# Scenario-Based Analysis of Component Compositions

Hans de Bruin

Vrije Universiteit
Faculty of Sciences
Mathematics and Computer Science Department
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
hansdb@cs.vu.nl

**Abstract.** The behavior of a system comprised of collaborating components tend to be difficult to analyze, especially if the system consists of a large number of concurrently operating components. We propose a scenario-based approach for analyzing component compositions that is based on Use Case Maps (UCMs), but is extended with a few additional constructs for modeling component interfaces and connections. UCMs provide a high level, behavioral view on a system that is easy to comprehend by humans. However, UCMs do not have well-defined semantics. For this reason, UCMs are augmented with formal component interface specifications as used in the concurrent, object-oriented programming language BCOOPL. The combination of UCMs and BCOOPL interface specifications enables formal analysis of component compositions. This involves two steps. In the first step, UCMs and BCOOPL interface specifications are translated into a BCOOPL program. In the second step, the interactions between components are analyzed for system properties like deadlock and reachability. An important result of the combination is that the complexity, which arises when concurrently collaborating components are brought together, is tamed by considering only those usages of components that are actually specified in UCM scenarios.

## 1 Introduction

A high level design notation targeted to model component compositions should serve two purposes. On the one hand, the notation should be easy to understand by humans, in particular, designers. A designer should gain insight in the behavioral aspects of the system as a whole almost effortlessly. On the other hand, the notation should be precise enough to reason about compositions.

The behavior of a system can be defined with formalisms like temporal logics and traces of externally observable events, which can be represented formally and graphically with notations such as Petri nets [14] and StateCharts [12]. With suitable abstraction mechanisms we can then define the components and events of interest to gain insight in the overall behavior. However, this approach does not give the big picture. What is needed is a notation with which designers can reason about the behavior at a high abstraction level. This purpose can be fulfilled with a scenario-based technique called Use Case Maps (UCMs) [2] as will be shown in a couple of examples. One of the strong points of UCMs is that they can show multiple scenarios in one diagram and the interactions amongst them. This allows a designer to reason about a system as a whole instead of focusing on details.

Real-size systems tend to be hard to analyze. Especially if a system is composed of concurrently running components, the number of possibilities to consider explodes exponentially. The complexity, which is caused by non-deterministic behavior, can be tamed by considering only those scenarios that can really happen. In particular, a component may provide a number of scenarios (behaviors) of which only a subset is actually used in a system. Thus, the scenarios provided by UCMs gives us a handle to prune all non-occurring behaviors of components beforehand.

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 will show in an example 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.

In this paper, we combine UCMs with BCOOPL (Basic Concurrent Object-Oriented Programming Language) interface specifications, which not only detail how operations should be invoked, but also when operations may be invoked and the parties that are allowed to do so [8]. The temporal orderings of operation invocations provide a good starting point for analyzing whether a system comprised of components violates the imposed orderings or not. Apart from the aspects mentioned above, BCOOPL supports other language features that are useful for component-oriented programming, such as the built-in support for the Observer design pattern for minimizing the coupling between components, delegation for supporting black-box composition, concurrency for dealing with distributed computing transparently, and component access control for addressing security issues. These language features allow components to be specified at a higher abstraction level with respect to general purpose languages like C++ and Java. In fact, BCOOPL can be regarded as a design language with which executable designs can be specificed. The combination of UCMs and BCOOPL interface specifications can be translated unambiguously into a BCOOPL program. This BCOOPL program can then be subjected to analysis in order to prove certain system properties such as deadlock and reachability.

This paper is organized as follows. We start with an overview of UCM and BCOOPL. Next we introduce a running example to exemplify the two steps involved in analyzing a system. This is followed by a detailed explanation of the two steps: translating a specification into a BCOOPL program and analyzing the resulting program for system properties. We end this paper with a discussion and concluding remarks.

## 2 Use Case Maps and BCOOPL Preliminaries

The material in this paper builds on previous work in which a grey-box approach to component specification is argued [6]. 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. The combina-

tion of UCMs and BCOOPL interfaces gives us the opportunity to document intra and inter component behavior at a high, but formal abstraction level.

### 2.1 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.

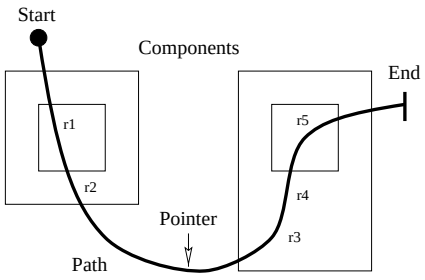



**Fig. 1.** UCM basic elements.

The UCM notation is quite rich, and only a small subset is used in this paper. In Figure 2, two frequently used UCMs constructs are shown. The AND-construct is used to spawn multiple activities along parallel paths. When a pointer reaches an AND-fork, this pointer is removed from the diagram and replaced by two pointers at the beginning of the parallel paths. An AND-join acts as a synchronization mechanism. When both pointers have reached the AND-join, they are replaced by a single pointer and execution is continued along the path following the join. The OR-construct should be interpreted

as a means to express multiple scenarios in a single diagram. It states that multiple scenarios are comprised of identical paths. Therefore, it is not necessary to specify conditions detailing the path to be followed at an OR-fork.

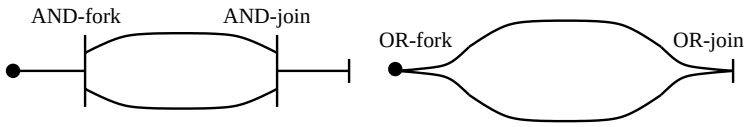



**Fig. 2.** Additional UCM notation.

## 2.2 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 path expressions [4], and the concurrent object-oriented programming languages Procol [15] and Talktalk [1]. One of the strong points of BCOOPL is the built-in support of design patterns catering for component-oriented programming (see [10] for a pattern catalog). In particular, BCOOPL supports the Observer, the Mediator and the Bridge design patterns directly. The Observer and Bridge pattern are particular useful for specifying stand-alone components, while the Mediator pattern is used to mediate interactions between those components [7]. 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++. A detailed account of BCOOPL and some of its application areas can be found in [8], which also addresses implementation issues.

**2.2.1 Core Language Features** BCOOPL is centered around two concepts: interfaces and patterns. An interface defines the operations that must be implemented by an object that conforms to that interface. By adhering to the principle of programming to an interface, a certain amount a flexibility is added to a system since new implementations can be provided without breaking existing code. A BCOOPL interface is specified as an augmented regular expression over operations. It not only describes how an operation can be invoked, but also when and by whom.

Class and method definition have been unified in patterns and sub-patterns. The term pattern has been borrowed from the object-oriented programming language Beta [13]. 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. It is specified as a regular expression over primitives using the same operators as in interface specifications. A pattern may contain sub-patterns which also define inlets, and so on. A top-level pattern can be seen as a class definition, whereas sub-patterns can be seen as (sub-)method definitions.

A notification pattern is part of a pattern definition. It specifies the output behavior of a pattern in terms of notifications. An object interested in a particular notification of a publishing object can subscribe to that notification. The subscription information is comprised of, amongst others, the name of the notification, the identity of the subscriber and the pattern to be invoked in the subscriber. 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 Observer design pattern, 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 [1].

The type or types of a pattern are provided by interfaces. A pattern that implements an interface has the type of that interface. As in Java, multiple interface inheritance is supported in BCOOPL. That is, an interface may extend one or more sub-interfaces. In contrast to Java, BCOOPL interfaces contain sequence information.

**2.2.2 Computational Model** The computational model of BCOOPL is based on message passing and concurrent objects. Objects are instantiated from patterns by executing the *new* primitive. A pattern may contain sub-patterns and their corresponding objects are instantiated implicitly whenever a super-pattern is instantiated. A conceptual model of an object and its sub-objects is shown in Figure 3. Each object has an unique I.D., which is used as an address for message exchange. Objects communicate with other objects by means of a restricted from of asynchronous message passing in the sense that the partial ordering of messages sent from one object to another is preserved. An object receiving a message does not process the message right away, instead the message is placed in an unbounded message buffer. The dispatcher searches the message buffer on a *first-come-first-served* basis of acceptable messages. The acceptability of a message is determined by the state of patterns in execution. If an acceptable message is found, the dispatcher passes the message on to the corresponding (sub-)object's inlet, otherwise it waits for the arrival of new messages. Thus, the communications between objects can be summarized as synchronous message buffering, but asynchronous message processing.

A Computation in BCOOPL is achieved by sending messages. This includes control flow structures like selection (*if-then-else*) and repetition (*while-do*). A computation proceeds as follows. After a message has been received in an inlet, the sequence of primitives following the inlet are executed until one or more inlets (sub-patterns) are encountered. Regular expression operators, such as the selection (+) and the repetition (*), imply choices. Each branch resulting from such a choice must be guarded with an inlet. That is, the choice to follow a particular branch is made by sending an appropriate message. There is no such concept as non-deterministic choices.

Within an object and its sub-objects the *one-at-a-time* principle of executing primitives applies. Multiple execution threads may occur within a tree of (sub-)objects, which are introduced with the interleave operator ($\|$). At most one thread, however, is active at any one time. Thread switching occurs at the time the active thread runs into one or more sub-patterns. Because almost every computation step is expressed in terms of message passing, thread switches occur frequently, which amounts to a semi-parallel object model. The next active thread is selected on the basis of the acceptability of the pending

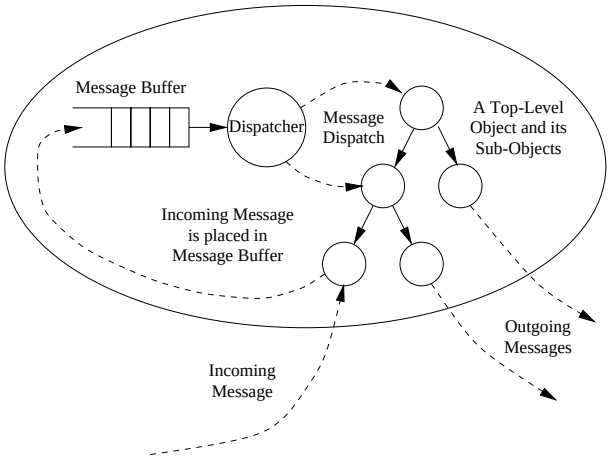



**Fig. 3.** Object model.

messages in the message buffer. This is a fair scheduling mechanism since the active thread cannot interfere with the scheduling of other threads, unless it claims explicitly the exclusive ownership of the object by means of the synchronize operator ($\ll expr \gg$). In contrast to intra-object concurrency, top-level objects (and their sub-objects) operate on a truly concurrent basis.

**2.2.3 Interface and Pattern 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 a hierarchical interface specification. The regular expression operators used for constructing an interface and their meaning are summarized in Table 1.

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

| Expression | Operator | Meaning |
|---|---|---|
| $\ll$ E $\gg$ | synchronize | E is executed uninterupted |
| E $\|\|$ 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.

interface name set, or by object reference set 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. 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 the interface for a scarce resource. 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 read as follows:

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

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

Patterns and sub-patterns are defined identically. The overall structure of a pattern is the following:

*client/server specifications* ↦ **pattern** *Pattern Name* (*input args*) ⇒ (*notification pattern*)
    **implements** [*interfaces*]$_{opt}$
    **declares** [*local variables*]$_{opt}$
**does** [
    *pattern implementation; a regular expression over primitives and sub-patterns*
]

Client/server specifications are defined in patterns similar to client specifications in interfaces. Client specifications identify the kind of objects that may communicate with a pattern, whereas server specifications denote declaratively specified linkages to notifications. The syntax for server specifications is as follows.

*Local Variable.Pattern Name*$_1$*.Pattern Name*$_2$*.*$\cdots$*.Pattern Name*$_n$*.Notification Name*(*formal arguments*)
*Object Set.Pattern Name*$_1$*.Pattern Name*$_2$*.*$\cdots$*.Pattern Name*$_n$*.Notification Name*(*formal arguments*)

Notification linkages are established at run-time. An assignment to a variable involved in notification linkage results in first abolishing the current link, provided the variable is not *nil*, then assigning to the variable, and finally establishing a new link if the variable does not equal *nil*. In the case of an object set, a notification link is established when an object is added to the set, likewise it is abolished when the object is removed from the set.

The behavior of a pattern is defined in the *does* section. It is defined as a regular expression over primitives and sub-patterns. The supported primitives are summarized in Table 2. As an example, an implementation of the *Resource* is given below.

```
Any ↦ pattern resource () ⇒ () implements [ Resource ] does [
    (
        Any ↦ pattern acquire () ⇒ (done()) does [ !! done() ] ;
        Any ↦ pattern use (in : SomeInArg) ⇒ (result(out : SomeOutArg))
            declares [resultArg : SomeOutArg ; ]
        does [
            // use the resource
            resultArg := ··· ;
            !! result(resultArg)
        ] * ;
        Any ↦ pattern relinquish () ⇒ (done()) does [ !! done() ]
    ) *
]
```

### 2.3 Augmenting UCMs with BCOOPL Interface Specifications

The UCM notation has been augmented with an extension and some notational shorthands in order to have a better match with BCOOPL's language features. The augmentations are depicted in Figure 4. The 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 interface specifications, (asynchronous) message exchanges, and synchronization.

## 3 Running Example

A simplified, but realistic model of a client/server system is used as a running example. 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 a component, in this particular case the server, operates perfectly well in isolation, but it may fail when it is subjected to a composition in which another component, in this case the client, uses the same shared resource. This

| Primitive | Abstract Syntax | Remarks |
|---|---|---|
| Assignment | *variable* := FQN*expression* | A FQN (Fully Qualified Name) denotes an object. It is specified as: *(Pseudo-)Variable.Pattern Name*$_1$.$\cdots$.*Pattern Name*$_n$ |
| New | **new***Pattern Name* | The designated pattern is instantiated along with its subpatterns resulting in an object tree. Unreferenced objects are reclaimed by a garbage collector. |
| Send | FQN (*message args*) | A message is sent to the object designated with the FQN |
| Request | **request** FQN (*message args*) | Identical to a send with the exception that a reply (i.e., a notification) is sent only to a subobject of the object that issued the request. |
| Inlet (Pattern) | *CS specs* $\mapsto$ **pattern** *Name* (*in args*) $\Rightarrow$ (*notifications*) **does** [ $\cdots$ ] | A message is received in an inlet which is implicitly defined in a pattern. |
| Lightweight Inlet (Pattern) | *CS specs* $\mapsto$ **pattern** *Name* (*vars*) | In contrast with an ordinary inlet, a lightweight inlet does not introduce a local scope. The received message arguments are stored in the designated variables. |
| Outlet | !!*Notification Name* (*message args*) | A notification is issued. |
| Client/Server Set Operations | *add* or *remove* | The add and the remove are currently the only supported operations. |
| Delegate | *not discussed further* | |

**Table 2.** Primitives.

is a typical illustration of architectural mismatch where components make (possibly undocumented) assumptions on the environment [11]. The interface for the server is shown below, the interface and the implementation of the resource was already given in a previous section.

```
interface Server defines [
    Any ↦ (resource : Resource) ⇒ () [
        Any ↦ service (in : SomeInArg) ⇒ (result(out : SomeOutArg)) *
    ]
]
```

A server is initialized with a resource. Most likely, the server will use this resource, although it is not clear when the resource is used. This kind of behavior specification cannot be deduced from the interface specification. For this and other reasons not discussed here we have proposed a grey-box approach for component specification comprised of UCMs and BCOOPL interface specification [6]. The combination of UCMs

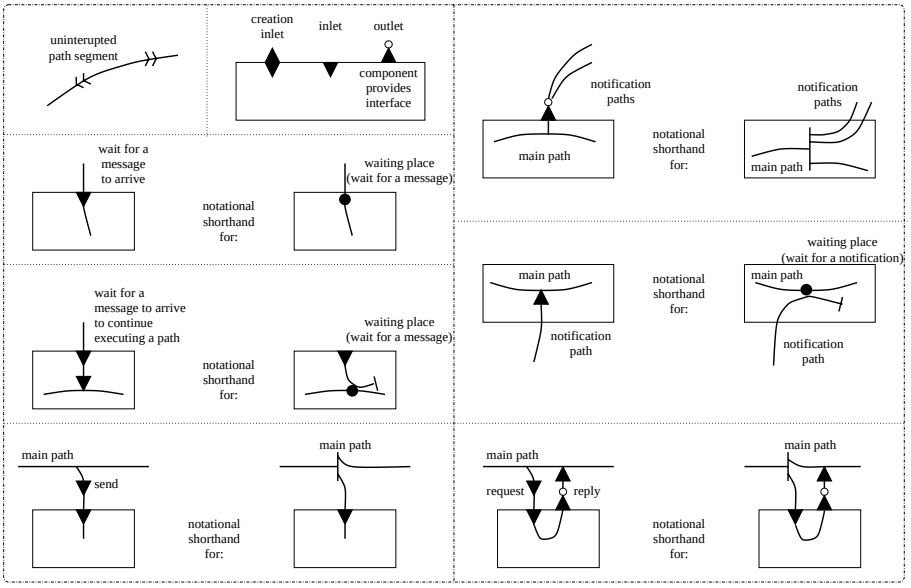



**Fig. 4.** UCM augmentations.

and BCOOPL interfaces gives us the opportunity to reason about the internal behavior of components and their interactions with other components.

The interactions between the server and the resource is depicted in Figure 5. A simplification is shown in the same figure in which the interactions between the two are modeled with responsibility points and the resource component is removed from the graphical representation altogether.

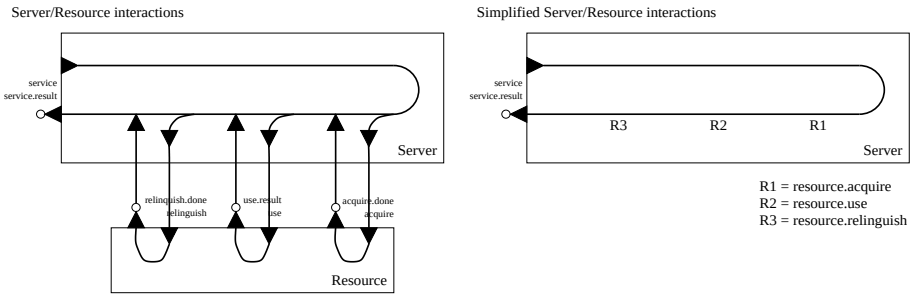



**Fig. 5.** Server/Resource interactions.

When the server is brought together with a client that uses the same resource in a manner depicted in Figure 6 the system might come to a halt because of deadlock. The interface for the client is given below.

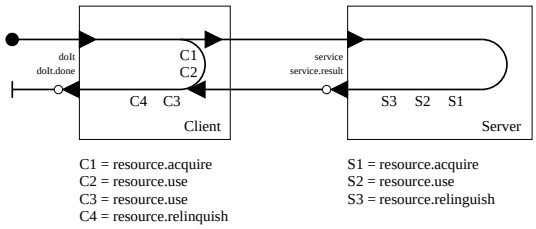



**Fig. 6.** Client/Server/Resource interactions.

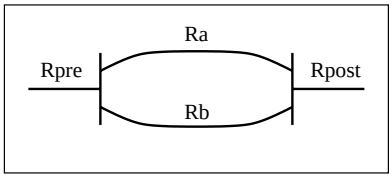
```
interface Client defines [
    Any ↦ (server : Server, resource : Resource) ⇒ () [
        Any ↦ doIt (in : SomeInArg) ⇒ (done(out : SomeOutArg)) *
    ]
]
```

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 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.

## 4 Preparing UCM/BCOOPL Interface Specifications for Analysis

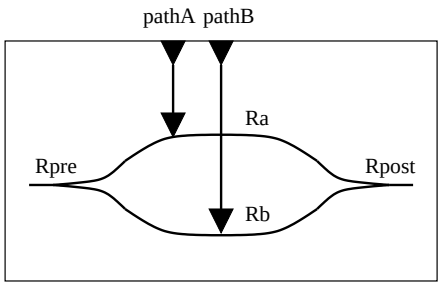
A component composition comprised of UCMs and BCOOPL interface specifications is translated into a corresponding BCOOPL program before it is analyzed. The translation can be done in an automatic and unambiguous way. We demonstrate how typical UCM constructs, like the AND/OR-fork/join and the request, translate into BCOOPL. The translation scheme for the AND-fork/join is given in Figure 7. The basic idea is to send two (asynchronous) messages to patterns defined in the object itself. The patterns that handle these messages are contained in an interleave (||) expression, so these two patterns are executed concurrently.




```
    ⋮
    ⋮
R_pre ;
self.pathA() ;
self.pathB() ;
(
    self ↦ pattern pathA () ⇒ () does [
        R_a
    ]
‖
    self ↦ pattern pathB () ⇒ () does [
        R_b
    ]
) ;
R_post ;
    ⋮
    ⋮
```

**Fig. 7.** Translation scheme for the AND-fork/join.

The translation scheme for the OR-fork/join is along the same lines as for the AND-fork/join as is depicted in Figure 8. The forked paths are embedded in an alternative (+) expression rather than an interleave expression. In contrast to the AND-fork/join, only one path will actually be executed. The path to be executed is selected on the basis of message receipt in an inlet defined at the beginning of a forked path.




```
    ⋮
    ⋮
R_pre ;
(
    Any ↦ pattern pathA () ⇒ () does [
        R_a
    ]
+
    Any ↦ pattern pathB () ⇒ () does [
        R_b
    ]
) ;
R_post ;
    ⋮
    ⋮
```

**Fig. 8.** Translation scheme for the OR-fork/join.

As discussed before, a request is split into a message send followed by an inlet in which the reply notification is received. In between the message send and the receipt of a reply in an inlet, an object can perform additional computations that run concurrently with the activities of the object that services the request. The translation scheme for the request is given in Figure 9.

Given these translation schemes, we show in Figure 10 how the client/server example is translated into BCOOPL.

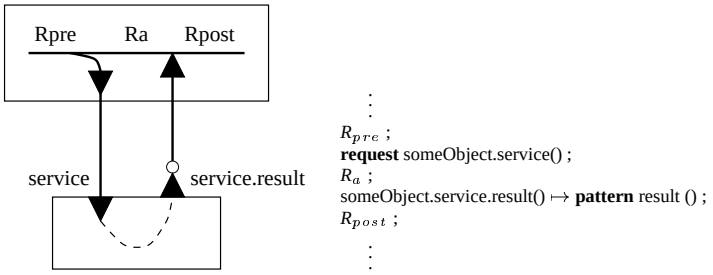



**Fig. 9.** Translation scheme for the request.

## 5 Analyzing Compositions

Finally, we show how a generated BCOOPL program is analyzed. This involves the following three steps:

1. type inferring pattern expressions;
2. building a communication graph annotated with thread information;
3. analyzing the composition with temporal orderings.

The first two steps are used to represent the program in such a way that it can be readily used in the final, analysis step.

The first step is type inferring the generated BCOOPL program in order to identify the types of pattern expressions as precisely as possible. Type information is used to deduce communication targets. In principle, communication targets can be distilled from a UCM. However, to better deal with polymorphism, type inferring a BCOOPL program yields more accurate type information. In addition, type information is also used to check the program's sanity. For instance, we can verify whether a variable that holds a target for a message send is initialized before it is used. Due to lack of space, type inferring BCOOPL programs is not discussed in this paper. A detailed discussion of type inference in Talktalk (a predecessor of BCOOPL), which is based on data flow equations, can be found in [5].

The second step is to build a communication graph detailing the *send-to* and *received-by* relations between (sub-)patterns. This graph is akin to a call graph for imperative programs. The communication graph is annotated with thread information to show the threads in which message sends and receipts occur.

In the final third step, the composition is analyzed by constructing directed graphs that show the temporal orderings between events (i.e., a message send and a message dispatch). Such graphs are constructed for each scenario specified in a UCM. Any violation of ordering constraints in a graph can be seen as a problem indicator.

For analysis purposes, BCOOPL's computational model is slightly adapted in order to simplify reasoning about component compositions without actually changing the semantics. A single message buffer is used instead of equipping each component with its own buffer. In reality, BCOOPL objects are equipped with individual buffers, which

```
// Client implementation
Any ↦ pattern client (client : Client, resource : Resource) ⇒ () implements [ Client ] does [
    Any ↦ pattern doIt (in : SomeInArg) ⇒ (done(out : SomeOutArg)))
        declares [ resultArg : SomeOutArg ; ]
    does [
        server.service(in) ;
        request resource.acquire() ; resource.acquire.done() ↦ pattern resourceAcquired () ;
        request resource.use(in) ; resource.use.result(out : SomeOutArg) ↦ pattern resourceResult (resultArg) ;
        request server.service.result(out : SomeOutArg) ↦ pattern serverIsDone (resultArg) ;
        request resource.use(in) ; resource.use.result(out : SomeOutArg) ↦ pattern resourceResult (resultArg) ;
        request resource.relinquish() ; resource.relinquish.done() ↦ pattern resourceRelinquished () ;
        !! done(resultArg)
    ] *
]

// Server implementation
Any ↦ pattern server (resource : Resource) ⇒ () implements [ Server ] does [
    Any ↦ pattern service (in : SomeInArg) ⇒ (result(out : SomeOutArg))
        declares [ resultArg : SomeOutArg ]
    does [
        request resource.acquire() ; resource.acquire.done() ↦ pattern resourceAcquired () ;
        request resource.use(in) ; resource.use.result(out : SomeOutArg) ↦ pattern resourceResult (resultArg) ;
        request resource.relinquish() ; resource.relinquish.done() ↦ pattern resourceRelinquished () ;
        !! result(resultArg)
    ] *
]

// System configuration (main start-up code)
Any ↦ pattern main () ⇒ ()
    declares [
        client : Client ;
        server : Server ;
        resource : Resource ;
        inArg : SomeInArg ;
        outArg : SomeOutArg ;
    ]
does [
    client := new client ;
    server := new server ;
    resource := new resource ;
    inArg := new someInArg ;

    resource() ;
    client(server, resource) ;
    server(resource) ;
    request client.doIt(inArg) ; client.doIt.done(out : SomeOutArg) ↦ pattern clientIsDone (outArg)
]
```

**Fig. 10.** Client-Server system translation into BCOOPL.

is of course a necessity for truly concurrent and distributed computing. The following rules hold:

- The temporal ordering of messages sent by one component is preserved in the message buffer.
- Message dispatching is based on a first-come-first-served basis of acceptable messages as far as an individual object (and its sub-objects) is concerned. Each object publishes its acceptable (sub-)patterns (i.e., there corresponding (sub-)objects are ready to be executed). A non-deterministic scheduling policy applies for choosing the next object to be executed.

As a consequence, the following general rule is valid: $t_{message\ buffering} < t_{message\ dispatching}$ (abbreviated as $t_{buf} < t_{disp}$). In words, a message is buffered before it is dispatched.

There is a one-to-one correspondence between regular expressions and state machines. Therefore, interface specifications, which are defined as regular expressions over interface interaction terms, imply temporal orderings. A typical example is given in the interface specification of a resource, which states that a resource cannot be used before it is acquired, and that it cannot be acquired again before it is relinquished. This can be formalized as shown below.

$$simplified\ Resource\ interface\ specification: \ (acquire; use*; relinquish)*$$

$$t_{acquire\ disp_i} < t_{use\ disp_{ij}} < t_{use\ disp_{ij+1}} < t_{relinquish\ disp_i} < t_{acquire\ disp_{i+1}}$$
$$with\ index\ i\ denoting\ the\ outer\ repetition\ cycle\ number, and$$
$$with\ index\ j\ denoting\ the\ inner\ repetition\ (use)\ cycle\ number$$

The temporal orderings can be shown as a directed graph in which the $<$ relation is represented as a directed edge between two nodes.

We are now in the position to analyze a composition on a scenario basis. Before discussing how all scenarios can be enumerated systematically, we show the general approach by taking the client/server system as an example. In this particular case, two scenarios can be recognized. In the first scenario, the client acquires the resource first. The temporal event orderings of this scenario are depicted in a slightly simplified form in Figure 11 (for one thing, the invocations of the resource's *use* pattern have been omitted). As can be seen in the figure, the scenario leads to a situation in which a notification (the *result* notification issued by the server) is dispatched before it is issued (i.e., buffered). This is a violation of the rule that a message just be buffered before it can be dispatched. Therefore, we conclude that the system comes to a halt as a result of deadlock. A second scenario can be devised in which the server requires the resource first. Due to space limitations we will not give a graphical representation here. In this case, however, no contradictions are found.

The enumeration of all scenarios is done by systematically considering all paths in a UCM. In particular, the OR-fork introduces two independent (sub-)scenarios, whereas the AND-fork leads to two concurrently running threads. In principle, all permutations of events in concurrent threads yield the set of scenarios to consider, which obviously leads to an explosion of scenarios. Fortunately, there are ways to prune the scenario set. First of all, we can check whether two threads are independent of each other by consulting the communication graph annotated with thread information. If components involved in communications originating from one thread do not use the components invoked from the other thread, then from an analysis point of view the execution of both threads can be regarded as a single scenario. Secondly, the temporal orderings implied by pattern expressions enforce particular message sequences (e.g., the *acquire ; use * ; relinquish* message sequence of a resource). This means that certain permutations of events in concurrent threads do not apply. For instance, in the client/server example, a resource can only be used if it has been acquired. Therefore, we do not have to consider the permutations of all resource *uses* stemming from the client and the server.

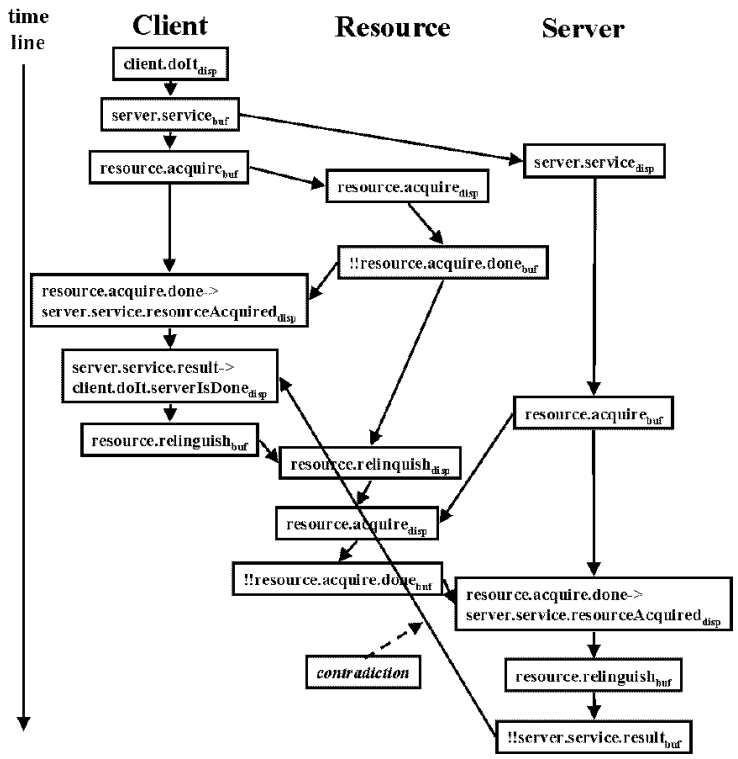



**Fig. 11.** Scenario leading to deadlock.

In conclusion, by analyzing temporal orderings, we might find contradictions which are an indication for errors in a specification, such as the presence of deadlock. Thus, erroneous behavior is found by considering all *possible* scenarios.

## 6 Discussion

We have shown how the presence or absence of deadlock in a system comprised of components can be proved. Another application area of this approach is to investigate reachability, a concept closely related to the notions of goals. By analyzing a scenario, we can verify whether the intended goals of that scenario are reached or not. This notion can be reformulated in terms of events. That is, we can analyze whether a particular event (message sending or message dispatching) or a set of events that is associated with a certain goal does happen.

As discussed before, systems comprised of compositions of concurrently running components are difficult to analyze due to the state explosions implied by nondeterministic behavior of concurrent components. Techniques have been developed to handle large number of states. For instance, a model-theoretic reasoning tool (i.e., a model checker), called the Symbolic Model Verifier (SMV), has been used to verify

models with more than $10^{20}$ states [3]. Although practical, but relatively small systems can be verified in this way, real-size systems can easily outdo the aforementioned number of states. A scenario-based approach like UCM reduces the number of possibilities to consider. The functionality provided by a system comprised of components is typically not equal to the sum of the functionalities provided by its parts. In general, components will be underutilized. UCM scenarios are employed to show what functionality of components is actually used by the system as a whole. This reduces the complexity involved in analyzing component systems. Furthermore, self-contained subsystems can be analyzed in isolation. An abstraction (a simplified model) of the subsystem can then be substituted in the overall system. Despite the reductions in complexity, the analysis of component compositions remains a computational expensive process.

It is interesting to explore the extent of what can be analyzed by the combination of UCM and BCOOPL interfaces. To put it differently, does the combination specify the behavior of a component composition completely? This turns out not to be the case. For one thing, pre- and post-conditions cannot be expressed formally in BCOOPL interfaces. (However, Pre- and post conditions can be specified informally by means of responsibility points in UCMs.) Although temporal orderings implied by regular expressions and client specifications can be seen as pre-conditions, there is no way to state pre- and post-conditions in terms of state variables. Thus, BCOOPL offers a static construct for specifying enabling conditions, rather than taking runtime, dynamic behavior into account. The advantage of this approach is that we do have a specifications of temporal orderings, which are hard to deduce from pre- and post-conditions alone.

There are no principal reasons not to support state variables in BCOOPL interfaces. It should be noted that the inclusion of state variables is not a violation of the principle that an interface should not commit to implementation details. For instance, size is an intrinsic property of a stack. Pre- and post-conditions for stack operations can be specified in terms of size without committing to an implementation yet. This notion is central in the component specification method Catalysis [9]. Catalysis also includes the concept of refinement. Interface (type) specifications can be gradually refined into an implementation as long as it is assured that an implementation is in conformance with its type.

Nevertheless, potential problems can be pinpointed by the combination of UCMs and BCOOPL interfaces. In some cases, however, more problems will be spotted than can actually happen at runtime due to the fact that enabling conditions cannot be specified precisely enough at present. The use of state variables in interface specifications will be explored in the near future.

## 7 Concluding Remarks

We have discussed an approach for modeling component compositions and analyzing them statically. UCMs and BCOOPL interfaces strike the balance between understandability (by humans) and preciseness (for reasoning about component compositions). Interacting UCM scenarios give a high level, easy to understand overview of the behavioral aspects of a system. UCM scenarios in combination with BCOOPL interface specifications allow to reason about it formally. In addition, the use of UCM scenarios helps in reducing the complexity of the analysis process.

Future work include modeling gradual refinements (from type specification to implementation) and investigating quality attributes (e.g., performance) other than behavior that are amenable for analysis purposes using the approach discussed in this paper. In addition, we want to look at tool support for analyzing real-size systems.

## References


1. 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.
2. R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
3. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Fifth Conference on Logic in Computer Science*, 1990.
4. R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science 16*, pages 89–102. Springer-Verlag, Berlin, Germany, 1974.
5. Hans de Bruin. *DIGIS: a Model Based Graphical User Interface Design Environment for Non-Programmers*. PhD thesis, Erasmus University Rotterdam, November 10, 1995.
6. 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.
7. Hans de Bruin. BCOOPL: A language for controlling component interactions. In H.R. Arbnia, editor, *Proceedings of the International Conference of Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 2, pages 801–807, Las vegas, Nevada, USA, June 26–29, 2000. CSREA, CSREA Press.
8. Hans de Bruin. BCOOPL: Basic Concurrent Object-Oriented Programming Language. *Software Practice & Experience*, 30(8):849–894, July 2000.
9. 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.
10. 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.
11. 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.
12. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
13. Ole Lehrman Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993.
14. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
15. Jan van den Bos and Chris Laffra. Procol: a concurrent object language with protocols, delegation and persistence. *Acta Informatica*, 28:511–538, September 1991.