A Component Coordination Model Based on Mobile Channels

Juan Guillen-Scholten[*][†]**, Farhad Arbab**[*] **and Frank de Boer**[*]

CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

{juan, farhad, frb}@cwi.nl

Marcello Bonsangue[*][‡]

LIACS, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands

marcello@liacs.nl

Abstract. In this paper we present a coordination model for component-based software systems based on the notion of mobile channels, define it in terms of a compositional trace-based semantics, 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 and exogenous manner. Our description of the Java implementation of this coordination model demonstrates that it is self-contained enough for developing component-based systems in object-oriented languages. 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 concerned with composition and coordination issues.

1. Introduction

In the last decades, 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 of its computation in 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 its coordination and its computational aspects. The importance of such high level logical descriptions of systems is growing in the Software Engineering community. For example, in the standard OO modeling language *UML* [8] extensions are now emerging to support logical entities as components, their interfaces, and connectors, which allow a logical decomposition and description of a system. An example of such an extension is UML-RT[26], which is an integration of the architectural description language *ROOM*[27] into UML.

In this paper we present and advocate a coordination model for component-based software that is based on mobile channels, give its description in terms of a transition system, and describe its implementation in the object-oriented language Java. 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 an 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 give a compositional trace-based semantics for our model. In section 5 we describe an implementation of our model in the Java language [18]. We conclude in section 6, where we discuss related work.

2.   Components and their Composition

In this section we briefly discuss the general notion of a component, the integration of components with object-oriented technology, and coordination mechanisms for composing components.

2.1. Components and their Interfaces

We define a *component* as a black-box 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 channel-ends through which a component sends and receives values. This set can be static or dynamic. The observable behavior can be expressed by using, for example, predicates, comments, or some graphical notation, e.g., protocol state machines as defined in *UML*. In section 4 we express the external observable behavior of a component in terms of a compositional trace-based semantics.

2.2. Integration of Components with 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. Objects that are used to implement a component should not cross the component boundaries. No other restrictions are imposed on a component implementation.

The Java implementation of our coordination model, presented in section 5, demonstrates that object-oriented languages are well-suited to implement components and their composition. This implementation ensures the stronger notion of encapsulation needed for components, allowing access to a component only through its interface (which is a set of mobile channel-ends).

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* [2].

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*) [29], a package based on the Java Message Service (*JMS*) [30] open standard. The Microsoft Message Queuing Services [15] for COM+ [22], is another example.

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* [19], which are seen as the components in Java, use the event mechanism.

Shared data spaces. In a shared data space, all components read and write values, usually tuples like in *Linda* [9], 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 [10], a powerful Jini service from Sun, is an example of a shared data space that is being used for components. *Lime* [24] (Linda in a Mobile Environment), is a Linda middleware that can also be used for components, especially if these are mobile.

Channels. A channel, see figure 1, is a one-to-one connection that offers two ends to components, either a *source*- or a *sink*-end. 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 channel into a component. The communication is *anonymous*: the components do not know each other, only 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* [16], *CSP for Java* [31], both based on the *CSP* model [17], and *Pict* [25], 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* [12, 7] and *Nomadic Pict* [32], a distributed version of *Pict*, do implement distributed mobile channels. However, the channels of *Nomadic Pict* do not have two distinct ends as defined above and are only synchronous. We explain *MoCha* in more detail in section 5.2.
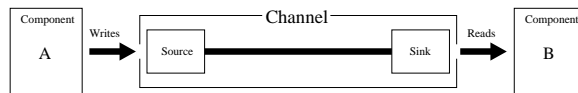

Figure 1.   A Channel.

We base our coordination model on (mobile) channels. The last three coordination mechanisms support true separation of coordination and computation concerns in a system. However, channels share many of the architectural strengths of *events* and *shared data spaces* while offering some additional benefits. Four of these benefits are: *efficiency*, *security*, *architectural expressiveness*, and transparent *exogenous coordination*.

First, although shared data spaces are useful in network architectures like blackboard systems, for most networks, like messaging, point-to-point channels can be implemented more *efficiently* in distributed systems. In shared data space models, the coordination middleware itself cannot generally know the potential receiver(s) of a message at the time that it is produced; *any* present or future entity with access to the shared data space can be the consumer of this message. In contrast, a channel-based coordination middleware always knows the connection at the opposite end of a channel, even if it changes dynamically. This additional piece of information allows the middleware to more efficiently implement the appropriate data transfer protocols. 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 is *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 more 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. Using channels, it is easy to see which components exchange data with each other, making it easier to apply tools for analysis of the dependencies and data-flow. Finally, in contrast to events, channels allow several different types of connections among components, e.g., synchronous, FIFO, etc., without the components knowing which channel types they are dealing with. This makes it possible to coordinate components from 'outside' (exogenous).

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 (*logical mobility*). 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 in a components executes (*physical mobility*). Because the communication via channels is *anonymous*, when a channel-end moves (physically or logically), 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. For example, mobile Internet agents can be seen as 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 distributed systems, a channel is a *resource* that must be shared among several component instances. Therefore, in our model, a component instance must successfully `connect` to a channel-end before it can use that channel-end, and `disconnect` from it when it is no longer needed. 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.

As a concrete example of the utility of mobile channels, suppose we want to use agents to search for some specific information, e.g., coffee prices, on the Internet. Agents consult different XML[33] 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 with these channel-ends available at their creation, or this information may be passed to them through channels. In our example, we use a mobile agent that moves to the information sources at various locations. An alternative that we will consider later is to create an agent at every location.
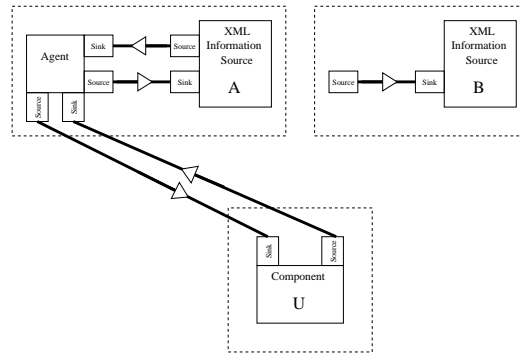

Figure 2.   An Example: a Hopping Agent.

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-bean 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 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
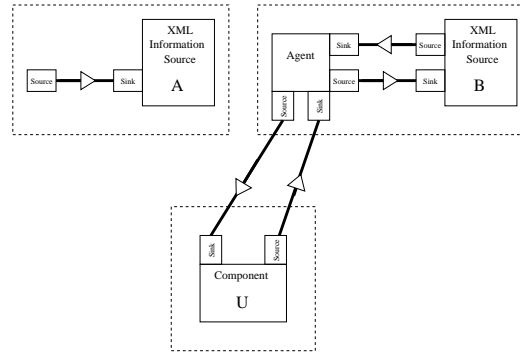
Figure 3.　Moving to Another Location.

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 one can extend the example by passing these channel-ends on to other components in the system.

4.　A Semantic Approach of our Model

In this section, we give a more precise and formal description of our coordination model, by presenting a compositional trace-based semantics of component-based systems. The semantics forms the formal basis of the notion of 'contracts' and provides a formal basis of the Java implementation in the next section.

We summarize the following from the previous sections. A component is a black-box entity that communicates through mobile channels. A channel has two ends each of which can either be a *source* or a *sink* end; a component writes values to the *source* and reads/takes values from the *sink*. The identity of channel-ends can also be communicated through channels, allowing dynamic reconfiguration of channel-end connections in a system. The data-flow is locally *one way*. Channels can be *synchronous* or *asynchronous*. Because in a distributed system a channel is a *resource* which must be shared among several component instances, a component instance must successfully *connect* to a channel-end before being able to use it; therefore, it must also *disconnect* from it when the channel-end is not needed anymore. In our model, at most one component instance can be connected to a particular channel-end at any given time, making the communication one-to-one. This ensures the soundness and completeness properties that are the prerequisites for compositionality [5]. Our one-to-one channels can still be composed into many-to-many connectors, while preserving these prerequisites for compositionality [3, 4].

Physical movement of channel-ends, see section 3, is present in our model for reasons of efficiency; to minimize the amount of non-local transfers in distributed systems. Therefore, both in the semantics and the implementation in section 5 components do not directly perform any kind of *move* operation on channel-ends. A physical channel-end *move* is indirectly performed when a component instance either successfully *connects* to the specific channel-end or moves itself to a new (physical) location, where all the connected channel-ends move with it. This means that the physical layout of the system, whether it is distributed or not, is of no concern for the semantics that rightfully abstract from it.

Below, we first describe the observable behavior of the interface of a component, that is, its external observable behavior, in terms of a transition system that abstracts away its internal behavior. Next, we introduce a global transition system which describes the behavior of a component-based system in terms of the interactions of its components and show how this behavior can be obtained in a compositional manner. Finally, as an alternative to our global transition system, we make some comments of how to model mobile channels in the π-calculus.

4.1. Component Transition System

Definition 4.1. Given a set $Astate$ of abstract states ranged over by a and (mutually disjoint) sets $Source$ and $Sink$ of all source and sink channel-ends, we specify a component by a transition system $Comp = \langle Conf, \longrightarrow, c_o \rangle$, where $Conf = AState \times \mathcal{P}(Source \cup Sink)$ is the set of configurations with its typical element c. The configuration of a component instance thus consists of a pair $\langle a, K \rangle$, where K is the set of channel-ends known in this particular configuration. The initial configuration c_0 is defined as $\langle a_0, \emptyset \rangle$, where a_0 denotes the initial abstract state. We define the transition relation as $\longrightarrow \subseteq Conf \times Act \times Conf$; as usual, we use $c \xrightarrow{act} c'$ to indicate that $(c, act, c') \in \longrightarrow$.
The set of actions Act consists of the following operations:

- $e \downarrow$ connect the executing component instance to the channel-end e.

- $e \uparrow$ disconnect the executing component instance from the channel-end e.

- $s!v$ write the value v to the source channel-end s.

- $t?v$ take the value v from the sink channel-end t.

- $t¿v$ read the value v from the sink channel-end t (read is the non-destructive version of *take*).

- $\nu\langle s, t \rangle$ create a new channel with source- and sink-ends s and t.

- $\nu\langle Comp, K \rangle$ create a new component instance with the initial set of known channel-ends K.

- τ is the invisible operation we use to denote all other component operations that are not related to channels.

Here v ranges over the set of values which includes $Source \cup Sink$. Furthermore, we have $s \in Source$, $t \in Sink$, and $e \in Source \cup Sink$.

4.2. Local Conditions

We assume that the transition relation of component satisfies the following conditions:

1. If $\langle a, K \rangle \xrightarrow{e\downarrow} \langle a', K' \rangle$ then $e \in K$ and $K' = K$.
   A component instance can connect only to a channel-end it knows, and this operation does not affect its set of known channel-ends.

2. If $\langle a, K \rangle \xrightarrow{e\uparrow} \langle a', K' \rangle$ then $e \in K$ and $K' = K$.
   The same is true for *disconnect*.

3. If $\langle a, K \rangle \xrightarrow{s!v} \langle a', K' \rangle$ then $s \in K$ and $K' = K$.
   A component instance can write only to a channel-end it knows, and its set of known channel-ends is not affected.

4. If $\langle a, K \rangle \xrightarrow{t?v} \langle a', K' \rangle$ and $v \in Source \cup Sink$ then $t \in K$ and $K' = K \cup \{v\}$.
   A component instance can take only from a channel-end it knows. If the value obtained is a channel-end, it becomes known to the component instance.

5. If $\langle a, K \rangle \xrightarrow{t?v} \langle a', K' \rangle$ and $v \notin Source \cup Sink$ then $t \in K$ and $K' = K$.
   A component instance can take only from a channel-end it knows. If the value obtained is not a channel-end, its set of known channel-ends is not affected.

6. All conditions for *take* also apply to the operation *read*.

7. If $\langle a, K \rangle \xrightarrow{\nu\langle s,t\rangle} \langle a', K' \rangle$ then $s \notin K$ and $t \notin K$ and $K' = K \cup \{s, t\}$.
   When a new channel is created, the two new channel-ends must be added to the set of known channel-ends of the component instance.

4.3. Global Transition System

We consider a component based system $\pi = \{Comp_1, ..., Comp_n\}$, where $Comp_i = \langle Conf_i, \longrightarrow_i, c_0^i \rangle$, for $i = 1, \ldots, n$. To identify component instances we use the infinite set CId of component id's, with its typical element id. A system configuration is a tuple $\langle \sigma, \gamma, Chan \rangle$, where σ and γ are two partial functions defined as:

$$\sigma \colon CId \rightharpoonup \cup_i Conf_i \text{ and } \gamma \colon (Source \cup Sink) \rightharpoonup CId,$$

and

$$Chan \subseteq Source \times Sink.$$

A function σ maps every existing (i.e., element of its domain) component instance of $Comp_i$ to its current configuration $c \in Conf_i$. On the other hand, a function $\gamma \colon Source \cup Sink \rightharpoonup CId$ maps every channel-end to the id of the component instance it is connected to. A channel-end e is disconnected if $\gamma(e)$ is undefined. The set $Chan \subseteq Source \times Sink$ indicates which channel-end pairs constitute a channel.

We now proceed by presenting a labelled transition system which describes the observable interaction of components and channels at the system level. We have the following global actions: $e \downarrow id$, which indicates that the component id connects to e; $e \uparrow id$, which indicates that the component id disconnects from e; $\langle s, t, v, ? \rangle$, which indicates that the value v has been taken from the sink t via a synchronous communication along channel $\langle s, t \rangle$; similarly, $\langle s, t, v, ¿ \rangle$ indicates that the value v has been read from the sink t via a synchronous communication along channel $\langle s, t \rangle$; $\langle id, s, t \rangle$, which indicates that the component instance id has created the channel $\langle s, t \rangle$; finally, $\langle id, id', K \rangle$, which indicates the creation by id of a new component instance id' with the initial set of channel-ends K.

The channels in our transition system are all *synchronous*, since this is the most basic type of channel. Other Channels can be viewed as special types of components whose communication with the rest of

the system can be described using the *synchronous* channels only. Therefore, our transition system generalizes to systems with any type of mobile channels.

connect

$$\frac{\sigma(id) \xrightarrow{e\downarrow} c \text{ and } \gamma(e) =_{\perp} id}{\langle \sigma, \gamma, Chan \rangle \xrightarrow{e\downarrow id} \langle \sigma', \gamma', Chan \rangle}$$

   where $\gamma(e) =_{\perp} id$ holds if $\gamma(e)$ is either undefined or is equal to id, $\sigma' = \sigma[c/id]$, and $\gamma' = \gamma[id/e]$.

   A component instance can connect to a channel-end if either the channel-end is disconnected or it is already connected to the same component instance.

disconnect

$$\frac{\sigma(id) \xrightarrow{e\uparrow} c}{\langle \sigma, \gamma, Chan \rangle \xrightarrow{e\uparrow id} \langle \sigma', \gamma', Chan \rangle}$$

   where $\sigma' = \sigma[c/id]$ and

$$\gamma' = \begin{cases} \gamma[\perp/e] & \text{if } \gamma(e) = id \quad \text{(i.e., } \gamma'(e) = \perp \text{ indicates that } \gamma'(e) \text{ is undefined).} \\ \gamma' = \gamma & \text{if } \gamma(e) \neq id. \end{cases}$$

   A component instance can disconnect from a channel-end if it is currently connected to it.
The *disconnect* operation also succeeds if the component instance was not connected to the channel-end in the first place.

take and **write**

$$\frac{\sigma(\gamma(s)) \xrightarrow{s!v} c \text{ and } \sigma(\gamma(t)) \xrightarrow{t?v} c' \text{ and } \langle s, t \rangle \in Chan \text{ and } \gamma(s) \neq \gamma(t)}{\langle \sigma, \gamma, Chan \rangle \xrightarrow{\langle s,t,v,? \rangle} \langle \sigma', \gamma, Chan \rangle}$$

   where $\sigma' = \sigma[c/\gamma(s)][c'/\gamma(t)]$. The operations *take* and *write* must be performed at the same time on the ends of the same channel. The channel-ends must be connected to the component instances, however, we do not have to check this since the function γ returns only a connected component instance. Since self-communication is a non-global internal issue of the component we must insist that $\gamma(s) \neq \gamma(t)$.

read and **write**

$$\frac{\sigma(\gamma(s)) \xrightarrow{s!v} c \text{ and } \sigma(\gamma(t)) \xrightarrow{t¿v} c' \text{ and } \langle s, t \rangle \in Chan \text{ and } \gamma(s) \neq \gamma(t)}{\langle \sigma, \gamma, Chan \rangle \xrightarrow{\langle s,t,v,¿ \rangle} \langle \sigma', \gamma, Chan \rangle}$$

   where $\sigma' = \sigma[c'/\gamma(t)]$. The case of the operations *read* and *write* is analogous to the case of *take* and *write,* with the exception that the operation *write* does not succeed yet. Only in combination with a *take* operation can a *write* operation succeed, and before then many reads can happen on the same channel. The component instance performing the *write* operation can be seen as an unbounded source of the same value v, until a *take* operation is performed.

new channel

$$\frac{\sigma(id) \xrightarrow{\nu\langle s,t \rangle} c}{\langle \sigma, \gamma, Chan \rangle \xrightarrow{\langle id,s,t \rangle} \langle \sigma', \gamma', Chan' \rangle}$$

   where $\sigma' = \sigma[c/id]$, $\gamma' = \gamma[\perp/s][\perp/t]$ and $\langle s, t \rangle \notin Chan$ and $Chan' = Chan \cup \{\langle s, t \rangle\}$. Upon creation of a new channel, the channel-ends pair must not already exist. The new pair is added to $Chan$. They are initially disconnected in γ.

```
new Component instance
```

$$\frac{\sigma(id) \overset{\nu\langle Comp_i, K\rangle}{\longrightarrow} c}{\langle \sigma, \gamma, Chan\rangle \overset{\langle id, id', K\rangle}{\longrightarrow} \langle \sigma', \gamma, Chan\rangle}$$

where id' does not occur in the domain of σ, $\sigma' = \sigma[c/id][c'/id']$, and $c' = \langle c_0, K\rangle$, with c_0 the initial configuration of $Comp_i$.

The creation of a new component instance consists of the selection of a new component identifier and initializing its configuration.

4.4. Trace Semantics

Given an initial set K of channel-ends, we define formally the interface $Int(Comp, K)$ of a component $Comp$ as the set of component traces

$$\{\theta \mid \langle a_0, K\rangle \overset{\theta}{\longrightarrow}\},$$

where a_0 denotes the initial (abstract) state of $Comp$ and $\overset{\theta}{\longrightarrow}$ is the transitive closure of the transitive relation $\longrightarrow$ of $Comp$ collecting additionally the action-labels into the sequence θ.

In order to obtain the global traces generated by the global transition system in a compositional manner from the interfaces of its components, we introduce a projection operator $P(\theta, id, K)$ that extracts from the global trace θ the local trace of component id assuming that it is (initially) connected to the channel-ends in K.

- `connect`:
$$P(e \downarrow id.\theta, id, K) \quad = e \downarrow .P(\theta, id, K \cup \{e\})$$
$$P(e \downarrow id'.\theta, id, K) \quad = P(\theta, id, K) \qquad\qquad id \neq id'$$

- `disconnect`:
$$P(e \uparrow id.\theta, id, K) \quad = e \uparrow .P(\theta, id, K \setminus \{e\})$$
$$P(e \uparrow id'.\theta, id, K) \quad = P(\theta, id, K) \qquad\qquad id \neq id'$$

- `take and write`:
$$P(\langle s, t, v, ?\rangle.\theta, id, K) = \begin{cases} s!v.P(\theta, id, K) & s \in K \\ t?v.P(\theta, id, K) & t \in K \\ P(\theta, id, K) & s, t \notin K \end{cases}$$

- `read and write`:
$$P(\langle s, t, v, \textrm{¿}\rangle.\theta, id, K) = \begin{cases} s!v.P(\theta, id, K) & s \in K \\ t\textrm{¿}v.P(\theta, id, K) & t \in K \\ P(\theta, id, K) & s, t \notin K \end{cases}$$

- *new channel*:

$$P(\langle id', s, t \rangle.\theta, id, K) = \begin{cases} \langle s, t \rangle.P(\theta, id, K) & id = id' \\ P(\theta, id, K) & id \neq id' \end{cases}$$

- *new component*:

$$P(\langle id', id'', K' \rangle.\theta, id, K) = \begin{cases} \langle id'', K' \rangle.P(\theta, id, K) & id = id' \\ P(\theta, id, K) & id \neq id' \end{cases}$$

We define $P(\theta, id)$ as $P(\theta, id, \emptyset)$.

We have the following compositionality result.

Theorem 4.1. The set of global traces of a system of components $\{Comp_1, ..., Comp_n\}$ generated by the global transition system equals the set

$$\{\theta \mid Ok(\theta) \text{ and } \forall id \in comp(\theta).\ P(\theta, id) \in Int(Comp, K)\},$$

where $comp(\theta)$ denotes the set of component instances occurring in θ. The predicate $Ok(\theta)$ rules out occurrences in θ of communications involving channel-ends that are disconnected.

The proof of this theorem proceeds by a straightforward induction on the length of the computation.

It would be interesting to investigate if the above trace semantics is fully abstract with respect to an appropriate testing equivalence [11].

4.5. Mobile Channels as π-calculus Processes

In section 4.3 we used a simple labelled transition system to model the observable interaction between the components and channels of a system, which is enough for the purposes of this paper. However, this observable interaction can also be modelled using more elaborated semantics like the π-calculus [23]. In [14] we introduce the MoCha-π calculus, an exogenous coordination calculus that extends the π-calculus and is based on mobile channels. Channels in MoCha-π are (special kinds of) π-calculus processes. This allows the calculus to have user defined channel types without having to change the rules of the calculus itself. MoCha-π offers high-level interface *write*, *take*, *connect* and *disconnect* operations on mobile channels. The *write* and *take* actions are dynamically transformed into a pattern of traditional π-calculus synchronous actions, when a process *connected* to a particular channel-end performs one of these I/O actions on it. Therefore, any mobile channel type, like FIFO, is dynamically transformed into a π-calculus process that interacts with its environment by means of synchronous π-calculus channels actions. Just like in our coordination model, in MoCha-π processes have no direct references to channels but only to channel-ends, and therefore, all interface operations are performed on channel-ends. Futhermore, another difference with the π-calculus is that MoCha-π treats channels as resources. Processes must compete with each other in order to gain access to a particular channel-end. More details about MoCha-π and the modelling of mobile channes in the π-calculus can be read in [14].

5. Implementation in Java

The coordination model we present in this paper can be implemented in any object-oriented programming language that supports distributed environments, like Java[18], or C++[28]. 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.

5.1. Components in Java

Usually, JavaBeans [19] are used to implement components in Java. However, they do not comply with our definition of components (see section 2.1) 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 (see section 2.2). Therefore, we impose some restrictions that must be verified by our 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 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 above 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 5.5), 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 consist of one or more objects, one or more threads, its implementation may be distributed, or it may be a channel-based component system itself, etc.

5.2. MoCha

Our Java implementation uses the *mo*bile *cha*nnels provided by the MoCha package. MoCha, is a framework for *mo*bile *cha*nnels in distributed environments that supports mobility as described in section 3. More details on MoCha can be found in [12, 7].

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:
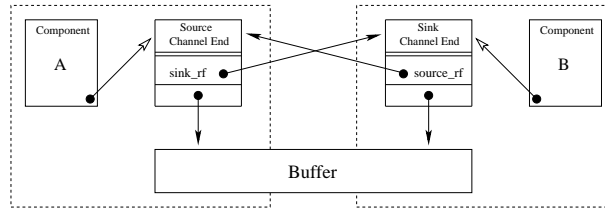
Figure 4.   A mobile Channel in MoCha.

create, *read*, *take*, *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.
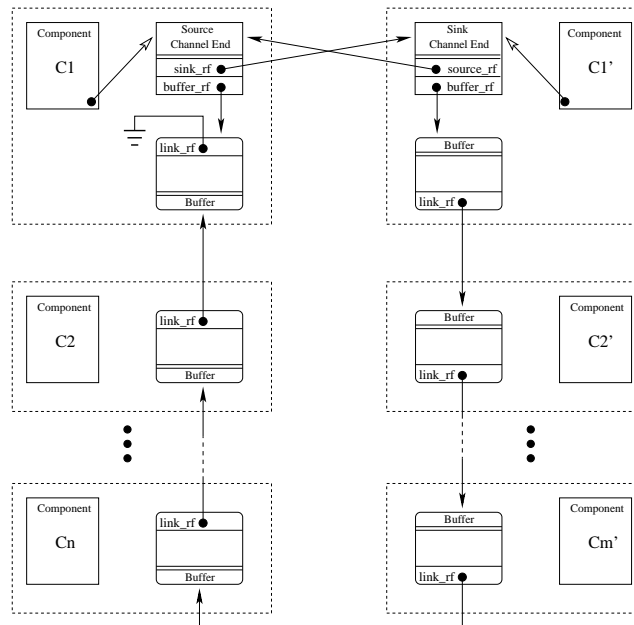

Figure 5.   A FIFO mobile Channel in MoCha.

Figure 5 shows the implementation of an asynchronous FIFO mobile channel in MoCha. The buffer is implemented as 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 *take* 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/take* 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 [12, 7].

MoCha has been implemented in Java using the Remote Method Invocation, *RMI*, package[20].

5.3. Implementation Overview

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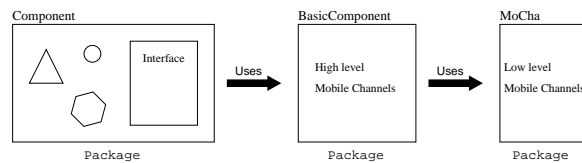

Figure 6.   Implementation Overview

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 may 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 5.5). 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.

5.4. 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. It 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(/* parameters.  For example, an initial set of channel-ends */)
   {
      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 */
   }
}
```

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

The `public` part of the interface consists of three parts (see figures 7 and 8): one or more constructors, a `getLocation` method, and a `finalize` method. 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 its interface class.

Optionally, a `finalize` method can be present to perform cleanup operations before a component instance is garbage collected.

Channel-end references can be passed on through the constructor of the interface. These channel-end references constitute the initial set of mobile channel-ends known to the newly created component instance as defined in section 2.2. Alternatively, a channel-end set reference can be passed on to the component instance for it to return a new set of channel-ends that it creates during the execution of the constructor.

In this implementation we do not describe, nor dictate, any particular way of expressing the observable behavior of a component. For example, one can use the compositional trace-based semantics given in section 4.

The `package private` part of the interface includes the coordination methods provided by the class `BasicInterface` (see figure 8), channel-end variables, and all the other methods and variables in the interface that are not `public`. We explain the coordination methods in section 5.5.

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

class BasicInterface
{
    BasicInterface(Location loc)
    public Location getLocation()
    Object[] CreateChannel(ChannelType type)
    boolean Connect(ChannelEnd ce, int timeout) throws Exception
    boolean Disconnect(ChannelEnd ce) throws Exception
    boolean 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
    boolean 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.

5.5.   The Coordination Operations

The interface of a component provides coordination methods for the active internal objects (i.e., *threads*) in an 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 successful 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 they remain 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 invalid, 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.

5.6. 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, together with 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 it is expected to visit, together with their respective 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

6. 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 [5], [6].

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 aspects. Coordination is expressed merely as operations performed on such channels. The mobility of channels allows dynamic reconfiguration of channel connections within a system.

The *PICCOLA* project [1] is related to our work. PICCOLA is a language for composing applications from software components. It has a small syntax and a minimal set of features needed for specifying different styles of software composition, e.g. *pipes and filters*, *streams*, *events*, etc. At the bottom level of PICCOLA there is an abstract machine that considers components as *agents*. These agents are based on the π-calculus, but they communicate with each other by sending *forms* through shared channels instead of tuples. Forms are a special notion of extensible, immutable records. In comparison with PICCOLA, our coordination model can be seen as a possible *mobile channel* style for component composition. Therefore, the interfaces of our components are defined in such a way that they already fit within this style. Because our model only focuses on the *mobile channel* style, it is much simpler to use when this style is desired. However, our model is not just a style but also, like PICCOLA, a composition language.

Certain aspects of and concerns in *ROOM* [27] and *Darwin* [21], two architectural description languages (ADL), are related to our work. In ROOM components are described by declaring their internal structures, their external interfaces, and the behavior of their sub-components (if they are composite components). 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 them 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.

Because our model provides *exogenous* coordination (see section 2.3), it 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 *Reo*[3, 4]. *Reo* 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 *Reo*.

Finally, although it is not the main purpose of this paper, the Java implementation presented in section 5 shows not only that components can be implemented using object-oriented languages, but also how this can be done. This demonstrates that a clear integration of components is possible in object oriented paradigms such as UML.

References

[1] F. Achermann, M. Lumpe, J. Schneider, and O. Nierstrasz. *Piccola - a Small Composition Language*, Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches, Howard Bowman and John Derrick (Eds.), pp. 403-426, Cambridge University Press, 2001.

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

[3] F. Arbab, *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science, Vol. 14, No. 3, pp. 329-366, June 2004.

[4] F. Arbab, *A channel-Based Coordination Model for Component Composition*, Tech. Report, Centrum voor Wiskunde en Informatica, Amsterdam, 2002. Available on-line `http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0203.pdf`

[5] 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.

[6] 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.

[7] F. Arbab, F.S. de Boer, M.M. Bonsangue, and J.V. Guillen-Scholten, *MoCha: a Middleware Based on Mobile Channels*, to appear in proceedings of 26th Int. Computer Software and Application Conference (COMPSAC 02) IEEE Computer Society Press, 2002.

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

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

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

[11] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988

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

[13] J.V. Guillen-Scholten, F. Arbab, F.S. de Boer, and M.M. Bonsangue, *Mobile Channels, Implementation Within and Outside Components*, A. Brogi and E. Pimintel, editors, Proceedings of Formal Methods and Component Interaction, ENTCS 66.4, Elsevier Science, 2002.

[14] J.V. Guillen-Scholten, F. Arbab, F.S. de Boer, and M.M. Bonsangue, *MoCha-pi, an Exogenous Coordination Calculus based on Mobile Channels* Proceedings of the 2005 ACM Symposium on Applied Computing, Santa Fe, New Mexico, USA, March 13-17, 2005.

[15] D. Hay, *COM+ Technical Series: Queued Components*, Microsoft Corporation, 1999.

[16] 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.

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

[18] Home Page of Java, `http://java.sun.com`

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

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

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

[22] Microsoft Corporation. Home page of COM+, `http://www.microsoft.com/com/tech/complus.asp`.

[23] R. Milner, *Communicating and Mobile Systems : The Pi-Calculus*, Cambridge University Press, May 20, 1999.

[24] A.L. Murphy, G.P. Picco, and G.-C. Romjan. *Lime: A coordination middleware supporting mobility of hosts and agents*. Technical Report WUCSE-03-21, Washington University, Department of Computer Science, St. Louis, MO (USA), 2003.

[25] 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`.

[26] 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.

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

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

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

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

[31] 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/`.

[32] P. Wojciechowski, and P. Sewell, *Nomadic Pict: Language and Infrastructure Design for Mobile Agents*, First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99), Palm Springs, CA, USA, 1999.

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