result is ready", formalised in the following subsection.

3.3.8. 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 let the caller know that the operation has been completed. In both cases, o becomes idle. If o is executing the run-to-completion step starting at q_x, then it becomes dead:

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



	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. 8. The 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*.

3.3.9. Picking up a result

Object o can pick up the result of a previous operation call if the callee has set the status of o's pending request table entry to "*completed*". Picking up a result means not only the change of the corresponding attribute, but also changes of caller's status and removing the corresponding entry in the pending request table:

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

The complete execution of an example of a triggered operation *engage*() is illustrated in Fig. 8. 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 Section 3.2) with the initial condition and combined transition relation specified in the following definition.

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

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

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

Initial condition. Initially a single object of class c_{root} exists and has status "*executing*". All other objects are dormant, all attributes have default values, there is no system failure, and there are no entries in the pending request table:

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

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 and additional conditions on their application within objects' life-cycles as follows:

$$
\begin{aligned}
\rho_0 \stackrel{df}{=}\ & \forall\, o \in O_C : o.status \neq executing \wedge o.eq = \varepsilon \\
& \vee \big(\neg sysfail \wedge \exists\, o = (c, n) \in O_C\ \exists\, (q, \gamma, q') \in c.tr : \\
& \quad o.sc = q \wedge \big(o.sc' := q' \wedge (\\
& \qquad [o.status = idle \wedge (\rho_{get\_event}(o) \oplus \rho_{accept\_op}(o))] \\
& \qquad \vee [(o.status = executing \vee o.status = dying) \\
& \qquad\quad \wedge (\rho_{skip\_guard}(o) \oplus \rho_{non\_op\_action}(o))] \\
& \qquad \vee [o.status = suspended \wedge \rho_{pick\_up\_result}(o)]) \\
& \qquad \wedge \rho_{becoming\_stable}(o)\big) \\
& \quad \vee \big(o.sc' := o.sc \wedge ([o.status = idle \wedge \rho_{discard\_event}(o)] \\
& \qquad \vee\ [o.status = executing \wedge (\rho_{init\_opcall}(o) \oplus \rho_{call\_create}(o))])\big)\big).
\end{aligned}
$$

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 (starting at Θ). □

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 definition of transitions in $STS(M)$ uses a very refined notion of step. The following definition formalises two coarser levels of steps in a complex system.

Definition 8 (*Run-to-Completion Step*). Let $V_0, \ldots, V_i \in \Sigma$ be snapshots of $STS(M)$.

(i) A *run-to-completion — RTC — step in an object o* is a subsequence $rtc(o) = (V_i \ldots V_{i+k})$ $(k > 0)$ of a run $r = (V_0 \ldots V_i \ldots V_{i+k} \ldots) \in runs(STS(M))$ such that all the following conditions hold:
- $sconf_i(o) \neq sconf_{i+k}(o)$ (changes required in the object o's configuration during the step);
- $sconf_i(o).sc = q_x \vee sconf_i(o).status = idle$ (object o was stable);
- $sconf_{i+k}(o).status \in \{idle, dead\}$ (object o became stable after the step);
- $\forall 0 < j < k : sconf_{i+j}(o).status \in \{executing, suspended, dying\}$ (object o is unstable during the step execution).

(ii) An *RTC step in a component* $Cm(o)$ is a subsequence $RTC(Cm(o)) = (V_i \ldots V_{i+k})$ $(k > 0)$ of a run such that all the following conditions hold:
- $sconf_i(o).ds = \mathsf{nil} = sconf_{i+k}(o).ds$ (no object is scheduled for an event reception at the beginning and at the end of the step);
- $sconf_i(o') \neq sconf_{i+k}(o')$ for some $o' \in Cm(o)$ (changes required in the component during the step);
- $sconf_i(o).status = idle$ (the active object of the component was stable)
- $sconf_{i+k}(o).status \in \{idle, dead\}$ (the active object of the component became stable after the step)
- $\forall 0 < j < k : (sconf_{i+j}(o).status \in \{executing, dying\} \vee sconf_{i+j}(o).ds \neq \mathsf{nil})$ (the active object of the component either was performing its own computation or scheduled a reception of an event to its passive servant).

(iii) For an object/component RTC step $(V_i \ldots V_{i+k})$, V_i is called the beginning of the RTC step, and V_{i+k} is called the end of the RTC step.

The relation between the notion of an object RTC step and a component RTC step in the proposed semantics can be formalised as the following proposition.

Proposition. *Let* $seq = (V_1 \ldots V_n)$ *be an RTC step in a component* $Cm(o)$ *and* $sconf_i(o'').status \neq dying$ *for all* $1 \leq i \leq n$ *and for all* $o'' \in Cm(o)$. *Then exactly one of the following holds:*

- *seq is an RTC step in object o or*
- *seq is an RTC step in some object* $o' \in Cm(o)$ *such that* $sconf_2(o).ds = o'$.

The following consequence from Definitions 3, 7 and 8 formalises the main properties of the *krtUML* semantics described.

Consequence 1. Let M be a *krtUML* model, $r \in runs(STS(M))$, where $r = (V_0 \ldots V_n \ldots)$ with $V_i = (sconf_i, prt_i, sysfail_i)$ $(0 \leq i)$. Let $o \neq \bar{o} \in O_C$. Then the following invariants hold:

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

(ii) (Asynchronous interference points)
$(V_i, V_{i+1}) \in \rho_{get\_event}(o) \implies V_i$ is the beginning of a RTC step $(V_i \ldots V_{i+k})$ of the component $Cm(o_1)$, where $o_1 = sconf_i(o).my\_ac$.

(iii) (Synchronous interference points)
$(V_i, V_{i+1}) \in \rho_{accept\_op}(o) \implies V_i$ is the beginning of a RTC step $(V_i \ldots V_{i+m})$ of object o. An object o can accept operation calls only on the borders of its own RTC steps. □

Thus, the semantics STS(M) encodes all system executions as interleavings of component RTC steps, allowing event receptions by its objects only at the borders of component RTC steps (when other objects from the component are not currently executing or suspended by an uncompleted operation calls). On the other hand, each component RTC step is a chain of invocations of objects' RTC steps, each of these started by suspending the previous one with an operation call.

Since the semantics is given from the local point of view of objects, the sequentialisation mechanism within a component is implemented via the shared variables: $o.my\_ac.ds$, $o.my\_ac.eq$, and $prt(o)$.
The following consequence summarises these means of the sequentialisation in the proposed semantics.

Consequence 2 (*Sequentialisation of Component Computations*). For each component $Cm(o)$ (with $o.isActive = true$), the shared variables $o.ds$, $o.eq$, and prt play the main roles for the scheduling between several computations available in the component objects. For all objects $o' \in Cm(o)$:

***eq*:** $o'.isActive = false \implies \forall r = (V_0 V_1 \ldots) \in runs(STS(M))\ \forall i \geq 0 : sconf_i(o').eq = \varepsilon$ (only one event queue is used in a component to store asynchronous stimuli sequentially).

***ds*:** $(V_i, V_{i+1}) \in \rho_{get\_event}(o') \implies sconf_i(o').my\_ac.ds = \mathsf{nil}$ (an object can receive an event only if no other object is executing its event reception).

***prt*:** $prt(o').status = busy \implies o'.status = suspended$ (beginning of a synchronisation: an operation can be executed only if the calling object is suspended); and $(V_i, V_{i+1}) \in \rho_{pick\_up\_result}(o') \implies (prt_i(o').dest = o'' \implies sconf_i(o'').status = idle)$ (end of a synchronisation: an object can proceed with the result of an operation call only if the callee became stable). □

It is easy to see that at each transition $\rho = (V_i, V_{i+1})$ in each system run $r = (V_0 \ldots V_n \ldots)$ at most one state-machine transition can be taken, and transitions enabled in different component are chosen non-deterministically. By considering all possible

runs, we provide semantics covering different execution speeds and scheduling between components.

4. Assessing the expressiveness of *krtUML*

In this section we indicate how to reduce richer UML models from *rtUML* as supported in the IST project Omega [8] to the *krtUML* subset defined in Section 2. Besides this, we explain the choice of the design decision behind the formal semantics. The subset of UML chosen to be translated to *krtUML* and called *rtUML* contains the following additional features (not presented in *krtUML*):

- Primitive operations in classes' definition, i.e. those implemented by methods (with actions defined in Definition 1(iv) extended with richer navigation expressions and constructs for branching and loops).
- Three kinds of operation concurrencies: sequential, guarded, concurrent.
- Two specific kinds of primitive operations for each class: constructor and destructor.
- Three kinds of associations between classes (semantically distinguished): composition, aggregation, neighbour.
- Three kinds of visibilities of attributes, operations, and association ends: public, private, protected.
- A generalisation relation (inheritance) between classes: (a) multiple inheritance under the assumption of no naming conflicts; (b) attributes and operations as redefinable elements; (c) dynamic classification (sometimes called casting). This implies polymorphism, in particular for abstract operations (corresponding to virtual in C++ or deferred in Eiffel [22]).
- Hierarchical state-machines containing:
  - Hierarchical states (in addition to simple states): both AND-states (concurrent regions) and OR-states.
  - Pseudo-states: initial, deep history, shallow history.
  - Instead of using join- and fork-vertices we consider transitions with multiple sources and targets.
  - Entry- and exit-actions in states.
  - Transitions can be complex, i.e. containing both guards and (non-primitive) actions.

4.1. Translating rtUML to krtUML

The translation from *rtUML* to *krtUML* comes in several steps. Most of them are technical and beyond the scope of this paper. In the following subsection we only outline some more interesting steps. The extended explanations can be found in [8].

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 pre-compilation introduces these as what we call *implicit attributes* (e.g. the attribute *self* of type *c* within each class *c*), and translates code invoked when creating compound

objects for establishing links employing a set of *implicit operations* such as '*add_to_association_end*', '*initialise_association_end*', '*delete_from_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[4] 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. Thus, for each class c with direct successors under the aggregate or composite relation to classes $c_1, \ldots, c_n$ we require that preprocessing defines operations $create_{c_j}(p_j)$ and $destroy(p'_j)$ with $type(p_j) = c_j.my\_ac$ and $type(p'_j) = c_j$. We preclude any user defined constructor and destructor bodies (invoked from $create_c(ref)$ and $destroy(ref)$ respectively) by the sequential composition of the action catering for the recursive creation and deletion of the component parts.

As a trivial pre-processing step, we eliminate complex navigation expressions by introducing auxiliary attributes, reducing the level of de-referencing to at most one (as used in Definition 1). In the scope of one thread, we also inline recursively primitive operation bodies directly into transitions of state-machines containing their calls.

Regarding class generalisation, we create private instances for each segment descriptor (of the class itself and of all its predecessors in the generalisation hierarchy) much as the creation of a compound objects induces creation of its components. Implicit "offset" attributes $parent\_type_{c'}$ serve to navigate from the current (segment) object to the definition of inherited attributes and operations (not overridden in the current class). Such hierarchical structure of object allocation allows us to keep access to all operation implementation and state-machines overridden in the specialised objects, e.g., for easy casting (assignment of a specialised object to an attribute of the generalised class with "forgetting" the specialised attributes and operations). Attributes $parent\_type_{c'}$, defined within each object and for each immediate ancestor c' in the generalisation class hierarchy, are also used for a sanity check of qualified operation calls: if an operation $C_i :: op_j$ is called from an object o of type C_k, then C_i must be a generalisation of C_k (op_j must be defined in C_k but may be overridden). Attributes $parent\_type_c$ are used for "static" polymorphism, where the "current" type of each object at each operation invocation is defined as the type of the attribute referring to it, which is statically detectable within class definitions. The "actual" type of reference attributes (or descriptor pointer), which is the type of the object at its creation time, is kept in another implicit association attribute which we call *type_table*.

The semantics proposed in this paper is defined from the objects' local point of view with statically inlined methods (which is necessary for the formal verification). "Current" type of an object is a specification from the point of view of a calling object aimed at hiding non-necessary details or unacceptable behaviour, whereas "actual" type is used to find the correct implementation of abstract operations, that is having deferred implementation.

[4] 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.

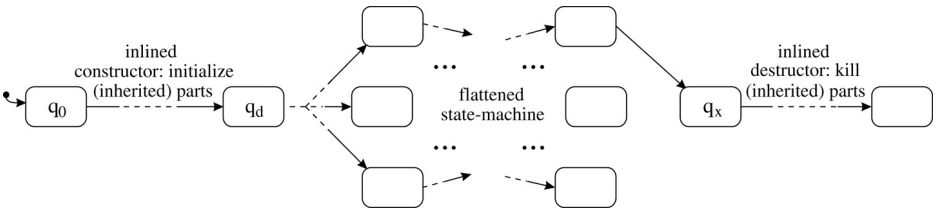


Fig. 9. Inlining object initialisation and destruction.

We also preclude any constructor and destructor bodies by the action sequences catering for the recursive creation and deletion of the segments in the hierarchical descriptors.

We do not require any restrictions on the state-machine inheritance: a subclass might have state-machine overwritten independently from that of the corresponding superclass. All private copies maintain their own object-configurations; 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.

Another pre-compilation step transfers hierarchical UML state-machines from *rtUML* to flat state-machines of *krtUML* without changing the behaviour. The states in a flattened state-machine correspond to state configurations from the original state-machine (sets of states which can be active at the same time) extended with a function called the history configuration (keeping information for the history connectors). A transition in a flattened state-machine relates two state configurations iff one configuration can be reached from another by triggering a transition with the corresponding guard in the original state-machine. The effect of such a transition in the flattened state-machine is constructed as a sequential composition of exit-actions, the effect of the corresponding transition in the original (hierarchical) state-machine, and entry-actions (which may comprise non-deterministic sequentialisation of concurrent exit/entry-actions from different concurrent substates). Besides this, for the state-machine of each 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) state-machine. Only the state-machine 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 vertices (from the original state-machine) has an outgoing transition to some auxiliary state without a triggering guard and with action *destroy*(*self*) (the result of the inlining of initialisation and destruction codes to a flattened state-machine is shown schematically in Fig. 9);
- Several "*internal*" states necessary to split complex transitions, e.g. transitions containing a sequence of actions or construction actions. An example of splitting a transition with sequential composition of actions and a branching construct is shown in Fig. 10.

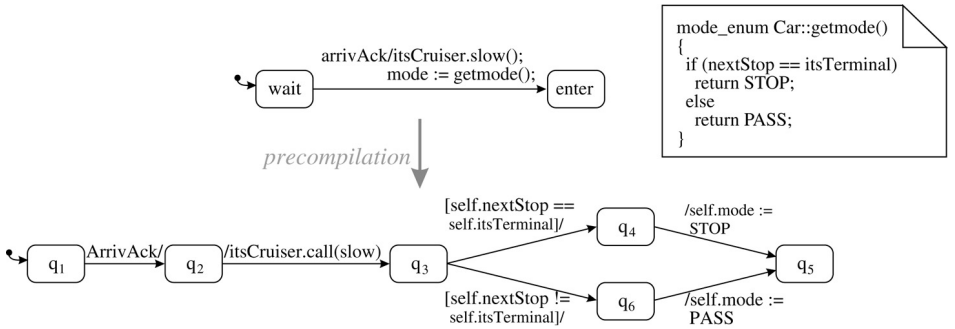


Fig. 10. Splitting a complex transition.

To preserve the original execution granularity and avoid some other transition being enabled when inside the execution of a split transition (containing an action block), we have to introduce some kind of semaphore, which blocks other transitions to be executed. To do this, every transition will obtain an additional guard *not(inside_trans)*. Splitting a complex action into simple parts will first set this Boolean variable to *true*. At the end that variable will be reset to *false*. This will avoid another transition being started while being in the middle of another one.

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 modelling 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 regards inter-object communication. On the basis of the pragmatics of active objects in UML, we mandate that each such component-group contains exactly one active object, and allow it to include (also dynamically in run-time) an arbitrary number of passive objects in the group. Reactive passive objects are required to delegate their event-handling to the one active object within the group.

Figs. 11 and 12 illustrate the concepts of components and inter-component communication using the Automated Rail Cars System example from [14]. The graphical representation of a snapshot of a model on Fig. 11 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. Fig. 11 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

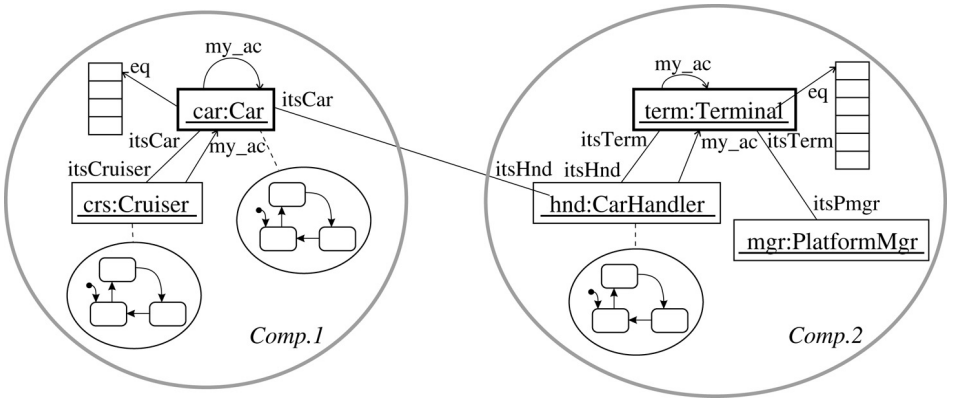


Fig. 11. Component structure. A snapshot of a model part shows active objects *car* and *term* (with their event queues) and passive objects *crs*, *hnd*, and *mgr*. Reactive objects *car*, *crs*, and *hnd* are denoted by associated schematic state-machines. Active objects *car* and *term* designate their components *Comp*.1 and *Comp*.2, respectively.

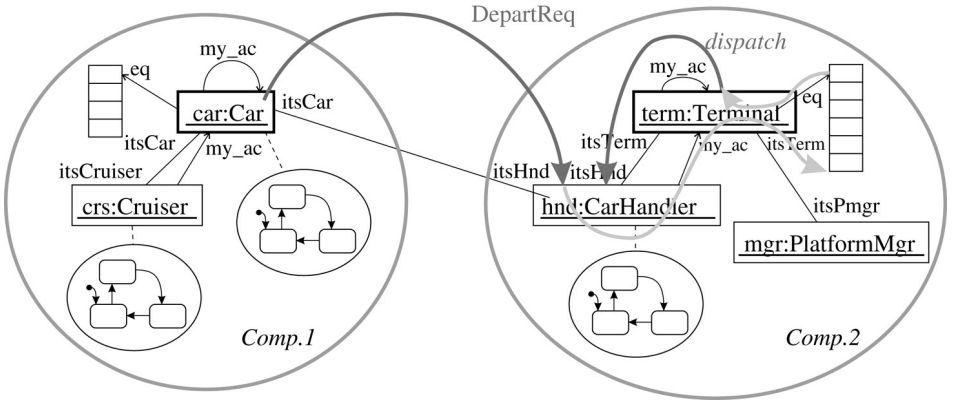


Fig. 12. Event communication between components. Sending an event of *DepartReq* from *car* to *hnd* in fact enters the event into the event queue of *term*, which is the active object associated with *hnd*.

take the event from the queue and dispatch it to the destination. This is indicated in Fig. 12 by light grey arrows. The semantics in Section 3 is explained from the perspective of the destination.

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 — [19,3] etc. on state-machines, [10] on collaboration diagrams, [12,15] etc. on class diagrams, [29] on use cases, [2] on activity diagrams — or just to give formal foundations for action language (e.g., [24,1]). In our approach a symbolic transition system represents both static and dynamic aspects. The combination of statics and dynamics is also given in [31] which considers the problem of defining active classes with associated state-machines. At variance with our approach, the authors do not give precise semantics 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 problem of an adequate formalisation for concurrency as well as for aspects of communication between objects, which have been uncovered in [31] and not addressed in the original UML documents. Open problems are typical for so-called *loose semantics* introduced in [15], where the aspects of concurrency and object communication are not fixed to some design decision, but cover different variation points. Such loose semantics is not suitable for formal verification. Our paper tries to overcome this problem by providing an executable semantics as an example of the feasibility of UML precise formalisation, in particular for verification purposes. On the other hand, there are a number of UML modelling and/or verification tools implementing precise semantics by translating UML models to programming languages or model checker internal formats [16,30,34,20]. These tools have different limitations on the supported UML features and do not provide a formal description of the implemented semantics. So, such translations can be used only at the later stages of system design, not at the modelling levels.

H. Hußmann [15] 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 [33] have suggested using a 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 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) *Meta-modelling semantics.* This group of approaches is based on the application of a “bootstrapping” principle [6], 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 [5,4,1]. Essentially, an algebraic specification is used to describe legal (local) snapshots of the system without treating actions. The biggest issue, not covered by these approaches, is how to deal with complex aspects of dynamic behaviour concerned with concurrency and inter-object communication. The study of A. Kleppe and J. Warmer [18] is based on the pUML OO meta-modelling approach. In addition, it takes into account that static and dynamic viewpoints on the system cannot be separated. But the formal semantics for state-machines is not really defined, the set

of primitive actions is very restrictive (object creation, attribute manipulation), 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 issues 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 [11], Object-Z [17], CASL [32], etc.). For example, G. Reggio et al. [32] proposed a general scheme 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 [3,2,23,7]. E. Börger et al. [3] 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 for solving transition conflicts and it is not clear how firable transitions are selected. Unlike these approaches, our study provides one formalism (STS) for both static and dynamic semantics, which also contains a (restricted) action language.

Indeed, different approaches mentioned above can be combined as shown in [23]. In this research, static semantics is defined using the 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. The aspects of inter-object communications were not really covered and the semantics of UML state-machines was not addressed, although it can be accompanied by the complementary papers [2] and [3]. But these articles consider state-machines separated from the rest of UML, whereas our approach provides one semantics for model structure (class diagrams) and behaviour (state-machines). We also allow more flexibility for the combination of different orthogonal aspects: concurrency and reactivity, synchronous and asynchronous inter-object communication.

6. Conclusion

As regards the investigation results sketched above, the main novelty of our approach is that it resolves the ambiguity of the formal UML specification w.r.t. concurrency and object communication by giving a formal semantics for a chosen concrete decision. W. Damm and B. Westphal [9] have shown that this semantics can be used for formal verification.[5]

In our approach we allow both active and passive objects to be reactive, thus considering event communication between all objects. We also capture the combination of two

[5] The proposed semantics choice was evaluated with a prototype of a discrete-time verification environment under the UML modelling tool Rhapsody [16] as well as with a more abstract, XMI-based, representation of UML models.

different kinds of inter-object communication — synchronous (via operation calls) and asynchronous (via signal events).

Thus, we have provided the semantical foundation for a 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 focused 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 laid down in this paper to develop a verification environment for real-time UML.

Acknowledgement

We gratefully acknowledge the contribution of our Omega partners in fine-tuning the semantics.

References

[1] J.M. Alvarez, T. Clark, A. Evans, P. Sammut, An action semantics for MML, in: Proc. UML 2001, 2001. http://www.cs.york.ac.uk/puml/mmf/AlvarezUML2001.pdf.

[2] E. Börger, A. Cavarra, E. Riccobene, An ASM semantics for UML activity diagrams, in: T. Rus (Ed.), Proc. AMAST 2000, LNCS, vol. 1816, Springer-Verlag, 2000, pp. 293–308.

[3] E. Börger, A. Cavarra, E. Riccobene, Modeling the dynamics of UML state machines, in: Y. Gurevich, Ph.W. Kutter, M. Odersky, L. Thiele (Eds.), Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Proceedings, LNCS, vol. 1912, Springer-Verlag, 2000, pp. 223–241. DBLP http://dblp.uni-trier.de.

[4] T. Clark, A. Evans, S. Kent, The metamodelling language calculus: foundation semantics for UML, in: Proc. FASE 2001, 2001, pp. 17–31. www.dcs.kcl.ac.uk/staff/tony/docs/MMLCalculus.ps.

[5] T. Clark, A. Evans, S. Kent, S. Brodsky, 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.

[6] T. Clark, A. Evans, S. Kent, 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.

[7] K. Compton, J. Huggins, W. Shen, A semantic model for the state machine in the UML, in: G. Reggio, A. Knapp, B. Rumpe, B. Selic, R. Wieringa (Eds.), Dynamic Behaviour in UML Models: Semantic Questions, Workshop Proceedings, UML 2000 Workshop, Bericht 0006, October 2000, Ludwig-Maximilians-Universität München, Institut für Informatik, 2000, pp. 25–31. http://www.kettering.edu/~jhuggins/papers/uml2000.ps.

[8] W. Damm, B. Josko, A. Pnueli, 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.

[9] W. Damm, B. Westphal, Live and Let Die: LSC-based Verification of UML-Models, in: F.S.d. Boer et al. (Eds.), Proceedings of the First International Symposium on Formal Methods for Components and Objects, FMCO, October, LNCS, vol. 2852, Springer-Verlag, 2003.

[10] G. Engels, J.H. Hausmann, R. Heckel, S. Sauer, Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML, in: Proceed. of the 3rd International Conference on the UML 2000, October 2000.

[11] A.S. Evans, 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.

[12] A. Evans, R. France, K. Lano, B. Rumpe, The UML as a formal modeling notation, in: The Unified Modeling Language: the First International Workshop, June 1998, Springer-Verlag, 1999.
[13] S. Graf, I. Ober, Semantics of time extensions, Omega Technical Report, Deliverable D1.1.4, Project IST-2001-33522 OMEGA, December, 2003.
Available from http://www-omega.imag.fr/doc/d1000199_2/D1.1.4-time-extensions-v2.pdf.
[14] D. Harel, E. Gery, Executable object modeling with statecharts, IEEE Computer 30 (7) (1997) 31–42.
[15] H. Hußmann, Loose semantics for UML, OCL, in: Proceedings 6th World Conference on Integrated Design and Process Technology, IDPT 2002, June, Society for Design and Process Science, 2002.
[16] I-Logix Inc. Rhapsody, 2002. http://www.ilogix.com/products/rhapsody/index.cfm.
[17] S.-K. Kim, D. Carrington, Formalizing the UML class diagrams using object-Z, in: France, Rumpe (Eds.), Proc. UML'99, LNCS, vol. 1723, Springer-Verlag, 1999, pp. 83–98.
[18] A. Kleppe, J. Warmer, Unification of static and dynamic semantics of UML, 2001.
http://www.klasse.nl/english/uml/unification-report.pdf.
[19] G. Kwon, Rewrite rules and operational semantics for model checking UML statcharts, in: Proceed. of the 3rd International Conference on the UML 2000, October, University of York, 2000.
[20] J. Lilius, I.P. Paltor, vUML: a tool for verifying UML models. Turku Centre for Computer Science, Abo Akademi University, Finland, 1999. Technical Report.
[21] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, New York, 1991.
[22] B. Meyer, Eiffel: The Language, Prentice-Hall, 1998.
[23] I. Ober, Harmonizing design languages with object-oriented extensions and an executable semantics, Ph.D. 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, September 2003, V.1.0, formal/03-09-01. Available at http://www.omg.org/docs/formal/03-09-01.pdf.
[26] Object Management Group. Unified Modeling Language: Superstructure, v.2.0, Final Adopted Specification ptc/03-08-02, August, 2003. Available from http://www.omg.org/cgi-bin/doc?ptc/03-08-02.
[27] G. Övergaard, Formal specification of object-oriented meta-modelling, in: T. Maibaum (Ed.), Proceedings Fundamental Approaches to Software Engineering, FASE, LNCS, vol. 1783, Springer-Verlag, 2000.
[28] G. Övergaard, Using the BOOM framework for formal specification of the UML, in: Proceedings of Defining Precise Semantics for UML, 2000.
[29] G. Övergaard, K. Palmkvist, A formal approach to use cases and their relationships, in: UML 1998, 1998.
[30] Rational Software Corporation. Rational Rose Family, 2003.
http://www.rational.com/products/rose/index.jsp.
[31] G. Reggio, E. Astesiano, C. Choppy, 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.
[32] G. Reggio, M. Cerioli, 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.
[33] M. Richters, M. Gogolla, On formalizing the UML object constraint language OCL, in: T.-W. Ling, S. Ram, M.L. Lee (Eds.), Proc. 17th International Conference Conceptual Modelling, ER'98, LNCS, vol. 1507, Springer-Verlag, 1998, pp. 449–464.
[34] Telelogic AB. Telelogic Tau, 2003. http://www.telelogic.com/products/tau/index.cfm.