OMEGA

Correct Development of Real-Time Embedded Systems

IST-2001-33522

Title : A Formal Semantics for a UML Kernel Language

Author(s) : W. Damm, B. Josko, A. Votintseva (OFFIS), A. Pnueli (WIS)

Editor : Verimag

Date : 06/01/03

Identifier : IST/33522/WP 1.1/D1.1.2-Part1

Document Version : 1.2

Status : Final

Confidentiality : Public

Abstract :

This report defines a sufficiently expressive sublanguage of the behavioural modelling constructs of UML allowing to capture industrial real time applications. Covered aspects include in particular the concept of Active Objects, polymorphism as well as a detailed presentation of UML statecharts. For the chosen restrictive sublanguage, the zero-time semantics is given at two levels of abstractions: defined as an execution scheme and as a formal representation in terms of symbolic transition systems.

Document history

Revision	Date	Author	Comments
1.2	03/01/2003	W. Damm, B. Josko, A. Pnueli, A. Votintseva	Definition of the Omega-subset of UML, revised polymorphism and execution scheme
1.1	04/06/2002	W. Damm, B. Josko, A. Pnueli, A. Votintseva,	Kernel model + inheritance
1.0	01/04/2002	W. Damm, B. Josko, A. Pnueli, A. Votintseva,	Kernel model draft

Table of Contents

Kernel Model for Behaviour Description

Introduction

Currently, standard UML does not provide completely formal semantics. In the specification of the standard UML 1.4 [1], the given semantics is incomplete and static: it is said that the meanings of the constructs are defined using natural language (p.2-9). This causes an ambiguity in the definition of computation within a UML model. Although [2] gives the semantics of a part of UML (action language) in the terms of UML metamodel, this is not sufficient for the formal verification, because it is still incomplete and not formal. The existing UML tools (e.g. Rhapsody [11], Rational Rose [12], TAU [13]) implement internally some semantics with their compilers and/or simulators, which differ from tool to tool. There is a number of papers investigating UML semantics. Thus, for example [7] motivates the need for a formal semantics for UML. The approach from [10] considers the UML semantic in terms of Time Object Model, but it focuses more on the methodology than on formal semantics. The articles [4] and [5] outline formalising UML by translating class diagrams into Z specifications, thus giving semantics only for static part of the UML models. The paper [8] describes the pUML approach introducing denotational semantics into the UML metamodel, which places emphasis on building a precise core semantics for the UML but without accent on real time. The paper [9] gives the ASM semantics for the UML with OMG actions definition based on the metamodel, but it does not treat statemachines, whereas [6] adapts and extends ASMs to get to UML state-machines. In this proposed semantics, statemachines are disconnected from the rest of the UML.

Thus the aim of this report is to select a sufficiently expressive sublanguage, allowing to capture real-time application, and specify formal semantics of the chosen part of UML. This part of the deliverable provides a zero-time semantics, concentrating only on the way of object communications and computations in the system as sequences of actions without time concepts and architectural description. For breaking down complexity, the report is split into three chapters.

The first chapter focuses on all intricacies of classes, operations, events, class diagrams, using only flat UML state machines. It specifies the way of modelling a quite expressive sublanguage of UML into one more restrictive but allowing to specify a formal semantics.

The second chapter concentrates on the behaviour of a single active object using the full complexity of UML statecharts. It also discusses how to transform such complex statecharts into flat state machines, considered in the first section with their fully defined semantics.

The third chapter summaries the definition of the Omega-subset of UML by listing the restrictions on the common UML notions. This is done for more suitable usage of the Omega-subset by customers.

1 Active Objects

1.1 Definition of the Kernel Model

In this paper, we describe a subclass of the UML language [1], called Omega-subset. We will specify explicitly all model elements and their constituents in the profile of the considered kernel models.

Classes and their constituents

1.1.1. We consider an object system created from a finite set of *classes* C, and use small c as metavariable for classes. We distinguish a special subset $A \subset C$ called *actors* to specify behaviour external to the system.

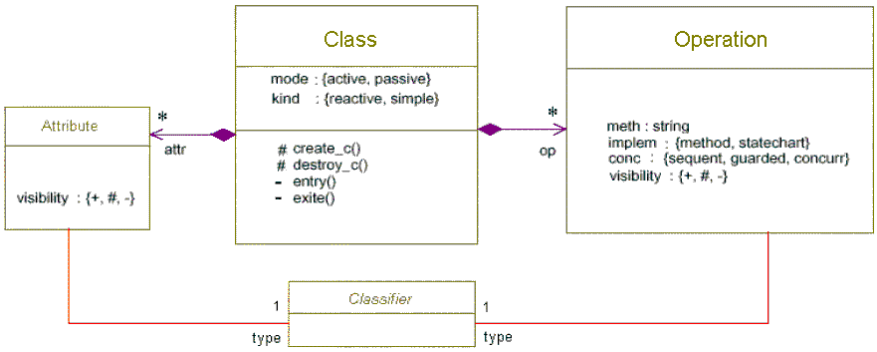


Figure 1. Class constituents

1.1.2. Each class *c* has associated a set of attributes *c.attr* , which we assume to be strongly typed. We use small *a* as metavariable for attributes. For the sake of this discussion, *types* can be either *class identifiers*, in which case we also refer to such an attribute as a *reference,* pointing to an instance of the class defined as its type, or some *predefined types* – like (array of) integer, boolean, char – and definable enumeration types whose internal structure is not relevant in the context of this paper.

1.1.3. Each class can either be *active* or *passive*. *Active* classes own their own thread of control (defined in Section 1.2.), their instances come equipped with their own *signal - dispatcher* (see below), which in particular will maintain all signals directed to this instance. In contrast, *passive* classes only execute on behalf of other objects, i.e. they cannot initiate any computation by themselves. Once a passive object has been activated it can initiates methods calls to other objects. We refer to this implicit attribute of a class *c* as its *mode*, technically denoted by *c.mode*.

1.1.4. A *reactive* class is a class which can process *events*. Event-processing is defined by a *statechart*. In this paper, we consider only two types of events: *signals* (asynchronous, also called signal events) and *call events* (synchronous). We call a class *simple*, if it is non-reactive. Technically, we associate with a class *c* an implicit attribute *c.kind* , telling us whether *c* is reactive or simple. We require all reactive classes to be parts of active ones (see below the composition relation) and direct their signals to the corresponding signal-dispatcher (specified in Section 1.2).

1.1.5. With each class we associate a set *c.op* of *operations*, which the object is willing to serve. Operations are used for synchronous communications. Operations may be parameterised, and are seen as always returning a value – *op.return* – including possibly the unique value *nil* of the return type *()*. To specify the name space of an operation *op1* – if *op1* is defined in different classes – we will also refer to this operation as *c::op1* where *c* is a class name such that $op1 \in c.op$. With each operation *op* we associate a list *op.param* of its parameters, which can be empty. The *type op.type* of an operation *op* defines the type of its parameters as well as the result type (also called *operation signature*): *op.type* = (*a1:type1*, *a2:type2*, …, *an:typen*; *op.return:type*), where (*a1*,*a2*,…,*an*)=*op.param*. In the current version we consider only *input* parameters (no output parameters).

1.1.6. We distinguish between *primitive* operations, whose implementation is given by a piece of code (in this paper assumed to be given in the restricted action language described below), and *triggered* operations, whose implementations are given in a statechart (by the corresponding call events).

1.1.7. For each primitive operation *op* we assume as given its method *op.meth* as a statement of the action language, involving only attributes visible in the containing class (see below) and formal parameters of the method – written as *op.meth*= {<action_sequence>} for some sequence of primitive actions and statements <action_sequence> specified in action language. For each triggered operation *op* we assume that the statechart processes the corresponding call event *op* with a return value.

1.1.8. A primitive operation may not call a triggered operation. Primitive operations have additional implicit attribute *op.virt* with values from {*virtual, non_virtual*} to specify the way of the delegation for operation call w.r.t. inheritance relation (see below). If *op1.virt = virtual*, then by any call of operation *op1* from an object *ob1* of class *c* – where *op1∈ c.op* – the method defined in the most specialised class *c´* (i.e. inheriting from *c*) will be executed, where *ob1* is also an object of class *c´*. If *op1.virt=non_virtual* and an object *ob1* is considered as an object of a generalised class *c*, then the method defined exactly in the class *c* will be executed by a call of *op1* from *ob1*.

1.1.9. Call trees of primitive operations must be well-founded, i.e. there is no recursive calls.

1.1.10. Operation calls are executed synchronously, that is, the caller is blocked until reception of the return value, in a sense discussed in more detail in the section on semantics.

1.1.11. In the context of multiple objects, the *concurrency* attribute *op.conc* tells us, how simultaneous request of multiple objects to execute a given operation *op* are to be handled. This can take values in the set *{sequential, guarded, concurrent}*, with intended meaning as follows:

1.1.11.1. If an operation *op* of class *c* is *guarded*, then the implementation must ensure the following predicate *mutex(o, op)* for any instance *o* of class *c*:
"no other thread of control is active in o while executing op".

1.1.11.2. If the operation *op* of class *c* is *sequential*, than the context of any invocation of *op* in any instance *o* of *c* must guarantee mechanisms for *mutex(o, op).*

1.1.11.3. If the operation is *concurrent*, than there are no restrictions regarding invocations of *op.*

1.1.12. In the current version of the paper, we require all triggered operations to be guarded or sequential.

1.1.13. We assume a set of predefined primitive operations for all predefined types.

1.1.14. In the current version of the paper, we assume that primitive operations are either sequential or free of side-effects (which is also called query).

1.1.15. Each class comes with the following *predefined operations*. *create_c(ref:c): c*, returns the identity of the created instance of class *c*, and initialises unique implicitly defined attribute *self* (in the newly created object) with the identity of the created instance. We will write *create_c()* (without parameters) as a shortcut for *create_c(nil)*. We will use this operation with actual parameters different from *nil* to describe a creation of objects with respect to the generalisation relation (see below). Operation *destroy_c(ref: c): ()* kills the object denoted by its actual parameter.

1.1.16. Each class *c* may contain an *constructor* resp. *destructor* *c.construct* (resp. *c.destruct*) to specify actions needed to be invoked during the creation resp. destruction of each object of class *c*. Note that constructor and destructor are special kinds of primitive operations, i.e. they are defined using the action language described below. The constructor *c.construct* is invoked at creation time of the object (by any invocation of operation *create_c*), the destructor *c.destruct* is executed at destruction time (by an invocation of operation *destroy_c*).

1.1.17. Each class defines the *visibility* of its attributes and operations, which can either be *public*, *private*, or *protected*. If an attribute or an operation is *private*, then it is only visible within the class itself (can be accessed by objects of this class). If it is *public*, then its visibility is unrestricted, hence any instance of any class can read and modify a public attribute, and any instance of any class can call a public operation. If an attribute or an operation is *protected*, then it is known to the class itself and any class inheriting from the class. Thus a protected attribute *a* of an instance of class *c* can be modified by an instance of class *c´* provided *c* generalises *c´* (see below the definition of class generalisation).

1.1.18. We assume as given a set *Sig* of signals (asynchronous messages). UML views signals as classes, in particular allowing specialisation of signals. The *type s.type* of signal *s* defines the type of its parameters *s.param*. In the current version of this document, signals are assumed to be *public* and considered as a special kind of classes.

1.1.19. Signal based communication is *asynchronous* – after emitting the signal, the sender continues processing without awaiting reception.

1.1.20. In the current version of the paper we consider no priority relation on signals.

1.1.21. With each class *c* we associate a set *c.sig* of *signals*, which its objects are willing to receive (can be handled by its statechart).

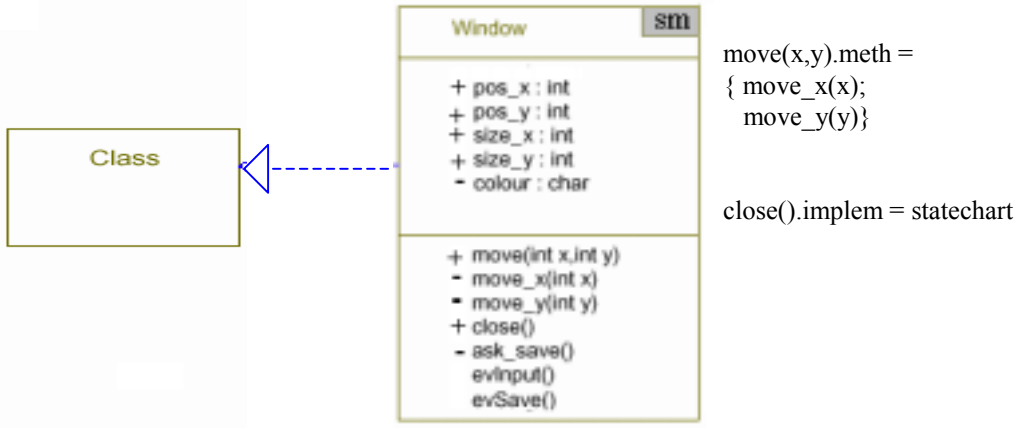


Figure 2. Example of a class

Class Interfaces

1.1.22. In extending the UML standard, we propose to associate with each class *c* its *interface c.int*. The interface of a class collects all attributes, operations, and signals, which can cross the class boundary. The definition of a class interface explicates, what aspects of an instance of a class are externally observable. This concept is mandatory as prerequisite in formally capturing requirements on the behaviours to be supported by a class, it also can be used to define interfaces in component based designs. Specifically, the interface lists:

1.1.22.1. all public attributes;

1.1.22.2. all public operations;

1.1.22.3. all operation calls emitted to other objects;

1.1.22.4. all signals declared as receptions (implying that any of their specialisation can be received as well). For signal *s* we denote $s \in c.int$ to specify that class *c* accepts reception of the signal *s*;

1.1.22.5. all signals emitted.

In the example from Fig. 2, *Window.int* = {pos_x, pos_y, size_x, size_y; move(x,y), close(), evInput, evSave} is the class interface of `Window`.

Static structure: relations between classes

1.1.23. A *class diagram* allows to capture information about instances of classes and their *relationships*.

1.1.24. Classes can be *related* according to one of the following *relations.*

1.1.25. If class *c* is a *generalisation* of class *c´* ($c' \lessdot c$, and class *c´* is a *specialisation* of class *c*), then

1.1.25.1 *c´* provides all operations and attributes (including association ends described below) of *c* – as well as its own – which are public

1.1.25.2 *c´* can call all operations of *c* which are public or protected

1.1.25.3 *c´* can read and modify all attributes of *c* which are public or protected

1.1.25.4 if *c* is a reactive (active) class, then *c´* is reactive (active, respectively). If a class inherits from several reactive classes, then all of them – immediate generalisations – must have the equal statecharts.

1.1.25.5 A statechart from the generalised class can be overwritten by a statechart defined in a specialised class, specifying new event (signals and/or triggered operations) receptions.

1.1.25.6 We use $<$ to denote the transitive closure of the generalisation relationship ($< = \lessdot^{+}$). Relation $\lessdot$ is also called inheritance.

1.1.26. We also consider the generalisation relation between signals. If signal *s* is a generalisation of signal *s´* ($s' < s$), then the parameter list of signal *s´* must contain all parameters of signal *s* and all classes accepting reception of signal *s* accept signal *s´* as well.

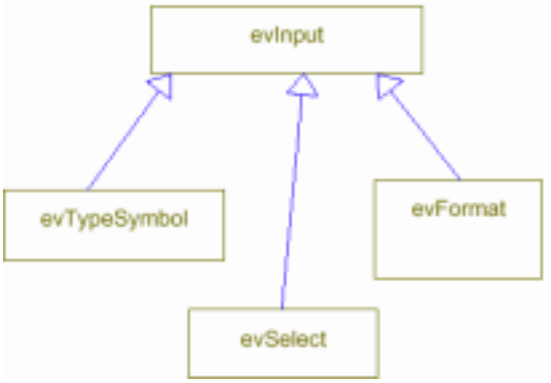


Figure 3. Example of event inheritance

The inheritance from Fig.3 imposes that the reception of *evTypeSymbol*, *evSelect* and *evFormat* can be accepted by all classes accepting event *evInput*. Lists of the parameters of the former events must contain the parameter list of *evInput*.

1.1.27. Acquaintanceship between classes is captured only by establishing *associations* between classes. Parameterised names for object communication are not supported here. We distinguish three kinds of associations: neighbour, aggregate (also called weak aggregation) and composition (also known as strong aggregation). Association classes are not considered here. Each association is given by its identifier $ac\_id \in ASSOC\_ID$ and defined as a triple $ac\_id = (agr, root, end\_points)$, where

1.1.27.1 $ac\_id.agr \in \{composite, aggregate, neighbour\}$ is an association kind.

1.1.27.2 $ac\_id.root \in C$ is a class possessing the knowledge about other classes.

1.1.27.3 $ac\_id.end\_points \subseteq C$ is a set of classes known by *ac_id.root*.

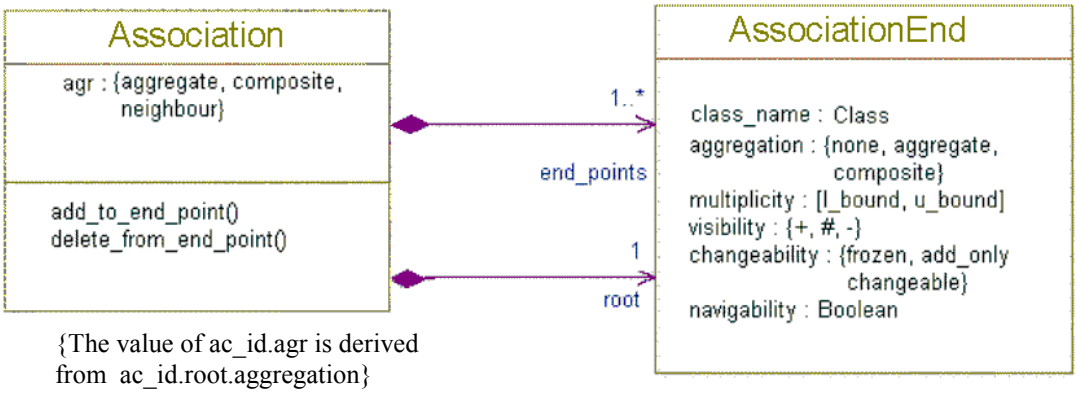


Figure 4. Definition of Association

1.1.28. Every *association end* – *ac_id.root* and *c* ∈ *ac_id.end_points* – come equipped with a number of *predefined attributes*.

1.1.28.1. The *aggregation* attribute takes values in the set *{none, aggregate, composite}* and thus actually defines the three classes of associations discussed above in their directed and multidirectional form.

1.1.28.2. The *multiplicity* attribute takes a subset M of ***N***. It defines, how many instances of the class attached to this end are associated with the class attached to the opposite association end. The special case *muliplicity* = * stands for *unbounded multiplicity*. There are certain well-formedness-conditions on multiplicity. As an example, association ends attached to the compound class must have multiplicity one (while, in contrast, many instances of a constituent class might be required to exist).

1.1.28.3. The *visibility* attribute takes values in the set *{private, public, protected}* and has the same semantics as visibility of class attributes and operations. If an association end is *private*, then it is only visible to the class attached to the opposite association end. If it is *public*, then other classes can get access to this association end if there are navigating associations through other public association ends to the class attached to this end. If it is *protected*, then classes specialising the class attached to the opposite association end inherit the visibility, i.e. can get access to this association end.

1.1.28.4. An association end can be specified via attribute *name*, in which case the attached class(es) can be referred to under this name from the acquainted object. We assume availability of a default name (such as *its_c*) for an unnamed end associated with a class *c*. Named association ends are also called *roles*.

1.1.28.5. If the multiplicity of an association end is greater than 1, the different instances of classes attached to this end at run-time can either be maintained as an ordered list, or a set, depending on the value – *true* or *false* – of an attribute *ordered*. If an association end with name *its_cj* is maintained as ordered list (*its_cj.ordered* = *true*), then we can refer to the instances of the class attached to this end as *its_cj*(1), *its_cj*(2) etc.

1.1.28.6. The attribute *changeability* restricts ways, how association ends can be manipulated. If *frozen*, then they will maintain the references obtained at initialisation time. *add_only* allows to add new instances to the association end without ever deleting already associated instances. Only *changeable* association ends allow unrestricted modification of their references.

1.1.28.7. The attribute *navigability* of type boolean. If the attribute is true then in the graphical representation of a directed association this is indicated by an arrowhead at the

corresponding association end (meaning that the attribute of the opposite association end is false). An association is bi-directional if attributes *navigability* of its both ends are true, in the graphical representation indicated by absence of arrowheads at both association ends.

Association ends of different kinds of association have several constraints on the values of their attributes, as defined in the following three tables.

Composite associations *ac_id.agr=composite*	*ac_id.root*	*ac_id.cj ∈ ac_id.end_points*
aggregation	*composite*	*none*
multiplicity	1	n > 0 , *
changeability	*frozen*	If multiplicity = n then *frozen*, else *add_only* or *changeable*
navigability	unrestricted	true
visibility	unrestricted	unrestricted
ordered	not applicable	prefer unordered (*ordered = false*)
name (default)	*its_c* where c = *ac_id.root*	*its_cj* where *cj ∈ ac_id.end_points*

Aggregate associations *ac_id.agr=aggregate*	*ac_id.root*	*ac_id.cj ∈ ac_id.end_points*
aggregation	*aggregate*	*none*
multiplicity	1	n > 0 , [m,n] ,*
changeability	*frozen*	unrestricted
navigability	unrestricted	true
visibility	unrestricted	unrestricted
ordered	not applicable	prefer unordered (*ordered = false*)
name (default)	*its_c* where c = ac_id.root	*its_cj* where *cj ∈ ac_id.end_points*

Neighbour associations *ac_id.agr=neighbour*	*ac_id.root*	*ac_id.cj ∈ ac_id.end_points*
aggregation	*none*	*none*
multiplicity	n > 0 , [m,n] ,*	n > 0 , [m,n] ,*
changeability	unrestricted	unrestricted
navigability	unrestricted	unrestricted
visibility	unrestricted	unrestricted
ordered	prefer unordered (*ordered = false*)	prefer unordered (*ordered = false*)
name (default)	*its_c* where *c = ac_id.root*	*its_cj* where *cj ∈ ac_id.end_points*

1.1.29. Pragmatically, the omposite association is used to denote a *"part of-"* relationship. This entails, that creation of the compound object induces creation of its constituents (as long as their multiplicity is bounded). Similarly, killing a compound object induces killing of its constituents. The composite relation is also sometimes referred to as *strong aggregation relation*. We require, that only the compound object itself can create and destroy its parts.

1.1.30. The *composite* association defines for a *compound* class *c* its constituent classes. We write $c' \hookleftarrow c$ to denote that compound class *c* has (possibly multiple) instances of classes *c´* as constituents, i.e. there is composite association *ac_id* such that *c = ac_id.root* and *c´ ∈ ac_id.end_points*. If

ac_id.root. navigability = true, this is denoted as $c' \perp c$. Selecting either $\hookrightarrow$ or $\perp$ makes the composite association *directed* resp. *bi-directional*.

1.1.31. Acquaintance between classes in the aggregation association is defined as for composite classes. The *aggregate* association denotes a *weaker* form of grouping of a compound class, in that constituents are not created nor destroyed automatically at creation resp. destruction time (time when operation *create_c()* resp. *destroy_c()* is performed) of the compound object (of class *c*). We write $c' \hookrightarrow_w c$ (resp. $c' \perp_w c$) to denote the *weak aggregation* relation between a compound class *c* and one of its constituent c' (in its directed and bi-directional version, defined similar to that of composite association in 1.1.30).

1.1.32. A compound (resp. aggregating) object always *knows* its constituent objects (the attribute *navigability* of the association end attached to a constituent class is true). If the composite (resp. aggregate) association is bi-directional, then all parts also know their compound object.

1.1.33. We suggest to use strong aggregation whenever meaningful, and consider the weak aggregation specifying the "possibility" for creation resp. destruction of a weak constituent invoked from its aggregating object during run-time.

1.1.34. Technically, *neighbour* association is a *derived* concept, defined in terms of the aggregation attributes of the association ends (see above). We explicitly name this type of association because of its relevance in defining the acquaintanceship relation between different parts of a system.

1.1.35. We use $c' \leftarrow c$ to denote the directed neighbour relation between two classes, i.e. if there is neighbour association *ac_id* such that *ac_id.root* = *c*, $c' \in$ *ac_id.end_points*, *ac_id.root.navigability* = *false* and *ac_id.end_points(c').navigability* = *true*, i.e. it is only *c* who knows neighbour c'. In case of a bi-directional relation $c' \leftrightarrow c$, both neighbours know each other: *ac_id.root.navigability* = *ac_id.end_points(c').navigability* = *true.*

1.1.36. Technically, we will refer to association ends by the value of their *name* attributes (of type $c \in C$, where *c* is attached to the corresponding association end) and index if the multiplicity is greater than 1. For all relations between classes we require that there is no clash of the association end names. This means for all classes *c*, *c1*, *c2* and associations *ac_id1*, *ac_id2* $\in$ *ASSOC_ID* (c = *ac_id1.root* & c = *ac_id2.root* & *c1* $\in$ *ac_id1.end-points* & *c2* $\in$ *ac_id2.end-points*) $\Rightarrow$ *ac_id1.end-points(c1).name* $\neq$ *ac_id2.end-points(c2).name*. We will include association ends as implicitly defined attributes in objects (see below).

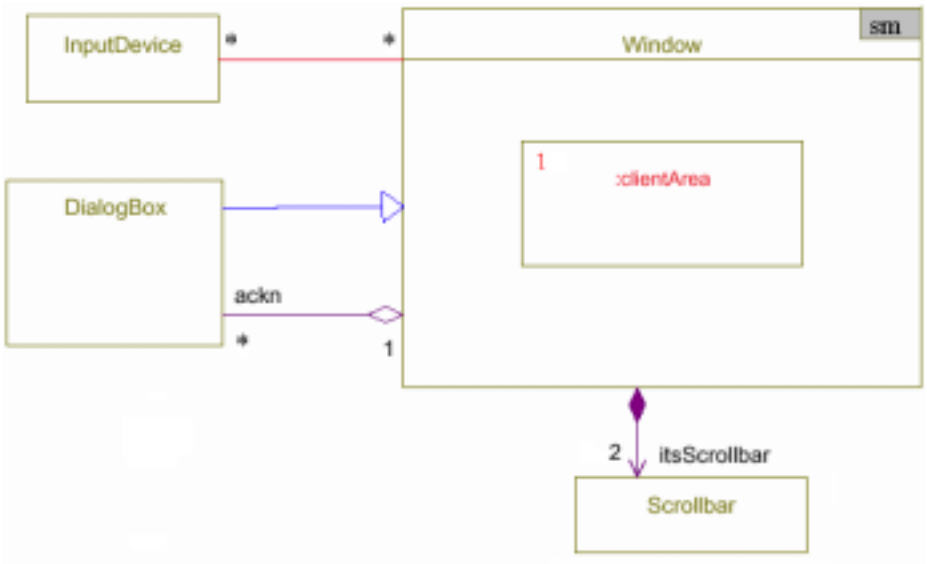


Figure 5. Example of a class diagram

In the example from Fig. 5, the following relations are pictured:
Window ↔ *InputDevice* : bi-directional neighbour,
DialogBox < *Window* : *DialogBox* inherits from *Window*,
DialogBox $\perp_w$ *Window* : an object of class *Window* may create any number of instances of *DialogBox*, known under name *ackn*,
Scrollbar ↵ *Window* and *clientArea* ↵ *Window* : one object of class *clientArea* and two objects of class *Scrollbar* are created at the creation time of an object of *Window*, the latter one knows objects of *Scrollbar* under default names *itsScrollbar(1)* and *itsScrollbar(2).*

1.1.37. We propose to require, that an object can only communicate with those objects it knows through associations, so that the creator of an object knows its children.

Action Language

1.1.38. We propose a restricted action language subsuming the following features. All statements must comply to visibility restrictions as described above. We partition attributes of an object into the following three groups:

1.1.38.1. *Navigation* attributes *nv_a* are all attributes induced from association ends and implicit attribute *self*, introduced as attributes by the preprocessing steps (see Section 1.3). In the action language they are represented via *a**, *aj** (*a*, *a´* or *aj* in the definition below can also represent navigation attributes under additional constrains).

1.1.38.2. *Auxiliary pointer* attributes *p_a* are typically user declared, and are used to temporary store pointers e.g. passed as parameters. They are not allowed to designate receivers of operation calls or signals and must be private. In the definition below they are represented via *a* or *a´* (on the right sides of assignments).

1.1.38.3. *Basic* attributes *b_a* are those of some basic predefined type, which are not references (represented via *aj, a*, and *a´* in the definition below).

1.1.39. We propose the following set of primitive actions. Here we describe actions with abstract syntax, just to give a list of action types and restrictions used inside primitive operations. These actions are basic in the sense that they can be mapped to different (programming or abstract) languages.

1.1.39.1. *Object creation: a* := create_c()* for *c* ∈ *C*, creates a new instance of class *c*, initialises implicit attribute *self,* and assigns the identity of the newly created object to the attribute *a** of type pointer to class *c´* (association end) with the following restriction: *c* α *c´* where *c´* is the current class containing the action and $\alpha \in \{\leftarrow\!\!\!\!\!\!, \perp, \leftarrow\!\!\!\!\!\!_w, \perp_w\}$.

1.1.39.2. *Simple assignment: a0 :=* <*primitive expression*> involving a set of predefined primitive operations (excluding navigation expressions), local (to the object where it occurs) attributes, and possibly visible formal parameters, complying to type restrictions: *a0* must be a basic attribute.

1.1.39.3. *Attribute values exchange: a := a0*.a1*.....an*.a´* or *b0*.b1*....bn*.b := b´* where *aj** and *bj** ($0 \le j \le n$) are navigation attributes complying to visibility restrictions, *a´* is an attribute (of any kind) of the class pointed by *an** and *b* is an attribute (basic or navigation) of the class pointed by *bn** – both visible in the current class *c´* containing the action. If *a* (or *b*) is an association end, then *c* α *c´* where *c* = *a.type* (or *c* = *b.type*, respectively) and $\alpha \in \{\leftarrow\!\!\!\!\!\!_w, \perp_w, \leftarrow, \leftrightarrow\}$.

1.1.39.4. *Operation call: a := a0*.a1*.....an*.!op(a1,...,ak)* with non-nil return value or *a0*.a1*.....an*.!op(a1,...,ak)* with nil return value from the class instance pointed by *an**, subject to restrictions on acquaintance between objects described above.

1.1.39.5. *Explicit operation call: a := a0*.a1*.....an*.!c::op(a1,...,ak)* with non-nil return value or *a0*.a1*.....an*.!c::op(a1,...,ak)* with nil return value from the object pointed by *an**

(complying to visibility), which is an instance of a class c' such that $c' < c$ (c is a generalisation of c').

1.1.39.6. *Setting return value:* $return := a$ of an operation call.

1.1.39.7. *Object destruction:* $destroy(a^*)$ denoted by the reference a^* with the following restriction: $c\ \alpha\ c'$ where c' is the current class containing the action, $c = a^*.type$ and $\alpha \in \{\hookleftarrow, \perp, \hookleftarrow w, \perp w\}$. Note that reaching the termination connector corresponds to *destroy(self)*.

1.1.39.8. *Signal emission:* $a0^*.a1^*.....an^*.!s(a1,...,ak)$ to the class instance pointed by an^*, subject to restrictions on acquaintance between objects described above.

1.1.40. We call expressions $a0^*.a1^*.....an^*$ *navigation expression* and require that they may only use navigation attributes. We propose to support sequential composition, branching, and (bounded or unbounded) iteration. The exact syntax for actions and control constructs is discussed in M2.2.1, *Definition of the tool exchange format*.

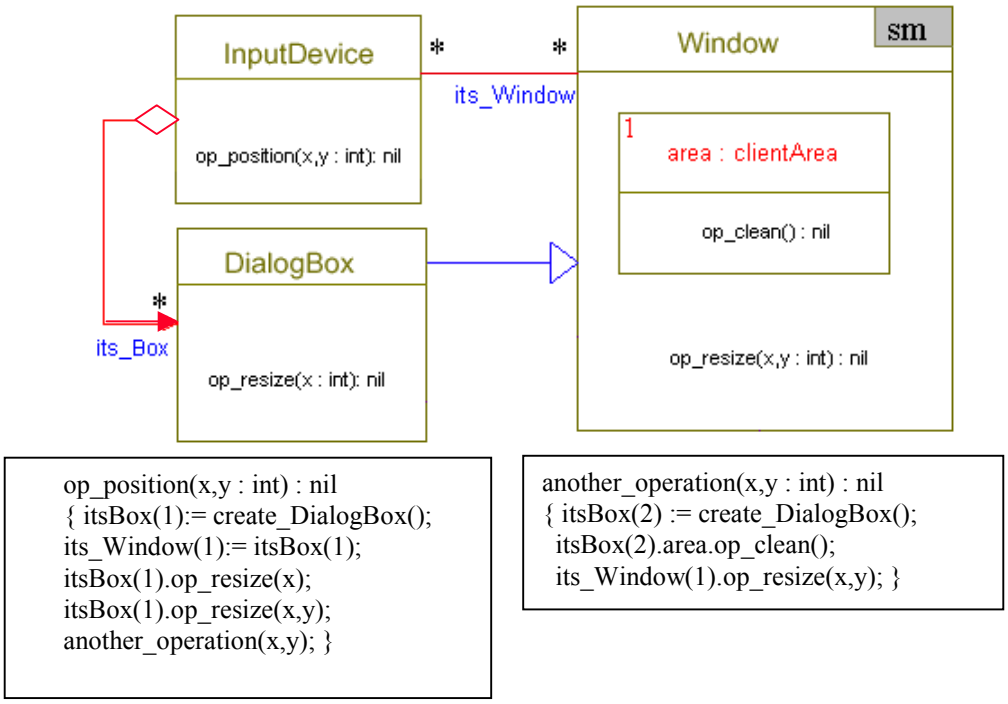


Figure 6. Example of legal operation definitions in class *InputDevice*

Flat UML state-machines

1.1.41. We restrict ourselves in this section to *flat* state-machines. Full UML statecharts are treated in Section 2 of this report.

1.1.42. An *event* (or *trigger event*) in a state-machine is specified as one of the following

1.1.42.1. $s(p1, ..., pn)$ reception of signal s, local parameters pj matching $s.type$

1.1.42.2. $op(p1, ..., pn)$ acceptation of operation call op (call event), local parameters pj matching $op.type$

1.1.43. A *guard* is a boolean expression containing attributes and primitive operations of the current class (where it is used). The trivial guard is *true*, which is omitted in the graphical representation.

1.1.44. A *guarded trigger* is a conjunction of a trigger event *t* and a guard *b*, written syntactically as *t[b]*.

1.1.45. A flat UML state-machine is a tuple

$$sm = (Q, T, D, q0)$$

where

- Q is a finite set of states
- $T \subseteq Q \times (\{<guarded\ trigger>\} \cup \{<guard>\}) \times <primitive\ action> \times Q$ is a finite set of transitions, where <*primitive action*> is an action with the syntax described above.
- $q0 \in Q$ is the initial state
- $D : Q \rightarrow \wp(Sig)$ gives for each state the set of deferred signals

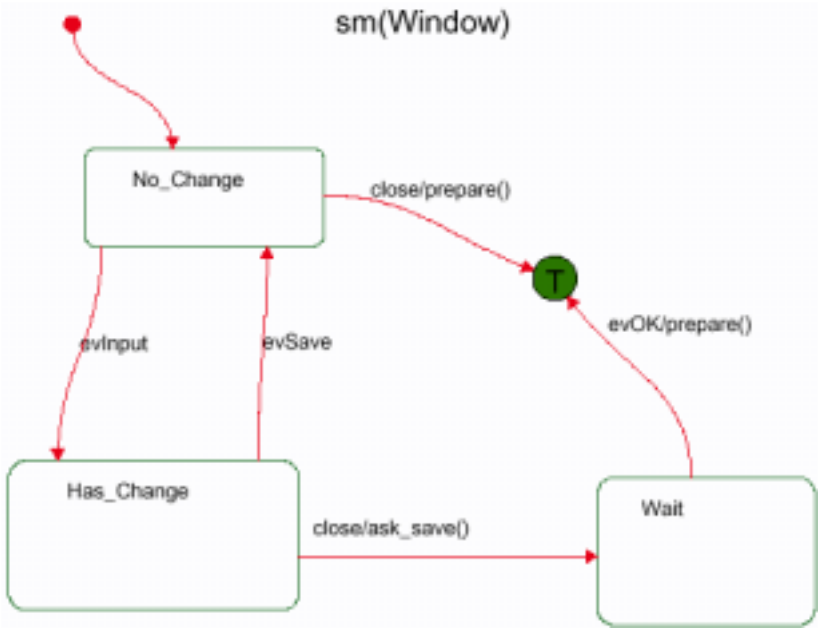


Figure 7. Example of a flat state-machine

UML model

1.1.46. A UML model is a tuple

$$M = (C, A, Sig, c0, \perp, \rightharpoonup, \rightharpoonup\!\!\!\!\wedge, \perp\!\!\!w, \leftarrow, \leftrightarrow, <, sm)$$

where

1.1.46.1.	C	is a finite non-empty set of classes
1.1.46.2.	$A \subset C$	is a non-empty set of actors We denote $C' = C \setminus A$ a set of internal (system) classes
1.1.46.3.	Sig	is a finite set of signals
1.1.46.4.	$c0 \in C'$	is the root class (which we require to be active)
1.1.46.5.	$<$	is the generalisation relation between classes C′ or between signals *Sig*
1.1.46.6.	$\perp \subseteq C' \times C'$	is the bi-directional composite relation between classes
1.1.46.7.	$\rightharpoonup \subseteq C' \times C'$	is the directed composite relation between classes
1.1.46.8.	$\rightharpoonup\!\!\!\!\wedge \subseteq C' \times C$	is the directed aggregate relation between classes
1.1.46.9.	$\perp\!\!\!w \subseteq C' \times C$	is the bi-directional aggregate relation between classes
1.1.46.10.	$\leftarrow \subseteq C \times C$	is the directed neighbour relation between classes

1.1.46.11. $\leftrightarrow \subseteq C \times C$ is the bi-directional neighbour relation between classes
1.1.46.12. *sm* assigns to each reactive class $c \in C$ a UML state-machine

such that

- the root class is the maximal aggregating class: $\forall\, c \in C' (\forall\, c' \in C' \;\forall\, \alpha \in \{\perp, \hookleftarrow, \hookleftarrow w, \perp w\} \neg(c' \alpha\, c) \Rightarrow c \hookleftarrow w\, c0)$ – i.e. all elements from C' maximal in the weak and strong aggregation hierarchy are successors of *c0* under (weak) aggregation
- the relation of the composite association defines a DAG
- no sharing of weak components between several weak composites are allowed to occur in run-time: if class c relates is a weak component (related by weak aggregation) of both c1 and c2, then any instance o of class c will either be associated to an instance of class c1 or to an instance of class c2 (but not both even at different points of time)
- in the multiple inheritance, there is no naming conflicts
- for all classes *c* , all inter-object communication supported by the behavioural aspects of *c* (i.e. its operations, its entry- and exit-script, and potentially its statechart) is compliant to *c.int*, the class interface of *c*

For uniformity, for every simple class *c* we assume $sm(c)=(\{q\}, \emptyset, \emptyset, q)$.

1.1.47. Instances of a class – or memory allocations in run-time – are called objects. We will use $obj \in c$ or $cl(obj)=c$ to denote that object *obj* is an instance of class *c*. Every object *obj* possess operations *obj.op* and attributes *obj.attr* defined in its class *cl(obj)* with values *val(obj.attr*) as well as state-machine $sm(obj)=sm(cl(obj))$.

1.2 Design Decisions

In this section we describe informally the behaviour of the UML-models from the Omega-subset specified in the previous section.

1.2.1. Intuitively, an *active* object (i.e., an instance of an active class) is like a signal-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 signal from a signal queue associated with the active object, and dispatches it for processing to either its own state-machine, or to one of the passive reactive objects associated with this active object.

1.2.2. This association must be defined for each reactive object; we thus assume the existence of a mapping

$$my\_ac : \{c \in C \mid c.kind = reactive\} \rightarrow \{c \in C \mid c.mode = active\}.$$

Technically we associate with each class *c* an implicit attribute *c.my_ac* as reference to an active class controlling computations in objects of class *c* (dispatching signals at the correct time and performing operations).

1.2.3. Typically, such association (with role name *my_ac*) is *derived* from the partial order induced by the transitive-reflexive closure of the composition relation $R=(\hookleftarrow \cup \perp)^*$: for any class *c*, the initial (or default) value of *c.my_ac* is

$$c.my\_ac = my\_default\_ac(c) = min_R\{c' \in C \mid c'.mode = active \wedge cR\, c'\}.$$

1.2.4. In the sequel, we assume, that the value of attribute *c.my_ac* is either explicitly or implicitly defined for each passive object (clearly *c.my_ac = self* for each active class *c*), and will collectively refer to the set of all (passive) classes associated with an active class as its servants:

$$servants(c) = my\_ac^{-1}(c) = \{ c' \in C \mid c'.my\_ac = c\}.$$

1.2.5. An important notion in the behavioural specification of concurrent systems is *thread of control* (or simply thread).

1.2.5.1. A *task* is a logical group of objects, it corresponds to a unit of computation maintained by a RTOS (subject of scheduling for RTOS).
1.2.5.2. Run-time correlate of a task is called *thread*.
1.2.5.3. Each invocation of a task corresponds to activation of a thread which performs a sequence of actions corresponding to a *run-to-completion step* at the semantic level.

1.2.6. In the UML, task structure is defined by grouping at least one active object and possibly associated passive objects. Thus, threads are controlled by active objects. For this aim, an active object contains an operation *execute()* which engages a thread by performing actions from the associated objects from *servants(c)* – method calls, dispatching signal and call events.

1.2.7. In this version we allow primitive operations called only in the scope of one task and we define inter-task communications via event sending (both signal and call events). Note that if a primitive operation calls are necessary in such communication, they can be represented (manually or automatically) as triggered operation by adding "self-loop" transitions to all states in the corresponding state-machine with operation name as trigger and operation method in the action part of the state-machine.

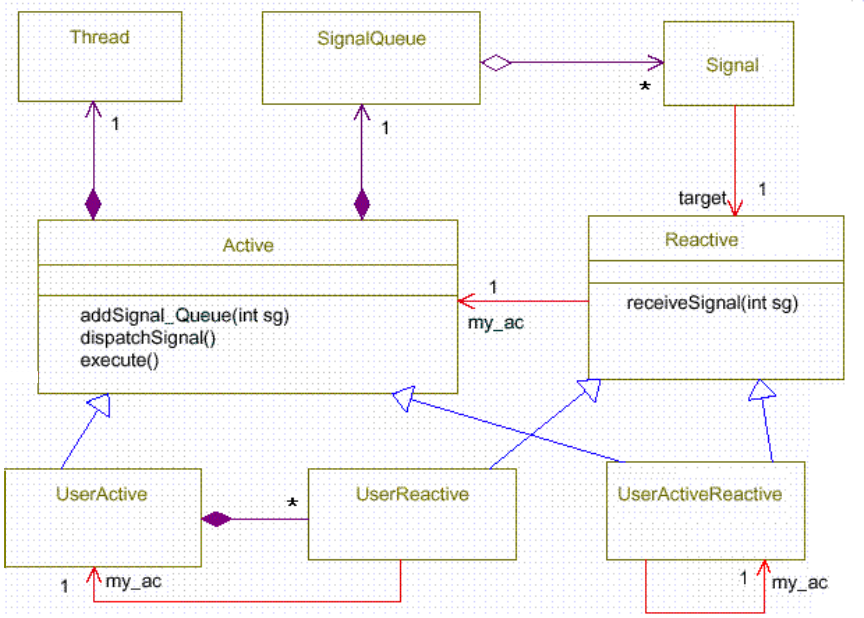


Figure 8. Execution structure

1.2.8. A fundamental concept for the execution semantics is the notion of a *run-to-completion step*. It is instructive to see the analogy of a run-to-completion step in sequential circuit design. In this domain, the circuits accepts new input values with the rising edge of the clock. These new values are propagated through the circuit; the time it takes for reaching the output of the circuit or new latches depends on gate delays and path length: at any intermediate point in time, some wires may have reached stable values, while the input is still propagated through other areas of the circuit, causing wires to switch values. In a well defined circuit, though, this propagation delay is bounded: eventually all wires will have taken values, which remain stable. It is only then that the clock will tick again, causing new values to be propagated through the circuit.

1.2.9. In the UML context, reactive classes with their state-machines take the role of the sequential circuit, and state-configurations take the role of circuit wires. So, assuming initially a stable state-configuration, the dispatcher picks a signal from a signal queue or triggered operation called from other thread and hands it over to the reactive class – this corresponds to the rising edge of the clock. The current state configuration will determine, how this input is processed. Lets assume the flat UML state machines introduced in the previous section. Assume, that there is a transition guarded by a trigger event matching the dispatched signal event or call event (and assume no local condition). Then this transition can be taken, toggling the "circuit input" one level. Consider the newly entered state. Assume, that we have a transition originating from this state, labelled with a local condition as guard which evaluates to true. Then – within the same run-to-completion-step – this transition will be taken, causing the state-machine to reach a new state. In a well designed state-machine, this propagation process will continue until a state is reached, from which all

originating transitions with only local conditions cannot be taken, since they evaluate to false – such a state is called a *stable state*, and it is exactly in such a situation, where we can allow a new clock tick – in our context: where we can accept a new event (signal or call), starting another run-to-completion step.

1.2.10. The proposed execution models ensures, that there is **at most a single thread of control** active in each object at each point in time, entailing that concurrent accesses must in some form be interleaved. We feel that violating this assumption leads to execution models, which are complex and incomprehensible, thus causing modelling errors.

1.2.11. We take a standard interleaving semantics, in which all active objects are running asynchronously. In interleaving object executions, we have to decide on the level of granularity of interference, indeed a key design decision. In general, verifying against an execution model supporting fine grained interference will yield stronger results, but raise the level of complexity of the model. In contrast, a coarse grained approach risks to hide interference possible in a real implementation, but will typically reduce the complexity of the model. Finding the right level is thus extremely critical.

1.2.12. The proposed execution models specifies discrete time, where every clock tick corresponds to one run-to-completion step, which is considered atomic – i.e. not interruptible by input stimuli – and without duration: every action in the scope of one run-to-completion step is supposed to be instantaneous. The execution with continuous time and duration of actions, allowing interrupt/exception, are described in the "Time Extensions in UML " (D1.1.2, part 3 from Verimag).

1.2.13. To explain our approach we start with the trivial but nevertheless fundamental observation, that an instance can at any time be in one of two roles, to which we refer to as the driver- and callee role (Fig. 9), respectively.

1.2.13.1. *The Driver role*

Assume that the object instance is in a stable state, in a sense to be made precise below. The instance will then consult its queue of pending (generated but not jet consumed) signals, and pick the top signal. Lets assume, that the current state of the instance is such, that an outgoing transition contains the dispatched signal as trigger event, and that the associated condition is true. Then a *run-to-completion* execution is initiated, in which the instance as initiator of this run to completion drives its execution: it can perform a primitive operation *op* called from an object of the same thread by executing actions from its body (*op.method*), or it can emit a signal to any object, or it can call triggered operation from another object. Note that the driver stays in unstable state during the time when another object (callee) serves its call and can continue its execution only after the termination of the call (at the callee side). Driver can be either an active object or reactive one performing the current computation on behalf of an active object.

1.2.13.2. *The Callee role*

The same object may – at a different point in time – be *serving* – directly or indirectly – some driver object in performing its run to completion, by executing an operation call "on its behalf". Such incoming calls can only be taken at well defined points, related to the choice of granularity of interference. Assuming that an object has reached such an "interference point", it is ready to accept a call of its operation *op* if one of the outgoing transitions in the current state is labelled with call event *op* and its associated condition is true. It will then execute the call, by evaluating all involved actions, possibly invoking other operations, and eventually reach a stable situation, thus completing the current call. Thus, as for signals, also call events (implementing triggered operations) induce a "run-to-completion", taking the state-machine to a configuration where further steps are only possible by accepting an operation call or dispatching a signal. Callee can be any object containing operation implementation.

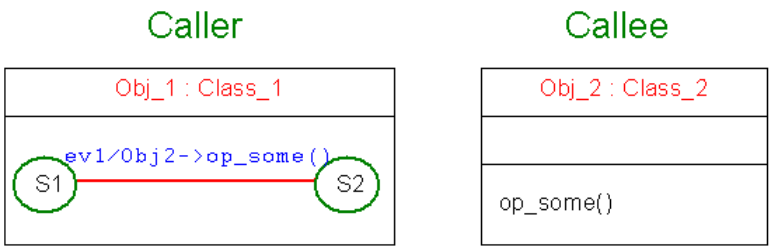


Figure 9. Two roles of objects: Caller (= Driver) and Callee

1.2.14. We interleave execution of different threads at a **coarse** level, and do **not** allow preemption of run to completion steps. This – in combination with the restriction to sequential operations or primitive operations free of side effects – clearly increases understandability of the model. Regarding model complexity, the coarse grained interleaving should allow significant optimisations, since no external communication – neither through operations, nor signals – occurs during a completion run; all occurring communications relate to objects in callee role, under the control of the driving object. Thus, the callee does not accept any event nor method until it has reached a stable state. For the computations within objects belonging to the same thread of control: the thread can re-enter an object for a primitive operation call at any time.

1.2.15. Fig. 10 shows the two principle run-time states of an object.

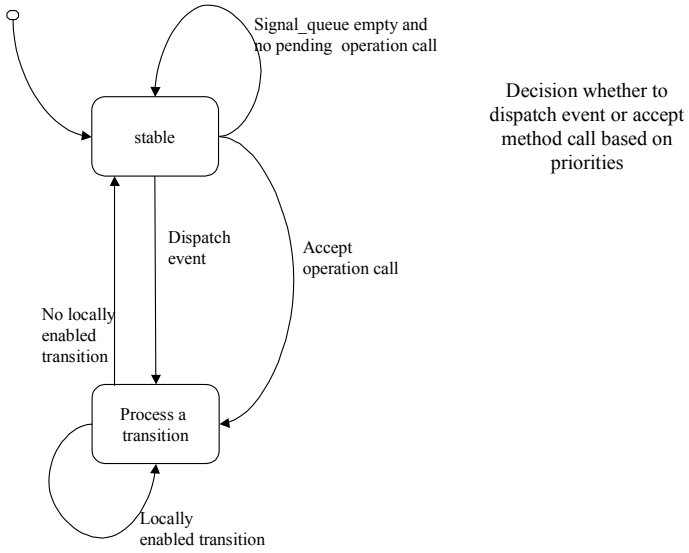


Figure 10. Run-to-completion execution scheme

1.2.16. Let us now define the concept of stability formally. A state *q* of reactive object *o* – with state machine *sm(obj)* and attribute evaluation *val(o.attr)* – is *stable* if *o* is executing (has been created) and there is no locally enabled transition.

$$stable_{val}(q,o) \Leftrightarrow \forall(q,l,\gamma,q') \in sm(obj).T \; \exists grd \; (\exists ev \; l \equiv ev[grd] \; \vee (l \equiv grd \; \wedge val(grd)=false))$$

Predicate $stable_{val}(q,o)$ characterises synchronisation points within one object *o*.

1.2.17. We can associate with each state a predicate characterising the willingness of the object to accept an event (operation call or a signal) in this state as follows.

$$ready_{val}(o, q, ev) \Leftrightarrow \exists q', grd, \gamma : (q, ev[grd], \gamma, q') \in sm.T \wedge val(grd)=true$$

We call *ready set* for an object state *q* defined as a set of events for which the predicate $ready_{val}$ *(o, q, ev) = true*.

1.2.18. With these concepts, we can now elaborate our discussion of the dispatching process, assuming that *o* is stable in state *q*. If the top-signal of the queue is ready in this situation, then a new run-to-completion step can be initiated in *o*. Otherwise, depending on whether *ev* is declared as deferred signal in state *q* (formally expressed as $ev \in D(q)$), it would either be discarded (if not deferred) or maintained in the FIFO for later reconsideration (at the beginning of the next run-to-completion step).

1.2.19. It is important for real-time extensions to explicate, when "clock ticks" occur in the above situations.

1.2.19.1. If the dispatched signal is discarded, then conceptually we consider this as a completion of a run-to-completion step (involving no processing). By selecting the next signal for dispatching, we initiate a new run-to-completion step, thus incrementing time.

1.2.19.2. If the dispatched signal is deferred (and thus maintained in the FIFO buffer), we again "increment time", by letting the dispatcher pick the next younger signal and passing it to the object for processing, also inducing a clock tick.

1.2.20. As a modelling guideline, we suggest that signals for which persistency must be ensured are monitored in appropriate orthogonal components, thus ensuring, that they are in the ready set of all stable configurations.

1.2.21. A similar problem arises for operation calls. Suppose, that object *o* is currently executing an operation call on behalf of active object *o1*, and that this call completes. In case of pending operation calls to *o* by another object, such as *o2* requesting a call of *op*, the operating system will now grant access to this request (thus in fact giving higher priority to pending operation calls over signals). Now suppose, that in the current stable state, *o* is *not* ready for *op*. In the current UML model, this call is then considered "completed" - though it never executed! This effectively entails, that also triggered operation calls can be *discarded*.(with *nil* return)

1.2.22. Again there are multiple ways to address this problem. First, we could enforce the same modelling guidelines, thus requiring, that all stable state-configurations offer transitions for all possible triggered operation calls. Again, this might be considered prohibitive from the modelling overhead – in which case the execution model must be changed, by only offering those operations to an object, for which it is ready. While this also would force us to give up the order of arrival of operation calls, we can in this case argue as follows that this should be acceptable.

1.2.23. Call a *logical channel* a pair of acquainted objects. A minimal requirement on any reasonable implementation is, that it preserves the order of signals and operation calls along a logical channel. For signals, we have seen, that simply moving a signal to the tail of the queue would even potentially destroy this requirement. Regarding operation calls, we can exploit the fact, that the caller is suspended until arrival of the return value, thus the queue attached to the semaphore ensuring exclusive access to the object will *never* contain to request from the same object, thus bypassing its head in order to give preference to a younger operation call *op´* for which *o* is ready will not violate the minimal ordering requirement. We thus propose in this case to actually change the execution model and only offer those operation calls, which are in the current ready set. This is the justification of storing all pending requests of operation calls in a *table* rather than in a *queue.*

1.2.24. This solution is close to the rendezvous concept of ADA: both the caller and the callee have to agree on the call. One can in fact view a state *q* with outgoing transitions containing operations *op1, ..., opn* as an *accept* statement *accept(cond1:op1, ..., condn:opn)*, where *condj* denote the guards of the corresponding transitions.

1.3 "Preprocessing" Semantics of the Omega-subset

We use standard preprocessing techniques to compile the Omega-subset defined above to a small kernel language handled in the section on formal semantics. We will in particular compile away all relations. To be able to dynamically set up neighbour associations, preprocessing must create implicit operations for adding, initialising, updating and deleting association end points as specified below.

Pre-processing comes in four parts.
First, we extensively introduce what we call *implicit attributes* and *implicit operations*. Secondly, we model generalisation relation as association by introducing the implicit attributes *uplink* and *downlink* changing the creation procedure for subclasses and mechanism of operation calls. In addition, to support polymorphism we replace user defined assignments with additional scripts checking the type of attributes. Thirdly, we extend user-defined constructors and destructors with additional actions dealing with creation and destruction of composite objects. The combination of the introduction of implicit attributes and this third preprocessing stage eliminates the need to handle associations explicitly in the semantics. Finally, we eliminate complex navigation expressions by introducing auxiliary attributes, reducing the level of de-referencing to at most one. In the scope of one thread, we also inline recursively primitive operation bodies directly into transitions of statecharts containing the calls (possible due to the choice of sequential operations and queries, i.e. operations free of side effects). This trivial step is omitted in the definition below.

Introduction of implicit attributes and operations

1.3.1. For each class *c* we create an implicit attribute *self* of type *c*.

1.3.2. If *its_c1*, ..., *its_ck* are (names of) navigable end-points of associations originating from class *c* or roots of associations with end-point in *c*, then

1.3.2.1. Class c has implicit attributes *its_cj* of type

- *cj* , if the multiplicity of the corresponding end-points is 1;
- *set of cj* , if the multiplicity of the corresponding end-points is greater than 1.

1.3.2.2. Assume that $c = ac\_id.root$ and $ac\_id.cj \in ac\_id.end\_points$ (or $cj = ac\_id.root$ and $ac\_id.c \in ac\_id.end\_points$, resp.) for some association *ac_id*. Then class *c* has implicitly defined unified operations for manipulating association ends with different multiplicity and different changeability attribute.

1.3.2.2.1. $init\_its\_cj(id_1{:}cj, \ldots, id_n{:}cj){:}()$ if multiplicity of this end-point (named *its_cj*) is $n \in \mathbf{N}$. The function associates identifier attributes $\{id_1, \ldots, id_n\} = ID(ac\_id.cj)$ with objects of class *cj* so that it can be used to get access to the corresponding association ends as $c.its\_cj(id_1), \ldots, c.its\_cj(id_n)$. Note that if the ordered attribute of this end-point is *true*, then there is an order $id_1 < \ldots < id_n$ so that the identifier can be considered as numbers $id_1=1, \ldots, id_n=n$.

1.3.2.2.2. $update\_its\_cj(id_i , p{:}cj){:}()$ if the changeability attribute of this end-point is *changeable* and $id_i \in ID(ac\_id.cj)$. The function changes the value of $c.its\_cj(id_i)$ to new value of *p*.

1.3.2.2.3. $add\_to\_its\_cj(id_i, p{:}cj){:}()$ if the changeability attribute of this end-point is *add_only* or *changeable* and multiplicity is not a fixed number $n \in \mathbf{N}$. Then the new set of the identifiers is $ID'(ac\_id.cj) = ID(ac\_id.cj) \cup \{id_i\}$. If the changeability attribute of this end-point is *frozen*, then this operation can appear only in a constructor body.

1.3.2.2.4. $delete\_from\_its\_cj(id_i{:}cj){:}()$ if the changeability attribute of this end-point is *changeable* and multiplicity of this end-point is not a fixed number $n \in \mathbf{N}$. If the changeability attribute of this end-point is *frozen*, then this operation can appear only in a destructor body.

1.3.2.2.5. *is_in_cj(p:cj):boolean* The function determines whether an object referred as *p* belongs to this association end, i.e. that the value of *p* is a value of some identifier from *ID(ac_id.cj)*.

These implicitly defined operations are public (i.e. can be invoked from objects of different classes). An attribute *its_cj.mult* (in class *c*) specifies the multiplicity of the association end attached to the class *cj* with the association root *c*.

1.3.3. For each class *c* with direct successors under the aggregate or composite relation to class *c1*, ..., *cn* we require that preprocessing defines operations *create_cj()*. If *c* is a composite class, we require these operations to be private. If *cj* α *c* with $\alpha \in \{\perp w, \leftarrow w\}$ (*c* is the root of an aggregation), then we require *create_cj()* to be public.

1.3.4. For each class *c* with direct successors under the aggregate or composite relation to class *c1*, ..., *cn* we require that preprocessing defines operations *destroy_cj(ref:cj)*. If *c* is a composite class, we require these operations to be private. If *c* is an aggregation, then we require these to be public.

1.3.5. For each class *c* with direct generalisation classes *c1*,…,*cn* ($c \lessdot ci$ for $1 \le i \le n$) we require that preprocessing defines protected operations *create_ci(ref:ci)* and *destroy_ci(ref:ci*).

Getting rid of generalisation

1.3.6. For each class *c* we create the implicit attributes *uplinks_number* of type integer and *uplink.1*, ..., *uplink.n* of types pointer to classes c_1, ...,c_n, respectively, where $\{c_1,...,c_n\}=\{c' \mid c \lessdot c'\}$ is the set of all immediate superclasses for *c*. For each class *c* we also create one implicit attribute *downlink* of type *c*.

1.3.7. Consider a subclass *c*, and let *{c1, ..., cn}* be all classes s.t. $c \lessdot cj$ (n≥1). We preclude any user defined constructor *c.construct(ref)* (invoked from *create_c(ref)*) by the sequential composition of the following action catering for the recursive creation of the inherited parts.

uplinks_number := n;

if *(ref=nil) downlink:=self* ***else*** *downlink:=ref* ***endif;***

for *j=1, ..., n* ***do***

uplink.j := create_cj(downlink);

endfor

For all other classes (without generalisation relation), we add actions

uplinks_number := 0;

if *(ref=nil) downlink:=self* ***else*** *downlink:=ref* ***endif;***

to their constructor body.

1.3.8. We preclude any user defined destructor body *c.destruct()* (invoked from *destroy_c(ref:c)*) by the sequential composition of the following action catering for the recursive deletion of the inherited parts.

for *j=1, ..., uplinks_number* ***do***

destroy_cj(uplink.j);

endfor

1.3.9. Each navigation expression *a0.a1*.....an*.a* such that *an** is an object of a class *c* with $c < c'$, $a \in c'.attr$, and $a \notin c.attr \cup \{c_0.attr \mid c < c_0 < c'\}$, we replace with the expression *a0.a1*...an*.uplink.i....uplink.j.a*, where o_n = *an*.uplink.i.....uplink.j* is an object of class *c´*.

1.3.10. We first change expressions *a0.a1*.....an*.!op0(b1,...,bk)* specifying virtual operation calls (*op0.virt = virtual*) replacing them with *a0.a1*.....an*.downlink!op0(b1,...,bk)*. After that we apply delegation algorithm to all operations as follows.

1.3.11. Each expression for an operation call *a0.a1*.....an*.!op0(b1,...,bk)* such that *an** is an object of a class *c* with $c < c'$, $op0 \in c'.op$, and $op0 \notin c.op \cup \{c_0.op \mid c < c_0 < c'\}$, we replace with the expression *a0.a1*...an*.uplink.i....uplink.j.!op0(b1,...,bk)*, where $o_n = an*.uplink.i.....uplink.j$ is an object of class *c´*.
Here the condition $op0 \in c.op$ means that there is an operation in class *c* with the signature corresponding to that specified by the calling *op0*.

1.3.12. Each expression for an explicit operation call *a0.a1*.....an*.!c::op0(b1,...,bk)*, where $c < c'$ for some *c´* such that $op0 \in c'.op$, and $op0 \notin c.op \cup \{c_0.op \mid c < c_0 < c'\}$, we replace with the expression *a0.a1*...an*.uplink.i....uplink.j.!op0(b1,...,bk)*, where $o_n = an*.uplink.i.....uplink.j$ is an object of class *c´*.

1.3.13. Since we require that every specialised class must specify its own statechart (the same as one from the generalised class or completely overwritten), we do not need to change expressions with signal emission, because we do not delegate them to the generalised object.

1.3.14. To support polymorphism, we modify the assignments *a*:= *expr* (where $a.type \in C$) of the following three kinds of expressions (which return references as results):

- *expr = a1.a1*.....an*.a** (navigation expression), then *expr.type = a*.type*.
- *expr = create_c'()*, then *expr.type=c'*.
- *expr = a0.a1*.....an*.!op(a1,...,ak)* (operation call with non-nil return value), then *expr.type=op.type*

If $expr.type \neq a.type$, we replace assignments *a*:= *expr* with *a:=expr.uplink.i....uplink.j*, where *o* = *expr.uplink.i.....uplink.j* is an object of class *a.type*.

1.3.15. We also modify assignments of references *nav_expr* := *a* (where *nav_expr* is a navigation expression *a1.a1*.....an*.a** and $a.type \neq nav\_expr.type = c \in C$) by adding uplinks: *nav_expr := a. uplink.i.....uplink.j*, where *o* = *a.uplink.i.....uplink.j* is an object of class *nav_expr.type*.

Getting rid of composites

1.3.16. The semantic difference between weak and strong aggregation is compiled away in this preprocessing step. For weak aggregation, creation of constituents, initialisation of association ends, and destruction rests with other objects – there is no pre-defined support for these. In contrast, for composite objects, we will ensure by extending the entry- and exit-script of a compound class, that all children with bounded multiplicity are created, that association ends are initialised, and that all parts are destroyed upon destruction of the whole.

1.3.17. Let $\alpha \in \{\perp, \hookleftarrow\}$. Consider a compound class *c*, and let *{c1,...,cn}* be all classes s.t. *cj* α *c* . We preclude any user defined constructor body *c.construct* by the sequential composition of the following action catering for part-creation. We require that the user defined constructor caters for all other role initialisations.

```
for j=1,...,n do
  if (its_cj.mult ∈ N)
        if (its_cj.mult=1)
              { init_ its_cj(id1:= create_cj()); if (α =⊥) its_cj.its_c := self endif }
        else
                    for r= 1,..., its_cj.mult do
```

```
                                { add_to_its_cj(id(r), p:= create_cj);
                                  if (α =⊥) its_cj(id(r)).its_c := self  endif;
                                  r:= r+1}
                                endfor
                        endif
                else its_cj:= nil;
        endif                   endfor
```

1.3.18. Let $\alpha \in \{\perp, \hookleftarrow\}$. Consider a compound class *c*, and let *{c1,...,cn}* be all classes s.t. *cj* α *c* . We include as postclude for any user defined destructor body *c.destruct* the following action.

```
for j=1,...,n do
   if (its_cj.mult=1) destroy(its_cj)
   else
        for (is_in_cj(ref)) do destroy(ref)        endfor
   endif                    endfor
```

1.4 Formal Semantics of the Kernel Language

We now give a formal semantics for the reduced kernel language, assuming the pre-compilation from Section 1.3, which implements the design decisions elaborated above.

We will use symbolic transition systems as proposed by Z. Manna and A. Pnueli [3] as the formal framework for capturing the semantics. In this approach, the state-space of the transition system is spanned by a set of typed variables, called *system variables*. The transition relation itself is represented *symbolically* by first-order predicates, relating the future state of the system variable (expressed by primed versions of system-variables) to the original state. The behaviour of a symbolic transition system is represented through the set of traces of variable valuations.

Symbolic transition systems

S = (V , Θ, ρ)
- V — typed set of variables
- Θ — initial condition on variables
- ρ — transition relation on valuations of variables typically defined by first-order predicates over V, V´

traces(S)
set of infinite sequences of valuations of variables satisfying:
- first valuation matches Θ
- successor valuations satisfy ρ

We thus associate with each UML model *M* a *symbolic transition system* *S = STS(M)* capturing asynchronous concurrent execution of *M* using an interleaving semantics.

We first explain the set of system variables used to capture the semantics of our kernel model, and then give the formal definition of the transition relation.

System Variables

S uses only two system variables V= {*sconf, lchan*}, which, however, have a complex structure.

1.4.1. The system variable *sconf* captures the *current system configuration*. This system variable defines for each class *c* and each of its instances *i* its current instance configuration. Fig. 11 illustrates the structure of the system configuration.

1.4.2. We will identify the set of instance identifiers with *N*, reflecting the fact, that there is no à priori bound on the number of instances of a class.

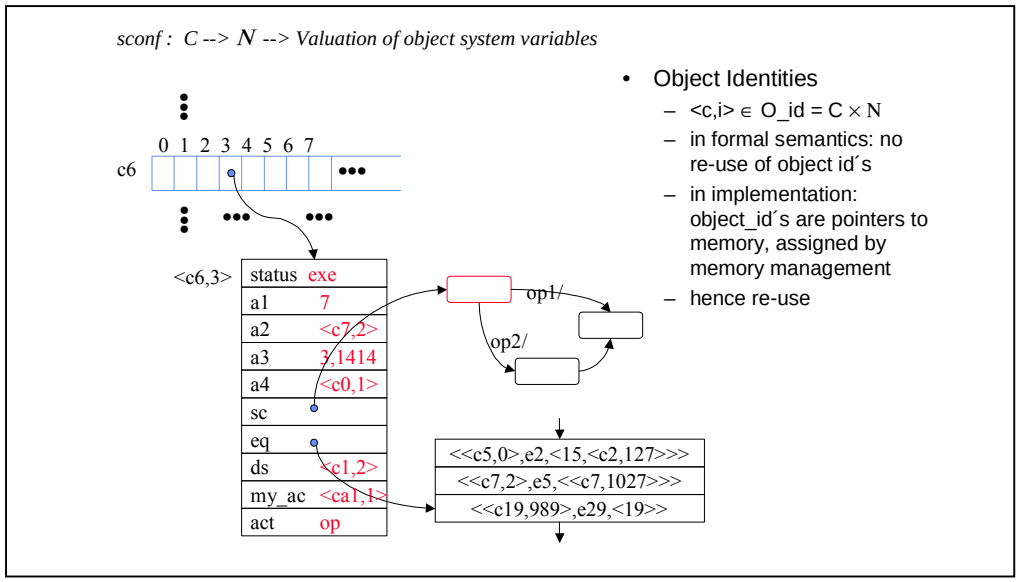


Figure 11. The system configuration

1.4.3. For each object *o* we collect the following pieces of information in its instance configuration.

1.4.3.1. *o.status ∈ {dormant, executing, suspended, call_completed, dead}*

1.4.3.1.1. initially the instance is either dormant or executing

1.4.3.1.2. creation of a new object of class *c* will choose an instance-id *i* and set its status to *executing*; by requirement on the initial state, the object will thus also be stable;

1.4.3.1.3. when an instance is stable, it can accept a waiting signal (dispatched by an active object) from the signal queue or a pending operation and initiate its execution, keeping status *executing*

1.4.3.1.4. when executing an operation call, the instance will become *suspended* , until the call has been served

1.4.3.1.5. when the result of call becomes available, the serving object (callee) switches the status of the driver to *call_completed*, the driver will pick up the result, and return to state *executing*

1.4.3.1.6. when the object is killed, its status becomes *dead*

1.4.3.1.7. Fig. 12 gives an overview of the different states an object and their interrelationship

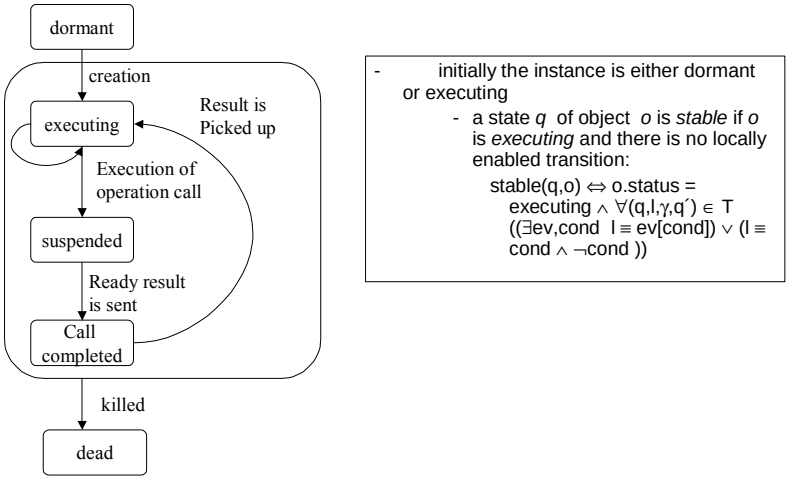


Figure 12. The object states during run-time

1.4.3.2. *o.a* gives the current value of attribute *a*

1.4.3.3. *o.sc* gives the current state configuration of *o*, if *o* is an instance of a reactive class *c*, i.e. *c = cl(o)*. For the simple case of flat UML state-machines, this degenerates to a single state *q* of *sm(c).*

1.4.3.4. *o.eq* is the signal queue associated with an active object *o* where *hd(o.eq)* points to the top signal in the queue.

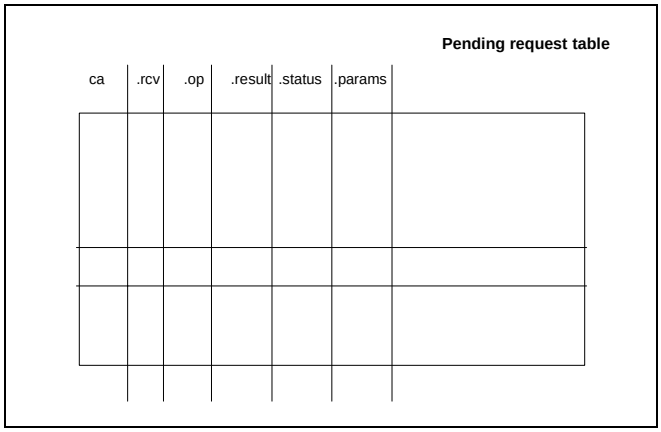
1.4.3.5. *o.ds* specifies the object performing service (dispatcher) at the current moment. This attribute is intended to be used by active objects to control the flow of computations.

1.4.3.6. *o.my_ac* gives the name of the associated active object controlling the current thread.

1.4.3.7. *o.act* keeps the specification of an action (operation or signal acceptance) currently being performed in the object *o*.

1.4.4. The second system variable, *lchan:* (O_id × O_id × op) → *{sw_on, sw_off, nill}*, specifies logical channels mentioned in 1.2.23 for synchronous communication (operation calls). A channel *lchan(o1,o2, op*) is switched on when object *o1* calls operation *op* from object *o2* and object *o2* is ready to accept this call:
lchan(o1,o2, op) = *sw_on* ⇔ **[** (*op = o1.act)* & (*o1.status=suspended*) & $ready_{sconf}$*(o2, o2.sc, op)* **]**
A channel *lchan(o1,o2, op*) is switched off when object *o2* completes the call (becomes stable) of operation *op* and sends the results to object *o1* : by changing status of *o1*, object *o2* lets him know that operation was completed:
lchan(o1,o2, op) = *sw_off* ⇔ **[** (*op = o1.act)* & (*o1.status=call_completed*) & $stable_{sconf}$*(o2.sc, o2)* **]**

1.4.5. One possible realisation of logical channels is via *pending request table* – variable capturing all pending operation requests during model execution in a global table.



1.4.6. Here we exploit the fact, that all objects become suspended when executing an operation call, and can thus model the table as a set of entries such that for each object there is at most one issued pending request. This allows to organise these as a table, which for each object or actor has either a nil value, if there is currently no pending request, or an entry specifying the kind and status of the request.

1.4.7. Each pending request maintains the *id* of the receiver, the name of the requested operation, the list of parameters, a result-field, and status information.

1.4.8. Letting *ca* range over Actors and Object_ids, we thus maintain non-nil entries *prt.ca* collecting the following information:

1.4.8.1. $prt[ca].rcv \in A \cup O\_id$ identity of the receiver

1.4.8.2. *prt[ca].op* the identity of the requested operation

1.4.8.3. *prt[ca].result* the return value of the call; only valid if status of request is completed

1.4.8.4. $prt[ca].status \in \{pending, busy, completed\}$

1.4.8.4.1. whenever the caller emits an operation call, its table entry is updated by entering all information pertinent to the call and setting its status to pending

1.4.8.4.2. once the receiver object has picked up the call, it changes the status to busy

1.4.8.4.3. once the receiver object has completed the call, it updates the result entry and changes the status to completed

1.4.8.4.4. once the caller has picked up the result, it changes the entry to nil

1.4.8.5. *prt[ca].params* is the parameters of the operation call

Definition of the Transition Predicate

1.4.9. We structure the transition system as a disjunction of the transition relation for all objects and actors, modelling the asynchronous interleaved execution of active objects.

1.4.10. This leads to the following overall structure of the transition predicate ρ from the symbolic transition system STS(M). The different clauses are elaborated in subsequent paragraphs below. In the following, primed (configuration) attributes of an object specify new value of the corresponding unprimed attributes.

$\exists o \exists (q, \alpha, \gamma, q') \in sm(cl(o)).T$

```
        o.sc = q ∧
        { -- case splitting by object and transition assuming state q
        [       o.sc´ := q´ ∧    --      transition relation for taking steps
        ∧
          {     Φ_<accepting signal or operation call>
         ∨      Φ_< unstable states>
         ∨      Φ_<picking up the result of a call>
          }
        ∧       Φ_<bookkeeping when becoming stable>
        ]
∨       [       o.sc´ := q ∧ o.status = executing       -- transition relation not leaving the state
        ∧       {Φ_<discarding or deferring signals>
        ∨       Φ_<initiating operation call>     }     ]     }
```

1.4.11. Accepting a signal or a call: Φ_<accepting signal or operation call> ⇐def⇒

```
        stable_val(q,o) ∧
{
∨       [       (α ≡ sg1(p1,...,pn)[guard] ∧ sg ≤ sg1 ∧ val(guard)   -- accepting a signal
        ∧       hd(my_ac(o).eq = <o,sg,<a1,...,am>> ∧ n≤m
        ∧       my_ac(o).ds = nil)
        ⇒       (my_ac(o).ds´ := o
        ∧       [ o.pj´ := aj | j ∈ { 1,...,n}])
        ]
∨       [       α ≡ op1(p1,...,pn)[guard] ∧ guard   --      accepting a call event
        ∧       ∃o1
                [       --      pick up a call from caller o1
        ∧       prt[o1].rcv = o
        ∧       prt[o1].op = op1
        ∧       prt[o1].status = pending
        ∧       prt[o1].params = <a1,...,an>
        ∧       [ o.pj´ := aj | j ∈ { 1,...,n}]
        ∧       prt[o1].status´ := busy
                ]
        ∧       prt[o1].result´ := nil
        ]
}
```

1.4.12. Processing unstable states: Φ_< unstable states> ⇐def⇒

```
        o.status = executing ∧
{ [     γ≡ a := exp     --      assigning an attribute
        ⇒       o.a´ := val(exp,o)
        ]
∨       [       γ≡ a0!sg(a1,...,an)     --      emitting a signal
        ⇒       my_ac(o.a0).eq´ := insert(<o.a0,sg,<a1,...,an>>,eq>
        ]
∨       [       γ≡ return(a)    --      setting return value
        ∧       ∀ o1: [(prt[o1].rcv = o
        ∧       prt[o1].status = busy)
        ⇒       (prt[o1].status´ = completed
        ∧       (o.status´ = call_completed
        ∧       prt[o1].result´ = a)
        ]
```

```
∨       [       γ≡ a:= create_c   --      creation of an object
        ⇒       (∃ i s.t. sconf(c)[i].status = dormant
        ∧       sconf(c)[i]´.status = executing
        ∧       o.a´:= i)
∨       [       γ≡ destroy(a)     --      killing of an object
        ∧       sconf(c) [o.a].status´ := dead  ]
}
```

where *val(exp,o)* replaces any occurrence of a local attribute $\hat{a}$ occurring in exp by $o.\hat{a}$.
(Note that one-level de-referencing of the form a1.a2 is subsumed as a special case of the first clause)

1.4.13. Picking up the result of an operation call Φ_<picking up the result of a call> ⇐def⇒

```
        o.status = call_completed ⇒
{ [     γ≡ a := a0!op(a1,...,an)    --      of a function call
        ⇒       prt[o]´ = nil
        ∧       o.status´ = executing
        ∧       o.a´ = prt[o].result
        ]
∨       [       γ≡ a0!op(a1,...,an)         --      of an operation call
        ⇒       prt[o].status´ = nil
        ∧       o.status´ = executing
        ]
}
```

1.4.14. Bookkeeping when becoming stable Φ_<bookkeeping when becoming stable> ⇐def⇒

$stable_{val'}(q,o) \Rightarrow$

```
{
[       --      becoming stable after evaluating a signal
        my_ac(o).ds = o
        ⇒       my_ac(o).ds´ = nil
        ]
∨       [       --      becoming stable after an operation call
                ∀o1     [       [prt[o1].rcv = o
                        ∧       prt[o1].status = busy]
                        ⇒
                                [       prt[o1].status´ := completed
                                ∧       o1.status´ := call_completed ]
        ∧       [ ∃o1 : (prt[o1].rcv = o
                ∧       prt[o1].status = pending)
                ⇒       prt[o1].status´ = busy  ]
                        ]
}
```

1.4.15. Discarding signals Φ_<discarding or deferring signals> ⇐def⇒

$stable_{val}(q,o)$

```
        ∧       hd(my_ac(o).eq) = <o,sg, - >
        ∧       my_ac(o).ds = o
        ∧       ∀(q,α,χ,q´) ∈ sm(cl(o)).T :     ¬ (α ≡ sg[-]∨ sg[-] < sg1[-] ≡ α)
        ∧       {
                (sg[-] ∉ sm(cl(o)).D(q)  ∧ my_ac(o).eq´ := tail(my_ac(o).eq)  )
        ∨       (sg[-]  ∈ sm(cl(o)).D(q) ∧ my_ac(o).eq´ := pass_queue(<o,sg[-]>, tail(my_ac(o).eq) )
                }
```

where the function pass_queue moves the reference to the next queue element leaving the deferred event(s) at the same place. The function can be implemented so that it moves each deferred event to the end of the queue.

1.4.16. Initiating an operation call Φ_<initiating operation call> ⇐def⇒

∧ {γ≡ a := a0!op1(a1,...,an) ∨ γ≡ a0!op1(a1,...,an)}
∧ o.status = executing
∧ o.status´ = suspended
∧ prt[o].rcv´ := o.a0
∧ prt[o].op´ := op1
∧ prt[o].params´ := <o.a1,...,o.an>
∧ prt[o].result´ := nil
∧ [[(∃ o1: prt[o1].rcv = o.a0 ∧ prt[o1].status ≠ nil)
⇒ prt[o].status´ = pending]
∨ (∀ o1: prt[o1].rcv ≠ o.a0
⇒ prt[o].status´ = busy)]

It is easy to see that the pending request table together with the transition predicate defined over its implement the mentioned logical channels, where:

$$lchan(o1,o2,op1) = sw\_on \Leftrightarrow$$
$$(prt[o1].rcv=o2 \ \& \ prt[o1].op=op1 \ \& \ prt[o1].status=pending \ \& \ prt[o1].status´=busy)$$
and
$$lchan(o1,o2,op1) = sw\_off \Leftrightarrow$$
$$(prt[o1].rcv=o2 \ \& \ prt[o1].op=op1 \ \& \ prt[o1].status=busy \ \& \ prt[o1].status´=completed)$$

Initial condition on variables

For the described symbolic transition system STS(M)=({*sconf, lchan*}, Θ, ρ), the initial conditions Θ is defined as follows. At the beginning of the model execution all logical channels are empty and only an object of the root class is created

$$\Theta\ (lchan) = nil$$
$$\forall o \in O\_id : \Theta(o.status)=dead \Leftrightarrow cl(o) \neq c0$$ (where *c0* is the root class)
$$\forall o \in O\_id : (\Theta(o.status)=executing \wedge \Theta(o.act)=c0.entry) \Leftrightarrow cl(o) = c0$$

2 UML Statecharts

A *statechart* (also called *state machine*) is defined in the scope of a class *c* and hence it inherits the set of attributes *c.attr*, the set of operations *c.op*, and the set of signals *c.sig*.

2.1 Constituents of Statecharts

A statechart *SC* consists of a set *vertices(SC)* of (hierarchical*) state vertices* and a set *trans(SC)* of *transitions*.

State Vertices

2.1.1 A state vertex may be a *state* or a *pseudostate* (which is a synch state or a stub state), i.e. *vertices(SC)* = *states(SC)* ∪ *pstates(SC).*

2.1.2 A state can be *simple*, *final* or a *composite state*. Final state specifies the termination, i.e. destruction of the object. A composite state can be *concurrent* or not. A composite state *s* has a set of *direct substates*, denoted by *child(s)*. For a simple or final state *s*, *child(s)* = ∅. For a state *s'* ∈ *child(s)*, *s* is also called the *father* of *s'*, denoted as *father(s')*. Direct substates of a concurrent composite states are called *regions*. A concurrent state will also be called an *AND-State* (a system is in an AND-state if it is in all of its direct substates concurrently) and a composite state which is not concurrent will be called an *OR-state* (the system is in an OR-state if it is in one of its direct substates). Fig. 13 shows an AND-state *s* which is composed of three concurrent regions *s1*, *s2*, and *s3*. On the other hand Fig. 14 shows an OR-state *s* with three direct substates *s1*, *s2*, and *s3*.

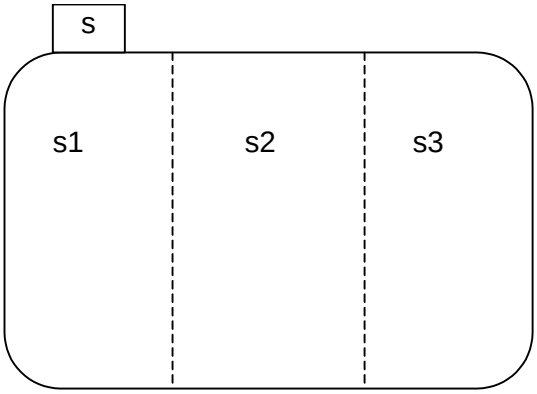


Figure 13. AND-state (concurrent composite state)

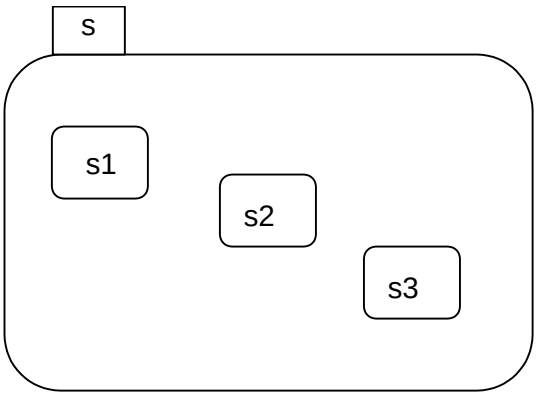


Figure 14. OR-state (non-concurrent – also called sequential – composite state)

2.1.3 We will use a function *mode* to identify the type of a state:

$$mode : states(SC) \rightarrow \{ \text{SIMPLE, FINAL, AND, OR} \}$$

2.1.4 Every state machine has a distinguished *top* state *top∈ states(SC)*. We assume that the top state is of mode OR [1].

2.1.5 A submachine-state is only a syntactical abbreviation and hence will not be considered in this paper. Furthermore StubStates are used to reference states within a submachine-state and hence are also not considered within this paper.

2.1.6 A pseudostate can be one of the following kinds:

- An *initial* pseudostate represents a default vertex that is the source of a single transition to the *default state* of a non-concurrent composite state.
- *DeepHistory* is used to store the most recent active configuration of the composite state that directly contains this pseudostates. This includes not only the information on the most recent direct substate but also the most recent substates of that substate etc.
- *ShallowHistory* is used to represent the most recent active substate of the composite state that directly contains this pseudostate.
- *Join* vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards and trigger events.
- *Fork* vertices serve to split an incoming transition into two or more transitions terminating on orthogonal regions. The transitions outgoing from a fork vertex must not have guards or trigger events.
- *Junction* vertices are used to chain together multiple transitions. They are used to construct compound transition paths between states. Guards are evaluated statically before performing a complete compound transition.
- *Choice* vertices which, when reached, result in dynamic evaluation of the guards of its outgoing transitions It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step.

2.1.7 Pseudostates come up with a function *kind*, determining its type:

$$kind : pstates(SC) \rightarrow \{initial, deepHistory, shallowHistory, join, fork, junction, choice\}$$

2.1.8 Instead of using join and fork vertices we will consider transitions with more than one source and one target state vertex. Junction are only used as an abbreviation and hence will not be considered here, too.

2.1.9 Choice vertices will not be handled here.

2.1.10 An OR-state *s* has a default substate *default(s) ∈ child(s)*. This default substate is defined as the target state of the transition starting at the initial vertex of the composite state *s*.

2.1.11 An OR-state *s* may contain one shallowHistory vertex, *sHist(s)*, and at most one deepHistory vertex, *dHist(s)*. These two kinds of pseudostates are also called *history connectors*.

2.1.12 For a history connector *h*, the enclosing OR-state will be denoted by *state(h)*.

2.1.13 The set of state vertices are ordered in a tree-like structure with *top* as its root and where the set *child(s)* gives the successors of a node *s* in the tree.

2.1.14 The depth of a state *s* w.r.t. the state hierarchy is inductively defined as

$$depth(s) := \begin{cases} 0 & \text{if } s = top \\ depth(\, father(s)) + 1 & \text{otherwise} \end{cases}$$

- [1] UML only requires that the top state is a composite state. If the top state is an AND-state we can simply add an additional enclosing OR-state without changing its behaviour.

2.1.15 The substate relation defines a partial order on the set of states:

$s \leq s$
$s \in child(s')$ then $s' \leq s$
$s \leq s'$, and $s' \leq s''$ then $s \leq s''$
$s < s'$ iff $s \neq s'$ and $s \leq s'$.
$top \leq s$
$s' \leq s$, and $s'' \leq s$, then $s' \leq s''$ or $s'' \leq s'$

For a set of states S and $s \in states(SC)$ we will write $s < S$ iff $s < s'$ for all $s' \in S$. We also will say that state $s´$ is younger that state s iff $s < s'$.

2.1.16 A *state configuration sc* is a set of state vertices with the following property:
- $top \in sc$
- if $s \in sc$ and s is an AND-state then $child(s) \subseteq sc$
- if $s \in sc$ and s is an OR-state then there exists exactly one $s' \in child(s)$ with $s' \in sc$

2.1.17 Furthermore, a state s may have associated a set of *deferred events*, an *entry*, *do*, and *exit* action.

deferred(s)	Denotes the set of deferred events in state *s*
entry(s)	Denotes the entry action of state *s*
do(s)	Denotes the do-activity executed while staying in state *s*
exit(s)	Denotes the exit action of state *s*

2.1.18 An event that is deferred in a composite state is automatically deferred in all directly or transitively nested substates.

2.1.19 Restrictions:
We will not support do-activity. The idea of do-activity is to invoke a concurrent computation which can be interrupted at any time. As we will support only one thread of control within a statechart and give the semantics of statecharts w.r.t. run-to-completion steps, do-activities will not be considered.

Transitions

A transition in a UML statemachine has only one source state vertex and only one target state vertex. To model more complex transitions, UML provides join and fork vertices. In this version, instead of the pseudostates of kind join and fork we will consider more general transitions having a set of source states and a set of target states. (cf. Fig. 15)

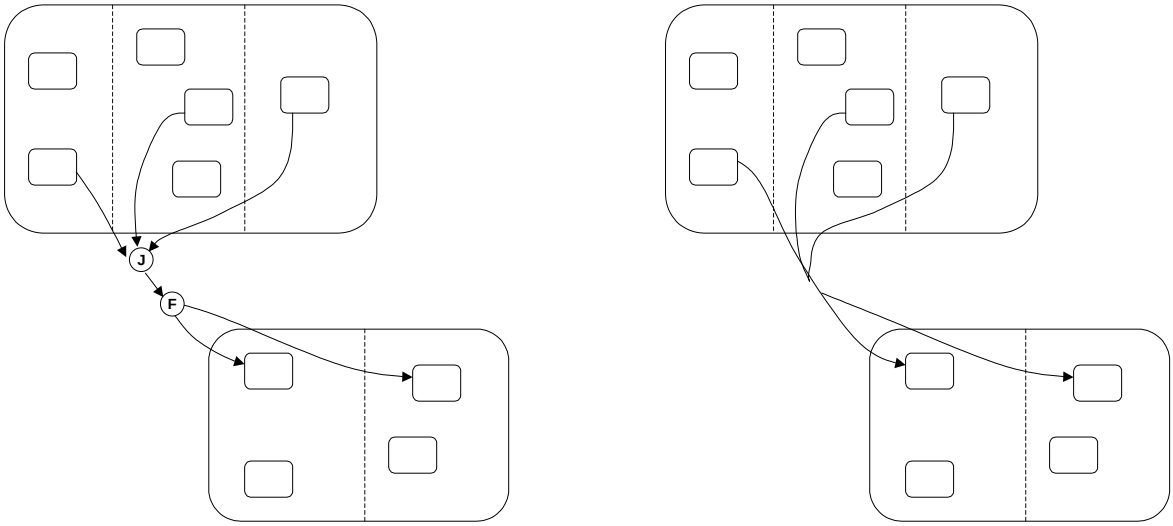


Figure 15. Modelling Join and Fork vertices as transitions with multiple sources and targets

2.1.20 A transition *t* is associated with

source(t) : a non-empty set of states denoting the source states of transition *t*.
target(t): a non-empty set of state vertices denoting the target states of transition *t*. Besides states, history connectors may be also contained in this set.
trigger(t): an optional trigger event (with parameters), which has to occur to enable transition *t*. We will use a pseudo trigger NONE to indicate that a transition has no trigger event.
guard(t): a guard expression. Transition *t* can only be executed if the guard evaluates to true.
effect(t): an action which will be executed when performing transition *t*.

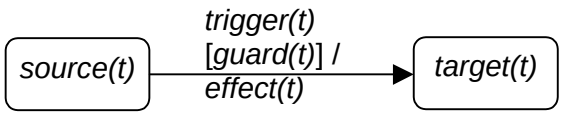


Figure 16. The ingredients of a transition

2.1.21 Different states from the set *source(t)* must belong (directly or transitively) to different regions of an AND-state (orthogonal substates) and the states of *target(t)* must also belong to different regions (orthogonal substates).

2.1.22 A trigger can specify reception of a signal event (asynchronous communication) or a call event (for triggered operation call) together with parameters which can be used in the guard and the following actions.

2.1.23 The top state cannot be the source or the target of a transition.

Well-Formedness Rules

Here, we will define a collection of auxiliary notions which will be used to define well-formedness conditions on statecharts, and which are necessary to define the effect of firing a transition. We also introduce the concept of configurations describing maximal subset of states allowed to be concurrently active. We will first define the smallest region where changes, due to the execution of a transition, may occur.

2.1.24 The *least common ancestor lca(S)* of a non-empty set *S* of states defines the closest state (w.r.t. transitive containment of substates) which subsumes all states of *S*. As the root state *top* is the largest ancestor of every state, *lca(S)* will exist for every subset *S* of states. It is defined by

1. $lca(S) \leq S$ ($lca(S)$ is an ancestor of every state of S) and
2. $\forall\, \bar{s} \in states(SC)$ with $\bar{s} \leq S : \bar{s} \leq lca(S)$ ($lca(S)$ is minimal w.r.t. containment of other states, i.e. youngest ancestor).

2.1.25 The *least common OR-ancestor* $lca^+(S)$ of a non-empty set S of states defines the youngest OR-state (i.e. minimal w.r.t. containment of other states) which subsumes all states of S and is not contained in S itself. If the least common ancestor is an OR-state not contained in S this is also the least common OR-ancestor, otherwise, we pick the closest OR-state above the least common ancestor. As we require that the top state is an OR-state, the least common OR-ancestor exists for every subset of states not containing the top state. If $top \in S$, then we define $lca^+(S) = top$. Hence, the least common OR-ancestor is defined as follows

$$lca^+(S) := \begin{cases} top, & \text{if } top \in S; \\ s, & \text{if } top \notin S \text{ and } s < S \text{ and } mode(s) = \mathsf{OR} \text{ and} \\ & \forall \bar{s} \in states(SC) : mode(\bar{s}) = \mathsf{OR} \text{ and } \bar{s} < S \Rightarrow \bar{s} \leq s \end{cases}$$

2.1.26 Two states s and s' are *orthogonal*, denoted by $s \perp s'$, if they belong (directly or by transitivity) to different regions of an AND-states, i.e. they are not comparable w.r.t. the $child^*$ relation and their common ancestor is an AND-state.

$$s \perp s' \text{ iff } \neg (s \leq s' \text{ or } s' \leq s) \text{ and } mode(lca(\{s, s'\})) = \text{AND}.$$

A set S of states is called *orthogonal*, denoted by $\perp(S)$, if the states of S are pairwise orthogonal.

2.1.27 A set of states $S \subseteq states(SC)$ is called *consistent*, denoted by $\downarrow(S)$ iff every two states s, s' of S are either related by the $child^*$ relation – i.e. $s \leq s'$ or $s' \leq s$ – or orthogonal.

2.1.28 A *state configuration sc* is a maximal consistent set of states.

2.1.29 The *scope* of a transition t, denoted by $scope(t)$, is the smallest range of states which is affected by firing the transition t. It is defined as the OR-state which is the lca^+ of the source and target states of the transition. As the target of a transition may also contain some history connectors, we replace these pseudo states by their enclosing OR-states to compute the common ancestor. For this aim, we introduce the transcription function

$$st: 2^{states(SC) \cup hist(SC)} \rightarrow 2^{states(SC)},$$

defined as follows (where $hist(SC) \subseteq pstates(SC)$ denotes the set of history connectors):

$$\forall\, S \subseteq states(SC) \;\&\; C \subseteq hist(SC) : \; st(S \cup C) = S \cup \{state(h) \mid h \in C\}$$

then the scope of a transition t is

$$scope(t) = lca^+\big(source(t) \cup st(target(t))\big).$$

2.1.30 Given a consistent set $S \subseteq states(SC)$, the *default completion*, denoted by $dcompl(S)$, is the smallest set S' such that

- $S \subseteq S'$,
- If $s \in S'$ and $s \neq top$, then $father(s) \in S'$,
- If $s \in S'$, $mode(s) = \text{OR}$ and $child^+(s) \cap S = \emptyset$, then $default(s) \in S'$,
- If $s \in S'$ and $mode(s) = \text{AND}$, then $child(s) \subseteq S'$.

The *partial default completion* below a given state s is given by

$$pdcompl(s) = dcompl(\{s\}) \cap \{ s' \mid s \leq s' \}$$

The completion of a consistent set of states w.r.t. history connectors will be defined below.

2.1.31 Two transitions are *consistent* if they are active in two orthogonal regions, i.e. if their scopes are orthogonal.

$$\downarrow(t1, t2) \text{ iff } scope(t1) \perp scope(t2)$$

This notion can be extended to a set of transitions. A set T of transitions is *consistent*, denoted by $\downarrow(T)$ iff the transitions of T are pairwise consistent.

2.1.32 If a set of possible executable transitions is not consistent, we will use an assignment of transition priorities to select a consistent subset. The *priority* of a transition is specified by the depth of its innermost source state:

$$prio : T \rightarrow \mathrm{N}$$
$$prio(t) = \max\{\, depth(s) \mid s \in source(t) \,\}$$ [2]

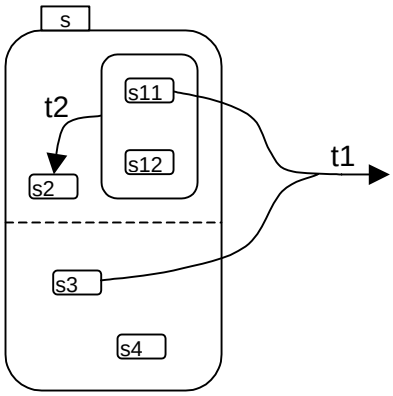


Figure 17. Priority of transitions: t1 has a higher priority than t2

2.1.33 A statechart SC is *well-formed* – denoted $wff(SC)$ – iff for all transitions t the following holds

- $\downarrow(source(t))$ and $\downarrow(st(target(t))$
- $\forall s \in source(t) : mode(s) = \mathrm{OR} \Rightarrow \forall s' : s < s' \Rightarrow s' \notin source(t)$
- $\forall s \in target(t) \cap states(SC) : mode(s) = \mathrm{OR} \Rightarrow \forall s' : s < s' \Rightarrow s' \notin target(t)$
- $\forall h \in target(t) \cap hist(SC) : \forall s' : state(h) < s' \Rightarrow s' \notin target(t)$
- $top \notin source(t) \cup target(t)$

In the rest of this paper we will only consider well-formed statecharts.

Effects of History Connectors

2.1.34 A *history configuration hc* is a set S of states such that for every OR-state s set S contains a child $s' \in child(s)$. A history configuration *hc* can be also defined as a function over all OR-states

$$hc : \{\, s \mid s \in states(SC) \text{ and } mode(s) = \mathrm{OR} \,\} \rightarrow states(SC)$$
with $hc(s) \in child(s)$.

2.1.35 The default completion *dcompl* is extended by a function *hcompl* for handling the history connectors.

2.1.35.1 The *state completion* for a shallowHistory vertex h and a history configuration *hc* is defined by
$hcompl(h, hc) = \{state(h), hc(state(h))\}$

2.1.35.2 For a deepHistory vertex h and a history configuration *hc* , we define their state completion $hcompl(h, hc) = S$, where S is the smallest subset of states satisfying
$state(h) \leq S$
$state(h) \in S$

- [2] The priority of a transition is defined on its source state: a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states. The priority of joined transitions is defined by the priority of the transition with the most transitively nested source state.

if $s \in S$ and *mode(s)* = AND then *child(s)* $\subseteq S$
if $s \in S$ and *mode(s)* = OR then *hc(s)* $\in S$

2.1.35.3 For a set *H* of history connectors the *completion set* is given by

$$hcompl(H, hc) := \bigcup_{h \in H} hcompl(h, hc)$$

2.1.36 An *extended state configuration ecs* is a pair consisting of a state configuration and a history configuration *ecs* = <*sc*, *hc*>.

2.1.37 The execution of a transition *t* in an extended state configuration <*sc*, *hc*> with *source(t)* $\subseteq$ *sc* will lead to a successor configuration *esc'* = <*sc'*, *hc'*> which is defined by the following:

2.1.37.1 States which are exited when performing a transition *t* starting from a state configutarion *sc*:
exited(t, sc) = { $s \in sc$ | *lca(source(t))* $\leq s$ }.

2.1.37.2 The states entered after the execution of *t* from a state configuration *sc* w.r.t. function *hc*:
entered(t, sc, hc) = dcompl(sc \ exited(t, sc) $\cup$ *st(target(t))* $\cup$ *hcompl(hist(target(t)), hc))* \ *(sc \ exited(t, sc))*
where *hist(target(t)) = target(t)* $\cap$ *hist(SC).*

2.1.37.3 The successor state configuration *sc'* is given by
sc' = dcompl(sc \ exited(t, sc) $\cup$ *entered(t, sc, hc))*

2.1.37.4 And the new history configuration *hc'* is given by

$$hc'(s) = \begin{cases} s', & \text{if } s \in exited(t, sc) \text{ where } \{s'\} = sc \cap child(s); \\ hc(s), & otherwise \end{cases}$$

2.2 Flattening the Statechart

In chapter 1 the semantics of UML models is given with respect to a flat statechart. This was done to concentrate on the main semantical issues discussed in chapter 1 and to avoid an overloading of that chapter with the orthogonal concepts of hierarchical state machines. In this section we describe how to flatten a statechart without changing its behaviour.

Given a statechart *SC* the flattened statechart *flattened(SC)* is given by

2.2.1 *states(flattened(SC))* = {<*sc, hc*> | <*sc, hc*> is an extended state configuration of *SC*}

2.2.2 *trans(flattened(SC))* = { *t'* = <*t, sc, hc*> | *t* $\in$ *trans(SC)*, <*sc, hc*> is an extended state configuration of *SC*, *source(t)* $\subseteq$ *sc* } with

- *source*(<*t, sc, hc*>) = {<*sc, hc*>}
- *target*(<*t, sc, hc*>) = {<*sc', hc'*>} where <*sc', hc'*> is the successor configuration after executing *t* from the configuration <*sc, hc*>
- *trigger*(<*t, sc, hc*>) = *trigger(t)*
- *guard*(<*t, sc, hc*>) = *guard (t)*
- *effect*(<*t, sc, hc*>) = *exit(sc, t)*; *effect(t); enter(sc, hc, t)*

2.2.3 The initial state of *flattened(SC)* is given by <sc_0, hc_0>, where sc_0 is the initial state configuration obtained by the default completion of the top state

$$sc_0 = dcompl(\{top\})$$

and hc_0 is given by the default states

$hc_0(s) = default(s)$ [3]

Flattening the statechart we cannot associate a unique exit action (enter action) to a state of the obtained state machine. Therefore, the corresponding actions are lifted to the transition.

2.2.4 The action *exit(sc, t)* is given by a sequence of the actions *exit(s)*, where $s \in exited(t, sc)$. The order should be from innermost states to outermost states.

2.2.4.1 $exit(sc, t) = \alpha_1; \ldots; \alpha_n$ where
- n = *sizeof(exited(t, sc))*
- for each $s \in exited(t, sc)$ there exists an index $I(s) \in \{1, \ldots, n\}$ with $\alpha_{I(s)} = exit(s)$
- if $s \neq s'$ then $I(s) \neq I(s')$
- $s \leq s'$ then $I(s) \geq I(s')$

2.2.4.2 The action sequence *exit(sc, t)* can also be defined recursively by

2.2.4.2.1 *exit(sc, t) = exit(top, sc, t)*

2.2.4.2.2 *exit(s, sc, t) =*

```
if mode(s) = SIMPLE then
        if s ∈ exited(t, sc) then exit(s) else nil fi
else if mode(s) = OR then
        let {s'} = sc ∩ child(s)
        if s ∈ exited(t, sc) then exit(s', sc, t); exit(s) else exit(s', sc, t) fi
else  // mode(s) = AND //
        let child(s) = {s1, …, sk}
        if s ∈ exited(t, sc) then exit(s1, sc, t); …; exit(sk, sc, t); exit(s)
        else exit(s1, sc, t);…; exit(sk, sc, t) fi
```

Note that in the case of an AND-state the ordering of the actions of the orthogonal substates are arbitrary.

2.2.5 The action *enter(sc,hc, t)* is given by a sequence of the actions *enter(s)*, where $s \in entered(t, sc, hc)$. The order should be from outermost states to innermost states.

2.2.5.1 $enter(sc, hc, t) = \alpha_1; \ldots; \alpha_n$ where
- n = *sizeof(entered(t, sc, hc))*
- for each $s \in entered(t, sc, hc)$ there exists an index $I(s) \in \{1, \ldots, n\}$ with $\alpha_{I(s)} = enter(s)$
- if $s \neq s'$ then $I(s) \neq I(s')$
- $s \leq s'$ then $I(s) \leq I(s')$

2.2.5.2 The action sequence *enter(sc, hc, t)* can also be defined recursively by

2.2.5.2.1 *enter(sc, hc, t) = enter(top, sc, hc, t)*

2.2.5.2.2 *enter(s, sc, hc, t) =*

```
if mode(s) = SIMPLE then
        if s ∈ entered(t, sc, hc) then enter(s) else nil fi
else if mode(s) = OR then
        let {s'} = sc ∩ child(s)
        if s ∈ entered(t, sc, hc) then enter(s); enter(s', sc, hc, t)
                else enter(s', sc, hc, t) fi
else  // mode(s) = AND //
        let child(s) = {s1, …, sk}
        if s ∈ entered(t, sc, hc)  then
         enter(s); enter(s1, sc, hc, t); …; enter(sk, sc, hc, t)
                else enter(s1, sc, hc, t); …; enter(sk, sc, hc, t) fi
```

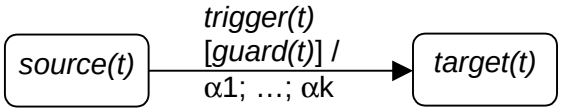
[3] This is a simplified approach. UML allows to specify an initial value for a history connector. Furthermore, without specifying an initial history state it is only allowed to enter an OR-state through the history connector whenever the state machine has been exited that OR-state sometimes in the past.

Note that in the case of an AND-state the ordering of the actions of the orthogonal substates are arbitrary.

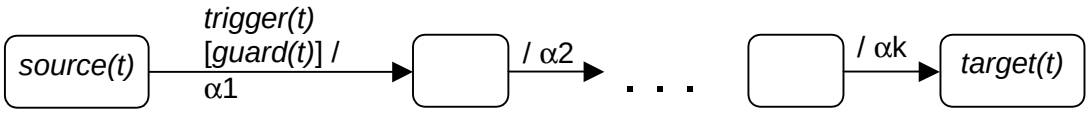
Splitting complex transitions

We define a transformation of statecharts in such a way that every transition can only perform one simple action (assignment, emitting an event, sending a reply etc.). Using this transformation we avoid that a system is blocked between two state configuration (waiting on the termination of an operation call).

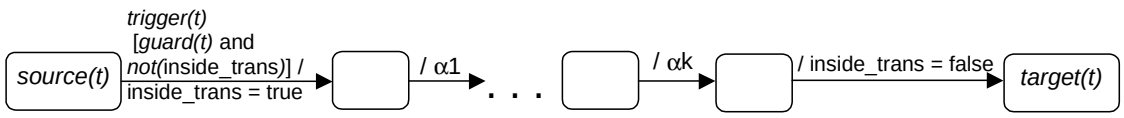
The basic idea is to replace



by



To avoid that some other transition is enabled when inside the execution of such 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 that another transition will be started when being in the middle of another one. That is, we perform the following transformation for each transition by introducing new states:



3 Summary: OMEGA-UML Restrictions

In this report a subset of UML – Omega-subset – is defined, for which a formal semantics is given. This semantics is defined at three levels. Section 1.2 has described informally an operational semantics (abstract level) of the chosen subset. Section 1.3 describes how the Omega-subset can be represented by more restricted subset of UML – Kernel language ("preprocessing semantics"). Finally, Section 1.4 gives a formal semantics for the Kernel language in terms of symbolic transition systems.

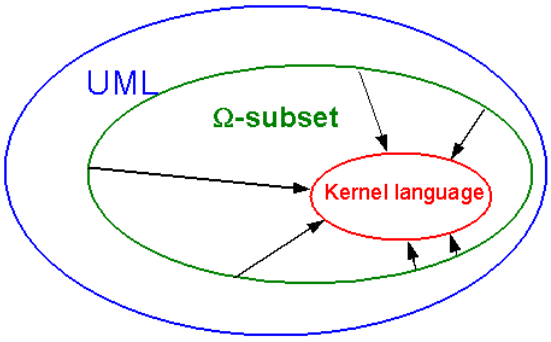


Figure 18. UML subsets

Note that the Kernel language is internal format, and Omega-subset (partially) specifies the language for costumers. Whereas the Kernel language is defined in Section 1.4 via variable *sconf*, the Omega-subset can be defined by the following list of restrictions.
First, for type usage we will allow only classes, enumerations, references to objects and such predefined types as integer, boolean, character. This types can be used for the constructors *array* and *records*.

3.1. Classes and Associations

3.1.1. No abstract classes (no abstract operations).
3.1.2. Currently, no stereotypes in the Omega-subset. Stereotypes can appear in extensions of the current language.
3.1.3. The only relations between classes: generalization, composition, aggregation, and (neighbour) association.
3.1.4. No association classes.
3.1.5. There is the root class for every (component) model – the maximal class under composition and aggregation relation $(\hookleftarrow \cup \perp \cup \hookleftarrow w \cup \perp w)^*$, which is active.
3.1.6. If an association relation is n-ary then only one class is the root-end, having navigable and visible end-points (all other classes). This root is navigable and visible in any of its end-points iff it is navigable and visible in all its end-points.
3.1.7. The composite association defines a DAG.
3.1.8. If ac_id.agr = *composite* then for all ac_id.cj ∈ ac_id.end_points :
- ac_id.cj.*mult* ∈ {n, *}, n>0.
- ac_id.cj.*navigability* = *true*
- ac_id.cj.*changeability* = *frozen* if ac_id.cj.*mult* = n OR
  ac_id.cj.*changeability* = *add_only* if ac_id.cj.*mult* = *
- ac_id.root.*mult* = 1 and ac_id.root.*changeability* = *frozen*.

3.1.9. If ac_id.agr = *aggregate* then for all ac_id.cj ∈ ac_id.end_points :
- ac_id.cj.*mult* ∈ {n, *, [m,n]}, m, n>0.
- ac_id.cj.*navigability* = *true*
- ac_id.root.*mult* = 1 and ac_id.root.*changeability* = *frozen*.

3.1.10. If ac_id.agr = *neighbour* then:
- for all ac_id.cj ∈ ac_id.end_points : ac_id.cj.*mult* ∈ {n, *, [m,n]}, m, n>0.
- ac_id.root.*mult* = ∈ {n, *, [m,n]}, m, n>0.

1.11 No sharing of weak components between several weak composites in run-time.

3.2. Operations, Events and Attributes

3.2.1. No naming conflicts of operations, attributes, classes and associations names – e.g., in multiple inheritance.
3.2.2. Currently we support only two types of events: signal and call events. Signal events have public visibility.
3.2.3. Primitive operations do not call triggered operations.
3.2.4. A dependency graph of operation calls is tree-like (without recursions).
3.2.5. Triggered operations are guarded or sequential.
3.2.6. Primitive operations are sequential or free of side effects.
3.2.7. If a signal s1 is generalization of signal s2, then the list of parameters of s1 is a subset of that of s2.
3.2.8. No priorities on signals, all signals are processed in FIFO-order.
3.2.9. For all $c \in C \setminus A$ operations *create_c*, *destroy_c* $\in c' \Leftrightarrow (c \hookleftarrow c' \vee c \perp c' \vee c \hookleftarrow w\, c' \vee c \perp w\, c')$.

3.3. Action Language

3.3.1. No variable declaration within operation bodies (all declarations should be specified at the level of class definition, i.e. as attributes with the desired visibility).
3.3.2. Restricted set of primitive actions and constructs (only those described in Section 1.1)
3.3.3. For all navigation expression a0*.a1*…..an* (n ≥ 0):
- all references a0*,…, an-1* are association role names (can be default names)
- these references and operation (or attribute, resp.) an* are visible in the current class.

3.3.4. For all n ≥ 0 and assignments a0*.a1*…..an* := value we require that an* is a basic or navigation attribute. If an* is a reference then the corresponding association end has attribute an*.changeability ∈ {changeable, add_only}.

3.4. Statecharts

3.4.1. Every statechart must have a distinguished top state which is of mode OR.
3.4.2. Pseudostates *Join, Fork, Junction* and *Choice* are not considered
3.4.3. *do*-actions in states are not considered.
3.4.4. The priorities of transitions rise from outmost to innermost source state, meaning that a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.
3.4.5. If a class inherits from several classes, then only one of the generalized classes has statechart or generalized classes have equal statecharts. If a new statechart is specified in a specialized class, then it completely overwrites any statechart from its generalized class, i.e. the delegation of signal and call events (to the definition of their reception in another class) is not supported.

References

[1] Object Management Group *Unified Modelling Language Specification*, v 1.4, September, 2001.
[2] Object Management Group. *UML 1.4 with Action Semantics, Final Adopted Specification, ptc/02-01-09*, January, 2002.
[3] Manna, Z., Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
[4] 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.pdf
[5] J-M. Bruel, R.B.France, Transforming UML models to formal specifications, In: *UML'98 - Beyond the notation, 1st Intenational Workshop*, Mulhouse, France, LNCS 1618, Springer, 1998.
[6] E. Boerger, A. Cavarra, and E. Riccobene, An ASM Semantics for UML Activity Diagrams. In T. Rust (editor) *Proc. AMAST 2000*, LNCS 1912, Springer-Verlag, 2000, p. 361-366. http://citeseer.nj.nec.com/288568.html
[7] A.S.Evans, R.B.France, K.C.Lano, B.Rumpe, The UML as a formal modelling notation. In: *UML'98 - Beyond the notation, 1st Intenational Workshop*, Mulhouse, France, LNCS 1618, Springer, 1998.
[8] A.S.Evans and S.Kent, Meta-modelling semantics of UML: the pUML approach. In: B.Rumpe and R.B.France (editors), *2nd International Conference on the Unified Modeling Language*, Colorado, LNCS 1723, 1999.
[9] I. Ober, *Harmonizing Design Languages with Object-oriented Extensions and an Executable Semantics*, PhD Thesis, Institut National Polytechnique de Toulouse, France, April 2001.
[10] S. Gerard, F. Terrier, Y. Tanguy, Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML, OOIS Workshops 2002, p.260-269.
[11] http://www.ilogix.com/products/rhapsody/
[12] http://www.rational.com/products/rose/index.jsp
[13] http://www.telelogic.com/products/tau/

Index

N

O

P

R

S

T

U

V