Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

A channel-based coordination model for components

F. Arbab, F.S. de Boer, M.M. Bonsangue,
J.V. Guillen Scholten

REPORT SEN-R0127 DECEMBER 31, 2001

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2001, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

A Channel-based Coordination Model for Components

Farhad Arbab[1]
Frank S. de Boer[1]
Marcello M. Bonsangue[2]
Juan V. Guillen Scholten[1]

[1]*CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mails: farhad@cwi.nl, frb@cwi.nl, and juan@cwi.nl

[2]*LIACS, Leiden University*
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

ABSTRACT

In this paper we present a coordination model for component-based software systems based on the notion of mobile channels, and describe its implementation in the Java language. Channels allow anonymous, and point-to-point communication among components, while mobility allows dynamic reconfiguration of channel connections in a system. This model supports dynamic distributed systems where components can be mobile. It provides an efficient way of interaction among components. Furthermore, our model provides a clear separation between the computational part and the coordination part of a system, allowing the development and description of the coordination structure of a system to be done in a transparent way. Our description of the Java implementation of this coordination model demonstrates that it is self-contained enough for developing component-based systems. However, if desired, our model can be used as a basis to extend other models that focus on other aspects of components that are less related to composition and coordination concerns.

2000 ACM Computing Classification: C.2.4, D.1.3, D.1.5, D.3.2, D.3.3, F.1.2
Keywords and Phrases: Channels, components, component-based software, coordination model, dynamic distributed systems, Java implementation of coordination model, mobility, mobile agents
Note: Work carried out under the project SEN3.2 "Component-based models and software architectures"

Contents

1 Introduction **3**

2 Components and their Composition **3**
2.1 Components and Object-Oriented Technology 4
2.2 Components and their Interfaces 4
2.3 Coordination Among Components 4

3 Mobile Channels **5**
3.1 MoCha 7

4 Implementation in Java **8**
4.1 Components in Java 9
4.2 Implementation Overview 9
4.3 The Interface of a Component 10
4.4 The Coordination Operations 11
4.5 A Small Example 13

5 Related Work and Conclusion **13**

1 Introduction

In the last decade, structured software development has emerged as the means to control the complexity of systems. However, concepts like modularity and encapsulation alone have shown to be insufficient to support easy development of large software systems. Ideally, large software systems should be built through a planned integration of perhaps pre-existing components. This means not only that components must be pluggable, but also that there must be a suitable composition mechanism enabling their integration.

Component-based software describes a system in terms of *components* and their *connections*. Components are black boxes, whose internal implementation is hidden from the outside world. Instead, the composition of components is defined in terms of their (logical) interfaces which describe their externally observable behavior. By hiding all system computation in the components, a system can be described in terms of the observable behavior of its components and their interactions. As such, component-based software provides a high-level abstract description of a system that allows a clear separation of concerns for the coordination and the computational aspects of a system. The importance of such high level logical descriptions of systems is growing in the Software Engineering community. Traditionally, the description of a system is limited to the physical layout of its software. For example, this is the case in the standard OO modeling language *UML* [6]. However, extensions of *UML* are now emerging to support logical entities as components, their interfaces, and connectors, which allow a logical decomposition and description of the system. An example of such an extension is UML-RT[17], which is an integration of the architectural description language *ROOM*[18] into UML.

In this paper we present and advocate a coordination model for component-based software that is based on mobile channels, and describe its implementation in the Java language. A mobile channel is a coordination primitive that allows anonymous point-to-point communication between two components, and enables dynamic reconfiguration of channel connections in a system. It also supports dynamic distributed systems where components can be mobile.

From a software development point of view, mobile channels provide a highly expressive data-flow architecture for the construction of complex coordination schemes, independent of the computation parts of components. This enhances the re-usability of systems: components developed for one system can easily be reused in other systems with different (or the same) coordination schemes. Also, a system becomes easier to update: we can replace a component with another version without having to change any other component or the coordination scheme in the system. Moreover, a coordination scheme that is independent of the computation parts of components can also be updated without the necessity to change the components in the system.

The Java implementation presented in this paper provides a general framework that integrates a highly expressive data-flow architecture for the construction of coordination schemes with the object-oriented architecture for the description of the internal data-processing aspects of components.

The rest of this paper is organized as follows. In section 2 we discuss components and several coordination mechanisms for their composition, and present our rationale for a model based on channels. In section 3, we introduce and show the advantages of the notion of mobility for channels. In section 4 we describe an implementation of our model in the Java language [12]. We conclude in section 5, where we briefly discuss some related work.

2 Components and their Composition

In this section we briefly discuss the general notion of a component and coordination mechanisms for composing components.

2.1 Components and Object-Oriented Technology

Components adhere to the fundamental principles that are the underpinnings of object-oriented technology:

- systemwide unique identity;
- bundling of data and functions manipulating those data;
- encapsulation for hiding detailed information that is irrelevant to its environment and other components.

However, components extend these principles by adhering to a stronger notion of encapsulation. Whereas the interface of an object involves only a one-way flow of dependencies from the object providing a service to its clients, an interface of a component involves a two-way reciprocal interaction between the component and its environment. This stronger notion of encapsulation accommodates a more general notion of re-usability because mutual dependencies are now more explicit through component interfaces. Furthermore, it allows components to be independently developed, without any knowledge of each other.

Components are self contained "binary" packages. All services provided by a component, as described in its interfaces, must be produced in the component itself. If objects are used to implement a component, implementation inheritance should not cross the component boundaries. No other restrictions are imposed on a component implementation.

2.2 Components and their Interfaces

We define a *component* as an entity that can be used (composed) by means of its *interface* only. Such an interface describes the *input*, *output*, and the *observable behavior* of the component. For example, the interface of a component may tell us that, given a specific input, a window with a message will appear on the screen. However, how this is implemented in the component is hidden from the outside world, i.e., a component is viewed as a *black box*. An interface of a component, therefore, provides an abstraction of the component which encapsulates its internal implementation details that are not relevant for its use.

In our channel-based coordination model a component interface consists of a set of mobile channels through which the component sends and receives values. This set can be static, dynamic, or a combination of both. The observable behavior can be expressed by using, for example, predicates, comments, or some graphical notation.

2.3 Coordination Among Components

Besides components, a system also needs *connections* among them. There are several coordination mechanisms for composing components. Because components must be pluggable, it is important that these mechanisms do not require a component to know anything about the structure of the system they are plugged into. We discuss four important types of coordination mechanisms: *messaging*, *events*, *shared data spaces*, and *channels* [1].

Messaging. With this type of connection, components send messages to each other. These messages need not be explicitly targeted; a component can send a message meant for any component having some kind of specific service (publish-and-subscribe model), instead of sending it to a particular component (point-to-point model). However, messaging is not really suitable for component-based software because it requires the components to know something about the structure of the system: even if they do not directly know their service providers, they must know the services provided in the system. An implementation example of this type of connection is the Java Message Queue (*JMQ*) [20], a package based on the Java Message Service (*JMS*) [21] open standard.

Events. With the event mechanism a component, called the *producer* or *event source*, can create and fire events, the events are then received by other components, called *consumers* or *event listeners*, that listen to this particular kind of events. *JavaBeans* [13], which are seen as the components in Java, use the events mechanism.

Shared data spaces. In a shared data space, all components read and write values, usually tuples like in *Linda* [7], from and to a shared space. The tuples contain data, together with some conditions. Any component satisfying these conditions can read a tuple; tuples are not explicitly targeted. The *JavaSpaces* technology [8], a powerful Jini service from Sun, is an example of a shared data space that is being used for components.

Channels. A channel, see figure 1, is a one-to-one connection that offers two ends, its *source* and its *sink*, to components. A component can write by inserting values to the *source*-end, and read by removing values from the *sink*-end of a channel; the data-flow is locally *one way*: from a component into a channel or from a a channel into a component. The communication is *anonymous*: the components do not know each other, just the channel-ends they have access to. Channels can be synchronous or asynchronous, mobile, with conditions, etc. Examples of systems based on channels include: *Communicating Threads for Java* [10], *CSP for Java* [22], both based on the *CSP* model [11], and *Pict* [16], a concurrent programming language based on the π-calculus. However, these systems either do not support distributed environments, or their channels are not mobile. MoCha (see section 3.1) implements distributed mobile channels.

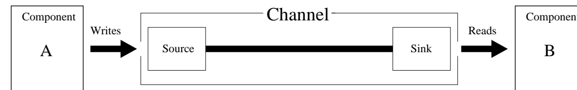

Figure 1: A Channel.

We base our coordination model on (mobile) channels. The last three coordination mechanisms allow true separation of coordination and computation concerns in a system. Like shared data spaces, channels support anonymous communication. However, channels have other advantages over events and shared data spaces for the systems that we are interested in.

First, like messaging and in contrast with shared data spaces, point-to-point channels can be implemented more *efficiently* in distributed systems. Second, like messaging and events, point-to-point channels support a more *private* means of communication that prevents third parties from accidentally or intentionally interfering with the private communication between two components. In contrast, shared data spaces are in principle "public forums" that allow any component to read any data they contain. Accommodating private communications within the public forum of a shared data space, places an extra burden on many applications that require it. Third involves *architectural expressiveness*. Like messaging, using channels to express the communication carried out within a system is architecturally much more expressive than using shared data spaces. With a shared data space, it is difficult to see which components exchange data with each other, and thus depend on or are related to each other, because in principle, any component connected to the data space can exchange data with any or all other components in the system. The point-to point channels that inter-connect the components of a system, express significant facts about the inter-dependencies among components. Finally, in contrast to events, channels allow several different types of connections among components, e.g., synchronous, FIFO, etc.

3 Mobile Channels

In our coordination model, components interact with each other through mobile channels. A channel is called *mobile* when the identities of its channel-ends can be passed on through channels to other

components in the system. Furthermore, in distributed systems the ends of a mobile channel can physically move from one location to another, where location is a *logical address space* where components execute. Because the communication via channels is *anonymous*, when a channel-end moves, the component at its other end is not affected.

Mobility allows dynamic reconfiguration of channel connections among the components in a system, a property that is very useful and even crucial in systems where the components themselves are mobile.

A component is called mobile when, in a distributed system, it can move from one location (where its code is executing) to another. Laptops, mobile phones, and mobile Internet agents are examples of mobile components. The structure of a system with mobile components changes dynamically during its lifetime. Mobile channels give the crucial advantage of moving a channel-end together with its component, instead of deleting a channel and creating a new one.

In our model, a component must perform a successful `Connect` operation on a specific channel-end before it can use it, and it must perform a `Disconnect` operation to release it (see section 4.4). At every moment in time, at most one component can be connected to a particular channel-end. Therefore, although many components may know the identity of a specific channel-end, the communication via mobile channels is still one-to-one. This ensures the soundness and completeness properties that are the prerequisites for compositionality [4]. Our one-to-one channels can still be composed into many-to-many connectors, while preserving these prerequisites for compositionality [2, 3].

As a concrete example of the utility of mobile channels suppose we want to use agents to search for specific information, e.g. coffee prices, on the Internet. Agents consult different XML[23] information sources, like databases and Internet pages. Each information source has a channel where requests can be issued, and an agent knows the identity of the source end of this channel plus the location of the information source. The agents may have a list made at their creation, or this information may be passed to them through channels. In our example, we use a mobile agent that moves among the different locations of the information sources. An alternative that we will consider later is to create an agent at every location.

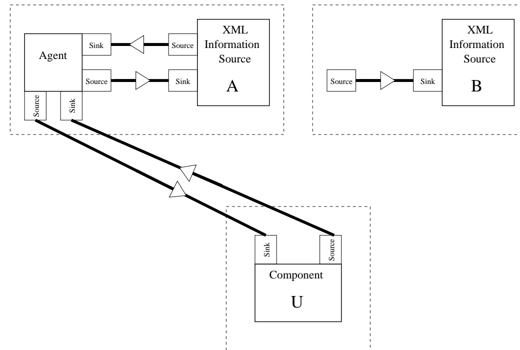

Figure 2: An Example: a Hopping Agent.

A component U has two channel connections for interaction with a mobile agent, one to send instructions and the other to receive results. At some point in time, U asks the agent to search for MoCha-beans prices. Figure 2 shows the situation after the agent moves to the information source A which is in a different Internet location, as expressed by the dashed lines in the figure. Right after the move, the agent creates a channel meant for reading information from the information source, and sends a request to A together with the identity of the source channel-end of the created channel.

At some point in time the agent finishes searching the information source A and writes all relevant information it finds for the component U into the proper source channel-end. Regardless of whether or not this information has already been read by U, the agent moves to the location of the next

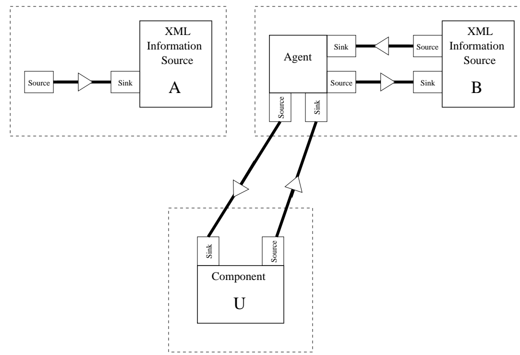

Figure 3: Moving to Another Location.

information source (see figure 3). Together with the agent, the two ends of the channels connecting it to U also move with it to this new location. However, the component U is not affected by this. It can still write to and read from its channel-ends, even during the move; all data in a mobile channel are preserved while its ends move. For the agent the advantages of moving the channel-ends along with it is that it avoids all kinds of problems that arise if it were to delete the channels and create new ones after the move, e.g., checking if the channels are empty, notifying U that it cannot use them anymore, perhaps some locking issues to accomplish the latter, etc.

In our alternative version, we have a different non-mobile agent at each location, instead of one mobile agent, and there are only two channels for interaction with the component U. The channel-ends meant for the agents then move from one agent to the other. From the point of view of the component U there is no difference between the two alternatives in our example.

In our example, the two channel-ends used by U do not move, but it is possible to have mobility at both ends of a channel, if desired, and extend the example by passing these channel-ends on to other components in the system.

3.1 MoCha

MoCha, is an implementation model for *mo*bile *cha*nnels in distributed environments that supports mobility as described above. More details can be found in [9] and in our future work.

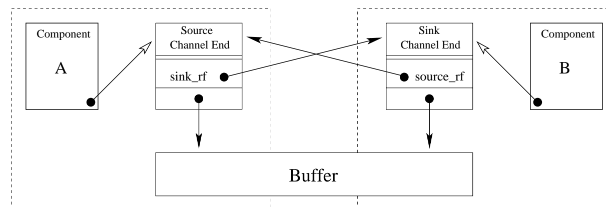

Figure 4: A mobile Channel in MoCha.

In figure 4, we show how a channel is realized in MoCha. For components, a channel consists of two data-structures, the *source* and the *sink* channel-ends, which they (separately) refer to through *interface references*. An *interface reference* is a reference from a component to a channel-end, restricting the access of the component to only the pre-defined operations on the channel. These operations include: *create*, *read*, *write*, *move*, and *delete*. The ends of a channel must internally know each

other to keep the identity of the channel and control communication. For this purpose, the ends have references to each other: the *sink_rf*- and *source_rf*-fields in the figure. If the type of a channel is *asynchronous* then its channel-ends also have references to a buffer. The implementation of this buffer depends on the asynchronous channel type.

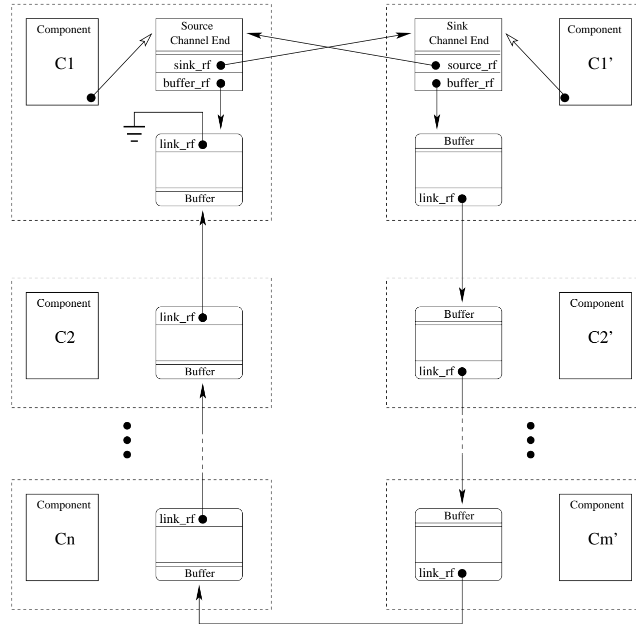

Figure 5: A FIFO mobile Channel in MoCha.

Figure 5 shows the implementation of an asynchronous FIFO mobile channel in MoCha. The buffer is implemented by a chain of unbounded FIFO buffers, each pointing to its next buffer through its *link_rf* reference. A local buffer is created by the *source* channel-end each time a component performs the operation *write* and no local buffer yet exists. This buffer is then added to the existing chain of buffers. Buffers get destroyed when they get empty due to a *read* operation on the *sink* channel-end. Both channel-ends have references, *buffer_rf*, to a buffer. If this reference is local and the channel-end moves to another location, then the local buffer it refers to does not move with it, instead, the *buffer_rf* reference is changed from local to non-local. With this implementation each *write* operation is always local. A *read* operation is either local or non-local depending on the amount of elements needed. *move* operations do not involve data-transfer of elements at all [9].

MoCha has been implemented in Java using the Remote Method Invocation, *RMI*, package[14]. We use MoCha for the Java implementation of our coordination model (see section 4.2).

4 Implementation in Java

The coordination model we present in this paper can be implemented in any modern programming language that supports distributed environments, like Java[12], or C++[19]. In this section we describe an implementation of our model in the Java language.

The implementation consists of a framework that provides (a) a *precompiler* tool for writing components, (b) mobile channels, and (c) operations on these channels. All the component source files have the extension `.cmp`, and the *precompiler* transforms them into normal Java files. We do not define a new language: the `.cmp` files contain Java code and the *precompiler* just verifies certain restrictions we

need to impose to have components in Java. We explain these restrictions gradually while describing the implementation.

4.1 Components in Java

Usually, JavaBeans [13] are used to implement components in Java. However, they do not comply with our definition of components (see section 2.2) for two reasons. First, a JavaBean consists of just *one* class, and this puts a serious restriction on the internal implementation of components. Second, JavaBeans communicate with each other through *events*, while we want to use channels (see section 2.3).

Instead of using JavaBeans to implement components, we use the `package` feature of Java. However, a `package` is too broad and does not provide the hard boundaries we need for components. Therefore, we must impose some restrictions that must be verified by the precompiler. These restrictions are (1) a component must have *at least* one `class` that represents the component's *interface*, through which all coordination and access to channels takes place; (2) these *interface* classes are the only `public` classes in a `package`; and (3) only *interface* classes can have methods and variables that are `public`. For simplicity, in the sequel we assume that the interface of a component consists of just one `class`.

Implementing a component as a `package` plus the restrictions explained in the last paragraph has two major advantages. One advantage is that access to a component is possible only through its interface. This combined with the fact that internal references cannot be sent through a channel (see section 4.4) makes it possible to protect the internal implementation of a component.

The second advantage is that restrictions (1),(2) and (3) are so minimal that they do not impose any real restrictions concerning the internal implementation of a component. A component may have one or more objects, one or more active entities, its implementation may be distributed, or it may be a channel-based component system itself, etc.

4.2 Implementation Overview

Figure 6 shows a general overview of the structure of our implementation. A component is a `package` that contains (a) a `class` which describes its *interface*, and (b) internal entities (*objects*) created by the component's programmer(s), which may also be active (*threads*). This `package` is created by the *precompiler* from its `.cmp` files.

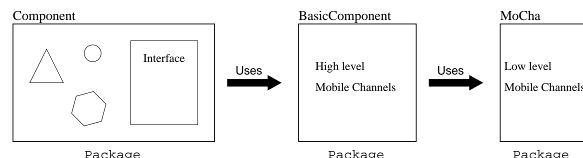

Figure 6: Implementation Overview

The component `package` uses, with the `import` feature of Java, our *BasicComponent* `package`. The BasicComponent `package` is an extra layer, between the component and the low level mobile channels of MoCha, needed in order to avoid dangling local references to channel-ends that result from mobility. The BasicComponent `package` provides channel-end *variables* that only indirectly refer to MoCha channel-ends.

A component can have `Sink` and `Source` channel-end variables. However, it can perform operations on these variables only through the coordination methods of its interface (see section 4.4). To accomplish this, the package BasicComponent provides methods that are `protected` and which

only the coordination methods of the interface can use. The `package` also provides a `Location` for the components. This data-structure is used to identify both the location of the component in the network, the IP-address, and the specific virtual machine where it is running.

Observe that instead of MoCha, we can use any other implementation of mobile channels, if desired.

4.3 The Interface of a Component

The interface of a component has two parts, a `package private` part accessible only to the internal entitie(s) of the component, and a `public` part accessible to all the entities in the system. A component interface is a normal Java `class` and should not be confused with the `Interface` feature of this language. Figure 7 shows the skeleton of a `.cmp` file for the interface. There is some syntactic sugar in this file that the *precompiler* translates into legitimate Java code:

- `Component` *CompName*, must appear as the header of each `.cmp` file of a component. This line is translated into `package` *CompName*; `import BasicComponent.*;`.
- `ComponentInterface` *IntName* is translated into `public class` *IntName* `extends BasicInterface`.

The interface class inherits from `BasicInterface`, a class that contains basic methods for both the `public` and the `package private` parts of the interface (see figure 8). The precompiler adds this class to the component's package, which precludes the possibility of change by the programmers.

```
Component CompName;
/* add import list here */

ComponentInterface [IntName] // default is CompNameInterface
{
   public IntName( )
   {
      super(loc); // call super class constructor
      /* Create and initialize here all the
         entities of the component */
   }
   public void finalize()
      {
      /* Method is optional, perform cleanup
         actions before the object is garbage collected */
   }
   /* Put your optional public variables of type ChannelEnd here */
}
```

Figure 7: The `.cmp` Skeleton File for the Interface of a Component

The `public` part of the interface consists of four parts (see figure 7 and 8): one or more constructors, a `getLocation` method, a `finalize` method, and variables of type `ChannelEnd`. The precompiler checks if these items are the only `public` ones in the interface.

The interface can have one or more `public` constructors. The class has a `super class` (see figure 8) that needs a `Location` as a parameter for its constructor. This way we *enforce* that each constructor of the interface class must provide a `Location`, which is either created in the constructor or passed through as a parameter. In the constructor(s) all internal entities of the component must be *created* and *initialized*. Thus, in order to create a component, it is enough to import the component's package and make an instance of it's interface class.

Optionally, a `finalize` method can be present to perform cleanup operations before a component instance is garbage collected. An interface can also have `public` channel-end variables if desired, or data-structures built using them (for example, arrays of channel-end variables).

The `package private` part of the interface includes the coordination methods provided by the class `BasicInterface` (see figure 8) and all the other methods and variables in the interface that are not `public`. We explain the coordination methods in section 4.4.

```
package CompName;
import MoCha.*;
import BasicComponent.*;

class BasicInterface
{
   BasicInterface(Location loc)
   public Location getLocation()
   Object[] CreateChannel(ChannelType type)
   boleaan Connect(ChannelEnd ce, int timeout) throws Exception
   boleaan Disconnect(ChannelEnd ce) throws Exception
   boleaan Write(Source ce, Object var, int timeout) throws Exception
   Object Read(Sink ce, int timeout) throws Exception
   Object Take(Sink ce, int timeout) throws Exception
   boleaan Wait(String conds, int timeout) throws Exception
}
```

Figure 8: The `BasicInterface` Class

For simplicity, we assumed that the interface of a component consists of just one `class`. However, we do allow components to have more than one `ComponentInterface class`. Therefore, a component can provide several interfaces to its users with different views and/or functionality.

4.4 The Coordination Operations

The interface of a component provides coordination methods for the active internal objects (i.e. *threads*) in a instance of that component for operations on channels. These methods are listed in figure 8. The threads cannot perform any operation directly on the channel-ends, because the channel-ends do not provide any methods for them, not even a constructor. Therefore, the only way to perform an operation on a channel is to use the coordination methods in the component interface. The coordination operations are divided in three groups: the *topological* operations, the *input/output* operations, and the *inquiry* operations.

These operations are basic operations and more complex operations can be created by composition of these basic ones. It is, also, the responsibility of the component to ensure proper synchronization for its internal threads, if they refer to the same channel-ends. Our basic coordination primitives can be wrapped in component defined methods to enforce such internal protocols.

For every method containing a `timeout` parameter, there is also a version without the time-out (not listed in the figure). When no time-out is given the thread performing the method suspends indefinitely until the operation succeeds or the method throws an `exception`. For uniformity of explanation, we assume that the time-out parameter can also have the special value of *infinity*. This way we need not define two versions of each operation.

Topological Operations

`CreateChannel` creates a new channel of the specified `type`. The value of this parameter can be `synchronous` or asynchronous channels like `FIFO`, `bag`, `set`, etc. The channel-ends, source and sink,

are created at the same location as the component and their references are returned as an array of type `Object`: `Object[0] = Source` and `Object[1] = Sink`. We return this array, instead of some `Channel` data-structure containing the channel-end references, in order to avoid introducing new unnecessary data types. If desired, this method can be wrapped to return such a `Channel` class but this is not necessary.

Connect connects the specified channel-end `ce` to the component instance that contains the thread that performs this operation. If the channel-end is currently connected to another component instance, then the active entity suspends and waits in a queue until the channel-end is connected to this component instance or, its time-out expires. The method returns `true` to indicate success, or `false` to indicate that it timed-out. When a connect operation is succesfull and other threads in the same component instance are waiting to connect to the same channel-end, they all succeed. If a thread tries to connect to a channel-end already connected to the component instance, it also immediately succeeds.

When the `Connect` operation succeeds the channel-end *physically* moves to the location of the component instance in the network. All channel-ends connected to the component move along with it while remaining connected.

Disconnect disconnects the specified channel-end `ce` from the component instance that contains the thread performing this operation. This method *always succeeds* on a valid channel-end. It returns `true` if the channel-end was actually connected to the component instance and `false` otherwise. If `ce` is invalid, e.g. `null`, then the method throws an exception.

Input/Output Operations

Write suspends the thread that performs this operation until either the `Object var` is written into the channel-end `ce`, or its specified time-out expires. Only `Serializable` objects, channel-end identities, and component locations can be written into a channel. The `Serializable` objects are copied before inserted into the channel, therefore no references to the internal objects of a component can be sent through channels. The method returns the value `true` if the operation succeeds, and the value `false` if its time-out expires. The method throws an exception if either `ce` is not valid, the component instance is not connected to the channel-end, the `Object var` is not `Serializable`, or it contains a reference to a non-`Serializable` object.

Read suspends the thread that performs this operation until a value is read from the sink channel-end `ce`, or its specified time-out expires. In the first case the method returns a `Serializable Object`, a channel-end identity, or a `Location`. In the second case the method returns the value `null`. The value is not removed from the channel. The method throws an exception if either `ce` is not valid, or the component instance is not connected to the channel-end.

Take is the destructive variant of the `Read` operation. It behaves the same as a `Read` except that the read value is also removed from the channel.

Inquiry Operations

Wait is the inquiry operation. It suspends the thread that performs it until either the conditions specified in `conds` become true or its time-out expires. In the first case the method returns `true`, and otherwise it returns `false`. The channel-ends involved in `conds` need not be connected to the component instance in order to perform this operation, but an invalid channel-end reference throws an exception. The argument `conds` is a boolean combination of primitive channel conditions such as `connected(ce)`, `disconnected(ce)`, `empty(ce)`, `full(ce)`, etc.

4.5 A Small Example

We use a simple implementation of the mobile agent component of the example in section 3, to show the utility of the coordination operations provided by our model. Figure 9 shows the Java pseudo-code for this agent. `AgentInterface` is the agent's interface and consists of the basic interface plus a method `Move`. This method moves the agent to the specified location including the channel-ends it is connected to, (`readChannelEnd`, `writeChannelEnd`, and `channel[1]`). The `readChannelEnd` and `writeChannelEnd` channel-ends are, respectively, the sink and the source of the channels for interaction with the component U. The agent has a list containing the locations of the information sources together with their source channel-end references where it can issue its requests.

```
void agentImplementation()
{
   AgentInterface.Connect(readChannelEnd);
   AgentInterface.Connect(writeChannelEnd);
   Object[] channel = CreateChannel(FIFOchannel);
   AgentInterface.Connect(channel[1]);
   For each entry in informationSourceList do
       AgentInterface.Move(List[InformationSource].location, channel[1]);
       AgentInterface.Connect(List[InformationSource].sourceEnd);
       AgentInterface.Write(List[InformationSource].sourceEnd,
           REQUEST + channel[0]);
       AgentInterface.Disconnect(List[InformationSource].sourceEnd);
       information.add(AgentInterface.Read(channel[1]));
       information.transformation();
       AgentInterface.Write(writeChannelEnd, information);
       String cond ="notEmpty(" + readChannelEnd + ")";
       information.clear();
       if ( AgentInterface.Wait(cond, 0) ) then
           read an instruction from this channelEnd and process it.
       fi
   od
   AgentInterface.Disconnect(readChannelEnd);
   AgentInterface.Disconnect(writeChannelEnd);
}
```

Figure 9: Simple Implementation of The Mobile Agent

5 Related Work and Conclusion

In this paper we presented a coordination model for component-based software based on mobile channels. The idea of using (mobile) channels for components has its foundations in the earlier work of some of the authors of this paper, e.g., in [4] and [5].

Our model provides a clear separation of concerns between the coordination and the computational aspects of a system. We force a component to have an *interface* for its interaction with the outside world, but we do not make any assumptions about its internal implementation. We define the interface of a component as a dynamic set of channel-ends. Channels provide an *anonymous* means of communication, where the communicating components need not know each other, or the structure of the system. The architectural expressiveness of channels allows our model to easily describe a system in terms of the interfaces of its components and its channel connections, abstracting away their computational parts. Coordination is expressed merely as operations performed on such channels. The mobility of channels allows dynamic reconfiguration of channel connections within a system.

Certain aspects of and concerns in *ROOM*[18] and *Darwin*[15], two architectural description languages (ADL), are related to our work. In ROOM components are described by declaring the internal

structure, the external interface, and the behavior of all its instances (if it is a composite component). The interface of a component is a set of *ports*. A port is the place where components offer or require certain services. The communication through these ports is bidirectional and in the form of asynchronous messaging. The components of Darwin are similar to the ones of ROOM, but instead of ports, Darwin components have *portals*. These portals specify the input and output of a component in terms of services, as in ROOM. However, the *binding* of portals is not specified, leaving it open for all kinds of possible bindings. Another difference between Darwin and Room, is that Darwin can describe dynamically changing systems, while ROOM can describe only static ones. This makes Darwin more suitable than ROOM for component-based systems that use our coordination model. Of course, to model mobile channels or the dynamic set of interfaces of a component, for instance, some extensions to Darwin would be necessary.

Other models for component-based software can benefit from the coordination model presented in this paper, because ours is a basic model that focuses only on the coordination of components. Our model can extend other models that are concerned with other aspects of components, for example, their internal implementation, their evolution, etc.

Our coordination model opens the possibility to apply more powerful coordination paradigms that are based on the notion of mobile channels to component-based software. One such paradigm, is $P\epsilon\omega$[2]. $P\epsilon\omega$ supports composition of channels into complex connectors whose semantics are independent of the components they connect to. We are currently extending our coordination model for component based systems in order to support all the features of $P\epsilon\omega$.

References

[1] G. Andrews, *Paradigms for process interaction in distributed programs*, ACM Computing Surveys, 23(1):49-90, 1991.

[2] F. Arbab,*Coordination of Mobile Components*, Electronic Notes in Theoretical Computer Science Vol 54, Elsevier Science B.V., 2001.

[3] F. Arbab, *A channel-Based Coordination Model for Component Composition*, Tech. Report, Centrum voor Wiskunde en Informatica, Amsterdam, 2001. Available on-line http://www.cwi.nl/ farhad/RewTechReport.ps.

[4] F. Arbab, F. S. de Boer, and M. M. Bonsangue. *A Logical Interface Description Language for Components.* , Proceedings of Coordination 2000, Lecture Notes in Computer Science, Springer, 2000.

[5] F. Arbab, M. M. Bonsangue, and F. S. de Boer. *A Coordination Language for Mobile Components.*, Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000), pp 166-173, ACM, 2000.

[6] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* , Addison-Wesley, Reading, Mass. USA, 1999.

[7] N. Carriero, D. Gelernter. *How to Write Parallel Programs: a First Course*, MIT press, 1990.

[8] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces TM Principles, Patterns, and Practice*, Chapter 1 of book, Addison-Wesley, September 1999.

[9] J.V. Guillen Scholten, *MoCha, a Model for Distributed Mobile Channels*, Internal Report 01-07, Master's Thesis, LIACS, Leiden University, May 2001.

[10] G. Hilderink, J. Broenink, and A. Bakkers. *Communicating Threads for Java*, Draft version available at Home Page: http://www.rt.el.utwente.nl/javapp/information/CTJ/main.html, The Netherlands, 2000.

[11] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, London, UK, 1985.

[12] Home Page of Java, *http://java.sun.com.*

[13] Home Page of JavaBeans, *http://java.sun.com/products/javabeans.*

[14] Home Page of RMI documentation, *http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html.*

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. *Specifying Distributed Software Architectures.* In Proceedings of 5th European Software Engineering Conference, Spain, 1994.

[16] B. C. Pierce, D. N. Turner, *Pict: A Programming Language Based on the Pi-calculus*, Technical report, Computer Science Department, Indiana University, 1997. Home Page of Pict, *http://www.cis.upenn.edu/ bcpierce/papers/pict/Html/Pict.html.*

[17] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr, *UML + ROOM as a Standard ADL?*, Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems, 1999.

[18] B. Selic, G. Gullekson, and P.T. Ward, *Real-Time Object-Oriented Modelling*, John Wiley and Sons, Inc., 1994.

[19] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

[20] Sun, *Java Message Queue, Quickstart Guide v1.1*, Sun Microsystems Inc., Palo Alto (USA), May 2000.

[21] Sun, *Java Message Service, Specification Document version 1.0.2* , Sun Microsystems Inc., Palo Alto (USA), November 1999.

[22] P. Welch, *CSP for Java (What, Why, and How Much?)*, Slides of Seminar, University of Kent at Canterbury, 2001. Home Page of JCSP, *http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.*

[23] World Wide Web Consortium, *eXtensible Markup Language*, `http://w3c.org/XML/`.