Use Case Maps
A Technique for Communicating and Validating Behavioral Aspects of Architectures

Hans de Bruin
Vrije Universiteit
Mathematics and Computer Science Department
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
tel: +31-20-4447745, fax: +31-20-4447653, e-mail: hansdb@cs.vu.nl

October 31, 2000

Abstract

This paper discusses Use Case Maps (UCM), a scenario-based technique for modeling behavioral aspects of a system at a very high abstraction level. UCM is an informal notation that is easy to comprehend by humans. Its strong point is to show how things work generally. As such it can be used in the communication with stakeholders, including the non-technical inclined stakeholders (e.g., end-users). The notation is not strong enough to reason about system behavior in a formal sense. However, for validation purposes, UCMs can be augmented with specifications of component interfaces in order to reason about component compositions formally. UCM is an overlooked notation that has not received the attention it deserves. In particular, it can play an important role in software architecture to bridge the gap between requirements and the rather detailed design models offered by the Unified Modeling Language (UML). The goal of this paper is to show how UCM can play an important role in communication with stakeholders and validation of behavioral aspects of an architecture.

1 Introduction

According to amongst others Bass et al. [1], software architecture serves the following purposes:

- architecture is the vehicle for stakeholder communication;
- architecture allows to gain insight in the quality of a system at an early stage;
- architecture manifests the earliest set of design decisions;
- architecture provides the means for planning and control;
- architecture is a transferable abstraction of a system.

In this paper we focus on software architecture as a vehicle for communication and quality control/assurance, in particular architecture validation. For effective communication with stakeholders, a notation is needed that can show system-wide behavior. This purpose can be fulfilled with a scenario-based technique called Use Case Maps (UCMs) [4, 3]. One of the strong points of UCMs is that

they can show multiple scenarios in one diagram and the interactions amongst them. This allows stakeholders to reason about a system as a whole instead of focusing on details.

UCM is an informal notation. This apparent shortcoming is precisely the reason why UCM can be used at high abstraction levels; low-level details are simply not part of the notation. By augmenting UCM with component interface specifications, it is possible to reason about component compositions formally. For instance, we discuss how the presence or absence of deadlock in a system can be detected. Another application area is to prove reachability properties, i.e., to verify whether a system can achieve certain goals or not.

The goal of this paper is to show that UCMs augmented with component interface specifications is a valuable technique for stakeholder communication and architecture validation. Naturally, UCM is not the final answer. For one thing, UCM concentrates on the behavioral aspects of a system and therefore its main application area is to assess runtime execution properties such as functionality, performance, security, and availability. Static properties, like modifiability and portability, are outside the scope of UCMs.

The paper is organized as follows. We start with a brief overview of the basic UCM notation and its semantics. Then we show how UCMs augmented with component interface specifications can be used to reason about system architecture properties in a formal way. Firstly, we argue that a grey-box approach to component specification is required. Secondly, aspects of component contracts are discussed. Thirdly, we demonstrate the role of UCM in a grey-box approach to component compositions. Fourthly, we discuss how component composition can be subjected to formal analysis in order to prove certain system properties. The paper is ended with some concluding remarks.

2 An Introduction to Use Case Maps

A UCM is a visual notation for humans to use to understand the behavior of a system at a high level of abstraction. It is a scenario-based approach showing cause-effects by traveling over paths through a system. UCMs do not have clearly defined semantics, their strong point is to show how things work globally.

The basic UCM notation is very simple. It is comprised of three basic elements: responsibilities, paths and components. A simple UCM exemplifying the basic elements is shown in Figure 1. A path is executed as a result of the receipt of an external stimulus. Imagine that an execution pointer is now placed on the start position. Next, the pointer is moved along the path thereby entering and leaving components, and touching responsibility points. A responsibility point represents a place where the state of a system is affected or interrogated. The effect of touching a responsibility point is not defined since the concept of state is not part of UCM. Typically, the effects are described in natural language. Finally, the end position is reached and the pointer is removed from the diagram. A UCM is concurrency neutral, that is, a UCM does not prescribe the number of threads associated with a path. By the same token, nothing is said about the transfer of control or data when a pointer leaves one component and (re-)enters another one. The only thing that is guaranteed is the causal ordering of executing responsibility points along a path. However, this is not necessarily a temporal ordering, the execution of a responsibility point may overlap with the execution of subsequent responsibility points.

A more realistic example is shown in Figure 2 depicting a distributed client-server system. Because the client communicates with the server over a network that can fail occasionally, a proxy server is

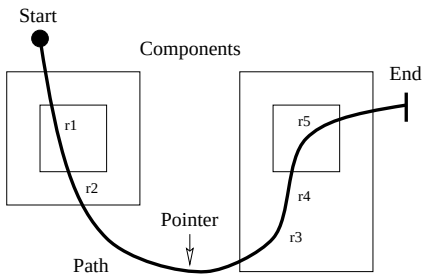


Figure 1: UCM basic elements.

included to provide transparent access to the real server. The proxy server is modeled as a stub for which two implementations are given: a transparent proxy server which passes the requests to and the replies from the server unaltered thereby denying the possibility of network failures, and a proxy server with a timeout facility with which failures are detected. The notation used in the figure is supposed to be self-explanatory.

It is interesting to see that many things are unspecified in UCMs, but the intended meaning is suggested strongly. For instance, distribution aspects (e.g., connection mechanism and the amount of concurrency in a component) are not dealt with. However, the client, the server and the proxy server are distinct components that are connected by a network, which is also modeled as a component. By using these names, it is natural to assume that the components are distributed over a number of computer systems. But again, it is not specified, it is all in the eye of the beholder.

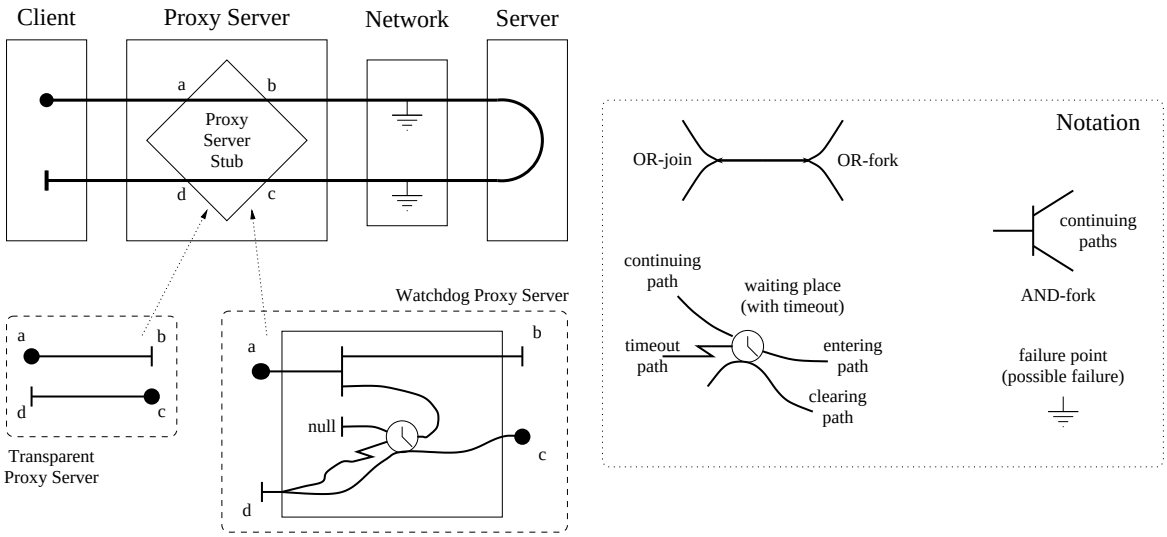


Figure 2: Distributed Client-Server UCM.

3 Reasoning with Use Case Maps

In the previous section we have illustrated how UCM can be used for modeling system behavior without committing to implementation details yet. Here we show how we can reason about systems formally by augmenting UCMs with component interface specifications. The material is this section is based on previously published papers. In one paper, we plea for a grey-box approach to component specification and composition [5]. We argue that some aspects of the component's internals must be exposed in order to successfully deploy a component. UCM seems to be a promising notation to capture the internal behavior of components and component compositions. In the other paper, it is shown how this approach can be used to reason about compositions in a formal way [7]. This section is intended to convey the flavor of the grey-box approach. For a detailed account, the reader is referred to the aforementioned papers.

3.1 Not White, not Black, but Grey-Boxing

Today, the notion of components is central in the development of software systems. The key idea is that a component encapsulates functionality, which can only be accessed through its interface published as part of the component's contract. In principle, we should favor a black-box approach to component deployment. That is, it should be possible to successfully deploy a component by just looking at its contract. Not only the functionality of a component, but also its non-functional properties, such as space and time requirements, must be specified unambiguously. Unfortunately, a black-box approach seems difficult to realize in practice. For instance, space and time requirements may depend on specific component usages, which may be hard to describe in a contract. Another problem is that a component may work perfectly in one setting, but may fail to operate correctly in a different one due to (possibly undocumented) assumptions made on the environment. This problem is known as architectural mismatch [8], and we will give an example later. These problems suggest a white-box approach with which we can investigate whether a component will perform correctly as part of a component system or not. However, it is not desirable to fully expose the internals of a component, since it can take a long time to master the details and we can become dependent on specific implementation details that might not survive the next releases of the component.

For the aforementioned reasons, we favor a grey-box approach that gives a high level view of the internals and clearly shows environmental constraints. We are not alone in our support for grey-box components. In [2], a justification for grey-box components is given following similar lines of reasoning. One can argue that a grey-box approach only partly describes the implementation and therefore is even worse than a white-box approach, which at least gives the full implementation. We do not agree with this point of view. By judiciously specifying the places where a component may be varied (e.g., extension or adaptation points), it is possible to avoid instable implementation dependencies. That is, a supplier of a component should guarantee that variation points remain invariant in subsequent releases. Moreover, a grey-box component can be specified without committing to a specific implementation yet. Such a specification can be seen as a type definition from which implementations can be derived all conforming to that type.

3.2 Aspects of Component Contracts

In this section, we take a closer look at the contents of the component contract. Before we can define required properties of components, we must give a definition of a component. Here we adopt the definition given by Szyperski [9]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

According to this definition, a software component should be regarded as a blueprint that can be instantiated. This notion is similar to an object being an instance of a class. Indeed, components are frequently comprised of classes, but this is not a prerequisite, it can also be a library of functions or even a set of macros. Also, the client of a component should not be able to tell the difference, unless certain parts of the component are made public through its *provides* interface.

A software component should be sufficiently self-contained in order to be subject to composition in third party products. In particular, a component should exhibit as little context dependencies as possible. If there are dependencies, these should be made explicit by means of a *requires* interface.

A contract states what the clients should do to deploy a component, it also states what services are provided by the implementer of the component. Obviously, an interface description comprised of operations and their signatures should be part of the contract. Many popular component standards do not go much further than this. However, only enumerating operations is not sufficient to successfully deploy a component. For one thing, a set of operations does not specify the behavior. This leaves us with the question of what else should be part of the contract. This is very much a research question. What follows is a tentative list of key elements of a contract.

Context dependencies Components are seldom useful in their own right. They typically require a context in which they can function. Frequently, a component framework provides the context. The component's dependence on the environment can be formalized in a requires interface.

Semantics The name and signature of an operation defined in the provides interface does not give the semantics of the operation, although the name of an operation may strongly suggest the provided functionality. Also, the set of operations does not prescribe the required sequences of operation invocations. Typically, both omissions are remedied by stating the pre- and post-conditions for each operation. Unfortunately, pre- and post-conditions do not give the complete semantics of a component since they only say something about the state of an instantiated component and they do not reflect the semantics of interactions with other components. To fully capture the semantics of a component and its behavior in a particular environment, part of the internals that specify inter-component interactions should be exposed in the contract.

Non-functional properties Besides defining the functionality of a component, it is also important to define non-functional properties such as space and time requirements. The non-functional properties must be included in the contract since it states whether a component can function in a system with a given upper bound of resources.

Configuration Typically, a component can be configured prior to instantiation. Such a configuration could be comprised of associations in the form of key-value pairs to initialize attributes. Also, generically defined components that can be instantiated with concrete types can be considered as a form of configuration. Configuration information should be part of the contract since it defines usages of a component.

As noted before, this list is not complete. More requirements will be added as we gain a better understanding in component specification and deployment. The current state of the art in component technology, which includes CORBA, (D)COM(+), Active-X, and JavaBeans, do not even address all the aforementioned contract issues. For this reason, they are often referred to as wiring standards.

3.3 Grey-Box Example

In our grey-box approach to component composition, UCMs are used for capturing the internal behavior of components and component compositions. The advantage of UCM of being a lightweight notation that can hide many details makes it well-suited for this purpose. The component interfaces are described in the programming language BCOOPL [6]. These interface specifications are akin to Java interface specifications but differ in the sense that not only operations are specified, but also when those operations may be invoked and by whom. A BCOOPL interface contains sequence information that is specified as a regular expression over interaction terms using familiar operations such as sequence (;), alternative (+), and repetition (*). An interaction term is comprised of client specifications denoting the parties that are allowed to invoked the operation, the name of the operation, input arguments, and a notification pattern. The latter specifies which notifications will be issued by an object.

As an example of a BCOOPL interface specification, consider a User Interface (UI) component like a button. The button is derived from from the base interface *UIComponent.*

```
interface UIComponent

interface Button extends [ UIComponent ] defines [
    Any ↦ (properties : PropertyTable) ⇒ ((arm() ; (disarm() ; arm()) * ; activate()[0,1] ; disarm())*) [
        Any ↦ setProperties (properties : PropertyTable) ⇒ () *
    ]
]
```

After a button has been created, it can be initialized by sending it an anonymous message with a property table as argument. A property table is a dictionary comprised of name-value pairs. For instance, to set the label of the button, it can be initialized with a property table that contains the name-value pair having *Label* as name and the desired string as value. A button supplies suitable default values for properties, so only properties that must be overruled should be included in a property table. If required, the values of properties can be changed during the life-time of a button by invoking *setProperties.*

The button's notification pattern captures the idea that if a mouse is moved inside a button area and the end-user presses a mouse button, an *arm* notification is sent. Moving the mouse outside the area causes a *disarm* notification, and moving the mouse back inside results in an *arm* notification again. When the mouse button is released while the mouse is positioned inside the button area, an *activate* notification is sent followed by a *disarm* notification. Nothing happens if the end-user releases the mouse button outside the button area.

The UCM notation has been augmented with some notational shorthands in order to have a better match with BCOOPL's language features. The augmentations are depicted in Figure 3. Notational shorthands have been provided for interface specifications, (asynchronous) message exchanges, and synchronization.

As an example of component composition, consider a dialog-box with which an end-user can be requested to enter a file name. The dialog-box is composed of several UI components, including buttons and a text field for entering a file name. The dialog-box acts as an intermediary synchronizing the notifications sent by its UI components thereby raising the abstraction level by providing a simple interface to its clients. Interacting with the dialog box results in issuing either a *fileName* notification or a *cancel* notification. A specification sheet is shown in Figure 4. Although, the specification of a dialog box is a toy example, it clearly shows how the principle of abstraction can be applied. The

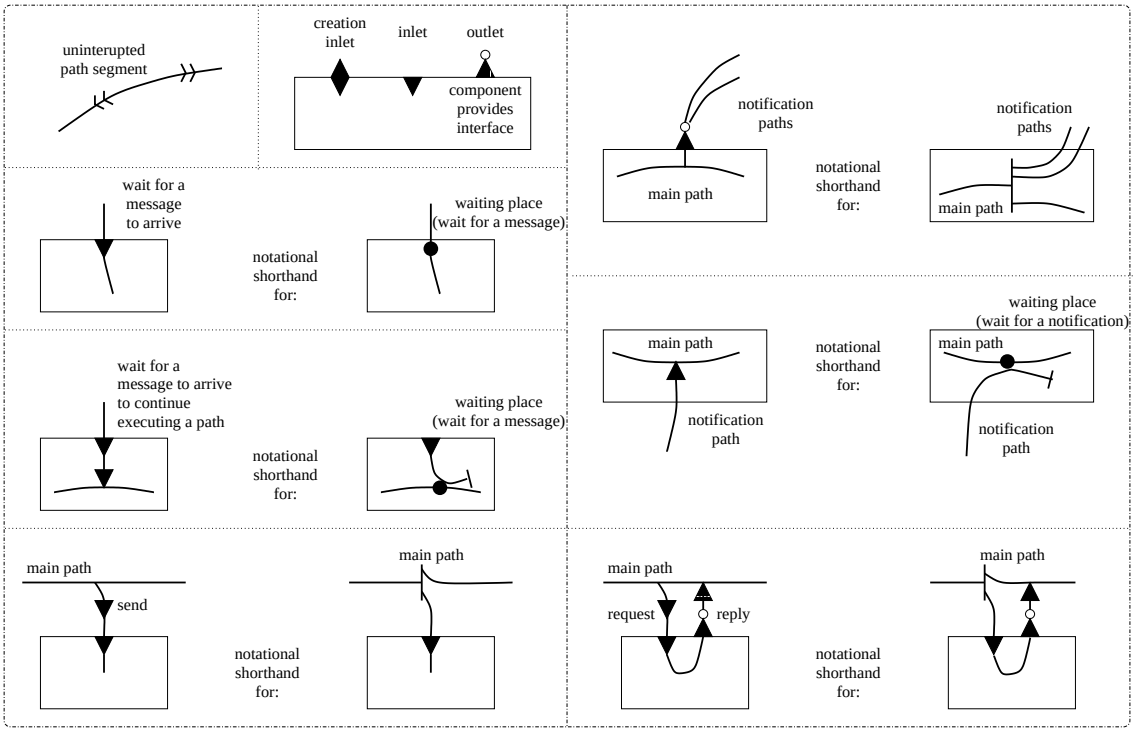


Figure 3: UCM augmentations.

internal behavior of the UI components that are used in the dialog-box (i.e., the buttons and the text field) is abstracted away. A black-box approach has been taken by only showing the notifications issued by the UI components that are actually handled by the dialog-box.

3.4 Validating a Component Composition

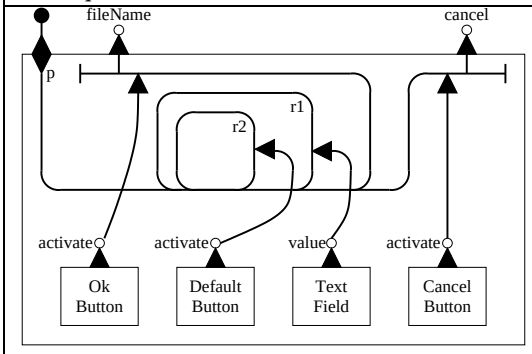
A simplified, but realistic model of a client/server system is used as an example to illustrate how component compositions comprised of UCMs and BCOOPL interfaces can be validated. The client and the server share a resource that can only be used by one party at a time. In this example we show that components may operate perfectly well in isolation, but they may fail when they are subjected to composition. This is a typical illustration of architectural mismatch, which was mentioned before.

The interface for the resource is shown below. A resource must be acquired before it can be used, and after it has been used, it must be relinquished to allow other objects to use it again. The interface for a resource could be specified as follows:

```
interface Resource defines [
    Any ↦ () ⇒ () [
        (
            Any ↦ acquire () ⇒ (done()) ;
            Any ↦ use (in : SomeInArg) ⇒ (result(out : SomeOutArg)) * ;
            Any ↦ relinquish () ⇒ (done())
        ) *
    ]
]
```

Specification Sheet: fileNameDialogBox

Description: A fileNameDialogBox is a UI component with which a user can be requested to enter a file name.



Properties		
Name	Type	Default Value
DefaultFileName	String	”component.doc”
X	Integer	0
Y	Integer	0
W	Natural	200
H	Natural	100

Responsibility Points:

p: the dialog box is displayed with the text field containing the file name obtained with the property named DefaultFileName, and is positioned at the coordinate obtained with the properties X and Y with a size given in the W(idth) and H(eight) property.

r1: the current file name becomes the string entered in the text field.
r2: the current file name becomes the default file name.

Provides Interface:

```
interface FileNameDialogBox extends [ UIComponent ] defines [
    Any ↦ (properties : PropertyTable) ⇒ (fileName(name : String) + cancel()) [
        Any ↦ setProperties (properties : PropertyTable) ⇒ () *
    ]
]
```

Requires Interface(s):

```
interface Button extends [ UIComponent ] defines [
    Any ↦ (properties : PropertyTable) ⇒ ((arm() ; (disarm() ; arm()) * ; activate()[0,1] ; disarm())*) [
        Any ↦ setProperties (properties : PropertyTable) ⇒ () *
    ]
]

interface TextField extends [ UIComponent ] defines [
    Any ↦ (properties : PropertyTable) ⇒ (value(text : String) *) [
        Any ↦ setProperties (properties : PropertyTable) ⇒ () *
    ]
]
```

Figure 4: FileNameDialogBox specification sheet.

This interface enforces cycles of acquiring, using, and relinquishing the resource.

When the server is brought together with a client that uses the same resource in a manner depicted in Figure 5 the system might come to a halt because of deadlock. Both the client and the server try to acquire the resource just after the client has sent a *service* request to the server. The system deadlocks if the client acquires the resource first. In that case, the client will never receive a reply (in the form of a *result* notification) from the server, since the server wants to acquire the resource but the resource

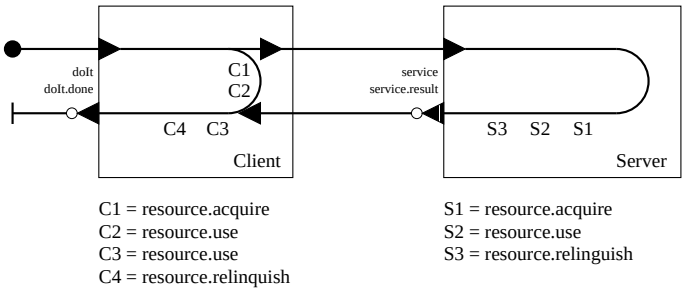


Figure 5: Client/Server/Resource interactions.

was already acquired by the client and will be relinquished only after the server has replied. So, the client as well as the server want to proceed, but they cannot do so because they are awaiting an event that will not happen. In other words, this is a classical example of deadlock. If, on the other hand, the server acquires the resource first, the system proceeds as intended. The server uses and then relinquishes the resource and continues with issuing a *result* notification. In the meantime, the client tries to acquire the resource and use it once before processing the result notification. Eventually, the client will succeed in doing so. After the client has processed the reply from the server, it proceeds with another use of the resource before relinquishing it.

It is beyond the scope of this paper to discuss the approach of analyzing the client/server system in detail. In [7], we have shown how a system can be subjected to analysis by validating the correct order of execution sequences of operations implied by UCMs and BCOOPL interfaces. By doing so, contradictions in temporal orderings can be detected, for instance, messages that are received before they are actually sent.

An interesting conclusion that can be drawn from this example is that a black-box approach to component composition will not suffice. The internal behavior of components must be exposed to a certain extent to pinpoint potential problems.

4 Concluding Remarks

In my experience, UCM is a versatile technique for modeling behavioral aspects of a system. On the one hand, they can be used in communication with stakeholders, including end-users, clients and commissioners. On the other hand, UCMs can form the basis for further developments. As such, they bridge the gap between requirements and detailed design models offered by UML. In addition, UCMs are scalable. Their application range from modeling low-level sub-systems such as dialog-boxes to high-level models of entire systems such as client/server systems. The expressiveness of UCMs can be explained from two UCM fundamentals. Firstly, the notation supports powerful abstraction mechanisms. Secondly, multiple scenarios and their mutual interactions can be visualized in one diagram.

Although UCMs do not have well-defined semantics, by augmenting them with component interface specifications we have the means to reason about component compositions formally. In this paper, we have discussed that the combination of UCMs and BCOOPL interfaces can be used to pinpoint problems like deadlock. A similar approach can be followed to assess other behavioral aspects. For instance, UCM responsibility points can be annotated with estimations of durations of particular ac-

tivities. This information can then be used to analyze whether performance requirements are satisfied or not.

UCM is a currently not a well-known technique. To remedy this problem, a Use Case Maps Web Page site [10] has been constructed devoted entirely to UCMs. Furthermore, there is work in progress to integrate UCM in UML. In my opinion, this would be a valuable addition, since UCM complement the design models of UML. Specifically, UCM operate on a much higher abstraction level than UML's behavioral counterparts being scenario (i.e., sequence and collaboration) diagrams, state-transition diagrams, and activity diagrams.

One final word, I strongly recommend to give UCMs a try as an architectural notation. You will not be disappointed.

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts, 1998.

[2] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical Report 122, Turku Centre for Computer Science (TUCS), Turku, Finland, August 1997. WWW: http://www.tucs.fi/publications/techreports/TR122.ps.gz.

[3] R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.

[4] R.J.A. Buhr and R.S. Casselman. *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[5] Hans de Bruin. A grey-box approach to component composition. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the First Symposium on Generative and Component-Based Software Engineering (GCSE'99), Erfurt, Germany*, volume 1799 of *Lecture Notes in Computer Science (LNCS)*, Berlin, Germany, September 28–30, 1999. Springer-Verlag.

[6] Hans de Bruin. BCOOPL: Basic Concurrent Object-Oriented Programming Language. *Software Practice & Experience*, 30(8):849–894, July 2000.

[7] Hans de Bruin. Scenario-based analysis of component compositions. In Greg Butler and Stan Jarzabek, editors, *Proceedings of the Second Symposium on Generative and Component-Based Software Engineering (GCSE'2000), Erfurt, Germany*, Lecture Notes in Computer Science (LNCS), Berlin, Germany, October 9–12, 2000. Springer-Verlag.

[8] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995. Carnegie Mellon University.

[9] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, New York, 1997.

[10] Use case maps web page. WWW: http://www.UseCaseMaps.org/.