# A Grey-Box Approach to Component Composition

Hans de Bruin

Vrije Universiteit, Amsterdam, The Netherlands
hansdb@cs.vu.nl

**Abstract.** Despite the obvious advantages of reuse implied by component technology, component based development has not taken off yet. Problems that inhibit general reuse include incomplete component contracts and (undocumented) dependencies of a component on the environment, which makes it hard to assess whether a component will behave in a particular setting as expected. In principle, a black-box approach to component deployment should be favored. In practice, however, we require information that cannot be described solely in terms of externally visible properties of components. For instance, non-functional properties (e.g., space and time requirements), environmental dependencies, and variation points (e.g., places where a component may be adapted or extended) do require insight in the internal construction of a component. In this paper, a grey-box approach to component deployment is discussed. It is based on a scenario-based technique called Use-Case-Maps (UCM), which uses scenarios to describe how several components operate at a high abstraction level. UCM is an informal notation. Its strong point is to show how things work generally. In order to reason about component compositions, we have augmented UCMs with formal specifications of component interfaces. These interface specifications have been borrowed from the concurrent, object-oriented language BCOOPL (Basic Concurrent Object-Oriented Programming Language). A BCOOPL interface is more than just a set of operations, it also describes temporal orderings of operations and the parties that are allowed to invoke a particular operation. The combination of UCMs and BCOOPL interfaces gives us the opportunity to document intra and inter component behavior at a high, but formal abstraction level.

## 1 Introduction

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 [9]. This suggests 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 [4], 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.

In this paper, we describe a grey-box approach to component-oriented system construction. The goal was to develop practical techniques for constructing systems out of components. With practical techniques we mean techniques that are easy to learn and to apply by software developers that do not necessarily have a formal computer science background. A second goal was to have a notation that can be used to validate the correctness of a system comprised of a composition of components. We use Use Case Maps (UCM) [5] for these purposes. UCM is scenario-based technique that bridges the gap between global requirement analysis models (e.g., use cases and class diagrams) and very detailed design models (e.g., interaction diagrams such as collaboration and message sequence diagrams). An important feature of a UCM is that it can show multiple scenarios in one diagram and the interactions amongst them.

However, a UCM is not a formal notation. Its strong point is that it can show in a glance how non-trivial systems work. Thus, UCMs alone are not sufficient to reason about system behavior in a formal sense. To suit our purposes, we annotate UCM with component interface descriptions. The notation for interfaces has been borrowed from the concurrent object-oriented programming language BCOOPL (Basic Concurrent Object-Oriented Programming Language) [6], a language specifically designed for component-oriented programming. An interface specification in BCOOPL is not just a collection of methods. It also describes the allowed sequences of method invocation and the parties that are allowed to

do so. As such, a BCOOPL interface is comparable with a message sequence diagram.

By combining UCMs with BCOOPL interfaces we obtain an intuitive notation that, on the one hand, is easy to understand and, on the other hand, can be used to reason about properties of component-oriented systems in a more formal way.

This paper is organized as follows. After a discussion on component-oriented programming in general, we explain our grey-box approach. First, a short introduction to UCMs and BCOOPL is given, followed by showing how these two techniques can be combined. We end this paper with discussing related work and we give some concluding remarks.

## 2 Aspects of Component-Oriented Programming

In this section, we discuss aspects of component-oriented programming. In particular, 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 [11]:

> 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 A Grey-Box Approach to Component Composition

As remarked before, the provides and requires interfaces of a component are not sufficient to fully capture the semantics of a component. Of course, a component can be understood by looking at its implementation, but this may be overwhelming, especially if a component has a complex behavior and interacts with many other components. Clearly, this level of detail is undesirable. What we need is a grey-box approach, which only exposes those details of a component that are required to assess different usages of a component. To this end, we combine Use Case Maps (UCM) [5] and BCOOPL [6] interface specifications. The result of

this combination is component specification technique that captures the behavior of a component at a high level of abstraction, but at the same time is precise enough to reason about compositions of components.

### 3.1 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.

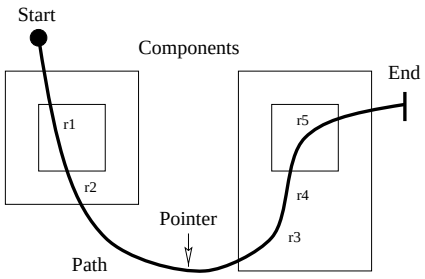



**Fig. 1.** UCM basic elements.

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 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.

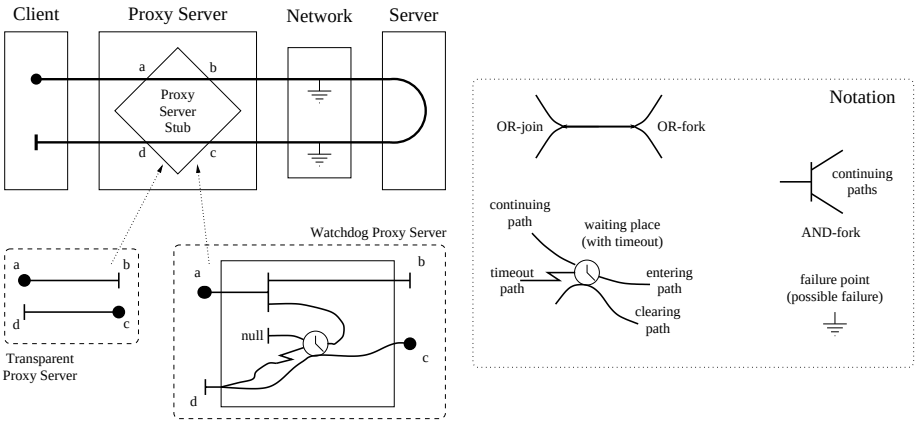



**Fig. 2.** Distributed Client-Server UCM.

### 3.2 BCOOPL

BCOOPL is a small, concurrent object-oriented programming language specifically designed to support component-oriented programming. BCOOPL has a long research history. It roots can be traced back to the concurrent object-oriented programming languages Procol [12] and Talktalk [3]. One of the strong points of BCOOPL is the built-in support of design patterns catering for component-oriented programming. In particular, BCOOPL supports the observer, the mediator and the bridge design patterns directly. Other design patterns frequently used in components, such as the facade and the proxy, can be implemented relatively easily in comparison with more traditional object-oriented programming languages like Java and C++.

**Language Features** BCOOPL is centered around two concepts: patterns and interfaces.

*Patterns* The concept of classes and methods specification have been unified in patterns and sub-patterns[1]. The term pattern has been borrowed from the object-oriented programming language Beta. The idea is that objects are instantiated from patterns and behave according to the pattern definition. A pattern describes the allowed sequences of primitives to be executed by an object after a message has been received in a so called *inlet*, which is implicitly defined in a pattern definition. A pattern may contain sub-patterns which also define inlets, and so on. When during the execution of a pattern one or more inlets are encountered, the execution is suspended until an appropriate message is received, which results in executing the corresponding sub-pattern. A pattern can therefore also be regarded as a synchronization mechanism coordinating the interactions with the object. It specifies when and which clients may communicate with an object. In this respect, a pattern resembles a protocol found in the concurrent object-oriented language Procol, which manages the access to an object.

A notification pattern is part of a pattern definition. It specifies the output behavior of a pattern in terms of notifications. Notifications are the OO abstraction of the call-back mechanism. In terms of design patterns, the call-back mechanism is known as the observer, the listener, the publish-subscribe and the dependency design pattern [8], while in the software architecture literature it is known as implicit invocation in event-driven systems [1]. An object interested in a particular notification of a publishing object can subscribe to that object. The subscription information is comprised of, amongst others, the name of the notification, the identity of the subscriber and the method to be invoked in the subscriber. A state change in a publisher will result in multicasting one or more notifications to the subscribers. Notifications are issued through an *outlet* by means of a *bang-bang* (!!) primitive. As a matter of fact, notifications are not only used for implementing the implicit invocation mechanism, but they are also used for getting a reply value as a result of sending a request to some object. The basic idea is to send a message to an object and then wait for a notification to be received in an inlet following the send primitive. The concept of notification patterns has been explored in Talktalk.

*Interfaces* The type or types of a pattern are provided by interfaces. A pattern that implements an interface has the type of that interface. Because a pattern may implement multiple interfaces, a pattern can have multiple types. A pattern implementing an interface must implement all the sub-patterns defined in the interface. The distinction between interface definition and interface implementation is not enforced by most popular languages. However, by adopting a disciplined programming style by using references to abstract classes only, the separation of interface and implementation can be realized in any object-oriented

[1] A pattern in BCOOPL should not be confused with a design pattern. The latter provides a solution for a design problem within a given context.

programming language. Java does support interfaces, although the use of interfaces is not enforced in Java.

In short, BCOOPL embraces the principle of programming to an interface. As in Java, multiple interface inheritance is supported in BCOOPL. That is, an interface may extend one or more sub-interfaces. Note that interface inheritance does not break encapsulation as is the case in implementation (e.g., class) inheritance. In contrast to Java, interfaces in BCOOPL contain sequence information specifying when a pattern may be invoked and by whom.

**Interface Specification** An interface is identified by a name and may extend one or more base interfaces. It is defined by means of an *interface interaction term*.

**interface** *Interface Name*
  **extends** $[$*interfaces*$]_{opt}$
**defines** [
  *interface interaction term*
$]_{opt}$

An interface interaction term is specified using the following syntax:

*client specifications* $\mapsto$ *Pattern Name* (*input args*) $\Rightarrow$ (*notification pattern*) [
  *regular expression over interface interaction terms*
$]_{opt}$

An interface interaction term corresponds with a (sub-)pattern definition that implements the interface. It defines the pattern name, the formal input arguments, a notification pattern that specifies sequences of notification messages, and client specifications. An interface interaction term is recursively defined as a regular expression over interface interaction terms leading to an hierarchical interface specification. The regular expression operators used for constructing an interface and their meaning are summarized in Table 1.

| Expression | Operator | Meaning |
|---|---|---|
| $\ll$ E $\gg$ | synchronize | E is executed uninterupted |
| E $\parallel$ F | interleave | E and F may occur interleaved |
| E + F | selection | E or F can be selected |
| E ; F | sequence | E is followed by F |
| E $*$ | repetition | Zero or more times E |
| E $[m,n]$ | bounded rep. | i times E with $m \leq i \leq n$ |

**Table 1.** Semantics of regular expression operators.

Client specifications denote the parties that are allowed to invoke the corresponding pattern. They are defined by any combination of the following: by interface name, by interface name set (specified with the @ modifier), or by

object reference set (specified with the *$* modifier). The sets are used to dynamically specify the clients that are allowed to interact. A pattern implementing such an interface is responsible for the contents of a particular set.

Notifications issued by a pattern are guaranteed to be emitted according to the defined sequences specified in its notification pattern. Note that a pattern may issue multiple and distinct notifications. A notification pattern is defined as a regular expression over notification terms that are specified as follows:

*Notification Name (output args)*

The co- and contra-variance rules apply for specifying interfaces. An interface interaction term may be redefined in a derived interface. The types of the input arguments must be the same as or generalized from the argument types of the base interface (i.e., contra-variance rule). In contrast, a notification pattern may be extended in a derived interface, both in terms of notification output arguments having derived interfaces (i.e., co-variance rule) and additional notifications.

The interface *Any* acts as a base type for every other interface. That is, every interface extends *Any* implicitly. *Any* is defined as:

**interface** Any

As an example of 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.

Note that this behavior cannot be deduced from the given notification pattern. For one thing, the relation between the notifications and the mouse events is not included in the interface specification. As will be shown later on, this relation can be clarified with a UCM.

### 3.3 Augmenting UCMs with BCOOPL Interface Specifications

In this section, we show how UCMs can be combined with BCOOPL interface specifications. The UCM notation has been augmented with extensions and notational shorthands in order to have a better match with BCOOPL's language features. The augmentations are depicted in Figure 3. The prime extension is the more rigorously defined semantics of a scenario in progress along a path within a component. As in BCOOPL, the *one-at-a-time* regime applies, which means that only one thread of control is active at any one time, although multiple threads may be in execution on an interleaved basis. However, a scenario in execution can claim exclusive control over a component by means of enclosing a path segment in $\ll$ and $\gg$ markers. Notational shorthands have been provided for the (creation) inlet, in which a message is received, and the outlet, which serves as a hook for connections to inlets.

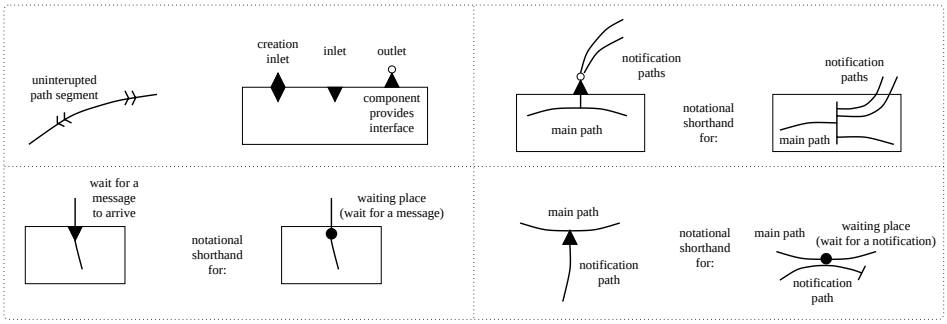



**Fig. 3.** UCM augmentations.

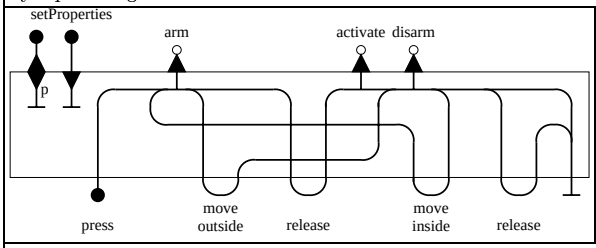
As an example, consider the previously introduced button again. The behavior of the button is captured in a UCM which is part of a so called specification sheet comprised of (see Figure 4):

- a description briefly describing the purpose of a component;
- a UCM showing the internal behavior of a component in terms of paths;
- a property table specification, consisting of the name of a property, its type and its default value;

– responsibility points describing in natural language how the state of a component is affected;
– the provides and requires interfaces specified as BCOOPL interfaces.

Specification Sheet: button

Description: A button is a UI component that allows the user to issue a UI command by "pushing" the button.




| Properties | | |
|---|---|---|
| Name | Type | Default Value |
| Label | String | "click on me" |
| X | Integer | 0 |
| Y | Integer | 0 |
| W | Natural | 100 |
| H | Natural | 50 |

Responsibility Points:

p: the button is displayed with a label obtained with the property named Label, 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.

press: the mouse button is pressed with the mouse pointer positioned inside the button area move outside: mouse pointer is moved outside the button area while keeping the mouse button pressed
move inside: mouse pointer is moved inside the button area while keeping the mouse button pressed
release: the mouse button is released

Provides Interface:

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

Requires Interface(s): *none*

**Fig. 4.** Button specification sheet.

Two kind of paths can be recognized, which can be defined in terms of external stimuli and notifications as follows:

press ; arm ;
( move outside ; disarm ; move inside ; arm ) * ;
release ; activate ; disarm

press ; arm ;
( move outside ; disarm ; move inside ; arm ) * ;
move outside ; disarm ; release

It is interesting to note that for each path infinitely many scenarios can be identified, since the mouse pointer can be moved inside and outside the button area infinitely many times.

The connection between a UCM and BCOOPL interfaces is established at two points:

- via the provides interface.
  Inlets (including the creation inlet) and outlets shown in a UCM must correspond with interface interaction terms in the provides interface.
- component interconnections via the requires interfaces.
  A component interconnection occurs when a path leaves one component and enters another one. If needed, an interconnection can be associated with a sub-expression of an interface. In most cases, however, an interconnection consists of linking an outlet of a used component to a path in the component being specified.

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. The specification sheet is shown in Figure 5. This example clearly shows how the principle of abstraction is applied. The 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. The UI components have a part-whole relation with the dialog-box. In particular, they exhibit the same life-time as the dialog-box and they are not shared with other components.

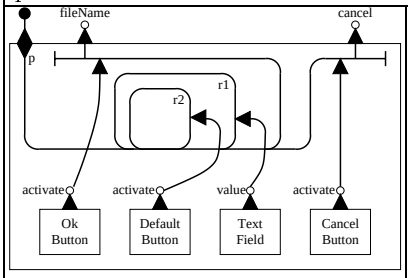
Although not shown in the examples, responsibility points can be used to state non-functional properties such as space and time requirements. This allows for the assessment of resource usage on a scenario basis. Responsibility points can also be used to further formalize a specification. At present, no such formalizations have been defined.

## 4 Discussion

In section 2, we have identified essential ingredients of a contract to successfully deploy a component. These include the specification of context dependencies, semantics, non-functional properties, and configurations. The combination of UCM and BCOOPL interfaces captures the essentials of a contract. Context dependencies are unambiguously specified with the requires interfaces specified as BCOOPL interfaces. They can be analyzed on a scenario-basis in the sense that typical usages of components or an agglomerate of components can be evaluated. The semantics of a component are defined by the combination of UCMs

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) ⇒ () *
    ]
]
```

**Fig. 5.** FileNameDialogBox specification sheet.

and a BCOOPL interface. The effects of a scenario in progress are described by responsibility points and by interactions with other components. These interactions interactions are formally captured in the requires interfaces. Responsibility points are not only used to describe internal state changes, albeit in a informal way, they can also be used for specifying non-functional aspects (e.g., resource usage), which can also be analyzed on a scenario-basis. Finally, a component is configured by means of a property table, which provides the means to vary and extend its behavior.

What is missing is a formal notation to describe the internal state of a component in terms of attributes. The semantics of a component can then be further formalized by associating each operation with pre- and post-conditions to describe when an operation may be invoked and its effects. This approach has been taken in Catalysis [7], a component-based development approach compliant with the UML notation. In our approach, the pre- and post-conditions are partly described in BCOOPL's interface specifications which specify when, how and by whom certain operations may be invoked. Attributes and attribute manipulations can be incorporated easily in a UCM by formalizing the use of responsibility points, but as noted before, this is part of future work when we gain more insight in the aspects of components that we want to model. This goes further than the functional behavior and includes non-functional properties.

An important concept provided by BCOOPL interfaces is the built-in support of the implicit invocation mechanism, which provides a flexible way to compose new components from existing ones. Components provide their services through notifications. It is up to a client to link to the required services. This approach to composition is similar to the approach taken in the Real-Time Object-Oriented Modeling (ROOM) technique [10]. The basic building block in ROOM is the actor, which has both a structural and a behavioral definition. The structural part is defined in terms of input and output ports of an actor. Each port is associated with a protocol, a kind of abstract data type defining data and routines. Actors exchange information by sending messages conform a particular protocol. The behavioral part of an actor is expressed in ROOMcharts, which can be characterized as an hierarchical and concurrent state machine.

It interesting to note that UCMs and ROOM have been combined in the UCM-ROOM design method [2]. This method defines a forward engineering path starting at a high abstraction level with UCMs and ending with detailed specifications in the form of ROOM models. Message Sequence Charts (MSC) are used to bridge the rather large conceptual gap between UCMs and ROOM models. UCMs serve a different role in our approach. Here UCMs are used to provide a high level, grey-box view of the internal behavior augmented with BCOOPL interfaces to formalize the specification. The intent is to reflect how an already existing component works, rather than providing the first step in a forward engineering process. For this reason, MSCs do not have an added value in our approach. Moreover, they are already contained implicitly in BCOOPL interfaces.

## 5 Concluding Remarks

We have argued that a grey-box approach to component specification is required to promote reuse. UCMs seem to offer a lightweight notation that reveals critical aspects of a component in a glance. A scenario view is given that shows typical usage of collaborating components. UCMs in combination with BCOOPL interfaces strikes the balance between, on the one hand, high level of abstraction, and on the other hand, preciseness. We have indicated that not only behavior can

be specified with UCMs, but they can also be used to express non-functional properties.

## References


1. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts, 1998.
2. F. Bordeleau and R.J.A. Buhr. The UCM-ROOM design method: from Use Case Maps to communicating state machines. Technical report, Department of System and Computer Engineering, Carleton University, Ottawa, Canada, September 1996. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/.
3. Peter Bouwman and Hans de Bruin. Talktalk. In Peter Wisskirchen, editor, *Object-Oriented and Mixed Programming Paradigms*, Eurographics Focus on Computer Graphics Series, chapter 9, pages 125–141. Springer-Verlag, Berlin, Germany, 1996.
4. 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.
5. R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
6. Hans de Bruin. BCOOPL: Basic Concurrent Object-Oriented Programming Language. WWW: http://www.cs.vu.nl/~hansdb/bcoopl/, 1999.
7. Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1998.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.
9. 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.
10. Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley and Sons, New York, 1994.
11. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, New York, 1997.
12. Jan van den Bos and Chris Laffra. Procol: a concurrent object language with protocols, delegation and persistence. *Acta Informatica*, 28:511–538, September 1991.