

# FORMAL SEMANTICS FOR INTERACTING UML SUBSYSTEMS

Jan Jürjens\*

*Department of Informatics, TU Munich, Germany*

*Computing Laboratory, University of Oxford, GB*

<http://www.jurjens.de/jan> – [juerjens@in.tum.de](mailto:juerjens@in.tum.de)

**Abstract** So far most work on formal semantics for the Unified Modeling Language (UML) has concentrated on single diagrams. To provide a formal foundation for complete system specifications, one needs to put models for the different diagrams into context. We provide a formal semantics for UML subsystems that incorporates a formal semantics of the diagrams contained in a subsystem. It provides message-passing between objects or components specified in different diagrams, including a dispatching mechanism for events, and the handling of actions. It enables one to compose subsystems from sets of subsystems and allows them to interact by passing messages.

We give consistency conditions for the diagrams in a subsystem and define a notion of behavioural equivalence and two kinds of refinement for UML subsystems.

This is version 10/12/01 of a paper at FMOODS'02. Please refer to [www.jurjens.de/jan](http://www.jurjens.de/jan) for the current version and more information.

**Keywords:** UML, formal semantics, refinement, Abstract State Machines.

## 1. Introduction

The Unified Modeling Language (UML) [RJB99] is the de-facto industry standard for specifying object-oriented software systems (for an introduction cf. [SP00]). Even though rather precisely defined compared with other modelling languages, its semantics is given only in prose form [UML01], leaving room for ambiguities (a problem especially when providing tool support or trying to establish behavioural properties of UML specifications). Thus we need a mathematically precise semantics for UML.

There has been a substantial amount of work towards providing a formal semantics for UML diagrams (including [EFLR99; BCR00; Ste01]).

\*Supported by the Studienstiftung des deutschen Volkes when carrying out this research.

However, most work only provides models for single UML diagrams in isolation. When trying to give a precise mathematical meaning to whole UML specifications, one needs to be able to combine the formal models for the different kinds of diagrams.

**Joint formal execution.** In this paper we describe some results on how to formally model UML diagrams in context. We provide a formal semantics for UML subsystems that incorporates the formal semantics of the diagrams contained in the subsystem (here we consider statechart and activity diagrams; a treatment of the remaining diagrams is given in [Jür02]). Specifically, our semantics

- models actions and internal activities explicitly (rather than treating them as atomic given events), in particular the operations and their parameters employed in them,
- provides message-passing between objects or components specified in different diagrams, including a dispatching mechanism for events and the handling of actions, and thus
- allows whole specification documents to be based on a formal foundation.

In particular, we show how to *compose* subsystems by including them into other subsystems.

Thus it prepares the ground for further work that may

- provide tool-support based on this precise semantics, in particular allowing complete specifications to be simulated, and
- provide the possibility of complete executable UML specifications.

**Statechart, activity diagrams, subsystems.** The semantics for statecharts presented here is based on [BCR00], which however had to be extended significantly to incorporate the features mentioned above. Our formal semantics for subsystems and their interactions seems to be the first given in the published literature.

**Consistency.** Furthermore we give some conditions for *consistency*, relating different diagram kinds in a UML system specification, which is an issue that seems to attract increasing attention.

**Equivalence.** We define a notion of behavioural equivalence between UML subsystems. This can be used e.g. to verify consistency of two subsystem specifications that are supposed to describe the same behaviour,

e.g. one of which uses statecharts to specify object behaviour, and the other sequence diagrams (the treatment of sequence diagrams can be found in [Jür02]).

**Refinement.** In UML, *refinement* denotes a certain kind of dependency relation between model elements [UML01, 2-18]. There is no constraint on the semantic relationship between the model elements (also in the heuristics for state machine refinement at [UML01, 2-177]).

When trying to establish system properties, behavioural conformance of refinement can lead to great savings (properties may be easier to establish at a more abstract level; preservation by refinement means that this is in fact sufficient). We define a kind of refinement (inspired by [BS01]) that provides full behavioural conformance.

**UML for security.** One of the motivations for this work is to provide a foundation for using UML for developing security-critical systems [Jür01b], where a mathematically precise modeling is indispensable. The semantics presented here has been extended to include possible adversary behaviour to reason about security requirements [Jür02]. In particular, the preservation results in [Jür01a] extend to the refinement presented here.

For readability, we consider simplified UML diagrams. Our approach works just as well without the simplifications. More details can be found in [Jür02].

**Outline.** In Section 2 we give basic definitions of Abstract State Machines needed for our semantics. We provide a sketch of the semantics of statechart diagrams and activity diagrams in Section 3, in order to be able to show how these fit together in the context of UML subsystems, the semantics of which is presented in Section 3.3. We define two kinds of refinement for UML subsystems in Section 4 and give an example in Section 5. We end with pointers to related work, a conclusion and indication of future work.

## 2. Abstract State Machines

ASMs [Gur95] provide a rather flexible framework for formal modelling. They have been used e.g. to give a formal semantics for a non-trivial part of Java [SSB01].

We collect some central concepts. A *vocabulary* is a finite collection of function names, each of a fixed arity. We assume a set of *variables*. A *state*  $A$  of vocabulary  $\mathbf{Voc}(A)$  is a non-empty set  $X$  containing distinct elements *true*, *false*, and *undef* together with interpretations of

the function names in  $\mathbf{Voc}(A)$  on  $X$ . Relations and sets are viewed as functions taking values in  $\{true, false\}$ . An ASM is executed by *updating* its state (e.g. changing the interpretations of the names in the vocabulary) iteratively by applying *update rules* some of which are given in the following.

**Update** The *update rule*  $f(\bar{s}) := t$  of the ASM  $A$  updates  $f$  at the tuple  $\bar{s}$  to map to the element  $t$ .

**Conditional** For a Boolean term  $g$  rules  $R, S$ , the rule **if**  $g$  **then**  $R$  **else**  $S$  executes  $R$  if  $g$  holds, otherwise  $S$ .

**Blocks** The rule **do – in – parallel**  $R_1, \dots, R_k$  **enddo** executes the rules  $R_1, \dots, R_k$  simultaneously if they are mutually consistent: for any two update rules  $f(\bar{s}) := t$  and  $f(\bar{s}) := t'$ , we have  $t = t'$ . Otherwise, execution stops.

**Sequential Composition** For rules  $R, S$ , the rule **seq**  $R, S$  **endseq** executes  $R$  and  $S$  sequentially.

**Do-forall** For a variable  $v$ , a Boolean term  $g(v)$ , and a rule  $R(v)$ , the rule **forall**  $v$  **with**  $g(v)$  **do**  $R(v)$  executes  $R(a)$  for all  $a$  such that  $g(a)$  holds, if they are mutually consistent. Otherwise, execution stops.

**Loop through list** For a variable  $v$ , a list  $X$ , and a rule  $R(x)$ , the rule **loop**  $v$  **through list**  $X$   $R(v)$  executes  $R(x)$  iteratively for all  $x \in X$ .

An *abstract state machine* consists of a set of states and an update rule. It is executed by iteratively firing the update rule.

For multi-sets, we write  $\{\!\!\{ \}$  instead of the usual brackets. For two multi-sets  $M$  and  $N$ ,  $M \uplus N$  denotes their union and  $M \setminus N$  the subtraction of  $N$  from  $M$ . For a multi-set  $M$  and a set  $X$ , we write  $M \setminus X$  for the multi-set of those elements in  $M$  (preserving their cardinalities) that are also elements of  $X$ .

### 3. Formal Semantics for a fragment of UML

The set  $\mathbf{MsgNm}$  of *message names* consists of finite sequences of names  $n_1.n_2.\dots.n_k$  where  $n_1, \dots, n_{k-2}$  are names of UML subsystems (to be defined below),  $n_{k-1}$  is a name of an object, and  $n_k$  is the local name of the message. The idea is that a message  $n_1.n_2.\dots.n_k$  will be delivered as the message with name  $n_k$  to the object with name  $n_{k-1}$  which is part of the (iteratively nested) sequence of subsystems  $n_{k-2}, \dots, n_1$ . Messages in  $\mathbf{MsgNm}$  can be operations, signals, and return messages. For each operation  $op$  there is a corresponding return signal  $\text{return}(op)$ , assumed to be given explicitly. We write **Events** for

the set of *events* of the form  $op(exp_1, \dots, exp_n)$  with  $op \in \mathbf{MsgNm}$  and  $exp_1, \dots, exp_n \in \mathbf{Exp}$  (for a set  $\mathbf{Exp}$  of *expressions* which includes a set  $\mathbf{Var} \subseteq \mathbf{Exp}$  of variables). We define  $\mathbf{msgname}(m) \stackrel{\text{def}}{=} msg$  to be the name of the message  $m = msg(exp_1, \dots, exp_n)$ . In our model, every object  $O$  has associated multi-sets  $\mathbf{inQueue}(O)$  and  $\mathbf{outQueue}(O)$ . We model sending a message  $msg$  from an object  $S$  to an object  $R$  as follows:

- (1) The object  $S$  puts  $R.msg$  into its multi-set  $\mathbf{outQueue}(S)$ .
- (2) A scheduler distributes the messages from out-queues to the intended in-queues (removing the head); in particular,  $R.msg$  is removed from  $\mathbf{outQueue}(S)$  and  $msg$  added to  $\mathbf{inQueue}(R)$ .
- (3) The object  $R$  removes  $msg$  from its in-queue and processes its content.

In the case of operation calls, we also need to keep track of the sender to allow sending return signals. Modelling communication is thus very flexible; we can adjust the scheduler according to the underlying communication layer (and take account e.g. of security [Jür01b] or performance issues).

In the following sections, we define the abstract syntax of various UML diagrams using mathematical notation, and then give a precise semantics of the modelled system behaviour using ASMs. For space reasons, we can only present simplified versions of the diagrams; more details are in [Jür02].

### 3.1. Statechart Diagrams

**Abstract syntax.** A statechart diagram  $D = (\mathbf{Obj}_D, \mathbf{Cls}_D, \mathbf{Sts}_D, \mathbf{Ini}_D, \mathbf{Trs}_D)$  is given by an object name  $\mathbf{Obj}_D$ , a class name  $\mathbf{Cls}_D$ , a set of *states*  $\mathbf{Sts}_D$ , an *initial state*  $\mathbf{Ini}_D$ , and a set of transitions  $\mathbf{Trs}_D$ .  $\mathbf{Sts}_D$  is a set of tuples  $S = (\mathbf{nm}(S), \mathbf{ety}(S), \mathbf{init}(S), \mathbf{sta}(S), \mathbf{int}(S), \mathbf{ext}(S))$  where

- $\mathbf{nm}(S)$  is a string of characters called the *name* of the state,
- $\mathbf{ety}(S) \in \mathbf{Action}$  is called the *entry action*,
- $\mathbf{init}(S) \in \mathbf{Sts}_D \cup \{\mathit{undef}\}$  is the *initial substate* of  $S$ ,
- $\mathbf{sta}(S) \subseteq \mathbf{Sts}_D$  is the set of substates of  $S$ ,
- $\mathbf{int}(S)$  is called the *internal activity* (or *do-activity*), and
- $\mathbf{ext}(S) \in \mathbf{Action}$  is the exit action.

Here we write  $\mathbf{Action}$  for the set of actions of the following forms:

**Call/send action:**  $\mathbf{call}(msg(a_1, \dots, a_n))$  resp.  $\mathbf{send}(msg(a_1, \dots, a_n))$  for  $msg \in \mathbf{MsgNm}$  and  $a_i \in \mathbf{Exp}$ .

**Assignment:**  $\mathbf{att} := exp$  for an attribute  $\mathbf{att}$  and  $exp \in \mathbf{Exp}$ .

We assume that for every internal activity  $actv$  there is an ASM rule **ActvRule**( $actv$ ).

$\text{Tr}_D$  is a set of tuples  $t = (\text{src}(t), \text{evt}(t), \text{grd}(t), \text{act}(t), \text{tgt}(t))$  where

- $\text{src}(t) \in \text{St}_D$  is the *source state* of  $t$ ,
- $\text{evt}(t) \in \mathbf{Events}$  is the triggering event of  $t$ ,
- $\text{grd}(t)$  is a Boolean expression called the *guard* of  $t$ ,
- $\text{act}(t) \in \mathbf{Action}$  is an action (to be performed when firing  $t$ ),
- $\text{tgt}(t) \in \text{St}_D$  is the *target state* of  $t$ .

$\text{evt}(t)$  must be of the form  $op(\text{exp}_1, \dots, \text{exp}_n)$  with mutually distinct  $\text{exp}_1, \dots, \text{exp}_n \in \mathbf{Var}$ . We assume a special event  $\text{CompEv} \in \mathbf{Events}$  (without parameters) as in [BCR00]. If  $\text{intern}(t) = \text{true}$  then  $t$  is called an *internal* transition, otherwise it is called *external*.

**Behavioural semantics.** We give a formal semantics of statechart diagrams using ASMs. It is based on [BCR00], which however had to be extended significantly to incorporate explicit modelling of actions and internal activities and message-passing between objects or components in different diagrams. We fix a statechart diagram  $D$  modelling an object  $O \stackrel{\text{def}}{=} \text{Obj}_D$  and give its behavioural semantics as an ASM  $\llbracket D \rrbracket^{SC}$ .

The vocabulary of  $\llbracket D \rrbracket^{SC}$  consists of the following names:

- the set name  $\text{currState}$  (storing the set of currently active states),
- the multi-set names  $\text{inQueue}(O)$ ,  $\text{outQueue}(O)$  (the input resp. output queue),
- the function name  $\text{trigsusy}()$  mapping each operation name to the object or subsystem that last sent it (to allow sending back return values),
- the function name  $\text{finished}$  (mapping states to Boolean values, indicating whether a given state is finished), and
- all variables names in  $\text{evt}(t)$  for all  $t \in \text{Tr}_D$ .

The Boolean  $\text{finished}_S$  is set to *true* at the end of an internal activity **ActvRule**( $\text{int}(S)$ ).

The ASM  $\llbracket D \rrbracket^{SC}$  has two rules, **Initialize**( $D$ ) and **Main**( $D$ ), given below (defined using other rules omitted for space restrictions; they are adapted from [BCR00] and can be found in [Jür02]). The former rule initializes the variables of the ASM. The latter rule consists of selecting the event to be executed next (where priority is given to the completion event) and executing it, and then executing the rules for the internal activities in a random order.

```

Rule Initialize( $D$ )
do – in – parallel
  inQueue( $Obj_D$ ) :=  $\emptyset$   outQueue( $Obj_D$ ) :=  $\emptyset$ 
  currState := { $lni_D$ }  finished $_{lni_D}$  := false
enddo
Rule Main( $D$ )
seq if Completed  $\neq \emptyset$  then eventExecution(CompEv)
  else choose  $e$  with  $e \in$  inQueue( $O$ ) do
    seq inQueue( $O$ ) := inQueue( $O$ ) \  $\{e\}$ 
    if  $e = op_{sender}[args] \in$  Operation
      then seq  $e := op[args]$ 
        trigsusy( $e$ ) := sender endseq
    eventExecution( $e$ )
  endseq
  loop  $S$  through set currState
    seq finished $_S$  := false
      ActvRule(int( $S$ ))
    endseq
  endseq
endseq

```

Here eventExecution( $e$ ) is a macro whose execution models the execution of the event  $e$ , which involves firing its associated action as follows.

```

Rule ActionRule(call( $op[args]$ ))
  outQueue( $O$ ) := outQueue( $O$ )  $\uplus$   $\{op_O[args]\}$ 
Rule ActionRule(send( $e$ ))
  outQueue( $O$ ) := outQueue( $O$ )  $\uplus$   $\{e\}$ 
Rule ActionRule(send(return( $op$ )( $a$ )))
  outQueue( $O$ ) := outQueue( $O$ )  $\uplus$   $\{trigsusy(op).return(op)(a)\}$ 

```

### 3.2. Activity diagrams

An activity diagram is a special case of a state machine that is used to model processes involving one or more classes [UML01, B-2]. The internal activities in an activity diagram are given as an object or subsystem name  $S$ . Thus activity diagrams coordinate the execution of objects and subsystems. Note that these can be specified to proceed at different speeds, for example by including delay actions.

### 3.3. Subsystems and System Specifications

A subsystem  $\mathcal{S} = (\text{name}(\mathcal{S}), \text{Msgs}(\mathcal{S}), \text{Ints}(\mathcal{S}), \text{Ssd}(\mathcal{S}), \text{Ad}(\mathcal{S}), \text{Bd}(\mathcal{S}))$  is given by

- the name  $\text{name}(\mathcal{S})$  of the system,

- a set of accepted messages  $\text{Msgs}(\mathcal{S})$ ,
- a set of interfaces  $\text{Ints}(\mathcal{S})$ ,
- a static structure diagram  $\text{Ssd}(\mathcal{S})$  (defined in Section 3.3)
- an activity diagram  $\text{Ad}(\mathcal{S})$ , and
- a set  $\text{Bd}(\mathcal{S})$  of statechart diagrams (each for a different object). Each statechart specifies the behaviour of one object in the subsystem, so  $\text{Bd}(\mathcal{S})$  is empty if there are no classes (but only subsystems) in  $\text{Ssd}(\mathcal{S})$ .

Subsystems may contain further kinds of diagrams omitted here.

For simplicity, we do not explicitly model creation and deletion of objects, but only activation and inactivation (following [KW01, p. 15]). The run-to-completion step for each object is performed in parallel, subject to the flow of control specified by the activity diagram.

**Class Diagrams or Static structure Diagrams.** A *class model*  $C = (cname, att, mess, int)$  is given by

- a name  $cname$ ,
- a set of attribute names  $att$ ,
- a set of message names  $mess$ ,
- and a set  $int$  of interfaces of the form  $I = (iname, msg)$  where  $iname$  is the interface name and  $msg$  a set of message names.

A *dependency* is a tuple  $(dep, indep, int, stereo)$  consisting of

- class or subsystem names  $dep$  and  $indep$ ,
- an interface name  $int$ , and
- a stereotype  $stereo \in \{\ll call \gg, \ll send \gg\}$ .

Note that the existence of a dependency  $(dep, indep, int, stereo)$  with  $stereo \in \{\ll call \gg, \ll send \gg\}$  implies that the object  $dep$  knows of the object  $indep$  (otherwise  $dep$  could not send a signal to or call the object  $indep$ ).

A *static structure diagram*  $D = (\text{SuSys}(D), \text{Dep}(D))$  is given by a set  $\text{SuSys}(D)$  consisting of class models or subsystems, and a set  $\text{Dep}(D)$  of dependencies  $(dep, indep, int, stereo)$ . Other modelling elements (such as associations) can be added without complication. A static structure diagram is called a *class diagram* if it contains no subsystems.

**Consistency between UML diagrams.** A subsystem is called *consistent* if the following conditions are satisfied:

- The object and subsystem names appearing as activities in the activity diagram are part of the static structure diagram. The behaviour of the objects is defined by statecharts.

- For each call resp. send action in a statechart diagram, the static structure diagram  $C$  must have a dependency stereotyped  $\ll call \gg$  resp.  $\ll send \gg$  between the objects (or their interfaces) in question. For each assignment action  $att := exp$  in  $S$ ,  $att$  is contained in the set of attributes of  $Cls_S$  given in  $D$ .
- The operations offered by the subsystem must be offered by class models or subsystems in the static structure diagram.
- For statecharts  $S$  and  $T$  we have  $S = T$  or  $Obj_S \neq Obj_T$ .

**Behavioural semantics of Subsystems.** Suppose we are given a consistent subsystem  $\mathcal{S}$ . The behavioural interpretation of  $\mathcal{S}$  is given by the ASM  $\llbracket \mathcal{S} \rrbracket^{SuSy}$  with the rules **Initialize**( $\mathcal{S}$ ) and **Main**( $\mathcal{S}$ ) defined in the following.

We write  $Acts(\mathcal{S})$  for the set of object and subsystem names giving the activities in  $Ad(\mathcal{S})$ . If  $act \in Acts(\mathcal{S})$  is an object whose behaviour is given by a statechart diagram  $D \in Bd(\mathcal{S})$  (i. e.  $Obj_D = act$ ), we write  $\llbracket \mathcal{S} \rrbracket_{act}^{BD}$  for  $\llbracket D \rrbracket^{SC}$ , **BDInitialize** $_{\mathcal{S}}(act)$  for **Initialize**( $D$ ) and **BDMain** $_{\mathcal{S}}(act)$  for **Main**( $D$ ). If  $act \in Acts(\mathcal{S})$  is a subsystem in  $Ssd(\mathcal{S})$ , we write  $\llbracket \mathcal{S} \rrbracket_{act}^{BD}$  for  $\llbracket act \rrbracket^{SuSy}$ , **BDInitialize** $_{\mathcal{S}}(act)$  for **Initialize**( $act$ ) and **BDMain** $_{\mathcal{S}}(act)$  for **Main**( $act$ ).

The vocabulary of  $\llbracket \mathcal{S} \rrbracket^{SuSy}$  includes the following names:

- the names in the vocabularys of  $\llbracket S \rrbracket^{SuSy}$  for all  $S \in SuSys(Ssd(\mathcal{S}))$ ,  $\llbracket Ad(\mathcal{S}) \rrbracket^{AD}$  and  $\llbracket Bd(\mathcal{S}) \rrbracket_S^{BD}$  for all  $S \in Acts(\mathcal{S})$ ,
- the names  $inQueue(\mathcal{S})$  and  $outQueue(\mathcal{S})$  (in and out queue of  $\mathcal{S}$ ).

We define the two rules.

*Rule Initialize*( $\mathcal{S}$ )

```
do – in – parallel
  Initialize(Ad( $\mathcal{S}$ ))
  forall  $S$  with  $S \in Acts(\mathcal{S})$  do
    BDInitialize $_{\mathcal{S}}(S)$ 
  inQueue( $\mathcal{S}$ ) :=  $\emptyset$     outQueue( $\mathcal{S}$ ) :=  $\emptyset$ 
enddo
```

*Rule Main*( $\mathcal{S}$ )

```
seq
  forall  $S$  with  $S \in Acts(\mathcal{S})$  do
    inQueue( $S$ ) := inQueue( $S$ )  $\uplus$ 
      { tail( $e$ ) :  $e \in (inQueue(\mathcal{S}) \setminus Msgs(\mathcal{S})) \wedge head(e) = S$  }
    inQueue( $\mathcal{S}$ ) :=  $\emptyset$ 
    Main(Ad( $\mathcal{S}$ ))
  forall  $S$  with  $S \in Acts(\mathcal{S})$  do
```

$$\begin{aligned}
& \text{inQueue}(\mathcal{S}) := \text{inQueue}(\mathcal{S}) \uplus \\
& \quad \uplus_{T \in \text{Acts}(\mathcal{S})} \{ \mathbf{tail}(e) : e \in \text{outQueue}(T) \wedge \mathbf{head}(e) = \mathcal{S} \} \\
& \text{outQueue}(\mathcal{S}) := \text{outQueue}(\mathcal{S}) \uplus \\
& \quad \uplus_{T \in \text{Acts}(\mathcal{S})} \{ \mathbf{tail}(e) : e \in \text{outQueue}(T) \wedge \mathbf{head}(e) = \mathcal{S} \} \\
& \mathbf{forall} \mathcal{S} \mathbf{with} \mathcal{S} \in \text{Acts}(\mathcal{S}) \mathbf{do} \\
& \quad \text{outQueue}(\mathcal{S}) := \emptyset \\
& \mathbf{endseq}
\end{aligned}$$

**System specifications.** A UML *system* is given by a UML subsystem which is intended to model the complete system under consideration (rather than just a part).

Given a system  $\mathcal{S}$  and a sequence  $\vec{I}_1, \dots, \vec{I}_n$  of multi-sets, the timed behaviour  $\llbracket \mathcal{S} \rrbracket^t(\vec{I}_1, \dots, \vec{I}_n)$  is defined to be the set of possible contents of  $\text{outlist}(\mathcal{S})$  after the execution of any instantiation of the following ASM rule on the ASM  $\llbracket \mathcal{S} \rrbracket^{\text{SuSy}}$ .

*Rule*  $\mathbf{tSys}_A(\mathcal{S})$

$$\begin{aligned}
& \mathbf{seq} \quad \text{outlist}(\mathcal{S}) := \emptyset \\
& \quad \mathbf{Initialize}(\mathcal{S}) \\
& \quad \mathbf{loop} \ i \ \mathbf{through} \ \text{list} \ [1 \dots n] \\
& \quad \quad \mathbf{seq} \quad \text{inQueue}(\mathcal{S}) := \text{inQueue}(\mathcal{S}) \uplus \vec{I}_i \\
& \quad \quad \quad \mathbf{Main}(\mathcal{S}) \\
& \quad \quad \quad \text{outlist}(\mathcal{S}) := \text{outlist}(\mathcal{S}).\text{outQueue}(\mathcal{S}) \\
& \quad \quad \quad \text{outQueue}(\mathcal{S}) := \emptyset \\
& \quad \quad \mathbf{endseq} \\
& \mathbf{endseq}
\end{aligned}$$

The untimed behaviour of a system  $\mathcal{S}$  is a function  $\llbracket \mathcal{S} \rrbracket^u()$  from multi-sets of events (the inputs to the system during its execution) to sets of multi-sets of events (the possible outputs of the system during its (possibly non-terminating) execution). It is defined at the meta-level. Given a multi-set  $I$ , we define  $\llbracket \mathcal{S} \rrbracket^u(I)$  to be the fix-point of the content of  $\text{outQueue}(\mathcal{S})$  after firing the rule

$$\mathbf{seq} \quad \mathbf{Initialize}(\mathcal{S}) \quad \text{inQueue}(\mathcal{S}) := I \quad \mathbf{endseq}$$

and iterating the rule  $\mathbf{Main}(\mathcal{S})$ . The fix-point exists, because  $\mathcal{S}$  changes  $\text{outQueue}(\mathcal{S})$  only by *adding* elements: it is the directed union of the contents of  $\text{outQueue}(\mathcal{S})$  after each execution of  $\mathbf{Main}(\mathcal{S})$ .

## 4. Equivalence and Refinement

**Definition 1 (Property Refinement)** *Suppose we are given two subsystem specifications  $\mathcal{S}$  and  $\mathcal{S}'$  and a set  $\mathcal{M}$  of message names.  $\mathcal{S}'$  is a*

timed (resp. untimed)  $\mathcal{M}$ -refinement of  $\mathcal{S}$  if conditions I and IIa (resp. conditions I and IIb) are fulfilled:

**I**  $\text{Msgs}(\mathcal{S}) \cap \mathcal{M} \subseteq \text{Msgs}(\mathcal{S}') \cap \mathcal{M}$

**IIa** for all sequences  $I_1, \dots, I_n$  of multi-sets of events such that  $\text{msgname}(e) \in \mathcal{M}$  for each  $e \in \bigcup_{i=1, \dots, n} I_i$ , we have

$$\llbracket \mathcal{S}' \rrbracket^t(I_1, \dots, I_n) \curvearrowright \mathcal{M} \subseteq \llbracket \mathcal{S} \rrbracket^t(I_1, \dots, I_n) \curvearrowright \mathcal{M}$$

**IIb** for each multi-set  $I$  of events such that  $\text{msgname}(e) \in \mathcal{M}$  for each  $e \in I$ , we have  $\llbracket \mathcal{S}' \rrbracket^u(I) \curvearrowright \mathcal{M} \subseteq \llbracket \mathcal{S} \rrbracket^u(I) \curvearrowright \mathcal{M}$

where for a set  $S$  of multi-sets of events we define  $S \curvearrowright \mathcal{M} \stackrel{\text{def}}{=} \{M \setminus \{e \in \mathbf{Events} : \text{msgname}(e) \in \mathcal{M}\} : M \in S\}$  to be the set of multi-sets obtained by removing all events  $e$  with  $\text{msgname}(e) \notin \mathcal{M}$  (and extend the definition to sequences of multi-sets by applying it to the elements of the sequence). A timed (resp. untimed) **MsgNm**-refinement is simply called a (resp. untimed) refinement.

For a motivation of the refinements and comparison to other kinds, see [Jür02]. Both kinds of  $\mathcal{M}$ -refinement are reflexive and transitive for each  $\mathcal{M}$ . They preserve all  $\mathcal{M}$ -safety properties (i. e. sets of sequences of event multi-sets with names in  $\mathcal{M}$ ). We show that refinement is preserved by substitution (i. e. a precongruence wrt. composition by subsystem formation).

A *parameterized subsystem*  $\mathcal{S}(\mathcal{Y}_1, \dots, \mathcal{Y}_n)$  is a subsystem specification where  $n$  of the subsystems are replaced by subsystem variables  $\mathcal{Y}_i$ . For subsystems  $\mathcal{S}_1, \dots, \mathcal{S}_n$ ,  $\mathcal{S}(\mathcal{S}_1, \dots, \mathcal{S}_n)$  is the subsystem obtained by substituting  $\mathcal{S}_i$  for  $\mathcal{Y}_i$ , for each  $i$ , in  $\mathcal{S}$ .

**Theorem 1** *If  $\mathcal{S}'_i$  is a timed refinement of  $\mathcal{S}_i$  for each  $i = 1, \dots, n$  then for any parameterized subsystem  $\mathcal{S}(\mathcal{Y}_1, \dots, \mathcal{Y}_n)$ ,  $\mathcal{S}(\mathcal{S}'_1, \dots, \mathcal{S}'_n)$  is a timed refinement of  $\mathcal{S}(\mathcal{S}_1, \dots, \mathcal{S}_n)$ .*

The theorem does not hold for untimed refinements [Jür02].

**Definition 2 (Behavioural Equivalence)** *Two subsystem specifications  $\mathcal{S}$  and  $\mathcal{S}'$  are timed (resp. untimed) behaviourally equivalent if  $\text{Msgs}(\mathcal{S}) = \text{Msgs}(\mathcal{S}')$ ,  $\mathcal{S}$  is a timed (resp. untimed)  $\text{Msgs}(\mathcal{S})$ -refinement of  $\mathcal{S}'$ , and  $\mathcal{S}'$  is a timed (resp. untimed)  $\text{Msgs}(\mathcal{S})$ -refinement of  $\mathcal{S}$ .*

The above facts on timed refinement imply that timed behavioural equivalence is a congruence wrt. composition by subsystem formation.

Behavioural equivalence can be used e.g. to verify consistency of two subsystem specifications that are supposed to describe the same behaviour.

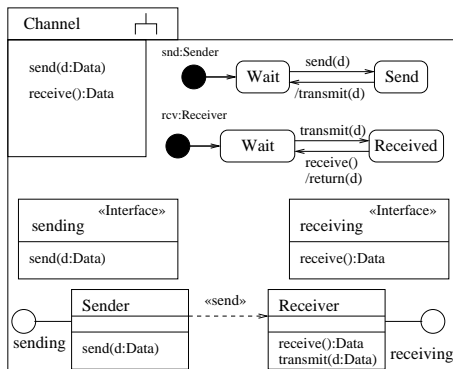


Figure 1. Sender and receiver

## 5. Example: Secure channel establishment

Figure 1 gives a high level system specification  $\mathcal{S}$  for communication from a sender object to a receiver object, including a class diagram with appropriate interfaces (we leave out the activity diagram which simply specifies that the two objects are executed in parallel).

Assume that an assessment of the physical layer of the system shows that security requirements are not provided. Thus we construct a refinement  $\mathcal{S}'$  in Figure 2. The class diagram needs to allow the receiver to send his certificate to the sender. The behaviour of the sender includes retrieving the public key and the certificate from the sender, checking the certificate, and encrypting the data. The receiver gives out the key and certificate first, and decrypts the received data. We assume that  $\text{Ver}_K(\text{Sign}_K(m), m) = \text{true}$  (i. e.  $\text{Ver}_K(m, O)$  is true if  $m$  is a valid signature of  $O$  with key  $K$ ).

**Theorem 2**  $\mathcal{S}'$  is an untimed  $\{\text{send}, \text{receive}, \text{return}_{\text{receive}}\}$ -refinement of  $\mathcal{S}$ .

The (straightforward) proof can be found in [Jür02]. Note that the theorem does not imply that the refined protocol is secure; this has to be established by other means (including an extension of the formal semantics with adversary scenarios), see [Jür02].

Note that  $\mathcal{S}'$  is not a *timed*  $\{\text{send}, \text{receive}, \text{return}_{\text{receive}}\}$ -refinement of  $\mathcal{S}$  (because of the delay caused by the key exchange).

## 6. Related Work

There has been a considerable amount of work towards a formal semantics for various parts of UML (see [Jür02] for an overview). [EFLR99]

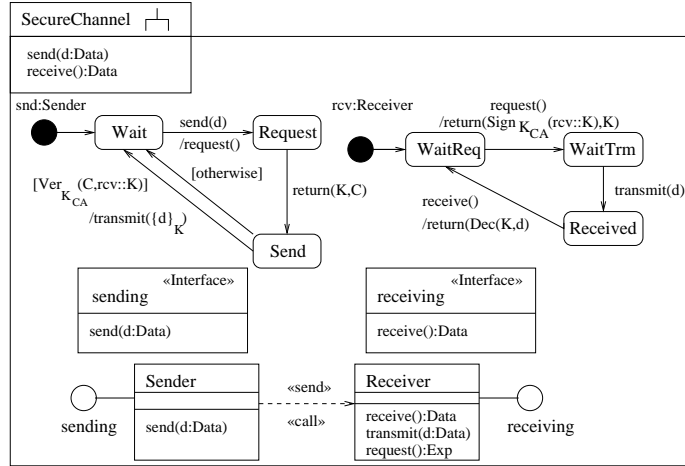


Figure 2. Secure channel

discusses some fundamental issues concerning a formal foundation for UML. [BCR00] uses ASMs to give a semantics for statecharts. [Ste01] gives a semantics for use case diagrams based on the process algebra CCS. [ÖP00] considers interacting UML subsystems, but without giving a formal semantics. A combined formal semantics for UML statecharts and class diagrams has been given in [RCA01]. [SKM01] gives a semantics for statecharts and shows exemplarily how to check whether a set of statecharts satisfies a collaboration. Refinements have been investigated in the object-oriented setting e.g. in [DS00; DB01] where the introduced structural refinement has a similar motivation as our interface refinement. In the context of subtyping, refinement has been considered e.g. in [PH97].

## 7. Conclusion and Future Work

Our work shows that giving a formal foundation for complete UML specifications, rather than just single diagrams, is possible. While in our presentation here we left out some of the more advanced features of some of the considered diagrams, incorporating them is not a problem (except for an increase in complexity). This paves the way for unambiguous modelling with UML, and in particular for tool-support that can simulate specifications in their entirety. This would seem to be very useful indeed, since one of the main challenges in constructing large-scale software systems is to ensure that different components act together as expected.

ASMs proved to be a quite adequate tool to handle the complexities in formally combining the different UML diagrams, due to their flexibility. In particular, the crucial extensions over the semantics for statecharts in [BCR00] could be done in a rather natural way.

The notions of subsystem equivalence and refinement seem to be valuable tools in the construction and analysis of UML specification (although here we could only present a toy example).

On the application side, the semantics presented here has been useful in the context of security-critical systems (cf. e.g. [Jür01b]).

In further work [Jür02] we have treated other kinds of UML diagrams and other kinds of refinements (including one inspired by action refinement [GR00]). We aim to give conditions for checking equivalence and refinement on the level of syntax or by combining an object-based logic (e.g. [DKR00]) with a logic for ASMs (e.g. [PH94]).

In terms of tool-support we intend to extend the ASM-based statechart simulator presented in [CCR01] to the semantics presented here.

**Acknowledgements.** Discussions with A. Cavarra about formal semantics for UML and constructive suggestions by the anonymous referees to improve the presentation of the paper are gratefully acknowledged.

## References

- [BCR00] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines. Theory and Applications*, volume 1912 of *LNCS*, pages 223–241. Springer-Verlag, 2000.
- [BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.
- [CCR01] R. Campo, A. Cavarra, and E. Riccobene. Simulating UML state machines. In E. Börger and U. Glässer, editors, *ASM'2001*, LNCS. Springer-Verlag, 2001. To be published.
- [DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and advanced applications*. Formal Approaches to Computing and Information Technology. Springer-Verlag, 2001.
- [DKR00] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a temporal logic for object-based systems. In Smith and Talcott [ST00], pages 305–326.
- [DS00] J. Derrick and G. Smith. Structural refinement in Object-Z / CSP. In W. Grieskamp, T. Stanten, and B. Stoddart, editors, *Integrated Formal Methods (IFM 2000)*, volume 1945 of *LNCS*, pages 194–213. Springer-Verlag, 2000.
- [EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. In J. Bézivin and P.-A. Muller, editors, *The Unified*

- Modeling Language - Workshop UML'98: Beyond the Notation*, LNCS, pages 297–307. Springer-Verlag, 1999.
- [GR00] R. Gorrieri and A. Rensink. Action refinement. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000.
- [Gur95] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. OUP, 1995.
- [Hu01] H. Hußmann, editor. *Fundamental Approaches to Software Engineering (FASE, 4th International Conference)*, volume 2029 of LNCS. Springer-Verlag, 2001.
- [Jür01a] J. Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (International Symposium)*, volume 2021 of LNCS, pages 135–152. Springer-Verlag, 2001.
- [Jür01b] J. Jürjens. Towards development of secure systems using UMLsec. In Hußmann [Hu01], pages 187–200.
- [Jür02] J. Jürjens. *Principles for Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, 2002. Submitted.
- [KW01] A. Kleppe and J. Warmer. Unification of Static and Dynamic Semantics of UML, 2001.
- [ÖP00] G. Övergaard and K. Palmkvist. Interacting Subsystems in UML. In A. Evans, S. Kent, and B. Selic, editors, *The Unified Modeling Language: Advancing the Standard (UML'2000)*, volume 1939 of LNCS. Springer-Verlag, 2000.
- [PH94] A. Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress '94*, pages 434–439. Elsevier, August 1994.
- [PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.
- [RCA01] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In Hußmann [Hu01].
- [RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [SKM01] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In S.D. Stoller and W. Visser, editors, *Workshop on Software Model Checking*, volume 55 of ENTCS. Elsevier, 2001.
- [SP00] P. Stevens and R. Pooley. *Using UML*. Addison-Wesley, 2000.
- [SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation*. Springer-Verlag, 2001.
- [ST00] S.F. Smith and C.L. Talcott, editors. *4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*. IFIP TC6/WG6.1, Kluwer Academic Publishers, 2000.
- [Ste01] P. Stevens. On use cases and their relationships in the Unified Modelling Language. In Hußmann [Hu01], pages 140–155.
- [UML01] UML Revision Task Force. OMG UML Specification v. 1.4. OMG Document ad/01-09-67. Available at <http://www.omg.org/uml>, 2001.