# A Method for the Design and Development of Distributed Applications using UML

M. Born
*GMD Fokus*
*Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany*
*born@fokus.gmd.de*

E.Holz, O. Kath
*Humboldt University Berlin, Dept. of Computer Science*
*Rudower Chaussee 5, 12489 Berlin, Germany*
*{holz|kath}@informatik.hu-berlin.de*

***Abstract***

*With this contribution we present a design and development method for distributed applications, which are running on top of advanced object middleware platforms. We introduce the basic concept of distinction between the set of concepts and their relations, the definition of a notation supporting such concepts and rules for automatic code generation that help to provide a smooth transition from the design step to the implementation stage. Allthough the set of concepts is based on RM-ODP, we refine a number of ODP definitions in order to reflect practical design tasks. The supporting notation we present here rests upon costumization of the Unified Modeling Language (UML). Automatic code generation issues are presented briefly, taking a extended CORBA 2.3 object middleware as target distributed infrastructure.*

## 1. INTRODUCTION

A dedicated and efficient design methodology contributes significantly to a reduction of the time to market distributed applications and telecommunication services. An appropriate treatment of all kinds of communication aspects lies in the very nature of the targeted application domain. These aspects span from functional requirements (e.g. transactionality) on object interactions over quality of service issues to security properties. Taking into account the broad acceptance of object middleware technology, middleware platforms provide an ideal implementation environment for such designs. Therefor the design method should take this into account.

Current approaches to the design of distributed applications do base on object-oriented analysis and design notations, namely on the Unified Modeling Language (UML). Examples here are the Rational Unified Process (RUP), Enterprise Distributed Object Computing (EDOC) or the CORBA Profile for UML. RUP is a very general design method, it does not focus on the specific problems that occur within systems that are distributed. EDOC is very specific to business process modeling, it is not a software design method in that sense, it heavily bases on CCM. CORBA Profile is a reflection of concepts of the CORBA Interface Definition Language (IDL) concepts in UML, hence focussing purely on the structure and signature definition, but not on behavioral aspects and object interaction specific concerns. Also aspects of the design process are not treated here.

Contrasting to these technologies the Basic Reference Model for Open Distributed Processing (RM-ODP) defines a set of commonly accepted concepts and terminology to model distributed systems. Although ODP provides features to separate and to relate different views during the design of distributed system, it does neither aim at a provision of a concrete notation nor a specific method in terms of procedural instructions for the development of distributed systems.

The method presented in this paper takes up the experiences of the different approaches mentioned above. A first objective for the work was the requirement that a design method should be based on well defined concepts, that form the concept space for the method, and should be accompanied by a suitable notation supporting the concepts. Moreover, the concepts as well as the supporting notation must not rely on a specific middleware technology. Instead, mappings shall be defined, that represent a concept or a set of concepts within a target middleware environment.

A design method in this sense should consist of three parts, which are (1) the concept space, that defines a notation and platform independent terminology for the specification of distributed applications, (2) one or more supporting notations, that visually reflect the complete concept space or a subset of it and enable different views on the system to be developed, and (3) one or more sets of mapping rules to enable a smooth transition from the design to the implementation on concrete platforms.The separation into concept space, notation and mapping to a runtime environment as introduced above has several advantages. First, the concept space is independent from a specific design notation. Design models can be developed in different notations but are based on the same concepts and terminology. Design information can then be exchanged on the basis of the common concept space. Secondly, both the notation and the concept space are independent from a specific runtime-environment. The same design can be mapped onto different environments. This enables a high flexibility and is also important for the aspect of re-usage of component design modelsThe starting points for the concept space of our method are a critical evaluation of the RM-ODP terminology and practical experiences in software development projects. The notation is specified as an UML profile again taking into account practical experiences and ideas of the UML CORBA Profile. Although mappings to different middleware platforms are currently under development, within this paper only the ideas for a mapping to a CORBA 2.3 middleware environment will be sketched in order to proof the applicability of the proposal.

The remaining part of this paper is structured as follows: Section 2 gives an overview over the concept space introduced for the specification of distributed applications and defines five different views to organize and structure such specifications. Section 3 defines a concrete language for visual representation of the concepts in terms of an UML-Profil. Finally Section 4 demonstrates the application of the proposed methodology and notation to a concrete development project. An outlook on future work is given in the closing Section 5.

## 2. CONCEPT SPACE

As motivated before, we introduce a concept space as the foundation of our design method independently from its supporting notation. The central terms here are interface, objects and components. An object is defined in [10] as a model of an entity, that encapsulates state and behavior and that is distinct from any other object. In order to be more clear on this term, it has to be distinguished between the type of an object (in the sense of an object template in [10]) and an object itself as instance of such a type. Due to the reason that we focus here on the functional decomposition of a distributed system, we specialize the term object type by referring to it as computational object type (CO). We use the term *CO instance* to denote instances of those types. COs define units of *distribution*, which interact with their environment (i.e. other computational objects) via well defined interfaces. Interactions between CO instances are transparently supported by a distributed infrastructure.

In contrast to CO the term *component* is used to refer to units of *deployment* in a distributed system. Components contain implementation artifacts (e.g. classes of an object-oriented programming language), that realize the behavior of one or more COs. Consequently there is an association between one or multiple COs as the unit of distribution and a component as the unit of deployment. Beside containing CO implementations, a component provides additional interfaces and accompanying implementations for the component life cycle management (e.g. creation and

destruction of CO instances). A component can be deployed onto nodes, but needs a runtime environment their. Nodes are the processing entities within the target distributed environment. The definition of a component here corresponds directly to the definition of [19]. Since ODP does not deal with the concept of deployment in detail, there is no direct corresponding entity to a component as part of the ODP viewpoint languages. However, an analogy can be seen to the definition of a basic engineering object (BEO) which needs support from a distributed infrastructure in the same way a component does, and to the definition of a cluster (grouping of BEOs). The problem however is the requirement of ODP, to have an 1:n relation between COs and BEOs, which is a contradiction to the component definition. In our concept space, there is a n:m relation, meaning that different components may contain the implementations for a CO and that one component may contain implementations of many COs. Practical experiences have shown, that such a n:m relation like introduced here is more realistic.

In the remaining part of this contribution, we mainly focus on the specification of COs and CO behavior, since we aim to provide a design method for distributed systems, leaving out a detailed definition of deployment aspects for the time being. Referring to [10], a CO instance communicates with its environment, i.e. other CO instances, at its interaction points. Those interaction points may be uniquely referenced to. From the computational perspective, interactions are classified into three kinds:

- operational interactions relate to remote method invocations (RPC style),
- signal interactions refer to asynchronous sending and reception of atomic information (information publishing style) and
- stream interactions which are a continuous sending/reception of information (continuous media delivery style).

An interface (type) defines the signature of an interaction point. In contrast to the definition of a computational interface given by [10], an interface here allows to combine all three interaction kinds within the scope of the same interface type. By doing so, the design of the computational entities of a distributed system turned out as to be more simple and intuitive. As it will be shown later, the separation of the interaction kinds is a technology issue and therefore subject for the mapping onto specific middleware platforms.

There are two different types of relations between COs and interface types:

- *supports* relation - a CO instance may provide instances of an interface supported by it and
- *requires* relation - a CO instance makes use of an instance of an interface required by it.

With these relations, we do not refer directly to configuration aspects, which deal with instances of COs and interfaces. Instead we constrain configuration definitions having to be defined only between instances of COs and interfaces for which a corresponding *supports* and *requires* relation is specified.

The signature of an interface type is defined as the set of its interaction elements. Interaction elements are distinguished with respect to the interaction kind they belong to:

- Operations and attributes (as a shortcut for get- and set-operations) for operational interactions,
- Consumed and produced signals for signal interaction,
- Sourced or received media sets for stream interaction.

Interaction elements in turn are also defined by signatures. An operation signature consists of the operation name and a set of parameters, each of them having a type and a direction specification (*in, out, inout*). Furthermore, operations may specify terminations in form of return types or exceptions. In fact, this definition is well known from existing interface definition languages. The only exciting issue is the data type system to be applied. On one hand a design method shall be open to different such type systems. On the other hand, it has to be concrete with respect to data types to allow for deterministic mapping onto specific middleware environments. Currently, we have integrated the CORBA IDL type system [8] into the design method. It is planned to take also other approaches into account, like that presented in [11].

Concerning the signature of signals, we distinguish between the concept signal itself and the information, which the signal is carrier for. The type of the information is described in terms of values, and a signal declares one or more values as being carried by it. When using the CORBA IDL type system, values directly correspond to IDL value types.

A signature of a media set is given by an aggregation of media, where each medium is interpreted according to one or more appropriate media types. A medium here refers to the continuous provision of information, were the information is formatted in conformance to one of its realizing media types. Given that, a medium characterizes the information delivered, while a media type characterizes the format of the information delivery. Commonly known media types are MIME types [22].

Besides the specification in an interface an CO definition itself may also contain directly operations and attributes. The intension is to allow for the specification of functionality which is closely related to the CO itself, e.g. for initialization purposes.

The concepts introduced so far are pure type information, namely signatures and potential structures and therefore are referred to as structural concepts. Consequently a specification of a distributed application given in these terms forms the structural view.

Besides these structural definitions, we introduce also concepts to describe configuration aspects of CO instances. With the term configuration we refer to mechanisms allowing to access instances of supported interfaces as well as to store references of required interfaces at an CO instance. The concept *port* is used to denote both, the access points to instances of supported interface and the points to store references to required interfaces. Ports are uniquely identifiable in the context of an CO. Since there can be potentially an infinite number of interface instances supported by a concrete CO instance at runtime, ports can be declared as being single or multiple ports. The property *single* implies that only a single interface reference can be registered or obtained at that port, whereas a port with the property *multiple* allow to dynamically register or obtain multiple interface references. The specification of the configuration of ports belonging to the COs make up the instance view. Currently, only one such configuration is foreseen per CO definition. Therefore, these definitions complete the view on a CO as a type in addition to the structural view. It requires further study, whether it is feasible and practical to allow the definition of more of such configurations. Especially we currently do not see an application case where such multiple configurations are necessary.

As motivated in Section 1 the focus of the design approach introduced in this contribution is on the communication between the distributed entities. For that reason, concepts are required to allow the specification of properties, rules for and constraints on interactions in certain contexts.

The main concept to support this is the concept of binding. Bindings are associations between instances of interfaces supported by instances of COs. A binding is a prerequisite for an interaction, that is interactions between CO instances may occur via the bound interface instances only. A binding defines always a (common) subset of the signature of the interfaces involved in the binding and by doing so it specifies the interaction elements which may be used for interactions in the context of that binding (binding context).

Bindings can be established implicitly or explicitly. In order to establish a binding, it is required that the subsets of the signatures of the interfaces involved in the binding are complementary to each other. Rules for a definition of the conditions under which signatures are complementary can be found in [10]. An example is the requirement that the interface types defining the signatures are in a subtype relation.

For an explicit binding there exists always an instance of a special CO, the binding CO, realizing this binding. The concept of binding CO is a specialization of the concept CO as already introduced. Binding COs have all capabilities as ordinary COs, they can declare supported and required interfaces as well as ports. A common example for a binding CO is the model of an event channel. The reason for having the notion of explicit binding is that some actions may have to be performed

before the interaction between COs can take place. Such behavior is performed by the binding CO instance. Explicit binding can be applied for all three kinds of interactions.

Implicit binding on the other side is only available for operational interaction. Here the binding is established in the moment a client CO instance invokes an operation at an interface instance provided by a server CO instance. The binding is deleted after the termination of the operation.

For both kinds of bindings, rules can be specified to determine the characteristics of the interactions in the binding context. The set of binding rules for the server and the client side of a specific binding is referred to be the binding contract for that binding. Binding rules themself are defined as constraints formulated as logical expressions over special types. We call these types Quality of Service (QoS) types. Their attributes may be assigned values according to desired QoS characteristics or policies. Examples for these attributes include security levels, bandwidth for stream interaction, response time for operational interaction or transactional policies. This concept of QoS types is general enough to express beside performance and reliability characteristics also security or transactional requirements for a certain binding. In order to restrict the QoS types available for the specification of a binding rule, it is required that the QoS types used are associated to the interface type involved in the binding.

The concept of QoS types declared for specific interface types is known from other QoS specification approaches like [20], however the approach to define rules for bindings that are possibly different in different contexts is more dynamic. To be able to identify a binding case at runtime we use the concept of predicates. Instances implementing COs are checked whether or not they fulfil predicates attached to binding cases. Taking the results of this check an CO instance wanting to participate in a binding selects an appropriate binding case and by that the binding contract to be used at runtime.

A specification given in terms of policies and rules on interactions and binding of interfaces is called interaction view. Together with the other two views (structural and instance) it provides a sufficient set of concepts to form a black box model of the system by concentrating on communication aspects only. However, as motivated in Section 1 the design should lead to an implementation and therefore some internal aspects regarding the implementation of COs have also to be covered. Similar to the RM-ODP our design approach defines an object as encapsulation of state and behavior, but until now it does not address the way, how this behavior is provided. While the term CO refers to an abstract entity, in concrete distributed systems the expected behavior of objects is realized by programming language elements, e.g. classes and their implementation in a object-oriented programming language. For that reason, a design method must consider the relationships between abstract, referable objects and concrete implementation language code, that implements the behavior of such objects. Questions wich are of interests here are:

- What are the structural elements implementing the COs behavior?
- What information can be considered as being the COs state?
- What is the relation between implementation elements and the interaction elements at the interfaces, i.e. who is responsible to implement a particular operation of a supported interface?
- What part of the state information is needed to provide the behavior of a certain interaction element?

To answer these questions in the design model, additional concepts are included in the concept space. The programming language elements realizing parts of the behavior of COs are called artifacts. An CO is implemented by a set of those artifacts. Interaction elements of the supported and required interfaces are associated to artifacts which means that the artifact implements those elements, i.e consumes a signal or provides behavior of an operation. State information is described by storage types. Storage types have attributes who’s values form the current state. Such storage types can be assigned to the association between an interaction element and an artifact. Hereby it is expressed, that the part of a COs state which is covered by the storage type is required for the implementation of the selected interaction element.

The concepts artifact, their relations to COs and interaction elements as well as storage types and their relation to realizations of interaction elements are considered as the implementation view in our design method.

The concepts we presented here are either to be understood as refinements of the ODP computational language concepts, like CO, interface or the different interaction elements or are extensions to the ODP computational concept space, like the implementation concepts.

Since the concept space introduced above covers a varietey of different information, we use different views to organize and structure the information. These views, the structural, instance, interaction and implementation view form a further refinement of the computational viewpoint of ODP. Corresponding entities for the concepts of the structural view can be found directly in the ODP computational language. The only difference lies in the more generic interface definition in our concept space, i.e. there is no distinction into different classes of interfaces with respect to the interaction kind. The instance view states requirements on all instances of a CO, therefore instance view specifications belong to the same abstraction level as the specifications of the structural view, i.e. they are computational specifications. The interaction view finally expresses rules for specific binding cases and interactions that concern sets of CO instances, what refines the concepts of computational binding as defined by ODP. On the other hand, it enhances the concept of implicit binding by a quality of service support. In ODP such a support is foreseen only for explicit bindings. As already mentioned, the implementation view is clearly an extension to the ODP concepts, it can be seen as covering parts of the computational and of the engineering viewpoint, focussing on the structure and behavior of the elements providing the behavior of one or multiple COs. Since ODP itself does not provide any explicit concepts relating a CO to the concrete realization of its behavior, it does also not provide sufficient means for the description of a collection of CO implementations together with their life cycle management - i.e. no means for the term component as unit of deployment. However, obviously components have static and dynamic requirements upon the execution environment they are hosted by. The definition of such requirements is the main concern of the additional view, the deployment view. Details of that view are not provided here. There is an ongoing international project performed in the EURESCOM program which deals with the definition of deployment aspects for components [21].

Our design method does not cover the other ODP viewpoints. The ODP enterprise viewpoint focuses on the requirement analysis for the system to be developed. This is within the software engineering process prior to the design, even if the development process is an iterative approach. The information viewpoint deals with the structure of information to be manipulated by the system to be developed. This does not need to be supported in a specific way because standard notations (e.g. UML) and methods for that purpose like the Entity Relationship Model (ERM) are already existent. However, one important relation between an ODP information model and our concept space is the reflection of information entities in signatures of the structural view and the state descriptions in the implementation view. That means, that an information model can serve as an input for our design method.

## 3. SUPPORTING NOTATION

A notation is a visual representation of the elements of the concept space. It shall ideally neither limit the expressive power of the concept space nor add additional specification overhead. Further requirements on a notation are intuitiveness and ease of use. There are two possibilities to obtain a notation for a concept space, the definition of a notation from scratch or the adaptation of an existing notation. While the first solution would usually lead to a notation fulfilling all requirements, it suffers from the perception to invent yet another language. This concerns not only the necessity of the user to learn that language but also the availability of supporting tools. Within our method we opted therefore for the second solution.

In order to select an appropriate existing notation, the set of requirements has been extended by:

- acceptance of the notation and availability of tools,
- ability to limit/extend the notation,
- graphical notation and object-oriented concept space as foundation.

By applying these criteria the number of candidates could be limited to object-oriented analysis and design notations and to formal description techniques like SDL. Eventually we have chosen UML as our notation mainly due to the ability to adapt the language to selected application area by means of profiles.

A UML profile is a specialization of the UML metamodel dedicated to a specific application domain. It provides a context for the definition of extensions to and of constraints on existing metamodel elements at the model level by applying the UML extensibility mechanisms. Examples for such profiles currently under development are the UML CORBA Profile, the Profile for Business Modeling or the UML Real-time Profile. In general the definition of a profile consists of the following constituents:

- Selected elements of the original metamodel,
- Definitions of stereotypes, tagged values and constraints,
- Description of the new elements,
- Visual representation
- Wellformedness and transformation rules.

The description of the new elements has already been given by the definition of the concept space, i.e. each new element corresponds to a concept of the concept space. There have also not been introduced any new graphical representations, instead we restricted us to the usual UML symbols with stereotypes in Guillemet-style. Hence we concentrate here on the remaining parts of the profile.

The starting point for our profile are the model elements used in the structural view of UML. All concepts of the concept space are reflected by corresponding stereotype definition in the profile. Due to the number of concepts within this paper only an idea of the definition can be given. The base classes for most of these stereotypes are the UML model elements *Classifier* and *Relationship*:

- *Classifier* stereotypes: CO, Interface, Port, Component, QoS, MediaType;
- *Relationship* stereotypes: supports, requires.

Tagged values have been used to specify the specific characteristics of the stereotypes whereas constraints in OCL are applied to define the wellformedness conditions and the restrictions. It may be noted that we did not use the standard UML concept of interface as the base for our specialized interface definition. The reason for doing so are the strong requirements of UML interfaces (e.g. no attributes).

The complete profile has been defined as a package with a set of additional wellformedness rules, again given in OCL. These rules defines the set of model elements which are mandatory, optional or forbidden in a concrete single diagram. Herewith the notion of a view in the concept space is reflected in the profile. Current work on the profile concerns its complete formalization and the definition of the transformation rules.

## 4. Example Specification and According Code Generation Aspects

Models, that were built using the design principles presented above, include a set of information regarding external and internal view on COs structure and their behavior, that can be considered to form a complete CO definition. Most design tools, that support a design method for distributed objects based software development end up supporting the developer at this stage. Our approach includes mapping strategies to component aware as well as component unaware middleware platform technologies. Component awareness here addresses the platform capabilities to support aggregation and deployment of software pieces that form building blocks for distributed

applications. The CORBA Component Model (CCM) [4] forms one candidate for such a platform. Component unawareness on the other hand means, that the platform technology supports a number of transparencies with regard to interactions between distributed objects, but doesn't provide a model of aggregating such distributed objects to deployable, identifiable components with implementation composition support. A platform that is made of CORBA 2.3 compliant products is an example for such a technology. Due to its mosts advanced state of realization, we focus here on the task to represent concepts of our design method in a purely CORBA 2.3 based environment.

The aspects of notation and code generation according to the design method we present here will be introduced using a hypothetical task which has to be fulfilled by a service designer summarized as follows:

*„An Interactive TV Service component shall be developed which provides a number of channels to clients. The channels combine audio and video data. The are two special channels, an advertisement channel and an initial channel, where all programs are for free. The service component shall be able to receive input from its clients in form of joystick and mouse events. The client component for this station shall also be designed, it must be able to receive the channels and to provide the mouse and joystick events which trigger some changes in the received channel. There has to be a possibility to obtain the actual costs for an ongoing connection of a channel. If the client and the server are not in the same domain, security to get this information is to be ensured.*"

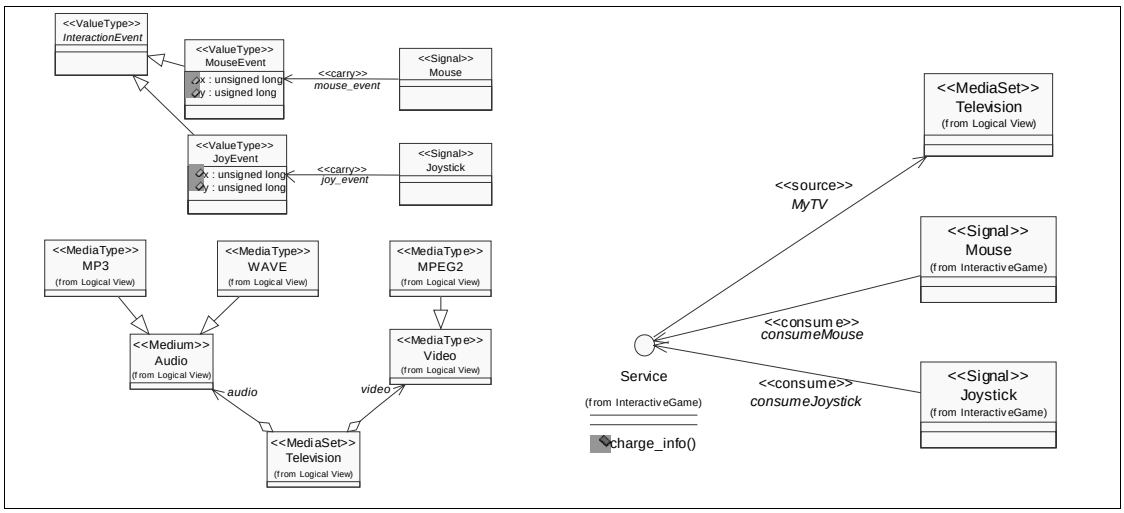


Fig. 1 Interaction elements for interface Service

### 4.1 Structural View Definitions

Fig. 1 shows the interface `Service`, which is defined for the TV example. This interface defines the concrete set of interaction elements for that example. As to be seen, the interaction elements are defined using some elementary definitions like data types or media types. It is the intension that those elementary definitions will be part of predefined packages which can be used for a particular design model. Especially for signal and continuous media communication those packages will be shipped together with supporting hardware, e.g. a Joystick comes together with a design model containing a description of the events it generates.

Given the definition of the interface `Service`, he structural view onto COs can be defined. Fig. 2 specifies two CO types, one representing a client object, that may require the services provided at interfaces of type `Service`, while the other provides such services.

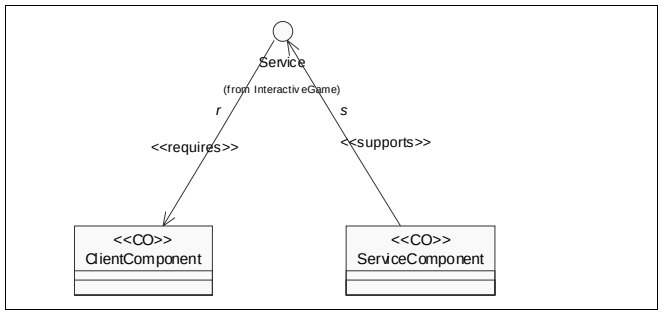


Fig. 2 Requires and supports relations

### 4.2 Instance View Definitions

The concepts of the structural view define, *which* services a CO may provide via its interfaces or which services it may use at interfaces of other COs. The concepts of the instance view define, *how* instances of a CO provide such services via instances of its supported interfaces or *how* a CO instance makes use of the services provided by another component instance. The corresponding term within the concept space is port of a CO, which may be either single port or multiple port.

In order to identify the ports of a component, they must have unique names in the scope of the component. In Fig. 3, ports for the `ServiceComponent` of the TV Service example are declared. One port (`channels`) is a multiple port, the others are single ports. Similar declarations are done on the client side for the required interface.

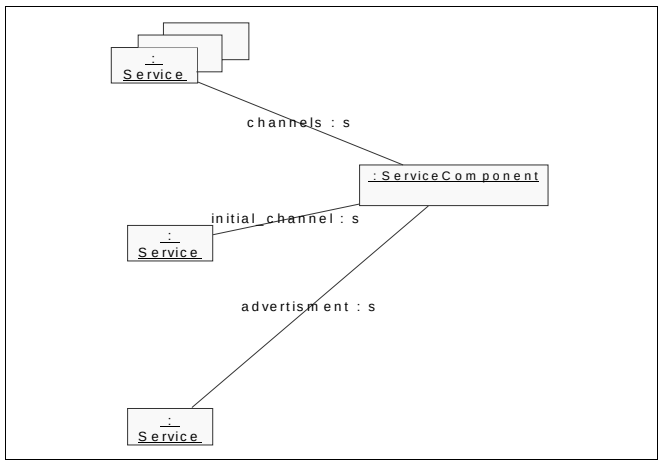


Fig. 3 Instance view for ServiceComponent

### 4.3 Implementation View Definitions

From the perspective of “black box modeling“ the structural and instance view would form a sufficient minimal set of definitions of a CO. However, as motivated before, the design method leads to an implementation and therefore some internal aspects of a component implementation are covered that realizes a COs behavior. In the TV Service example, the implementation of the `ServiceComponent` is structured into artifacts as to be seen in Fig. 4:

As already mentioned, there is an implicit interface for each CO were attributes and operations are provided. To implement those interaction elements, there shall be an artifact (`a3`) which is associated with the component directly. This special case is also to be seen in the figure.

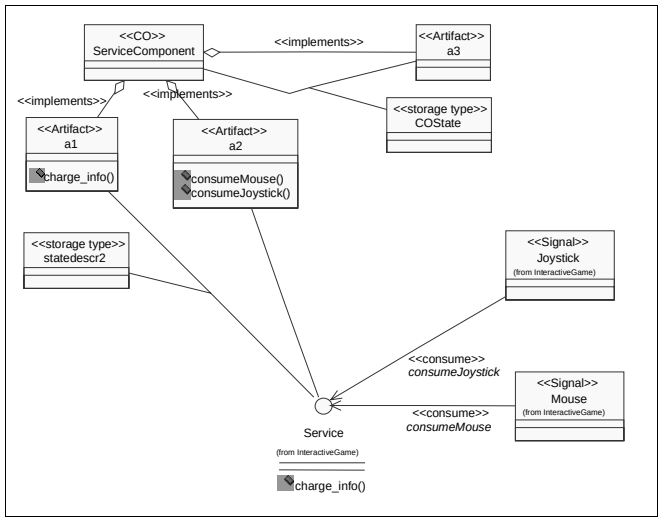


Fig. 4 Implementation view for ServiceComponent

### 4.4 Interaction View Definitions

The main design goal of the interaction view is to specify contracts for bindings of COs via their interfaces which have to be applied under certain conditions. Such contracts can be considered as being a QoS specification for the binding. As shown in the example, they do not only include traditional QoS aspects like response times or bandwidths but also required security or transactional policies. Contracts are related to *instances* of interfaces of instances of COs, they can be assigned both on the client and the server side (i.e. at a used or provided interface instance). To determine which contracts have to be used for a binding between a set of interface instances the concept of predicate is introduced.

Contracts itself are expressions on special QoS type specifications. Such QoS type specifications are part of the structural view and are associated to interface specifications. A contract can only be specified for an interface instance when there is an QoS type associated to the interface. Compared with traditional methods of QoS type and contract specification the introduced approach has some advantages:

- The QoS contract is assigned to bindings between interface instances not to interface types. This allows to distinguish several cases for the interface binding.
- It is clearly distinguished between the QoS type and the usage of that type in a contract.
- Contracts can be described for the client and the server part of bindings separately. This allows a negotiation of QoS at runtime between the participating instances.

Fig. 5 shows the binding specifications for the TV example: Assuming that there are the predicate `Intranet` and the QoS type `OperationalQoS` specified, the specification expresses the requirement to have a secure binding (security level > level_0) when the client is not in the same domain as the server. The contracts are expressed in terms of OCL constraints.

### 4.5 Code Generation Aspects

Generally, concepts of the design method not only map to definitions of interfaces that are supported by a particular component, but also to implementation language specific code skeletons of implementation artifacts, descriptions of the state of such artifacts and programming language specific code that form the runtime environment for a component instance. In particular, the CORBA Interface Definition Language (IDL) is used as a platform specific mapping target for the concepts of the structural view. An interface that is required or supported by a component of the

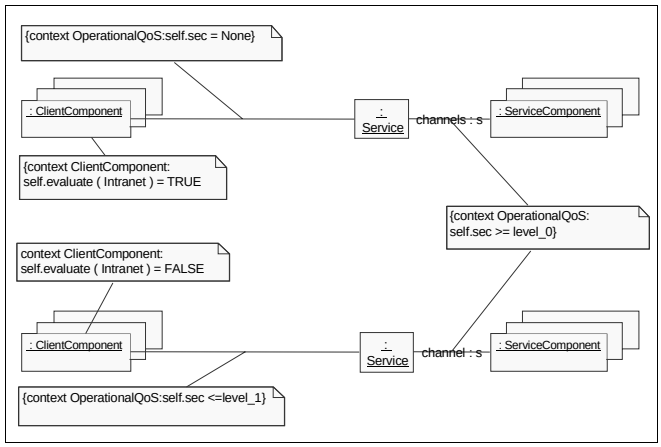


Fig. 5 Bindings for Service interface

design model maps to a set of IDL-interfaces, that support the different interaction kinds defined for the interface in the model. The resulting IDL interface definitions for a component's supported or required interfaces are referred to as implied interfaces. Beside the mapping to IDL interfaces, model elements are added to the design model, that are refined during the definition of the component implementation. The definitions of the instance view extend the implied interfaces definition by a platform view onto the CO itself, i.e. interfaces are generated that represent the CO and its ports. Also these definitions are provided through code generation as IDL interface definitions.

The concepts of the implementation view provide a model of a composition of artifacts that implement the services provided by a component via its interfaces. The platform specific mapping targets to implementation language definitions that represent the defined composition. To support this representation, the delegation design pattern [18] is applied to the mapping, and realized by delegator classes that couple the CORBA runtime system and the implementation artifacts. Through this approach, the realization of the component behavior is decoupled from the actual activation procedures for the implementation artifacts. The flexibility of this approach also allows the implementation class definition to be independent from the implied interface definition, in a way that an implementation artifact can implement only parts of the services that are provided through the defined interfaces. The implementation view also allows to assign persistency information to implementation artifacts. To represent this, the recently adopted Persistent State Service specification [17] is applied, and the Persistent State Description Language (PSDL) defined there is used to represent persistent state information of implementation artifacts.

The concepts of the binding view describe specific binding cases between component instances. A platform that is based on CORBA doesn't provide a view on the binding of objects, but provides hooks, were a so called ORB service can be plugged into, that e.g. may provide a binding negotiation service integrated with the ORB. The Portable Interceptors (PI) specification [5] defines such hooks in a portable way, i.e. interceptor implementations can be integrated with different ORB products in a portable manner. The PI technology, together with servant management capabilities of the Portable Object Adaptor (POA), is used to negotiate the binding case as specified by the binding view, and to negotiate the actual binding policies according to the constraints given by the model.

Whereas for the operational and signal based interaction kinds, standard CORBA mechanisms are used as a platform mapping target, the elements of the design model, that regard continuous media interactions, are mapped onto a proprietary multimedia content composition and delivery platform. Within the scope of that platform, multimedia content composition is treated as a process of describing a structure, how media are presented to a consumer. This process is modeled as a composition of meta information on media (content description), were the media data themselves

may be distributed over a network. Such meta information regard the content of the medium itself as well as the properties of its physical representation. Such a content description is taken as input to the content delivery task of the platform, that calculates a plan for the delivery of referenced media (content schedule) and delivers media data according to it in a quality of delivery adoptive way. The platform architecture conceptually supports the distinction between logical media flows and physical media data transmission as well as the openness to specific network and media presentation technologies. The platform concepts are prototypically implemented in a CORBA 2.3 environment and are presented in more detail in [3].

## 5. CONCLUSIONS

The definition of a concept space independently of a notation has turned out as very helpfull. Limitations on a modelling terminology which are often introduced by an early selection of a concrete notation could be prevented. With the selection of UML also a notation was found, which could be easily adapted to the concepts space. Nevertheless, some shortcomings have been identified, which mainly concern the unavailability of a so-called heavy-weight extension mechanism (i.e. introduction of new model elements). On the other side a smooth transition from the design models to a concrete implementation environment could be established. Although beeing not yet conform to the UML standard the extensibility mechanism of RationalRose could be applied for the implementation of our profile and of the transformation rules therey ensuring the applicability of the methodology for real-life development projects.

## REFERENCES


[1] Fischbeck, Fischer, Holz, Kath, v. Löwis, Schröder: *Improving the Development and Validation of Viewpoint Specifications*; Proceedings of FMOODS ’97
[2] Born, Hoffmann, Li, Schieferdecker: *Combining Design Methods for Services Development*; Proc. of FMOODS ’99
[3] Kath, Takita: *OMG A/V Streams and TINA NRA: An integrative Approach*; Proc. of TINA ’99
[4] BEA Systems et. al.: *CORBA Components - Volume I*; OMG doc. orbos/99-07-01
[5] BEA Systems et. al.: *Portable Interceptors*, OMG doc. orbos/99-12-02
[6] OMG: *OMG Unified Modeling Language Specification, Version 1.3*; OMG doc. ad/99-06-08
[7] Data Access Corporation et. al.: *UML Profile for CORBA*; OMG doc. ad/00-02-02
[8] OMG: *The Common Object Request Broker: Architecture and Specification, Revision 2.3.1*; OMG formal/99-10-07
[9] OMG: *A revised version of the Notification Service Specification*; OMG doc. telecom/98-11-01
[10] ITU-T: Rec. X.903/X.904 | ISO/IEC 10746-3/-4: 1995, Open Distributed Processing - Reference Model Part 3/4.
[11] Janssen et. al.: *HTTP-ng Architectural Model*; W3C Internet Draft; www.w3c.org
[12] Sun Microsystems, Enterprise JavaBeans TM
[13] W3C: *Extensible Markup Language (XML)*, version 1.0, Febr. 1998
[14] ITU-T Recommendation Z.100: *Specification and Description Language*; ITU-T, 2000
[15] ITU-T Recommendation Z.109: *SDL combined with UML*; ITU-T, 2000
[16] OMG: *Requirements for UML Profiles*; OMG doc. ad/99-12-32
[17] OMG: *Persistent State Service 2.0*; OMG doc. orbos/99-07-07
[18] Gamma, Helm, Johnson, Vlissides: *Elements of Reusable Object-Oriented Software*; Addison-Wesley ’99
[19] Szyperski: *Component Software - Beyond Object-Oriented Programming;* Addison-Wesley ’99
[20] Frolund, Koistinen: *QML: A Language for Quality of Service Specification*; Hewlett-Packard Laboratories
[21] Bonnet, Dubois, Efremidis, Leonardo, Malavazos, Vincent: „*Cooling the Hell of Distributed Applications' Deployment*“, Proc. of IS&N 2000
[22] Internet RFC 2046: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*
[23] http://www.rational.com