

A Formal Model for Componentware

Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, Manfred Broy

Institut für Informatik, FORSOFT A1

Technische Universität München

80290 Munich, Germany

<http://www4.informatik.tu-muenchen.de>

Abstract

We present a formal model for component systems which provides precise mathematical definitions for concepts like component, interface, type and instance, as well as dynamic behavior and changes of system structure over time. Based on these concepts, we define the semantics of intuitive, commonly used graphical description techniques as an interpretation in terms of the presented system model. Several exemplary description techniques illustrate the feasibility of our approach.

1.1 Introduction

The essence of componentware is to build well-structured systems out of independently understandable and reusable building blocks. However, with the rise of pragmatic technologies like ActiveX [Mic98], JavaBeans [Jav99b], and Enterprise JavaBeans [Jav99a] as well as the growing interest in reuse and reengineering of legacy components it is evident that a solid, unified understanding and scientific models of componentware are still missing. Even central concepts like “component” or “interface” are defined and used in very different ways by different authors. Although existing Architecture Definition Languages, like Rapide [LKA⁺95], UniCon [SDK⁺95], or Wright [AG97] introduce these concepts on a more formal basis, they do not adequately consider all relevant behavior-related aspects of a component system and are sometimes difficult to apply in practice [NR99].

We believe that a clearly defined conceptual model designed for componentware development is essential, especially as a foundation for an overall componentware methodology comprising the integration of suitable graphical description techniques, tool support, and development process models [BRSV98]. Only then it is possible to answer typical questions like “What does this particular component do?”, “Will these interfaces match?”, and “Are these different system descriptions equivalent?” which are commonly posed during system development.

A fruitful discussion about such issues requires a precisely defined language as well as a comprehensive theory to reason about the behavior of a component sys-

tem and its constituents. While a thorough treatment of the latter requirement is beyond the scope of this paper, we provide rigorous mathematical definitions for all essential concepts of componentware, and illustrate the translation of intuitive description techniques into this system model. Based on this common metamodel, many practical applications are conceivable, like the development of consistency checkers, transformation tools, and code generators. Moreover, it leads to a common understanding of the involved concepts and terminology.

1.2 Overview

Initially, the design and development of a concise formal system model for componentware has to consider the following main requirements:

Adequacy: On the one hand, the proposed system model should be able to represent the central concepts of existing, pragmatic componentware technologies like ActiveX or JavaBeans. It has to provide common abstractions for these concepts which are easily recognizable for the users of componentware technologies. On the other hand, it should be possible to map such a system model to existing formal theories of software systems. Thereby, a large part of proven, theoretical concepts may be applied or reused, providing a firm ground for the proposed system model.

Expressiveness: Like all systems, component-oriented systems may be characterized by structure and behavior, i.e. the kind and organisation of their constituents as well as the exchange of information and the resulting change of state and structure over time. The proposed system model should be able to express both aspects adequately, without imposing unnecessary or unrealistic restrictions on represented component systems.

Clarity: Given the previous requirements, the proposed system model should rely on a minimal number of basic concepts with clearly defined relations between them. This facilitates understanding, communication and application of the model.

Obviously, these requirements are partly conflicting, thereby imposing a certain trade-off in the design of a proposed system model. For example, a faithful representation of a pragmatic componentware technology could require the introduction of a large number of detailed conceptual entities which in turn impedes the mapping to existing formal theories as well as the clarity of the model.

Regarding such trade-offs, we decided to restrict the model to a small number of basic concepts which are sufficiently abstract to incorporate most of the current technical componentware approaches. The presented model builds on related work on description techniques [Gro99, BHH⁺97] as well as existing models of distributed systems [Bro95, KRB96] based on the FOCUS methodology [BDD⁺92]. Its main

concepts to model component-oriented systems are instances, types, and descriptions:

Instances represent the individual operational units of a component system that determine its overall behavior. A system consists of a family of component instances. Each of them has interface instances and is connected to other component instances by connections between these interface instances.

Types represent subsets of interface resp. component instances with similar properties. Each instance is associated to exactly one type.

Descriptions characterize types and thus all instances associated with them. They consider syntactical as well as behavior-related properties, like interface and component signature, expected input/output communication, and collaboration between different components.

Regarding the adequacy of such a model, it is important to note that all of these concepts are present in current technical and practical approaches to componentware. A typical JavaBeans component package [Jav99b], for example, may contain serialized component instances of different types (i.e. Java classes) accompanied by Java byte code that defines their behavior. It also includes machine-readable signature descriptions of the contained component types, as well as arbitrary additional files with further information. Although the package format does not yet require standardized, machine-readable descriptions of the structural and behavioral properties of contained component types, such descriptions would be very valuable, both for human developers as well as for supporting development tools.

With respect to behavior, a component-oriented system may be further characterized by business-oriented and technical aspects. In order to fulfil a dedicated business functionality, the components of a system usually rely on a standardized technical infrastructure consisting, for example, of object request brokers [COR], Enterprise JavaBeans containers [Jav99a], databases, message queuing middleware, and so on. Most of the technical aspects as well as the details of the underlying technical protocols and services are not dealt with explicitly by application programmers. In the proposed model, the separation of business-oriented and technical characteristics is achieved by modeling the properties of the respective technical infrastructure within the behavior of interfaces, whereas abstract, business-oriented service protocols and the corresponding structural changes are associated with component behavior. This approach is similar to the concept of connectors in architecture description languages [AG97], which try to capture technical mechanisms and properties of connections.

The presentation of the proposed system model is structured as follows: Section 1.3 first introduces the basic concepts on the level of instances, and defines behavior with respect to communication and structure. Section 1.4 then considers types and descriptions. We provide a general formalization of these concepts and outline their interpretation for a number of exemplary descriptions. A brief section

with a conclusion and an outlook about necessary future work is given at the end of the paper.

1.3 Instances

In this section, we introduce the instance part of the formal system model. An instance may be defined as an individual operational unit that exists in a running system. We distinguish between three different kinds of instances, namely component, interface, and connection instances. In the following subsections, we define and characterize these basic concepts and the relations between them.

1.3.1 Basic Concepts

Components are the basic building blocks of a component system at runtime. Each component instance possesses a set of interface instances which may be connected to other interface instances via connections. As described in Section 1.3.3, messages generated by a component flow via its interfaces and the corresponding connections to other interfaces and their components.

Figure 1.1 illustrates a possible snapshot of a component system with five components ($c1$ to $c5$) shown as rectangles, their interfaces ($i1$ to $i9$) shown as circles, and the connections between the interfaces shown as lines between them. This kind of diagram resembles the well-known object instance diagrams in UML [Gro99].

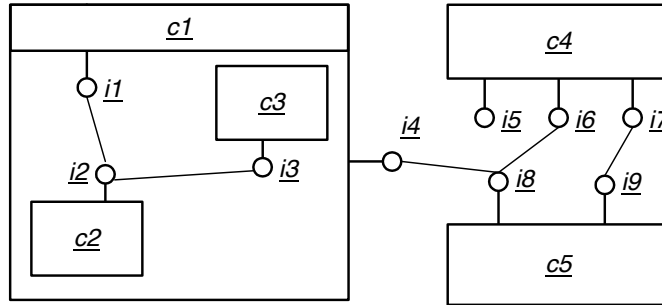


Fig. 1.1. Component Instance Diagram

In order to uniquely address the elements of a component system, we introduce the disjoint, infinite sets *Interface*, *Component*, and *Connection*. Interfaces are associated with components via the total but not necessarily injective function *assigned*

$$\textit{assigned} : \textit{Interface} \rightarrow \textit{Component}$$

This function is total as an interface cannot exist independently. It is associated with exactly one component. Note that *assigned* is generally not injective, as a component may have multiple interfaces.

A connection relates exactly two distinct interfaces. The total function *connIfs* determines which interfaces are related by a connection:

$$\text{connIfs} : \text{Connection} \rightarrow \{\{i, j\} \mid i, j \in \text{Interface} \wedge i \neq j\}$$

For a given interface, several different connections may exist. Thus, the presented model supports one-to-many connections between interfaces, as shown in Figure 1.1. Note that the elements of *Connection* represent symmetric connections between interfaces, as the set $\{i, j\}$ cannot be distinguished from the set $\{j, i\}$. Furthermore, the previous definition of *connIfs* excludes loop connections for a single interface. We also forbid multiple connections between the same two interfaces (cf. next section).

The remaining basic concept is component containment as illustrated by Figure 1.1: the components *c2* and *c3* are contained in component *c1*. This concept is modeled by the partial function *parent*

$$\text{parent} : \text{Component} \rightarrow \text{Component}$$

which returns for each component the parent component it is contained in. The function *parent* is partial since the top-level components of the system do not have a parent component. We require *parent* to be an acyclic function, as it is not possible for a component to contain itself or to be contained in one of its subcomponents. We also assume that there are no infinite chains of subcomponents in order to ensure that every subcomponent has a well-defined root component.

With regard to proper encapsulation of components, we exclude connections that cross a component boundary. Components may only be connected via their interfaces if one component is the parent of the other component or if they both have the same parent component (or no parent component at all). Connections from one interface of a component to another interface of the same component are not excluded. Formally, we obtain the following restriction on the set *Connection*:

$$\begin{aligned} \forall cn \in \text{Connection}, c, d \in \text{Component}, i, j \in \text{Interface} : \\ \text{connIfs}(cn) = \{i, j\} \wedge \text{assigned}(i) = c \wedge \text{assigned}(j) = d \Rightarrow \\ \text{parent}(c) = d \vee \text{parent}(d) = c \vee \text{parent}(c) = \text{parent}(d) \end{aligned}$$

Note that top-level components without a parent component are also covered by this equation, as we use strong equality on components. In the example of Figure 1.1, a direct connection between the interfaces *i3* and *i8* would be invalid, as it violates the encapsulation boundary of component *c1*.

1.3.2 Time and System Configuration Histories

Like related formal models [KRB96], we work with discrete time and regard time as an infinite chain of time intervals of equal length. We use \mathbb{N}^+ as an abstract time axis, and denote it by *T* for clarity. Furthermore, we assume a time synchronous

model because of the resulting simplicity and generality. This means that there is a global time scale that is valid for all parts of the modeled system.

We use *timed streams*, i.e. finite or infinite sequences of elements from a given domain, to represent histories of conceptual entities that change over time. A *timed stream* (more precisely, a stream with discrete time) of elements from the set X is an element of the type $X^T =_{\text{def}} T \rightarrow X$. Thus, a timed stream maps each time interval to an element of X . The notation x_t is used to denote the element of the valuation $x \in X^T$ at time $t \in T$. By $x \downarrow t$ we denote the stream containing only the elements of the first t time intervals. Operators on streams induce operators on sets of streams by element-wise application.

Streams may be used to model the creation and deletion of instances in a system. Within a given time interval t , only finite subsets of the sets of all possible component, interface, and connection instances may exist in the system. Following our notational conventions, $\text{interface} \in \wp(\text{Interface})^T$, $\text{component} \in \wp(\text{Component})^T$, and $\text{connection} \in \wp(\text{Connection})^T$ denote the changing subsets of instances that exist over time.

As already mentioned in the previous section, we exclude multiple connections between two interfaces in order to keep the resulting connection structure as simple as possible. Formally, this requires $\text{connIfs}|_{\text{connection}_t}$ to be injective for every subset connection_t . Furthermore, we require that a connection may only exist if both of its connected interfaces exist, as expressed by the following condition:

$$\forall t \in T : \forall cn \in \text{connection}_t : \text{connIfs}(cn) = \{i, j\} \Rightarrow i, j \in \text{interface}_t$$

Once a component, interface, or connection instance is removed from the system, it cannot be reactivated later. For a given interface i , this is stated by the condition:

$$i \in \text{interface}_t \wedge i \notin \text{interface}_{t+1} \Rightarrow (\forall n \in \mathbb{N}^+ : i \notin \text{interface}_{t+n})$$

Analogous conditions are required for components and connections. Obviously, the removal of a component, interface, or connection does not prohibit the creation of new components, interfaces, or connections with the same properties at a later time.

As expected for the containment relation, we require that once a parent component does not exist anymore, all its interfaces and subcomponents are removed from the system as well:

$$\begin{aligned} \forall c \in \text{Component}, t \in T : c \notin \text{component}_t \Rightarrow \\ (\forall x \in \text{Component} : \text{parent}(x) = c \Rightarrow x \notin \text{component}_t) \wedge \\ (\forall y \in \text{Interface} : \text{assigned}(y) = c \Rightarrow y \notin \text{interface}_t) \end{aligned}$$

The basic concepts introduced above may now be used to describe the changing configuration space of a component system over time. Let Conf^T denote the type of all system configuration histories. A given system configuration history $\text{conf} \in$

$Conf^T$ consists of a timed stream of tuples

$$conf_t =_{def} (interface_t, component_t, connection_t)$$

which captures the changing sets of instances at different times during system execution.

Note that the functions *assigned* and *parent* do not depend on time and are therefore not part of the system configuration history. Thus, the model essentially only allows to consider the activation and deactivation of interfaces, components, and connections—it cannot handle mobile components which migrate to another parent component, for example. A more detailed formal model for mobile components may be found in [BGR⁺99].

1.3.3 Interface and Component Behavior

Components possess a behavior with respect to communication and structural changes: a component may send and receive messages via its interfaces, and it may change the connection structure of the system by creating or deleting interfaces, components, and connections. In the context of the presented model, interfaces are active entities with an associated behavior, too. In contrast to components, however, they may not change the connection structure of the system.

1.3.3.1 Interface Behavior

Based on the current FOCUS system model [BDD⁺92], sequences of messages represent the fundamental units of communication. Within each time interval components resp. interfaces receive message sequences arriving at their interfaces resp. connections, and send message sequences to their respective environment. In order to model message-based communication, we denote the set of all possible messages with M , and the set of arbitrary finite message streams with M^* . By $\langle \rangle$, $\langle a \rangle$ and $\langle a, b \rangle$ we denote the empty sequence, the one-element sequence containing only a , and the two-element sequence containing a and b , respectively. The notation $x \hat{\ } y$ is used to denote the concatenation of the two sequences x and y .

An interface merges multiple incoming message sequences from its connections to a single message sequence for its assigned component. Likewise, it distributes the outgoing message sequence from its assigned component to a number of outgoing message sequences. A typical example is a *CORBAMethodServer* interface which serves a number of connected clients, buffering their method call requests so that the assigned component can handle them sequentially.

Figure 1.2 illustrates the behavior of the interface *i2*. It receives incoming message sequences from the connections *cn2* and *cn3*, and creates a single message sequence for the component *c1* (and vice versa for the outgoing messages).

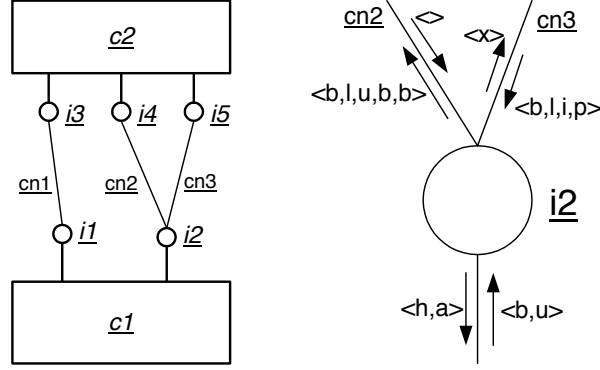


Fig. 1.2. Interface Behavior: Mapping Incoming to Outgoing Messages

Behavior Relation: Generally, the interface behavior of a system run may be represented by a pair relating its input history to its output history. We use the types *IfIn* resp. *IfOut* to denote an evaluation of the incoming resp. outgoing message sequences.

An interface receives messages from the component it is assigned to as well as from its attached connections, and sends messages in the opposite direction. Formally, the sequences of exchanged messages may be described by two relations

$$\begin{aligned} \text{IfIn} &=_{\text{def}} M^* \times (\text{Connection} \rightarrow M^*) \\ \text{IfOut} &=_{\text{def}} (\text{Connection} \rightarrow M^*) \times M^* \end{aligned}$$

Given an input evaluation $(x, y) \in \text{IfIn}$ of interface i , x represents the stream of messages flowing from the component $\text{assigned}(i)$ to i , and y represents the multiple input streams for all the connections that connect to i . Note that the used connection evaluation functions are partial, as the behavior of a given interface generally depends only on inputs resp. outputs of a small subset from the set of all connections.

As an example, consider the input and output message sequences for interface $i2$ in Figure 1.2: During the given time interval, $i2$ receives the sequence $\langle h, a \rangle$ from its assigned component and sends the sequence $\langle b, u \rangle$ to it. At the same time, it receives the empty sequence $\langle \rangle$ and the sequence $\langle b, l, i, p \rangle$ from its connections $cn2$ resp. $cn3$, and sends the sequences $\langle b, l, u, b, b \rangle$ and $\langle x \rangle$ to them.

Complete input resp. output histories over time have the type IfIn^T resp. IfOut^T . Accordingly, the behavior for all interfaces is provided by the function ifBeh

$$\text{ifBeh} : \text{Interface} \rightarrow \wp(\text{IfIn}^T \times \text{IfOut}^T)$$

which relates input histories to the corresponding output histories.

The behavior of an interface only depends on input streams from its attached connections which have to exist at the considered time. For instance, the behavior of

interface $i2$ in Figure 1.2 does not depend on the streams sent via the connection $cn1$. Likewise, we have to ensure that an interface only sends messages on attached and existing connections. The following condition is used to express these restrictions on interface behavior:

$$\begin{aligned} ifBeh(i) \subseteq \{ (ifIn, ifOut) \mid \forall t \in T : \\ ifIn_t \in (M^* \times (\{cn \in connection_t \mid i \in connIfs(cn)\} \rightarrow M^*)) \wedge \\ ifOut_t \in ((\{cn \in connection_t \mid i \in connIfs(cn)\} \rightarrow M^*) \times M^*) \} \end{aligned}$$

Note that the previous definition of interface behavior relies on the evaluation of connections and does not directly relate input and output of connected interfaces. This would impose an unnecessary restriction on the expected behavior of connections. Within the presented model, it is possible for connections to drop or even invent arbitrary messages. Of course, the communication behavior has to be specified in more detail to model particular technical infrastructures.

Causality: In order to be realizable, we require interface behaviors to be *causal* (also called *time-guarded*). Intuitively, this property means that the output of an interface at a given point in time does not depend on input that arrives at the same time or later in the future. This is stated by the following condition for all time intervals t :

$$ifIn_a \downarrow t = ifIn_b \downarrow t \Rightarrow ifOutSet_a \downarrow t + 1 = ifOutSet_b \downarrow t + 1$$

where $ifIn_a$ and $ifIn_b$ denote two input histories in the behavior of an interface, and $ifOutSet_a$ and $ifOutSet_b$ denote the corresponding sets of related output histories (cf. [BDD⁺92] for a more detailed discussion of causality).

1.3.3.2 Component Behavior

A component receives a set of message streams from its assigned interfaces, and, depending also on the configuration history of the system, produces output message streams for its interfaces. Additionally, it may also change the connection structure of the system. This enables components to interact with other components in a flexible way to realize advanced functionality.

Figure 1.3 visualizes an example of an execution step of the component $c2$, including its incoming resp. outgoing message sequences and the resulting changes in the system configuration. The execution step is triggered by the current system configuration and the incoming message sequences $\langle a \rangle$ via the interface $i4$ and $\langle \rangle$ via the interface $i5$. The execution results in changes to the system configuration—deletion of interface $i4$ with its attached connection to $i2$, and creation of component $c4$ together with its interface $i6$ and the corresponding connection to interface $i5$ —and the outgoing message sequence $\langle o, x \rangle$ on interface $i5$.

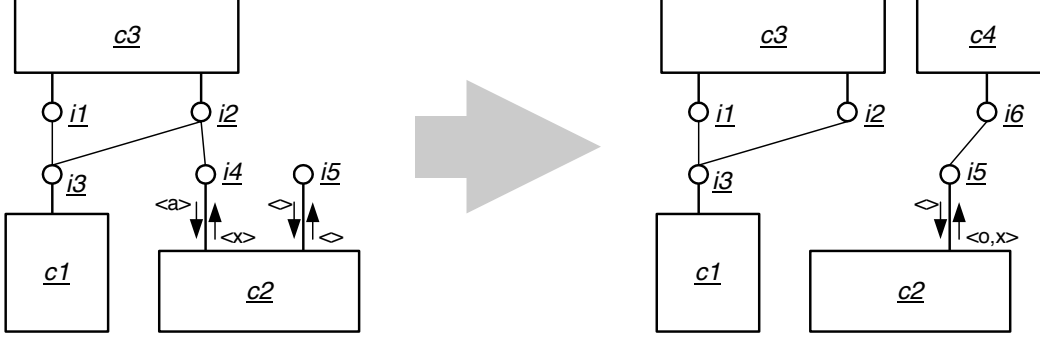


Fig. 1.3. Component Behavior

Behavior Relation: Similar to interfaces, we describe the sets of incoming and outgoing message sequences of a component by two evaluation functions $CompIn$, $CompOut =_{def} Interface \rightarrow M^*$. Note that these evaluation functions are partial, in general, as the behavior of a component usually depends only on a small subset of the set of all interfaces. Analogous to interface histories, complete component input resp. output histories over time have the type $CompIn^T$ resp. $CompOut^T$. In contrast to interface behavior, however, the behavior of a component also depends on the history of the system configuration $Conf^T$. Consequently, the behavior for all components is provided by the function

$$compBeh : Component \rightarrow \wp(CompIn^T \times CompOut^T \times Conf^T)$$

which assigns combinations of input, output and configuration histories to each component of the system. Note that all components share the same configuration history.

A plausible restriction requires that only existing and assigned interfaces are relevant for component behavior at time t :

$$compBeh(c) \subseteq \{(compIn, compOut, conf) \mid \forall t \in T : \\ compIn_t, compOut_t \in (\{i \in interface_t \mid assigned(i) = c\} \rightarrow M^*)\}$$

Note that the required causality of component behavior may be stated by a condition analogous to the previously defined condition for interfaces (cf. Section 1.3.3.1). In the following paragraph, we state more restrictions on the behavior of components with respect to their composition.

Environments and Compositionality: So far, the presented model is highly non-compositional, as components may change the connection structure of arbitrary other components located anywhere in the system. Obviously, additional restrictions on component behavior with respect to structural changes are necessary to ensure the principle of encapsulation.

We first define the *immediate environment* of a component with respect to its components or more precisely connections by the functions

$$\begin{aligned} imCompEnv &: Component \rightarrow \wp(Component) \\ imConnEnv &: Component \rightarrow \wp(Connection) \end{aligned}$$

The immediate environment consists of the component itself, its parent component, its subcomponents, as well as all brother and sister components and the connections between them:

$$\begin{aligned} \forall c, ec \in Component : ec \in imCompEnv(c) &\Leftrightarrow \\ &ec = c \vee parent(ec) = c \vee parent(c) = ec \vee parent(ec) = parent(c) \\ \forall c \in Component, en \in Connection : en \in imConnEnv(c) &\Leftrightarrow \\ \forall i \in connIfs(en) : assigned(i) &\in imCompEnv(en) \end{aligned}$$

Figure 1.4 visualizes the immediate component environment of component $c1$: It consists of $c1$ itself, its child components $c2$ and $c4$, its brother component $c6$, and its sister component $c8$. If there existed a parent component of component $c1$, it would also be contained in the immediate environment.

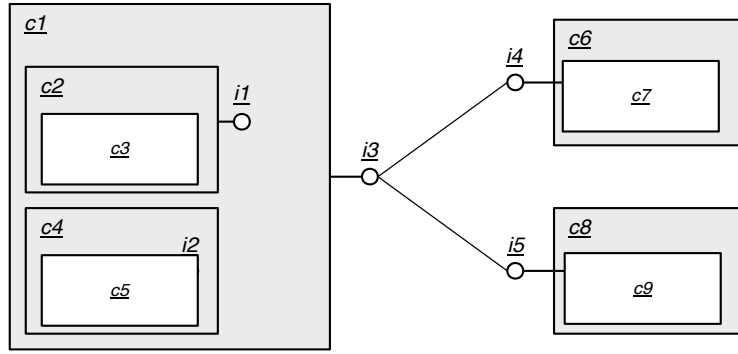


Fig. 1.4. Immediate Environment of Component $c1$

This definition allows to restrict the structural behavior of a component to the creation resp. deletion of its assigned interfaces and the creation resp. deletion of components and connections in its immediate environment:

$$\begin{aligned} compBeh(c) \subseteq \{ (compIn, compOut, conf) \mid \forall t \in T : conf_t \in \\ (\{i \mid i \in interface_t \wedge assigned(i) = c\} \times imCompEnv(c) \times \\ imConnEnv(c)) \} \end{aligned}$$

Note that creation and deletion of component instances, for example, does not mean that new elements are added to the set *Component*, but that $component_{t+1}$ contains a component not contained in $component_t$.

The previous restriction results in a compositional model of the structural changes inside a component, as they may only be affected by the component itself or by its immediate subcomponents. However, it is generally not possible to derive the overall behavior of a parent component from the individual behaviors of its subcomponents, because the parent component may react on them according to its own behavior. In current technical approaches, this additional overall component behavior is known as the “glue code” of the parent component.

1.4 Types and Descriptions

The basic concepts and their relations as covered in the previous sections provide mathematical definitions for the constituents of a component system at runtime. In order to be useful for practical development, however, such a formal model is not sufficient, as it requires experience in mathematical techniques, which most likely application developers and end users do not have. Therefore, additional concepts are needed to describe a component system in a more intuitive and illustrative way.

As already mentioned in Section 1.2, such descriptions usually not only characterize single instances, but consider the properties of many instances of the same interface or component type.

In the following, we introduce formal definitions for types and descriptions in the context of the presented system model. Based on these definitions, we outline a formalization for a selection of exemplary description techniques, namely signature descriptions, component structure diagrams, and extended event traces. Our intent is to demonstrate the adequacy and flexibility of the proposed system model, and to provide a basic methodological framework for the translation of more comprehensive, professional description techniques.

Using formally founded descriptions has many advantages, as it combines the rigour of formal methods with the intuitiveness of textual and graphical techniques that are widely used in practice. While end users can be protected from the intricacies of mathematical formulae, developers of description techniques and methodologists have the possibility to define consistency conditions and development steps in a precise and unambiguous way based on the formal foundation of the system model.

Ideally, the properties defined by descriptions are fulfilled for all described instances of the system. In practice, however, this goal is very difficult to reach, as it requires all descriptions to be consistent with each other as well as with the system’s implementation. Currently, only very few descriptions may be checked or enforced statically—in general, this requires formally founded description techniques with adapted refinement and proof calculi that allow the verification or generation of code. In practice, extensive testing is usually performed instead. Such tests may be regarded as the dynamic validation of special test case descriptions.

1.4.1 Basic Concepts

In order to cover all kinds of descriptions in an abstract way, we introduce the infinite set *Description*. Each element of *Description* is represented by a syntactical notation which may be graphical, textual or a combination thereof. Additionally, there exists an interpretation which translates the notation into terms of the formal system model, i.e. a representation of the description's semantics. It precisely states the required properties of the described part of the system. Accordingly, we associate with each description a corresponding predicate which determines the fulfillment of these properties.

As mentioned previously, sets of similar interface and component instances are usually grouped together by *types*. Formally, we define the sets *ComponentType* and *InterfaceType* to be the infinite sets of component resp. interface types. We introduce two total functions *typeOf* which assign a unique type to each interface resp. component instance of the system.

$$\begin{aligned} \textit{typeOf} & : \textit{Component} \rightarrow \textit{ComponentType} \\ \textit{typeOf} & : \textit{Interface} \rightarrow \textit{InterfaceType} \end{aligned}$$

Note that this simple formalization excludes instances with multiple types, as provided by certain technical componentware approaches that offer subtyping. Note furthermore, that *typeOf* does not depend on time; thus, it is not possible for an instance to change its determined type at runtime.

As explained in Section 1.2, descriptions may be explicitly assigned to certain types in order to state the desired restrictions on their instances. The assignment of a set of descriptions to a certain type is given by the following two functions:

$$\begin{aligned} \textit{descOf} & : \textit{ComponentType} \rightarrow \wp(\textit{Description}) \\ \textit{descOf} & : \textit{InterfaceType} \rightarrow \wp(\textit{Description}) \end{aligned}$$

Figure 1.5 illustrates the resulting many-to-many relationship between instances, types, and descriptions.

Most descriptions are assigned to a single type. Signatures are a good example—as illustrated in Section 1.4.2, they define the set of admissible messages for the behavior of instances of a given, single type. Other descriptions, like extended event traces, consider the interaction of instances of multiple types. Consequently, they are assigned to all these types (cf. Section 1.4.5).

Note that a description may refer to a certain type without being explicitly assigned to it. An example are, again, signature descriptions—although they are assigned to exactly one type, they usually mention a variety of other types to characterize parameter and result values.

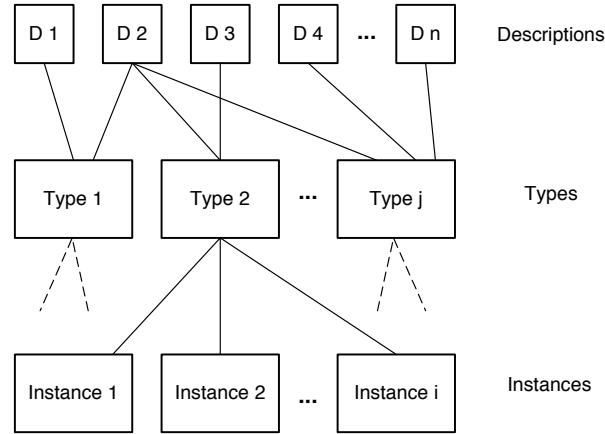


Fig. 1.5. Relation between Instances, Types and Descriptions

1.4.2 Signature Descriptions

Signatures are a description technique widely used in software engineering. They are present in all current technical componentware approaches. Generally, signatures characterize the set of messages exchangeable at an interface. As such, they are fundamental for programmers as well as for composition tools. Most dynamic description techniques rely on the message sets defined within signatures, and many programming languages offer static type checking based on purely syntactical properties of the considered program. Good examples for signature descriptions are CORBA IDL files [COR, OH98] and Java interfaces [Fla96].

In the following, we demonstrate how interface signatures may be formalized based on the proposed formal system model. The presentation is structured in two parts: the first paragraph provides the translation of the example signature **Counter** into the corresponding predicate. The second paragraph generalizes the example, yielding a general predicate for arbitrary interface signatures.

Exemplary Translation: The following IDL fragment

```
interface Counter {
    long getSum();
    void setSum(long x);
    long increment();
};
```

introduces the IDL interface type **Counter** with its three methods **getSum**, **setSum**, and **increment**. We model this IDL fragment as a description *CounterIDL*, which is an element of the set *Description*. As the description characterizes all instances

of the interface type *Counter*, it is explicitly assigned to it by stating

$$\text{CounterIDL} \in \text{descOf}(\text{Counter})$$

For this example, we assume that method calls are translated to message sequences according to a simple scheme: A call of a method with a certain number of parameters is translated to a message sequence that starts with a method identifier and ends with the list of the actual parameters. The return value is translated to a message sequence of length one.

At a high level of abstraction, the given IDL method declarations in **Counter** restrict the possible input and output messages of the interface instances of type *Counter*, as given by the following two sets:

$$\begin{aligned} \text{CounterInMsg} &= \{\text{increment}, \text{getSum}, \text{setSum}\} \cup \mathbb{N} \\ \text{CounterOutMsg} &= \mathbb{N} \end{aligned}$$

CounterIDL specifies that interface instances *i* of type *Counter* obey the given signature. Formally, this may be expressed as follows:

$$\begin{aligned} \text{CounterIDL} \in \text{descOf}(\text{typeOf}(i)) &\Rightarrow \forall(\text{ifIn}, \text{ifOut}) \in \text{ifBeh}(i), t \in T : \\ \text{ifIn}_t &\in (\text{CounterOutMsg}^* \times (\text{Connection} \rightarrow \text{CounterInMsg}^*)) \wedge \\ \text{ifOut}_t &\in ((\text{Connection} \rightarrow \text{CounterOutMsg}^*) \times \text{CounterInMsg}^*) \end{aligned}$$

Note that, in order to keep the example simple, this formalization neglects information about the grouping of method identifiers with their corresponding parameters and return values.

General Translation: In order to generalize the previous example for arbitrary IDL files, we first need to model the set of all IDL descriptions as $\text{IDLText} \subseteq \text{Description}$. An abstract mathematical representation of the information represented by an *IDLText* description is given by two functions $\text{inMsg}, \text{outMsg} : \text{IDLText} \rightarrow \wp(M)$. Interface instances which obey an *IDLText* description may thus be characterized by the predicate

$$\text{fulfillsIDLText} : \text{IDLText} \rightarrow (\text{Interface} \rightarrow \mathbb{B})$$

with the following definition

$$\begin{aligned} \text{fulfillsIDLText}(d)(i) &\Leftrightarrow \\ d \in \text{descOf}(\text{typeOf}(i)) &\Rightarrow \forall(\text{ifIn}, \text{ifOut}) \in \text{ifBeh}(i), t \in T : \\ \text{ifIn}_t &\in (\text{outMsg}(d)^* \times (\text{Connection} \rightarrow \text{inMsg}(d)^*)) \wedge \\ \text{ifOut}_t &\in ((\text{Connection} \rightarrow \text{outMsg}(d)^*) \times \text{inMsg}(d)^*) \end{aligned}$$

Based on interface signatures, it is also possible to define component signatures. Following the relationship between interfaces and components as described in previous sections, a component signature is defined by the types of its assigned interfaces.

1.4.3 Component Structure Descriptions

As implied by the name, component structure descriptions are intended to describe the dynamic structural behavior of a component system. Due to the inherent problems associated with the presentation of large, dynamic graph structures, most of the current structural techniques are restricted to static configurations or system snapshots. The presented exemplary description technique is no exception: *Component Structure Diagrams* define the required static internal structure for components of a given type as shown in the example of Figure 1.6.

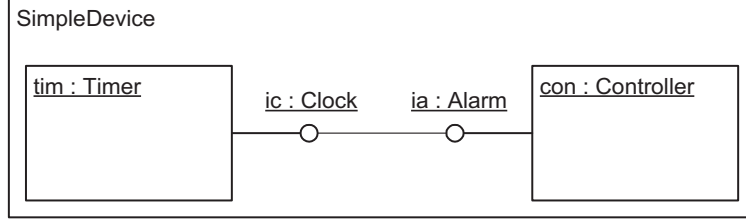


Fig. 1.6. Exemplary Component Structure Diagram

This diagram visualizes the required internal white-box structure of component instances of type *SimpleDevice*. Internally, each component instance consists of two other components, *tim* of type *Timer* and *con* of type *Controller*, which are connected via a pair of interfaces.

This diagram may be formalized as a description $\text{SimpleDeviceCSD} \in \text{Description}$. Although it mentions three component types (*SimpleDevice*, *Timer*, and *Controller*) and two interface types (*Clock* and *Alarm*), it is exclusively assigned to the component type *SimpleDevice* by stating

$$\text{SimpleDeviceCSD} \in \text{descOf}(\text{SimpleDevice})$$

The translation of this description to a predicate on the system model results in the following formula:

$$\begin{aligned} &\text{SimpleDeviceCSD} \in \text{descOf}(\text{typeOf}(c)) \Rightarrow \\ &\quad \exists \text{tim}, \text{con} \in \text{Component}, \text{ic}, \text{ia} \in \text{Interface}, \text{cn} \in \text{Connection} : \\ &\quad \text{typeOf}(\text{tim}) = \text{Timer} \wedge \text{parent}(\text{tim}) = c \wedge \\ &\quad \text{typeOf}(\text{con}) = \text{Controller} \wedge \text{parent}(\text{con}) = c \wedge \\ &\quad \text{typeOf}(\text{ic}) = \text{Clock} \wedge \text{assigned}(\text{ic}) = \text{tim} \wedge \\ &\quad \text{typeOf}(\text{ia}) = \text{Alarm} \wedge \text{assigned}(\text{ia}) = \text{con} \wedge \\ &\quad \text{connIfs}(\text{cn}) = \{\text{ia}, \text{ic}\} \wedge \\ &\quad (\forall t \in T : c \in \text{component}_t \Rightarrow \\ &\quad \quad \text{tim}, \text{con} \in \text{component}_t \wedge \text{ic}, \text{ia} \in \text{interface}_t \wedge \text{cn} \in \text{connection}_t) \end{aligned}$$

Note that this predicate only states the required internal configuration, and does not exclude the presence of additional subcomponents. This results in underspecification and thus allows us to refine the diagram by adding further components, interfaces, and connections without violating the specification.

We refrain from generalizing this example to a general predicate for all component structure diagrams. Although this could be done following the presented approach, a complete formal treatment is beyond the scope of this paper.

1.4.4 Input/Output Behavior Descriptions

In this section, we consider a simple *state transition diagram* as an example for a component input/output behavior description technique. An exemplary component state transition diagram is shown in Figure 1.7.

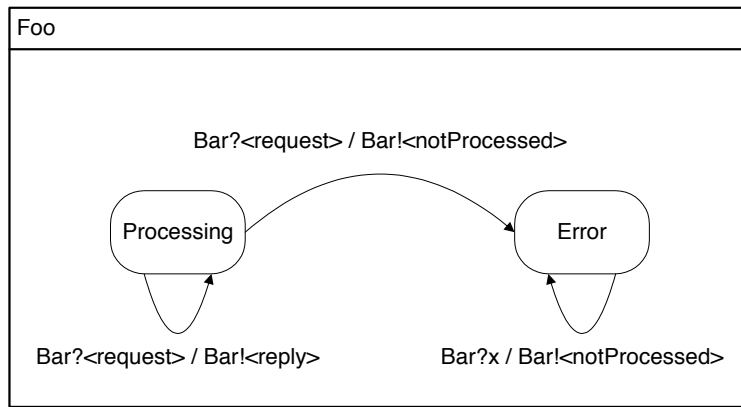


Fig. 1.7. Exemplary Component State Transition Diagram

This diagram visualizes the input/output behavior of components of type *Foo* with respect to the communication of assigned interface instances of type *Bar*. The semantics of this diagram is as follows: whenever a *Foo* instance receives the message sequence $\langle request \rangle$ via a *Bar* interface, it either reacts by sending the message sequence $\langle reply \rangle$ on the same interface at the following time step, or it decides to stop message processing and continues sending $\langle notProcessed \rangle$ message sequences from there on.

This diagram may be formalized as a description $FooSTD \in Description$, which is explicitly assigned to the component type *Foo* by

$$FooSTD \in descOf(Foo)$$

The translation of this description to a system model predicate is expressed as

$$\begin{aligned} FooSTD \in descOf(typeOf(c)) &\Leftrightarrow \\ \forall (compIn, compOut, conf) \in compBeh(c), \forall i \in Interface : \\ typeOf(i) = Bar \wedge typeOf(assigned(i)) = Foo &\Rightarrow \\ compOut|_i = \langle \rangle \wedge processing(compIn|_i) \end{aligned}$$

where $compIn|_i$ resp. $compOut|_i$ denotes the restriction of the component's complete input resp. output to the timed message stream corresponding to the interface i . The predicate employs two auxiliary functions:

$$processing, error : (M^*)^T \rightarrow (M^*)^T$$

The function *processing* is defined by

$$\begin{aligned} x = \langle request \rangle &\Rightarrow \\ processing(x \hat{s}) &= \langle reply \rangle \wedge processing(s) \vee \\ processing(x \hat{s}) &= \langle notProcessed \rangle \wedge error(s) \wedge \\ x = \langle \rangle &\Rightarrow \\ processing(x \hat{s}) &= \langle \rangle \wedge processing(s) \end{aligned}$$

whereas *error* is defined by

$$\forall x \in M^* : error(x \hat{s}) = \langle notProcessed \rangle \wedge error(s)$$

Note that, similar to the component structure description in the previous section, the behavior of *Foo* components in state *processing* is underspecified and may be refined in a more detailed description. Again, we refrain from a generalization of this example. A comprehensive treatment may be based on [Rum96], which considers a formal, stream-based semantics for state transition diagrams in the context of object-oriented systems.

1.4.5 Interaction Descriptions

Interaction descriptions specify the communication between two or more interfaces. They are useful for elaborating the decomposition of a system into individual interfaces and components, and for designing interaction patterns between a set of components as communication via their interfaces. Familiar interaction descriptions are Message Sequence Charts [IT93] as well as UML sequence and collaboration diagrams [Gro99]. The example of Figure 1.8 introduces a variation called *Extended Event Trace*.

This diagram visualizes the interaction between a set of components with an *Observer* interface and a component with an *Observable* interface. The communication follows the well-known design pattern from [GHJV95]: all registered observers

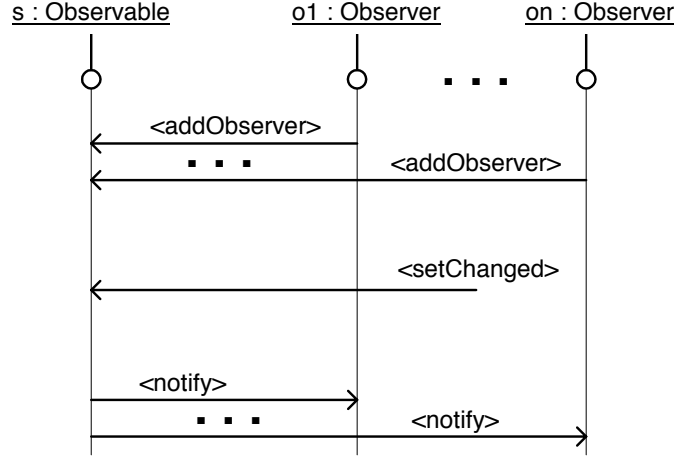


Fig. 1.8. Extended Event Trace: The Observer Pattern

must be notified when one of them sends a **setChanged** message to the observed entity.

This diagram may be formalized as $ObserverEET \in Description$. In contrast to the previous descriptions, $ObserverEET$ cannot be assigned to a single component or interface type. Instead, it is assigned to two interface types as $ObserverEET \in descOf(Observer)$ and $ObserverEET \in descOf(Observable)$, although we will only consider the communication obligations imposed on the *Observable* interface: For the interpretation of the diagram, we introduce two auxiliary functions

$$ifInConn, ifOutConn : Interface \times Connection \times T \rightarrow M^*$$

which return the input resp. output message sequences of a given interface's connection at a given time interval. Based on these functions, the diagram may be translated to the following logic formula:

$$\begin{aligned}
 & ObserverEET \in descOf(Observable) \wedge typeOf(i) = Observable \Rightarrow \\
 & \forall t1, t2 \in T, cn1, cn2 \in Connection : \\
 & \quad t1 < t2 \wedge cn1 \in connection_{t1} \wedge cn2 \in connection_{t2} \wedge \\
 & \quad ifInConn(i, cn1, t1) = \langle addObserver \rangle \wedge \\
 & \quad ifInConn(i, cn2, t2) = \langle setChanged \rangle \Rightarrow \\
 & \exists t3 \in T : \\
 & \quad t2 < t3 \wedge cn1 \in connection_{t3} \wedge \\
 & \quad ifOutConn(i, cn1, t3) = \langle notify \rangle
 \end{aligned}$$

Note that the previous formalization implies a rather strict observer-observable relation, as it requires all observers to be connected until they have received all

pending notification events. A more relaxed variant would allow observers to disconnect themselves at any time. Again, a general interpretation of extended event traces is beyond the scope of this paper. A formal treatment of message sequence charts may be found in [BK98].

1.5 Conclusion and Outlook

In this paper, we have proposed a formal system model for component systems. It is organized in different layers: the instance layer describes the relations and behavior of components, interfaces, and connections with respect to the flow of messages and the structural changes that may occur at runtime. The type layer groups component and interface instances into disjoint sets which may then be characterized by descriptions. In order to illustrate the adequacy of this approach, we formalized a selection of simple exemplary textual and graphical description techniques for syntactic and behavioral properties. Our approach here is to translate these descriptions into predicates that impose restrictions on the described components.

In order to keep the presentation manageable and understandable, we deliberately chose to simplify certain concepts or even omitted unnecessary details. For example, it is desirable to employ structured messages instead of the flat set of messages M that was used throughout this paper. Moreover, the proposed model does not yet provide support for mobile components. In the context of our model, this could be done by allowing the containment relation *parent* to change over time. However, we believe this simple extension is not sufficient—a more general approach requires the introduction of a location concept, as proposed in [BGR⁺99].

Another restriction is more serious: the system model is not yet able to express the modification of a component’s behavior, and its types and descriptions during runtime. Such modifications take place, for example, when a programmer introduces new types and descriptions, or when the implementation of a certain component is replaced during runtime. In the context of componentware, runtime and design time are not clearly distinguished, as configuration and adaptation tools and runtime environments allow the manipulation of running systems. An integrated formal model has to consider such modifications within the same time scale used to describe the ‘normal’ behavior of the system.

With respect to development methodology and description techniques, there remains much work to be done. First, it would be desirable to have a set of common interface specifications modeling the properties of existing technical middleware approaches. These specifications could then be used as exchangeable building blocks for communication while concentrating on business-oriented system functionality. Furthermore, a toolkit of coordinated, intuitive description techniques for componentware is required. The syntax and semantics should be precisely defined in terms of the system model as outlined for the exemplary description techniques in Section 1.4. Finally, methodical recommendations have to be provided, ranging from

formal proof and refinement rules to consistency checks, test strategies, and overall process patterns.

It is our hope that the proposed system model is able to serve as a foundation for further work. Essentially, there are currently two kinds of conceivable extensions. On the one hand, additional concepts may be added to the system model itself, in order to represent the properties of existing technical approaches more adequately. On the other hand, the system model may be used as a foundation for a toolkit of carefully coordinated description techniques and a corresponding development methodology.

Acknowledgements

We thank Bernhard Rumpe and Bernhard Schätz for interesting discussions and comments on earlier versions of this paper.

Bibliography

- [AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997.
- [BDD⁺92] Manfred Broy, Franz Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München, Institut für Informatik, January 1992.
- [BGR⁺99] Klaus Bergner, Radu Grosu, Andreas Rausch, Alexander Schmidt, Peter Scholz, and Manfred Broy. Focusing on mobility. In Ralph H. Sprague, Jr., editor, *Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 1999.
- [BHH⁺97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.
- [BK98] Manfred Broy and Ingolf Krüger. Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts. In *Proceedings of the ICFEM 98*. IEEE Press, 1998.
- [Bro95] Manfred Broy. Mathematical system models as a basis of software engineering. *Computer Science Today*, 1995.
- [BRVS98] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. An integrated view on componentware - concepts, description techniques, and development process. In Roger Lee, editor, *Software Engineering : Proceedings of the IASTED Conference '98*. ACTA Press, Anaheim, 1998.
- [COR] Object Management Group CORBA. OMG website, <http://www.omg.org>.
- [Fla96] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., 2nd edition, 1996.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Gro99] UML Group. Unified Modeling Language. Version 1.3, Rational Software Corporation, 1999.
- [ITU93] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1993.
- [Jav99a] JavaSoft. Enterprise JavaBeans website, <http://java.sun.com/products/ejb/>, 1999.

- [Jav99b] JavaSoft. JavaBeans website, <http://java.sun.com/beans/>, 1999.
- [KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems: The SysLab system model. In J.-B. Stefani E. Naijm, editor, *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, pages 323–338. ENST France Telecom, 1996.
- [LKA⁺95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [Mic98] MicroSoft Corporation. MicroSoft COM homepage, <http://www.microsoft.com/com>, 1998.
- [NR99] E. Di Nitto and D. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proceedings of the 1999 International Conference on Software Engineering*. ACM Press, 1999.
- [OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 2nd edition, 1998.
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, Technische Universität München, 1996.
- [SDK⁺95] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.