Domain specific modeling and analysis

Jacob, J.F.

Chapter 4

The OMEGA Component Model

Author: Joost Jacob

4.1 Introduction

In this paper we introduce a formal model of components as developed in the IST project OMEGA (IST-2001-33522, [OME]) sponsored by the European Commission. The aim of this project is the definition of a development methodology in UML for embedded and real-time systems based on formal techniques. The approach followed in OMEGA is based on a formal semantics of a suitable subset of UML 1.4 which includes class and state diagrams, Object Constraint Language (OCL), use case diagrams, and Live Sequence Charts ([DH01], an extension of UML's sequence diagrams). Some of the OMEGA members have been involved in the design of a component model for UML 2.0 that will be finished in the course of 2004 and the OMEGA component model has influenced the UML 2.0 component model; therefore the approach presented in this paper is compatible with the approach taken in UML 2.0.

The main rationale of our component model is to extend the above subset of UML as used in OMEGA with additional structuring and abstraction mechanisms which allow a modeling discipline and the application of formal techniques based on "interfaces". The basic idea of a component presented in this paper is that of a high-level software abstraction like a module which encapsulates its internal structure and which provides an interface specifying the exported (also called provided) and imported (also called required) operations and signals. Components can be hierarchically composed from basic components and relations between provided and required interfaces. Basic components are defined as sets of classes together with the provided and required interfaces. Components interact via Ports. In our model a Port is an object realizing an interface and/or depending on an interface of another component. Ports, like any other object-instances, can be created dynamically. In this sense our notion of a Port differs from the usual UML definition of an interface. Since we view components as a software abstraction, components themselves cannot be instantiated but only its Ports are instantiated. If there is only one Port instantiated for a component then this Port can be regarded as "the component instance" or "the component" and this phrase is sometimes used to make text more readable.

We show how our component model provides a general framework for the classification of and relationships between the UML concepts mentioned above as used in OMEGA, by adding *component diagrams* and *architectural diagrams*. Architectural diagrams are used to describe certain run–time properties of components which are independent of the actual deployment on a certain platform. There is an analogy between component diagrams and class diagrams and likewise between architectural diagrams and object diagrams, and this analogy can be used to design our new diagrams using CASE tools that do not support the new component model diagrams yet. Finally, we discuss the possible usage of the OMEGA component model for verification purposes.

The first version of our component model was presented as an OMEGA milestone document in June 2002.

4.2 The Component Model

In this section, we present a meta–model for our notion of components. In this meta–model we extend standard UML entities, the building blocks, like class and interface. Since we only use a few UML entities it will be easy in the future to make the meta–model compatible with UML 2.0 [OMG] once that has reached a stable version, and to fit it in with the new MOF [MOF] version that is under development. To avoid confusion with existing UML entities, in the rest of this paper we will use a capital for the first letters of the names of entities that are our extensions to UML.

Component Models as high–level class diagrams

Our starting point is a model of components which provides a high-level software abstraction like that of a *module* which encapsulates its internal structure and which provides an interface specifying the exported (also called provided) and imported (also called required) operations and signals (as defined by the OMEGA kernel model language in [OME] and [DJPV03]. The interface of a component is structured into *Component Interfaces.* Component Interfaces are like ordinary UML interfaces but they have to adhere to the usage rules for Component Interfaces we specify below in this section. A Component Interface consists of a collection of signatures of operations and signals, but contrary to ordinary UML interfaces Component Interfaces do not contain attributes. In comparison with UML diagrams, a component model is similar to a class diagram. Later, in Sect. 4.4, we will introduce diagrams for components, so-called architectural diagrams, that are similar to UML object diagrams.

The underlying class diagram

In an OO setting there is always a class diagram underlying an application. The same is true for a component based application designed with our component model. In the OMEGA deliverable D1.1.2 we have presented a formal reduction from a hierachical component model to a flat class diagram. In this paper we will present in Sect. 4.5 a formal justification of our component model in terms of a compositional trace semantics and its corresponding logics.

Introducing Ports

Component Interfaces are grouped into Ports. Component Ports correspond with special purpose classes inside components that provide the only interaction points between components. At runtime, all communication between components is going via instantiated Ports. In our component model, a Port is used as a class, and it is also used as a type specification for one or more runtime objects. Ultimately these runtime objects are instances of classes in the underlying class diagram, because our model is designed in an OO setting.

Why is an object-oriented component model useful?

The underlying class diagram can possibly be huge; this is one place where a component model can be useful because one component can abstract from many classes. Also, it is possible to design a component with a Port, and to be specific about the services the Port requires and provides, without having to specify already exactly what class will used for instantiating the Port; this supports better top–down design methodologies. Our component diagram groups classes in an underlying class diagram into components, and it groups associations in that class diagram into Ports and Component Interfaces and the associations and connections between them. As such it provides a high-level view of a class-based application which is both suited for top-down design and compositional analysis.

UML 2.0

Syntactically the components in our component model are much like the components in the UML 2.0 submission by U2Partners in September 2002 and in Januari 2003. One of our main objectives in OMEGA is the development of an OMEGA component model which is compatible with their UML 2.0 submissions. But there are some semantic differences that will appear in the rest of this paper. We can mention here already one of the most important differences: in the submissions by U2Partners a component *itself* is instantiable whereas in our model it is the Component *Ports* that are instantiated (as instances of UML classes); this way the component provides a conservative extension of the underlying object-orientation so it can remain a software abstraction. Another difference is that in order to keep our model small, simple and elegant, we do not explicitly model connectors and therefore we have not defined new UML entities for connectors. This provides a user of our component model with a choice: the user may decide to extend our model and use the UML 2.0 connectors, or the user can choose to model connectors as components themselves.

4.2.1 Blackbox Components

A Blackbox Component gives a *blackbox view* of a component in a blackbox diagram. Inside a Blackbox Component nothing is visible, only the Interfaces of the component that are to be used in a design outside of the component are visible, and the grouping of these Interfaces into Ports is visible.

The meta–model for a Blackbox Component is contained in Fig. 4.1; the Basic Component and UML Class and Component System boxes and their relations do not belong to it but will be introduced later. As can be seen in the figure, we have modeled a Blackbox Component as a specialization of a UML Classifier. In a future MOF version a component could well be better modeled specializing another (future?) MOF construct that is more suitable for our purposes, or perhaps more than one construct. For now we are basing our meta–models on UML 1.4 and therefore we use the Classifier.

The same goes for the other specializations from UML entities we use in the figure, again for now we make do with UML 1.4 entitities. Adapting the meta–model to the next MOF or UML version should pose no problems that cannot be overcome easily.

In Fig. 4.1 we can see that Blackbox Components can have several Ports, and a Port can have several Provided Component Interfaces and several Required Component Interfaces. The other way around, a Component Interface is associated with one Port, and a Port belongs to one Blackbox Component.

A Blackbox Component is drawn as a box, optionally with the UML 1.x component symbol in a corner to make it extra clear that the box is a component. Ports are drawn as small squares on the edges of a component box. In the blackbox view the association between a Port and Component Interfaces can be shown with the “lollypop” notation, or with UML dependency and UML realization associations to expanded interfaces (boxes with the name of the interface and a stereotype indiciation and a list of services). This is in accordance with standard UML 1.x notation; an example with the two notations is shown in Fig. 4.2. We therefore propose to extend the Kernel Model Language with UML realization associations. Note that these associations do not affect the semantics, i.e., they encode only static information which can be checked by a preprocessor.

There can be UML dependency relations from Provided Component Interfaces to Required Component Interfaces on the same component. This means that if a user wants to use one of the services of the Provided Interface, the Required Interface must be realized, or else the service is not

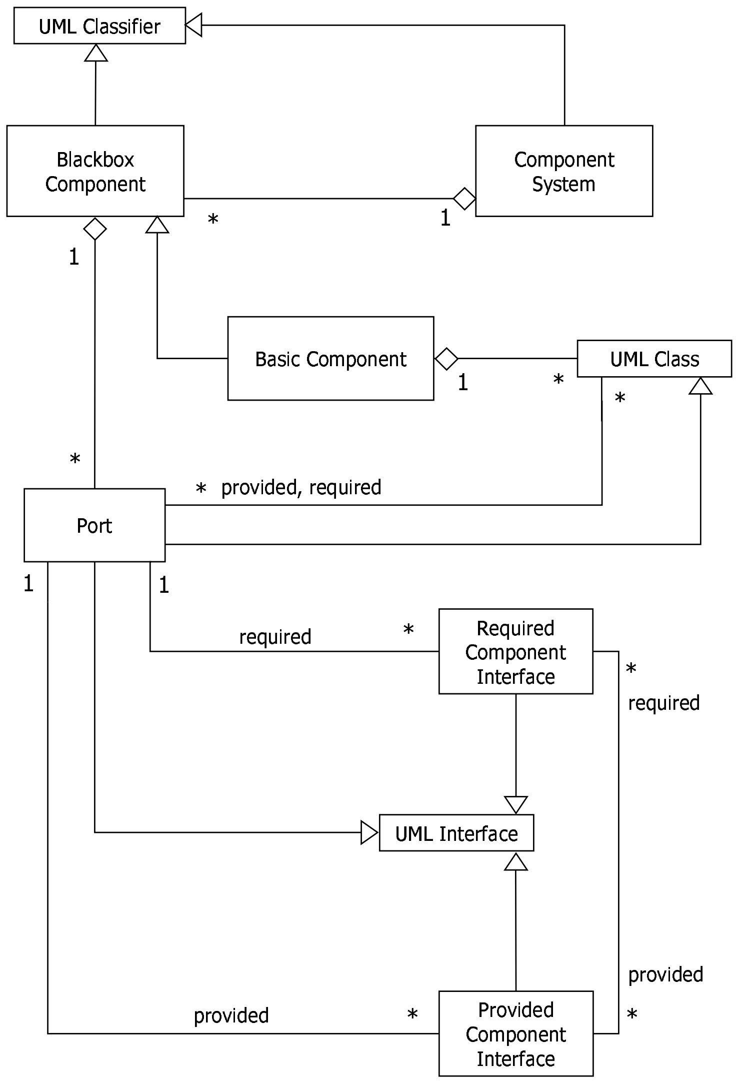
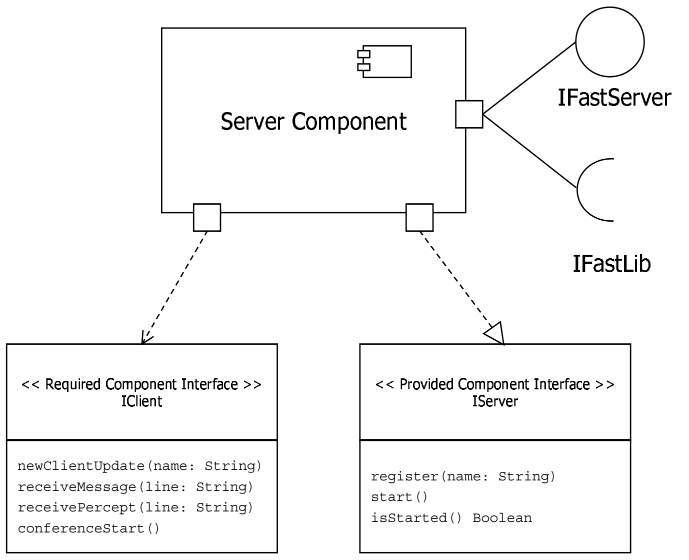


Figure 4.1: The combined UML Meta-Model for our component models


Server Component
IFastServer
IFastLib
<< Required Component Interface >>
IClient
newClientUpdate(name: String)
receiveMessage(line: String)
receivePercept(line: String)
conferenceStart()
<< Provided Component Interface >>
IServer
register(name: String)
start()
isStarted() Boolean

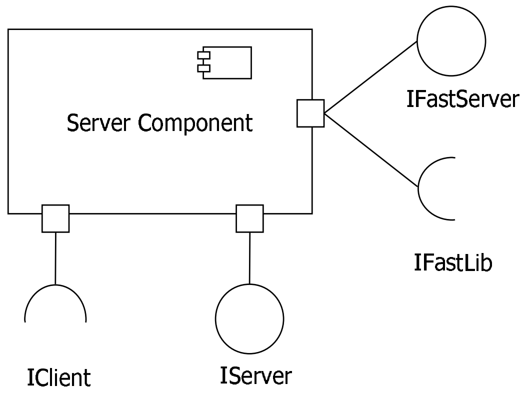


Figure 4.2: A Blackbox Component; top: with 2 expanded interfaces, bottom: with all interfaces in elided form.

guaranteed. Note that we see here a coupling between a Provided and a Required Interface on the *same* component. This special dependency expresses the fact that the same object that provides the services in the Provided Interface, depends on the services in the Required Interface. Such an object is a Port, and this special case is one of the reasons to introduce Ports.

4.2.2 Basic Components

Blackbox views of basic components form the basic building blocks of the hiearchical composition of components. The structure of a Basic Component consists of a set of classes and their relations (as defined by the OMEGA kernel model language), a subset of some of its classes associated with its Ports, a set of (Provided and Required) Component Interfaces which are associated with its Ports (same as with Blackbox Components), and, finally, connections between Provided and Required Component Interfaces. There are no other components inside a Basic Component. Some of the classes inside a Basic Component have nothing to do with Ports, some of the classes are helper classes that help realize Ports, other classes have (part of) their interface(s), expressed in Component Interfaces, exposed to the outside world via Ports. That ouside world consists of other components, as discussed in Sect. 4.2.4.

Figure 4.1 shows the UML meta–model of the representation of the internal structure of a Basic Component, together with that of the Blackbox Component we saw earlier. Just like a class definition in OO has a class name, we define a component model name for a component model. A model for a Basic Component and a model for a Blackbox Component refer to the *same* component model if their component models have the same name. A Blackbox Component model and a Basic Component model with the same name will have the same Provided Component Interfaces and the same Required Component Interfaces.

4.2.3 Extensions to the OMEGA UML subset discussed so far (Fig. 4.1)

Provided Component Interface

A Provided Component Interface is modeled as a specialization of a UML interface. The Provided Component Interface can be realized by a Class via a Port, or by one of the Ports of a Blackbox Component (thus hiding, encapsulating, classes in a Basic Component).

Required Component Interface

A Required Component Interface is a specialization of a UML interface. The Required Component Interface can be required by a Class via a Port, or by one of the Ports of a Blackbox Component.

Port

A Port is a specialization of a both a UML Class and a UML Interface. A Port can be regarded as a UML Class, whereby the interface of the Class is known but the name of the Class is unknown. Creating an instance of a Port means creating an object with a known interface, but without the need of knowing the class of the object.

One Port can group several Component Interfaces, both Required and Provided. More than one class inside a Basic Component can be involved realizing a Port. More than one class can require services from outside the component via a Port. In Basic Components there can be dependency relations and realization relations from classes to Ports, as shown in Fig. 4.1 with the *required* and *provided* rolenames respectively. It is possible that one or more classes realize a port and one or more other classes depend on the same port. A Blackbox Component, and so by inheritance also a Basic Component, can have several Ports. A designer can give names to Ports so they can be identified when the same Port is appearing in different diagrams.

Blackbox Component

A Blackbox Component is a model for a component where only its Ports and the Provided and Required Component Interfaces are visible from the outside. It is a specialization of a UML Classifier.

Basic Component

A Basic Component is a component consisting of classes and their relations as defined in the `OMEGA` kernel model [OME]. Some of the classes are associated with Ports: they can depend on them or they can realize them. The Basic Component is a specialization of a Blackbox Component.

> A Basic Component inherits Ports with their Provided Component Interfaces and Required Component Interfaces from a Blackbox Component.

We would like to give a few extra remarks about the associations in Fig. 4.1.

A class can depend on several Ports. Since Ports inherit from UML interfaces we can draw dependency relations from classes to Ports. A class depending on a Port implies that that Port depends on a Required Component Interface. Via a dependency relation from a class to a Port a class exports information about its implementation in terms of required services. Our Component Interfaces inherit from UML interfaces so, when drawing a component diagram, we can use the UML dependency relation from Ports to Required Component Interfaces outside of the component, or the corresponding lollypop notation. These notations are the same as for the Blackbox Component. For every Required Component Interface there will be one Port depending on it. For every Port there can be several classes depending on it. In the case of Basic Components the same special dependency relation from Provided Interface to Required Interface on the same component is possible like mentioned in the case of Blackbox Components.

A class can realize a Port by itself, or it can realize "part" of the Port: there can be more than one class realizing the same Port. A class can also be involved in the realization of several Ports. In designing component based systems this is where the designer can abstract from the underlying class diagram; future versions of the component design can use a class diagram that is different from earlier versions, corresponding to a new version of the implementation of the component. In the diagram we can draw realization relations from classes to Ports. A class realizing a Port implies that that Port realizes a Provided Component Interface, drawn with a UML realization relation from a Port to a Provided Component Interface that is outside the component, or with the lollypop notation. For every Provided Component Interface there will be one Port realizing it.

Next we describe some further aspects of the classifiers in our component model.

The outside of a Basic Component is drawn like a Blackbox Component, the inside of a Basic Component uses UML 1.4 syntax for class diagrams, with dependency relations and realization relations from classes to Ports. Figure 4.3 shows an example Basic Component. It models a Client component that needs services from a Server component via the IServer interface. The Client offers services to outside components like `receivePercept` which is used to send data to a Client. The SWC class inside the Client provides the clients' services in this specific application. The XMLRPC class inside the Client is for making a connection with a Server component via its IServer interface: it provides the protocol used between components and it establishes proxies when they are needed.

Both classes and components can engage in provided–required relationships, since a class can be a Port. Here we call the interfaces between them *Component Interfaces* to make clear we are talking about components and to ensure the interfaces adhere to the rules we give for Component Interfaces in this section. There can be classes *and* components depending on the same Required Component Interface via the same Port.

A Required Component Interface can not depend on something inside a Basic Component. That would be a design error since the component can supply the needed services by itself.

An interface (an ordinary UML interface, not a Component Interface) in the class diagram inside the Basic Component that depends on something from outside, should be modeled as a Component Interface. The designer is free to allow class libraries from outside that can be used inside a Basic Component, but this would be a strange design: it would raise the question why the designer did not turn the interface into a Component Interface. Although it would be a design some would frown upon, we do not want to go as far as to forbid it completely. There can be practical considerations, for example it could be difficult to use an existing library in a component framework setting because there is not enough library source code available.

4.2.4 Component Systems

Now that we have defined Basic Components and Blackbox Components, we can finally define components that have other components inside: a Component System can be viewed as one component but with an internal structure consisting of Blackbox Components. This recursive definition gives us the hierarchical structure we need for modeling component based applications.

We use diagrams for Component Systems to show how components are used together, and to show what components need services from which other components. In Component System diagrams only components, their Ports, and their Component Interfaces and their connections are shown, using the

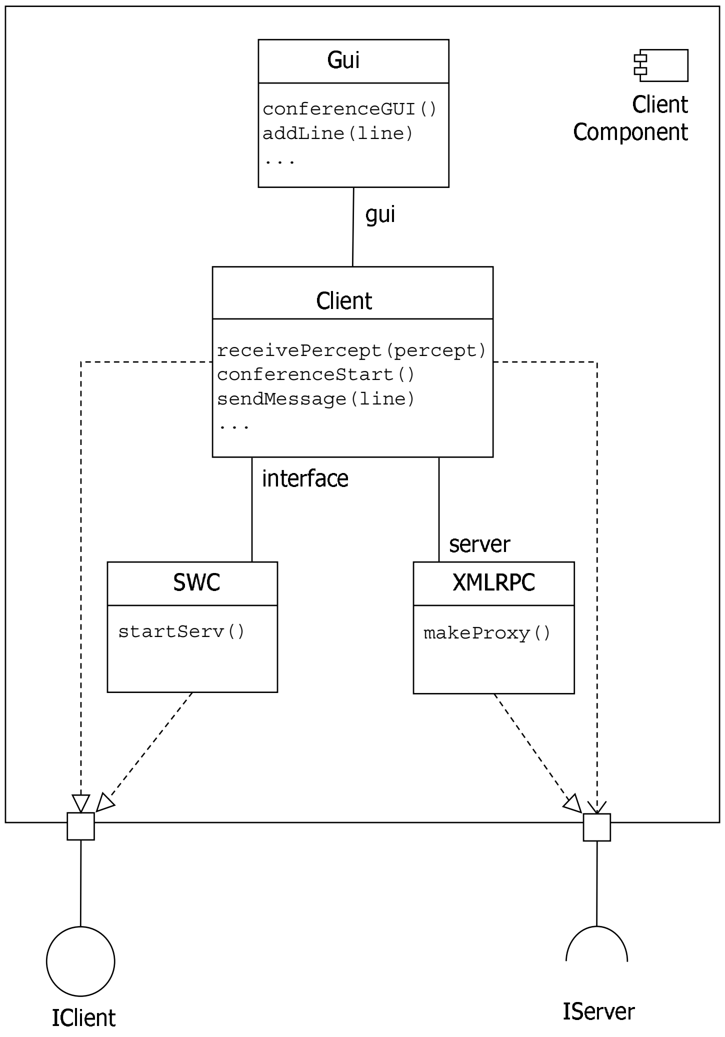


Figure 4.3: A Basic Component, with a class diagram inside

notation of Blackbox Components extended with notation to connect Blackbox Components. Such connections are given by dependency relations from a Required Component Interface to a Provided Component Interface, necessarily crossing a border between two components. There should not be a dependency relation from a Required Component Interface to a Provided Component Interface on the same component. This would mean that the Required Component Interface is depending on services from other components while the component can provide for these services by itself, so the Required Component Interface is redundant. A dependency relation of a Provided Component Interface to a Required Component Interface on another component can not readily be given a useful meaning: we consider such a relation a syntax error.

A Component System is a specialization of a UML Classifier. As described in the section about Blackbox Components, a future UML version could well give us a more suitable entity to specialize from. A Component System is also a collection where its internal Blackbox Components can be seen inside. This is shown in Fig. 4.1 with the generalization association to UML Classifier and the composite association from Component System to Blackbox Component. A Component System also has Ports via the Blackbox Component inside. This means that some of the Ports of its Blackbox Components are exported in order to serve as its interaction points. In fact, a Component System also has a *blackbox view*: the Component System as a whole can be seen as a Blackbox Component that has the same *name* as the Component System and the same Provided and Required Component Interfaces and the same Port names, but nothing can be seen inside. Blackbox views of Component Systems provide levels of abstraction: a Component System can contain Blackbox Components that are Component Systems themselves.

Figure 4.4 shows an example Component System. It models how the Client component from Fig. 4.3 is connected to a Server component. The designer has also decided to turn the Graphical (GUI) User Interface part of the Client (which was just a class called `GUI` in 4.3) into a separate component, so the GUI of the client can be changed and replaced easily. The GUI component forms a Component System together with the Client component that could also be viewed as one "GUIClient" Blackbox component. Also, all the components in Fig. 4.4 together form a Component System.

Inside a Component System, a Provided Component Interface of one component can provide for several Required Component Interfaces of other components, and a Required Component Interface can depend on several

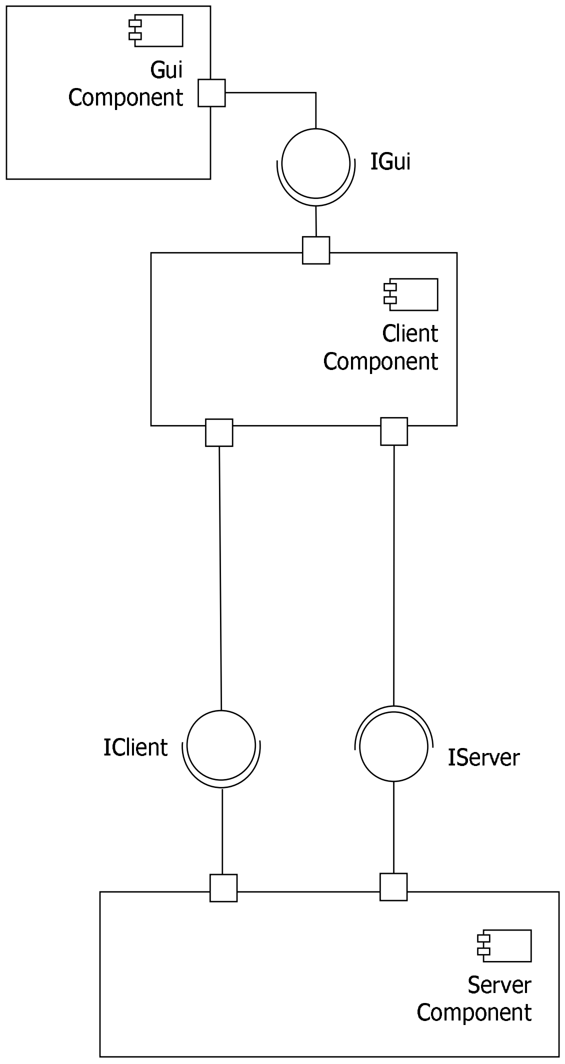


Figure 4.4: A Component System

Provided Component Interfaces.

Figure 4.1 shows the meta–models for Basic Components, for Blackbox Components, and for Component Systems in one figure. They have been combined into one figure so the reader is given a good general overview.

What is not specified in the meta–model however, is the following important condition which ensures encapsulation: The signatures of operations and signals in a Component Interface should only contain standard classes and data–types from the implementation language (for example the OMEGA kernel model language) and classes that are exported as Ports. Note that thus encapsulation is ensured because we do not allow inheritance relationship across component borders (we only allow dependency relation between Required and Provided Interfaces).

4.3 Runtime Behaviour

In OMEGA we associate with each class a statechart which describes the runtime behavior of its instances. Because ultimately an OMEGA component model can be flattened to its underlying class diagram (this reduction is formally worked out in an OMEGA deliverable), this association completely defines the runtime behaviour of a component. It is important to observe that here we are referring to the runtime behaviour which abstracts from the actual deployment on a specific execution platform.

The labels on the arrows in these statecharts contain OMEGA action language and they are of the form `[guard] trigger / action` where guard is a boolean expression, trigger is an event or a method name with its parameters and action is a primitive action in the OMEGA action language. These primitive actions use standard OO dereferencing with the dot notation and are of the form `a := a0.a1`, `a0.a1 := a`, `return := a` and other simple statements; see the OMEGA kernel model document [OME] and [DJPV03] for a complete enumeration.

In our model the required services of a component are specified by means of interfaces, as described formally in the meta–model. Acquiring an object that provides the functionality of a component with interface `I`, requires the instantiation of a class whose interface is known but not its definition (which is given in another component). Therefore, in OMEGA we have extended the UML action language used in statecharts with this notion of "instantiable interfaces", that is, in the action language we allow assignments `x :=`

`new(I)`, where I is a (required) interface. This way we can make instances of classes that are defined in other components, but without the need to know the name of the class in the other component (which would be impossible in the case of a future implementation of the other component). There are several ways to actually implement this scheme, in Fig. 4.3 it is the SWC class that makes sure that the correct class is instantiated in a Client component. This class is simply called Client in the figure, but in a future version it could be a class called NewClient.

As such we are instantiating a class but we only know the interface (`I`) of the class, we do not know the name of the class nor its implementation. However, in the case of one *complete* component application it *is* known which class implements `I` so we can simply compile `x := new(I)` into the corresponding `x := new(C)`, where class `C` implements `I`.

Our component model thus abstracts from the underlying component framework (for instance CORBA). To provide services of class `X` to other components in a component framework, the designer can assume that a class `Y` exists that does introduce the services of `X` to the component framework. This class `Y` is a class that realizes a Port in the model. There are several ways class `Y` can do this: it can accept an object that is an instantiation of class `X` as a parameter to one of its methods and delegate the desired services to this object; or it can use a "mixin" technique that extends the interface of class `Y` with the desired services of class `X` and instantiate a new object of type `XY`; or it can create a new object of class `X` and delegate desired service calls to this object; or it can use another mechanism.

To summarize, we do not have an explicit notion of "instances of components" but we only have instances of Ports. Of course it is possible to design software in such a way that objects, referenced by variables in the source code, are created that act like instances of components. But we do not enforce creation of component instances: if the designer wants to model a component as a software abstraction only, it is possible.

4.4 Architectural Models

Architectural diagrams show component interaction configurations. They are snapshots that can be used to describe the initialization of a component system, invariant properties of the configuration, and others useful runtime characteristics. In OMEGA we will use a very restricted subset of OCL for expressing the semantics of architectural diagrams formally. In architectural diagrams components interact by means of Ports. These Ports are different from the Ports in a component model, the emphasis here is on *objects* with a specific interface, not *classes*. If there is a possible confusion then we will call the Ports in architectural diagrams Runtime Ports, and Ports in component diagrams Component Ports. In an architectural model the Runtime Ports can be viewed as named interfaces. Although normal OO interfaces are not instantiable, in our model, as discussed previously, Component Interfaces are instantiable via their Component Ports, resulting in Runtime Port objects: this gives us the possibility to model component interaction like other OO interactions. In an architectural model, Runtime Ports are instantiated interfaces. In an actual implementation a Runtime Port can be an object that delegates to several other objects, or it can be a channel–like object with an address and location; what choice is made exactly is not important for the design: it is an object that realizes a Component Interface. In our model we define interaction between Ports as standard OO interaction.

Figure 4.5 shows an example architectural model, together with a component model above and an object diagram below. In the component model can be seen that `Component B` requires services of `Component A`. In the architectural model can be seen that there are, at some point in runtime, exactly two Ports of `Component B` connected with `Component A` and they are using the same services but from different Ports. The connections are *directed* from requiring to providing Port. In the case of two Ports that use services from each other an undirected connection can be shown by drawing a line without an arrow. The Ports of `Component B` are instances of `PB`, the Ports of `Component A` are instances of `PA`. Such a configuration can be specified with OCL, but the architectural model is also useful: it is easier to draw a picture like this than to have to learn OCL. The bottom diagram in Fig. 4.5 shows an object diagram that corresponds with the architectural model above. It shows the objects that realize the Runtime Ports. This makes it clear that the components in an architectural model are not software abstractions but collections of objects.

Figure 4.6 shows the meta–model for architectural models. For the connections between Ports we use UML associations, we are awaiting the next MOF to decide what the final meta–model will look like, viz. the `Object Collection` entity.

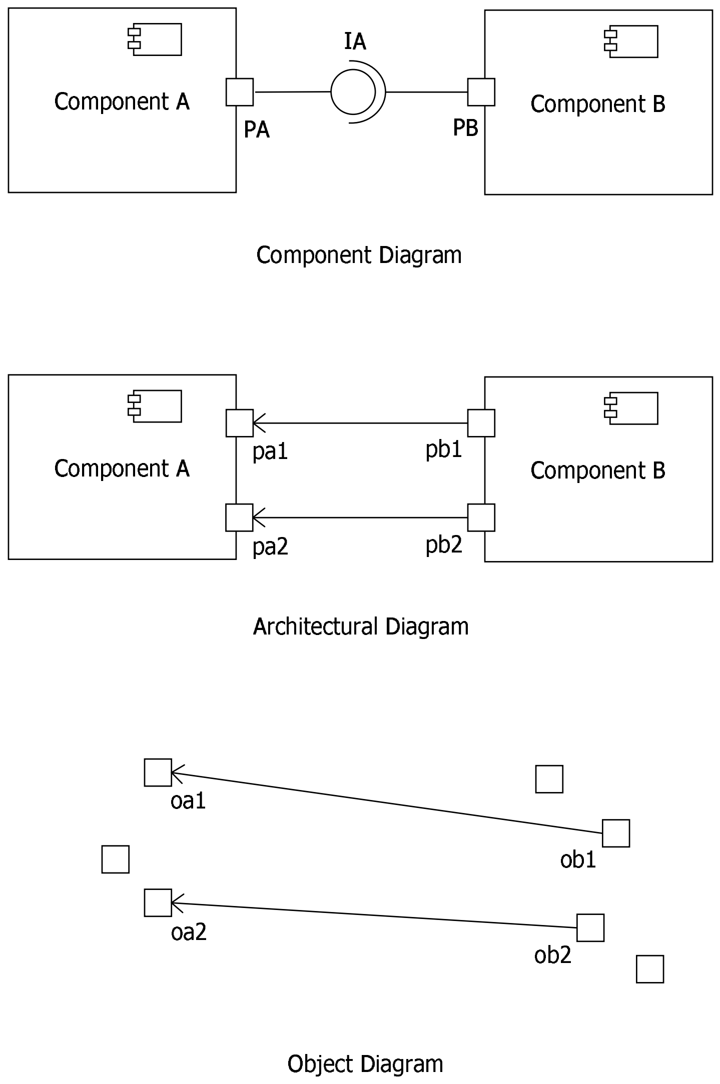


Figure 4.5: The same application modeled with a Component Diagram, an Architectural Diagram and an Object Diagram

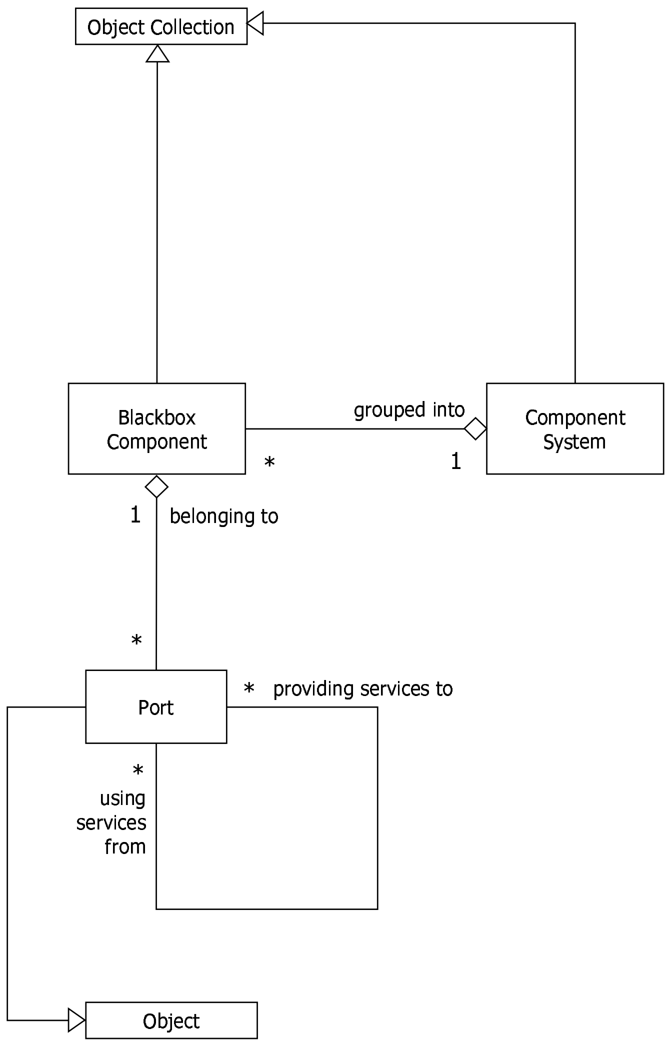


Figure 4.6: The Meta–Model for architectural models

4.4.1 Overview

As mentioned in the Introduction, in the OMEGA project we have chosen a subset of UML 1.4 so we will be able to use formal techniques. This subset consists of class diagrams and object diagrams for structural modeling, and statecharts and OCL for behavioral modeling. We also use Live (LSC) Sequence Charts [DH01] in OMEGA but they are not part of UML. The subject of this paper is about the extension of this OMEGA UML subset with components. An overview of all the UML diagrams we now have is available in Table 4.1.

Definition	Deployment	Behaviour
Component Diagrams	Architectural Diagrams	Specification: Statecharts, OCL
Class Diagrams	Object Diagrams	Implementation: Statecharts

Table 4.1: UML Diagrams in the OMEGA project

In our component model we define component diagrams that relate to architectural diagrams like class diagrams relate to object diagrams. When designing OO software, both class diagrams and object diagrams are useful; they serve different purposes: with class diagrams the designer gives a definition view, with object diagrams a runtime view is given. Likewise, both component diagrams and architectural diagrams are useful.

Summarized, *class diagrams and component diagrams are used for modeling definitions; object diagrams and architectural diagrams are used for modeling configurations.*

If a class diagram is complete and if there are constraints added with a powerful enough constraint language like for example OCL, then all corresponding possible object diagrams can in theory be derived from it. A design goals of our component model was to offer similar diagrams and possibilities for modeling components.

To model behaviour in UML statecharts can be used. They are associated with class diagrams and they give an implementation. On the *component level* we can also use statecharts to model behaviour. To be able to model behaviour on the component level requires a different action language than that used in statecharts associated with classes: statecharts associated with components describe the interaction between, and coordination of, different objects, whereas statecharts associated with classes describe the behaviour of one object. On the component level we use statecharts together with OCL to specify behaviour: this is indicated in Table 4.1 in the right–most column, with statecharts in the top row for *Specification*, and statecharts in the bottom row for *Implementation*.

Table 4.1 gives a classification of the UML diagrams we use for modeling components and their relation to the existing class diagrams, object diagrams and statecharts. Together with the explanations in this paper we hope this sufficiently answers often heard questions from users like “When to use what UML diagram?”

4.5 Trace Semantics

In order to provide a semantic basis for the compositional verification of components, in this section we briefly outline the formal trace semantics of components which describes the external observable behavior of a component as determined by its ports. OMEGA deliverable D1.1.2 describes a formal operational semantics of UML class-diagrams where the behavior of the object instances of each class is described by a statechart ([Har87]). This semantics abstracts from the actual deployment unto a specific execution platform. It is formalized in terms of a translation relation on object-diagrams which specify for each existing object the values of its attributes and the values of some system variables which encode some relevant control information (such as the current state in the associated statechart).

On the basis of this operational semantics for UML class diagrams we can define inductively the trace semantics of a component. For basic components, the internal structure of which is given by an UML class diagram, we extend the above transition relation to a labelled transition relation

$$\sigma \xrightarrow{\lambda} \sigma',$$

where σ and σ' denote object-diagrams which represent the internal object-structure of the component before and after the transition and λ is a label indicating an internal computation step or an external event. An internal computation step is indicated by τ. An external event is of the form

$$o.m(o', p_1, \ldots, p_n),$$

where

- o denotes the callee of the event,
- o' denotes the caller,
- $p_1, \ldots, p_n$ denote the actual parameters, and finally,
- m denotes the kind of message.

Note that adding the caller as an explicit parameter, together with the encapsulation condition, implies that all interaction between components is via their Port instances. That is, we do not allow an internal object of a component (i.e., objects that are not instances of a Port class) to call the provided services of a Port instance of another component.

For technical convenience only, we restrict in the current presentation to messages of the following kind:

- op, which indicates the invocation of an operation call,
- $return.op$ which indicates the return of an operation call, i.e., $o.return.op(o', v)$ denotes the return of a call from o' to o with return value v.

Object-instances are denoted by pairs of the form (id, I), where id denotes the identity of the object and I denotes its interface. An object is external (to a given component) if its implementation is not known, that is, if its interface is a Required Component Interface. For an internal object we identify its interface with its class using existing UML. In an external event either the caller of callee denotes an external object.

The above transition relation generates the traces of external events of a basic component. The global behavior of a system of components $Comp_1, \ldots, Comp_n$ we can now describe compositionally in terms of the traces of external events of its components by means of a projection operator: Given a global trace of events t, a component $Comp$, the trace $Comp(t)$ denotes the subtrace of t consising of external events involving port-instances of $Comp$. More specifically, we have also to rename the identity of an external object (id, C) to (id, I), where I is the Required Component Interface of $Comp$ provided by port-class C (defined by another component). It is important to observe here that at the level of a component system the (high-level) dependency–realization relations between Component Interfaces provide information about which are possible events. Namely, an event like $o.m(o', ...)$ is possible if $o = (id, C)$ and $o' = (id', C')$ implies that there exists a connection between the ports C and C'.

Definition 1 *Given a system of components* $\mathcal{C} = \{Comp_1, \ldots, Comp_n\}$ *we define*

$$Trace(\mathcal{C}) = \{t \mid Comp_i(t) \in Trace(Comp_i), i = 1, \ldots, n\}.$$

Note that $Trace(\mathcal{C})$ specifies the global behavior of the component system $\mathcal{C}$. We can define the externally observable behavior of the blackbox view of $\mathcal{C}$ in terms of a hiding operator which removes all internal events.

The above trace semantics forms the basis of a corresponding trace logics for specifying invariant properties of the traces of components (see also OMEGA deliverable D1.2.1). We are working on a tool based on the semantic tableaux method which allows to check the compatibility of the trace-invariants of the components in a system in terms of a logical formulation of the above compositional definition.

4.6 Modeling with Components

In this section we discuss more practical aspects of modelling applications with components in OMEGA. In the OMEGA User Guide the concrete syntax for component models can be found, here we suffice to say that in the absence of a CASE tool that supports UML 2.0 components, the correspondence between component diagrams and class diagrams gives the possibility to use class diagrams to model components. It will give the user a little more administration to do to remember which diagrams are for components and which are for classes. Likewise, object diagrams can be used to model architectural diagrams.

The notion of an interface as specifying a set of provided and required operations (or signals), respectively, supports a development process of component-based software systems in UML that distinguishes two main levels of abstraction, promoting a separation of concerns between the external communication of data and the internal processing of data. At the higher level of abstraction, a system is described in terms of the interactions among its components, abstracting from their actual internal implementation. This level provides the black-box view of a component. The lower level concerns the modeling of the data-processing aspects within each component. The resulting *hierarchy* object-class-component provides a natural and powerful scheme for distribution and abstraction, hiding and structuring the complexity of large distributed object-oriented software systems. More specifically, the dynamic creation of any number of port-instances allows a component to interact in a really distributed manner. This is to be contrasted with the run-time notion of a component as a group of objects associated with an instance of an active class which share a single thread of control and an event queue of asynchronous signals.

The additional structuring and abstraction mechanism provided by the notion of component allows a considerable simplification of an underlying kernel model language like in OMEGA, which basically consists of removing the distinction between active and passive classes. More specifically, every instance of any class has its own single thread of control and its own event queue. Acceptance of signals and operation calls by an object are defined only in terms of the local state of the object itself. Objects are grouped together only by means of the static structuring mechanism of components.

This simplification of the OMEGA kernel model language (and its semantics) allows both for more transparant models and more efficient verification techniques. The additional complexity provided by components then can be dealt with by means of the application of compositional verification techniques.

4.6.1 Examples of software developed with the component model

We have developed several applications with our component model to see whether it is useful practice. New versions of the component model reflected the experiences with the designs. Most figures in this paper are based on an example called "Conference". With this system users can have a distributed conference where they communicate with each other by typing messages, somewhat like IRC chat on the internet. The system consist of a server application and a client application. The central server of a conference can be setup by any of the users and is accessible at a HTTP URI via XMLRPC. This means that the client application that is used to connect to the server can be written in any programming language. We have example clients written in Python and in Java. As another example the OMEGA partner FTRD has modeled their OMEGA application with our component model.

Using the component model turned out to be a natural and intuitive way of designing software. The software engineer can concentrate on high level system designs first and design lower levels later. Of course this could also be done with a class hierarchy but there the designer has for example no "instantiable interfaces" (our component ports), forcing the designer to make decisions about class names and class hierarchies and the like much earlier in the design phase.

The example applications are available at our OMEGA component model website [Jacc].

4.7 Conclusion and related work

In this paper we have presented a model for components to address architecture and component based development. The main idea is that a component is an abstraction, like a class or a module, A component is a grouping of classes, some internal and others, the so–called Ports, denoting interaction points with the component environment. Only Ports are visible to the environment. Each Port is attached to a set of provided and required interfaces.

Components are used in two type of diagrams: *component diagrams* and *architectural diagrams*. Component diagrams are for describing the structural dependencies among the provided and required interfaces of components in a system, while architectural diagrams are for the description of the run-time architecture of the system. In architectural diagrams Port instances are linked together by means of UML associations which indicate that the connected Port instances know each other.

Considering component as an abstraction of its internal parts, in contrast to the concept of component used for deployment in UML 1.4 [SP99], implies that components are not units of instantiation and do not need to have a unique run-time identity. Moreover, having Ports as instantiable interfaces, in comparison with the recent component model proposed by the U2 partners for UML 2.0 [U20], has the advantage of permitting the existence at run time of multiple Ports with the same set of interfaces per component, each Port attached to the necessary number of runtime links. These runtime links are modeled as connectors in UML 2.0.

Our model offers a coherent view for the design of architecture and component-based systems. Components serve as a naming mechanisms for abstracting from the internal parts, interfaces as declaration mechanisms of services (either provided or required) and Ports together with the dependency–realization relations as abstraction mechanisms of object interactions.

Architecture description languages (ADLs) define also high-level concepts for the design and modelling of architectures of systems, such as components, Ports, and configurations. A large number of ADLs have been proposed, some of them with a sound formal foundation. We only mention here Wright [AG97], Rapide [LKA$^+$95] and ACME [GMW97]. Closer to UML are the architectural descriptions provided by SDL [BH89], ROOM [SGW94] and UML-RT [Sel98] (the latter is in fact a UML profile interpreting ROOM concepts in terms of UML stereotypes). In [GCK02] and [MRRR02], several strategies for modelling components and other architectural concepts within UML are investigated, with as conclusion that these concepts are hard to describe in UML as it is.

Many models for components have been proposed in the last years, some informal and remaining within the realm of the existing UML (see for example [CD00]), and others founded on a logical and mathematical basis (e.g. Broy's component model based on streams of messages [BS01]. Similar to Broy's component model, the semantics of our model is also based on sequences of messages (like those used for the semantics of CSP [Hoa85]). However OMEGA components have dynamic aspects (e.g. Port instances) not fully covered by Broy's model. Moreover our component model is a conservative extension of an object-oriented model and therefore it requires the addition of only a couple of extra concepts to the standard UML 1.4. It is interesting to note that these additional concepts are also required by the component model proposed for UML 2.0 by the U2 partners [U20]. As described above, however, the semantics of these concepts is different between the two models.

Other interesting approaches are the one taken by Catalysis [DW98] and the precise UML group [pUM]. In OMEGA we are currently investigating the relationships between these approaches and our model and possible ways of integration.

Finally, we have shown how to exploit in a formal mathematical manner the hierarchical structure of components in compositional verification. Currently, we are working on the development of a tool for checking mutual consistency of the behavioral specifications of a set of components.

Acknowledgement The author is grateful for the input and helpful discussions with members of OMEGA, especially Frank de Boer and Marcello Bonsangue, in the design and evaluation of this component model.