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.

Manuscript received January 20, 2002; revised August 13, 2002.

M. Shell is with the Georgia Institute of Technology.

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

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. Types and Design-by-Contract

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 [4] [1] into consideration this has several implications:

- The precondition of every operation of the type has to be weaker than the supertype's operation precondition.
- The postcondition of every operation of the type has to be stronger than the supertype's operation postcondition.

[1] A formal definition can be found in the book by Meyer [4]

- Any invariant of the supertype has also to be true for all subtypes.

Note that every type which is used in any kind of declaration is already a contract stating that the values contained in the declared variable conform to the specified type. Hence, the rules given above can be applied to type declarations as well. Applying this to the parameter and the return types of an arbitrary operation being part of an inheritance hierarchy leads to a type system which is contra-variant. A discussion on the advantages and disadvantages of so doing can be found in [4, p. 628ff].

Another use of a type system is 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 run-time error. For this, a necessary condition is that the required interface has to be a supertype of the provided interface it is bound to.

C. 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" [3]) from methods to software components. A component guarantees to offer the functionality specified by its provided 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 [6] 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.

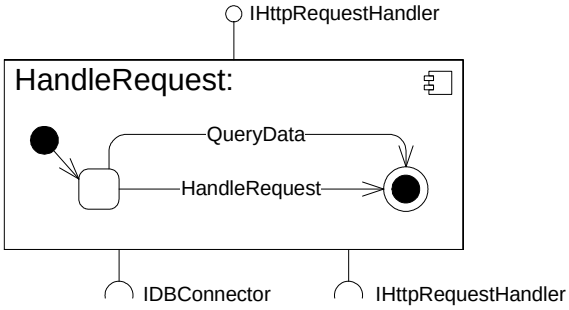

Fig. 1. Service effect specification of the method HandleRequest in the DynamicFileProvider.

Figure 4 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.

Discuss what is modelled by a service effect specification. What is an external service?

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 components interfaces are a key concept as they serve multiple purposes. As in object oriented languages, which have the concept of interfaces, they can be used to build a type system as introduced in section III-B. In our model interfaces can have an arbitrary amount of super-interfaces. A common constraint for the hierarchy of interfaces is that any given interface can not be supertype of itself, which gives us a acyclic subtype hierarchy.

In order to define a subtype relationship on interfaces, 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. In our model we have

some constraints on the types which can be used. As the external view of components is characterized by its interfaces, only interface references and basic data types (integer, string, ...) are allowed but no component references.

Interfaces are applied to specify the allowed communication between communicating entities. The contracts specified in the interface (method contracts, invariants) characterize the valid behaviour of these entities. In object oriented languages an object can act in two roles with respect to an interface: server or client. In the server role, the object "'implements"' or "'realizes"' the operations specified in the interfaces and observes the method pre- and postconditions. In the client role, the object calls services offered in a given interface by fulfilling the precondition and expecting the postcondition. However, in both cases the interface and its associated contracts serve both roles as contract on which they can rely.

As with legal contracts, interfaces can exist even when no one actually declared their commitment to them, i.e., there is no specific client or server. For example, this is used to define a certain set of standardised interfaces of a library to enable the construction of clients and servers of these libraries independently. Thus, in the Palladio Component Meta Model the concept *Interface* exists as first class entity which can be specified independent from other entities.

The specification of the contract which is represented by an interface can be enhanced by including a specification of the sequence in which the interface's operations can be used. This kind of information is called a protocol. The protocol is a special class of the more general concept of arbitrary preconditions for methods. Any kind of protocol can be expressed via preconditions. Thus, the protocol is an abstraction of the set of all preconditions. The abstraction is often made based on the expressiveness of the used specification formalism. For example, consider using finite state machines as protocol specification formalism. With this formalism it is impossible to express the valid call sequences of a stack exactly (the amount of push calls always has to be equal or greater then the amount of pop calls). Nevertheless, FSM-protocols can be analysed with quite efficient algorithms.

Using the information described above, the subtype relationship of any two arbitrary interfaces I_1, I_2 can be specified as follows. Interface I_1 is subtype of I_2 if it is able to fulfil at least the contracts of I_2. In detail, this means it has to be able to handle all the (single) method calls which I_2 can handle. Additionally, it must also at least support the call sequences which I_2 supports.

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, re-entrance 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

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?

Interpretations of Interface Relations

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.

VI. Component Type Hierarchy

The different meanings of provided and required interfaces and the possiblity to specify the inner structure of a software component using UML 2.0 and/or Parametric contracts brings up the question of substitutability. What has to be fulfilled by a software component that shall replace another component in its context?

To answer this question is not an easy task, especially, if we want to allow different mean-

ings of required and provided interfaces and support the analysis of QoS attributes. We need a component type system that supports different kinds of substitutability and, furthermore, handels QoS relevant information.

Current component meta models address this issue only briefly. Furthermore, most of them do not clearly distinguish between components and component types. For example, the UML 2.0 states:

> "[...] 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 [5, p.142]."

Here, the terms *component type* and *conformance of components and types* are introduced, but their meaning is not further clearified. Within the scope of the UML 2.0 superstructure components and component-types are actually treated equivalently [**?**]. This might be approapriate in many cases, but might lead to a blury concept of software components.

In the following, we introduce the component type hierarchy as it is used by the Palladio Component Meta Model. There, we distingish the provided-component-type and complete-component-type as two different type concepts for software components. Additionally, we consider a component-implementation-description that specifies the internal structure of a component.

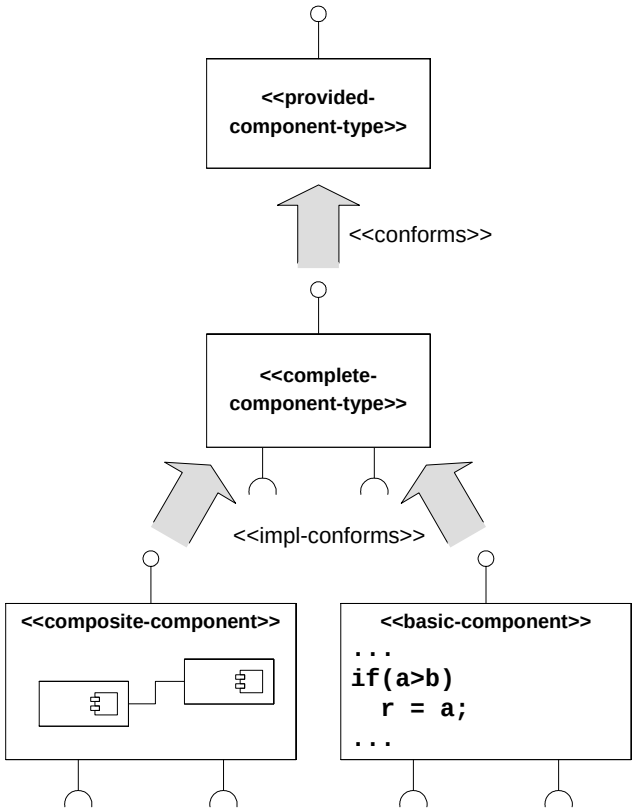

Fig. 2. Different type levels of a software component.

Figure 2 shows the component type hierarchy. The provided-component-type is mainly characterised by provided interfaces. In addition, the complete-component-type considers required interfaces that can be used by an implementation of the type. We allow different kinds of component implementation descriptions: Basic components and composite components. Basic components describe the intra-component dependencies of provided and

required interfaces using parametric contracts. Composite components are used to construct new components out of other components.

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

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

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 complete-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 problems cannot occur. Moreover, complete-types allow a complete specification of the externally visible behaviour of a software component. The co-existence of both concepts offers a high flexibility in software architecture design.

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 3. The web server has been implemented in C# using the Microsoft .Net 1.1 framework. It consists of a set of components that communicate via a clear defined set of interfaces only. The web server uses all concepts relevant for component based software development. Moreover, its architecture is comparable to many three tier applications.

The main part of the web server is realised by three components: Dispatcher, RequestParser, and HttpRequestProcessor. The Dispatcher listens for incoming connections. For each HTTP request, it spawns a new thread 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 [2]. 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 component in the chain of responsibility. The ErrorHandler component 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.

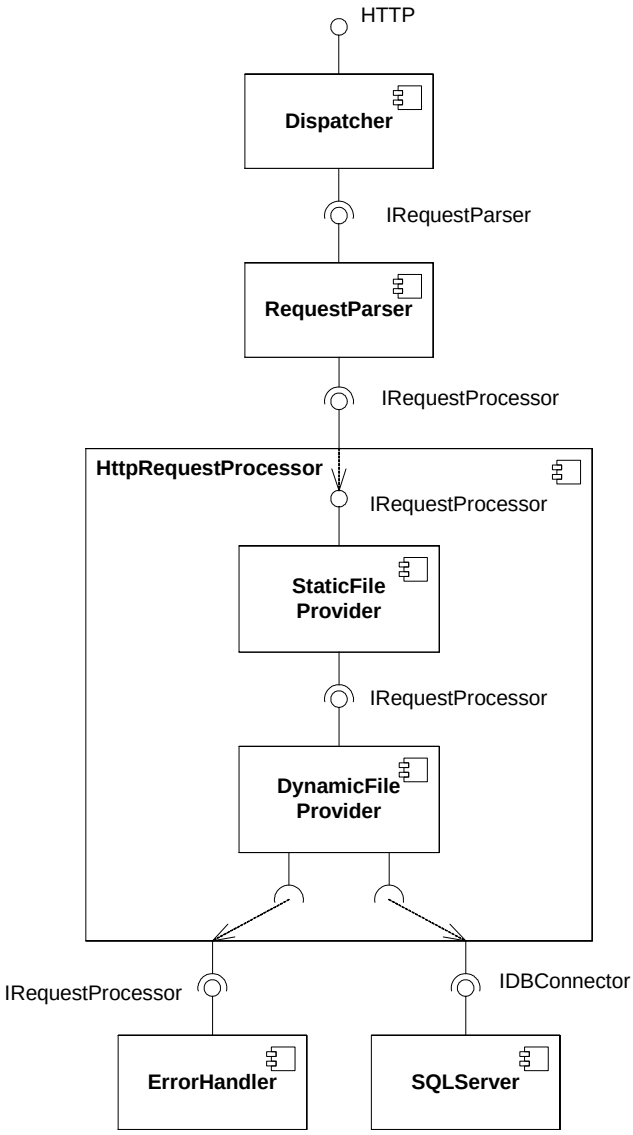

Fig. 3. Component architecture of a web server.

Component Implementation Description

Today's understanding of software components is strongly influenced by state-of-the-art technologies, like J2EE, CORBA, and COM+. Most software architects think of a software

component as a 'physical thing', like a piece of code, a special kind of class, or a binary file. All of them realise a certain functionality and have special requirements to the environment in order to function correctly. The component implementation description specifies the provided and required interfaces and the inner structure of such components.

A component implementation description is always associated with at least one binary ('physical') component. Certainly, multiple implementations can exist for a single description. For example, the same specification can be implemented in different programming languages like Java and C# and/or component technologies, like CORBA, J2EE, or COM+.

Different possibilities exist to specify the internal structure of a component. For example, the UML 2.0 superstructure [5] allows to describe the inner component structure as a class diagram. On the other hand, parametric contracts specify the dependency of provided and required interfaces by means of service effect specifications. Moreover, some components are constructed by composing other components. These inner components and their interconnection are also a representation of the inner component structure.

In this article, we focus on the component implementation description with parametric contracts and composite structures, since our main aim is the prediction of QoS attributes. Both concepts have been proofed to be beneficial for this task. Thus, we distinguish between two types of component implementation descriptions: Basic components and composite components.

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 has 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. This information is abstracted away in the description as a basic component. In the web server example shown in figure 3, the components StaticFileProvider and DynamicFileProvider are described as basic components.

Figure 4 shows the service effect specification of the method HandleRequest of the DynamicFileProvider modelled as a finite state machine. If the DynamicFileProvider is responsible for the incoming request, it executes a query on the connected database. It uses the

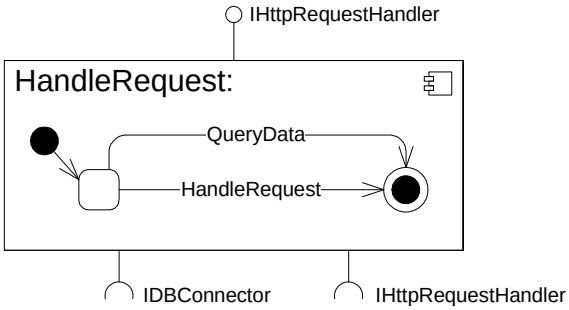

Fig. 4. Service effect specification of the method HandleRequest in the DynamicFileProvider.

queried information to construct the web page and returns the result to the client. This is realised by internal component code that is not modelled by the service effect specification shown in figure 4. If the DynamicFileProvider is not responsible for the incoming request, it forwards it 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.

A component cannot only be implemented by writing source code that realises the desired functionality, but also by composing components. We explicitly consider the composition of components and the writing of source code as equivalent means of implementation.

A *composite component* consists of a set of interconnected subcomponents that realise its functionality [5, TODO:page].

As in UML 2.0, the interconnection of the inner components is realised by assembly connectors while delegation connectors describe how the externally visible interfaces are mapped to the inner structure of the component.

It is important to mention that the composite structure of a component can be hidden from the deployer. The service effect specifications of a composite component can be computed out of the inner components and their interconnections [**?**]. Unfortunately, composite components can hardly be represented with today's component technologies. The construction of a new component out of existing components can often only be done manually. So, composite components remain logical constructs in most cases. More sophisticated technologies are needed, which offer an easy way to hide the inner structure of composite components and generate its physical representation.

The HttpRequestProcessor is the only composite component shown in the architecture of the web server in figure 3. It contains a set of different request processors that are organised in a Chain of Responsability. This hides the different processors from the other components and presents them as a single entity to the outside world.

Next, we introduce the complete component type as a more abstract view on software components.

Complete Component Type

A complete component type generalises the component implementation description by omitting the inner structure of a component. Thus, it only contains provided and required interfaces.

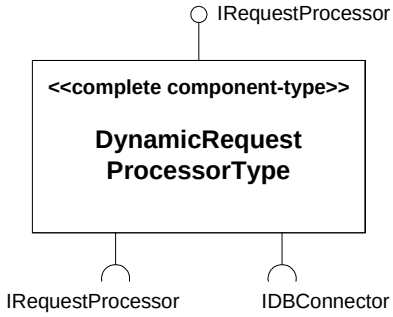

Fig. 5. Component complete type for dynamic request processors.

Looking at the architecture of the web server, we can identify different complete component types. For example, the components HttpRequestProcessor and DynamicFileProvider provide the IRequestProcessor interface and use the interfaces IDBConnector and IRequestProcessor. The first one is used to retrieve dynamic content of web pages from a data base. 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 5.

The type of the StaticFileProvider component shown in figure 6 provides and requires

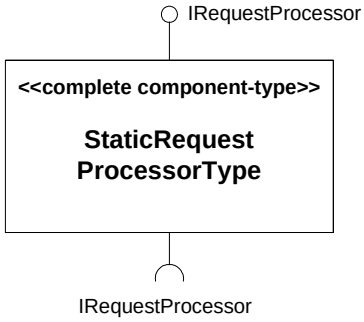

Fig. 6. Another component type used in the web server.

the IRequestProcessor interface only. So, it differs from the DynamicRequestProcessorType, which additionally requires the IDBConnector interface.

The types shown in figures 5 and 6 are derived from existing components and describe their complete provided and required functionality. A component conforms to a type if it provides and requires exactly the interfaces specified by the type. This very basic definition of conformance is not sufficient for a type system of component. A component is likely to differ from its more general type, but, nevertheless, still be conformant to the type. Thus, we can define the conformance relation between a component implementation description and a complete component type as follows:

A component implementation description conforms to a complete component 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. This type of conformance is labelled with the UML stereo-type ≪implementation-conforms≫.

For example, a component must offer the IRequestProcessor interface to conform to one of the types in figure 5 or 6, 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.

The relation ≪implementation-conforms≫ allows us to define substitutability of components. A component A can be substituted by a component B if the description of B conforms to the type defined by A. The component complete type defined by a component and its description includes all its provided and required interfaces. Therefore, a component implementation description always conforms to at least the type it defines. This definition of substitutability is very conservative. It ensures that the new component can function in the existing environment, but, on the other hand, might be to restictive. Therefore, we introduce the provided component type next, which allows a more flexible notition of substitutability and type conformance.

Provided Component Type

In many cases, we are not solely 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. This view is strongly influenced by object oriented programming, where required interfaces and/or classes are often not listed explicitly. There, substitutability of classes and interfaces is defined on the basis of the ≪realize≫ and ≪extends≫

relations [**?**].

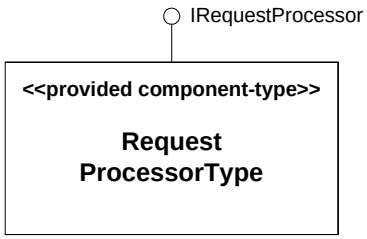

Fig. 7. provided-type of the RequestProcessorType and DynamicRequestProcessorType.

Therefore, we introduce a *provided-component-type*, which only considers provided interfaces for conformance checks. The provided-component-type of the RequestProcessortType and the DynamicRequestProcessorType is shown in figure 7. It only contains the single provided interface IRequestprocessor. Required interfaces can be specified for the provided-component-type as well, but their meaning differs from the ones specified in the complete-component-type. Here, required interfaces are used in the sense of can-require-interfaces, whose definition does not enforce any restrictions. Thus, the conformance relation of provided- and complete-component-types considers provided interfaces only.

A complete-component-type conforms to a provided-component-type, if it offers at least the functionality specified by the provided interfaces of the provided-component-type This type of conformance is labelled with the UML stereo-type ≪provided-conforms≫.

Since it exists an equivalent type for each component implementation description, the ≪provided-conforms≫ relation is indirectly defined between component-implementation-descriptions and provided-component-types.

The provided-implementation-type completes the hierarchy introduced in the beginning of this section. Next, we summarise the results of this discussion.

The Component Type Hierarchy

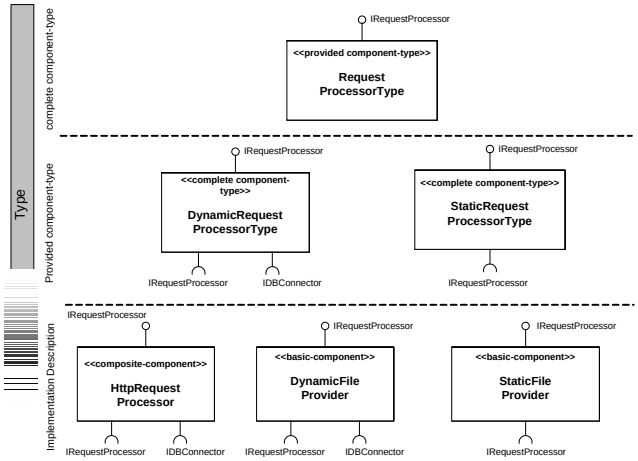

Fig. 8. Type Hierarchy

Figure 8 gives an overview of the relation between provided-component-types, complete-component-types and component implementation descriptions. On top of the hierarchy stands the provided-component-type. It is the most general and flexible view on software

components. Its focus is on provided interfaces. It is also possible to add required interfaces, but with a very loose semantic. The complete-component-type is placed on the mid level of the hierarchy. It considers required interfaces with a more strict semantic, which prohibits the usage of interfaces other than the specified required interfaces. The component implementation description on the bottom of the hierarchy adds knowledge about the internal structure of a component to the model. As indicated by the fading bar on the left hand side, we do not consider the description as a type.

With the implementation description of software components, we provide a lot of information, which allows us to compute QoS attributes with a higher precision. For example, we can determine the influence of the QoS of required services on the provided services [**?**], [**?**], [7]. However, the QoS attributes are not only influenced by external services, but also by the underlying hardware and software resources. Thus, our next step is the modelling of the deployment context.

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

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

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

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

[6] 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.

[7] Ralf H. Reussner, Heinz W. Schmidt, and Iman Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes*, 66(3):241–252, 2003.

[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.