COMPONENT-BASED SOFTWARE DEVELOPMENT - A PRACTITIONER'S VIEW

Alexander Bilke*, Olaf Klischat*, E. Ulrich Kriegel*and Rainer Rosenmüller*
*Fraunhofer ISST
Mollstrasse 1
D-10178 Berlin
Germany
ulrich.kriegel@isst.fhg.de

Abstract: *Software components are a useful abstraction to manage software systems during their whole lifecycle from early analysis to maintenance. Components can be combined in order to build more complex components or "composed software systems". However, components do not exist independently of component platforms which provide them with a runtime environment. Therefore, developing components means developing components in the context of a dedicated component platform and in many cases in the context of a supporting component framework too. However, there is a rapidly growing number of modern technology platforms which have to coexist with mature but legacy technology. To cope with that challenge the Model-Driven Architecture (MDA™) paradigm was created. The idea is to use a hierarchy of platform-independent models of applications and application domains. Transformation rules allow the generation of platform-dependent code from models. That might work well for newly built systems, but how to include legacy systems or off-the-shelf components in this paradigm? We will discuss a much simpler approach - a platform-independent markup-language for components and a set of tools to support classical round-trip engineering.*

1. Introduction

During the last thirty years there have been a lot of discussions about how to cope with the increasing complexity of software systems and infrastructures. Although it is not a "silver bullet", Component-Based Software Engineering (CBSE) is best practice which helps to reduce complexity. It is primarily concerned with

- developing software from pre-produced parts,
- the ability to reuse those parts in other applications and contexts

- easy maintenance and customization of parts to produce new functions and features.

In this context a component is an independent unit with a specific task or functionality. Components can be combined in order to build more complex components or "composed software systems".

One property of a software component is that it can be used solely on the basis of its published interfaces and without knowledge of implementation details.

Unfortunately, to build software from components, the dichotomy of component models and component platforms has to be taken into account. A component platform separates components from the underlying operating system and hardware, enables the communication among components and provides non-functional services for them. A component platform implements dedicated protocols and defines contracts, e.g. in form of callback interfaces which components have to implement. Those interfaces enable a platform to manage the component's lifecycle, i.e. there are no universal components running on any possible component platform. Components are always developed with respect to a particular component platform. Most component platforms provide support for general non-functional services like load balancing, transaction, security, and persistence. And modern object-oriented platforms like e.g. Enterprise JavaBeans (EJB, 2001) allow developers to use these services declaratively instead of invoking them programmatically. Based on the declared usage of a service, the implementation of the component platform generates the corresponding calls automatically. However, there are applications requiring domain-specific services, e.g. writing special history files or the ability to clientele processing in finance. Until now, commercial component platforms cannot be extended to provide such domain specific services, even though there are first ideas how to do that[1]. So these services have to be provided by a non-standard supporting framework on which the components additionally depend. As a result, their portability is reduced which in turn prevents them from being reused. There is a large number of component platforms ranging from well-known mainframe transaction monitors to object-oriented platforms like COM+, CORBA and Enterprise JavaBeans. On a very high level of granularity even ERP systems like SAP can be considered component platforms with a very limited number of parameterizable components.

As a brief conclusion, today the idea of building software systems on the basis of universal off-the-shelf components (COTS) cannot be realized due to the technological restrictions given above. Component-based software engineering (CBSE) is possible only in the context of a component platform and, where appropriate, of a domain-specific supporting framework. There are blueprints for platform-specific component-based enterprise architectures, e.g. the Java Enterprise Architecture (see e.g. (J2E, 2001)), which can be used to develop new component-based software systems. However, it becomes a real challenge if legacy systems have to be encapsulated as components and included in an application or if components have to run on different types of component platforms. That challenge becomes much stronger if large heterogeneous software infrastructures must be adapted to new or changing requirements. The reasons for such requirements are manifold and range from modifications in the business model or the introduction of new technologies to the reconfiguration of the software infrastructure of two companies due to a merger. In the following we will discuss two approaches to tackle this problem, the "Model Driven Architecture (MDA^{TM})" initiative of the Object Management Group (OMG) (Wasciewicz, 2002), (Soley, 2000) and a more lightweight approach, elaborated in the context of the research project "Continuous Software Engineering: Continuous Engineering for Evolutionary Information- and Communication Infrastructures (CSE project)"[2].

2. The Model-Driven Architecture

It's a well-established practice to use models and formal specifications throughout the whole software development process from the inception up to the implementation and maintenance phases. The main aims

[1] In the context of the CSE project (see below) and as diploma theses there are ongoing investigations on how to extend Enterprise JavaBeans containers.

[2] This work was supported by the German Federal Ministry of Education and Research under grant 01IS 901.

of MDA are to overcome the problems resulting from the increasing number of component models and platforms (Wasciewicz, 2002). The OMG proposes a new and consistent way to specify and build software systems. The specification is based on the enhanced and modified Unified Modeling Language (UML 2.0) (UML, 2002). A hierarchy of platform-independent (PIM) and platform-specific models (PSM) combined with their metadata and a strictly specified UML semantic shall give full lifecycle support for a complete software system. Based on these models all artifacts necessary for building an executable software system should be generated automatically via a sequence of model transformations. Since the PIMs and PSMs serve as a very high-level programming language, it is stated that there is no need for an iterative and incremental software development process (Björkander, 2002). The software development process modeled in UML2.0 corresponds to the well-known waterfall model (Royce, 1970). Development shall be done only on the basis of these models. Since the standardization process for UML 2.0 is not yet finished and the specification of transformations is still under development, MDA in its platform-bridging elegance is not yet available. Nevertheless, there are already tools supporting certain parts of the MDA process, e.g. the execution of models[3] or code-generation for J2EE-compliant platforms[4]. It will be more or less a question of time that MDA becomes a reality. However, one should not underestimate the expenditure connected with introduction or use of MDA. MDA cannot coexist with other software development processes which are at least partially code-centric. So MDA requires the replacement of established processes for implementation and maintenance. When maintaining legacy systems with MDA, one cannot neglect the re-engineering effort needed to create the corresponding PIMs and PSMs. We claim that at least for the next few years more lightweight processes and methods integrating established processes and tools will coexist with MDA as development paradigms.

3. The Continuous Software Engineering Paradigm

The demand for continuous preservation of software and their quality during long periods leads to a view on software as long-living infrastructure. Evolution of such infrastructures consists of gradual modification steps over all levels of the software development process depicted in Fig. 1.

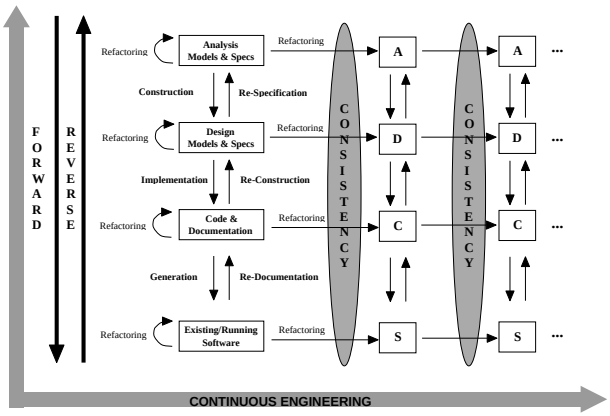

Fig. 1 The Continuous Software Engineering Reference Model

In general, every modification step in analysis, design or implementation will require further consistency-preserving modification steps within each level of the development process.

[3] As an example, state charts can be executed in *Rhapsody* (see http://www.ilogix.com/products/rhapsody) and *Poseidon* (see http://www.gentleware.de.

[4] The tool *ArcStyler* (see http://www.arcstyler.com) generates J2ee-compliant applications.

Up to this point we are conform with the intention of the MDA initiative – support for long-living software infrastructures. However, we have chosen a lightweight technology to reach that goal. Especially for legacy systems, there are often no UML-conformant models and sometimes there are no models at all. If these models cannot be created, MDA methodology cannot be applied. To support the evolution of such systems too, we focused on the idea of "instruction leaflets" which are assigned to each component in the development process. Since components can be used solely on the basis of their published interfaces, in contrast to MDA we do not consider a component's action semantics and the process of its realization. It is only required that there is an implementation of the component's body which can be bound to the component's published interfaces. These leaflets contain relevant metadata[5] concerning a component and will be updated in every evolution step. To represent these leaflets in a platform-independent format, a component markup language called "ComponentML" (Kriegel, 2002) was designed in the CSE project which reflects and generalizes the component structure described in section 4. Storing the leaflets in form of ComponentML expressions in a repository, either tools can be developed or existing commercial tools can be adapted to support the CSE process shown in Fig. 1. Instead of having one MDA-aware tool and one MDA process, a company can use their established processes and tools for building components and the whole process will be assisted by ComponentML-aware tools.

4. The CSE Component Model

In the context of the CSE project it was discussed how continuous software engineering can benefit from component technology. First a common abstract component model was defined to be used in all stages of software development and maintenance (Mann *et al.*, 2000).

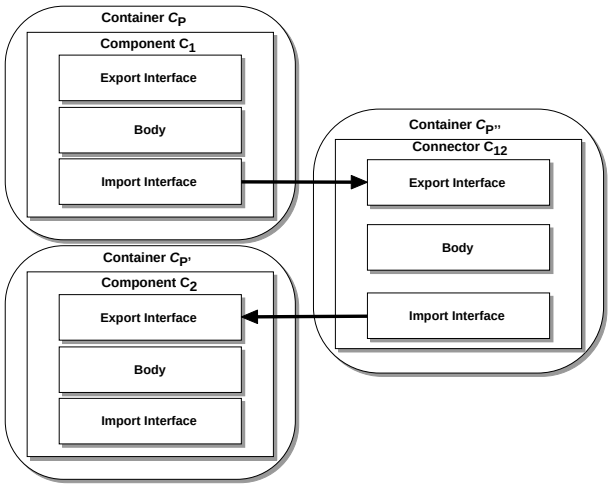

Fig. 2 A System of Components

Fig. 2 gives a generalized view of a component system composed of such components. A component consists of three parts:

- Export interface
  A component's export interface describes the externally visible functionality in form of methods and

[5]There is an obvious need for additional information that goes beyond what is possible using common language constructs like interface definitions or comments. As an example, the Enterprise JavaBeans(EJB, 2001) specification requires bean developers to augment components with metainformation in the XML-based deployment descriptor.

definitions of types a particular component provides to its environment. It can be split up into different sub-interfaces.

- Body
  A component's body implements the exported functionality on the basis of the imported functionality provided by other components and the components runtime environment.

- Import interface
  An import interface describes the requirements of a component to its environment, i.e. the services a component needs to operate properly and to provide the functionality specified in its export interface.

4.1. Component Container and Component Platform

As shown in Fig. 2, components C_i reside on containers C_P which conform to a component platform P. A component platform P specifies protocols which a container has to implement and defines contracts between components and containers, e.g. as callback interfaces which components C_i have to implement in order to enable a container to provide the specified services. A container C_P is a standardized runtime environment that provides specific component services defined in the specification of the component platform P. It separates components from the underlying operating system and hardware, enables communication among components developed for the same platform and provides non-functional services for them, e.g. security and transaction management.

4.2. Connectors

In general, programming languages do not provide means to realize a component's import with the semantics described above. Therefore connections between two components C_i and C_j may be mediated by a connector C_{ij} which is considered a very special kind of component. Its functionality is restricted to mapping the export of a component C_j to the required import of another component C_i. The location of a connector is transparent to the respective client and server components.

5. ComponentML - A Component Markup Language

In order to represent information about components in a platform-independent format, an XML-based markup language called "ComponentML" (Com, 2002) (Kriegel, 2002) which reflects and generalizes the component structure described in section 4., was designed. Storing component metadata in form of ComponentML expressions in a repository, tools can be developed to support continuous software engineering.

The root of a ComponentML document is formed by a *cml-document* element whose structure is shown in fig. 3. To be compatible with different types of component platforms, types, interfaces, components and connectors are specified separately inside a *cml-document* element.

Instances of the specifications shown in fig. 3 have a unique name which can be used for referential purposes.

ComponentML is a specification language under development. To ensure a stable structure and preserve flexibility, every language element can be extended by means of *properties*, i.e. triplets containing a key, a value, and a constraint which is a property by definition. By these means, not yet specified component-related data can be annotated without modifying the structure of ComponentML[6]. For that reason a cml-document as well as any of the contained specifications can be specified further by optional *property* elements.

[6]Later, in a refinement process existing specifications will be searched for recurring universal properties, which then could be promoted to language elements.

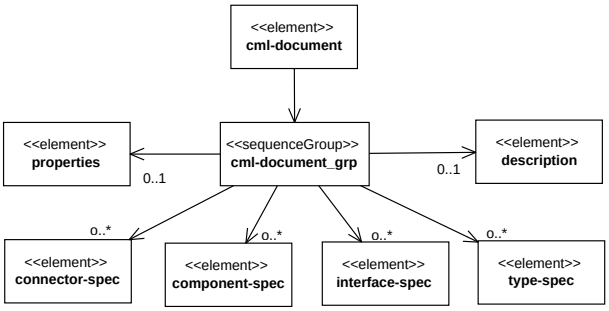

Fig. 3 The Structure of a ComponentML Document

A *type-specification* describes a type which itself can be composed of other types or be a specialization of another type. Types can have named attributes (accessed by reader- and writer-methods) and methods, e.g. to represent implicit attributes. In ComponentML a method is characterized by its signature which consists of

- a name,
- a return value,
- its formal parameters,
- raised signals,

and by additional information, like a description and *properties* (see above). A signal raised by a method can either be an exception or an event, both of which are types. To describe parameterizable types, a rudimentary template mechanism was introduced which has to be further elaborated. The mapping of specified types to platform-dependent types is described via *properties.*

An interface specification has an unique name and contains beside an optional description a set of method-declarations. *Property* elements can be used to specify methods further, e.g. with pre- and post conditions.

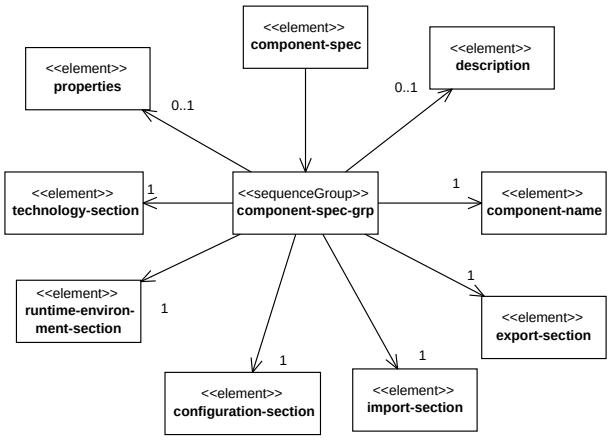

Fig. 4 The Structure of a Component-Specification

Fig. 4 sketches the general structure of a component specification. A component always has a name (*component-name*), which is a unique identifier. An optional *description* can be attached to explain the

purpose of the component. A more detailed description of the component's functionality is given in the other specification sections.

In contrast to other component description languages (see (Medvidovic and Taylor, 2000) for an overview), ComponentML is intended to be used in all stages of a development process and not only for code generation. For that reason, it is discriminated between a component's import and a component's configuration. During the early phases of the development process it is possible to describe services a component will need from other components to operate properly without specifying the required components. The *import-section* serves that purpose. Later, when the collaborating components are known, the import of a component can be described in more detail in the *configuration-section* as a prerequisite for generating platform-dependent glue code for a component application. As already mentioned in section 4., connectors are used to map exports and imports of components, respectively. Therefore the *configuration-section* consists of a sequence of references to component or connector specifications, each of them having an import and an export specification. The *configuration-section* itself is divided into two parts, one describing the simple usage of components and another one, describing the composition of components.

The previous three sections are only concerned with interfaces, which provide a black box view on a component and can be used to trace dependencies between components. Additional dependencies exist between a component and its runtime environment. These are described in the *runtime-environment-section* in terms of

- pairs of keys and values as an abstraction of environment parameters,
- required external resources, e.g. a database driver or a special connector to an external system,
- security rules, and
- transaction requirements.

Interface specifications and runtime requirements are the contract a component implementation has to fulfill. The *technology-section* contains information about one or many such implementations, which can be used to deploy and maintain a certain component. The data provided in this section may contain additional runtime requirements or concretize the abstract specifications enclosed in the other sections. As opposed to MDA, we do not focus on action semantics. Components are considered entities which provide services, but how these services are implemented is of no concern. Instead, we assume that there is an implementation of the component available which is bound to the specification in the technology section.

In the current state of development of ComponentML we only consider functional connectors which are able to map the export of one ore more components to the import of another component. Therefore, a connector specification is a specialization of a component specification. In addition to the latter, it contains a so-called *propagation-section* which describes mappings from methods declared in the connector's exported interfaces to methods declared in the connector's imported interfaces. Based on the discussion in (Smith *et al.*, 1998) we decided not to focus on automatical transformations. Instead, the implementer of the connector's body is responsible for the realization of the correct transformation.

XML was chosen as an external representation language for ComponentML. An XML schema definition exists which can be used to validate component specifications using an existing validating parser, i.e. *Xerces* (XER, 2001). For more detailed information on ComponentML we refer to the ComponentML home page (Com, 2002) where the schema and an UML representation as well as additional papers and examples can be found.

6. A Proposal for a Component Workbench

Fig. 5 conceptualizes a component workbench which supports the development process given in Fig. 1. The

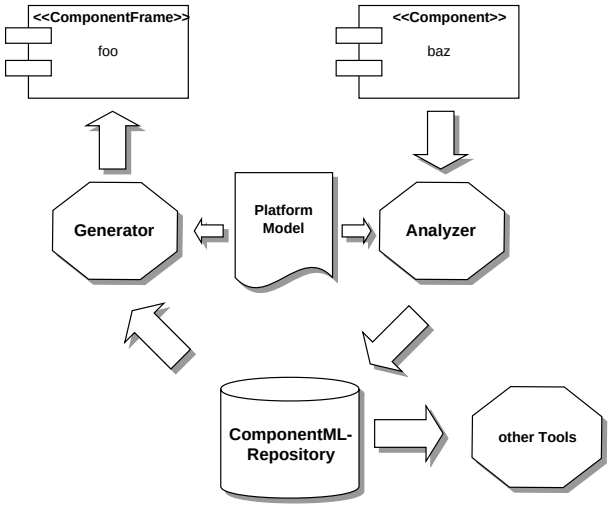

Fig. 5 Conceptualization of ComponentML-based workbench.

central part of the workbench is a repository which manages component specifications in conjunction with other artifacts of the development process . Different tools can access the information stored in that repository in order to assist developers in building and modifying component-based applications. If components are built on the base of a given programming model using defined programming idioms and the realization language supports means for introspection there should be no problem to construct analyzing tools which extract information about real components. As a verification for this, a prototypical analyzer was built which generates output in ComponentML format from existing Enterprise JavaBeans.

6.1. Generating Platform-Dependent Code Frames

To support basic round-trip engineering, a generator which takes a platform-independent component specification and generates as much platform-specific code as possible to assist developers implementing or modifying components is under development.

To facilitate future developments, it is also desirable to factor out parts of the generator's implementation and make them available to other tools. In particular, Java-based representations for the data the generator operates on (i.e. meta-models for CSE components and the platform-specific component model) might prove useful to other tools that deal with CSE components and their platform-specific incarnations in some way or other. Therefore, the generator was divided into several parts, as depicted in Fig. 6

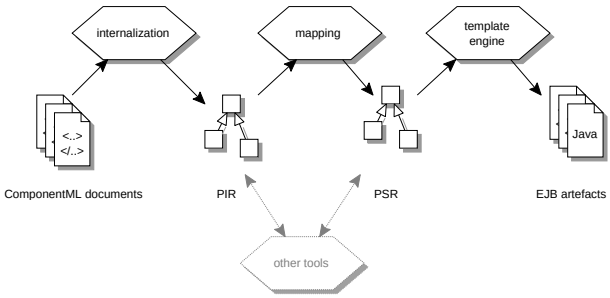

Fig. 6 Implementation of the EJB Generator

The *platform-independent representation* (PIR) contains an "internalized" representation of the essential data model of the ComponentML documents. It serves the purpose of providing the generator with a view on its input that is convenient, consistent, and independent of the external representation (XML). We currently use Castor (Cas, 2002) for creating the PIR; JAXB (JAX, 2002)-compliant XML binding frameworks or similar tools would be suitable, too. As an aside, the PIR could be reused in the future for other tools that also operate on ComponentML documents or on CSE components in general. Tools that manipulate and modify components directly on the model level (as opposed to a platform-specific level) are imaginable as well.

Similar in concept to the PIR, the *platform-specific representation* (PSR) contains Java-based structure descriptions of the essential artefacts that go into the generated platform-specific code. Again, the PSR might be reused in other tools, in this case tools dealing with meta-descriptions of platform-specific components.

The most demanding part of the generator is the mapping code which implements the transition from the ComponentML meta-model to the platform-specific artifacts. It actually defines the semantics of the mapping from the platform-independent ComponentML description to the platform-specific data model.

As a first step we realized a mapping to the Enterprise JavaBeans platform. Each component specification from the PIR gets mapped to one Enterprise JavaBean, consisting of

- EJB interfaces: local, remote, local home, remote home (whatever is applicable),
- Java interfaces for exported types,
- a Java interface for the formal service import,
- interfaces for imported types described in the formal type import,
- a code frame for the bean's implementation class,
- if applicable, connector beans, and
- the deployment descriptor.

In contrast to the MDA approach we generate only platform-dependent glue code, the component's body must be implemented with other means. The generated EJB-specific artifacts ensure that the bean provider can write an implementation for the bean that depends only on the bean's formal import. If connectors were specified for the component, appropriate delegator objects will be created transparently to the bean provider.

Finally, the PSR has to be translated to real source code skeletons and deployment descriptors for use by the bean provider as a foundation for the beans' implementations. This is done using Velocity (Vel, 2001), an open source template-based macro processor written in Java by the Jakarta Project (Jak, 2002). The EJB generator contains a set of velocity macros which perform relatively simple and straightforward transformations. However, we tried to ensure that the PSR only contains the "semantically relevant" pieces of information about the EJBs to be generated, while syntactical transformations go into the Velocity macros. For example, there are no special classes in the PSR representing EJB deployment descriptors. Instead, other information stored in the PSR – e.g. objects describing in detail the existing EJBs and (optionally) the connectors between them – provides the Velocity macros with enough input to generate the deployment descriptors.

Parameterizations for other platforms, e.g. Jini(Kumaran, 2001), are under development. There are even discussions to use ComponentML to produce glue code for COBOL on a mainframe. As soon as these are available, cross-platform generation experiments can be performed to improve the expressiveness of ComponentML.

6.2. Concepts for Additional Tools

Code generation and analysis is not sufficient to support the whole CSE process. There are other concepts from CSE whose realizations as assistant tools are under discussion or already under construction:

- In order to trace dependencies in component-based software systems, the concept of a matrix of propagation was developed (Berthomieu, 2002).
- In general, component-based systems might be distributed across different platforms and different locations. Designing and maintaining distributed systems requires consideration of dependencies and invariants among components. To describe them, the concept of context-based constraints was developed (Bübl, 2002) (Leicher and Bübl, 2002). A prototypical tool which extends UML diagrams with a graphical notation of context-based constraints was implemented at the Technical University Berlin. These diagrams can be externalized as ComponentML specifications.
- When evolving systems, it might happen that different versions of one component coexist temporarily. The effort of configuration management can be reduced if one uses the information represented in ComponentML to identify the newest version of a component whose export is consistent with the required input of another component.

7. Forthcoming Work

The component workbench sketched above gives an initial support for the CSE process. However, it must be improved in two directions. First, ComponentML itself has to be extended. Today the means of describing inter-component dependencies are insufficient for automatic generation of connectors. Even relationships between a component and its container are described only on a very abstract level. The structure of the configuration and runtime environment section has to be further elaborated. Second, additional tools and assistants are needed to especially support evolution steps inside a CSE process. ComponentML can be used to describe invariants which must be preserved during every evolution step. Tools could be implemented to check invariants inside one evolution step of the CSE process and across many of them. This is planned in a forthcoming CSE project.

References

Berthomieu, Caroline (2002). Matrix of propagation - a concept to trace dependencies in component-based software systems. Master's thesis. Technical University Berlin.

Björkander, Morgan (2002). Uml2.0 and how it affects you. In: *OMG Information Day 2002, Munich.*

Bübl, Felix (2002). Introducing context-based constraints. In: *5th Int. Conference Fundamental Approaches to Software Engineering (FASE), Grenoble.*

Cas (2002). The castor project. See http://castor.exolab.org.

Com (2002). The componentml home page. See http://componentml.isst.fhg.de.

EJB (2001). *Enterprise Javabeans^{TM} Specification, Version 2.0.*
See http://java.sun.com/products/ejb.

J2E (2001). *Java^{TM} 2 Platform Enterprise Edition Specification, Version 1.3.*
See http://java.sun.com/j2ee/docs.html.

Jak (2002). The apache jakarta project. See http://jakarta.apache.org.

JAX (2002). Java™ architecture for xml binding (jaxb).
See http://java.sun.com/xml/jaxb/index.html.

Kriegel, E. Ulrich (2002). Componentml - a platform-independent markup language for components. Technical report. Fraunhofer ISST. in preparation.

Kumaran, S. Ilango (2001). *JINI™ Technology: An Overview*. Prentice Hall.

Leicher, Andreas and Felix Bübl (2002). External requirements validation for component-based systems. In: *Conference on Advanced Information Systems Engineering, Toronto.*

Mann, Stefan, Alexander Borusan, Hartmut Ehrig, Martin Grosse-Rhode, Rainer Mackenthun, Asuman Sünbül and Herbert Weber (2000). Towards a component concept for continuous software engineering. Technical Report TR-55/2000. Fraunhofer ISST, Berlin.

Medvidovic, Nenad and Richard N. Taylor (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering.*

Royce, Winston W. (1970). Managing the development of large software systems: Concepts and techniques. IEEE Computer Society Press.

Smith, Glenn, John Gough and Clemens Szypersky (1998). Conciliation: The adaption of independently developed components. In: *Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98)*. pp. 31–38.

Soley, Richard (2000). Model driven architecture. Technical report. Object Management Group. See http://www.omg.org/mda.

UML (2002). The unified modeling language. See http://www.omg.org/uml.

Vel (2001). *Velocity User's Guide*. See http://jakarta.apache.org/velocity.

Wasciewicz, Fred (2002). Model driven architecture: Making architecture last 20 years. In: *OMG Information Day 2002, Munich*.

XER (2001). *The Xerces Java Parser*. See http://xml.apache.org.