An Approach for an UML Profile for Software Development Process Modeling [⋆]

Ralf Buschermöhle[1] and Wilhelm Hasselbring[2]

[1] Oldenburger Forschungs- und Entwicklungsinstitut
für Informatik-Werkzeuge und -Systeme (OFFIS) e.V.
Department Safety Critical Systems
Escherweg 2
D-26121 Oldenburg, Germany
Email: buschermoehle@offis.de

[2] University of Oldenburg
Software Engineering Group
Department of Computing Science
D-26111 Oldenburg, Germany
Email: hasselbring@informatik.uni-oldenburg.de

Abstract. The status of our current work on an UML profile to express software engineering process models is presented. We will discuss our requirements on such a profile and introduce identified syntactical and semantical implications. Concluding a first application scenario is presented.

1 Introduction

Nowadays various development process models exist for efficient development of software. They range from agile processes, like eXtreme Programming (XP [5]) as a set of "best practices", customizable frameworks such as the Rational Unified Process (RUP [21]) towards very detailed process descriptions such as the V-Model [17]. They all have in common that the process specifications are expressed informally. This often leads to different interpretations which is obviously a problem for (automatic) process execution. The informal description of development processes leads to the problem that it is not well understood what should be coordinated/expressed in a software development process and what not: "To which extent should a software development process be specified?"

There exist a lot of formal software process modeling languages which allow to avoid ambiguities when interpreting the process specifications [18, 19, 15]. Unfortunately none of them achieved the status of a standard notation for software process modeling. All of these approaches define their own syntax and semantics which implies e.g., proprietary data formats. Furthermore all of them are limited to express a fixed set of certain aspects of a process software model (e.g., time,

[⋆] This research was supported by the German Department of Education and Research (BMBF), research project 'ViSEK', under grant 01ISA02G.

data, concurrency) and it is not possible to extend them easily with new modeling constructs e.g., to cover new research topics like "knowledge management" or different semantics (e.g., finite automata, Petri net variants).

On the other hand the Unified Modeling Language (UML [37]) of the Object Modeling Group (OMG [1]) achieved the status of an industrial standard for modeling object oriented systems which resulted in considerable tool support. Even more, UML provides easy to use extension mechanisms (stereotypes, tagged values, constraints) within so called "UML profiles". Unfortunately the UML does not provide formal semantics. There are different approaches published trying to solve this issue (compare Section 2). But they all have in common that they focus only some aspects of the UML diagram conglomerate.

For these reasons we present an UML-based approach to support software development process modeling and execution. The remainder of the paper is structured as follows. In Section 2 we briefly introduce some informal basic definitions and motivate the approach. In section 3 we give a brief overview of the accomplished work. Section 6 concludes and elaborates on further research.

2 Fundamentals

In this paper we discuss the "software dimension" of software development processes, this includes all software related elements that are used in the process, like artifacts and tools. We aim at automating as much of these software elements as possible. We start the discussion with some requirements sketching the "big picture". Then we discuss briefly implications of these definitions with respect to modeling static and dynamic concerns inside an UML profile. Requirements of software development process modeling are:

1. A *software development process* consists of (disjunctive) sets of activities, actors, products and tools ((A)APT).
2. Each AAPT can be *ready* or *not ready* and describes for example the availability of an actor. Even more a resource can be several "times" *ready* meaning that several products are available.
3. Each *activity* has a set of *inputs* and *outputs* that refer to a subset of defined AAPT.
4. If all referred AAPTs of an input are *ready* then the *input* is *enabled.*
5. An *activity* is (possibly) *executed* if at least one input is enabled.
6. A *step* is subset of (possibly executed) activities that is executed concurrently. Especially the concurrent execution of two equal activities is possible e.g., regarding a "coding" activity where more then two persons code at the same time.
7. If an activity is executed the corresponding input is disabled and it's referred AAPTs are set to *not ready*.

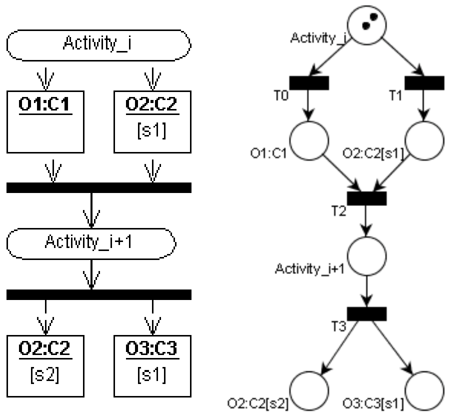
8. If an *activity completes* at least one output is enabled.
9. If an *output is enabled* all referred AAPT are *ready*.
10. Each APT has a set of attributes to store data and states (e.g., actor name, product state).
11. The APTs may have relations among each other (e.g., product a is related to exactly 3 subproducts).
12. *Tools* are able to *change states of products* within a defined set of functions that map a set of products in certain states into a set of products in certain states.
13. Actors can be persons interacting with *tools* or just tools.
14. Actors interact with tools in that way that certain functions are executed.
15. An application of these functions is referred as *activity execution* which can consume resources.
16. Resources can be time, cost, space etc.

If one tries an initial (syntactical) mapping of the identified requirements towards UML diagrams it is obvious that we need to be able to model static and dynamic aspects of a process. The kind of static aspects we need to express are covered in UML only by class diagrams - so there is no design choice. But there are a lot of diagrams dealing with behavioral aspects like state charts, activity charts and message sequence charts. If we assume, that the static instances of actors, products and tools can be described by class diagrams then activity charts offer an adequate syntax to meet the required inputs and outputs (requirement 3). Regarding possible semantics the main characteristic of the process dynamics we sketched so far is "resource driven". Which means, if all required resources are available or ready the activity is executed. Including all UML specifications until version 1.5 [37] the OMG proposes to derive the behavior of activity machines from state machines. This would not be consistent with our requirements e.g., it is not possible to have multiple active states in one state chart. Fortunately, in the actual UML 2 proposals [26] the OMG is changing the behavioral of activity machines towards a Petri net-based semantics which match our identified requirements. A Petri net PN is a tuple (P,T, $\mathbb{F}$, $\mathbb{B}$)

- a finite, ordered set $P = \{p_1, \ldots, p_{|P|}\}$ of places,
- a finite, ordered set $T = \{t_1, \ldots, t_{|T|}\}$ of transitions,
- $P \cap T = 0$,
- $|P| \times |T| - forward - matrix$ $\mathbb{F}$ over $\mathbb{N}$, and
- $|P| \times |T| - backward - matrix$ $\mathbb{B}$ over $\mathbb{N}$
- $F : P \times T \cup T \times P \Rightarrow \mathbb{N}$ the edge function, defined as $\forall x, y \in P \cup T$:
  $$F(x,y) = \begin{cases} \mathbb{F}_{i,j}, if x = p_i \wedge y = t_j \\ \mathbb{B}_{i,j}, if x = t_j \wedge y = p_i \end{cases}$$
- The State space of the Petri net is $\mathbb{N}^P$.
- A mapping $s : P \rightarrow \mathbb{N}$ is called state/marking of the net.
- There exist exactly one initial state s_0.

- We call a transition t of N s-enabled, if $s \geq \mathbb{F}(t)(\forall p \in P : s(p) \geq \mathbb{F}_{p,t} = F(p,t))$.
- t fires from s to s', if s' = s - $\mathbb{F}$(t) + $\mathbb{B}$(t).

Figure 1 shows a simple example of a Petri net-based execution frame for activity charts. In sub-figure (a) the activity chart is presented and in sub-figure (b) the corresponding Petri net. Note that is is not necessary that "Activity_i+1" is ever reached, because $T0$ or $T1$ can also fire twice in standard Petri net token game semantics. Petri nets are widely accepted in the area of business process



(a) simple activity chart (b) simple Petri net

Fig. 1. Example illustrating the semantical petri-net-based execution frame of activity charts

models regarding the execution of so called "workflow specifications" in so called "workflow management systems" [3, 31]. But workflows differ from our point of view from our software process models - which reflects also in the Petri net token game semantics that is used to describe the behavior of workflows.

One characteristic of Petri nets is that they generally describe closed systems. This assumption is appropriate if the process model should "only" be simulated but this is not a realistic assumption if we want to accompany a "real" software development processes where we possibly have to deal with variations over time regarding the resources that can be used. So we need to support semantics that allows to specify so called "reactive systems". Furthermore as one of their main characteristic workflow management systems do not allow to modify the controlled artifacts. But exactly this is an important feature for automating software development processes.

There exist a lot of approaches to express Software Engineering Process Models in UML and UML profiles (e.g., [30, 34, 35]). Most of them have an informal semantics and thus they are outside of our scope — since we aim at allowing (computer supported) software process execution. Regarding the formal semantics one could classify the approaches regarding the level of UML coverage. Many people have defined the semantics of single UML diagrams — e.g., [22, 7] etc., on state machines, [12] on collaboration diagrams, [13, 16] etc. on class diagrams, [28] on use cases, [6] on activity diagrams — or just to give formal foundations for action language (e.g., [25, 2]). This restriction on a single diagram is problematic because the main advantage of UML is the possibility to use different diagrams during the model building process to describe a system. Most of the diagrams are related to each other and thus the formal semantics should have "interfaces" to connect to other approaches in order to complete the picture.

One example of such interfaces in our approach concerns the tool dimension of a software development process. All software development processes have in common that tools are required to support their activities [32, 14, 38]. This is due to the fact that process models usually specify the activities that *should* be accomplished and the tools determine the activities that *can* be accomplished — with various levels of automation. Similar relationships between artifacts and process models resp. tools exist. The relevance of these mutual dependencies is proportional related to the amount of activities for producing artifacts that can be supported by tools. Only then, tools are able to serve as bridges between the process models and the actual processes. The amount of possible automation is steadily increasing — especially in the domain of system development — since only tools are able to support the efficient development of the systems that have a continuously increasing complexity [11]. The relationship between process models and tools is relevant in both directions. On one hand, process models have an impact on the tools On the other hand, the tools have an impact on the process models. The main concern for this direction is the fact that software development processes are similar to software or at least several aspects of them are expressible in software. Therefore, we have to deal with the question how the software environment that is used in a development process can be integrated into a process modeling language. In the next section we are going to introduce some of our design choices and results so far.

3 Work status

First we developed a mapping regarding the syntactical constraints of an established development process model in order to see whether all required elements of the process specification could be expressed. Then we expressed this process model with the chosen UML subset. After this we extended these models with additional activities and products concerning domain specific standards in the area of safety critical systems (like ARP4754 [33] in the area of avionics) and also introduced model-based development extensions (like Life Sequence Charts [9]) to be able to verify systems very early in the process. For this example mapping we used the V-Model for its widespread use in the domain of safety critical systems, and its property of being both very detailed and general. Furthermore this process model is mandatory for software suppliers working for German government organizations. Activities in the V-Model context

> "are work-steps in the IT development process; its results and execution can be described exactly. Activities may consist of a set of sub-activities as long as each of these sub-activities results in defined interim results." ([17], GD250-2EINF, p.5)

In each activity objects are processed, called products. And all products that have to be developed are described (method independently) in detail in the specification which serves also as contract between customer and contractor.

The V-model comprises the four submodules project management (PM), configuration management (CM), quality assurance (QA), and system development (SD). We concentrate on the most elaborated part SD. Instead of insisting on particular languages for products, the V-model in detail specifies what a product shall describe and recommends specific methods, i.e. languages or notations such as flow-charts, or even gives general product templates to be used in particular situations. A product template, for example, specifies that each requirement must have a unique identification number. Furthermore on the method level there are "method interfaces" specified that explain how the methods relate to each other. In order to illustrate the mapping of the V-Model specification elements towards the UML profile elements we will sketch some examples in the next two subsections.

3.1 Activity related specification elements in the V-Model

There are two main specification elements that cover the execution of activities and methods in the V-Model, e.g. "V-Model activity diagrams" and product flows. V-Model activity diagrams are used to sketch a grain granular picture of the relationships between a set of activities and products (example see figure 2). Activities are depicted in rectangles and products in ellipses. Control- and data flows are displayed by arrows, whereby data flows have at least one product as source. These "pictures" have not much in common with the specifications on deeper levels because the are only data flows specified. As already mentioned the lower specification elements provide a more detailed picture of the development activities. Each activity has a so called "product flow" that describes all input and output products. Figure 3 describes all products involved in activity SD 2.5 "Interface Description". For each (sub-) product (column 3) referred to by the activity to be described the state at the beginning (column 2) and that at the end of the activity (column 5) is entered. If the activity does not influence the state of the product or should no such state of the product exist, then this is

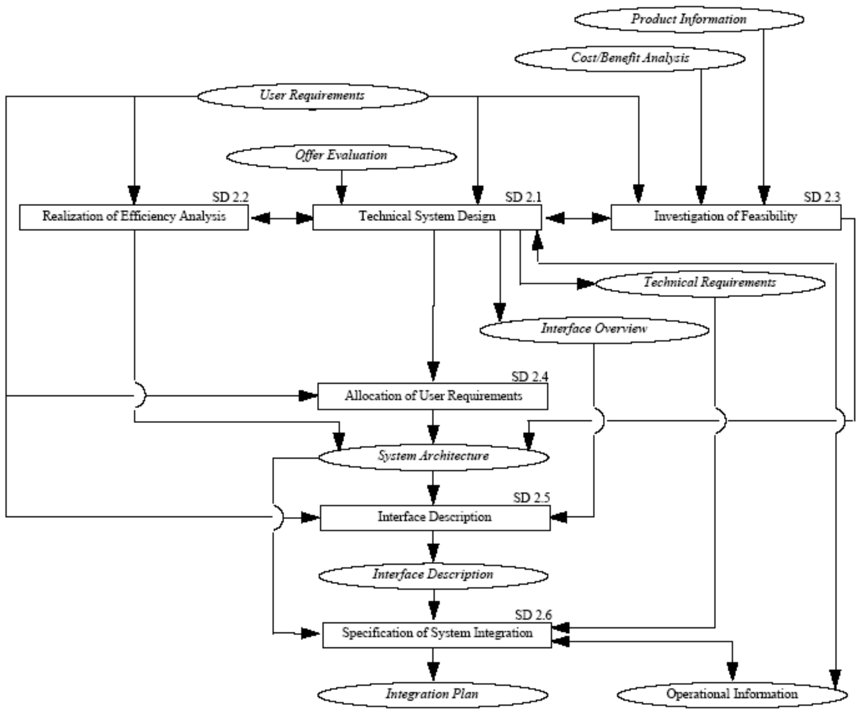
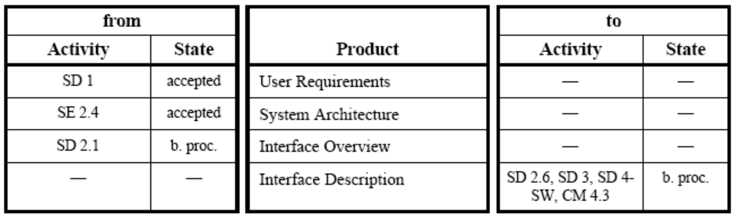


Fig. 2. Example Activity SD 2: System Design (V-Model Spec.)

marked in the table by a dash. The input products of an activity are identified in such a manner that the columns "from Activity" and "from State" are filled in and the columns "to Activity" and "to State" are marked by a dash. For output products the "From"-entries are not applicable. Only the columns "to Activity" and "to State" are filled in. In the cases where a product has both "from" and "to" entries it is modified in the corresponding activity. All output products of an activity whose end states are "b. proc." should have "planned" as beginning states according to the model. In order to be able to distinguish better visually between input and output products, the beginning state has been substituted by a "—". This should be interpreted as "planned" in these cases. Furthermore, it is noted for each (sub-) product, from which activity the product results (column 1) and to which activity the product will be passed (column 4). If there are neither "from" nor "to" activities for a (sub-) product this is illustrated in the table by a dash. If sub-products of a product are created in different (sub-) activities (see, e. g., activity QA 2.2 "Definition of Assessment Environment"), it will become necessary to assemble the product by integrating the sub-products. This is realized in the activity where the last generated sub-product of a product is created. In the product flow this is represented by referring to following main activities in column "to Activity". For products that are not updated, the state



from			to	
Activity	State	Product	Activity	State
SD 1	accepted	User Requirements	—	—
SE 2.4	accepted	System Architecture	—	—
SD 2.1	b. proc.	Interface Overview	—	—
—	—	Interface Description	SD 2.6, SD 3, SD 4-SW, CM 4.3	b. proc.

Fig. 3. Example product flow SD 2.5: Interface Description (V-Model Spec.)

in "to State" is "submitted".

The example shown in Figure 3 means that "User Requirements" and "System Architecture" come from activities SD1 and SD2.4, and represent input products. Both products have state "accepted". Product "Interface Description" is newly created. The product leaves the activity having the state "p.proc" and is input product for activities SD2.6, SD3, SD4-SW, CM4.3. Figure 4 shows an aggregated activity diagram of figure 2 and the relevant part of figure 3. In the

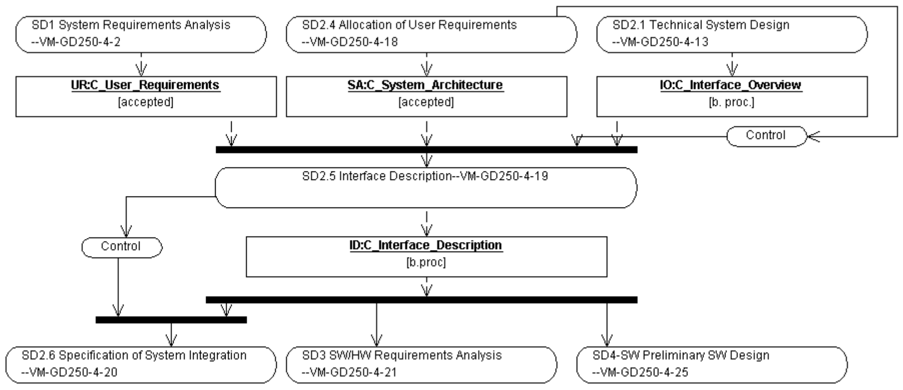
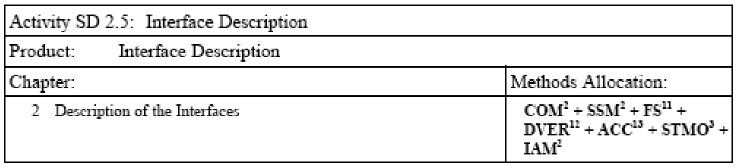

SD1 System Requirements Analysis --VM-GD250-4-2
SD2.4 Allocation of User Requirements --VM-GD250-4-18
SD2.1 Technical System Design --VM-GD250-4-13
UR:C_User_Requirements [accepted]
SA:C_System_Architecture [accepted]
IO:C_Interface_Overview [b. proc.]
Control
SD2.5 Interface Description--VM-GD250-4-19
Control
ID:C_Interface_Description [b.proc]
SD2.6 Specification of System Integration --VM-GD250-4-20
SD3 SW/HW Requirements Analysis --VM-GD250-4-21
SD4-SW Preliminary SW Design --VM-GD250-4-25

Fig. 4. SD 2.5: UML activity diagram

activity diagrams all resources (products, actors, tools) are objects. Furthermore in the diagram of figure 4 the objects have a state that corresponds to the "product flow" of the V-Model. The next activity level that is specified in the V-Model are the methods. There are a lot of information that describe for example what to do in each method, limits and recommendations during the method application or sketch interfaces among the methods. But the assignment of methods to activities is done by allocation a set of methods to each activity. Figure 5 shows an assignment of the methods "Class and Object Modeling" (COM), "SubSystem Modeling" (SSM), "Formal Specification" (FS), "Design VERification" (DVER),



Activity SD 2.5: Interface Description	
Product: Interface Description	
Chapter:	Methods Allocation:
2 Description of the Interfaces	COM^2 + SSM^2 + FS^{11} + DVER^{12} + ACC^{13} + STMO^5 + IAM^2

Fig. 5. SE 2.5 method assignment

"Analysis of Covered Channels" (ACC), "State Machine Modeling" (STMO) and "InterAction Modeling" (IAM) to activity SD2.5 "Interface Description". We further assumed that methods are just activities without a specified activity flow because this would exceed the limits of a process model description (for example the V-Model specifies over 60 methods). Nevertheless it would be useful to have such descriptions - especially if most parts are inherently dynamic between most process runs. Figure 6 depicts a possible flow of methods. First

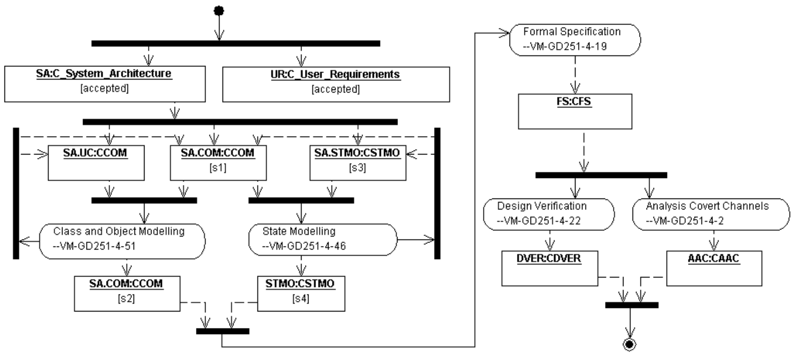


Fig. 6. SE 2.5 methods: UML activity diagram

the input products of the super activity (SE2.5) are instantiated, namely the objects "System Architecture" (SA) and UR (User Requirements) according to the product flow table of the activity. Then these products are split into the required sub-products that should be modified by method application, namely in figure 6 "System Architecture.Use Cases" (SA.UC), "System Architecture.Class and Object Model" (SA.COM) in state "s1" and "System Architecture.State Modeling" (SA.CSTMO) in state "s3". The methods "Class and Object Modeling" and "State Modeling" should be executed sequentially infinitely often till both produced the objects "SA.COM" in state "s2" and "SA.STMO" in state "s4". The changed states denote in this diagram an object change. What changed exactly is specified somewhere else. Figure 7 shows the corresponding underlying Petri net semantics. Now we are able to use activity diagrams for software

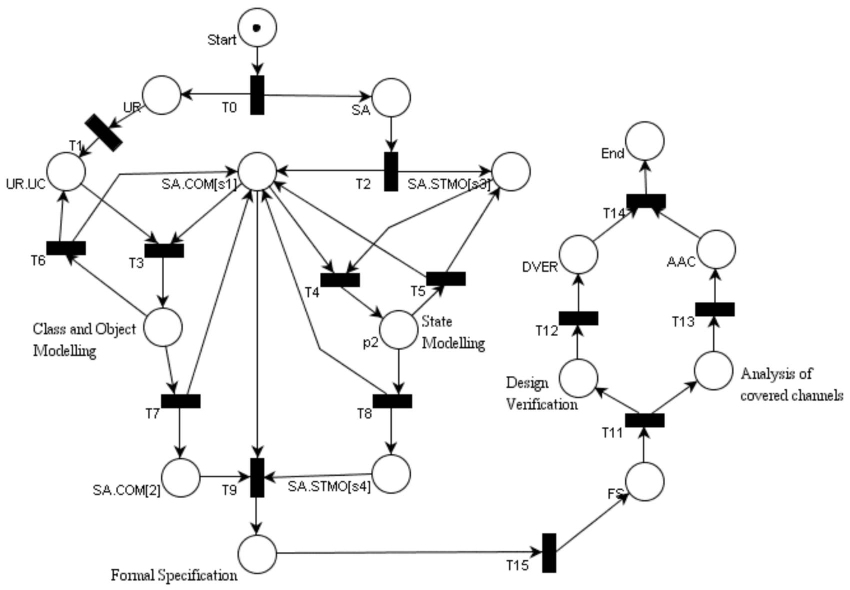


Fig. 7. SE 2.5 methods: (underlying) Petri net semantics

process modeling. But we did not show yet how static aspects (data) can be expressed. Even more we assumed that objects are "created" - but we did not specify how. According to our defined requirements we would assume that tools exist that are able to create, modify und (possibly) destruct the products.

3.2 Product related specification elements in the V-Model

There are several product related specification elements in the V-Model. This includes for example general product information (like: name and identification of the document, version, person in charge), product information (for example for user requirements) and activity-related product information (like: user requirements that describe the actual status of an existing old system, that should be (partly) incorporated into the new one in activity SD1.1 "Recording Actual Status and Analysis"). Even more relationships between methods are specified, for example

> "Interface Design Verification - Formal Specification
> DVER requires a formally specified detailed specification for to be verified and formally specified starting specification. These specifications should be written in the same specification language."

We mapped these specifications into UML class models. Within their syntactical boundaries we were able to express

– Product relationships with multiplicities that denote that a product has a relationship to other products, for example each state chart has exactly one class diagram.
– Product instances that describe inheritable product attributes, for example each products inherits all general product requirements.
– Product hierarchies describe product affiliations, for example that product "System Architecture" consists of class diagrams, state charts and a design verification proof.

Figure 8 depicts some method results of activity SE2.5. The "Interface Description" product has the two sub-products "Class and Object Model" and "State Model". The "C" at the beginning of the class names show that we talk about "Classes" - thus product templates in the V-Model context. Furthermore each of the sub-products of the class "Interface Description" inherits the attributes "InterfaceID" of type "Unique ID" (UID) and a string that informs about the purpose of the interface. Even more relation "R3" ensures that each class has exactly one state chart.

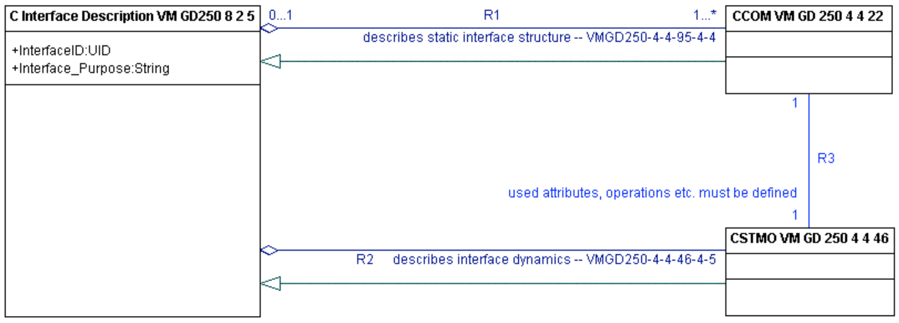

C Interface Description VM GD250 8 2 5
+InterfaceID:UID
+Interface_Purpose:String
0...1
R1
1...*
describes static interface structure -- VMGD250-4-4-95-4-4
CCOM VM GD 250 4 4 22
1
R3
used attributes, operations etc. must be defined
1
CSTMO VM GD 250 4 4 46
R2
describes interface dynamics -- VMGD250-4-4-46-4-5

Fig. 8. SE 2.5: method products

Comprising the above we were able to express specification elements that were syntactically clear separable e.g., product flows, method assignments and method interfaces inside a UML profile that consists syntactically mainly of activity machines and class diagrams. We did not express large text elements in "pure" natural language directly, since the possible interpretation range would be impractically wide. In this case we referred in the diagrams to the unique specification ID. Nevertheless we assume that it is possible to express these elements also inside the chosen UML set. We introduced our models on the last user conference of the V-model, called "ANSSTAND" in October 2003 [4].

4 Application Scenario: Process Model Monitoring

Unfortunately the diagrams provide a very grain granular picture of the development activities that have to be done because they are not complete. Regarding the main activities the flow was incomplete and unclear product flow as well as control flow. Regarding the activity level (every dynamic aspect above the method level) it was for example unclear what products should be locked and it is obvious that the QS states are not enough to express all (relevant) product changes. Regarding the method level the V-Model authors specified just a set of methods but did not how these should be executed. In several discussions during the ANSSTAND user forum and in several other meetings with industrial partners it turned out that exactly these were the major problems when the V-Model is applied. Even more when using process models a lot of experience is needed to ensure that a process is executed in the "right way" due to several from project to project changing context factors, like process targets (time-to-market, budget, quality) resources (people, software) and evolving methods and techniques. Nevertheless we believe that it is possible to "sharpen" these pictures of a process with appropriate tool support. Figure 9 shows the "big picture" regarding a process model monitoring. Process model monitoring means tracking a certain set of process information and comparing them with an actual process model to find out how "good" the development works. There exist already some approaches in this area but their problem is that the tracking is mostly manual which suffers from several drawbacks, like:

– Incompleteness because often only a certain detail level can be tracked.
– Cost and time intensiveness because extra personal is needed to track the development activities and their results.
– Disturbance of the "normal" development activities.
– Faults in the mapping towards the chosen PML.
– Inconsistencies of several development tracks, especially when different "trackers" are involved.
– Snapshot character of the captures because the tracking is done once or twice and often even not the whole life cycle of a product.

For these reasons we try to develop techniques that are able to sketch development activities more automatically through a "tool's perspective" [8]. The first step in this direction is in figure 9 depicted. The arrows in this figure denote information flows. As already mentioned the UML diagrams of the V-Model are not complete. Nevertheless these diagrams provide a (first) base to structure the development. For this reason we use these information to inform a process executer via the "Mediator" about possible process steps that should be executed (A). Afterwards the process executer uses the "Mediator" to inform about the artifacts that were developed (B). All the modified or created artifacts are analyzed by a static analysis based on the JavaCC parser [20] to see what changes in the artifacts (for example the number of methods in a Java program). Furthermore we mapped the grammar files also in UML class diagrams that describe the

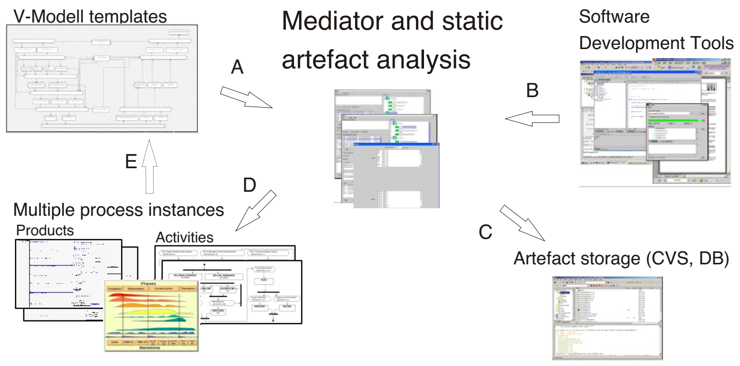

V-Modell templates
Mediator and static artefact analysis
Software Development Tools
A
B
C
D
E
Multiple process instances
Products
Activities
Artefact storage (CVS, DB)

Fig. 9. Process Monitoring

static structure of the artifacts in the process model context. Therefore we are able to draw (detailed) versions of the V-Model templates according to the modified and created artifacts (D). Since we would like to provide a "post mortem analysis" of the executed process we also store the artifacts on a CVS/DB server (C). If enough of these process runs are captured they have an influence of the templates (E).

5 Formal Kernel Semantics

In the first step to formalize the chosen UML subset we defined a so called "kernel semantics" that can deal with every element of the syntax but provides only the necessary semantics on a low level. This is done because the developed semantics is much easier to handle if it is necessary to add additional constructs later on and we can already integrate necessary "interfaces" to other semantics as needed. The two bases for these semantics are the activity diagram semantics introduced in the thesis of Erik Eshuis [31] and the kernel language of the state machine semantics developed by Damm et.al [10]. The activity machine semantics was chosen because Eshuis developed so called "reactive" Petri nets that are able to deal with events. But his semantics do not include several aspects activity machines offer like an action language, object flows or triggered operation calls. The state machine semantics was chosen to have a base for a kernel semantics that can deal with every aspect of state machines in order to complete the "big picture". This section will only give a short overview on the developed semantics in order to sketch ones of the differences between activity machine and state machine semantics we had to deal with. A *krtUML* model $M = (T, F, Sig, <, C, c_{root}, A)$ consists of the following:

- $T \supseteq \{\texttt{void}, \mathbb{B}, \mathbb{N}\}$: A set of *basic types* comprising at least booleans and natural numbers.
- F: A set of typed *predefined primitive functions*.

- Sig: A finite set of *signals*. Every instance of a signal is called *signal event*, or *event* for brevity.
- $< \subset Sig \times Sig$: A *generalization relation* on signals, i.e. the transitive closure $<^+$ is irreflexive, where $ev_1 < ev_2$ denotes that ev_2 is a generalization of ev_1. In the following, we use $\leq$ to denote the reflexive transitive closure of $<$.
- C: A finite, non-empty set of *classes*. A class

$$c = (c.isActive, c.attr, c.ops, c.sm, c.act)$$

  consists of:
  - $c.isActive$: A predicate. Class $c \in C$ is called *active* iff $c.isActive = true$.
  - $c.attr$: A finite set of typed *attributes*, which may not be of type `void`.
  - $c.ops$: A finite set of typed *triggered operations*.
  - $c.sm$: A *c-state machine* in terms of c-actions over c-expressions.
  - $c.act$: A *c-activity machine* in terms of c-actions over c-expressions.
- $c_{root} \in C$: The class of the *root object* (serving to specify system initialization).
- $A \subset C$: A subset of active classes called *actors* and used to denote external objects (part of the environment).

Each definition element (e.g., Class, Action, Expression, Guard) is typed consisting according to the defined *basic types*. An c-activity machine for a class $c \in T_C$ is a tuple $c.act = (c.Q, c.q_0, c.Q_x, c.Tr)$, where:

- $c.Q$ is a finite set of *activities*.
- $c.q_0 \in c.Q$ is the *initial activity*.
- $c.Q_x \subset c.Q$ is a set of *termination activities* with $c.q_0 \notin c.Q_x$.
- $c.Tr \subseteq \{S | S \subseteq c.Q\} \times (\{\gamma \mid \gamma \text{ is a } c\text{-guard or } c\text{-action}\}) \times \{T | T \subseteq c.Q\}$ is the *transition relation* and $\forall tr = (S, \gamma, T) \in c.Tr$:
  - $c.q_0 \in S \Rightarrow S = \{c.q_0\} \wedge \gamma = \text{“}create_c\text{”}$
  - $c.Q_x \cap T \neq 0 \Rightarrow T \subseteq c.Q$.
- Class $c \in C$ is called *reactive* if there is a transition $(S, \gamma, T) \in c.tr$ such that $c.q_0 \notin S$ and γ is in the form $ev[expr]$ or $op[expr]$ for some $ev \in Sig$ or $op \in c.ops \setminus \{create_c\}$.

We manage objects in so called *object configurations* that we assume to exist for each object. Such object configurations store the status (e.g., *dormant*, *idle*, *executing*, *suspended*, *dying*, *dead*), the attribute configuration (the values of all attributes), the configuration (in a state machine the active state, in an activity machine similar to a marking of a Petri net) and the event queue of an object. The *system configuration* is a set of all object configurations.
We use so called *pending request tables* to store triggered operation calls. Each triggered operation call consists of the destination of the call, the status of the operation (*unused*, *pending*, *busy*, *completed*), the result of the operation and

the parameters of the operation. One of the main differences between objects of a state machine and objects of an activity machine we made was that the first ones can be grouped inside a "component" where only one of the objects is active in a certain point in time. The rest of the objects is not executed (for example *dormant* or *dead*) or *suspended* due to pending operation calls to other objects. In contrast activity machine objects are always active because of their resource driven token switching. This has for example an impact on the destination attribute of a pending request table. If the destination of a call is a state machine object then the destination field contains a reference to one object of the component. Otherwise the destination field contains a transition of an object's activity machine. The semantics of the *krtUML* was defined in terms of a "symbolic transition system", proposed in [23] under the name Synchronous Transition System. In such a system all variables are mapping to values of their domains in so called "snapshots".

A *symbolic transition system* (STS) $S = (V, \Theta, \rho)$ consists of V, a finite set of typed *system variables*, Θ, a first-order predicate over variables in V characterizing the initial states, and ρ, a *transition predicate*, that is a first-order predicate over V, V', referring to both primed and unprimed versions of the system variables (their current and next states).

An STS *induces* a transition system on the set of interpretations of its variables as follows. Let $S = (V, \Theta, \rho)$ be an STS and $\mathcal{T}$ the set of types of variables in V. Let $\mathcal{D}_\tau$ be a semantic domain for each $\tau \in \mathcal{T}$.

– A *snapshot*
$$s : V \rightarrow \bigcup_{\tau \in \mathcal{T}} \mathcal{D}_\tau$$
of S is a type-consistent interpretation of V, assigning to each variable $v \in V$ a value $s(v)$ over its domain. Σ denotes the set of snapshots of S.

– A snapshot $s \in \Sigma$ inductively defines the value $[\![expr]\!](s)$ for first-order predicates '*expr*' over V and the value $[\![expr]\!](s, s')$ for first-order predicates '*expr*' over V, V', where s provides the interpretation of unprimed and s' the interpretation of primed variables in '*expr*'.

– A snapshot $s \in \Sigma$ is called *initial*, iff $[\![\Theta]\!](s) = true$.

– Let $s, s' \in \Sigma$ be snapshots of S. Snapshot s' is called S*-successor* of s, iff $[\![\rho]\!](s, s') = true$.

– A *computation*, or *run*, of S is an infinite sequence of snapshots $r = s_0\, s_1\, s_2 \ldots$, satisfying the following requirements:
  - *Initiation:* s_0 is initial.
    for each $j \in \mathbb{N}_0$.

– The set of all computations of S is denoted as $runs(S)$.

Thus we elaborated in this semantics the way the snapshots can evolve, defining for each of the possible cases a transition predicate. Finally, we defined the predicate characterizing initial snapshots and collect all pieces of the transition relation into a full predicative definition of the transition relation, leading to a definition of the symbolic transition system associated with the *krtUML* model.

5.1 Example interface between state and activity machines for process monitoring

Objects are needed to express various process related information, like all kinds of artifacts, persons and software. Objects are created and live until they are destroyed. In general they should have a system-wide scope, but it must be possible to put them into local scopes (e.g., an activity). Furthermore they have additional attributes and methods for example to support product locks for write access. Events are needed because it is not always possible to distinguish two product according to their state (for example imagine refactoring activities) or to model timers (and thus timing events). In contrast to objects events live exactly one step after the are send (no fixpoint semantics) and they are always related to a certain element (object, transition).

Central for the whole discussion of software process modeling are objects and their way they are treated inside activity diagrams. As already depicted in figure 1 the first approach is to sketch the artifacts an activity "consumes" and "produces". This picture is complete for tracking purposes as long as we don't want to monitor software development processes. If we want to monitor a software development process than we must be able to react on a certain set of objects that is developed during the execution of an activity and integrate these objects into the picture. But this set of objects can't be specified initially before the process is executed because we don't know how many objects we will have at a certain point of time. We additionally required that tools are able modify this object set. But tools don't "behave" on base of a Petri net based execution semantics. They react on a state chart-based semantics. Thus we need to discuss the possible combination of activity-charts and state charts in relation to what these diagrams should be able to exchange. As already mentioned we need objects as permanent "things" and events as temporally restricted ones to a single step. Regarding an object modeling we made these design choices:

1. There exist object places for all objects $o_1, \ldots, o_n \times \{lock, unlocked\} \times \{a1, \ldots a_m, no - scope\}$ all activities.
2. The object places are connected with the activities that produce and consume the objects.
3. There exist two additional places for each object as object token sources and drains.
4. Each time a creation of object o_i is executed a token is put onto it's object token source.

5. Each time a destruction of object o_i is executed a token is removed from one object token place (that is designated for token removal).
6. Tokens are inside a scope of an activity if the tokens are moved to the designated places of the activities.

Figure 10 shows a simple Petri net invoking a tool a. This figure bases one the activity diagram of figure 1, whereby on "Activity_i+1" a "tool (a)" in invoked — this can be expressed in the activity diagram by an object "a" with stereotype "tool".

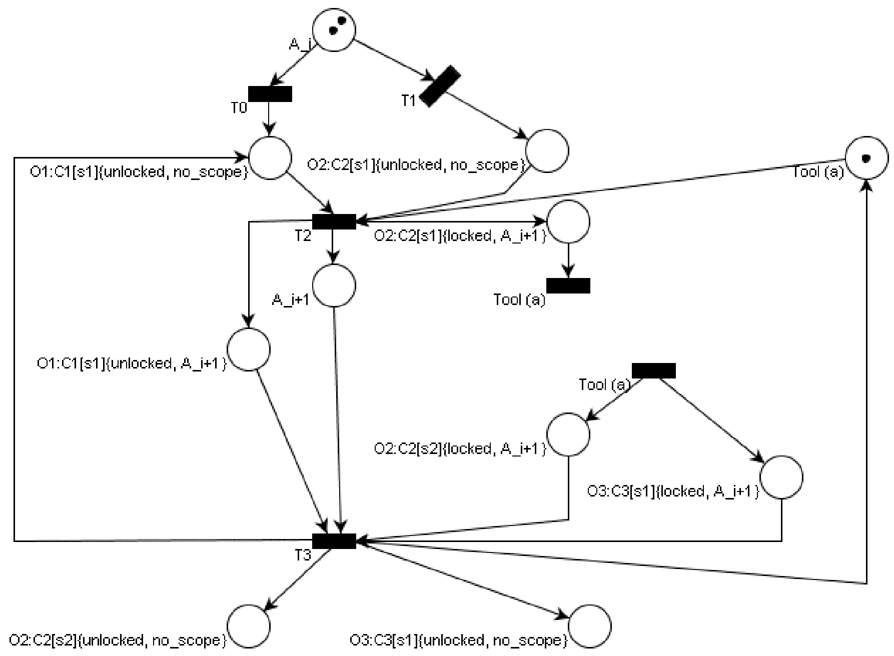

A_i
T0
T1
O1:C1[s1]{unlocked, no_scope}
O2:C2[s1]{unlocked, no_scope}
Tool (a)
T2
O2:C2[s1]{locked, A_i+1 }
A_i+1
Tool (a)
O1:C1[s1]{unlocked, A_i+1 }
Tool (a)
O2:C2[s2]{locked, A_i+1 }
O3:C3[s1]{locked, A_i+1 }
T3
O2:C2[s2]{unlocked, no_scope}
O3:C3[s1]{unlocked, no_scope}

Fig. 10. Simple Petri net semantics including tool

One difference compared to figure 1 is that the object state space is a subset of the Cartesian product we defined in our design choices. Furthermore figure 10 depicts token sources and drains as connection between activity and state machines, illustrated as transition without source place or without destination place. If it is not possible to determine object changes according to their object states then events should be used to inform the activity-machine about ongoing process. Regarding events we made these design choices:

1. There exist event places for all events that can be received.
2. An event starts to "live" one step after it it send (no fix-point semantics).
3. An event lives exactly one step.

Figure 11 shows a Petri net distributing an event. After informing the event source that the event occurred the token source puts five token onto the event distribution place (ed). Then all event consumers (ec1, . . . , ec5) can fire.

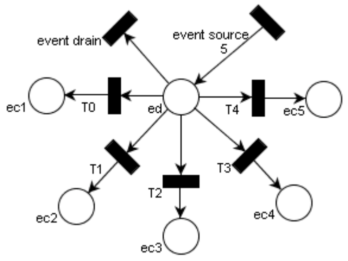


Fig. 11. Event distribution

When distributing an event it is necessary to remove the unused event tokens after one step from the distribution place. This is done by transition t6 in the example. Notice that we have to make three assumptions regarding the firing of transitions compared to place/transition Petri nets. The first is that we have a maximality constraint that says that in each step the set of fired transitions include as many enabled ones as possible. The second is that we have two steps when choosing the set of enabled transitions that will fire. The first one includes all transitions without that ones that have "sink" destinations. The second one includes the other transitions. The third is that event consuming transitions are only allowed to fire once in one step.

To model the diagrams we used Together 6.1 [36] - mainly because of three reasons. It is the only UML tool we know that allows to express activity machines in such an elaborated way (for example Rational Rose [29] v2003 does not support object flows). The second reason was that Together offers a wide range of possible XMI [39] variants in order to exchange models. Unfortunately the exports of version 6.1 were not compliant to the OMG meta-models but it was possible to develop a mapping. With these changes we were able to parse the models into an OFFIS [27] development of a MOF [24] repository.

In order to summarize the first results so far, we developed a syntactical mapping of a process model towards a snapshot of the UML language, the Kernel semantics for the execution of activity machines in relation to state machines and an architecture including a first prototype implementation of a framework to track the execution software development processes.

6 Conclusion and Further Research

The status of our work on defining an approach for a UML profile for software development process modeling was presented. We started with a motivation for such a profile and defined general requirements. Then we briefly discussed why the previous work on UML formalization do not match and explained our approach shortly.

Our next research activities focus on the definition of the precompilation phase and the implementation of the software based process monitoring framework. The precompilation phase describes the unfolding of all used diagram elements into the specified kernel model language. Afterwards this language has to be extended in a direction to be able to express software development process templates. In such a template we have to deal for example with the question what are universal and what existential elements. At the same time we plan to extend our prototype implementation towards a stable and easy to use software system in order to get more "real" world process descriptions. After extending the capabilities of the kernel language with respect to the specification of process templates we have to deal with the question how the framework is adoptable to this requirement. We are going to integrate the "Mediator" component more and more directly into the development tools so that we are directly informed when objects are created, modified or destroyed.

References

1. Object management group. http://www.omg.org. last visited May 2004.
2. J.M. Alvarez, T. Clark, A. Evans, and P. Sammut. An Action Semantics for UML. In *Proc. UML 2001*, 2001. http://www.cs.york.ac.uk/puml/mmf/AlvarezUML2001.pdf.
3. An introduction to workflow management systems. http://ctg.albany.edu/publications/reports/workflowmgmt/workflowmgmt.pdf. last visited May 2004.
4. Verein der anwender des software-entwicklungsstandards der öffentlichen verwaltung (ansstand) e.v. http://www.ansstand.de/alle/2003/ea2003v2.pdf. last visited May 2004.
5. K. Beck. *Extreme Programming explained: embrace change*. Addison-Wesley Professional, Boston, 2000.
6. E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. AMAST 2000*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.
7. E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich, Ph.W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Proceedings*, volume 1912 of *LNCS*, pages 223–241. Springer-Verlag, 2000. DBLP, http://dblp.uni-trier.de.
8. Ralf Buschermöhle and Wilhelm Hasselbring. Taking advantage of the symbiotic relationship between tools and processes to support executable process models.

In Ban Al-Ani, H.R. Arabnia, and Youngsong Mun, editors, *Proceedings of the International Conference on Software-Engineering Research and Practice (SERP 2003)*, volume 1, pages 279–285, Las Vegas, Nevada, USA, June 2003. CSREA Press.

9. W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. In *FMOODS'99 IFIP TC6/WG6.1.* Third International Conference on Formal Methods for Open Object-Based Distributed Systems, 1999.
10. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, number 2852 in LNCS. Springer-Verlag, 2003.
11. Werner Damm and Moshe Cohen. Advanced validation techniques meet complexity challange in embedded software development. *Embedded Systems Journal*, 2001.
12. G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceed. of the 3d International Conference on the UML 2000*, October 2000.
13. A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In *The Unifed Modeling Language: the first international workshop, June 1998*. Springer-Verlag, 1999.
14. R. Hightower and N. Lesiecki. *Java Tools for Extreme Programming, Mastering Open Source Tools, Including Ant, JUnit and Cactus*. Jon Wiley & Sons, Indianapolis, 2001.
15. Dirk Höltje, Nazim H. Madhavji, Tilmann Bruckhaus, and WonKook Hong. Eliciting formal models of software engineering processes. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 28. IBM Press, 1994.
16. H. Hussmann. Loose Semantics for UML, OCL. In *Proceedings 6th World Conference on Integrated Design and Process Technology (IDPT 2002)*. Society for Design and Process Science, June 2002.
17. http://www.v-modell.iabg.de IABG. Das V-Modell - Entwicklungsstandard für IT-Systeme des Bundes. last visited 06.07.2004.
18. J. A. Keane J. Sa and B. C. Warboys. Software process in a concurrent, formally-based framework. In *IEEE International Conference on Systems, Man and Cybernetics, Beijing, China*. IEEE Computer Society Press, January 1996. ISBN 0780332806.
19. Maria Letizia Jaccheri, Gian Pietro Picco, and Patricia Lago. Eliciting software process models with the "e3" language. *ACM Trans. Softw. Eng. Methodol.*, 7(4):368–410, 1998.
20. Javacc hompage. http://www.dev.java.net/. last visited: 07.04.2003.
21. P. Kruchten. *The Rational Unified Process - An Introduction*. Addison-Wesley Professional, Boston, 2000.
22. G. Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statcharts. In *Proceed. of the 3d International Conference on the UML 2000, University of York*, October 2000.
23. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
24. MOF - Meta Object Facility v1.4 (complete specification). http://www.omg.org/cgi-bin/doc?formal/2002-04-03. last visited 28.02.2003.

25. Object Management Group. *UML 1.4 with Action Semantics, Final Adopted Specification, ptc/02-01-09*, January 2002.
Available from http://www.kc.com/as_site/home.html.
26. Object Management Group. *Unified Modeling Language: Superstructure, v.2.0, Final Adopted Specification ptc/03-08-02*, August 2003. Available from http://www.omg.org/cgi-bin/doc?ptc/03-08-02.
27. Oldenburger forschungs- und entwicklungsinstitut für informatik-werkzeuge und -systeme (offis) e.v. http://www.offis.de. last visited May 2004.
28. G. Övergaard and K. Palmkvist. A Formal Approach to Use Cases and Their Relationships. In *UML 1998*, 1998.
29. Rational rose software. http://www-306.ibm.com/software/rational. last visited May 2004.
30. Ribo, J.M. and Franch X. Building expressive and flexible process models using a uml-based approach. *Software Process Technology - 8th European Workshop, EWSPT 2001*, pages 152–173, 2001.
31. Rik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, 2002.
32. Rational Unified Process (whitepapers, process workbench). http://www.rational.com/products/rup/index.jsp?SMSESSION=NO. last visited: 20.03.2003.
33. Society of automotive engineers (sae)hompage. http://www.sae.org/. last visited: 07.04.2003.
34. Software Process Engineering Metamodel (complete specification). http://www.omg.org/cgi-bin/doc?formal/2002-11-14. last visited: 28.02.2003.
35. Störrle. Describing process patterns with uml (position paper). *Software Process Technology - 8th European Workshop, EWSPT 2001*, pages 173–182, 2001.
36. Together control center 6.1. http://www.borland.com/together. last visited May 2004.
37. UML - Unified Modeling Language v1.5 (complete specification). http://www.omg.org/cgi-bin/doc?formal/03-03-01. last visited May 2004.
38. G. (Hrsg.) Versteegen. *Das V-Modell in der Praxis - Grundlagen, Erfahrungen, Werkzeuge*. Dpunkt Verlag, Heidelberg, 2000.
39. XMI - XML Metadata Interchange Specification v1.2 (complete specification). http://www.omg.org/cgi-bin/doc?formal/2002-01-01. last visited May 2004.