T U M

I N S T I T U T F Ü R I N F O R M A T I K

Semantics of UML

Towards a System Model for UML

Part 2: The Control Model

Version 1.0

Manfred Broy, María Victoria Cengarle, Bernhard Rumpe

with special thanks to

Michelle Crane, Jürgen Dingel, Bran Selic

TUM-I0710

Februar 07

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-02-I0710-20/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Semantics of UML

Towards a System Model for UML

Part 2: The Control Model

Version 1.0

Manfred Broy[1]
María Victoria Cengarle[1]
Bernhard Rumpe[2]

[1] Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

[2] Institut für Software Systems Engineering
Technische Universität Braunschweig

with special thanks to
Michelle Crane
Jürgen Dingel
Bran Selic

Table of Contents

1 Introduction

The system model is designed to constitute the core and foundation of a formal definition of the UML semantics. Semantics in our terms is the "meaning" of a UML model – regardless whether this is a structural or a behavioural model [HR04]. This is the second part of the underlying system model, namely the "control part". The first part of the system model, namely the "static" part, defines how data stores are built and structured. This static part, together with a rationale and motivation of this effort, as well as the roadmap to define the complete necessary system model, is given in [BCR06]. We assume the reader is familiar with the definitions from [BCR06], as we refer to these definitions without reintroducing them here.

To shortly repeat the main concepts from Part 1 [BCR06], a data store, roughly speaking, consists of a set of objects, each equipped with an object identifier. An identifier among other things acts as reference to the locations (the attributes) of an object. The store furthermore contains a mapping from locations to values contained in the store at a certain point of time. Object identifiers and locations are ordinary values and thus can be stored and passed around. Further values are e. g., integer numbers or Booleans. Additional concepts like associations and links are integrated into objects. Various variation points allow, for instance, locating links in the objects on both sides of the associations, or additional objects of another class that also can have identity of their own.

This report focuses on the so-called control part of the system model. The control part defines the constituents of the structure used to store control information. Recall that the system model defines a universe of state machines intended to give semantics to any UML specification. Thus the control store contains information needed to determine the state transitions. In particular, we are interested in expressing within the system model:

- how control flows (as part of method calls) through active and passive objects,
- what it means for an object to be active or passive,
- how messages are passed, delayed and handled,
- how events are handled,
- how threads work in a distributed setting,
- how synchronisation of all these concepts takes place, and
- how actions and their scheduling are defined.

Besides the rather general rules that specify how control and data structure fit together, the present report also describes possible instantiations, e. g., for single-threaded or completely distributed and asynchronously communicating systems.

One result of this report is an explicit and flexible mechanism to describe control structures of various kinds resembling quite a number of implementation languages.

This variability is enforced by the UML and leads to a rather complex formalisation of control. In fact, UML does not allow us to abstract away from control primitives. In the systems we describe with UML, we do not only have various types of control and interaction, but also very often their combinations within a single system.

2 Control Part of the System Model

In addition to its data store (as introduced in [BCR06]), a state machine of the system model has a control store. The control store contains information about the behaviour of the intended system and is used by the state machine in order to decide which transition to perform next. A control store consists of:

- a stack of method/operation calls, each with its arguments and local variables,
- the progress of the running program (e. g., a program counter), and
- possibly information about one or more threads.

In any setting, be it distributed or not, any state machine of the system model also has to cope with receiving and sending of messages that trigger activities in objects. More general events, such as “message arrived” or “timeout”, must be handled by any object. As a first step, these events are put into an event store, which consists of an event buffer for each object where handling of events is managed.

Unfortunately, the rather abstraction-less definition of stacks, events, and threads is not very elegant. However, this lack of elegance accurately covers the lack of elegance in distributed object-oriented systems, where method calls, events and threads of activity are orthogonal concepts that can be mixed in various ways. On the one hand, these many concepts provide the system developer with great flexibility. On the other hand, they make it difficult to understand the behaviour of the resulting systems. In addition, these many concepts make it very awkward to describe a system model that covers all of them, because any combination (useful or useless) needs to be covered. The resulting complexity becomes apparent in modelling the control part of the system model.

2.1 State

A state of a state machine in the system comprises a data store, a control store, and a store for events.

Definition of state

- USTATE denotes the universe of states defined by

  $USTATE = DataStore \times ControlStore \times EventStore$

2.2 Operations

Objects are assessed through their methods/operations. Here, the term *operation* refers only to the signature, whereas the term *method* refers (also) to the implementation (or body). UML operations can be called and they may provide a

return value as given by the corresponding implementation. In the system model, each operation has a name, a signature (which includes a return value that may be of type void) and some additional information about its implementation.

A signature consists of a (possibly empty) list of types for parameters and a type for the return value.[1] Note that parameter names are not present in the signature; parameter names are only part of the implementation. For each operation, its signature and its implementation, as well as the class it belongs to, are uniquely specified. This approach does not explicitly specify overloading, signature and implementation inheritance, overriding and dynamic binding, but allows us to specialize in a flexible way to various actually used mechanisms of method binding. This even includes binding mechanisms such as that in Modula-3. These concepts, thus, are to be decided and defined by the time the mapping from UML to the system model is devised.

Definition of operations

- UOPN denotes the universe of operations
- UOMNAME denotes the universe of operation (i. e., method) names
- nameOf: UOPN → UOMNAME returns the name of the operation
- classOf: UOPN → UCLASS returns the class an operation belongs to
- parTypes: UOPN → List(UTYPE)
  returns the list of parameters of the operation
- resType: UOPN → UTYPE returns the result type of the operation

In order to complete the definition of signatures, we clarify how these signatures fit together with the subclassing mechanism. Subclassing (c sub d, which means c is subclass of d) defines a constraint on signatures and, in many languages, also on externally promised behaviour of its related classes c and d. In UML, interestingly, subclassing does not impose clear constraints on the implementation, as the implementation may be redefined according to some "compatibility". The notion of compatibility, however, is a semantic variation point that can be clarified in the system model, by adding additional constraints for redefined method behaviour. Subclassing in general allows a renaming of parameters in the implementation, as those are not part of the signature. The signature (in the form of a list of types), however, is either constant or in a generalisation/specialisation-manner the types of parameters can be generalised, and the type of the return value can be specialised. This is the well-known contra-variant way [Mey97] that ensures type safety in a language. It is unclear whether in UML operation parameters can actually be redefined, as e. g. Java allows, so we just have it in the system model, although UML may not need it.

[1] If there is more than one return value, in the system model all these return values are packed into one record. This explains the use of singular when speaking of the type for the return value. Similarly, in/out parameters, i.e., parameters that are both argument and return value of the operation, can be regarded as syntactic sugar; see below.

Definition of type safety on operations

- For each operation signature op1 $\in$ UOPN and class c $\in$ UCLASS with (classOf(op1) sub c), there is a signature op2 $\in$ UOPN with

  $\subset$ classOf(op2) = c $\wedge$
  nameOf(op1) = nameOf(op2) $\wedge$
  len(parTypes(op2)) = len(parTypes(op1)) $\wedge$
  $\forall 1 \leq i \leq len(parTypes(op2))$:
  CAR(parTypes(op1)[i]) $\supseteq$ CAR(parTypes(op2)[i]) $\wedge$
  CAR(resType(op1)) $\subseteq$ CAR(resType(op2)).

This approach allows overloading of homonymous methods with different parameter sets as well as extensible signatures like those of Java 1.5, even though it does not contain explicit infrastructure to describe this easily.

However, this definition must not necessarily hold in all approaches. In particular, object-oriented languages exhibiting errors "Message not understood", to which a program can react, do not enforce this type safety requirement.

UML furthermore provides "out" and "in/out" parameters. The system model allows to encode these parameters by passing locations of the variables where the "out"-values are to be stored.[2]

In UML, there is also the notion of "object behaviour", which, strictly speaking, is not a method. However, for simplification of the system model, we encode object behaviour as a special kind of method associated with the object, whose parameters define the signature of the method.

2.3 Methods

Recall that the term operation only refers to the signature, whereas the term method refers (also) to the implementation. Methods, thus, have a signature and an internal implementation. The signature of a method consists of a list of pairs of parameter names and parameter types. Projected on the list of types, this list coincides with the parameter type list of the associated operation(s).

To provide all information necessary for a detailed understanding of method interactions, we need an abstract notion of a program counter, a binding between argument values and the corresponding formal parameter variables, and a store for local variables. Furthermore, a method is equipped with the class name, where it is defined. Note localsOf and parOf result in records that are part of the UTYPE universe.

[2] Some authors advise against the use of (in/)out parameters. The recommendation in the present context is to use a variation point where, if several "out"-values are to be assigned, each of these is assigned through method call or message passing. In this way, object encapsulation is kept.

Definition of methods

- UMETH denotes the universe of methods
- UPC denotes the universe of program counter values
- nameOf: UMETH → UOMNAME
- classOf: UMETH → UCLASS
  denotes the class the method (implementation) belongs to
- parNames: UMETH → List(UVAR)
  denotes a list of formal parameter variables
- parOf: UMETH → UTYPE
  denotes a record of parameter variables with their types
- resType: UMETH → UTYPE
  denotes the result type of the method
- localsOf: UMETH → UTYPE
  denotes a record with the local variables with their types
- pcOf: UMETH → $\wp$(UPC)
  denotes a set of possible program counters

Here we have fully decoupled the concept of method (implementation) and operation (signature) to describe them independently. However, there usually is a strong link between methods and operations: A method can only implement operations with compatible signatures. However, as implementations can be inherited, one method can implement a number of operations in subclasses. In this way, on the one hand the operation signature can be adapted (e. g, made more specific) without changing the implementations, and on the other the implementation can be redefined using a new method in a subclass.

Definition of relationship between method and operation

- opn: UMETH → $\wp$(UOPN)
  denotes the operations (of several classes) that are implemented by the method
- $\forall$ m $\in$ UMETH, op $\in$ opn(m):
  nameOf(m) = nameOf(op) $\wedge$
  classOf(m) sub classOf(op) $\wedge$
  CAR(resType(m)) $\subseteq$ CAR(resType(op)) $\wedge$
  (parOf(m) = Rec{a_1:T_1, ..., a_n:T_n} $\Rightarrow$
  parNames(m)=[a_1, …, a_n] $\wedge$
  $\forall 1 \le i \le n$: $\exists S_i$: CAR(S_i) $\subseteq$ CAR(T_i) $\wedge$ parTypes(op)=[S_1, …, S_n])

Note that the external signature of an operation is defined through the list of types (parTypes(op)), whereas the parameter list of the corresponding method implementation also includes variable names to refer to these parameters (parNames(m)). Knowing which names the parameters have (from parNames), we

can use a bidirectional mapping between these two structures called rec[parNames(m)] to map any list of parameter values into the corresponding record.

The need for the operations on UMETH, in particular for parOf and pcOf, will eventually become apparent when mapping the UML to the system model. It is not ruled out that their need is refuted; for now, the authors feel that they will be confirmed.

The UML specification documents sometimes say “behaviour executions” are kinds of objects. We do not require every “behaviour execution” to be an object on its own, but if for some reason that would be necessary, it is not a problem to encode it that way.

2.4 Single-Thread Computation

A stack is a well-known storing mechanism that bears the structure necessary to handle chained and even (mutual) recursive method calls. In order to describe nested operation calls and, in particular, object recursion[3], we cannot abstract from the control stack. Object recursion is a common principle in object orientation and provides much flexibility and expressiveness. Almost all design patterns [GHJV94] as well as callback-mechanisms of frameworks [FPR01] rely on this principle.

Actually, to resume a computation after a method call, the information where computation is to be continued must be available. Therefore, the system model provides an abstract notion of stack frames, including even (abstract) program counters.

To introduce the general, multi-threaded case in an understandable fashion, we start by introducing the simplified case with one thread only. A stack frame consists of the identifier of the object that the method being (or to be) executed belongs to, the method name, the program counter of the method, the identifier of the invoking object and two record values, for parameters and local values. The first record value associates to each formal parameter the corresponding actual parameter value, the second associates to each local variable its corresponding value. The universe of frames is a rather general definition and many additional conditions can be found to constrain the actual set of actual states in the system model.

[3] That is, a method calls another method of the same object. However, calling the same method of an other object is method, but not object recursion.

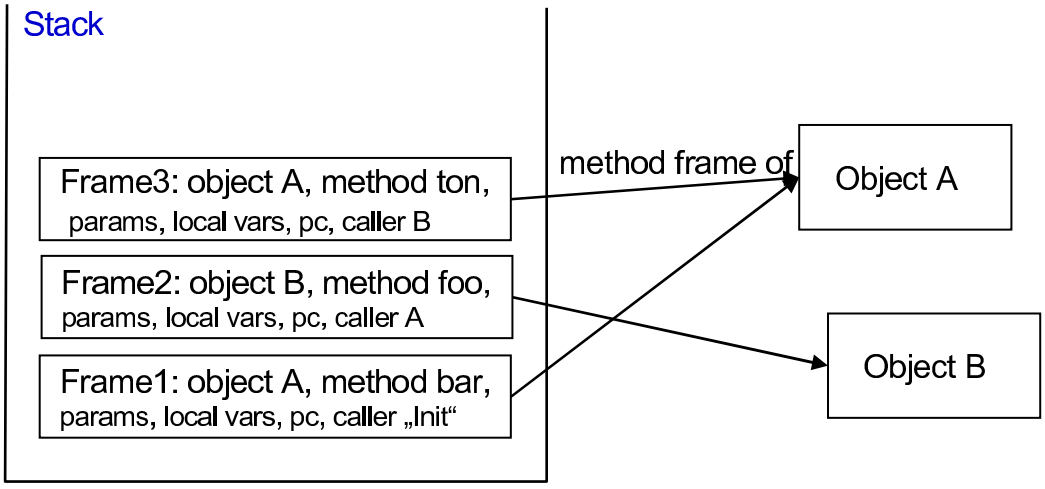
Definition

- UFRAME = UOID × UOMNAME × UVAL × UVAL × UPC × UOID
  is the universe of frames
- framesOf: UMETH → ℘(UFRAME)
  is the set of possible frames for a given method
- framesOf(m) = { (callee, nameOf(m), pars, locals, pc, caller) :
  ∃op∈opn(m): callee ∈ Oid (classOf(op)) ∧
  pars ∈ CAR(parOf(m)) ∧
  locals ∈ CAR(localsOf(m)) ∧
  pc ∈ pcOf(m) }

The method may include forks in its flow. Nevertheless, frames have only one program counter. When a fork takes place, a new thread is started. This thread has its own stack of frames, each of which contains again only one program counter. In this case, we are no more in the single-thread computation.

We illustrate the use of stacks of frames by means of an example. Let operations *ton* and *bar* belong to an object A, and operation *foo* belong to an object B. Suppose further that first operation *bar* on A is invoked, which then calls operation *foo* on B, which again calls operation *ton* on A. Let all these operations each be implemented by a homonymous method. Then the stack of frames looks as depicted below:

Example:



In this stack, both frames Frame1 and Frame3 have value callee pointing to object A. Thus some object recursion occurred on object A, which has two method calls open. Frame1 is the frame at the bottom and Frame3 is the top of the stack; thus Frame3 is the currently active frame.

In the case of a single threaded system, the only existing thread could then be defined as element of type

Thread = Stack(Frame).

In the following section, we extend the definitions introduced so far in order to model also distributed threads running in parallel. For this purpose, we develop two different, but compatible views.

2.5 Multiple-thread Computation, Centralised View

There are quite a number of approaches to combine object orientation and concurrency. Some approaches argue that each object is a unit of concurrency on its own. Others group passive objects into regions around single active objects, allowing operation calls only within a region and message passing only between regions. The programming languages that are commonly used today, however, have concurrency concepts that are completely orthogonal to objects. This means, various concurrent threads may independently and even simultaneously "enter" the very same object. In the following, we add a model of threads to our system model that handles this general case and allows us to specialise to all these approaches.

However, we do have a basic assumption that there is a notion of atomic action. These atomic actions are the basic units for concurrency; their exact definition is deferred until the UML actions are studied in detail. On top of atomic actions, we assume forms of concurrency control that are provided through appropriate concepts in UML (like "synchronised" in Java). Possible units of concurrency are, for example, a variable assignment or an operation invocation.

To model multiple threads, we introduce the (abstract) universe of possibly infinitely many threads. The control store maps a stack of frames to each thread.

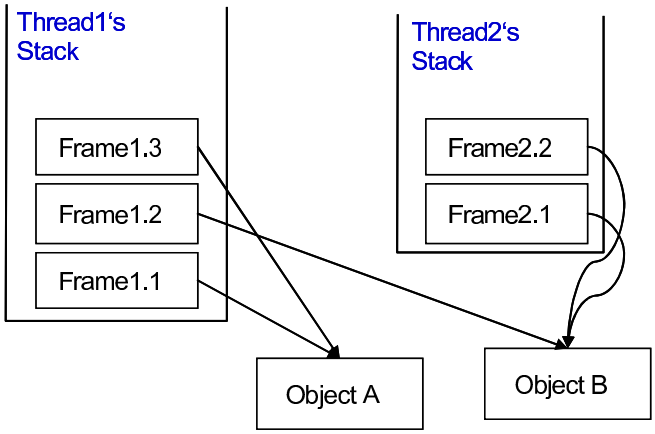
Definition (centralised version)

- UTHREAD is the universe of threads
- CentralControlStore : UTHREAD $\rightarrow$ Stack(UFRAME)

A thread may be called active if its associated stack is not empty. Active objects obtain a thread of their own when created (or initialised), passive objects do not own but can be visited by threads.

The figure below illustrates the situation where two threads are active, and both object recursion as well as concurrency occurs. Here "frame x.y" denotes that the frame is in thread x at position y, where the highest y-numbers denote the active frames:

Example



2.6 Multiple-thread Computation, Object-Centric View

The model of threads defined above is rather general, but so far does not cover how concurrent threads are executed within an object. To allow a more general mechanism for scheduling and definition of priorities, we rearrange the representation and therefore provide a different view on threads (i. e., without changing the described model). The key idea is to move from a thread-centric view to an object-centric view. As an important side effect, objects can then be described in a self-contained way, meaning that they control all incoming and outgoing messages, and in this way a compositional view on object-oriented systems is made possible.

Definition of threads and control store

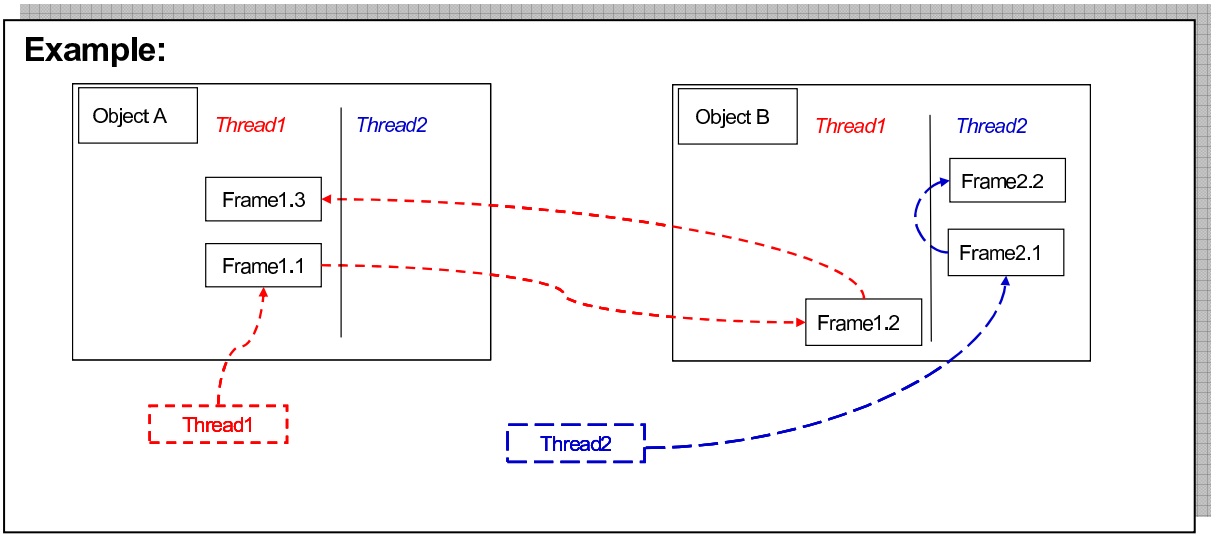
- ControlStore = UOID $\rightarrow$ UTHREAD $\rightarrow$ Stack(UFRAME)
- CentralControlStore ~ ControlStore if

  $\forall$ oid $\in$ UOID, $\forall$ t $\in$ UTHREAD:
  filter(CentralControlStore(t),oid) = ControlStore(oid)(t)

  where filter removes the frames of its first argument whose callee is not identical to its second argument

Note that .~. defines an isomorphism, because the caller object identifier is part of the frame of the called object and therefore the original stack can be reconstructed from its parts. So both representations of the control store provide exactly the same information.

The following figure shows the above example as represented by the object-centred control store:



According to the definitions below, we will see that an object is able to recognize that it is being called a second time within the same thread. This is important when, for instance, scheduling or blocking incoming messages from other threads. The Java synchronisation model distinguishes recursive calls from the same thread and calls from other threads.

3 Messages and Events in the System Model

3.1 Messages, Events and the EventStore

UML provides a rather general notion of events. Examples for events are

- sending and reception of messages, which resembles method calls with parameters or return values,
- a timeout,
- a simple signal, or
- a spontaneous state change.

In general, we assume events may be handled by an operation being executed or continued (in case of a return event) or may be ignored. Events need not be handled in the order they appear, but a sophisticated management (scheduling) is allowed.

Note that the relationship between messages and events in UML is circular: On the one hand, events are instantaneous state changes that can be handled by creating a message that is then handled. On the other hand, UML also states that the arrival of a message at an object is an event. To get out of this cycle, we have decided that messages and events are separate entities on their own. Events will be handled in the event buffer. Sending and receiving messages are two specific kinds of events.

This leads to a uniform handling of events and messages and the more general concept of “Events”. In general, handling of event occurrences may be delayed, ignored or stored until it is possible to handle it. To capture this rather general notion

of event, we introduce a universe of events that can occur in systems described with UML, without structuring it further at this point. Later we introduce several kinds of messages, like method call and return, as special forms of events, but leave open which other kinds of messages exist.

Based on the universe of events, we can define the EventStore, where events that have occurred are buffered for processing. A buffer is a rather general structure to store and handle messages, deal with priorities etc.

Definition of the EventStore and object event signature

- UEVENT is the universe of events
- events: UOID $\rightarrow \wp$(UEVENT) are the events that may occur in an object
- EventStore = UOID $\rightarrow$ Buffer(UEVENT) where
  - $\subset$ EventStore(oid) $\in$ Buffer(events(oid))

We introduce two special kinds of events, namely the receive and send events (ReceiveEvent, SendEvent) for messages. The sending and receiving of a message at a given object o are different events; they moreover occur one after the other. Receive and send events know their associated messages which can be accessed by using the function msgOf.[4] Messages are a general mechanism to handle any kind of synchronous method call, as well as asynchronous message passing. Here we describe that any message has a unique sender and receiver. It does not distinguish between the various forms of messages. (Note that this does not allow us to model broadcasting or multicasting directly. Multicasting, for example, needs to be simulated through sending many messages.)

[4] Note that two equal message events, for instance the sending of message *m* from sender *s* to receiver *r*, can still be distinguished since they are equal but not the *same* event. Time stamps and other such mechanisms are disregarded in this document.

Definition of the object message signatures

- UMESSAGE is the universe of messages
- ReceiveEvent, SendEvent : UMESSAGE $\rightarrow$ UEVENT
- sender, receiver : UMESSAGE $\rightarrow$ UOID
- sender(m) = oid $\Rightarrow$ SendEvent(m) $\in$ events(oid)
- receiver(m) = oid $\Rightarrow$ ReceiveEvent(m) $\in$ events(oid)
- msgIn: UOID $\rightarrow \wp$(UMESSAGE) are the incoming messages for an object
- msgOut: UOID $\rightarrow \wp$(UMESSAGE) are the outgoing messages
  - msgIn(oid) = { m | receiver(m) = oid }
  - msgOut(oid) = { m | sender(m) = oid }
- msgOf: UEVENT $\rightarrow$ UMESSAGE is the associated message for a send or receive event
  - msgOf(ReceiveEvent(m)) = m
  - msgOf(SendEvent(m)) = m

We assume that a Buffer is a given, rather general, structure for storing and retrieving occurring events. It also allows the scheduler to rearrange the order of their processing with respect to priorities. In a simple version, a Buffer behaves like a FIFO-Queue without any priority mechanisms. In the case of single threaded programs without specific event structures, the Buffer will only contain incoming (message) events. These events trigger method calls and returns. Because of single threaded computing, the Buffer never contains more than one element and can thus safely be ignored.

3.2 Method Call and Return Messages

Here we introduce some very common kinds of events: namely messages that describe method call and return. Call messages carry the usual information, like called object, method name, parameter values, as well as the caller and the thread. All the parameters of a call that may arrive to invoke a message in an object are packed by the function callsOf into an appropriate message.

Definition of call messages

- callsOf: UOID × UOPN × UOID × UTHREAD → ℘(UMESSAGE), with

  callsOf(receiver,op,sender,thread) =
  { (receiver, nameOf(op), params, sender, thread) ∈
  UOID × UOMNAME × List(UVAL) × UOID × UTHREAD |
  receiver ∈ Oid (classOf(op)) ∧
  length(params) = length(parTypes(op)) ∧
  ∀1≤i≤length(params): params[i] ∈ CAR(parTypes(op)[i]) }

- Message receiver and sender are determined:
  - callsOf(receiver,op,sender,thread) ⊆ msgIn(receiver)
  - callsOf(receiver,op,sender,thread) ⊆ msgOut(sender)
- For ease of description we also use this function
  - callsOf(receiver) = $\bigcup_{s\in UOID,t\in UTHREAD,op\in UOP}$ callsOf(receiver,op,s,t)

Return messages carry the return value, the thread, the sender and receiver of the result value.

Definition of return messages

- returnsOf describes the set of returns that may arrive:
- returnsOf: UOID × UOP × UOID × UTHREAD → ℘(UMESSAGE), with
  returnsOf (receiver,op,returner,thread) =
  { (receiver, result, returner, thread) ∈
  UOID × UVAL × UOID × UTHREAD |
  returner ∈ Oid (classOf(op)) ∧
  result ∈ CAR(resType(op)) }
- Message receiver and sender are determined:
  - returnsOf(receiver,op,sender,thread) ⊆ msgIn(receiver)
  - returnsOf(receiver,op,sender,thread) ⊆ msgOut(sender)
- For ease of description we also use this function
  - returnsOf(receiver) =
    $\bigcup_{s\in UOID,t\in UTHREAD,op\in UOP}$ returnsOf(receiver,op,s,t)

Given the definition of message structures above, it is actually possible to unify the concepts of method calls and returns, on the one hand, and of messages, on the other, into one single concept. This allows the handling of composition and provides a clear interface definition for objects and object groups. Method calls and returns are then just special kinds of messages and can thus be treated together with other kinds of incoming messages.

3.3 Object State and Signature

The signature and the state space of an object can now be defined. The state space consists of the frames in the control store, the events in the event store and, of course, the attribute values of the data store. The above defined msgOut function describes the output set of messages and the events function describes the set of arriving events (where e. g. vals(_,_) is as defined in [BCR06]):

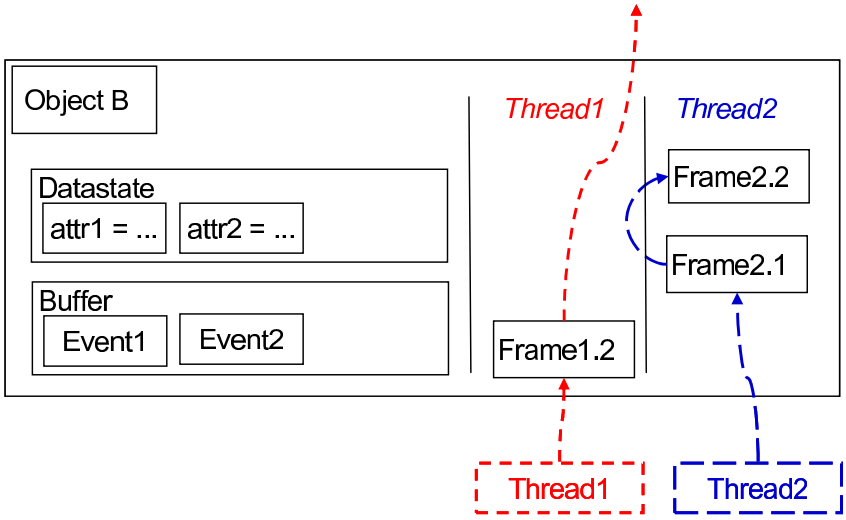
Definition the state space of an individual object

- The state of an object consists of its actual attribute values, the messages and the threads belonging to an object:
- state: $USTATE \times UOID \rightarrow (ULOC \rightarrow UVAL) \times (UTHREAD \rightarrow Stack(UFRAME) \times Buffer(UEVENT))$
  with $state((ds,cs,es),oid) = (vals(ds,oid), cs(oid), es(oid))$ for all $oid \in oids(ds)$
  The possible state space of an object is defined by
  $states: UOID \rightarrow \wp((ULOC \rightarrow UVAL) \times (UTHREAD \rightarrow Stack(UFRAME) \times Buffer(UEVENT)))$
  with $states(o) = \{ state(st,o) \mid st \in USTATE \}$

The following example demonstrates the structure of an object state:

Example:

- The storable state of an object in each snapshot of a system run consists of three parts:
  - The DataState consisting of a number of attributes in the store
  - The buffer of events that have arrived, but are unprocessed
  - A number of frames that store the current execution: There may be threads active (like Thread 2) or waiting for a return message event (like Thread 1 that has an outgoing call to another object).



UML provides the notion of "active" vs. "passive" objects. An active object owns a scheduler that is capable of handling asynchronously received messages. A passive object, on the contrary, only reacts to incoming method calls. An "active" object can thus be formalised as an object that has a thread associated which can be retrieved through a partial function thread: UOID → UTHREAD. The first frame on the Stack of the ControlStore(oid, thread(oid)), if thread(oid) is defined, is a permanently running method observing and controlling the input message queue of object oid.[5]

3.4 Asynchronous and Broadcast Messages

Not every message needs to carry a thread marker. There may be asynchronous messages that an object may accept. In this case, an object needs to be "active" in the sense that it already has an internal thread to process the message. However, it maybe that an object is not in itself active, but it belongs to a group of objects that has a common scheduling concept for the processing of messages that come from outside. This concept resembles the situation in classical language realisations, where one process contains many objects.

Messages can be multicasted and even broadcasted. Multicasted messages are put in the set of input messages msgIn(o) of many objects o, broadcasted messages in

[5] An object could be forced to be passive by e. g. demanding that the incoming messages msgIn(oid) be just calls and returns.

the set of input messages of all objects currently present in the system. Multicasted messages, before being replicated, do not carry the identifier of a receiver; they are, however, associated to a number of intended addressees. Broadcasted messages need not be associated to any receiver, since they are to be sent to every object.

It is interesting to notice that multicasted (or broadcasted) messages not necessarily are being waited for by the multiple recipients. By nature they are thus asynchronous and, as explained above, do not carry a thread identifier.

Depending on the kind of system, asynchronously sent messages can be distinguished from ordinary calls, or they can be treated as ordinary call messages with a special thread marker for the "asynchronous case". We therefore do not further formalise asynchronous messages here. However, it is necessary in the first case to add signatures for received and sent asynchronous messages.

A special kind of events models timeouts. Timeout events can be defined as ordinary messages sent by a special timer object to tell the receiving object that a certain amount of time has passed. Of course, timeout events should have a high priority in event buffers. As an alternative, the formalisation of state machines in the system model provides the use of time ticks. Thus an object can react to passing time, for instance by counting time ticks and reacting via creation of a timeout event that is buffered at first and handled (possibly immediately afterwards) like an ordinary message.

3.5 Computation and Scheduling

The handling of message arrival and storage in the queue is unspecified yet. This is part of the scheduling strategy that a system, a subsystem, a component or even a single object may have. This strategy determines the next step of an object. In fact, a centralised scheduling strategy (e. g., for all objects of a processor) may be modelled in the system model as easily as a decentralised version, where each object has its own scheduler. Also the scheduling may be defined for groups of objects for example belonging to the same group (processor, virtual processor, scheduling domain). Because of the huge variety, it seems difficult at this point to further constrain the possible forms of scheduling and to provide additional infrastructure for easier definition of these. Furthermore, we cannot directly define examples of scheduling strategies here, as these scheduling strategies rely on the notion of actions as well as the state machine model (which is introduced in the Part 3 of the series of documents for the definition of the System Model).

4 Conclusion

The control part of the system model defines two further parts of the store that object-oriented systems have. The control store is defined in a general form to represent concurrently working threads that contain frames of method invocations. Thread activity crossing objects is modelled by passing around thread markers.

A general notion of event, which deals with the arrival and sending of messages, as well as various other kinds of events, describes the general handling of events. It

opens the possibility of a great variety of events, as well as various forms of scheduling how these events will be processed, delayed or ignored.

This second part of the system model is now a sound basis for the third part, where state machines are introduced to describe behaviour. Furthermore, the state machines we are using are compositional and therefore can be used to model subsets of objects (e. g., “components”), as well as the overall system.

The complexity of this part of the system model has shown that the integration of objects, threads and concurrency is somewhat arbitrary. It is particularly complex to model the possible interactions between these, leading us to the assumption that it is particularly difficult to master these not so well integrated concepts.

5 References

[BCR06] M. Broy, M. Cengarle, B. Rumpe. Towards a System Model for UML. The Structural Data Model. Munich University of Technology, Technical Report. TUM-I0612. June 2006.

[FPR01] M. Fontoura, W. Pree, B. Rumpe. The UML Profile for Framework Architectures. Addison-Wesley. 2001.

[GHJV94] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns, Addison-Wesley, 1994.

[HR04] D. Harel, B. Rumpe. Meaningful Modeling: What’s the Semantics of „Semantics“? In: Computer, Volume 37, No. 10, pp 64-72. IEEE, October 2004.

[Mey97] Bertrand Meyer. Object-Oriented Software Construction. Second Edition. Prentice Hall. 1997.