Taking advantage of the symbiotic relationship between tools and processes to support executable process models

Ralf Buschermöhle
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

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

***Abstract*— An approach for tight coupling of process models and software development tools — with the metaphor of component-based software development environments — supporting "eXecutable Process Models" (XPM) is presented. In this paper, we focus on the direction from the components of a software development environment towards the process models in order to automatically acquire process model information solving various drawbacks compared to a manual acquisition of this information. Requirements on a process modeling language for using this information are discussed.**

I. Introduction

Various computer-based system development process models exist to structure activities and resources for efficient development of software and hardware. They range from agile processes [Bec00] as a set of "best practices," customizable frameworks such as the Rational Unified Process [Kru00] towards very detailed process descriptions such as the the V-Modell [DHM98]. They all have in common that tools are required to support their activities [RUP], [HL01], [Ver00]. 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, the tools are able to serve as bridges between the process models and the actual processes. This 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 [DC01].

The relationship between process models and tools is relevant in both directions. On one hand, process models have an impact on the tools — this is often referred to as "enactment" [Fug00] usually expressed in specific process supporting systems [ACF97]. On the other hand, the tools have an impact on the process models. The main concern for this direction is the consideration 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 of how the software environment that is used in a development process can be integrated into a process model language (PML) meeting these requirements. In this paper we concentrate on the direction from the software environment towards a process model.

Section II starts with a description of the elements a software environment consists of, followed by a discussion of our XPM language approach in section III which offers flexibility for extensions. Then we describe the mapping of the software development environment towards the XPM language in section IV. We conclude in section V with a discussion of additionally required language elements.

II. Describing the software environment

In the following we consider tools as components according to the (informal) component definition of Szyperski [Szy98]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition."

The main motivation for the consideration of tools as components is the possibility to decompose and describe the *entire* software development environment of a process executer in a modular way. This approach could be extended to components of the operating system, repositories etc. As connecting "glue" between the components an appropriate component model implementation (e.g. the Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJBs), or Microsoft .NET) is assumed. This way, the "local" environment of a process executer can be extended towards a distributed environment. This becomes even more important when the

whole development team is distributed. For a more general account of dealing with software as a set of components refer to [Has02], [HC01].

A. Component behavior

In order to describe the component's relevant internal static and dynamic elements, it is necessary to take "a view inside." This can be achieved on the origin component description(s) - in the following referred as "OCDs" (similar to source code, data base schemata, deployment descriptors, etc). But the use of OCDs in this context has some disadvantages, whereby the major problem is the vast variety of different OCDs — what makes it necessary to know each OCD for a tight integration — and the low abstraction level, e.g. due to technical details that are not interesting for process models. The latter problem could be solved if we were able to substitute all OCDs by some model that provides just the right abstraction level.

Even more if it is possible to express these models as instance of a (standard) modeling language it would be possible to cover the variety of OCDs. There exist a variety of modeling languages for different purposes but from a pragmatic point of view the Unified Modeling Language (UML [UMLa]) is a promising candidate for this purpose since it addresses all required concerns [HC01], provides different abstraction layers, and more or less coherent perspectives on the system. In particular, it is an accepted industrial standard and many modeling tools are available. These tools usually offer the opportunity to translate their models into OCDs (code generation) and vice versa (re-engineering).

An additional benefit from using UML models instead of OCDs is the possibility for a (virtual) composition of software development environments. This can be a basis for adding new components that support (additional) process activities of the chosen development process. Specific advantages of specifying models are:

- Meanwhile, models become part of the deliverables between customer and software house.
- The recent OMG efforts to allow for executable UML models to decrease the differences between models and OCDs.

The Unified Modeling Language provides several notations for different purposes [RJB99], whereby the core diagrams can be summarized as follows:

- *Use cases* specify actions that a system (part) can perform by interacting with process executers.
- *Class diagrams* specify the static structure in form of classes and relationships among those classes.
- *State charts* specify behavior for objects of individual classes by means of state machines.
- *Message sequence diagrams* specify interaction in (imaginary) scenarios.

Additionally, there exist action languages (in the new UML 1.5 [UMLa]) and constraint languages [WK99], sometimes used for executable models.

In the following we concentrate on statecharts, augmented with an action language that describe dynamic elements, and on class diagrams that describe static structure. The remaining UML diagrams, such as activity, collaboration, component, or deployment diagrams are not needed for our purposes of executable models [Sta02], [MB02]. Therefore, we assume an existing statechart-based description of the component. In order to connect component descriptions to process models, it is necessary to identify the places where actions can take place which are able to modify objects, as these are the interesting places to track or control the behavior of components. In a statechart, this includes state transitions and {entry, internal, exit} "actions" of a state.

Our vocabulary is based on the action language approach of [UMLa]:

Object manipulations include object creation, destruction and attribute assignment. More advanced constructs — such as multiple assignments or the combination of object creation and attribute assignment — are not included, since they can be expressed by a set of the basic constructs.

```
Object.Attribute = Value;          // assignment

delete Object;                     // deletion
```

Control structures include conditionals (if-then-else), multi-way branching, and state transition conditions. The latter two are boolean expressions that can be annotated at transitions.

```
if ( BooleanExpression ) { /* some action */ }

[ elseif ( BooleanExpression ) /* multi-way */
       { */ some additional action */ } ];
```

Object selections include selection and storage of class instances, i.e. objects

```
select ObjectSet from Instances of Class
       where Attribute == Value;
```

Object linking include construction, navigation and destruction of links between objects.

```
relate ObjectSource to ObjectDest /* construction */
       across RelationName ClientRole SupplierRole;

select one Object related by /* navigation */
ObjectSource -> ObjectDest[RelationName];

unrelate ObjectSource from /* destruction */
ObjectDrain across RelationName;
```

Messaging actions include notification of events (asynchronous) and requests (synchronous).

```
Object.Attribute = /* operation call */
OperationName(Parameters) on Object;

generate EventName /* event sending */
(Parameters) to Object;
```

B. Component interaction

After the short introduction to the specification of (internal) behavior of components in the previous subsection we discuss the (external) representation of component interaction. For component integration it is necessary to specify how the components are connected. For this purpose we use a component

model implementation. An appropriate candidate is the Common Object Request Broker Architecture (CORBA [Bol01]). Several implementations for a broad number of platforms exist (operating systems, programming languages etc.) and also a UML Profile providing CORBA extensions [UMLb]. The interface vocabulary of CORBA is based on the so called “Interface Definition Language” [Int]. The interface specification has to reflect all required tracking and control functions for the relevant actions. Such “component contracts” that base on component protocols [Nie93] can provide useful information about the predicted properties of a component [RS02] including the best (re-)configuration time to avoid unexpected behaviour [MMW03].

III. **Choosing an XPM Frame**

In this section, we discuss the requirements on a PML and possible instances of such a PML. For this purpose a flexible approach of a PML is needed, which is extensible. Typical PML representations focus on specific application areas or they are not extensible [SK], [Rum99], [BLRV]. Furthermore, we need the above mentioned connection from process elements to components. For several reasons the UML is an interesting candidate for this purpose:

- Firstly, a lot of successful approaches to express process models in UML exist [JSW]. The suitability of UML as PML was as well confirmed in the project ViSEK (Virtual Software Engineering Competence Center) [ViS], in which the so called V-Model [Ver00] — the standard process model in Germany that is mandatory in projects with governmental institutions and which is widely used in the area of safety critical systems — was represented in the UML.
- Secondly, this representation was extended through various standards of the application areas automobile, railway and avionics and optimized with respect to the application of formal methods. For this purpose we used class diagrams to describe artifacts together with their relationships, and activity diagrams to set instances of these classes into the context of the process flow.

As an example, figure 1 displays the complete artifacts of phase SE2-5 of the V-Modell and their relationships. A zoomed extract of figure 1 is displayed in figure 2, whereby the classes `C_class_model` and `C_state_model` are parts of the product `C_interface_description` which has to be developed in phase SE2-5. These two classes are artifacts that are developed through the application of the methods KOM (class and object modeling) and Zusto (state modeling) — the numbers in the braces indicate where to find the method descriptions. Additionally there exists a class `C_class` as connector between `C_class_model` and `C_state_model` to express additional constraints. Figure 3 displays an activity diagram to put the products of figure 1 into the context of the activities of phase SE2-5. The zoomed extract of figure 3 that is displayed in figure 4 shows instances of figure 1 between a part of the activities of phase SE2-5.

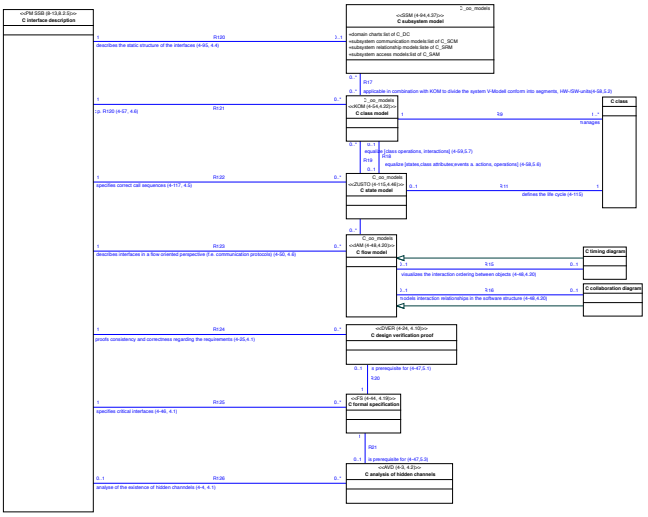

Fig. 1. Products in Phase SE2-5

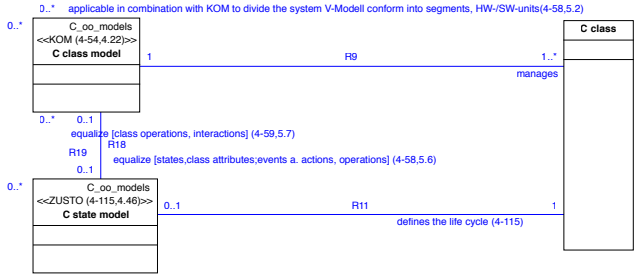

Fig. 2. Products SE2-5 (clip)

The evaluation in our project showed that we were able to express the V-Modell in UML using activity and class diagrams, whereby we used stereotypes and constraints as UML extensions. For a detailed description of the modeling refer to www.visek.de. An interesting aspect in this context is that the UML specification and even more the new Model Driven Architecture (MDA [MDA]) approach of the Object Management Group (OMG [OMG]) offer a wide range of features to extend these descriptions accomplishing all necessary requirements that might come. The UML offers the possibility to extend it via stereotypes, tagged values and constraints for an identified UML subset within a so called UML Profile. If it is not possible to express the language in such a profile, the MDA approach offers the opportunity for the definition of an instance of the so called Meta Object Facility (MOF [MOF]) layer — the meta-meta model of the other meta-models. Such an instance could conceive several usable UML elements and provide some additional necessary constructs in order to obtain as much conformance as possible. This is useful with respect to the necessary tools for a partial transformation of the process model description into “real” software. Furthermore, several research groups work on the formalization of UML diagrams [The] to provide the executability for simulation

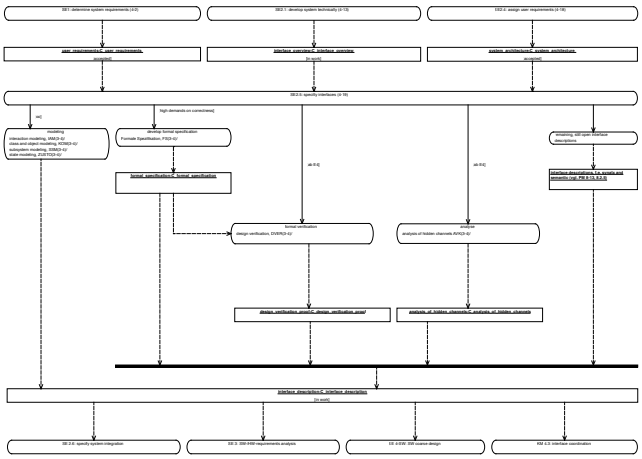

Fig. 3. Activity SE2-5

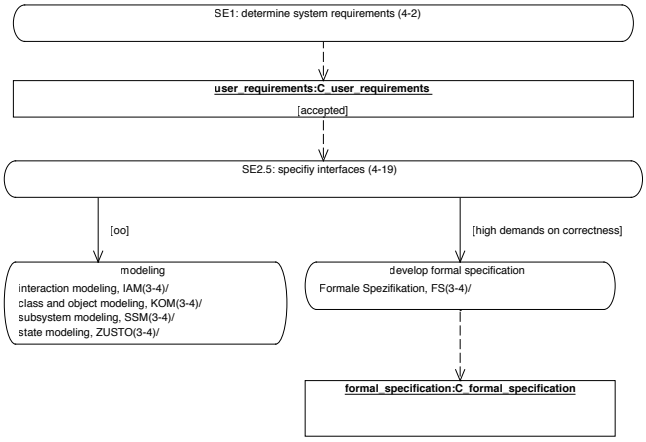

Fig. 4. Activity SE2-5 (clip)

and formal verification purposes. Consequently, we focus in the following on the Unified Modeling Language as a basis for our process modeling language.

IV. From software environments process models

Usually, process activities are captured information manually. This has several drawbacks, like modeling errors due to the manual acquisition, incomplete data, and high costs. Because of the high number of activities that can be supported through software tools in a system development process, it should be possible to capture as much process information as possible automatically. Additionally, it becomes possible to capture this information on a much more fine granularity than it would be possible in a manually manner.

A. Specifying component behaviour

All possible operations depend on the employed action language; thus, capturing can be realized by adding appropriate operation calls which describe the actions together with the applied changes and the involved objects. Exemplarily, we present our extensions of the action language constructs introduced in Section II-A.

To describe *object manipulations* we need to track the changed input and output objects resp. newly created and destroyed objects:

```
/* creation */
Object = create Class;
call track_object_creation( UID, ObjectRef );

/* assignment */
call track_object_manipulation_before( UID );
Object.Attribute = NewValue;     // assignment
call track_object_manipulation_after( UID,
        Attribute, NewValue );

/* deletion */
delete Object;
call track_object_deletion( UID );
```

First, each objects gets a global Unified Identifier (UID) to identify the objects in the action flow. Then, each assignment is tracked until the object is deleted and the UID released. Figure 5 shows a possible activity diagram of tracked object

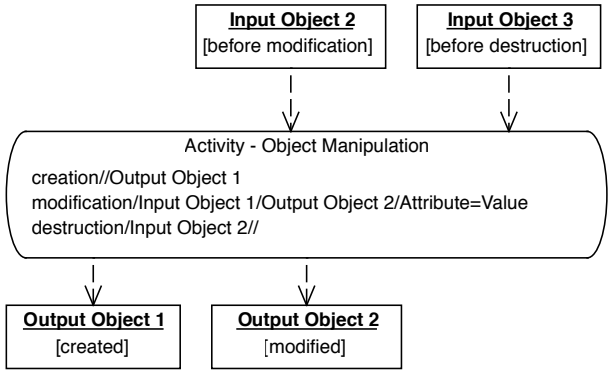

Fig. 5. Object Manipulations

manipulations. The names of the objects correspond to their unique identifiers and the states indicate whether the manipulation has already been executed. The other identifiers of all activity diagrams examples are taken from the corresponding tracking calls. The internal actions of ”Activity Object Manipulation” are an object creation, modification and destruction, whereby the relationships to the referred objects are seperately annotated.

Control structures must be tracked since they are able to influence the rest of the enriched constructs.

```
call track_control_sequence_before( UID,
        BooleanExpression );
if ( BooleanExpression ) {

call track_control_sequence_taken( UID );
/* some action */

}
call track_control_sequence_after( UID );
```

```
/* analog fr multi-way decisions */
[ elseif ( BooleanExpression )
        { */ some additional action */ } ];
```

Similarly, the control structures obtain unique identifiers to capture the area they might influence. Furthermore, it is relevant on which boolean expression the decision is based. Figure 6 shows a possible activity diagram of tracked control

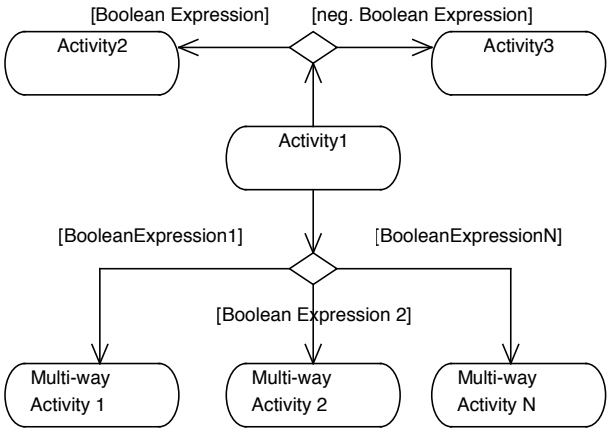

Fig. 6. Control Structures

structures which are mapped into according decisions.

Since the *selection of objects* is just a special case of an object manipulation (creation), we use the same tracking method with information about the selection criterion:

```
select ObjectSet from Instances of Class
        where Attribute == Value;
call track_object_selection( UID, ObjectRef,
        Attribute, Value );
```

The other operations done on the set, like assignments and destructions, are tracked like ”normal” object manipulations.

The *linkage of objects* is relevant to track relevant object structures and their dynamic dependencies

```
relate ObjectSource to ObjectDest /* construction */
        across RelationName ClientRole SupplierRole;
// assuming the relation name is unique
call track_relation_creation( UID, RelationName );

select one Object related by /* navigation */
ObjectSource -> ObjectDest[RelationName];
call track_object_navigation ( UID, ObjectDest );

unrelate ObjectSource from /* destruction */
ObjectDrain across RelationName;
call track_relation_destruction( UID );
```

Figure 7 shows a possible activity diagram of tracked object linkages. Please notice that the relations { Relation1, Relation2} are not a UML standard [UMLa] complient annotation.

Events are necessary for different purposes, for instance to be informed when the execution of another statechart continues that was blocked for an event:

```
call track_operation_call( UID, OperationName,
Parameters, ObjectDest );
```

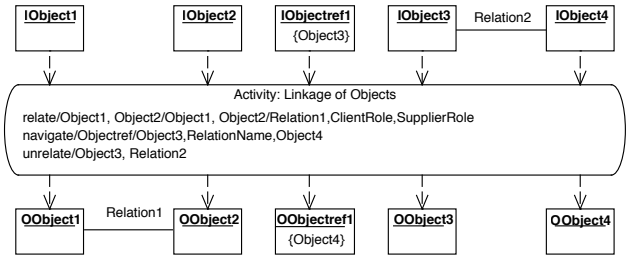

Fig. 7. Linkage of Objects

```
Object.Attribute = /* operation call */
OperationName( Parameters ) on Object;

call track_event_sending( UID, EventName,
Parameters, ObjectDest );
generate EventName /* event sending */
( Parameters ) to Object;
```

The tracking impacts of operations and events can be compared to object manipulation and/or control structure tracking.

The enrichment of statecharts can, in principle, be divided into the classes of direct and indirect object actions. Direct object actions have a direct impact on an object set, like object manipulations, object selections, and object relations. Conversely, the class of indirect object actions consists of all control structures, which have an impact on the class of the direct object manipulations, like control structures and guarded conditions of transitions.

We assumed a complete statechart as specification of the component to be traced. Since such models can become very detailed and, thus, for the derived process information it would be useful to provide additional techniques to aggregate and/or focus only on some specific concerns of statecharts and, this way, constrain the process information. These issues become even more relevant when the enrichment is done automatically. This brings us to the third class of modeling constructs which allow for abstraction, such as high-level composed states and events. It is possible to “cut” a statechart into hierarchical abstraction levels. Such an abstraction could be complemented by additional filter techniques allowing to focus on relevant attributes, objects neglecting internal model elements.

So far we focused on the internals of a component to capture process information. In the next section we extend this towards the interaction with other components that are needed to manage the information.

B. Specifying component interaction

Similar to the specification of operation calls we specify interfaces based in the Interface Definition Language [Int] that are able to receive them. Exemplarily, for object manipulations this looks like:

```
interface tracking {

        // additional interface information,
```

```
        // like attributes
        // ...
        exception UIDUnavailable{ };
        // ...

        //  tracking object manipulations
        track_object_creation( out UID uid,
                ObjectRef or)
                raises ( UIDUnavailable );
        track_object_manipulation_before(
                in UID uid );
        track_object_manipulation_after(
                in UID uid, Attribute ar,
                NewValue nv);
        track_object_deletion( in UID uid );

}
```

The interface specification should be based on the specification of the component's behavior.

V. **XPM Requirements**

To answer the question which elements a PML must contain to achieve the requirements, we describe in the following the procedure of automatic acquisition of process information.

The first step is the collection of multiple process executions in a certain time interval, whereby the goal is to collect as much process information as possible. This includes in addition to the already mentioned activities additional data, such as the required time. Therefore an adequate PML approach must be able to describe all artifacts and activities that are developed and realized with the components.

This data is the base for aggregations and classifications, whereby the aggregations of artifacts and activities. This will be done on the corresponding meta layers. On each layer it is possible to use classifications with an appropriate process model vocabulary which will result in at least one process modeling language and a top layer process which is able to describe all captured process executions. For this purpose it is necessary to apply the process modeling language across an "architecture" of meta modeling languages.

Afterwards different language elements should be fixable to define "the process" or to describe the requirements and the derived development process of a new project. For this purpose the language has to provide constructs for specifying what is mandatory and what is optional. Furthermore, this development process model should be understood as a template rather than as restricting description. It allows the integration of new components even when the project has already started.

Derived from this description the components are composable to support the development process (enactment), whereby the interactions and data structures should describe process model activities and artifacts which are instances from the new process model. Therefore, the PML needs constructs to support the enactment of the process, which includes the connection of functional components to support the process flow. An example: "if all unit tests were successfully executed, then try to integrate the different components".

Additionally, the PML should be based on a formal notation in order to use it directly as "enactment" environment or for verification purposes.

VI. **Summary and conclusion**

In this paper we propose an approach for automatic retrieval of process information through the software development environment of the process executers. This is achieved by enriching statecharts through corresponding CORBA calls which have to be specified in their interfaces. Statecharts are employed for representation of components of the software environment. The information collected afterwards provides the basis to discuss a possible process modeling language approach which is able to involve and distribute this information.

REFERENCES

[ACF97] Vincenzo Ambriola, Reidar Conradi, and Alfonso Fuggetta. Assessing process-centered software engineering environments. *ACM Transactions on Software Engineering and Methodology*, 6(3):283–328, 1997.

[Bec00] K. Beck. *Extreme Programming explained: embrace change*. Addison-Wesley Professional, Boston, 2000.

[BLRV] Alfred Bröckers, Christoper M. Lott, H. Dieter Rombach, and Martin Verlage. Mvp-l language report version 2.

[Bol01] F. Bolton. *Pure CORBA: A Code-Intensive Premium Reference*. Sams, Indianapolis, 2001.

[DC01] Werner Damm and Moshe Cohen. Advanced validation techniques meet complexity challange in embedded software development. *Embedded Systems Journal*, 2001.

[DHM98] W. Dörschel, W. Heuser, and R. (Hrsg.) Midderhoff. *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97*. Oldenbourg Verlag, München, 1998.

[Fug00] Alfonso Fuggetta. Software process: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 25–34. ACM Press, 2000.

[Has02] W. Hasselbring. Component-based software engineering. *S.K. Chang (ed.): Handbook of Software Engineering and Knowledge Engineering*, 2:289–305, 2002.

[HC01] G.T. Heinemann and W.T. Councill. *Component Based Software Engineering - Putting The Pieces Together*. Addison-Wesley, Boston, 2001.

[HL01] R. Hightower and N. Lesiecki. *Java Tools for Extreme Programming, Mastering Open Source Tools, Including Ant, JUnit and Cactus*. Jon Wiley & Sons, Indianapolis, 2001.

[Int] Interface definition language specification - syntax and semantics chapter, corba 3.0. http://www.omg.org/cgi-bin/doc?formal/02-06-07. last visited: 15.04.2003.

[JSW] Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using uml for software process modeling.

[Kru00] P. Kruchten. *The Rational Unified Process - An Introduction*. Addison-Wesley Professional, Boston, 2000.

[MB02] Mellor, S. J. and Balcer, M. J. *Executable UML - A foundation for model-driven architecture*. Addison-Wesley, Boston, 2002.

[MDA] MDA - Model Driven Architecture. http://www.omg.org/mda. last visited: 16.02.2003.

[MMW03] J. Matevska-Meyer and Hasselbring W. Enabling reconfiguration of component-based systems at runtime. In J. Bosch J. van Gurp, editor, *Proceedings of Workshop on Software Variability Management*, pages 123–125, Groningen, The Netherlands, February 2003. University of Groningen.

[MOF] MOF - Meta Object Facility v1.4 (complete specification). http://www.omg.org/cgi-bin/doc?formal/2002-04-03. last visited 28.02.2003.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.

[OMG] OMG - Object Management Group. http://www.omg.org. last visited: 31.01.2003.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., 1999.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, April 2002.

[Rum99] F. J. Rump. *Geschäftsprozessmanagement auf der Basis ereignisgesteuerter Prozessketten - Formalisierung, Analyse und Ausführung von EPKs*. Teubner Verlag, Wiesbaden, 1999.

[RUP] Rational Unified Process (whitepapers, process workbench). http://www.rational.com/products/rup/index.jsp?SMSESSION=NO. last visited: 20.03.2003.

[SK] Eswar Sivaraman and Manjunath Kamath. On the use of petri nets for business process modeling.

[Sta02] L. Starr. *Executable UML - How to build class models*. Prentice-Hall, Inc., Upper Saddle River, 2002.

[Szy98] C. Szyperski. *Component Software*. Addison-Wesley, Hawlow, England, 1998.

[The] The precise uml group homepage. http://www.puml.org. last visited: 01.04.2003.

[UMLa] UML - Unified Modeling Language v1.5 (complete specification). http://www.omg.org/cgi-bin/doc?formal/03-03-01. last visited 30.03.2003.

[UMLb] UML Profile CORBA v1.0 (complete specification). http://www.omg.org/cgi-bin/doc?formal/02-04-01. last visited 27.02.2003.

[Ver00] G. (Hrsg.) Versteegen. *Das V-Modell in der Praxis - Grundlagen, Erfahrungen, Werkzeuge*. Dpunkt Verlag, Heidelberg, 2000.

[ViS] ViSEK - Virtuelles Software Engineering Kompetenzzentrum). www.visek.de. last visited 05.05.2003.

[WK99] J. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Professional, Boston, 1999.