The Palladio Component Meta Model: Towards an Engineering Approach to Software Architecture Design

Ralf Reussner, Steffen Becker, and Jens Happe

***Abstract*— The abstract (100-200 Words).**

***Index Terms*— Component, Quality of Service**

I. Introduction

THIS is the introduction [1]. Components are for composition. Divide and Conquer. Provide a hierarchical structure of a software system. COTS failed. But Product-Lines work.

Szyperski's Component: - unit of independent deployment - unit of third party composition - has no (externally) observable state Our focus: Unit of independent deployment, all agree, but nobody knows what it is. - For us, it is more than copying a DLL.

For the Quality of Service prediction of a software architecture, we need a component model with clear semantics and enough information to conduct analyses. Current component models lack these capabilities.

II. Components in Software Engineering and Practice

What is a software component today? UML 2.0 (history, UML 1.0 Component = unit of deployment) - Encapsulate their contents - Are replaceable within its environment - Define behaviour in terms of provided and required interfaces

Corba Component Model J2EE, COM+

SOFA

Software Engineering Processes (RUP, speziell CBSE etc.)

QoS Prediction models (Trivedi, Balsamo, QoSA Reliability Survey Paper)

Cheesman and Daniels

Manuscript received January 20, 2002; revised August 13, 2002.
M. Shell is with the Georgia Institute of Technology.

So, what is a Software Component? o Realisation - Component = class(es) ? - How to bind provided and required interfaces? - How does the component interact with its environment? o Quality of Service - Assembly, deployment, internal structure o Runtime - How does the system look like at runtime? o Substitutability and Interoperability

III. Foundations

The Palladio component meta modell mainly bases on concepts for components presented by the OMG in the UML 2.0 standard and parametric contracts that model the intra-component dependencies of provided and required interfaces. Both concepts are introduced in the following.

A. UML 2.0 Component Concepts

An *assembly connector* connects a required interface of an inner component to a provided interface of another inner component.

Delegation connectors map provided and required interfaces of the composite component to interfaces of its inner structure.

A provides delegation connector maps a provided interface of the composite component to provided interface of an inner component.

A requires delegation connector maps a required interface of an inner component to a required interface of the composite component.

B. Parametric Contracts

Component contracts lift the Design-by-Contract principle of Meyer ("if a client fulfils the precondition of a supplier, the supplier guarantees the postcondition" [4]) from methods to software components. A component guarantees to offer the functionality specified by its provide d interfaces (postcondition) if all required interfaces are offered by its environment. Design-by-Contract cannot only be applied to functional properties, but also to non-functional properties. Adding Quality of Service (QoS) attributes to the contract, a component ensures that it provides a certain QoS if its environment offers the required QoS attributes.

Parametric contracts [7] extend the design by contract principle of software components. The provided interfaces (postcondition) of a component are computed in dependence on the actual interfaces offered by the environment (precondition). The relationship between provided and required interfaces of a software component can be used to compute the influence of external services used by the component on its QoS attributes as well. We distinguish between *basic components* and *composite components* as two different ways of implementing a component.

We state that a component can still be considered a black-box entity if the additional information used by parametric contracts (a) does not expose intellectual property of the component vendor and (b) has not to be understood by human users. So, the additional information and its analysis has to be transparent for the deployer.

A *service effect specification* describes the relationship between provided and required interfaces. It defines the externally visible behaviour of a provided service. This includes the used external services, possible call sequences, and, for example, probabilities of service calls. A service effect specification can be modelled in different ways. It can be a signature list that only contains the called external services. It can model all possible call sequences executed by a provided service. This can be expressed by different means, for example, finite state machines, Petri nets, regular expressions, or any other kind of grammar. In any case, a service effect specification is an abstraction of component source code. It models calls to external services only and neglects the internal component code.

Figure 1 shows a simple service effect specification modelled as a finite state machine. If the method HandleRequest of the DynamicFileProvider is responsible for the incoming request it executes a query on the connected database. In the internal code that is not visible here, it creates the web page and returns the result to the client. If it is not responsible for the incoming request it forwards the request to the next component in the chain of responsibility. This is done by the call of the HandleRequest method of the component connected to the IHttpRequestHandler interface.

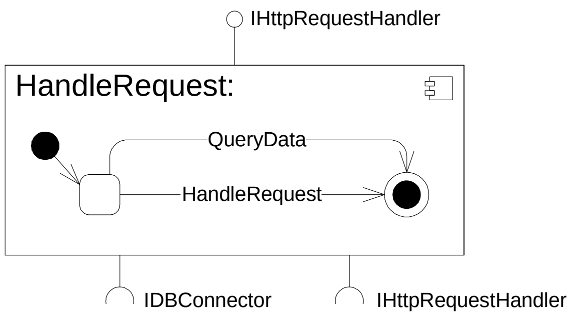

Fig. 1. Service effect specification of the method HandleRequest in the DynamicFileProvider.

IV. Interfaces

In the book by Szyperski et al. the section on components and interfaces starts with

> "'Interfaces are the means by which components connect [8, p. 50].''

For a components interfaces are a key concept as they serve multiple purposes. Consider an interface taken from common programming languages like Java or C#. There, an interface specifies a set of operation signatures, consisting of an operation name, its parameter names and type, return types and exceptions. The corresponding operations can either be called by clients or have to be implemented by servers. In this sense, they can be seen as the contract on which communicating entities agree in both roles, client and server. As with legal contracts, interfaces can exist even when no one actually declared their commitment to them. For example, this is used to define a certain library standard to enable the constructions of clients and servers of these library independently. Thus, in the Palladio Component Meta Model the concept *Interface* exists as first class entity which can be specified independent from components.

Interfaces can be used to build a type system. A central idea of a type system is to allow *subtyping*. The basic idea is that a subtype of a given supertype can be used at every place where an instance of the supertype was expected. Taking design-by-contract [5] into consideration this mean that the subtype's precondition has to be weaker than the supertype's and that the subtype's postcondition has to be stronger than the supertype's [1]. Applying the design-by-contract principle to the parameter types and return types of the signatures in an interface results in a type system which is contra-variant. We prefer this kind of type system for our interface model. A discussion on the advantages and disadvantages of so doing can be found in [5, p. 628ff].

Additionally, we can also apply the type system to perform conformance checks. Conformance checks are useful if one has to decide whether a certain communication between a client and a server is contractually safe, i.e., does not result in a runtime error. For this, a necessary condition is that the required interface has to be a supertype of the provided interface it is bound to.

[1]A formal definition can be found in the book by Meyer [5]

Hence, the analysis of the subtype relationship is essential for modelling a component based architecture and is therefore part of our component model. The information available can be further used to allow semi- and fully automatic adaptation of components, e.g., by using the Adapter design pattern [3].

An even more sophisticated subtype relationship is built by introducing interface protocols in our model. An interface protocol is used to characterise a set of allowed call sequences sent by a client to a server. However the actual meaning of the protocol of the interface depends on its relationship to a specific component. TODO: Soll das wirklich hier hin? Provided und Required Roles/Ports?

However, having the concept of signatures and their protocol, there a still some open questions. For example, Szyperski et al. [8] highlight some subtle problems in component communication resulting from additional concepts which are also important during the interaction of components. The insufficient specification of multi-threaded interaction, reentrance or transactional behaviour are only examples of ongoing research in this field. Additionally, there are no widely established formalisms to specify Quality of Service constraints on a contractual basis. Hence, as there are no settled results in these fields of research yet, our model currently only includes the concepts of signatures and basic protocol information.

V. Interface Relations

VI. Component Type Hierarchy

Introduce types for substitutability of components

Most of today's component meta models do not clearly distinguish between a component and its type. Often, the difference between both concepts is not clarified. Furthermore, the conformance of a component to a type is not defined. So the introduced concepts are very vague and only a few knowledge about the substitutability of a component exists.

> "As such, a component serves as a type whose conformance is defined by these provided and required interfaces [. . .]. One component may therefore be substituted by another only if the two are type conformant [6, p.142]."

Here, the terms *component type* and *conformance* of components and types are mentioned. However, both concepts are not further clarified.

There are mainly two different concepts of component types. Both differ in their interpretation of substitutability.

The first one originates from object oriented software development. There, a class can be substituted by another if it implements at least the same interfaces. Translating this view to the world of software components, a component can be substituted by another if it offers at least the same set of provided interfaces.

The second one does not only consider provided interfaces, but also the required interfaces of a component.

Depending on the task, both concepts have their advantages and disadvantages. Thus, we do not want to limit on one of the concepts.

Both schools specify the required interfaces of a software component. However, the interpretation of a required interface is completely different.

So far, we identified three different meanings of the specified required interfaces of a type:

1) A required interface can be used, but additional interfaces can be added.
2) Only the listed required interfaces can be used.
3) A required interface has to be used in a predefined way and certain call sequences have to be executed.

Required interfaces of the provided-type are not considered explicitly. However, if they are specified, their meaning corresponds to the first case in the list, since, for provided-types, the use of external services is not limited. The second case corresponds to the complete-type. The usage of external services is limited to the required interfaces specified for the component and no further interfaces can be used by the component. The last case expresses additional component requirements. For example, we could specify that all information has to be stored in a database. So, the interface of the database must be called using certain call sequences.

The Palladio Component Meta Model combines the two schools for component types and the different meanings of required interfaces to provide a more flexible type system for components.

Figure **??** shows the component type hierarchie of the Palladio component meta modell. ¡¡provided component-type¿¿ ¡¡complete component-type¿¿ component implementation description ¡¡basic component¿¿ ¡¡composite component¿¿

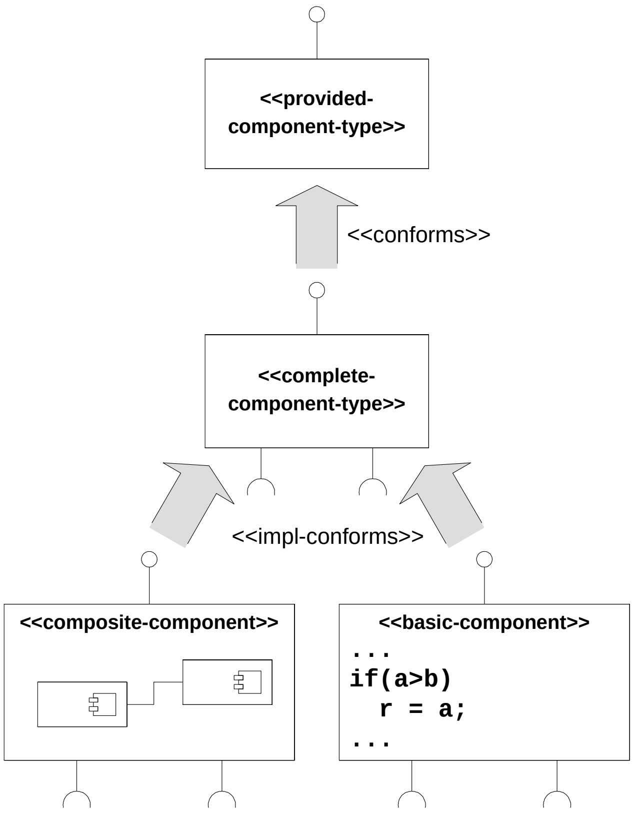

Fig. 2. Different type levels of a software component.

A. Web Server

In the following, we describe the concepts mentioned above in more detail. To do so, we use the architecture of a prototypical web server shown in figure 7. The web server has been implemented in C# for the .Net 1.1 framework. It consists of a set of components that communicate via a clear defined set of interfaces only. – Architecture good for reasoning about software components.

The main part of the web server is realised by three components: Dispatcher, RequestParser, and HttpRequestProcessor. The Dispatcher listens for connections. For each incoming HTTP request, it spawns a new tread and activates the RequestParser. The parser analyses the request and passes the result to the HttpRequestProcessor. The request processor is organised as a chain of responsability [3]. Each of its subcomponents checks whether it can handle the incoming request. If so, it returns the result, otherwise it passes the request to the next compo-

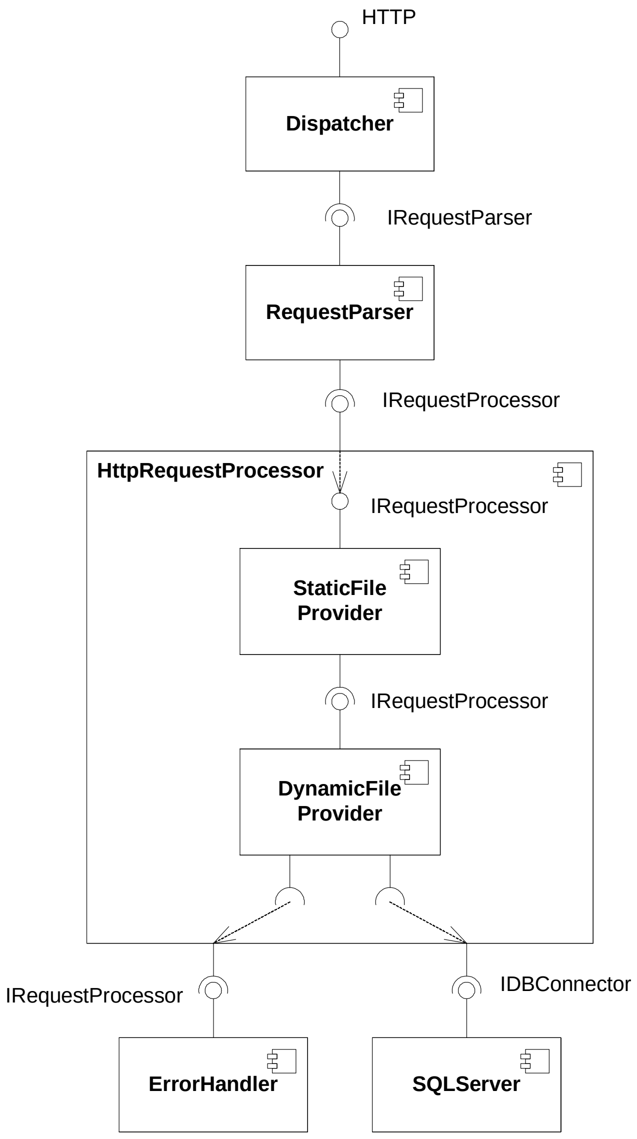

Fig. 3. Component architecture of a web server.

nent in the chain of responsibility. The ErrorHandler represents the end of the chain and returns an error message if the request could not be handled by any of the components. Additionally, the SQLServer is required to create dynamic HTML pages.

B. Component Implementation Description

This is what most software engineers think of if they are talking about software components.

A component implementation description is always associated with a real component implementation binary component. Certainly, multiple implementations can exist for a single description, implementing the component in different languages like Java and C# and technoligies, like Corba, J2EE, or .Net.

different specifications

UML the implementation of a component is described by a set of classes and their interactions.

Parametric contracts need a different view. They specifiy the dependency of provided an required interfaces by means of service effect specifications.

Since our main aim is the prediction of QoS, we focus on the description using parametric contracts in this articel. However, any kind of implementation description is possible.

Different implementation desriptions are possible. Composite components Basic components

In general, a component model does not provide any information about the inner structure of its components, since components are regarded as a black-box entities. However, black-box components inhibit the analysis of Quality of Service (QoS) attributes of the system, since important information about the dependency of these attributes on external services, hardware and software resources, and the usage profile cannot be specified. With black-box components, only static QoS contracts, like the ones specified by the QoS Modelling Language (QML) [2], can be used, but these are not sufficient for this task. If the required QoS profile of a component cannot be provided by its environment, the component is likely to work, but with lower quality. Therefore, we need additional information on the inner structure to analyse QoS attributes of a component in dependence on its environment at design time.

A *basic component* represents a basic block that is not further subdivided. It implements a certain functionality and is defined by its provided interfaces, required interfaces, and service effect specifications.

The fact that a basic component represents a single block does not imply that the real implementation hast to be one ’block’ as well. Since we are talking about the implementation description and not the actual implementation, this difference is possible and actually desired in some cases. For example, the actual implementation of a component might consist of a set of classes and/or subcomponents. This information is abstracted away in the description as a basic component. So, the description of a basic component only represents one view on a software component. Another view on a component are, for example, a class diagram or its source code.

A *composite component* consists of a set of interconnected subcomponents that realise its functionality.

The HttpRequestProcessor of the web server architecture.

As described in section V, a component implementation conforms to a complete-type if it offers at least the functionality specified in by the provided interfaces of the complete type and does not require more functionality than specified in the required interfaces of the complete-type.

C. Complete Component Type

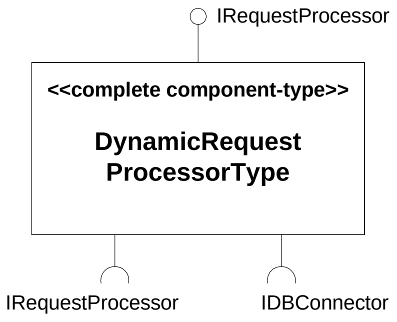

Fig. 4. Component type for dynamic request processors.

Looking at the architecture of the web server, we can identify different complete component types. For example, the components HttpRequestProcessor and DynamicFileProvide provide the IRequest-Processor interface and use the interfaces IDB-Connector and IRequestProcessor. The first one is used to create dynamic content of web pages. The second one is required to forward the request to the next component in the chain of responsibility. The corresponding type of both components is shown in figure 4.

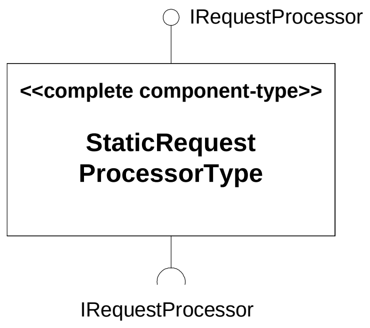

Fig. 5. Another component type used in the web server.

The type of the StaticFileProvider component shown in figure 5 provides and requires the IRequestProcessor interface only. So, it differs from the DynamicRequestProcessorType, which additionally requires the IDBConnector interface.

The types shown in figures 4 and 5 are derived from existing components and describe their complete provided and required functionality. It is obvious that a component conforms to a type if it provides and requires exactly the interfaces specified by the type. However, this is not the case in general. A component is likely to differ from its more general type, but, nevertheless, still be conformant to the type. So, when does a component conform to a type?

¡¡implementation-conforms¿¿ A component conforms to a type, if it offers at least the functionality specified by the provided interfaces of the type and uses only the functionality specified by the required interfaces of the type.

For example, a component must offer the IRequestProcessor interface to conform to one of the types in figure 4 or 5, but it can provide additional interfaces. Furthermore, a component can only use services that are specified in the required interfaces of the type. The DynamicFileProvider does not conform to the RequestProcessorType, since it uses the IDBConnector interface. Note that a component does not have to use all required interfaces of its type. So, all components that conform to the RequestProcessorType also conform to the DynamicRequestProcessorType, since they do not require the IDBConnectorInterface.

The definition of conformance corresponds to the view on required and provided interfaces as pre- and postconditions of components. The precondition can only be weakened. Thus, the component implementation must not use interfaces other than the required interfaces specified by the type. Furthermore, the postcondition can only be strengthened. The component can offer any interfaces, but it must at least provide the ones specified by the type. With this notion of conformance, the type system of components is contravariant.

This understanding of conformance between components and types allows us to define substitutability of components. A component A can be substituted by a component B if B conforms to the type defined by A. The type defined by a component includes all its provided and required interfaces. This is a very pessimistic definition of substitutability, which can be weakened under certain conditions.

D. Provided Component Type

In many cases, we are not only interested in the substitution of complete components including provided and required interfaces, but in a substitution with respect to provided interfaces only. This is the case if the functionality is more important than the requirements of a component or if we compare the functionality offered by different components. Furthermore, it might not be possible to specify all required interfaces of a component in advance.

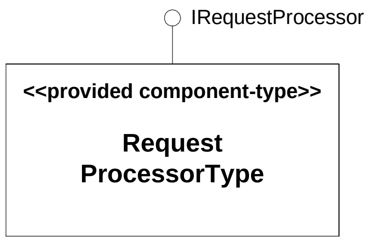

Fig. 6. provided-type of the RequestProcessorType and DynamicRequestProcessorType.

Hence, we want to allow conformance with respect to provided interfaces only. Therefore, we introduce a *provided-type* which only considers provided interfaces. The provided-type of the RequestProcessortType and the DynamicRequestProcessorType is shown in figure 6. It represents a more abstract view on software components. This is needed, since, during design time, it is often not clear what other components will be needed to implement the desired functionality. Expecting the software architect to decide this would require nearly the same effort as implementing the software component. Thus, the development process of a component should not be fixed to the required interfaces specified by a component type. It rather evolves with the development of the component. Provided and required interfaces are added and removed over time. The system architect specifies a basic set of interfaces that is used by the components to communicate. More interfaces might be added later to implement the functionality of the components.

The distinction of provided- and complete-types brings several advantages. By provided-types, we gain a high flexibility during software development. Furthermore, we can define a substitutability focussing on the functionality of a component. On the other hand, complete-types define a strict substitutability. If a component is replaced by another and both conform to the same complete type, we can assure that certain classes of interoperability problem cannot occur. Moreover, complete-types allow a complete specification of the externally visible behaviour of a software component. The coexistence of both concept offers a high flexibility in software architecture design.

E. The Component Type Hierarchy

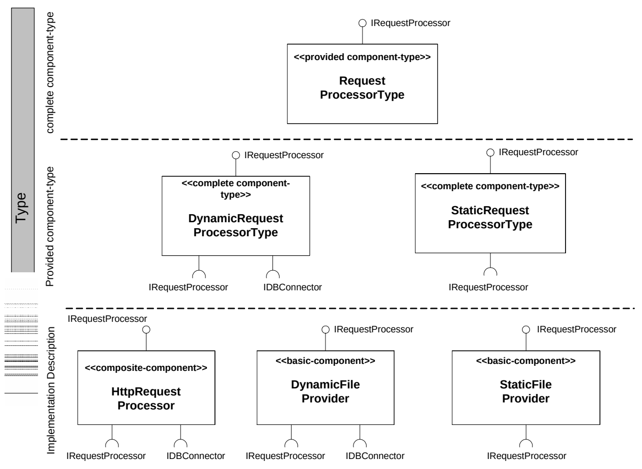

Fig. 7. Type Hierarchie

Figure **??** gives an overview of the relation of provided-types, complete-types and component implementation descriptions. TODO:Figure and description

With the implementation description of software components, we have enough information to compute QoS attributes of the component in dependence on external services. TODO:Example

VII. Deployment

The step of placing or installing software on the target systems. This includes configuration steps if necessary.

General understanding for software components: Placement of a component in an execution environment.

Our understanding: The deployment of a software component defines its context. 1. Assembling of components (Connections) 2. Allocation of components on resources

Both steps can be done independently (two roles, different persons). Nowadays they are generally considered as deployment.

A component can be embedded into different contexts, since it is a unit of independent deployment. This can happen several times within the same system. Leaving type theory.

Within the same system, we have that the same component has different contexts, including the wiring, the mapping to resources and the containment.

Explicit modelling of the context. Two dimensions: Assembly and Allocation, Computed vs. Specified Values.

–Table with different context aspects–

Assembly and allocation can contain special configurations of a component.

Connections and their deployment: Today, either fixed connection between components or naming services. Both do not allow individual connection of components. So, components are no real units of independent deployment at the moment. Technical realisation required.

Composite Components can only be deployed as one piece. Resource diagram and assembly diagram are specified independently.

Communication nodes, which only model network structures.

VIII. Deployment of Connections

Deployment of Connections on paths (a path describes the route between two nodes, acyclic)

– Overview –

IX. Components at Runtime

Cardinalities

Concurrency modelling

Benefit: QoS Prediction

X. Conclusion

Future Work - MOF-Schema and OCL-Constraints - MDA-Transforms - Description approximating the run-time view of the system (virtual interactors) - Modelling of concurrency, asynchronous method calls - QoS analysis - Process model for CBSE with the Palladio Component Meta Model

Acknowledgments

Thanks to everyone.

References

[1] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[2] Svend Frølund and Jari Koistinen. Quality-of-Service Specification in Distributed Object Systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory, September 1998.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[4] Bertrand Meyer. Applying “Design by Contract”. *IEEE Computer*, 25(10):40–51, October 1992.

[5] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1997.

[6] Object Management Group (OMG). Unified modeling language specification: Version 2, revised final adopted specification (ptc/05-07-04), 2005.

[7] 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.

[8] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.