# A UML statecharts semantics with message-passing

Jan Jürjens*
Computing Laboratory, University of Oxford, GB and
Software & Systems Engineering, Dep. of Informatics, TU München, Germany
juerjens@in.tum.de

## ABSTRACT

We give a formal semantics for one of the main UML diagram types for dynamical system behavior: statechart diagrams. This is the first semantics which explicitly models message-passing between different diagrams. It therefore lays a first foundation for executable UML modeling, allowing whole systems of UML specifications (rather than single diagrams) to be simulated.

## Keywords

Unified Modeling Language (UML), formal semantics, statecharts, message-passing, executable specifications

## 1. INTRODUCTION

The Unified Modeling Language (UML) [13] is an industry standard for specifying object-oriented software systems. Compared to other modelling languages, UML is rather precisely defined.

To *reason* about system behavior in a precise way, however, we need a precise (mathematical) semantics for the behavioral model elements of UML. In the specification document [15], a semantics for dynamic model elements is given only in prose form, which leaves room for some ambiguities and gives problems when trying to provide tool support.

There has in fact been some work towards providing a formal semantics for behavioral UML diagrams (specifically, our work extends the semantics given in [1] using Abstract State Machines (ASMs)). However, so far it only provides models for single UML diagrams seen in isolation. When trying to give a precise mathematical meaning to whole UML specifications (which is neccessary to provide tool support), one needs to be able to combine the formal models for the different diagrams to give a coherent whole.

In this paper, we present work towards this goal. Specifically, we provide a formal semantics for UML statecharts which is the first to

- model actions and internal activities explicitly (rather than treating them as atomic given events), as well as the operations and their parameters employed in them,
- provide message-passing between different diagrams, including a dispatching mechanism for events and the handling of actions, and thus
- allow whole specification documents be based on a formal foundation, allowing to
- provide tool-support based on this precise semantics, in particular allowing complete specifications to be simulated, and
- ultimately provide the possibility of complete executable UML specifications.

Note that the fact that events can carry parameters is also one of the the major differences from Harel's statecharts [15, 2-180] (which may be why so far it has not been addressed).

While the work here is to be seen within the context of the greater approach which also deals with the other diagram types (such as sequence diagrams) and which combines them using UML packages (in particular subsystems), here we can only give the case of statecharts, for space limitations. However, since activity diagrams are just a special case of statechart diagrams [15], they are also covered by our semantics.

The more general motivation for this work is to widen the impact of formalism on the actual software development process, going beyond what traditional formal methods have achieved in the context of industrial practice. Also, it allows use of UML in contexts where a mathematically precise modeling is indispensable (such as security-critical systems [9, 10]).

In the following section, we give basic definitions of Abstract State Machines needed for our semantics We then provide the semantics and give examples. We end with pointers to related work, a conclusion and indication of future work.

## 2. ABSTRACT STATE MACHINES

We collect some central concepts. 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 $X$. An ASM is executed by *updating* its state iteratively by applying *update rules*:

$\mathbf{f}(\bar{s})\mathbf{:=t}$ updates $f$ at the tuple $\bar{s}$ to map to the element $t$.

**if** $g$ **then** $R$ **else** $S$ If $g$ holds, the rule $R$ is executed, otherwise $S$.

*http://www.jurjens.de/jan

**do − in − parallel** $R_1, \ldots, R_k$ **enddo** $R_i$ execute simultaneously, if for any two update rules $f(\bar{s}) := t$ and $f(\bar{s}) := t'$, we have $t = t'$; otherwise nothing changes.

**seq** $R$, $S$ **endseq** $R$ and $S$ are executed sequentially.

**loop** $v$ **through list** $X$ $R(v)$ iteratively execute $R(x)$ for all $x \in X$.

**case** $v$ **of** $x_1$ : **do** $R_1$ ... $x_n$ : **do** $R_n$ **else** $S$ execute by case distinction.

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

## 2.1 Interactive ASMs

We use ASMs to specify components of a system that interact by exchanging messages which are dispatched from resp. received in multi-set buffers (*output queues* resp. *input queues*).

The set **MsgNm** of *message names* consists of finite sequences of names $n_1.n_2.\ldots.n_k$ where $n_1, \ldots n_{k-2}$ are names of ASM systems (to be defined below), $n_{k-1}$ is a name of an interactive ASM, and $n_k$ is the local name of the message. The idea is that a message $n_1.n_2.\ldots.n_k$ will be delivered as the message with name $n_k$ to the ASM with name $n_{k-1}$ which is part of the (iteratively nested) sequence of ASM systems $n_{k-2}, \ldots, n_1$. We assume a set **Exp** of expressions as given. Given a set of message names $\mathcal{M} \subseteq$ **MsgNm**, we write **Events** for the set of terms of the form $msg(exp_1, \ldots, exp_n)$ where $msg \in$ **MsgNm** is an $n$-ary message name and $exp_1, \ldots, exp_n \in$ **Exp** are expressions. We define $\mathbf{Args}(m) \stackrel{\text{def}}{=} [exp_1, \ldots, exp_n]$ to be the list of its arguments of $m = msg(exp_1, \ldots, exp_n)$, and $\mathbf{msgname}(m) \stackrel{\text{def}}{=} msg$ to be the name of its message. For multi-sets, we write $\{\!\!\{\ \}\!\!\}$. For two multi-sets $M$ and $N$, $M \uplus N$ denotes their union and $M \setminus N$ the subtraction of $N$ from $M$.

**Definition 1** An *interactive ASM* $(A, in, out)$ is given by an ASM $A$ and two sets $in$ and $out$ of multi-set names contained in the signature of $A$, such that the rules in $A$ change the multi-sets in $out$ only by *adding* elements, unless they are also in $in$.

Here, each interactive ASM $A$ has two rules, $\mathbf{Initialize}(A)$ and $\mathbf{Main}(A)$, and is executed by first firing $\mathbf{Initialize}(A)$ and then iterating $\mathbf{Main}(A)$ a finite number of times.

**Definition 2** The *input/output behavior* of an interactive ASM $(A, \mathsf{inQueue}(A), \mathsf{outQueue}(A))$ is a function $[\![A]\!]$ from finite sequences of multi-sets of events to sets of finite sequences of multi-sets of events defined as follows. Given a sequence $I_1, \ldots, I_n$ of multi-sets, the execution of the following ASM rule defines a value for $\mathsf{outlist}(A)$ depending on the resolution of possible **choice** rules in $A$. $[\![A]\!](I_1, \ldots, I_n)$ is defined to be the set of possible contents of $\mathsf{outlist}(A)$.

```
Rule IO(A)
seq outlist(S) := ∅
    Initialize(A)
    loop i through list [1...n]
        seq inQueue(A) := inQueue(A) ⊎ I_i
            Main(A)
            outlist(A) := outlist(A).outQueue(A)
            outQueue(A) := ∅
        endseq
endseq
```

# 3. FORMAL SEMANTICS FOR UML

Objects, and more generally components in a system, can communicate by exchanging messages. These consist of the message name and arguments. The set of message names **MsgNm** is partitioned into sets of operations **Operation**, signals **Signal**, and return messages **Return**. For each operation $op \in$ **Operation** there is a return signal $\mathsf{return}_{op} \in$ **Return**. The set **Events** consists of messages $msg(exp_1, \ldots, exp_n)$ for $msg \in$ **MsgNm** and $exp_i \in$ **Exp**. We model sending a message $msg = op(exp_1, \ldots, exp_n) \in$ **Events** from an object $S$ to an object $R$ as follows:

(1) The object $S$ places the message $R.msg$ into its multiset $\mathsf{outQueue}(S)$.

(2) A scheduler distributes the messages from out-queues to the intended in-queues (while removing the message head); in particular, $R.msg$ is removed from $\mathsf{outQueue}(S)$ and $msg$ added to $\mathsf{inQueue}(R)$.

(3) The object $R$ removes $msg$ from its in-queue and processes its content.

We write **Action** for the set of actions which are expressions of the following forms:

**Call action:** $\mathsf{call}_{op(a_1,\ldots,a_n)}$ for an $n$-ary operation $op \in$ **Operation** and expressions $a_i \in$ **Exp** (called the *arguments* of $op$).

**Send action:** $\mathsf{send}_{sig(a_1,\ldots,a_n)}$ for an $n$-ary signal $sig \in$ **Signal** and argument $a_i \in$ **Exp**.

**Return action:** $\mathsf{send}_{\mathsf{return}_{op}(a)}$ for an operation $op \in$ **Operation** with return value $a \in$ **Exp**.

**Assignment:** $att := exp$ where $att \in$ **Attribute** is an attribute and $exp \in$ **Exp** an expression.

**Void action:** $nil$

We fix a set **Activity** whose elements represent the activities that may be used or explained in a UML specification. We assume that it contains an element $nil \in$ **Activity** representing absence of activity. We assume that for every activity $actv \in$ **Activity** there is an associated ASM rule $\mathbf{ActvRule}(actv)$. The activity $nil$ has the associated ASM rule that sets $\mathsf{finished}$ to $true$.

# 4. STATECHART DIAGRAMS

For readability, we give the formal semantics for statecharts that are simplified as follows (our semantics can however be extended straightforwardly to the general case).

- Events can not be deferred.
- There are no history states.
- Transitions may not cross boundaries within or outwith composite states; transitions from composite states must be completion transitions.

## 4.1 Abstract Syntax of Statechart Diagrams

We will define the abstract syntax of statechart diagrams.

A statechart diagram $D = (\mathsf{Object}_D, \mathsf{Class}_D, \mathsf{States}_D, \mathsf{Initial}_D, \mathsf{Transitions}_D)$ is given by an object name $\mathsf{Object}_D$, a class name $\mathsf{Class}_D$, a set of *states* $\mathsf{States}_D$, an *initial state* $\mathsf{Initial}_D$, and a set of transitions $\mathsf{Transitions}_D$. $\mathsf{States}_D$ is a set of tuples $S = (\mathsf{name}(S), \mathsf{kind}(S), \mathsf{entry}(S), \mathsf{init}(S), \mathsf{state}(S), \mathsf{internal}(S), \mathsf{exit}(S))$ where

- $\mathsf{name}(S)$ is the *name* of the state,
- $\mathsf{kind}(S) \in \{\text{initial, final, simple, concurrent, seq}\}$
- $\mathsf{entry}(S) \in \textbf{Action}$ is called the *entry action*,
- $\mathsf{init}(S) \in \mathsf{States}_D \cup \{undef\}$ is the *initial substate*,
- $\mathsf{state}(S) \subseteq \mathsf{States}_D$ is the set of substates of $S$,
- $\mathsf{internal}(S) \in \textbf{Activity}$ is the *internal activity*,
- $\mathsf{exit}(S) \in \textbf{Action}$ is the exit action,

under the following conditions:

- for states $S, T \in \mathsf{States}_D$ with $S \neq T$ we have $\mathsf{name}(S) \neq \mathsf{name}(T)$ and $\mathsf{state}(S) \cap \mathsf{state}(T) = \emptyset$,
- $\forall S \in \mathsf{States}_D.\mathsf{init}(S) \in \mathsf{state}(S) \wedge S \notin \mathsf{state}(S)$,
- $\mathsf{kind}(S) \in \{\text{initial, final, simple}\} \Rightarrow \mathsf{state}(S) = \emptyset$,
- if $\mathsf{kind}(S) \in \{\text{initial, final}\}$, we have $\mathsf{entry}(S) = nil$, $\mathsf{internal}(S) = nil$, and $\mathsf{exit}(S) = nil$
- $\mathsf{kind}(S) \in \{\text{concurrent, seq}\} \Rightarrow \mathsf{internal}(S) = nil$,
- if $\mathsf{kind}(S) = \text{concurrent}$, we have $\mathsf{init}(S) = undef$.

$\mathsf{Transitions}_D$ is a set of tuples $t = (\mathsf{source}(t), \mathsf{event}(t), \mathsf{guard}(t), \mathsf{action}(t), \mathsf{target}(t), \mathsf{intern}(t))$ where

- $\mathsf{source}(t) \in \mathsf{States}_D$ is the *source state* of $t$,
- $\mathsf{event}(t) \in \textbf{Events}$ is the triggering event of $t$,
- $\mathsf{guard}(t) \in \textbf{BoolExp}$ is a Boolean expression,
- $\mathsf{action}(t) \in \textbf{Action}$ is an action,
- $\mathsf{target}(t) \in \mathsf{States}_D$ is the *target state* of $t$, and
- $\mathsf{intern}(t) \in \textbf{Bool}$ is a Boolean.

$\mathsf{event}(t)$ must be of the form $op(exp_1, \ldots, exp_n) \in \textbf{Events}$ where $exp_1, \ldots, exp_n \in \textbf{Var}$ are *variables* (called *parameters*), which must be mutually distinct. As in [3], we assume a special event $\mathsf{ComplEv} \in \textbf{Events}$ (with no parameters) and call a transition $t$ with $\mathsf{event}(t) = \mathsf{ComplEv}$ a *completion transition*. If $\mathsf{intern}(t) = true$ then $t$ is called an *internal* transition, otherwise it is called *external*. Transitions from initial states must have the guard *true*. Final states can not have outgoing transitions. Multiple completion transitions leaving the same state must have mutually exclusive guard conditions. We assume that return messages are given explicitly in the diagrams.

## 4.2 Behavioral semantics

We give a semantics extending the one in [1]. According to our aims, we add mechanisms to model actions and internal activities explicitly (rather than treating them as atomic given events), as well as the operations and their parameters employed in them, and to provide message-passing between different diagrams, including a dispatching mechanism for events and the handling of actions.

We fix a statechart diagram $D$ modeling an object $O \stackrel{\text{def}}{=} \mathsf{Object}_D$ and give its behavioral semantics as an interactive ASM $([\![D]\!]^{SC}, \mathsf{inQueue}(O), \mathsf{outQueue}(O))$.

The signature of $[\![D]\!]^{SC}$ consists of the following names:

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

The Boolean $\mathsf{finished}_S$ may be set to *true* at the end of an ASM interpretation $\textbf{ActvRule}(\mathsf{internal}(S))$ of an internal activity at state $S$ to indicate that $S$ is finished.

The formal interpretation of the actions (when executed by an object $O$) is given by ASM rules of the following form:

**Call action:** We define the ASM rule:

*Rule* $\textbf{ActionRule}(\mathsf{call}_{\mathbf{op[args]}})$
  $\mathsf{outQueue}(O) := \mathsf{outQueue}(O) \uplus \{op_O[args]\}$

**Send action:** We define the ASM rule:

*Rule* $\textbf{ActionRule}(\mathsf{send}_{\mathbf{e}})$
  $\mathsf{outQueue}(O) := \mathsf{outQueue}(O) \uplus \{e\}$

**Return action:** We define the ASM rule:

*Rule* $\textbf{ActionRule}(\mathsf{send}_{\mathsf{return}_{\mathbf{op}}(\mathbf{a})})$
  $\mathsf{outQueue}(O) :=$
    $\mathsf{outQueue}(O) \uplus \{\mathsf{trigsusy}(op).\mathsf{return}_{op}(a)\}$

**Assignment:** $att := exp$ is interpreted (trivially) by the ASM rule

*Rule* $\textbf{ActionRule}(\mathbf{att} := \mathbf{exp})$
  $att := exp$

**Void action:** $nil$ is interpreted as the ASM rule **skip**.

State machines process one event at a time and finish all consequences before processing the next event. There are semantic variation points wrt. dispatching events and choosing between conflicting transitions, which in our semantics are left open (following [3]). In accordance with the UML specification, among conflicting transitions with nested source states those transitions with the innermost source state have priority.

The ASM $[\![D]\!]^{SC}$ has two rules, $\textbf{SCInitialize}(D)$ and $\textbf{SCMain}(D)$, given below (both are defined using other rules defined in the rest of the subsection). 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* $\textbf{SCInitialize}(\mathbf{D})$
**do − in − parallel**
  $\mathsf{inQueue}(\mathsf{Object}_D) := \emptyset$
  $\mathsf{outQueue}(\mathsf{Object}_D) := \emptyset$
  $\mathsf{currState} := \{\mathsf{Initial}_D\}$
  $\mathsf{finished}_{\mathsf{Initial}_D} := false$
**enddo**

*Rule* **SCMain(D)**
**seq if** $\mathsf{Completed} \neq \emptyset$ **then** $\mathsf{eventExecution}(\mathsf{ComplEv})$
**else choose** $e : e \in \mathsf{inQueue}(\mathsf{Object}_D)$
**seq** $\mathsf{inQueue}(\mathsf{Object}_D) := \mathsf{inQueue}(\mathsf{Object}_D) \setminus \{e\}$
**if** $e = op_{sender}[args] \in$ **Operationthen seq**
$e := op[args]$ $\mathsf{trigsusy}(e) := sender$ **endseq**
$\mathsf{eventExecution}(e)$
**endseq**
**loop** $S$ **through set** currState
**seq** $\mathsf{finished}_S := false$
**ActivityRule**(internal(**S**))
**endseq**
**endseq**

Here Completed is a syntactic macro as follows:

$$\{S \in \mathsf{States}_D : (\exists t \in \mathsf{Transitions}_D.\mathsf{source}(t) = S$$
$$\wedge \mathsf{event}(t) = \mathsf{ComplEv}) \wedge$$
$$(\mathsf{kind}(S) = \text{initial} \vee \ \mathsf{finished}_S$$
$$\vee\ (\mathsf{kind}(S) \in \{\text{seq, concurrent}\}$$
$$\wedge\ \forall T \in \mathsf{state}(S) \cap \mathsf{currState}.\mathsf{kind}(T) = \text{final})\})$$

The macro $\mathsf{eventExecution}(e)$ (for an event $e$) is defined as follows:

$\mathsf{eventExecution}(e) \equiv$
**choose** $t : t \in \mathsf{FirableTrans}(e)$
**if** $\mathsf{intern}(trans)$ **then** $\mathsf{execEv}(trans, e,)$
**else**
**seq**
$\mathsf{exitState}(\mathsf{source}(t))$
$\mathsf{execEv}(t, e)$
$\mathsf{enterState}(\mathsf{target}(t))$
**endseq**

$\mathsf{FirableTrans}(e)$ is defined as follows.

For any transition $t$ we define $\mathsf{enabled}(t, \mathsf{ComplEv}) \stackrel{\text{def}}{=} true$ if the following conditions are fulfilled (otherwise it is false):

- $\mathsf{event}(t) = \mathsf{ComplEv}$,
- $\mathsf{guard}(t)$ is $true$,
- $\mathsf{source}(t) \in \mathsf{currState}$,
- $\mathsf{source}(t) \in \mathsf{Completed}$.

For any transition $t$ and any event $e \neq \mathsf{ComplEv}$ we define $\mathsf{enabled}(t, e) \stackrel{\text{def}}{=} true$ if the following conditions are fulfilled (otherwise it is false):

- the operation or signal names of $\mathsf{event}(t)$ and $e$ coincide: $\textbf{msgname}(\mathsf{event}(t)) = \textbf{msgname}(e)$,
- $\mathsf{guard}(t)$ evaluates to $true$ when its variables are substituted with the arguments of $e$,
- $\mathsf{source}(t) \in \mathsf{currState}$,

Given an event $e$, the nesting of states induces a total order $\leq$ on the set of transitions such that $\mathsf{enabled}(t, e)$ holds, by defining $t_1 \leq t_2$ if the source state of $t_1$ is a (possibly nested) substate of the source state of $t_2$. Let $\mathsf{FirableTrans}(e)$ be the set of transitions $t$ with $\mathsf{enabled}(t, e)$ that are minimal wrt. $\leq$ (the set of enabled transitions with the innermost state).

We define the macro $\mathsf{exitState}(S)$ for a state $S$:

$\mathsf{exitState}(S) \equiv$
**do − in − parallel**
**if** $\mathsf{state}(S) \cap \mathsf{currState} \neq \emptyset$
**then**
**loop** $T$ **through set** $\mathsf{state}(S) \cap \mathsf{currState}$
$\mathsf{exitState}(S)$
**else do − in − parallel**
$\mathsf{currState} := \mathsf{currState} \setminus \{S\}$
**ActionRule**(exit($S$))
**enddo**
**enddo**

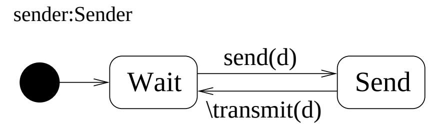


**Figure 1: Sender**

The macro $\mathsf{execEv}(t, e)$ (for a transition $t$ and an event $e$) is defined as follows:

$\mathsf{execEv}(t, e) \equiv$
**seq**
**Args**(event($t$)) := **Args**($e$)
**ActionRule**(action($t$))
**endseq**

We define the macro $\mathsf{enterState}(S)$ for a state $S$:

$\mathsf{enterState}(S) \equiv$
**do − in − parallel**
$\mathsf{currState} := \mathsf{currState} \cup \{S\}$
**ActionRule**(entry($S$))
**if** $\mathsf{kind}(S) = \text{seq}$ **then** $\mathsf{currState} := \mathsf{currState} \cup \{\mathsf{init}(S)\}$
**else** $\mathsf{currState} := \mathsf{currState} \cup \mathsf{state}(S)$
**enddo**

## 4.3 Example

The statechart *sender* given in Figure 1 is interpreted by the interactive ASM $(\llbracket sender \rrbracket^{SC}, \mathsf{inQueue}(sender), \mathsf{outQueue}(sender))$ whose main behavior is equivalent to that given by the following rule.

**case** currState **of**
$\{\mathsf{Initial}_{sender}\}$ : **do** $\mathsf{currState} := \{Wait\}$
$\{Wait\}$ : **do**
**choose** $e : e \in \mathsf{inQueue}(sender)$
**do − in − parallel**
$\mathsf{inQueue}(sender) := \mathsf{inQueue}(sender) \setminus \{e\}$
**if** $\textbf{msgname}(e) = send$ **then**
**do − in − parallel**
$\mathsf{currState} := \{Send\}$
$d := \textbf{Args}(e)$
**enddo**
**enddo**
$\{Send\}$ : **do**
**do − in − parallel**
$\mathsf{currState} := \{Wait\}$
$\mathsf{outQueue}(sender) := \mathsf{outQueue}(sender)$
$\uplus \{transmit(d)\}$
**enddo**

## 5. RELATED WORK

There has been a considerable amount of work towards a formal semantics for various parts of UML; a complete overview has to be omitted. [4] discusses some fundamental issues concering a formal foundation for UML. [11, 12] gives an approach using algebraic specification. [2] uses a framework based on stream-processing functions. [7] employs graph transformations. [14] gives a semantics for use case diagrams based on the process algebra CCS. Finally, [1] uses ASMs. There has been a lot of work on formal methods for object-orientation in a more general setting beyond UML, cf. e.g. [6, 5].

## 6. CONCLUSION AND FUTURE WORK

To conclude, the formal semantics for UML statechart diagrams presented here seems to provide a significant further step towards formal modeling of complete UML specifications, going beyond the formal models of single diagrams in isolation presented so far. Since our semantics is the first semantics to explicitly model actions, internal activities, operations with their parameters, message-passing between different diagrams and event dispatching, it provides a first foundation for executable UML modeling. For space reasons, we only present the semantics for a simplfied kind of statecharts; the extension to the full definition of UML statecharts gives increased complexity, but no problems in principle.

While this work has already been extended to the other UML diagrams (such as sequence diagrams), this has to be left out here for space reasons.

The ultimate goal is to allow whole systems of UML specifications (rather than single diagrams) to be simulated.

*Acknowledgements.* Discussions with A. Cavarra about formal semantics for UML and constructive comments by the referees on the presentation of the paper are gratefully acknowledged.

## 7. REFERENCES

[1] 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 *Lecture Notes in Computer Science*, pages 223–241. Springer-Verlag, 2000.

[2] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, views and models of UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, 1998.

[3] A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method.* PhD thesis, DMI, Universitá di Catania, 2000.

[4] 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*, Lecture Notes in Computer Science, pages 297–307. Springer-Verlag, 1999.

[5] P. Gibson. Formal object oriented requirements: simulation, validation and verification. In *Modelling and Simulation: A tool for the next millenium, vol II.*, pages 103–111. SCS, 1999. European Simulation Multi-conference (ESM99).

[6] P. Gibson and D. Méry. Fair objects. In *Object-oriented technology and computing systems re-engineering.* Horwood Publishing, 1999.

[7] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *PSMT'98.* TU München, TUM-I9803, 1998.

[8] H. Hußmann, editor. *Fundamental Approaches to Software Engineering (FASE, 4th International Conference)*, volume 2029 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[9] J. Jürjens. Towards development of secure systems using UMLsec. In Hußmann [8], pages 187–200. Also OUCL TR-9-00 (Nov. 2000), http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-9-00.html.

[10] J. Jürjens. Using UMLsec and Goal-Trees for Secure Systems Development. In *Symposium of Applied Computing.* ACM, 2002.

[11] G. Reggio, E. Astesiano, C. Choppy, and H. Hußmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE2000)*, volume 1783 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 2000.

[12] G. Reggio, M. Cerioli, and E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *AMiLP 2000*, 2000.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[14] P. Stevens. On use cases and their relationships in the Unified Modelling Language. In Hußmann [8], pages 140–155.

[15] UML Revision Task Force. OMG UML Specification v. 1.4. OMG Document ad/01-09-67. Available at http : //www.omg.org/uml, Sept. 2001.

## 8. BIOGRAPHY

**Jan Jürjens** is a researcher at the Technical University of Munich (Germany). He has (co)authored about 20 papers in international refereed journals and conferences, mostly on computer security and software engineering. He has created and lectured a course on secure systems development at the University of Oxford. Received awards include a scholarship from the German National Merit Foundation (Studienstiftung des deutschen Volkes, awarded to top 0.5% of German students) and a best student paper award from IFIP SEC. He has studied Mathematics and Computer Science at the Univ. of Bremen (Germany) and the Univ. of Cambridge (GB) and received a M.Sc. degree with Distinction from the Univ. of Bremen. He has done research towards a PhD at the Univ. of Edinburgh (GB), Bell Laboraties (Palo Alto, USA), and the Univ. of Oxford (GB) and is currently finishing his DPhil (Doctor of Philosophy) thesis in Computing from the Univ. of Oxford.