A Formal Model for Componentware

Klaus Bergner, Andreas Rausch, Marc Sihling,
Alexander Vilbig, Manfred Broy
Institut für Informatik, FORSOFT A1
Technische Universität München, Germany

Abstract

In this paper, we outline and clarify our view of the essential concepts of componentware, focusing on the role of formal foundations in the context of an overall development methodology. We provide a formal system model which allows for clear definitions of concepts like, for instance, component, interface, connection, aggregation, and refinement. While existing approaches in the field of interface and architecture description languages are mainly restricted to static, structural properties of a system, the proposed model covers also essential dynamic characteristics.

On the one hand, this pertains to the flow of data and control between the system's components. Based on precise definitions of the basic concepts, we evolve an understanding of communication as an abstract service protocol between component interfaces. On the other hand, a system's dynamics correspond to changes of the connection structure between component instances and to the creation and destruction of component instances.

This work is preceeding another publication which focuses on advanced topics such as types and descriptions in the scope of componentware.

1 Introduction

The main goals of componentware, namely, building well-structured systems out of independently understandable and reusable parts, are not new in computer science. A variety of approaches have tried to achieve these goals, providing a foundation of knowledge about many aspects of system architecture, program development, and software reuse. However, with the rise of new component technologies like ActiveX [Mic98], Java Beans [Jav99b], and Enterprise Java Beans [Jav99a] as well as the growing interest in reuse and reengineering of legacy components, it is evident that a solid, unified understanding of componentware is still missing. Even central concepts like "component" or "interface" are defined and used in very different ways by different authors.

We believe that a clearly defined conceptual model for component-oriented software development is essential, especially as a foundation for an overall componentware methodology. In [BRSV98b], we motivate the need for such a methodology and outline its basic constituents:

Formal System Model: The system model comprises the basic concepts of technical componentware approaches, like interfaces, components, and communication. They are clearly and unambiguously defined in mathematical terms.

Description Techniques: As the understanding of a formal system model requires expertise in mathematical techniques, intuitive graphical and textual description techniques are required for practical system development. The common formal system model may be used to define the semantics of the employed description techniques [BHH+97].

Process Model: System development should be organized according to a process model tailored to componentware. A proposal for a novel, flexible approach based on so-called process patterns [BRSV98a] addresses these aspects.

Tool Support: It is necessary to support large-scale system development with dedicated tools. They should be able to generate parts of the implementation and documentation of the system, check the consistency of system descriptions, and to verify critical system properties based on the formal system model (cf. [Tec96]).

In this paper we concentrate on the development of a formal system model for componentware. The existing approaches in the field of interface and architecture description languages are not sufficient as they are mostly restricted to the description of syntactic and structural properties of component systems. The majority of interface description languages, like CORBA IDL [COR, OH98] or Java interfaces [Fla96], only allow to specify the signature of component interfaces, thus neglecting additional information about the semantics and the behavior of interfaces as well as components.

Architectural description languages, like MILs, Rapide, Aesop, UniCon, or Wright as summarized in [BDRS97] introduce the concepts of components and communication between them via connectors, but do not consider all behavior-related aspects of a component system. Behavior is not limited to the communication between pairs of components, but also includes changes to the overall connection structure, the creation and destruction of instances, and even the introduction of new types at runtime. In the context of componentware, these aspects are essential because dynamic changes of a system may happen both during its construction at design-time as well as during its execution at runtime, either under control of the system itself or initiated by human component developers and component assemblers.

2 Overview

The proposed formal system model builds on related work on description techniques [Gro99, Thu98, BHH+97] and existing models of distributed systems [Bro95, KRB96, GKR96] based on the FOCUS methodology [BDD+92, Bro98]. The model considers both static syntactical and structural properties, as well as dynamic and behavioral aspects. The latter include the flow of data and control between components of a system, leading to an understanding of communication as an abstract service protocol between interfaces. Additionally, component behavior also comprises the changes in system structure, i.e. the creation and destruction of components, interfaces, and the connections between them.

The main concepts of the presented system model are instances, types, and descriptions:

- *Instances* represent the individual operational units of a component system that determine its overall behavior. We distinguish between component, interface, and connection instances, and define a number of relations and conditions that model properties of existing technical componentware approaches.
- *Types* address a subset of interface resp. component instances with similar properties. Each instance is associated to exactly one type.
- *Descriptions* characterize each type and thus all instances associated with it. They consider syntactical as well as behavior-related properties, like interface and component signature, expected input/output communication, and collaboration between different components, for example.

In the following, we strictly distinguish between these different concepts, speaking of component instances, component types, and component descriptions, respectively. The terms 'in-

terface' resp. 'component' are used as abbreviations for 'interface instance' resp. 'component instance' throughout this paper.

All of these concepts are present in current technical approaches to componentware. A typical Java Beans component package [Jav99b], for example, may contain several component instances of different types (i.e. Java classes) with the 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.

The presentation of the proposed system model is structured as follows: Starting out at the instance level from the basic ideas in Section 3.1, we introduce the concept of time as well as system configurations in Section 3.2 and briefly talk about interface and component behavior in Section 3.3. A section with a conclusion and an outlook about necessary future work ends the paper. A discussion about component types and descriptions would be beyond the scope of this paper and is thus postponed to an appearing publication (cf. [BRS$^+$ar]).

3 Instances

In this section, we introduce the instance view 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.

3.1 Basic Concepts

Components are the basic building blocks of a running component system. Each component possesses a set of interfaces which may be connected to other interface instances via connections. As described in Section 3.3, messages sent by a component flow via its interfaces over the corresponding connections to other interfaces.

Figure 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].

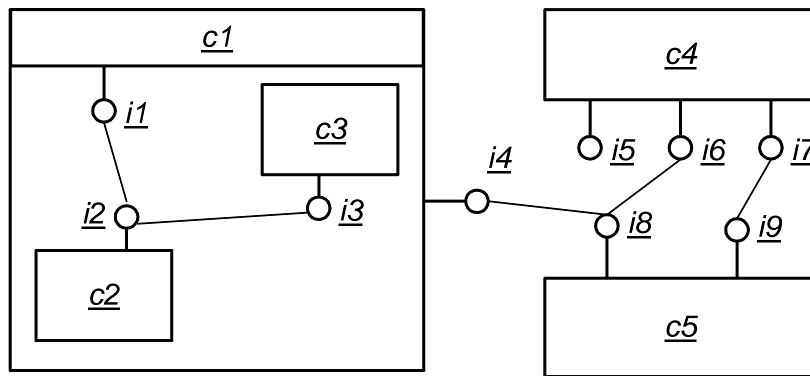

Figure 1: Component Instance Diagram

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 compo-

nents via the total but not necessarily injective function *assigned*:

$$assigned \quad : \quad Interface \rightarrow Component$$

Because this function is total, an interface may not exist by itself, but has to be 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:

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

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. Note that the elements of *Connection* represent symmetric connections between interfaces, as the set $\{i, j\}$ cannot be distinguished from the set $\{j, i\}$.

The last basic concept is component containment as illustrated by Figure 1: the components $c2$ and $c3$ are contained in component $c1$. This concept is modeled by the partial function

$$parent \quad : \quad Component \rightarrow Component$$

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 assume *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). Furthermore, connections from one interface of a component to another interface of the same component are also valid. This leads to the following restriction on the set *Connection*:

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

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, a direct connection between the interfaces $i3$ and $i8$ would be invalid, as it violates the encapsulation boundary of component $c1$.

3.2 Time and System Configuration Histories

Similar to related approaches [Bro98], we 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

$$X^T \quad =_{def} \quad \mathbb{N}^+ \rightarrow X$$

with $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. 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 the chosen notational conventions, $interface \in Interface^T$, $component \in Component^T$, and $connection \in Connection^T$ 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 $connIfs|_{connection_t}$ to be injective for every subset $connection_t$. Furthermore, we define that a connection may only exist if both connected interfaces exist, as expressed by the following condition:

$$\forall t \in T : \forall cn \in connection_t : connIfs(cn) = \{i, j\} \Rightarrow i, j \in interface_t$$

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

$$\begin{aligned} &\forall i \in Interface, t \in T : \\ &\quad i \in interface_t \wedge i \notin interface_{t+1} \Rightarrow (\forall n \in \mathbb{N}^+ : i \notin interface_{t+n}) \end{aligned}$$

Analogous conditions hold for components and connections. Of course, 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:

$$\begin{aligned} &\forall c \in Component, t \in T : c \notin component_t \Rightarrow \\ &\quad (\forall x \in Component : parent(x) = c \Rightarrow x \notin component_t) \wedge \\ &\quad (\forall y \in Interface : assigned(y) = c \Rightarrow y \notin interface_t) \end{aligned}$$

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 $conf \in Conf^T$ consists of a timed stream of tuples which captures the changing sets of instances during system execution.

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

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 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 detailled formal model for mobile components may be found in [BGR+99].

3.3 Interface and Component Behavior

Components possess a certain 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.

This distinction separates business-oriented functionality as services provided by components and their interactions from technical communication protocols provided by interfaces. Similar to the concept of connectors in architecture description languages [BDRS97], we propose to model most properties of technical infrastructure within the behavior of interfaces whereas abstract service protocols and the necessary connection structure are associated with component behavior.

3.3.1 Interface Behavior

Based on the current FOCUS system model [Bro98], 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 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 for its connections. A typical example is a CORBA MethodServer interface which serves a number of connected clients, buffering their method call requests so that the assigned component can handle them sequentially.

Behavior Relation: Generally, the interface behavior during 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, these exchanged messages may be described by two relations

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

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

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 $ifBeh$

$$ifBeh \;:\; Interface \rightarrow \wp(IfIn^T \times IfOut^T)$$

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. 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:

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

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 existing technical infrastructures.

Causality: In order to be realisable, 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:

$$\begin{aligned}
&ifIn_a \downarrow t = ifIn_b \downarrow t \Rightarrow \\
&\quad ifOutSet_a \downarrow t+1 = ifOutSet_b \downarrow t+1
\end{aligned}$$

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.

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 very flexible way to realize advanced functionality.

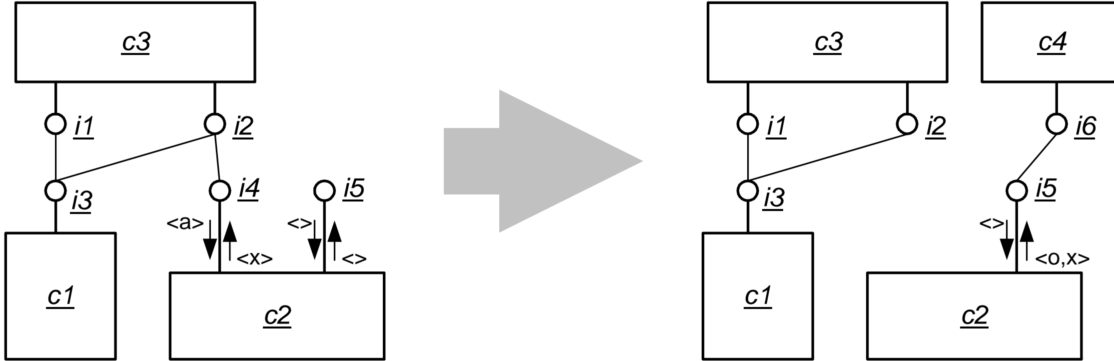

Figure 2: Component Behavior

Figure 2 visualizes a possible execution step of the component $c2$, including the 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 $i5$.

Behavior Relation: Similar to interfaces, we describe the sets of incoming and outgoing message sequences of a component by the following two evaluation functions:

$$\begin{aligned} CompIn &=_{def} Interface \rightarrow M^* \\ CompOut &=_{def} Interface \rightarrow M^* \end{aligned}$$

Note that the evaluation functions are partial, in general, as the behavior of a component usually depends only on a small subset from 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

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

which assigns combinations of input, output and configuration history 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:

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

In the following paragraphs, we state further restrictions on the behavior of components.

Causality: Similar to interface behavior, we require component behaviors to be causal. This is stated by the following condition for all components c:

$$\begin{aligned} &compIn_a \downarrow t = compIn_b \downarrow t \wedge conf_a \downarrow t = conf_b \downarrow t \Rightarrow \\ &\quad compOutSet_a \downarrow t+1 = compOutSet_b \downarrow t+1 \end{aligned}$$

where $compIn_a$ and $conf_a$ resp. $compIn_b$ and $conf_b$ denote two pairs of corresponding input histories and system configurations in the behavior of a component, while $compOutSet_a$ and $compOutSet_b$ denote the corresponding sets of related output histories. Note that the system configuration at time $t+1$ is not covered by the causality condition, as it is also affected by the behavior of other components. We could, of course, state an analogous causality condition that also holds for the system configuration by covering all relevant inputs of all components in the system.

The presented system model stresses the difference between the purely local input/output behavior and the global structural behavior. Another possible model would be to employ explicit structural change requests for each component, which then would affect the structure in the next time step. We do not use such a model here, as we wanted to abstract away from all details of the structure changing mechanism. A disadvantage of our approach is that it requires the definition of additional mechanisms in order to speak about the state of a component with respect to former connection change requests. This would, for example, be necessary to forbid components to delete instances that they have not created themselves.

4 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 syntactical 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, there is no means of subtyping for interface or component types, although some of the current technical componentware approaches, like Java Beans [Jav99b], provide it. However, we do not view subtyping as a central concept of componentware, because it may be easily replaced by delegation.

The proposed model does not yet provide support for mobile components. This would, for example, require the containment relation *parent* to change over time. However, we believe this extension is not sufficient—a more general approach requires the introduction of a location concept, as proposed in [BGR+99].

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. 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.

References

[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.

[BDRS97] Manfred Broy, Ernst Denert, Klaus Renzel, and Monika Schmidt (eds.). Software architectures and design patterns in business applications. Technical Report TUM-I9746, Institut für Informatik, Technische Universität München, 1997.

[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.

[Bro95] Manfred Broy. Mathematical system models as a basis of software engineering. *Computer Science Today*, 1995.

[Bro98] Manfred Broy. A logical basis for modular systems engineering. Internal paper, 1998.

[BRS+ar] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, and Manfred Broy. A Formal Model for Componentware. In *Foundations of Component-Based Systems*. Murali Sitaraman and Gary Leavens, to appear.

[BRSV98a] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. Technical Report I-9823, Technische Universität München, Institut für Informatik, 1998.

[BRSV98b] 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.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab system model with state. Technical Report TUM-I9631, Technische Universität München, Institut für Informatik, 1996.

[Gro99] UML Group. Unified Modeling Language. Version 1.3, Rational Software Corporation, 1999.

[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.

[Mic98] MicroSoft Corporation. MicroSoft COM homepage, `http://www.microsoft.com/com`, 1998.

[OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 2nd edition, 1998.

[Tec96] Technische Universität München, Institut für Informatik. *AutoFocus Project Page* `http://autofocus.informatik.tu-muenchen.de/`, 1996.

[Thu98] V. Thurner. A formally founded description technique for business processes. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Software Engineering for Parallel and Distributed Systems, PDSE98, 254-261*. IEEE Computer Society, 1998.