

Mobile Channels, Implementation Within and Outside Components

Juan Guillen Scholten ^{a,1} Farhad Arbab ^{a,1} Frank de Boer ^{a,1}
Marcello Bonsangue ^{b,2}

^a *CWI, Amsterdam, The Netherlands*

^b *LIACS, Leiden University, The Netherlands*

Abstract

In this paper we advocate and promote the use of mobile channels for component-based software. Channels allow anonymous and point-to-point communication among components, while mobility allows dynamic reconfiguration of channel connections in a system. Models based on mobile channels provide 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 way. Besides promoting channels we also present such a model for component composition and coordination that supports dynamic distributed systems where components can be mobile. We do this by giving an implementation of mobile channels (outside components), and a basic and extendable implementation of components that interact with each other through them (implementation within components).

1 Introduction

The importance of 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* [7]. 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*[19], which is an integration of the architectural description language *ROOM*[20] into *UML*.

¹ Email: {juan, farhad, frb}@cwi.nl

² Email: marcello@liacs.nl

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. 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. 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.

To promote this clear separation of concerns we advocate the use of mobile channels for component composition and coordination. 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. In [6] we present a model for coordination and composition for components based on mobile channels. In this 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 of the component is expressed by using, for example, predicates, comments, or some graphical notation.

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 [3]. 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.

In this paper, besides promoting the use of mobile channels, we discuss a model based on them for component composition and coordination. We do this by giving the implementation of mobile channels (outside components), and a basic and extendable implementation of components that interact with each other through them (implementation within components). At the end we discuss related work.

2 Mobile Channels and their Benefits

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 val-

ues into 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, just the channel-ends they have access to. Channels can be synchronous or asynchronous (FIFO, Set, Bag, etc.).

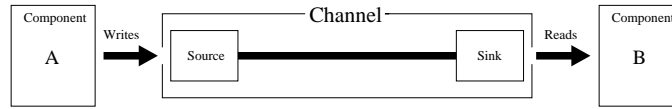


Fig. 1. A Channel.

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 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. For example, mobile Internet agents (see section 3).

In [6] we discuss and compare mobile channels with three important coordination mechanism for component composition: *messaging*, *events*, and *shared data spaces*. We argue that channels share many of the architectural strengths of these mechanisms while offering some additional benefits. The major commonalities and additional benefits are (in short):

- *efficiency*. For most networks point-to-point channels can be implemented very efficiently in truly distributed systems.
- *security*. Point-to-point channels support a *private* means of communication that prevents third parties from accidentally or intentionally interfering with the private communication between two components.
- *architectural expressiveness*. Using channels to express the communication carried out within a system is architecturally very expressive, because it is easy to see which components exchange data with each other. This makes it easier to apply tools for analysis of the dependencies and data-flow.
- transparent *exogenous coordination*. Channels allow several different types of connections among components, e.g., synchronous, FIFO, etc, without the components knowing which channel types they deal with. This makes it possible to coordinate components from 'outside' (exogenous).

3 An Example of Mobile Channel Usage

In this section we illustrate the utility and benefits of mobile channels by giving an example that involves mobile Internet components.

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.

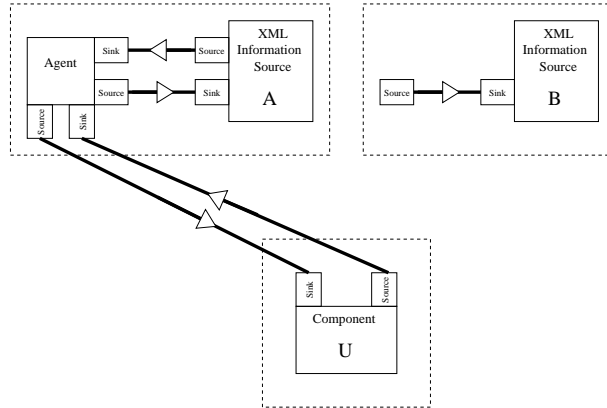


Fig. 2. An Example: a Hopping Agent.

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 represented 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 its 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 (we assume that the channel is of type asynchronous), 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

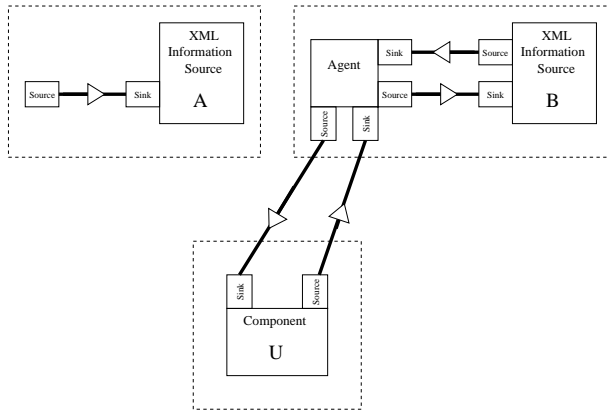


Fig. 3. Moving to Another Location.

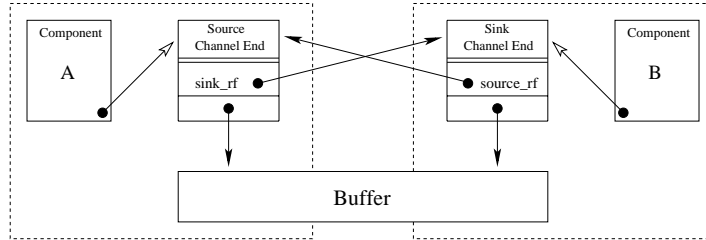


Fig. 4. A mobile Channel in MoCha.

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 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.

As explained in the last section, mobile channels allow *exogenous coordination*. Therefore, we can choose the types of the channels in order to coordinate the components (U , *Mobile Agent*, and *Information Source*) from 'outside'. For example, we can choose either synchronous mobile channels between the *Mobile Agent* and the *Information Sources* to synchronize the data transfer between the two, or we might consider using asynchronous channel types. All this can be done without rewriting or recompiling the components in the example.

4 Implementation of Mobile Channels Outside Components

MoCha[10,11], is an implementation model for *mobile channels* in distributed environments that supports mobility as described above.

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 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 become 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; with proper heuristics for data-transfer between buffers, in real distributed systems, statistically most reads turn out to be local operations. *move* operations do not involve data-transfer of elements at all [10].

MoCha has been implemented in Java [14] using the Remote Method Invocation, *RMI*, package [15].

5 Implementation of Mobile Channels Within Components

In [6] we give a basic and extendable implementation of components in the Java [14] language to support coordination through mobile channels.

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 (component access is only possible through its interface). Therefore, we must impose some restrictions that must be verified by a *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`.

One major advantage of these restrictions is that they 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,


```

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

```

Fig. 6. The Coordination Methods of the Interface

ods provided by its interface(s). Our implementation provides basic operations on channels. 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.

The basic operations are listed in figure 6. We proceed by giving a short description of these operations. More details, including examples, can be found in [6].

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`.

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.

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.

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.

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. The value is not removed from the channel.

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.

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.

6 Related Work

The idea of using (mobile) channels for components has its foundations in the earlier work of some of the authors of this paper [4,5], in the CSP model [13], in the π -calculus [17], and in Broy's work on streams [8].

Besides *MoCha* [10,11], other systems that use and implement channels include: *Communicating Threads for Java* [12], *CSP for Java* [21], both based on the CSP model, and *Pict* [18], a concurrent programming language based on the π -calculus. However, these systems either do not support distributed environments, or their channels are not mobile. *Nomadic Pict* [22], a distributed version of *Pict*, does implement distributed mobile channels. However, its channels do not have two distinct ends and their type is only synchronous.

The *PICCOLA* project [1] is related to our coordination model for components. *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. 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 [2,3].

Also, certain aspects of and concerns in *ROOM* [20], *Darwin* [16], and *LEDA* [9], three architectural description languages (ADL), are related to our mobile channel model for component composition and coordination.

7 Conclusion and Future Work

In this paper we advocate the use of mobile channels for component-based software. We show the benefits of using these channels, and present a model based on them for component coordination and composition. This model consists of an implementation of mobile channels and components that support communication through these channels.

Other models for component-based software can benefit from the coordina-

tion 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 supports *exogenous* coordination, 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 $P\epsilon\omega$ [2,3]. $P\epsilon\omega$ supports composition of channels into complex connectors whose semantics are independent of the components they connect to. We are currently extending our coordination model in order to support all the features of $P\epsilon\omega$.

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] 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 SEN-R0203, Centrum voor Wiskunde en Informatica, Amsterdam, 2002.
- [4] F. Arbab, *Manifold Version 2: Language Reference Manual*, Technical Report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996. Available online at the URL:
<http://www.cwi.nl/ftp/manifold/refman.ps.Z>
- [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] F. Arbab, F. S. de Boer, M. M. Bonsangue, and J.V. Guillen Scholten, *A channel-based coordination model for components*, Tech. Report SEN-R0127, Centrum voor Wiskunde en Informatica, Amsterdam, 2001.
- [7] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, Mass. USA, 1999.
- [8] M. Broy, K. Stolen, *Specification and development of interactive systems : FOCUS on streams, interfaces, and refinement*, Springer, ISBN 0-387-95073-7, New York, 2001.
- [9] C. Canal, E. Pimentel, and J. M. Troya, *Specification and Refinement of Dynamic Software*, in Software Architecture, Kluwer Academic Publishers, pages 107-126, 1999.

- [10] J.V. Guillen Scholten, *MoCha, a Model for Distributed Mobile Channels*, Internal Report 01-07, Master's Thesis, LIACS, Leiden University, May 2001.
- [11] J.V. Guillen Scholten, F. Arbab, F.S. de Boer, and M.M. Bonsangue, *MoCha: a Middleware Based on Mobile Channels*, to appear in proceedings of 26th Int. Computer Software and Application Conference (COMPSAC 02) IEEE IEEE Computer Society Press, 2002.
- [12] 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.
- [13] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, London, UK, 1985.
- [14] Home Page of Java, <http://java.sun.com>
- [15] Home Page of RMI documentation,
<http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html>.
- [16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*. In Proceedings of 5th European Software Engineering Conference, Spain, 1994.
- [17] M. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes, parts I and II*, Information and Computation 100:1, 1992, pp. 1-77.
- [18] 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>.
- [19] 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.
- [20] B. Selic, G. Gullekson, and P.T. Ward, *Real-Time Object-Oriented Modelling*, John Wiley and Sons, Inc., 1994.
- [21] 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/>.
- [22] 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.
- [23] World Wide Web Consortium, *eXtensible Markup Language*,
<http://w3c.org/XML/>.