Mobile Channels for Exogenous Coordination of Distributed Systems

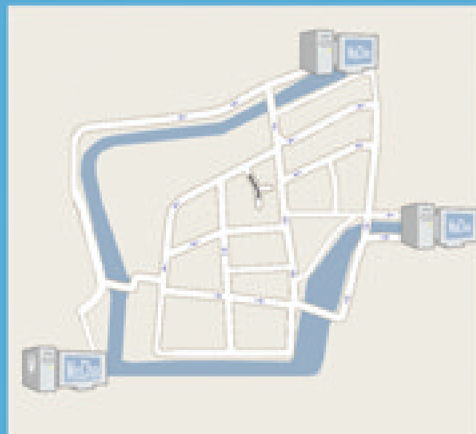
Semantics, Implementation and Composition

Juan V. Guillen Scholten

Mobile Channels for Exogenous Coordination of Distributed Systems

Semantics, Implementation and Composition

Mobile Channels
for Exogenous Coordination
of Distributed Systems
Semantics, Implementation and Composition



Juan V. Guillen Scholten

Mobile Channels for Exogenous Coordination of Distributed Systems

Semantics, Implementation and Composition

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschapen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 10 januari 2007
klokke 16.15 uur

door

Juan Visente Guillen Scholten

geboren te Delft
in 1974

Promotiecommissie

Promotor:	Prof. Dr. F. Arbab
Co-promotores:	Dr. F.S. de Boer
	Dr. M.M. Bonsangue
Referent:	Prof. Dr. J.-M. Jacquet *University of Namur, Belgium*
Overige leden:	Prof. Dr. A. Brogi *University of Pisa, Italy*
	Prof. Dr. J.N. Kok
	Prof. Dr. W.-P. de Roever *Christian-Albrechts-University of Kiel, Germany*
	Prof. Dr. S.M. Verduyn Lunel

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Cover design by Federico Guillen Scholten.

IPA Dissertation Series 2006-21
ISBN 90-6196-541-1

Copyright © 2006 by Juan V. Guillen Scholten

Contents

I Introduction 1

1 Introduction 3
- 1.1 Context . . . 3
- 1.2 The Thesis . . . 4
- 1.3 Contributions . . . 4
- 1.4 Outline of the Thesis . . . 6

2 Mobile Channels 7
- 2.1 Introduction . . . 7
- 2.2 Mobile Channels . . . 8
  - 2.2.1 Operations . . . 8
  - 2.2.2 Features and Benefits . . . 9
- 2.3 Mobile Agent Example . . . 11
- 2.4 A Set of Channel Types . . . 13
  - 2.4.1 Coordination and Communication Types . . . 13
  - 2.4.2 Coordination Only Types . . . 15
  - 2.4.3 Other Types . . . 16
- 2.5 Discussion . . . 17

II Semantics 19

3 Semantics without Mobility 21
- 3.1 Introduction . . . 21
- 3.2 A Short Introduction to Petri Nets . . . 22
  - 3.2.1 Elementary Net Systems . . . 23
- 3.3 PN Semantics for Mobile Channels and Components . . . 27
  - 3.3.1 Mobile Channel Interface . . . 27
  - 3.3.2 Component Interaction . . . 28
  - 3.3.3 PN Composition of Components and Channels . . . 29
  - 3.3.4 A Set of EN and P/T-net Mobile Channels Systems . . . 30
- 3.4 Analysis and Simulation of PN Models . . . 36
- 3.5 Discussion and Concluding Remarks . . . 38

4 Semantics with Mobility **41**
4.1 Introduction 41
4.2 Extending the π-calculus 42
4.3 The MoCha-π Calculus 43
4.3.1 Threads, Channels, Processes and Resources 44
4.3.2 Structural Congruence 46
4.3.3 Actions 47
4.3.4 Reaction and other Rules 49
4.3.5 Sequential Composition 49
4.4 The MoCha Framework Design Pattern 50
4.4.1 Specifying Channel Types 50
4.5 Examples 54
4.5.1 Producer/Consumer Examples 54
4.5.2 Mobile Phones 58
4.5.3 Mobile Agent 61
4.6 Conclusions and Related Work 63

5 Channel-based Semantics for Component Based Software **65**
5.1 Introduction 65
5.2 Components and their Composition 66
5.2.1 Component-Based Software 66
5.2.2 Components and their Interfaces 67
5.2.3 Coordination Among Components 68
5.3 A Semantic Approach 70
5.3.1 Component Transition System 70
5.3.2 Local Conditions 71
5.3.3 Global Transition System 72
5.3.4 Trace Semantics 73
5.4 Discussion 75

III Implementation **77**

6 The MoCha Middleware: API and Applications **79**
6.1 Introduction 79
6.2 The Application Programming Interface (API) 81
6.2.1 Location and Keys for Components 81
6.2.2 Mobile Channels: Creation and Types 83
6.2.3 Source & Sink Channel-End 85
6.2.4 Channel-End 90
6.3 Examples 91
6.3.1 Producer/Consumer 91
6.3.2 Producer/Producer and Consumer/Consumer 93
6.3.3 Competing Producers and Consumers 95
6.3.4 Untyped Producer/Consumer 96
6.3.5 Crazy Producer and Cooperating Consumers 97
6.4 Applications 100
6.4.1 Component Based Systems 100

6.4.2 Web Services 101
6.4.3 Hybrid and Pure P2P Networks 103
6.5 Related Work 107
6.5.1 Channel-Based Middleware 108
6.5.2 Coordination Middleware 111
6.5.3 P2P Middleware 113

7 The MoCha Middleware: Implementation Details 115
7.1 Introduction 115
7.1.1 Figure Legend 116
7.2 Choosing the Right Infrastructure 117
7.3 Remote Method Invocation 119
7.4 MoCha's Mobile P2P Architecture 120
7.4.1 Using RMI for P2P Networking 120
7.4.2 Mobility on top of RMI 123
7.5 MoCha's Channels 125
7.5.1 General Mobile Channel Implementation Overview 125
7.5.2 Example: FIFO Channel Implementation 127
7.5.3 Channel Implementation Issues 135
7.6 Performance Measurements 139
7.6.1 RMI and MoCha 139
7.6.2 Comparing Channel Types 139
7.6.3 (Static) MoCha vs. LighTS 142
7.6.4 Movement of Channel-ends 144
7.6.5 Mobile Components (Moderated Mobility) 144
7.6.6 Mobile Components (High Mobility) 146

8 An Implementation of the Channel-Based Component Model 149
8.1 Integration of Components with Object-Oriented Technology 149
8.2 Implementation in Java 150
8.2.1 Components in Java 150
8.2.2 Implementation Overview 151
8.2.3 The Interface of a Component 151
8.2.4 The Coordination Operations 153
8.2.5 A Small Example 155
8.3 Related Work and Conclusions 155

IV Composition 159

9 Composition of Mobile Channels 161
9.1 Introduction 161
9.2 Reo 162
9.2.1 Composition of Connectors 163
9.2.2 More about Reo 165
9.3 Issues Concerning the Distributed Implementation of Reo Connectors 166
9.4 MoCha's Coordination Components Model 167
9.4.1 Replicator 168

9.4.2 Non-Deterministic Multiplexer 172
9.4.3 Ordered Multiplexer 175
9.4.4 Non-Deterministic Demultiplexer 176
9.4.5 Ordered Demultiplexer 178
9.4.6 Write Gate Transistor 179
9.4.7 Take Gate Transistor 181
9.4.8 Write Switch 182
9.4.9 Take Switch 185
9.4.10 Useful Connectors 186
9.4.11 Composition of Connectors 188
9.5 Distributed Dining Philosophers 189
9.6 Comparisons and Conclusions 193
9.6.1 Implementing a Subset of Reo 196

V Conclusion 197

10 Conclusion 199
10.1 A Short Summary 199
10.2 Answering the Main Question of this Thesis 200
10.3 Future Work 202

Bibliography 204

A MoCha's Abstract Algorithms 215
A.1 Synchronous Channel 217
A.2 Lossy Synchronous Channel 222
A.3 Asynchronous FIFO Channel 223
A.4 Synchronous Drain Channel 230
A.5 Asynchronous Spout Channel 234

Summary 236

Samenvatting 239

Curriculum Vitae 241

Part I

Introduction

Chapter 1

Introduction

In the last years, there has been a growing interest for distributed systems in computer science. A distributed system is a collection of independent computers that appears to its users as a single coherent system [TS02]. An example of such a system is the Internet, which is the biggest distributed system in the world. The independent computers of a distributed system are connected to each other through a network. On each of these computers there is at least one software entity (like threads, components, databases, applications, etc.) that needs to communicate with software entities on other computers to achieve some goal. These software entities are not only distributed among the several computers of a network but they also run in parallel. Therefore, one of the main challenges in distributed systems is to develop appropriate theory and infrastructures for the communication and the coordination of its concurrently running software entities.

In this thesis we investigate the notion of mobile channels. The main question is: are mobile channels suitable as a communication and coordination mechanism for distributed systems? To answer this question, we throughly investigate mobile channels. We do this by defining *semantics* for them, *implementing* them, and providing a model for *composition* of channels into connectors.

1.1 Context

The work of this thesis is categorized under the computer science field of *coordination.* An introduction and explanation of this field is given by Arbab in [Arb98], where coordination is defined as the study of the dynamic topologies of interactions among concurrent programs, processes and components of a system, with the goal of finding solutions to the problem of managing these interactions. For this purpose, there exist many coordination models, languages, applications and mechanisms.

Arbab classifies these coordination frameworks as either *data-oriented* or *control-oriented.* The activity in a data-oriented coordination framework tends to center around a substantial shared body of data; the framework is essentially concerned with what happens to the data. On the other hand, the activity in a control-oriented coordination framework tends to center around the flow of control; the data flow in the models and the implementations of these kind of frameworks is more important

than the data itself.

Another classification that Arbab makes for coordination frameworks is for them to be either *endogenous* or *exogenous.* The first kind provides coordination primitives that must be incorporated *within* a computation for its coordination. In applications that use such frameworks, primitives that affect the coordination of each module are inside the module itself. In contrast, the second kind of frameworks provide primitives that support coordination of entities from *without.* In applications that use such frameworks, primitives that affect the coordination of each module are outside the module itself.

According to above classifications, an example of a data-oriented and endogenous coordination language is Linda [CG90], and an example of a control-oriented exogenous coordination language is MANIFOLD [Arb96a]. We position our work in the coordination community by classifying the mobile channels of this thesis as control-oriented and exogenous. For more in depth details about these classifications and the field of coordination in general we refer to [Arb98].

1.2 The Thesis

In this thesis we present a *novel* coordination framework that is based on mobile channels. We call this framework *MoCha.*

MoCha offers semantics for the specification of the coordination part of distributed systems and an implementation of mobile channels for the realization of these specifications. The MoCha semantics and implementation are decoupled from each other so that they can be used independently if desired.

MoCha is a *practical* framework, for there is a straightforward relation between its semantics and its implementation. This makes it easy to actually implement the desired coordination specified with the semantics. This relation also ensures that the MoCha semantics are (immediately) implementable. MoCha supports many coordination scenarios with different architectural styles and with their corresponding distributed architecture.

With MoCha, we *demonstrate* the usefulness of mobile channels as a communication and an exogenous coordination mechanism for distributed systems.

1.3 Contributions

The specific contributions of this thesis can be divided in three groups: *semantics*, *implementation* and *composition.*

In general, semantics is used for modeling, specification and verification purposes. We define three different kinds of semantics for mobile channels:

- We focus on the concurrency aspect of mobile channels by specifying them in the Petri Nets formalism. We show how to construct Petri Net models consisting of mobile channels and components that use them, for the purpose of analyzing and simulating these models with existing Petri Net theory and tools.

- We focus on process interaction by extending the π-calculus with mobile channels. We introduce exogenous coordination in this process algebra by allowing user defined channel types. These channels are not links as in the traditional π-calculus but special kinds of processes. We also introduce the notion of channels as resources; processes must compete with each other in order to gain access to a particular mobile channel.

- We give compositional trace-based semantics for a component-based software model based on the notion of mobile channels. We demonstrate how suitable mobile channels are for this kind of system, by showing that they provide a highly expressive data-flow architecture for the construction of complex coordination schemes, independent of the computation parts of components; i.e. mobile channels enhance the separation of concerns between the coordination and the computational aspect of component-based systems.

We present and discuss the details of two implementations that we realized regarding mobile channels:

- We implemented mobile channels in distributed systems by developing the MoCha middleware. With this middleware we show that mobile channels can efficiently be implemented in distributed systems; this holds for both centralized client/server and decentralized peer-to-peer distributed systems architectures. We also show that mobile channels have an easy and intuitive interface towards components, processes and threads. More technical contributions of the middleware are:
  - A discussion of the technical difficulties and our particular solutions regarding the synchronization of distributed software entities.
  - A peer-to-peer architecture that is build upon a client/server one.
  - An architecture for object mobility in distributed systems.

- We implemented the model for component-based software that we mention above. This implementation demonstrates that object-oriented languages are well-suited to implement components and their composition using our coordination framework of mobile channels. We also show how to integrate the mobile channels of the MoCha middleware with this implementation of component-based technology.

We provide a model for composition of mobile channels that takes some of the ideas of the already existing model *Reo* [Arb02]. Our main contribution here is the actual implementation of a subset of Reo in distributed systems. For this purpose, we introduce the idea of coordination components. These are lightweight components that are meant for linking channels together according to some transparent coordination behavior; for example, the coordination specified by parts of a Reo model (or connector). Our model can be used independently of Reo; the specification of our models (or connectors) is the composition of the semantics we define in this thesis, for the actual implementation of these models we use the MoCha middleware.

1.4 Outline of the Thesis

We organized this thesis as follows:

- In Chapter 2, we give an intuitive explanation of mobile channels. This explanation is needed to understand the other chapters of this thesis.
- In Chapter 3, we use Petri Nets to define a semantics for mobile channels without considering mobility. This semantics helps us understand the channel's (static) basic behavior, and serves as a guideline to understand the semantics in the next chapter.
- In Chapter 4, we give a semantics to mobile channels by defining an exogenous coordination calculus. This calculus, called MoCha-π, (also) models the mobility aspect of channels.
- In Chapter 5, we present a coordination model for *component-based software* systems based on the notion of mobile channels. We give a semantics for this model, taking the point of view of the components that use mobile channels. This is in contrast with the two previous chapters above, where we focus more on the mobile channels themselves.
- In Chapter 6, we discuss the *MoCha middleware*. This is the middleware that we developed to implement the mobile channels in this thesis. We have distributed the explanation of the middleware in two chapters. In this first chapter, we take the point of view of a distributed system developer who wants to use the middleware but does not want to know anything about its internal implementation details.
- In Chapter 7, we continue with the *MoCha middleware* by discussing its implementation details. We conceptually explain the many algorithms and the internal architecture of the middleware. We also provide performance measurements.
- In Chapter 8, we present the implementation of the coordination model for component-based software that we introduced in Chapter 5.
- In Chapter 9, we investigate the composition of channels. We do this by providing a model for composition based on the notion of *coordination components*, where we compose mobile channels into connectors. For specifying these components we use the Petri Net semantics of Chapter 3 and the MoCha-π semantics of Chapter 4.
- In Chapter 10, we end with conclusions and discuss future work.

Chapter 2

Mobile Channels

A mobile channel is a coordination primitive that allows *anonymous* and *point-to-point* communication between components in a distributed system. The ends of such a channel are mobile. This makes it possible to have dynamic reconfiguration of channel connections between components over time in arbitrary ways. Furthermore, mobile channels provide (basic) exogenous coordination. Channels allow several different types of connections among components without the components themselves knowing which channel types they are dealing with.

2.1 Introduction

From a general point of view a *channel* is a path between two end points. For example: a passage for water to flow through, a path over which electrical signals can pass, or a path over which radio waves are transmitted. In this thesis we view a channel as a communication and coordination primitive between active (software) entities like *processes*, *threads*, and *components*. This view is not new, and has its foundations on well-known computer science channel-based models such as Hoare's *CSP* [Hoa85], Milner's *CCS* [Mil80], and Milner's π-calculus [Mil99]. In these models processes use channels for communication between them.

The motivation for focusing more on the coordination aspects of channels comes from Arbab's *IWIM* model [Arb96b], where not only the communication between processes is important but also their coordination. The idea and definition of *mobile channels* as coordination and communication primitives between components in distributed systems comes from Arbab's *Reo* model [Arb02, Arb04].

In this chapter we give an intuitive explanation of mobile channels. In section 2.2 we start with an overview, where we present the major channel-end operations and discuss the mobile channel features and benefits. In section 2.3 we give an example of how to use mobile channels illustrating their benefits. We end with section 2.4, where we give a representative set of mobile channel types. These are the channel types that we use in this thesis.

2.2 Mobile Channels

In this section we introduce the general notion of mobile channels. This notion guides us through the other chapters of this thesis. Observe that, in our explanation and in the rest of this thesis we often refer to *component instances* as *components* when the context is clear enough to allow such a simplification.

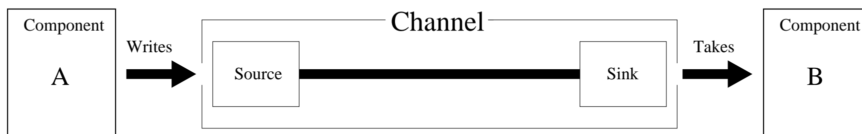


Figure 2.1: General View of a Channel.

A mobile channel is a coordination primitive for communication and coordination between components in distributed systems. A channel consists of exactly two distinct ends: usually ⟨*source*, *sink*⟩ for most common channel-types, but also ⟨*source*, *source*⟩ and ⟨*sink*, *sink*⟩ for special types (see Section 2.4). From the point of view taken in Figure 2.1, one can see that the ends of a channel are *internally* (somehow) related to each other. Therefore, we can regard the source-end as the input-point of a channel, and the sink-end as the output-point of the same channel. However, the components of a system don't have any knowledge of channels nor references to them. Instead, components may know and refer to channel-ends only. This means that any component that has a reference to a specific channel-end does not automatically have a reference to the other end of the channel. Furthermore, it also means that all operations are performed on channel-ends instead of channels.

2.2.1 Operations

We define six basic main operations that can be performed on channel-ends. All channel operations are *synchronous*, or *blocking*; i.e. the active entity performing the operation suspends its execution until the operation is finished. We omit less fundamental operations, like *inquiry* ones, because we don't need them until the introduction of the MoCha middleware in Chapter 6. All these operations are defined in and come from *Reo* [Arb02]. We start with the *Input/Output operations*:

- *write.* Components can write to insert values into the source-end of a channel.
- *take.* Components can take values by removing them from the sink-end of a channel.
- *read.* Read is the non-destructive version of take. The read operation copies the value offered by the sink-end and leaves the original behind for another take or read operation.

The data-flow of the I/O operations is locally *one way*: from a component into a channel-end or from a channel-end into a component. Besides values, components may write/take/read channel-end references. This way, components can increase

their knowledge of currently available channel-ends in the system.
We continue with the *topological operations*:

- *move*. The move operation *physically* moves a channel-end from one location to another target location in the network. This operation will become more clear when we introduce channel-end mobility in Section 2.2.2.

The first four operations are enough to properly work with mobile channels. Some versions of the MoCha middleware, as presented in Chapter 6, support this basic set of operations. These middleware versions offer *many-to-many* mobile channels to their users. This means that, many components can use a particular channel-end at the same time; however, a channel-end performs only one operation at a time.

In many cases we want the communication to be *one-to-one*. This means that components need to have exclusive access to channel-ends. In the *chocoMoCha* middleware (see Chapter 6) and in the semantics given in the next chapters, we view channel-ends as resources. Therefore, components must now compete with each other in order to gain access to a particular channel-end. For this purpose, we introduce two more operations:

- *connect*. A component successfully connects to a particular channel-end if either no other component is currently connected to it, or it is already connected to this particular channel-end. In all other cases the component must wait until this channel-end becomes available.
- *disconnect*. A component always succeeds in executing this operation. Either the performing component is connected to a particular channel-end and gets disconnected from it by this operation, or it is not connected to this channel-end in the first place and, therefore, stays disconnected after this operation.

2.2.2 Features and Benefits

Channels are *point-to-point*, they provide a (locally directed) virtual path between the components involved in a connection. This leads to three immediate benefits: *support for several architectural styles*, *efficiency*, and *architectural expressiveness*.

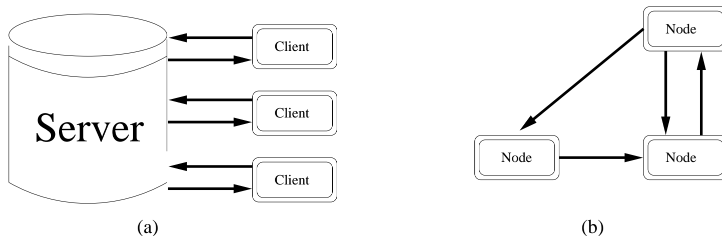


Figure 2.2: A Client/Server and Peer-to-Peer Architecture using Channels

Channels offer *support for several architectural styles*, because channels can both model and implement the communication/coordination aspect of several *centralized*

and *decentralized* architectures [Sch01]. For example, in Figure 2.2 we show the two most common architectural styles: (a) a centralized *client/server* architecture, and (b) a decentralized *peer-to-peer* (P2P) architecture. The black thick arrow lines denote the channels between the various components. In the first architecture, servers are components dedicated to specific tasks like managing of disk drives, printers, or network traffic, whereas clients are components that rely on servers for resources. Therefore, in Figure 2.2(a) we implement the architecture by simply placing two channels between each client and the server. In the second architecture, each node component is both a client and a server at the same time. Therefore, the nodes are said to be *equal*. They have equivalent responsibilities, enabling applications that focus on collaboration and communication in a decentralized and self organizing way. In principle, each node can communicate with any other node, there are no architectural restrictions. In Figure 2.2(b) we show a possible P2P network with a specific channel configuration.

Point-to-point channels can be *efficiently* implemented in distributed systems. Especially for decentralized systems, like P2P-networks. For this to be the case, channels themselves need to be implemented in a decentralized way, like in our MoCha middleware in Chapter 6.

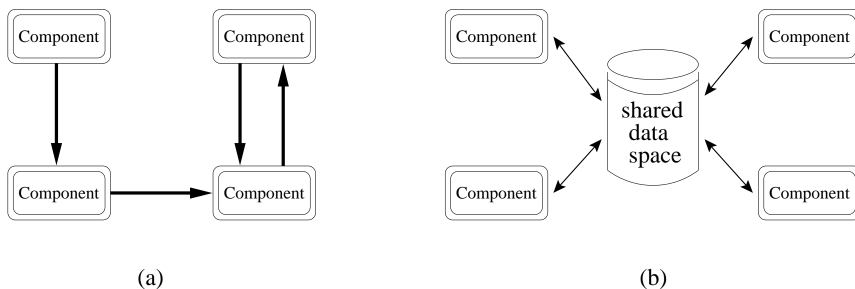


Figure 2.3: Architectural Expressiveness.

Using channels to express the communication carried out within a system is *architecturally very expressive*, because it is easy to see which components (potentially) exchange data with each other at a particular point in time. This makes it easier to apply tools for analysis of the dependencies and data-flow. In Figure 2.3 we give an example of a system using channels (a), and the same system using a shared data space [And91] (b). In 2.3(a), it is indeed very easy to see which components (potentially) exchange data. In 2.3(b), we have the same four components. The only thing that we can conclude from this figure is that every component potentially exchanges data with every other component. However, we know from Figure 2.3(a) that this is not the case. Therefore, we need extra information about the components and the shared data space to rule out some non-existing component interactions. In Figure 2.3(a) this information is already present due to the channel connections.

Besides providing point-to-point communication, channels also provide *anonymous communication*. This enables components to exchange messages with other components without having to know *where* in the network those other components reside, *who* produces and consumes the exchanged messages, and *when* a particular

message was produced or will be consumed. Since the components do not know each other, it is easy to update or exchange any one of the components without the knowledge of the components at the other sides of the channels it is connected to.

Channels provide transparent (basic) *exogenous coordination* [1]. Channels allow several different types of connections among components without the components themselves knowing which channel types they are dealing with. Only the creator of the channel knows its type, which is (usually) either synchronous or asynchronous. This makes it possible to coordinate components from the 'outside' (exogenous), and thus, change the system's behavior without changing the components.

The *anonymous communication* and *exogenous coordination* features of channels promote and enhance the separation of concerns between the coordination and the computational aspect of a system. This makes it easy to develop, maintain and update the coordination part of a system independently of its computational part.

The ends of a channel are *mobile*. We introduce here two definitions of mobility: logical and physical. The first is defined as the property of passing on channel-end identities through channels themselves to other components in the system; spreading the knowledge of channel-end references by means of channels. The second is defined as physically moving a channel-end from one location to another location in a distributed system, where location is a *logical address space* where components execute. Both kinds of mobility are supported by the MoCha-framework.

Because communication via channels is also *anonymous*, when a channel-end moves, the component at the other side of the channel is not aware nor affected by this movement.

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 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.

2.3 Mobile Agent Example

In this section we illustrate the use and benefits of mobile channels through an example that involves mobile Internet components. Suppose we want to use agents to search for some specific information, e.g., coffee prices, on the Internet. Agents consult different XML[XML00] 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

[1] Not all exogenous coordination can be done with a set of single mobile channels. However, mobile channels provide basic exogenous coordination. By regarding them as basic blocks and composing them together, we can actually cover all exogenous coordination (as we demonstrate in Chapter 9).

locations. An alternative that we will consider later is to create an agent at every location.

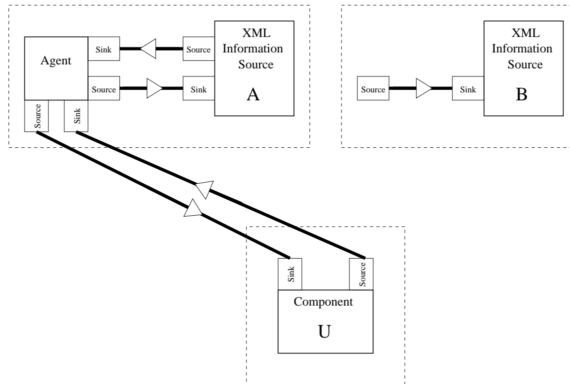


Figure 2.4: An Example: a Hopping Agent.

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 2.4 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.

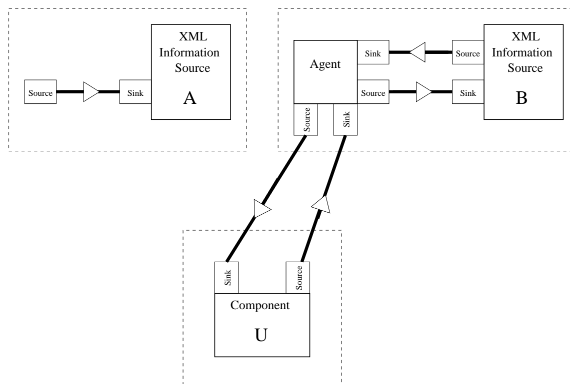


Figure 2.5: Moving to Another Location.

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

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

As explained in Section 2.2.2, 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.

2.4 A Set of Channel Types

We can create any channel type that we want, as long as we obey the basic properties given in Section 2.2; namely channels with two ends, and certain operations and features as defined in that section. Our channel-type can be: synchronous, asynchronous or both, it can be lossy, it can generate values, etc. And, we can always create a channel type by taking the behavior of simpler types and composing them together into a new complex one.

In this section, we introduce a representative set of eleven mobile channel types. These are the channel types that we use in the examples, implementations, models and semantics of this thesis. Furthermore, all these channel types are implemented in the MoCha middleware (see Chapter 6).

2.4.1 Coordination and Communication Types

We start with five channel types that have two distinct ends, $\langle$*source*,*sink*$\rangle$. All these types come from and were first defined as mobile channels in *Reo* [Arb02]. Their graphical representation is given in Figure 2.6.

- *Synchronous channel*. The I/O operations on the two ends of this channel are synchronized. A write on the source-end can succeed only when a take operation also atomically succeeds on the sink-end, and vice-versa. A read operation can succeed only when a write operation is being performed on the

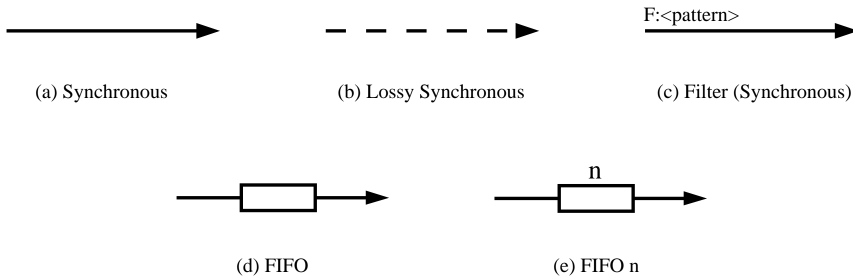


Figure 2.6: Graphical Representation of Channel Types.

source-end, but the component thread/process performing the write operation *still has to wait* until a take operation is performed. Therefore, many read operations can occur (all reading the same value) until a take operation frees the waiting writing thread/process.

The synchronous type already appears in early models based on channels, for example in the π-calculus [Mil99], for it is the most basic type of channel. The synchronization of this channel is done in the so called 'third party synchronization' way: while the components are synchronized due to the synchronous operations that they perform on the channel-ends (see Section 2.2.1) the channel itself internally synchronizes its ends. The result is the synchronization of the three entities: the two components and the channel(-ends).

- *Lossy synchronous channel.* If there is no I/O operation performed on the sink channel-end (take or read) while writing a value to the source-end, the write operation always succeeds but the value gets lost. In the case that there is a take operation being performed on the sink channel-end, the channel behaves like a normal synchronous type. In the case that there is a read operation being performed on the sink channel-end, a (future) write operation on the source-end always succeeds; i.e. the take and read operations behave the same.

  This channel type is useful for coordinating a *writing component* that constantly produces values with a *taking component* that occasionally needs the most recent value that the first component produces. For example, we can coordinate a *thermometer component* with a *display component.* The first component, monitors an outside thermometer and periodically writes values to the source-end of the channel. The second component, occasionally takes values from the sink-end of the channel every time its user requests it. Naturally, the display component is not interested in the 'old' values that were produced by the thermometer component in between takes. Thankfully, these values are lost by the lossy synchronous channel.

- *Filter (synchronous) channel.* The Filter channel behaves like a synchronous type. However, the filter channel type has a user-defined pattern regarding the data that goes through it; values that do not match the channel's pattern are filtered out (lost). Write operations where the value is filtered out of

the channel have no influence on, nor are they influenced by, take or read operations that are performed on the same channel.

We can use this channel type for components that expect a certain pattern of values from the sink-end.

- *Asynchronous unbounded FIFO channel.* The I/O operations performed on the two channel-ends succeed asynchronously. Values written into the source channel-end are stored in the channel in a FIFO (First In, First Out) buffer until taken from the sink-end.

  This channel type, besides offering buffered communication, outputs the values in the same order as they where written into it.

- *Asynchronous bounded FIFO (FIFO n) channel.* This channel behaves in the same way as the unbounded FIFO one, except that is has a capacity of n elements. If the channel is full a write operation has to wait until an element is taken out of the channel first.

  This channel type is useful for slowing down fast writing components that would otherwise insert an enormous amount of values into the channel.

2.4.2 Coordination Only Types

The following six channel types are meant for situations where we are not interested in communication (data-transfer between components) but only in coordinating the components using these channel. All of the types have two ends of the same type; ⟨*source*, *source*⟩ or ⟨*sink*, *sink*⟩. These channel types are useful in cases where we cannot modify the components of a system, in particular, their interaction pattern with the environment. With these channel types we are, nevertheless, able to coordinate these components in an exogenous way. The graphical representation of these channel types is given in Figure 2.7. Except for the spout and the drain channel, all these types come from and were first defined as mobile channels in *Reo* [Arb02].

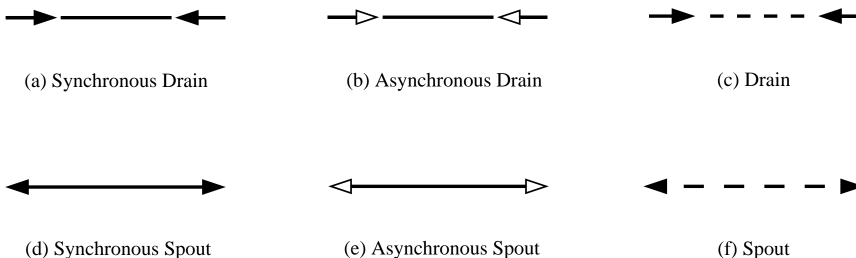


Figure 2.7: Graphical Representation of Channel Types.

Two Source-ends Channels

The following three channels have two source-ends. The values written into these channels are lost.

- *Synchronous drain channel.* The I/O operations performed on the two ends are synchronized (i.e. succeed atomically). So a write operation succeeds only when there is also a write operation being performed on the other channel-end as well.

- *Asynchronous drain channel.* The I/O operations performed on the ends of this channel succeed one at a time exclusively. So the write operations on its two ends *never succeed simultaneously.*

- *Drain channel.* The two source-ends of this channel type are completely independent of each other; i.e. write operations on its ends do not affect each other. The difference between this channel type and the asynchronous drain channel is that with this type the two writes can succeed at the same time.

  The rationale for this channel is that, if we are exogenously imposing a synchronization pattern for components by using the two channels above, we must also be able not to do so by using this channel type.

Two Sink-ends Channels

The following three channels have two sink-ends. The channels produce random values for the components to take or read.

- *Synchronous spout channel.* The I/O operations performed on the two ends are synchronized (i.e. succeed atomically). A take operation succeeds only when there is a matching take operation being performed on the other channel-end. Read operations behave the same as take operations, except that a new value is made only after a take operation.

- *Asynchronous spout channel.* The I/O operations performed on the ends of this channel succeed one at a time exclusively. So the take operations on its two ends never succeed simultaneously. Read operations behave the same as take operations, except that a new value is made only after a take operation.

- *Spout channel.* The two sink-ends of this channel type are completely independent of each other; i.e. take and read operations on its ends do not affect each other. The difference between this channel type and the Asynchronous spout channel is that with this type the two operations can succeed at the same time.

  The rationale for this channel is the same as for the drain channel.

2.4.3 Other Types

The channel type set we presented is a small one. As we mentioned in the beginning of this section, there is no limit on the channel types that can be defined. We give a few examples of other channel types. We didn't include them into our set, for the channels that we choose are already enough to form a representative set that we can use in this thesis.

- *Asynchronous FIFO Filter.* This type is a mix of the (synchronous) Filter channel and the FIFO channel type. It behaves as a FIFO channel with the add-on that values that do not match the channel's pattern are filtered out (lost).
- *Asynchronous shift-lossy FIFO n channel.* This is a bounded FIFO channel with capacity n. If the channel is full a write operation triggers the channel to delete the oldest value to make room for the new one; i.e. the values in the channel 'shift' toward the sink-end of the channel.
- *Asynchronous Bag channel.* Unlike with the FIFO channel, this channel type does not follow any ordering when outputting the values that it contains.
- *Synchronous FIFO channel.* This channel type combines the behavior of the synchronous channel type with the FIFO one. If there is a simultaneous write and take operation and the internal buffer is empty, then the channel behaves like a synchronous one and bypasses its internal buffer. Otherwise, the channel behaves like a FIFO one.
- *Synchronous PlusTwoInteger channel.* This channel reacts on the data that it receives. The channel adds two to every integer value that it receives. Any other data type goes through the channel unchanged.
- *Chameleon n channel.* This channel changes its type after every n values that successfully go through it. For example, it alternates between a FIFO and a Bag channel type.

2.5 Discussion

The careful reader could remark that according to the definition of exogenous and endogenous coordination (see Section 1.1) there is no difference between the MoCha and Linda operations, in the sense that the coordination primitives of both frameworks seem to be "inside" the module itself. Therefore, the reader could object to the classification of MoCha as an exogenous coordination framework, while we classify Linda as an endogenous one. However, despite the fact that both the Linda and MoCha operations are available within a component, the classification is correct. In the Linda framework the components know the (fixed) behavior of the Linda tuple space. They also know what the influence of each Linda coordination primitive has on this tuple space. Thus, by choosing the type and order of execution of the Linda operations that each component performs, the components themselves can (in principle) determine how they are coordinated in the system. Therefore, we regard the Linda coordination primitives as being "inside" the module that they coordinate. In contrast, in the MoCha framework the components don't know anything about the behavior of the mobile channel(s) that they are using. Moreover, there is not just one fixed mobile channel coordination behavior but there are many mobile channel types (each with its own particular coordination behavior). Thus, the components are not able to determine how they are coordinated in the system; coordination is done and specified from the "outside". Therefore, we regard the MoCha coordination primitives as being "outside" the module that they coordinate.

As we mentioned in the beginning, MoCha uses the same notion of mobile channels as Reo. Moreover, the mobile channel types that we use in this thesis (see Section 2.4) are a subset of the ones defined in Reo. Nevertheless, MoCha and Reo are two different models (or coordination frameworks). In our composition chapter (Chapter 9) we explain the (main) differences and similarities between Reo and MoCha.

Part II

Semantics

Chapter 3

Semantics without Mobility

We have divided the semantics for mobile channels in two chapters. In this first chapter, we give a semantics to our channels without considering mobility. For this purpose we use Petri Nets. In Chapter 4, we give a semantics that includes channel mobility. In this chapter, we give a Petri Net for each mobile channel type. This will help us understand the channel's (static) basic behavior. We also discuss how to compose channels together with components, so that we can make Petri Net models of systems whose components communicate and are coordinated by using mobile channels.

3.1 Introduction

A well-known graphically and mathematically founded formalism for the concurrent behavior of systems is *Petri Nets* [Pet96]. Petri Nets, named after their creator Petri, offer well-defined semantics with a clear theoretical foundation [RR98]. This theory includes extensive analysis and simulation possibilities for the Petri Net models. The most common analysis are *causality*, *concurrency*, *conflicts*, *confusions*, *deadlocks*, and *equivalence*.

The Petri Nets formalism is widely used in many different application areas, both in the academic world and in industry. For example, in [GV02] we can read how to use Petri Nets in system engineering. Other examples of application are: modeling of object-oriented systems [Lak01], modeling of Web Service composition [HB03], modeling of business process management [ADO99, LO03], modeling of digital circuits [YK98], and modeling of distributed algorithms [Rei98]. Furthermore, the Petri Nets formalism is well supported by a huge number of commercial and university tools for design, simulation, and analysis of its models. For an extensive list of these tools see the state-of-the-art work in [Sto98, BBBKS00, PNW05].

In Chapter 4, we give the full semantics for mobile channels. However, due to the complexity of this semantics it is hard to understand the internal behavior of channels. Petri Nets have an intuitively appealing graphical form of presentation that facilitates the understanding of both information and control flow within the same formalism. Therefore, this formalism is a good choice for giving the first semantics to mobile channels in this thesis. This semantics does not model channel

mobility. However, the internal behavior of channels specified using Petri Nets are easy to understand. The Petri Net based semantics that we provide in this chapter are helpful to understand the semantics we give in Chapter 4; it is also interesting to compare the two semantics since the first concentrates on *concurrency* and the second on *process interaction* (π-calculus [Mil99]).

Although not the main purpose of this chapter, Petri Nets are often used as a modeling language. Since we provide a Petri Net for each mobile channel type (in this chapter) we can model systems whose components are already specified as Petri Nets and use channels for the interactions between them. We then automatically get all the advantages we discussed above: extensive theoretical support, easy usage, model analysis, simulation of the models, immediate application in different areas, and extensive tool support. Naturally, we are limited to those systems where there is no channel mobility; thus we model the interactions and the exogenous coordination of static systems.

In Section 3.2 we give a short introduction to Petri Nets, where we restrict ourselves to the theory that is needed in order to understand this chapter. In Section 3.3, we give Petri Net semantics to mobile channels, and show how to compose channels with components to obtain Petri Net models of systems. In Section 3.4, we briefly discuss analysis and simulation of these models. We conclude in Section 3.5 with a discussion.

3.2 A Short Introduction to Petri Nets

Petri Net(s), PN for short, is actually a generic name for a whole class of net-based models which can be divided into three main layers [RE98]. The first layer is the most fundamental and is especially well suited for a thorough investigation of foundational issues of concurrent systems. The basic model here is that of *Elementary Net Systems* [Roz86], or *EN systems*. The second layer is an "intermediate" model where one folds some repetitive features of EN systems in order to get more compact representations. The basic model here is *Place/Transition Systems* [DR98], or *P/T systems*. Finally, the third layer is that of high-level nets, where one uses essentially algebraic and logical tools to yield "compact nets" that are suited for real-life applications. *Predicate/Transition Nets* [Gen87] and *Colored Petri Nets* [Jen97a] are the best known high-level models.

Any PN of the three layers above is suitable to model a system. The difference between the low-level and the high-level PN is best described as the difference between writing a program in a high-level language as opposed to writing it in a low-level one [Jen97b]. Furthermore, any PN of any layer can be transformed/translated into a PN of another layer [Eng04]. Going from a low-level PN to a higher level is trivial since each level has the same properties as its immediate lower level plus other added features. Usually, this means that we get the *same* PN model; with the *same* we mean that, we get an equivalent net with the same structure. Going from a high-level PN to a lower-level one is more difficult. For this purpose a technique that is called *unfolding* [Eng04] is used. For example, the unfolding of Colored Petri Nets to P/T systems is discussed in [Jen97b]. The unfolding of P/T systems to EN systems is discussed in [Eng91].

For the simple channel semantics without mobility that we want to give in this

chapter, it is sufficient to use the EN systems. This kind of PN offers clear non-changeable semantic rules and constructs that are easy to understand and follow (unlike, for example, colored Petri Nets). The theory of EN systems are simpler than the theory of any PN in the other layers. Furthermore, the theoretical work available for EN systems is quite extensive and there is more tool support available for then than for high-level PN.

3.2.1 Elementary Net Systems

We give a short introduction to *EN systems.* We restrict ourselves to the definitions that we need in this chapter. For an extensive introduction that also covers several properties of EN systems, equivalences, and EN analysis we refer to the tutorial given in [RE98]. A net is the most basic definition of all PN:

Definition 3.2.1 *A* net *is a triple $N = (P, T, F)$ where*
(1)*P and T are finite sets with $P \cap T = \emptyset$,*
(2)*$F \subseteq (P \times T) \cup (T \times P)$,*
(3) *for every $t \in T$ there exist $p, q \in P$ such that $(p, t), (t, q) \in F$, and*
(4) *for every $t \in T$ and $p, q \in P$, if $(p, t), (t, q) \in F$, then $p \neq q$.*

The elements of P are called *places*, the elements of T are called *transitions*, elements of $X = P \cup T$ are called *elements* (of N), and F is called the *flow relation* (of N).

Each place $p \in P$ can be viewed as representing a possible local state of a system. At each moment in time a set of local states (places) participate in the global state of the system. Such a set of places is called a *configuration.* Graphically, the places that are part of a configuration are denoted with a token; a small black filled circle.

Definition 3.2.2 *A* configuration *C of a net $N = (P, T, F)$ is a subset of P.*

Thus, a *configuration* C of a net is a subset of P where each place contains a token. We now define an EN system as given in [RE98]:

Definition 3.2.3 *An EN system is a quadruple*
$M = (P, T, F, C_{in})$ where:
(1)*(P, T, F) is a net and*
(2)*$C_{in} \subseteq P$ is the* initial configuration

An example of such an EN system is given in Figure 3.2.

Every transition in an EN system can perform an action called *fire.* This action takes a token from all the input places and places a token to each output place of the transition. This action represents a sequential step of a system. However, for this to happen all the input places of the transition must have a token and its output places must be empty, since a place can have at most only one token at a time.

Definition 3.2.4 *Let $M = (P, T, F, C_{in})$ be an EN system and let $t \in T$.*
(1) $^\bullet t$ are the input places of t, and $t^\bullet$ the output places of t.
(2) Let $C \subseteq P$ be a configuration. Then t has concession *in C (or t can be fired in C) if $^\bullet t \subseteq C$ and $t^\bullet \cap C = \emptyset$, written as t* **con** *C.*
(3) Let C,$D \subseteq P$. Then t fires from *C* to *D* if *t* **con** *C and $D = (C -^\bullet t) \cup t^\bullet$, written as $C[t\rangle D$; this firing of t is also called a sequential step.*

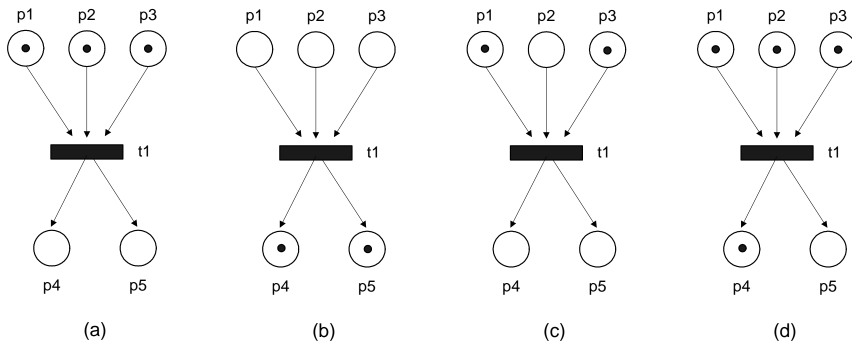


Figure 3.1: Fire and no Fire situations

An example of *firing* is given in Figure 3.1(a). This figure shows an EN system that is ready to fire. Figure 3.1(b) is the result of this firing. The EN systems of Figures 3.1(c) and 3.2(d) cannot fire, the first does not have a token on each input place, and the second already has a token on an output place that prevents the firing from happening.

EN Producer/Consumer Example

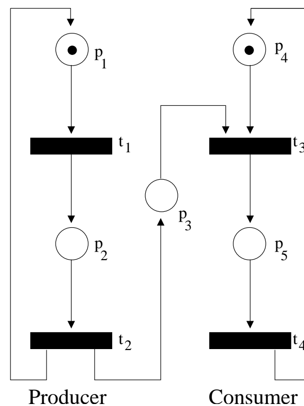


Figure 3.2: Producer/Consumer Example

Figure 3.2 shows an elementary net system that models a producer/consumer system. The processes share a buffer that has the capacity of one element. The producer inserts an element in this buffer when it is empty, and the consumer takes an element when it is available in the same buffer. The elementary net of this example is defined as $P = \{p_1, p_2, p_3, p_4, p_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $F = \{(p_1, t_1), (t_1, p_2), (p_2, t_2),$ $(t_2, p_1), (t_2, p_3), (p_3, t_3), (p_4, t_3), (t_3, p_5), (p_5, t_4), (t_4, p_4)\}$, and $C_{in} = \{p_1, p_4\}$.

It is easy to see that the first fire action must be $C_{in}[t_1\rangle\{p_2, p_4\}$, and the next

$\{p_2, p_4\}[t_2\rangle\{p_1, p_3, p_4\}$. At this point the producer has to wait until the consumer takes the element out of the buffer place p_3 before producing any other element. There is now a choice in firing transition t_1, t_3, or both at the same time. If we fire the first we get $\{p_1, p_3, p_4\}[t_1\rangle\{p_2, p_3, p_4\}$. The system now has to fire t_3 next, $\{p_2, p_3, p_4\}[t_3\rangle\{p_2, p_5\}$. This last firing action symbolizes the taking of the element out of buffer p_3 by the consumer. We could indefinitely go on firing since there is no system termination, nor deadlock, in this example.

Another firing sequence could be the following $C_{in}[t_1\rangle\{p_2, p_4\}$ $[t_2\rangle\{p_1, p_3, p_4\}$ $[t_3\rangle\{p_1, p_5\}[t_4\rangle C_{in}$, however, other sequences are possible as well. This is due to the non-deterministic choice that the system has, it can choose the set of transitions it wishes to fire from a particular configuration. If we maximize the number of transitions that can happen at the same time we get the following firing sequence: $C_{in}[t_1\rangle\{p_2, p_4\}[t_2\rangle\{p_1, p_3, p_4\}$ $[\{t_1, t_3\}\rangle\{p_2, p_5\}$ $[\{t_2, t_4\}\rangle\{p_1, p_3, p_4\}$, at this point the sequence repeats itself.

(Sequential) Configuration Graph

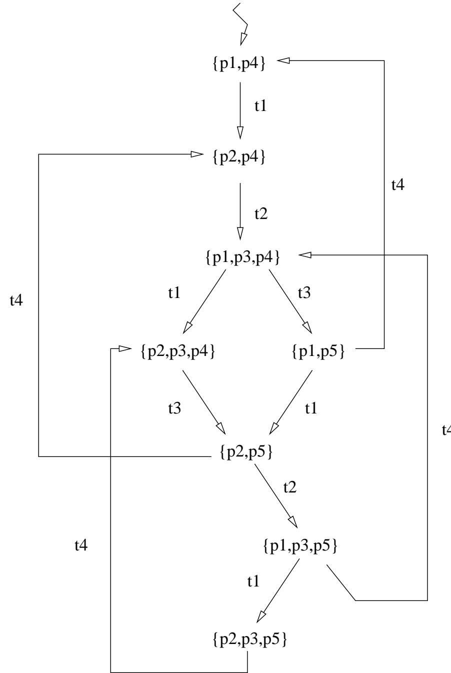


Figure 3.3: Sequential Configuration Graph

When analyzing the behavior of an EN system it is often useful to construct a *sequential configuration graph* (SCG) as defined in [RE98]. This graph represents all possible firing sequences. In Figure 3.3 we give the SCG of our producer/consumer EN system example. The nodes of the graph represent possible *configurations*, and the arrows represent the firing of a particular *transition*. This transition is given as a

label of the arrow. The initial configuration is $\{p_1, p_4\}$, therefore, this configuration constitutes the root node of the graph.

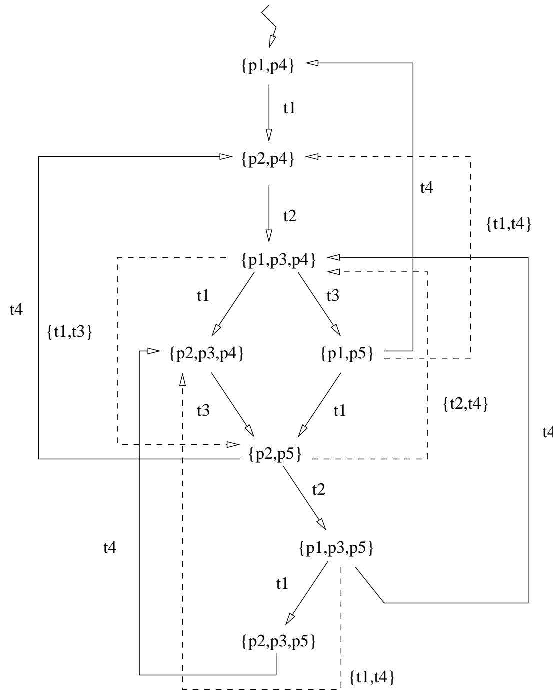


Figure 3.4: Configuration Graph

By analyzing the SCG we can determine the transitions that can concurrently fire in an EN system. If we add these concurrent firing of transitions steps to the SCG, we then get the *configuration graph* (see [RE98]). Figure 3.4 shows the configuration graph of our producer/consumer EN system example. We took the SCG of Figure 3.3 and added the concurrent steps as dashed arrows to it. For example, we can get from configuration $\{p_1, p_3, p_5\}$ to configuration $\{p_2, p_3, p_4\}$ by first firing transition t_1 and then t_4, or vice-versa. Therefore, we added the concurrent step $\{p_1, p_3, p_5\}[\{t_1, t_4\}\rangle\{p_2, p_3, p_4\}$ to the graph.

For the precise definition of the (sequential) configuration graph, the theory and the methods for finding concurrent steps we refer to the tutorial given in [RE98]. For the purposes of this chapter, it is enough to know that the sequential configuration graph represents the sequential steps of an EN system, and, that the configuration graph represents both the sequential and the concurrent steps.

3.3 PN Semantics for Mobile Channels and Components

In this section we give PN semantics to the channels we defined in Section 2.4. This semantics models the main I/O operations *write* and *take*. Since we are not modeling mobility we omit the topology operations: *move*, *connect*, and *disconnect*. Besides mobile channels we also characterize the minimal interaction behavior that components need to have to be able to use our channels. We give a PN for this behavior. For the purpose of being able to model (static) systems where components use mobile channels, we show how to compose the PN of channels and components.

We first give the general interface of the mobile channel PN specifications. Next, we give the PN of the components that use our channels. We, then specify a PN composition function σ. Finally, we give an EN system for each channel type.

3.3.1 Mobile Channel Interface

From the point of view of the components of a system, the mobile channel EN systems that we present in this section have all the *same interface*. We give this interface in Figure 3.5. Each channel EN system has an internal part that is determined by it's type, and an interface that is common to all channel types consisting of four *interface places*, two for each channel-end. We graphically denote these places by marking an extra symbol I on the outside of the circle. These interface places are part of a protocol to ensure that all write and take operations are *blocking*; i.e. an active entity performing such an operation blocks until the operation succeeds and terminates.

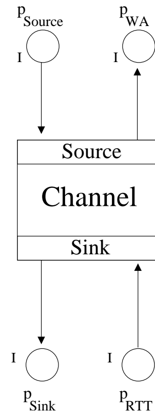


Figure 3.5: Interface of a Mobile Channel PN

The places p_{Source} and p_{WA} constitute the interface of the source channel-end. A component that wants to perform a *write operation* on this end, puts a token into place p_{Source}. This token represents the fact that a data element is available but has not yet been accepted by the channel. In other words, the write operation is pending between the component and the source channel-end. When the token is

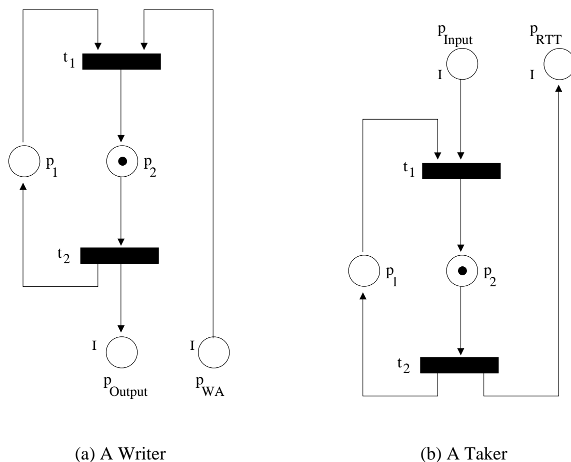


Figure 3.6: A Writer and a Taker

removed from this place by the channel, it means that the channel is processing the write operation. Upon completion of the operation, the channel puts a token in the interface place p_{WA} as a write acknowledgment.

The places p_{Sink} and p_{RTT} constitute the interface of the sink channel-end. A component that wants to perform a *take operation* on this end, puts a token in place p_{RTT} (Ready To Take). This token reveals the desire and willingness of the component to take a data element from the channel. The channel learns that there is a component waiting (and wanting) to take an element only when the token in p_{RTT} successfully "enters" the channel due to a fire action. The channel completes the take operation by putting a token in the p_{Sink} interface place. The component can then take the token from this place.

We don't explicitly model a source- and a sink-end in the mobile channel EN systems. A channel-end is implicitly modeled by its two interface places and the internal transitions where these places are used as either input or output. Observe that the semantics of the write and take operations are analogous with the ones defined in Section 4.3.

3.3.2 Component Interaction

The components of a system interact with mobile channels thought the interface that we defined in Figure 3.5. From our point of view, a component consists of one or more active entities (threads or processes) that perform write and take operations. In Figure 3.6 we give the EN systems of two single entity components. They represent the minimal behavior that components need to implement regarding the write and take operations toward channels; i.e. they implement their side of the blocking protocol as described in Section 3.3.1.

Figure 3.6(a) shows the PN of a simple writer. This net has two interface places that are meant for composition with channels: $\{p_{Output}, p_{WA}\}$. The initial configuration of the net is $\{p_2\}$. The writer starts the write operation by executing

$\{p_2\}[t_2\rangle\{p_{Output}, p_1\}$. At this point it is blocked for it must wait until it receives a write acknowledgment; i.e. a token is placed in p_{WA}. If the writer is interacting with a source channel-end, at the time that it receives the acknowledgment the token in place p_{Output} is already gone. Therefore, we end up with the configuration $\{p_1, p_{WA}\}$. The writer completes the operation by performing the sequential step $\{p_1, p_{WA}\}[t_1\rangle C_{in}$. At this point, it may start writing again.

Figure 3.6(b) shows the PN of a simple taker. This net also has two interface places that are meant for composition with channels: $\{p_{Input}, p_{RTT}\}$. The initial configuration of this net is $\{p_2\}$. The taker starts the take operation by executing $\{p_2\}[t_2\rangle\{P_{P1}, p_{RTT}\}$. At some point in time the channel it is interacting with takes the token from p_{RTT}, and later puts a token back in place p_{Input}. The resulting configuration is $\{p_1, p_{Input}\}$. The taker completes the operation by performing $\{p_1, p_{Input}\}[t_1\rangle C_{in}$. At this point, it may start taking again.

3.3.3 PN Composition of Components and Channels

We have introduced the interface of channels and the interface of components toward channels. We now show how to compose them in order to obtain one big PN model of a complete system consisting of channels and components. There are several construction strategies possible. Our major requirement is for such strategy is, that one should be able to distinguish the individual components and channels in the composed system, and it must be easy to decompose and rearrange the system; i.e. updating and replacing components and channels without having to change the rest of the system. Therefore, for example, we cannot do composition and optimize the resulting PN for it may not be possible to decompose after several composition steps anymore. Our strategy then is to do composition on the interface places. With this strategy, we don't have to change the internals of components and channels, it is easy to recognize the individual parts, and decomposition is also clear and easy to do. For this purpose we define a composition function that merges interface places:

Definition 3.3.1 *Let X_1 and X_2 be two disjoint EN systems (where $X_1 = (P_{X_1}, T_{X_1}, F_{X_1}, C_{in-X_1})$ and $X_2 = (P_{X_2}, T_{X_2}, F_{X_2}, C_{in-X_2})$). Let P_1 and P_2 be two finite sets of (interface) places, with $P_1 \subseteq P_{X_1}$, $P_2 \subseteq P_{X_2}$ and $|P_1| = |P_2|$. Typical elements of these sets are $a \in P_1$ and $b \in P_2$. We define $\sigma(X_1, P_1, X_2, P_2)$ to be the composed EN system Y (where $Y = (P_Y, T_Y, F_Y, C_{in-Y})$). We construct Y as follow:*
(1) $P_Y = (P_{X_1} \setminus P_1) \cup (P_{X_2} \setminus P_2) \cup P_{new}$, where
$P_{new} = \{\langle a_i, b_i\rangle |\ a_i \in P_1 \wedge b_i \in P_2\}$,
with i as an index from 1 to $|P_1| = |P_2|$, and $|P_1| = |P_2| = |P_{new}|$.
P_{new} is a (new) finite set of places, with typical element $c \in P_{new}$.
(2) $T_Y = T_{X_1} \cup T_{X_2}$.
(3) $F_Y = (F_{X_1} \cup F_{X_2} \cup F_I) \setminus F_{Rem}$,
where $F_I \subset F_Y$ and $F_{Rem} \subset F_{X_1} \cup F_{X_2}$ and are constructed as follow:
$\forall(i \in 1\, to\, |P_1|)$ if $(a_i, t) \in F_{X_1}$ then $(a_i, t) \in F_{Rem} \wedge (c_i, t) \in F_I$,
and if $(t, a_i) \in F_{X_1}$ then $(t, a_i) \in F_{Rem} \wedge (t, c_i) \in F_I$,
$\forall(i \in 1\, to\, |P_2|)$ if $(b_i, t) \in F_{X_2}$ then $(b_i, t) \in F_{Rem} \wedge (c_i, t) \in F_I$,
and if $(t, b_i) \in F_{X_2}$ then $(t, b_i) \in F_{Rem} \wedge (t, c_i) \in F_I$.
(4) $C_{in-Y} = (C_{in-X_1} \setminus P_1) \cup (C_{in-X_2} \setminus P_2) \cup C_{in-new}$, where
$(\forall i \in 1\, to\, |P_{new}|)$if $a_i \in P_1 \wedge a_i \in C_{in-X_1}$ then $c_i \in C_{in-new}$,

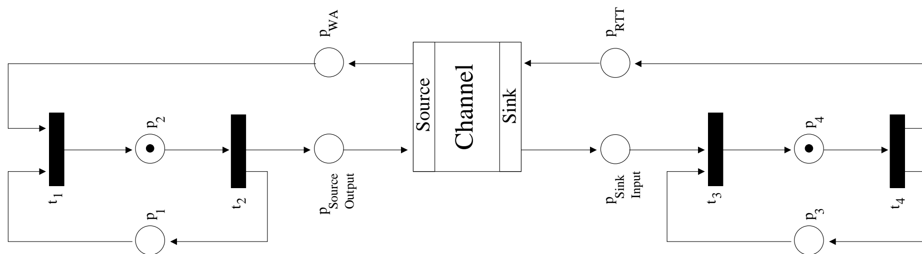


Figure 3.7: Composing a Writer and a Taker with Mobile Channels

and if $b_i \in P_2 \wedge b_i \in C_{in-X_2}$ *then* $c_i \in C_{in-new}$.

The function σ takes two disjoint EN Systems X_1 and X_2 as parameters. The function also takes the sets of places P_1 and P_2 that correspond to the interface places of, respectively, X_1 and X_2 that we want to compose (or "merge"). The result of the function is a new EN System Y that is constructed as follow: (1) Each place of X_1 and X_2 is present in Y, except for the interface places of P_1 and P_2. Each pair (a, b), whose places are related to each other for having the same index number, is substituted by a new place c ($\in P_{new}$) that is inserted in Y. (2) The composition is done on interface places so the transitions of Y are simply the union of the ones in X_1 and X_2. (3) Every flow relation present in either X_1 or X_2 is also present in Y. The flow relations that involve the interface places in P_1 and P_2, represented in flow relation F_{Rem}, are changed to be involved in the new added places of P_{new}, represented in flow relation F_I. (4) The C_{in} of Y is the union of the ones of X_1 and X_2. However, the places of P_1 and P_2 may also be present at the initial configurations of these two last EN systems. Since these places do not exist anymore in Y, we add their corresponding new places from P_{new} into the initial configuration.

We now can compose components and channels using our function σ. For example, we obtain the EN system *Comp* of figure 3.7, by applying the σ function to the writer and taker components and the channel interface of Figure 3.5: $Comp = \sigma(Taker, \{p_{Input}, p_{RTT}\}, Tmp, \{p_{Sink}, p_{RTT}\})$, $Tmp = \sigma(Writer, \{p_{Output}, p_{WA}\}, Channel, \{p_{Source}, p_{WA}\}$. In Section 3.4, we give an example using a concrete channel.

3.3.4 A Set of EN and P/T-net Mobile Channels Systems

We give a PN (EN system) for the majority of the channel types we discussed in Section 2.4. We omit the PN for the filter channel type because the tokens of EN systems have no data, and thus the behavior of the filter EN system becomes equal to the one of the synchronous EN system. We also omit the PN for the *asynchronous unbounded FIFO* channel type. The structure of this PN is equal to the one of the *asynchronous FIFO n* channel except that the first is a PN with an infinite number of places and the second a PN with a finite number of places. The reader may find it interesting to compare the semantics of these mobile channel EN systems with the semantics that we give in Section 4.4.1.

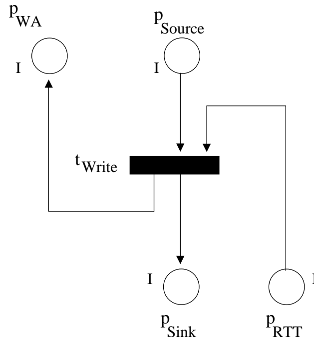


Figure 3.8: The Synchronous Channel EN System

The Synchronous Channel Type

The I/O operations on both ends of a *synchronous channel* are synchronized. Figure 3.8 shows the EN system of this channel. The internals of this channel type is just a transition t_{Write} that synchronizes the four interface places as defined in Section 3.3.1. The places p_{Source} and p_{RTT} are input places of transition t_{Write}. Therefore, only when both the writing and the taking components have each inserted a token in these places, the I/O operations atomically succeeds (at the same time). We give the sequential firing step: $\{p_{Source}, p_{RTT}\}[t_{Write}\rangle\{p_{Sink}, p_{WA}\}$. At the end a token is inserted in the places p_{Sink} and p_{WA}. This indicates the completion of the I/O operations.

The Lossy Synchronous Channel Type

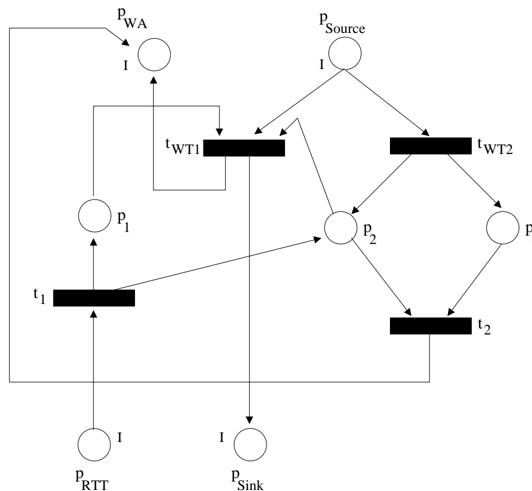


Figure 3.9: The Lossy Synchronous Channel EN System

With the *lossy synchronous* channel type, if there is no I/O operation performed on the sink channel-end while writing a value to the source-end, the write operation always succeeds but the value gets lost. In all other cases, the channel behaves like a normal synchronous type.

Figure 3.9 gives the EN system for this channel type. There are two paths for a successful write operation in this PN. One is determined by the t_{WT1} transition and exhibits the behavior of a synchronous channel. The other is determined by the t_{WT2} transition and exhibits the lossy behavior of this channel type. The choice between the first or the second path depends on whether, from the point of view of the channel, there is a component waiting to take a value or not. A component indicates its willingness to take a value by putting a token in place p_{RTT}. The presence of this token is not detectable by the channel. Only after transition t_1 fires the channel knows that a component is ready to accept a value: $\{p_{RTT}\}[t_1\rangle\{p_1, p_2\}$.

In configuration $\{p_{Source}, p_1, p_2\}$ there is a component trying to write (due to the token in place p_{Source}) and a component waiting to take (due to the tokens in places p_1 and p_2). Transition t_{WT1} can fire due to the tokens in places p_1 and p_2. At this time, transition t_{WT2} is blocked because of the token in place p_2. Therefore, the written value synchronously flows from p_{Source} to p_{Sink}: $\{p_{Source}, p_1, p_2\}[t_{WT1}\rangle\{p_{Sink}, p_{WA}\}$.

In configuration $\{p_{Source}\}$ there is a component trying to write but no component to take. This time transition t_{WT1} cannot fire due to the lack of a token in place p_1. Transition t_{WT2} is free to fire, and when it does the value gets lost while the write operation succeeds. Observe that, there is no need to model an explicit garbage to delete the written token since the firing of transition t_2 already takes care of this. We give the sequential steps of this lossy path: $\{p_{Source}\}[t_{WT2}\rangle\{p_2, p_3\}[t_2\rangle\{p_{WA}\}$.

The EN system of 3.9 gives an approximation of the actual behavior of a lossy synchronous channel type. Ideally we want the synchronous path to be taken every time a component puts a token in place p_{RTT}, i.e. every time a component indicates its willingness to take, a (future) written value must not get lost. This is easy to model in higher level PNs (for example, by using constraints). However, up to our knowledge there is no EN system with finite places that can model this behavior (EN systems with infinite places can, but we want to avoid them). Nevertheless, the EN system we give for the lossy synchronous channel type is acceptable for the purposes of this chapter.

The FIFO n Channel Type

The I/O operations that are performed on the ends of an *asynchronous FIFO n* channel succeed asynchronously. Values written into the source channel-end are internally stored in a buffer until taken from the sink-end. Figure 3.10(a) shows the EN system of a FIFO 1 channel. As one may expect, the internal buffer of capacity 1 is modeled by place p_{buf}. We write a value into the channel by performing the sequential step $\{p_{Source}\}[t_{Write}\rangle\{p_{buf}, p_{WA}\}$, and we take a value by performing the step $\{p_{buf}, p_{RTT}\}[t_{Take}\rangle\{p_{Sink}\}$. In Figure 3.10(b) we give the EN system for the FIFO 2 channel. Naturally, it contains two buffer places. Figure 3.10(c) gives the general scheme for a FIFO EN system channel with the buffer capacity of n. Observe, that if n is *infinite* (unbounded FIFO channel) we get an EN system with infinite places.

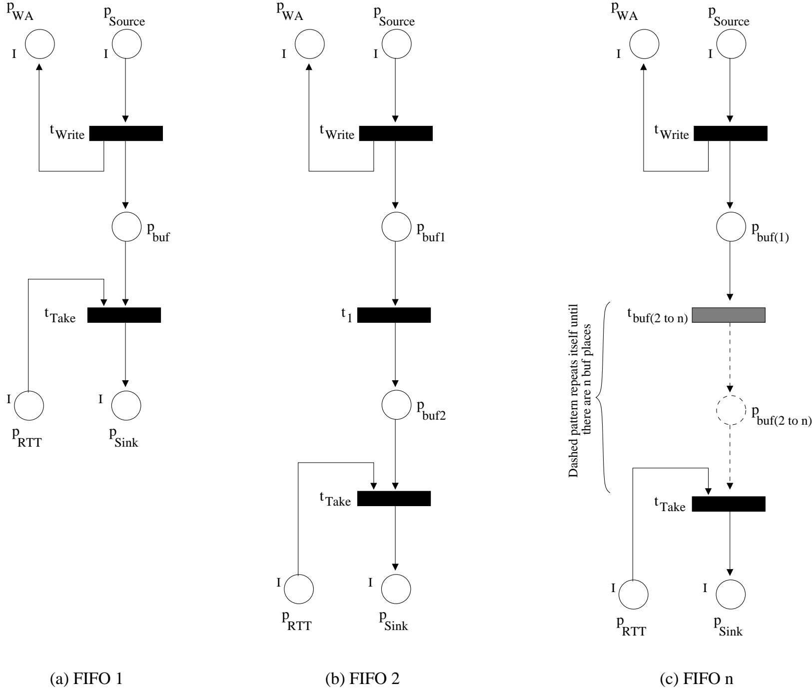


Figure 3.10: The FIFO 1, 2 and n Channel EN Systems

The Synchronous Drain Channel Type

This channel type has two source-ends where the write operations performed on them are synchronized. Figure 3.11(a) gives the EN system of the *synchronous drain* channel. There are no surprises here. The net looks like the one of the synchronous channel type, except that we now have two source places, two "write acknowledgment" (WA) places, and no sink place nor "ready to take" (RTT) place. The only sequential step that can happen with this PN is: $\{p_{_{Source1}}, p_{_{Source2}}\}[t_{Write}\rangle\{p_{WA1}, p_{WA2}\}$.

The Asynchronous Drain Channel Type

The write operations performed on the two source ends of the *asynchronous drain* channel type never succeed atomically. This is reflected in the EN system we give in Figure 3.11(b). Place p_3 makes sure that either transition t_{Write1} or transition t_{Write2} fires, but not both at the same time.

Let's assume that there are two simultaneous writes available. The net configuration for this situation is $\{p_{_{Source1}}, p_{_{Source2}}\}$. We can choose to perform the write operation on the left source-end first: $\{p_{_{Source1}}, p_{_{Source2}}\}[t_{Write1}\rangle\{p_{_{Source2}}, p_1, p_3\}$. In this new configuration transition t_{Write2} is blocked by the token in place p_3 (no write by the right source-end can happen). However, transition t_1 can fire to complete the write of the left source-end: $\{p_{_{Source2}}, p_1, p_3\}[t_1\rangle\{p_{_{Source2}}, p_{WA1}\}$. At this point the other write can start. Alternatively, we can perform the write operation on the

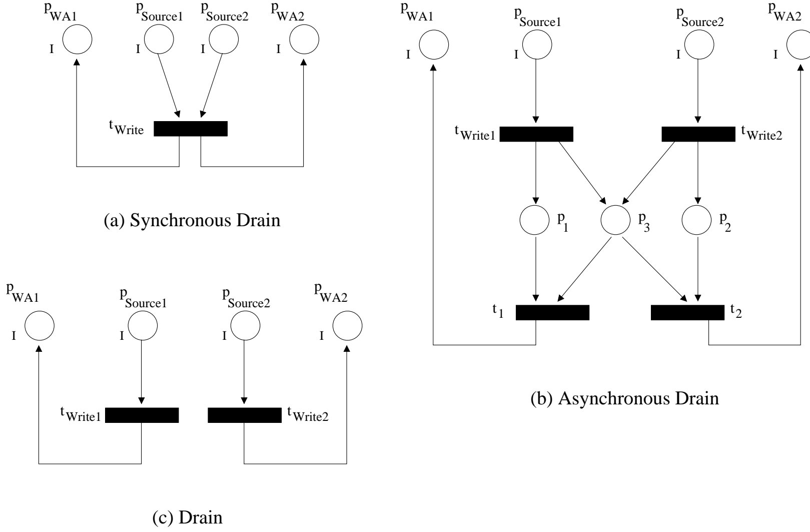


Figure 3.11: The Synchronous-, Asynchronous-, and Drain Channel EN System

right source-end first: $\{p_{_{Source1}}, p_{_{Source2}}\}[t_{Write2}\rangle\{p_{_{Source1}}, p_2, p_3\}$. In this configuration transition t_{Write1} is blocked by the same place p_3. Transition t_2 fires to complete the write action: $\{p_{_{Source1}}, p_2, p_3\}[t_2\rangle\{p_{_{Source1}}, p_{WA2}\}$. At this point the other write can start. We see that we can never fire transitions t_{Write1} and t_{Write2} concurrently.

The Drain Channel Type

The write operations on the two source-ends of the *drain* channel type do not affect each other. We give the EN system in Figure 3.11(c). We see that, indeed, the two transitions, $\{t_{Write1}, t_{Write2}\}$, are completely independent of each other; i.e. they don't share input nor output places. The sequential steps $\{p_{_{Source1}}\}[t_{Write1}\rangle\{p_{WA1}\}$ and $\{p_{_{Source2}}\}[t_{Write2}\rangle\{p_{WA2}\}$ can happen concurrently.

The Synchronous Spout Channel Type

This channel type has two sink ends where the take operations performed on them are synchronized. Figure 3.12(a) gives the EN system of the *synchronous spout* channel type. The net looks like the one of the synchronous channel type, except that we now have two sink places, two pair of "ready to take" and "write acknowledgment" (RTT) places, and no source nor "write acknowledgment" (WA) place. The only sequential step that can happen with this PN is: $\{p_{_{RTT1}}, p_{_{RTT2}}\}[t_{Take}\rangle\{p_{_{Sink1}}, p_{_{Sink2}}\}$.

The Asynchronous Spout Channel Type

The take operations performed on the two sink ends of the *asynchronous spout* channel type never succeed atomically. This is reflected in the EN system we give in

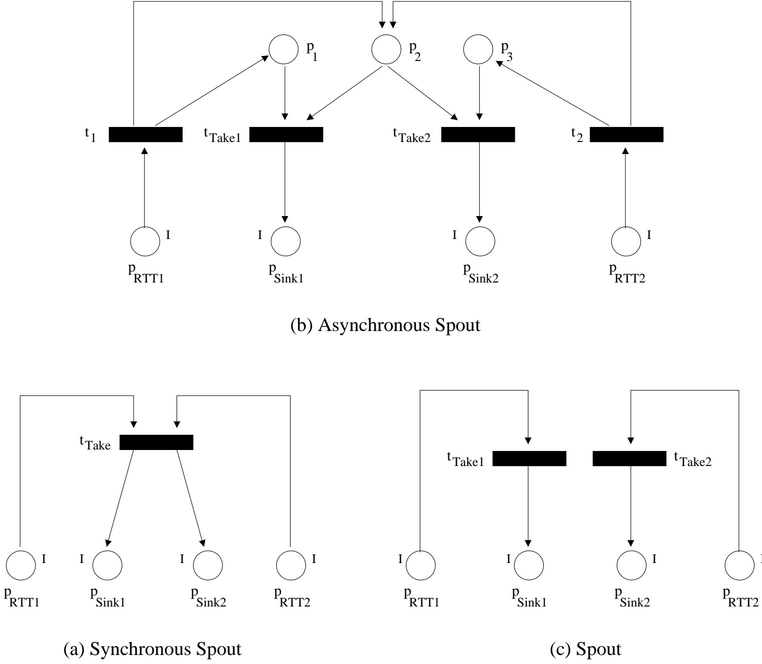


Figure 3.12: The Synchronous-, Asynchronous-, and Spout Channel EN System

Figure 3.12(b). Place p_2 makes sure that either transition t_{Take1} or transition t_{Take2} fires, but not both at the same time.

Let's assume that there are two simultaneous takes available. The net configuration for this situation is $\{p_{RTT1}, p_{RTT2}\}$. We can choose to perform the take operation on the left sink-end first: $\{p_{RTT1}, p_{RTT2}\}[t_1\rangle\{p_{RTT2}, p_1, p_2\}$. In this configuration the token in place p_2 blocks the firing of transition t_2 (no take by the right sink-end can happen). However, transition t_{Take1} can fire to complete the take of the left sink-end: $\{p_{RTT2}, p_1, p_2\}[t_{Take1}\rangle\{p_{RTT2}, p_{Sink1}\}$. At this point the other take can start. Alternatively, we can perform the take operation on the sink-end at the right first: $\{p_{RTT1}, p_{RTT2}\}[t_2\rangle\{p_{RTT1}, p_2, p_3\}$. In this configuration the token in place p_2 blocks the firing of transition t_1. Transition t_{Take2} fires to complete the take action:$\{p_{RTT1}, p_2, p_3\}[t_{Take2}\rangle\{p_{RTT1}, p_{Sink2}\}$. At this point the other take can start. We see that we can never fire transitions t_{Take1} and t_{Take2} concurrently.

The Spout Channel Type

The take operations on the two sink-ends of the *spout* channel type do not affect each other. We give the EN system in Figure 3.12(c). We see that, indeed, the two transitions, $\{t_{Take1}, t_{Take2}\}$, are completely independent of each other; i.e. they don't share input nor output places. The sequential steps $\{p_{RTT1}\}[t_{Take1}\rangle\{p_{Sink1}\}$ and $\{p_{RTT2}\}[t_{Take2}\rangle\{p_{Sink2}\}$ can happen concurrently.

3.4 Analysis and Simulation of PN Models

We now know how to model systems that use our mobile channels by means of PN; we create such a model by composing the PN of the components with the PN of the channels we give in this chapter. In this section we briefly discuss the analysis and simulation of these models. By analyzing and simulating PN models we are able to identify the (concurrent) exogenous coordination behavior of a system that follows from the interaction between the components and the mobile channels.

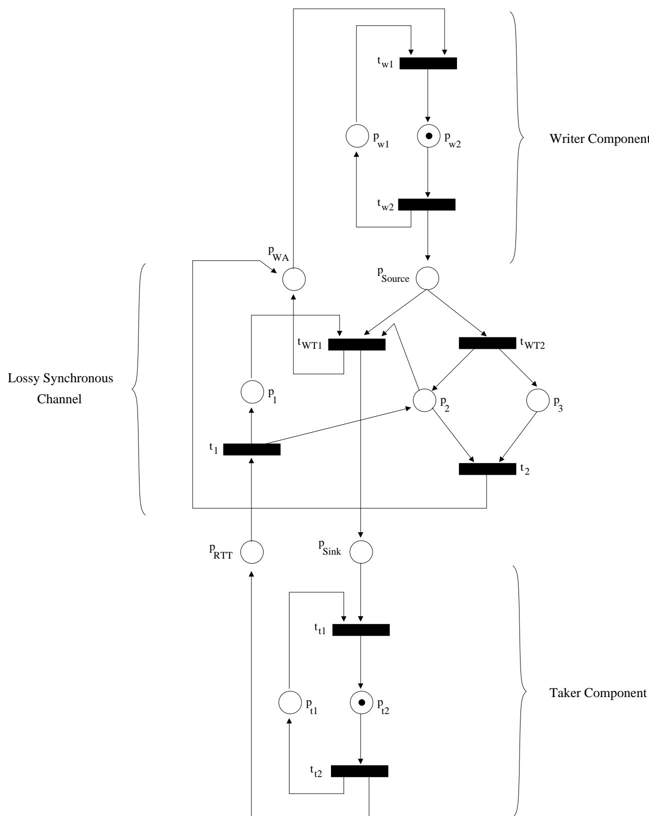


Figure 3.13: An Example of Composition

In Figure 3.13 we give the PN model of a simple system consisting of two components and one lossy synchronous channel. These components are the taker and the writer defined in Section 3.3.2. Bigger systems can be made by using more complex components, for example by using the ones given in Chapter 9. The model of Figure 3.13 is obtained as $\sigma(\mathit{Taker}, \{p_{Input}, p_{RTT}\}, \mathit{Tmp}1, \{p_{Sink}, p_{RTT}\})$, where $\mathit{Tmp}1$ is $\sigma(\mathit{Writer}, \{p_{Output}, p_{WA}\}, \mathit{LossySynchronous}, \{p_{Source}, p_{WA}\}$.

We can *simulate* our example system by playing the "token game" as defined in [RE98]. This game consists of firing transitions, when possible, to get the system from one state into another. By doing this we get all the possible states and all the

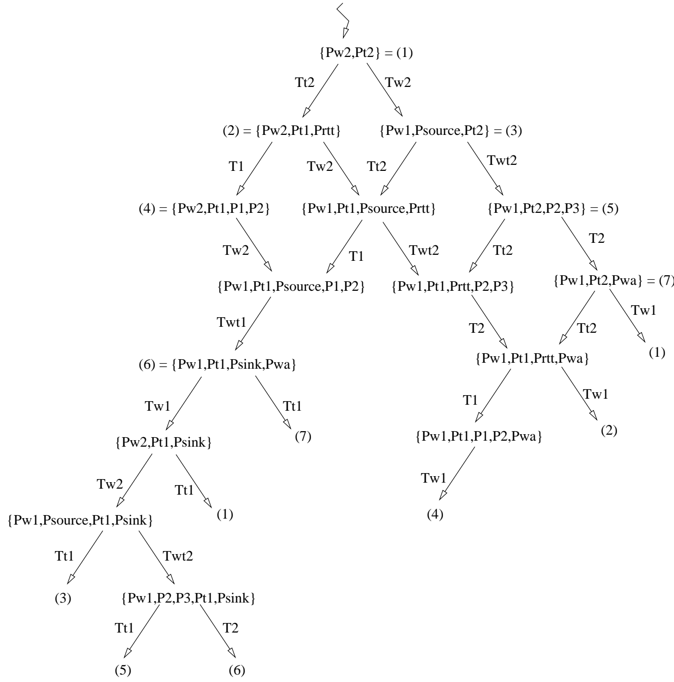


Figure 3.14: The Sequential Configuration Graph of Figure 3.13

possible firing sequences of a system. This information is graphically given in the sequential configuration graph of Figure 3.14.

Besides simulation, we can also *analyze* the models for desired, or undesired, properties and features. For example, by looking at Figure 3.14 we can recognize (among others) the synchronous and the lossy firing path that we gave for the lossy synchronous channel in Section 3.3.4; thus, we can verify the exogenous coordination behavior of this particular channel type and see how it affects the behavior of the components using this channel.

More general, common and well-supported PN analysis that we can use on our models include *causality*, *concurrency*, *conflicts*, *confusions*, *deadlocks*, and *equivalence*. For example, we can analyze the sequential configuration graph (of Figure 3.13) for *concurrent steps*: Basically, every diamond shape in the graph represents such a step. Figure 3.15 shows the configuration graph of our example system, where we added dashed lines to the sequential configuration graph for the concurrent steps. This last graph provides us with even more information about the behavior of the lossy synchronous channel. For example that the first possible concurrent step is $\{p_{w2}, p_{t2}\}[\{Tt2, Tw2\}\rangle\{p_{w1}, p_{t1}, p_{source}, p_{rtt}\}$; we can arrive from configuration $\{p_{w2}, p_{t2}\}$ to configuration $\{p_{w1}, p_{t1}, p_{source}, p_{rtt}\}$, by first firing transition $Tw2$ and then transition $Tt2$, or vice-versa.

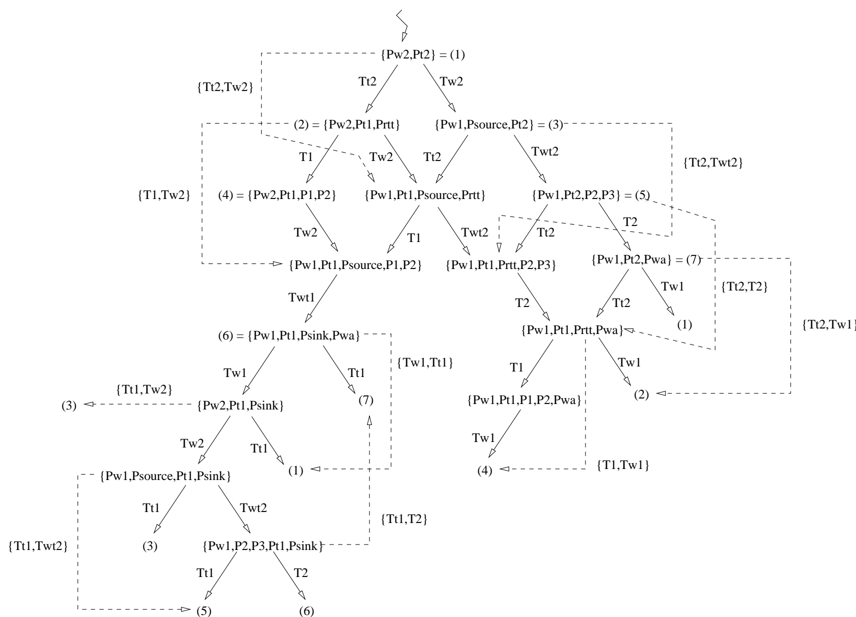


Figure 3.15: The Configuration Graph of Figure 3.13

The system we modeled is quite small and simple. As we have seen, analyzing and simulating this system "by hand" is possible but already not a pleasant task. For example, look at the size of Figure 3.14. A real application consists of many components and many mobile channels. Therefore, modeling such an application quickly results in a big Petri Net model that is not human-tractable anymore. Fortunately, there are many tools available for EN systems. For an extensive list of these tools we refer to the state-of-the-art work in [Sto98, BBBKS00, PNW05]. One such tool we recommend is the *Platform Independent Petri Net Editor* (*PIPE*) tool [BCCGKT05]. The PIPE tool is an open source project, it is *free of charge*, *platform-independent*, offers *simulation* and *analysis* modules, and gives XML support.

3.5 Discussion and Concluding Remarks

In this Chapter we gave semantics to mobile channels by using the PN formalism. This semantics does not model channel mobility, and therefore, all the channel PN we presented in Section 3.3 are static; the ends of these channels don't move. We showed how to achieve PN models from systems whose components communicate through and are coordinated by mobile channels. In Section 3.4, we briefly discussed some of the analysis and simulation possibilities for these models.

The next logical step is to extend the channel semantics of this chapter to support mobility. The question is then whether EN systems are still the right kind of PN

to use. For the purposes of this chapter EN systems are a good choice, but we have seen that the nets of some channels can become quite large. This is the case with the FIFO n channel PN where the higher the capacity the more buffer places the PN has. In the case of the unbounded FIFO channel there are even an infinite number of buffer places. We have also seen that the PN models we produce by composing channels together with components quickly become huge in size. To solve this problem we use tools. However, this size explosion of PN suggests that if we extend each channel PN with a protocol for mobility its semantics will get very complex to follow. Moreover, even small models consisting of few components and channels (like the one we gave in Section 3.4) will quickly become human intractable. A better choice then is to use a higher level PN where we can use constructs such as constraints to obtain much smaller nets. For example, Colored Petri Nets [Jen97a] are much more suitable for modeling channel mobility.

Instead of translating all the channel EN systems of this chapter to Colored Petri Nets, in the next chapter, we are going to use another formalism (or model) that is far more suitable for giving semantics to our mobile channels. This doesn't make the semantics we give in this chapter obsolete. The PN specifications for each channel type are far more easier to follow than the ones we give in the Chapter 4. For example, compare the PN specification of a FIFO n channel type in Section 3.3.4 with the specification of the same channel type in Section 4.4.1. Therefore, the mobile channel semantics of this chapter can be used as a guideline to understand the ones of Chapter 4. Furthermore, by specifying (static) mobile channels in PN we get the following advantages: PN theoretical support, PN model analysis and simulation, immediate application in different areas, and tool support.

Chapter 4

Semantics with Mobility

In this chapter we present MoCha-π, an exogenous coordination calculus that is based on mobile channels. Our calculus is an extension of the well-known π-calculus [Mil99]. The novelty of MoCha-π is that its channels are a special kind of process that allow other processes to communicate with each other and impose exogenous coordination through user defined channel types. Also new, is the fact that in our calculus channels are viewed as resources. Processes must compete with each other in order to gain access to a particular channel. This makes the calculus more in line with existing systems.

4.1 Introduction

We introduce an exogenous coordination calculus that is based on mobile channels. We call this calculus: MoCha-π. With MoCha-π we give semantics to mobile channels. This semantics includes the interactions between components and channels. Since MoCha-π is a process algebra [Fok99], in this chapter we talk about processes instead of components. The difference is that a component consists of one or more process(es). The specific definition of a component is given in Chapter 5.

Besides giving semantics to channels, our calculus is also very suitable as a logical framework for modeling (distributed) systems. With MoCha-π we can specify the communication and coordination aspects of systems. This is useful, for example, for model-checking purposes in the software development phase.

In the next section we briefly discuss the π-calculus and how it relates to MoCha-π. In Section 4.3 we present the MoCha-π calculus. Our calculus provides channels that are more general than the ones of the MoCha framework. Therefore, in Section 4.4, we give a design pattern for specifying channels that are compatible with the framework. Afterward, we show three examples of how to use our calculus in Section 4.5. We conclude with related work in Section 4.6.

4.2 Extending the π-calculus

The π-calculus [Mil99] is a basic mathematical model that focuses on the interaction between processes. It is based on the CCS [Mil80] calculus. The main difference with CCS is that the π-calculus allows dynamical changes in the interconnections between processes (as they interact). The basic, and only, communication primitive of the calculus is the *channel*. Processes write to a channel by means of a *send* action, and take from a channel by means of a *receive* action. The type of the channels in the standard π-calculus is exclusively *synchronous*. Besides being able to send values through a channel, we can also send channels to other processes. The recipient of a channel can, then, use it for further interaction with other parties. This makes the changing of channel connections among processes possible in a system.

It seems natural to use the π-calculus to give semantics to our mobile channels. Here are the major reasons:

- The π-calculus focuses only on process communication and coordination. It abstracts away from the system's computational part.
- The π-calculus doesn't explicitly model the notion of *location*. The location of a process in a system is implicitly determined by the channel connections that it has with other processes. This allows us to model the mobile channels in such a way that we abstract away from the distribution of a system. This idea follows from our desire of processes not having to see a difference between local and inter-local communication. In a real implemented system, the mobile channels take care of the internal distributed communication.
- Processes in the π-calculus already communicate through channels.
- Like with mobile channels, π-calculus channels can be sent through channels themselves.
- The π-calculus is easy, but powerful enough, to use.
- There is extensive theoretical work about the π-calculus available. For example, see [Mil99], [Par01], and [SW01].
- There is tool support for the π-calculus. For example, the *Mobility Workbench* [VM94].

However, there are differences between the MoCha framework and the π-calculus. Mainly:

- MoCha supports several channel types, the π-calculus supports only one type.
- Components/processes in MoCha know and use *channel-ends*, the processes of the π-calculus use *channels* as a whole instead of their ends.
- In MoCha components/processes see channel-ends as resources. Therefore, a process has to *connect* to a particular channel-end first before it can use it. In the π-calculus, any process can use a channel that it knows at any time.

- The mobile channels of MoCha provide exogenous coordination. In the π-calculus there is no built-in support for exogenous coordination.

To cope with these differences we considered doing two things. On the one hand, we can provide π-calculus specifications that implement all the above requirements. On the other hand, we can extend the π-calculus by introducing higher-level constructs to dynamically (i.e. during execution) translate the interaction operations of MoCha into the basic π-calculus ones.

In the first approach, we need to implement all the missing features that we want into the π-calculus. We briefly discuss what needs to be done. We start by implementing a channel-end. The only way to do it in the π-calculus is by defining it as a process. This channel-end then needs to, somehow, be related with the other end of the channel. We can choose between linking the channel-ends via π-calculus synchronous channels, or by creating a process channel that internally relates the two ends. Naturally, for each channel type we need to define new channel-end processes and relate them. For the interactions between the 'normal' processes and the channel-ends we need to implement a well-defined interface. Most likely, interactions with channel-end processes go through π-calculus channels. We must make sure that, somehow, 'normal' processes know these π-calculus channels and that they follow the patterns imposed by the channel-end interface. The most difficult feature to implement is the *connect* and *disconnect* operations. Somehow, channel-ends must keep a list of processes that want to connect to them. Then, they must let a process know that it has exclusive permission to use the end, and must accept I/O operations only from that specific process. Afterward, when the process disconnects they must select another process and notify it.
All of this is implementable in the π-calculus. However, the main reason for us not to follow this approach is that, since these specifications need to be present with every model that we make, the specifications of these models become huge and unreadable. Furthermore, the models get full of details that we are not interested to see. For example, we don't want to see how a channel-end process internally handles the connect operation.

In the second approach, the one we use in this chapter, we create a calculus that provides high-level constructs and definitions for the notions of channel-ends, resources, and mobile channel actions. A mobile channel is still defined as a π-calculus process, but this process is much simpler to define than the one we had to define with the first approach. We don't have to define a separate process channel-end, and the calculus assures a certain interface. We also don't have to worry about the implementation of the connect and disconnect operations. The calculus handles all these operations. We call this calculus: MoCha-π.

4.3 The MoCha-π Calculus

In this section we present the MoCha-π calculus. Our calculus offers high-level interface *write*, *take*, *connect* and *disconnect* operations on channels whose behavior is user-defined. Just like in the MoCha-framework, processes have no direct references to channels but only to channel-ends, and therefore, all interface operations are performed on channel-ends.

We use the π-calculus to implement the I/O interface actions of MoCha-π. Our calculus transforms all *write* and *take* actions into a pattern of traditional π-calculus ones. It does this transformation when a process is *connected* to a particular channel-end and performs an I/O action on it. Therefore, this transformation can be done only dynamically, when the system is executing. Static transformation of the MoCha-π interface actions into traditional π-calculus actions is not possible here.

We begin by defining the notions of *names*, *threads*, *channels*, *runtime processes* and *resources*. Afterward, we define each of the MoCha-π actions. Finally, we discuss structural congruence and introduce the general rules of the calculus.

4.3.1 Threads, Channels, Processes and Resources

We assume that there exists an infinite set $\mathcal{N}$ of *names*, with lower-case elements that range over $\mathcal{N}$. In the π-calculus names can refer to both data and *channels*. In our calculus names, among other things, refer to both data and *channel-ends*. A MoCha-π process operates on and exchanges with other processes channel-end names instead of channel names. To avoid confusion, from now on we refer to π-calculus channels as *links*. As we shall see, a MoCha-π channel is a process that uses *links* to communicate with other processes. We denote links with $c \in \mathit{Links} \subseteq \mathcal{N}$. For channel-ends we use $e \in \mathit{ChannelEnds} \subseteq \mathcal{N}$. Data is represented by $d \in \mathit{Data} \subseteq \mathcal{N}$. Observe that the type sets $\mathit{Links}, \mathit{ChannelEnds}$, Data are mutually disjoint. However for convenience, in this chapter we often use the same name for the channel-end and the link that it is translated to. All data, links, and channel-ends are represented by $a, b, x \in \mathcal{N}$. We assume that our calculus knows the right type of each name.

A system in MoCha-π consists of four kinds of processes: *threads*, *channels*, *runtime processes* and *resources*. The first two types are process specifications defined by the user. The third type consists of the runtime operational semantic processes of the first two. The last type contains processes without a body. We use capital letters to denote processes. For example, we use words like: $\{T, \mathit{PRODUCER}\}$ for *threads*, $\{K, \mathit{FIFO}\}$ for *channels*, $\{P, \mathit{PROCESS}\}$ for *runtime processes*. For *resources* we use a special notation given in Definition 4.3.2. A system definition is given by $\mathit{System} = \langle D|S\rangle$, where D is the system declaration consisting of threads and channels. S is the main statement; an initial thread that creates all other processes.

To model process creation, we assume an infinite set Id of *process identifiers*. We refer to $A \in \mathit{Id}$ as an identifier for a runtime process. We refer to A_K as an identifier for a runtime process of a channel. In the process specification we write $A(y_1, ..., y_n)$ to indicate the creation, or invocation, of process-*id* A with parameters $y_1, ..., y_n$. This identifier A has a defining equation of the form $A(x_1, ..., x_n) \stackrel{def}{=} P$, where all the parameters are distinct and free names in P. From the congruence rule (6) in Section 4.3.2, we can see that the creation of a process consists of substituting all free names $x_1, ..., x_n$ by the actual parameters $y_1, ..., y_n$. In [Mil99], Milner presents the basic π-calculus without process identifiers. Instead he introduces the replication action $!P$ to replace recursive process definitions. However, in MoCha-π we explicitly want to model dynamic process creation. This seems to be more natural and closer to the implementation of the MoCha framework.

Definition 4.3.1 *A* Thread *is a user-defined process specification with grammar* L^{φ}

that has the following syntax:

$$T ::= \sum_{i \in I} \varphi_i . T_i \mid T_1 | T_2 \mid new\ x\ T \mid A(y_1, ..., y_n)$$

where I is any finite indexing set. The actions φ of threads are:

$e \downarrow$ *connect to channel-end* e
$e \uparrow$ *disconnect from channel-end* e
$e!\langle x \rangle$ *write* x *to channel-end* e
$e?(x)$ *take* x *from channel-end* e
τ *unobservable action*

The processes $\sum_{i \in I} \varphi_i . T_i$ are called summations or sums. If $I = \{1, 2\}$, for example, then we get the summation $\varphi_1 . T_1 + \varphi_2 . T_2$. If $I = 0$ then we get the empty sum $\mathbf{0}$. The *composition* $T_1 | T_2$ indicates that these two processes run concurrently. The *restriction new* x T restricts the scope of the name x to process T. Threads may use the general identifier A. Thus, a thread can dynamically create any process of type *thread* or *channel* in the system.

In our calculus channels, and thus channel-ends, are viewed as resources. Therefore, processes must compete with each other in order to gain access to a particular channel-end by connecting to it.

Definition 4.3.2 *A resource is a process without a body that always runs in parallel with the processes of a system. There is a set of resources associated with every channel-end. Therefore, we define a relation between the name of a channel-end and its resource names. We denote by $\mathcal{R}^e$ a resource that belongs to channel-end e.*

Channels are processes too. This gives us the advantage that the behavior of a particular channel type can be defined in terms of actions. Moreover, we shall see that introducing a new type of channel consists of just defining a new process without having to make any changes to the existing actions or rules.

Definition 4.3.3 *A* Channel *is a user-defined process specification with grammar L^ϑ that has the following syntax:*

$$K ::= \sum_{i \in I} \vartheta_i . K_i \mid K_1 | K_2 \mid new\ x\ K \mid \mathcal{R}^e \mid A_K(y_1, ..., y_n)$$

where I is any finite indexing set. The actions ϑ of channels are:

$\overline{c}\langle x \rangle$ *send* x *along link* c
$c(x)$ *receive* x *along link* c
τ *unobservable action*

Channels are special kinds of processes because they can perform only the original π-calculus actions. Channels can use only the channel identifiers A_K. Thus, a channel can create only channel sub-processes and no threads.

Each channel receives at its creation a user defined number of *ends* $e_1, e_2, e_3, ...$ to communicate with the non-channel MoCha-π processes. These ends are specified

as the parameters of the channel process. Upon invocation of a channel process the parameter ends are automatically translated to their respective π-calculus links that comply with the I/O channel-end actions; see Section 4.3.3. The channel process specifies the behavior of its ends. For example, whether an end is a sink, source or both. The process also specifies the relation between the various ends of the channel in order to obtain a certain desired behavior. Each end e of a channel process K has a user defined number of resources $\mathcal{R}^e_1, \mathcal{R}^e_2, \mathcal{R}^e_3, ...$ We use $\mathcal{R}^e \in \{\mathcal{R}^e_1, \mathcal{R}^e_2, \mathcal{R}^e_3, ...\}$ to refer to any resource of a particular channel-end e.

We now define the (runtime) processes of the user-defined threads and channels.

Definition 4.3.4 *A* runtime process *is an operational semantic process for either a* thread *or a* channel. *Its definition is given by $L^\pi = L^{\varphi \cup \vartheta}$. The runtime process expressions are defined by the following syntax:*

$$P ::= \sum_{i \in I} \pi_i.P_i \mid P_1|P_2 \mid new\ x\ P \mid e[P] \mid \mathcal{R}^e \mid A(y_1, ..., y_n)$$

where I is any finite indexing set, and the actions $\pi = \varphi \cup \vartheta$.

All the expressions in this grammar are already defined except for $e[P]$. This expression symbolizes the fact that process P is currently *connected to* channel-end e.

We introduce two convenient abbreviations for the syntax of *threads*, *channels*, and *runtime processes*. We omit '$.\mathbf{0}$'; for example we write $\pi_1.\pi_2.\mathbf{0}$ as $\pi_1.\pi_2$. Also, we write $new\,(a_1, ..., a_n)$ instead of $new\ a_1 \ldots new\ a_n$; for example $new\ a\ new\ b$ is abbreviated to $new\,(a, b)$.

4.3.2 Structural Congruence

We define a *structural congruence* relation. We need this relation to identify the process expressions that are intuitively equivalent by having the same structure, but are nevertheless syntactically different. It is clear that for channel processes we can take the π-calculus definition of structural congruence as given in [Mil99]. For runtime and thread processes we need to extend this definition due to the addition of the connected scope $e[\,]$ to the MoCha-π calculus.

Definition 4.3.5 *Two process expressions P and Q in the MoCha-π calculus are structurally congruent, written $P \equiv Q$, if we can transform one into the other by using the following equations (in either direction):*

1. *Systematic change of bound names (alpha-conversion)*

2. *Reordering of terms in a summation*

2'. $P + \mathbf{0} \equiv P,\ P + Q \equiv Q + P,$
$P + (Q + R) \equiv (P + Q) + R$

3. $P \mid \mathbf{0} \equiv P,\ P \mid Q \equiv Q \mid P,\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

4. $new\,x\,(P \mid Q) \equiv P \mid new\,x\,Q \quad if\ x \notin fn(P)$
$new\,x\,\mathbf{0} \equiv \mathbf{0},\ new\,(x,y)\,P \equiv new\,(y,x)\,P$

5. $A(\vec{y}) \equiv \{^y/_x\}P$ *if* $A(\vec{x}) \stackrel{def}{=} P$

6. $e[g[P]] \equiv g[e[P]]$
   $e[P + Q] \equiv e[P] + e[Q]$
   $e[P] \mid \mathcal{R}^d \equiv e[P \mid \mathcal{R}^d]$

where $fn(P)$ *are all the free names in process* P*, with* $n(\mathcal{R}^e) = e$.

The last law of our added equation (6) states that a resource can "enter" and "leave" any connected scope. Due to the rules we define in Section 4.3.4, this law is not needed. However, it is still convenient to define it for achieving more readable process expressions. Observe that we do not add $e[\mathbf{0}] \equiv \mathbf{0}$, for this is not the case in our calculus. We want processes to explicitly disconnect from channel-ends.

4.3.3 Actions

We now define the actions of our calculus. For the MoCha-π channel-end topological actions we define a transition relation $\longrightarrow$. For the MoCha-π I/O actions we define a structural congruence relation.

send: $\overline{c}\langle x \rangle$
This is a π-calculus action where a name x is sent through link c.

receive: $c(x)$
This is the complementary action of send, where a name x is received through the link c.

The actions $\overline{c}\langle a \rangle$, $c(b)$ combined with the reaction rule of Section 4.3.4 model the synchronous sending of a name a through link c. The process that receives the name substitutes all occurrences of name b by a.

connect: $(e \downarrow .P \mid Q \ + \ M) \mid \mathcal{R}^e \longrightarrow e[P] \mid Q$
For a successful channel-end connection one of the resources $\mathcal{R}^e$ of the channel-end e must be available. After the action the resource $\mathcal{R}^e$ is removed from the expression (hidden) and is, therefore, not available anymore for any other process outside the scope [ ] that might know e. By counting the consumed resources, we know how many processes are currently connected to the channel-end. Processes that try to connect to a particular channel-end while all its resources are already taken by other processes, must wait until a resource becomes available again.

In the MoCha framework design pattern (see Section 4.4) each channel-end has exactly one resource. The success of a connect operation, therefore, guarantees exclusive channel-end access for its performing process P.

Components can also successfully connect to a channel-end they are already connected to (see the semantics given in Chapter 5). However, since we are dealing with processes we omit this particular connect action.

disconnect: $e[e \uparrow .P \mid Q \ + \ M] \longrightarrow P \mid Q \mid \mathcal{R}^e$

If a process P is currently connected to a channel-end e it can disconnect from it by performing the disconnect action $e \uparrow$. After a successful disconnect a resource $\mathcal{R}^e$ becomes available to other processes.

Components can also successfully disconnect from a channel-end they are not connected to in the first place. We omit this particular disconnect action in our process algebra calculus. We do model this action in Chapter 5, where we introduce the notion of components.

We now present the rules for the actions *write* and *take*. The idea is to dynamically transform these high-level interface actions into a communication pattern consisting of the standard $\overline{c}\langle x\rangle$ and $c(x)$ π-calculus actions. These patterns are needed to ensure exogenous coordination, by making every MoCha-π I/O action between a thread and a channel-end synchronous. Instead of a transition relation we use a structural congruence one for the write and take actions. The rationale for this choice is to avoid unnecessary transitions in our calculus.

`Write`: $e[e!\langle a\rangle.P \mid Q \; + \; M] \equiv e[\overline{e}\langle a\rangle.e(\lambda).P \mid Q \; + \; M]$

A write action on a channel-end e is transformed into a communication pattern using link e if process P is currently connected to the channel-end. For simplicity, in this chapter we use the same name for the channel-end as well as for the link it is translated to. In this pattern, first a value a is communicated to the channel process, then we wait for an acknowledgment λ to be received through e. We shall always use λ as a special reserved name for acknowledgments and requests (see the take action). As stated in Section 4.3.1, channels match this I/O link pattern. They do so by first matching a thread's send $\overline{e}\langle a\rangle$ with a receive $e(x)$. Later, at some point in time, they match a thread's receive $e(\lambda)$ with a send $\overline{e}\langle\lambda\rangle$.

Observe that we don't have any means to check whether the channel-end e is a source-end or not. However, we don't need to. If a sink-end is given, the structural law translates the interface action into the π-calculus actions anyway. However, these actions deadlock because they do not match the communication pattern used by the channel process.

`Take`: $e[e?(b).P \mid Q + \; M] \equiv e[\overline{e}\langle\lambda\rangle.e(b).P \mid Q \; + \; M]$

A take action on a channel-end e is transformed into a π-calculus communication pattern using its corresponding link e if process P is currently connected to the channel-end. First, a request λ to take is sent to the channel, then a name b is received from the channel process through e. As stated in Section 4.3.1, channels match this link I/O pattern. They do so by first matching a thread's send $\overline{e}\langle\lambda\rangle$ with a receive $e(\lambda)$. Later, at some point in time, they match a thread's receive $e(b)$ with a send $\overline{e}\langle x\rangle$. Just like with the write action, we don't put any restrictions on the type of the channel-end.

`Tau`: $\tau.P \; + \; Q \longrightarrow P$

Finally, τ represents the unobservable action.

Parallel
$$\frac{P \longrightarrow P'}{P|Q \longrightarrow P'|Q}$$

Restriction
$$\frac{P \longrightarrow P'}{new\ x\,P \longrightarrow new\ x\ P'}$$

Connection(1)
$$\frac{P \longrightarrow P'}{e[P] \longrightarrow e[P']}$$

Connection(2)
$$\frac{P \mid Q \longrightarrow P' \mid Q'}{M\ +\ e[P] \mid Q \longrightarrow P' \mid e[P'] \mid Q'}$$

Structural Rule
$$\frac{P \longrightarrow P'}{Q \longrightarrow Q'} \quad if\ P \equiv Q\ and\ P' \equiv Q'$$

Reaction:
$$(c(a).P + M) \mid (\overline{c}\langle b\rangle.Q + N) \longrightarrow \{^b/_a\}P \mid Q$$

Figure 4.1: The General Rules of MoCha-π

4.3.4 Reaction and other Rules

We now define the reaction and other support rules of MoCha-π. We take the π-calculus rules and add two connection rules. All the rules are given in Figure 4.1.

The reaction rule works at the π-calculus level. Therefore, this rule is oblivious to whether or not processes are connected to particular channel-ends. However, in MoCha-π we need to take into account that processes may be connected to one or more channel-ends. Connection rule (1) gives us the means to let a reaction happen within a connected scope. Connection rule (2) allows a reaction to happen when one of the two processes involved is in a connected scope, which is the case when a thread communicates with a channel process. By applying the connection rules several times a reaction is able to succeed in cases when processes are connected to multiple channel-ends. For example, the communication in $z[e[M\ +\ d[e(a).P]] \mid \overline{e}\langle b\rangle.Q]$ succeeds.

The connection rules also make it possible for the connect and the disconnect rules to succeed when a process executing a connect or a disconnect action is connected to multiple channel-ends. For example, the connect action in $f[d[e \downarrow .P]] \mid \mathcal{R}^e$ succeeds, and the disconnect action in $e[Q \mid d[e \uparrow .P]]$ succeeds as well.

4.3.5 Sequential Composition

The sequential composition is not a primitive operator of MoCha-π nor the π-calculus, for it can easily be modeled in both calculi. We take the same approach as in [Mil99]:

Definition 4.3.6 *We define the sequential composition, $P\,;Q$ ("Q starts, when P finishes"), by providing the following construction:*

$$P\,;Q \stackrel{def}{=} new\ (start, d)(\ \{start/done\}P \mid start(s).Q\ )$$

where we assume that process P ends with a $\overline{done}\langle d\rangle$ action, and both start and d are not free in P or Q.

We use the ';' operator in our examples for both channel and thread processes definitions. There is no need to define sequential composition using MoCha-π actions for when we interpretor the ';' operator for threads we are already dealing with runtime processes. For simplicity, in this thesis when working out an example we assume ';' to be a primitive operator instead of substituting it for above construction.

4.4 The MoCha Framework Design Pattern

The MoCha-π calculus allows channels to have a user defined number of channel-ends. These ends are of types source, sink or both. For each end there is a user defined number of associated resources. All of this is specified in the definition of the channel process type. Therefore, the calculus is more general than the MoCha framework where there are certain restrictions on the channel-ends and their resources. In order to be able to focus on modeling only the MoCha framework, we introduce a *design pattern*. This pattern states that (1) all channels have exactly two channel-ends; (2) their end types are either sink or source but not both; and (3) every channel-end has exactly one resource.

To define our own channel types in the MoCha-π calculus, we must write a channel process that receives the channel-end links as parameters, together with the capacity of the channel (if any). This process then must match the communication patterns of the interface *write* and *take* operations (see Section 4.3.3) and relate the ends of the channel using π-calculus actions.

We make one further restriction in our pattern: (4) we demand that the actual channel-end parameters are all unique and distinct from each other for each channel. This restriction obligates us to bind the channel-ends before creating a channel, and to use them for the invocation of only one channel process. For example, $S \stackrel{def}{=} new\ (e_1, e_2, e_3, e_4)(\ K(e_1, e_2) |\ K(e_3, e_4)\ )$, where S creates two channel processes of type K is allowed, but not $S \stackrel{def}{=} new\ (e_1, e_2)(\ K(e_1, e_2)\ |K(e_1, e_2)\ )$, where S creates two channel processes that share their ends.

4.4.1 Specifying Channel Types

We now give a MoCha-π specification for the majority of the channel types we discussed in Chapter 2. We omit the filter channel type because in MoCha-π its behavior is equal to the one of the synchronous channel type. All of these specifications conform to the MoCha design pattern. The behavior of these channels is already given in Section 2.4. The channel processes carry the name of the type. However, for simplicity in the definition of each channel we refer to it as the channel process K instead of, for example, $SYNCHRONOUS$. All the channels receive two links $\{l, r\}$ as actual parameters. Each link corresponds to a channel-end used by thread processes. For convenience, in the examples we write $CE(l)$ or $CE(r)$ to denote the

channel-end that corresponds to, respectively, link l or r. Upon initialization, all the channels create one resource for every channel-end. These resources are $\mathcal{R}^l$ and $\mathcal{R}^r$.

Synchronous

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r)$$
$$K'(l,r) \stackrel{def}{=} l(x).r(\lambda).(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle x\rangle);K'(l,r)$$

This channel process has a source-end $CE(l)$, and a sink-end $CE(r)$. Initially the process first receives a name, x, from its source-end, then sequentially it receives a request from its sink-end. Finally, it sends in parallel both an acknowledgment to its source-end and the name x to its sink-end. Afterward, the process loops and starts again waiting for the next write on its source-end.

Observe that, a synchronous channel allows the two (take and write) operations on its ends to succeed atomically. This does not imply that these operations must be performed simultaneously.

Sometimes the order in which we service the channel-ends does not matter for a specific channel type. In this case, for the synchronous channel type, it does not matter if we take a value from the left channel-end first and then a request from the right one (see above) or vice-versa:

$$K'(l,r) \stackrel{def}{=} r(\lambda).l(x).(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle x\rangle);K'(l,r)$$

We can even put these actions in parallel:

$$K'(l,r) \stackrel{def}{=} (l(x) \mid r(\lambda));(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle x\rangle);K'(l,r)$$

Thus, when the order does not matter, it is up to the taste of the user defining the particular channel type to decide which order he likes.

Lossy Synchronous

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r)$$
$$K'(l,r) \stackrel{def}{=} (\ l(x).(\ (r(\lambda).(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle x\rangle)) + \bar{l}\langle\lambda\rangle\ )\ );K'(l,r)$$

where the left term of the summation has priority over the right one

The *lossy synchronous* channel consists of two parts: a synchronous and a lossy part. This is represented by the summation in the channel's specification. The left side of the summation is the synchronous part. This is the same specification as the one of the *synchronous* channel type given above. The right side of the summation is the lossy part, where the written name x gets lost. Ideally, we would like to execute the lossy part only in those cases where there is a *write* but not a *take* action on the channel. However, since the channel has no global information, it has a non-deterministic choice in choosing any side of the summation. To cope with this problem we locally impose a priority on the left term of the summation. This way, we enforce the channel to behave like a synchronous type every time that there is a

take operation available.

FIFO

$$
\begin{array}{ll}
K(l,r) \stackrel{def}{=} & \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r,0) \\
K'(l,r,\vec{v})^{\{|\vec{v}|=0\}} \stackrel{def}{=} & l(v).\bar{l}\langle\lambda\rangle.K'(l,r,\langle v\rangle) \\
K'(l,r,\vec{v})^{\{|\vec{v}|\geq 1\}} \stackrel{def}{=} & (\ l(v).\bar{l}\langle\lambda\rangle.K'(l,r,\langle v_1,..,v_{|\vec{v}|},v\rangle)\ )+ \\
 & (\ r(\lambda).\bar{r}\langle v_1 \in \vec{v}\rangle.K'(l,r,\langle v_2,..,v_{|\vec{v}|}\rangle))
\end{array}
$$

This is the *asynchronous unbounded FIFO channel* where we model the buffer as a vector, or sequence, of names $\vec{v}$ that is passed on as a parameter of the channel process. Initially, the buffer is empty so the only action possible is a *write*. After this action, the written name v is added to the vector. The next possible actions are either again a *write*, represented by the left term of the summation, or, a *take*, represented by the right side of the same summation. In case of a *take* action the first value of the vector, v_1, that symbolizes the first written value into the buffer, is removed from the vector. If after a *take* action the buffer gets empty, i.e. the vector has no values, we return to the initial state where there is only a *write* action possible.

Observe, that if we want to model a LIFO channel type we merely need to take $v_{|\vec{v}|}$ out of the channel each time, instead of v_1. If we want to model a BAG channel type we can take any v_k where $k \leq |\vec{v}|$ out of the channel instead of v_1.

FIFO n

$$
\begin{array}{ll}
K(l,r) \stackrel{def}{=} & \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r,n,0) \\
K'(l,r,n,\vec{v})^{\{|\vec{v}|=0\}} \stackrel{def}{=} & l(v).\bar{l}\langle\lambda\rangle.K'(l,r,n,\langle v\rangle) \\
K'(l,r,n,\vec{v})^{\{1\leq|\vec{v}|\leq n-1\}} \stackrel{def}{=} & (\ l(v).\bar{l}\langle\lambda\rangle.K'(l,r,n,\langle v_1,..,v_{|\vec{v}|},v\rangle)\ )+ \\
 & (\ r(\lambda).\bar{r}\langle v_1 \in \vec{v}\rangle.K'(l,r,n,\langle v_2,..,v_{|\vec{v}|}\rangle)\ ) \\
K'(l,r,n,\vec{v})^{\{|\vec{v}|=n\}} \stackrel{def}{=} & r(\lambda).\bar{r}\langle v_1 \in \vec{v}\rangle.K'(l,r,n,\langle v_2,..,v_n\rangle)
\end{array}
$$

The *FIFO n* channel type has the same specification as the unbounded FIFO one, with the difference that this channel process has: an extra parameter n that represents the channel's capacity, and an extra process specification. This extra specification states that when the channel reaches its capacity, the length of the vector sequence $\vec{v}$ is n, the only possible action is a *take*. Any *write* operation in that state is automatically suspended by the calculus until the channel process arrives at a state where the length of the vector is less than n.

Synchronous Drain

$$
\begin{array}{l}
K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r) \\
K'(l,r) \stackrel{def}{=} (\ l(x_1).r(x_2).(\bar{l}\langle\lambda\rangle \mid \bar{r}\langle\lambda\rangle)\ );K'(l,r)
\end{array}
$$

This channel type has two source-ends, $\{CE(l), CE(r)\}$, that allows operations to succeed on them only atomically. The channel first receives a value from each source-end in a sequential manner. Then, it sends an acknowledgment back to the ends in parallel. Afterward, the process loops and starts again waiting for the next

pair of *write* actions to be performed on its source-ends.

Asynchronous Drain

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r)$$
$$K'(l,r) \stackrel{def}{=} (\ (l(x_1).\bar{l}\langle\lambda\rangle) + (r(x_2).\overline{r}\langle\lambda\rangle)\ ); K'(l,r)$$

This is the asynchronous version of the channel above. The two sources of the *asynchronous drain* behave in an exclusively asynchronous way; i.e. never synchronous. To achieve this behavior, the channel specification gives a summation between the write actions of each channel-end. This way, there is always a choice between writing to $CE(l)$ or $CE(r)$ but never to both of them at the same time.

Drain

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l) \mid K'(r)$$
$$K'(z) \stackrel{def}{=} z(x).\overline{z}\langle\lambda\rangle.K'(z)$$

In the *drain* channel type the two source-ends are independent of each other. Therefore, at initialization, the channel process divides itself into two independent processes; a process for each source-end that handles the *write* actions performed on this end.

Synchronous Spout

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r)$$
$$K'(l,r) \stackrel{def}{=} new\ x\ (\ l(\lambda).r(\lambda).(\bar{l}\langle x\rangle \mid \overline{r}\langle x\rangle)\ ); K'(l,r)$$

This channel type has two sink-ends, $\{CE(l), CE(r)\}$, that allows operations to succeed on them only atomically. The channel first receives a request, */lambda*, from each sink-end in a sequential manner. Then, it sends a value back to the ends in parallel. Afterward, the process loops and starts again waiting for the next pair of *take* actions to be performed on its source-ends.

Asynchronous Spout

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l,r)$$
$$K'(l,r) \stackrel{def}{=} new\ x\ (\ (l(\lambda).\bar{l}\langle x\rangle) + (r(\lambda).\overline{r}\langle x\rangle))\ ); K'(l,r)$$

This is the asynchronous version of the channel above. The two sink-ends of the *asynchronous spout* channel behave in an exclusively asynchronous way; i.e. never synchronous. To achieve this behavior, the channel specification gives a summation between the *take* actions of each channel-end. This way, there is always a choice between taking from $CE(l)$ or $CE(r)$ but never from both of them at the same time.

Spout

$$K(l,r) \stackrel{def}{=} \mathcal{R}^l \mid \mathcal{R}^r \mid K'(l) \mid K'(r)$$
$$K'(z) \stackrel{def}{=} new\ x\ (z(\lambda).\overline{z}\langle x\rangle); K'(z)$$

In the *spout* channel type the two sink-ends are independent of each other. Therefore, at initialization, the channel process divides itself into two independent processes; a process for each source-end that handles the *take* actions performed on this end.

4.5 Examples

In this section we give three examples that show how to use the MoCha-π calculus and its benefits. We first give simple producer/consumer examples, then, we continue with a more representative example about mobile phoning in cars, and we end with implementing the mobile agent example of Section 2.3.

4.5.1 Producer/Consumer Examples

This is the classical producer/consumer example where, a process P inserts values into a channel and a process Q takes values from the same channel. With this example we want to demonstrate the use of the MoCha-π high-level interface actions, how they are dynamically derived into traditional π-calculus actions, and how these derived actions make the MoCha-π processes anonymously interact with each other, while they are being exogenously coordinated. We start by giving the specification of our system:

$$
\begin{aligned}
&S \stackrel{def}{=} new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ SYNCHRONOUS(l,r)\ )\\
&P(e) \stackrel{def}{=} new\ d\ (\ e\downarrow.e!\langle d\rangle.e\uparrow);P(e)\\
&Q(e) \stackrel{def}{=} e\downarrow.e?(x).e\uparrow.Q(e)
\end{aligned}
$$

We have a system S that creates three processes: a producer P, a consumer Q, and a channel process $SYNCHRONOUS$. The system binds the channel-end names and passes them on to the channel. It also passes on the source-end of this channel to the producer and the sink-end to the consumer. The producer process, P, then connects to the source-end of the channel, writes a value d to it, and then disconnects from the end to start the cycle again. Analogous to P, the consumer process, Q, connects to the sink-end of the channel, takes a value, and then it disconnects from this end to start again.

Notice, that due to the *exogenous coordination* property of the calculus, we can change the behavior of the system by simply choosing another type of channel between the processes. For example, we can chose a FIFO channel process. This gives us the following specification:

$$
\begin{aligned}
&S \stackrel{def}{=} new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ FIFO(l,r)\ )\\
&P(e) \stackrel{def}{=} new\ d\ (\ e\downarrow.e!\langle d\rangle.e\uparrow);P(e)\\
&Q(e) \stackrel{def}{=} e\downarrow.e?(x).e\uparrow.Q(e)
\end{aligned}
$$

Observer that, we don't have to change the specification of either the producer or the consumer process. In fact, these processes don't notice anything of this change.

Working out the Synchronous Channel Example

To illustrate how the MoCha-π calculus works, we now show a possible path of actions of the system where one value is transmitted from the producer to the consumer process, using the synchronous channel type.
First we initialize the system:

$$\begin{aligned}
&S = new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ SYNC(l,r)\ )\\
&P(l) = new\ d\ (\ l\downarrow .l!\langle d\rangle .l\uparrow);P(l)\\
&Q(r) = r\downarrow .r?(x).r\uparrow .Q(r)\\
&SYNC(l,r) = \mathcal{R}^l\ |\ \mathcal{R}^r\ |\ SYNC'(l,r)\\
&SYNC'(l,r) = l(x).r(\lambda).(\bar{l}\langle\lambda\rangle\ |\ \bar{r}\langle x\rangle);SYNC'(l,r)
\end{aligned}$$

We abbreviate the name of the channel process *SYNCHRONOUS* into *SYNC* for space saving reasons. Since the *SYNC* process consists of two resources and a process $SYNC'$ in parallel, we can re-write above specification into a more convenient form:

$$\begin{aligned}
&S = new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ SYNC'(l,r)\ |\ \mathcal{R}^l\ |\ \mathcal{R}^r\ )\\
&P(l) = new\ d\ (\ l\downarrow .l!\langle d\rangle .l\uparrow);P(l)\\
&Q(r) = r\downarrow .r?(x).r\uparrow .Q(r)\\
&SYNC'(l,r) = l(x).r(\lambda).(\bar{l}\langle\lambda\rangle\ |\ \bar{r}\langle x\rangle);SYNC'(l,r)
\end{aligned}$$

We now start the data transfer where we denote each step by using a numbered arrow. The action or rule corresponding to a particular step is given on top of the arrow. We begin by letting the producer *connect* to the channel-end l:

$$\begin{aligned}
&\xrightarrow{l\downarrow}_1\\
&S = new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ SYNC'(l,r)\ |\ \mathcal{R}^r\ )\\
&P(l) = new\ d\ (\ l[l!\langle d\rangle .l\uparrow);P(l)]\\
&Q(r) = r\downarrow .r?(x).r\uparrow .Q(r)\\
&SYNC'(l,r) = l(x).r(\lambda).(\bar{l}\langle\lambda\rangle\ |\ \bar{r}\langle x\rangle);SYNC'(l,r)
\end{aligned}$$

We see indeed that, after the *connect* action, the resource $\mathcal{R}^l$ is not available anymore, because it is removed from the specification. We can also see that, the *connect scope* is not limited by the *new scope*. In fact they don't affect each other. Now that we are connected, our next action is a *write*:

$$\begin{aligned}
&\xrightarrow{l!\langle d\rangle}_2\\
&S = new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ SYNC'(l,r)\ |\ \mathcal{R}^r\ )\\
&P(l) = new\ d\ (\ l[\bar{l}\langle d\rangle .l(\lambda).l\uparrow);P(l)]\\
&Q(r) = r\downarrow .r?(x).r\uparrow .Q(r)\\
&SYNC'(l,r) = l(x).r(\lambda).(\bar{l}\langle\lambda\rangle\ |\ \bar{r}\langle x\rangle);SYNC'(l,r)
\end{aligned}$$

The write action of P is transformed into a pattern of traditional π-calculus actions since it is connected to the source-end of the channel. Process P is now ready to interact with process *SYNCHRONOUS* in order to execute the MoCha-π high-level *write* action.
We use the reaction rule to start the high-level *write* action:

$$\xrightarrow{Reaction(\bar{l}\langle d\rangle, l(x))}_3$$
$$\begin{array}{l} S = new(l,r)\ (\ P(l) \mid Q(r) \mid SYNC'(l,r) \mid \mathcal{R}^r\ ) \\ P(l) = new\ d\ (\ l[l(\lambda).l \uparrow); P(l)] \\ Q(r) = r \downarrow .r?(x).r \uparrow .Q(r) \\ SYNC'(l,r) = r(\lambda).(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle d\rangle); SYNC'(l,r) \end{array}$$

At this point P has to wait for an acknowledgment that will only come if Q performs a take interface action.
We now let the consumer *connect* to the channel-end r:

$$\xrightarrow{r\downarrow}_4$$
$$\begin{array}{l} S = new(l,r)\ (\ P(l) \mid Q(r) \mid SYNC'(l,r)\ ) \\ P(l) = new\ d\ (\ l[l(\lambda).l \uparrow); P(l)] \\ Q(r) = r[r?(x).r \uparrow .Q(r)] \\ SYNC'(l,r) = r(\lambda).(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle d\rangle); SYNC'(l,r) \end{array}$$

At this point, because we removed them from the specification, both of the resources are not available anymore for other processes.
Now that the consumer is connected to channel-end r, we execute the *take* action:

$$\xrightarrow{r?(x)}_5$$
$$\begin{array}{l} S = new(l,r)\ (\ P(l) \mid Q(r) \mid SYNC'(l,r)) \\ P(l) = new\ d\ (\ l[l(\lambda).l \uparrow); P(l)] \\ Q(r) = r[\overline{r}\langle\lambda\rangle.r(x).r \uparrow .Q(r)] \\ SYNC'(l,r) = r(\lambda).(\bar{l}\langle\lambda\rangle \mid \overline{r}\langle d\rangle); SYNC'(l,r) \end{array}$$

The take action of Q is transformed into a pattern of traditional π-calculus actions since it is connected to the sink-end of the channel. Just like process P, process Q is now also ready to interact with process *SYNCHRONOUS*.
We use the reaction rule to start the high-level *take* action:

$$\xrightarrow{Reaction:(\overline{r}\langle\lambda\rangle, r(\lambda))}_6$$
$$\begin{array}{l} S = new(l,r)\ (\ P(l) \mid Q(r) \mid SYNC'(l,r)) \\ P(l) = new\ d\ (\ l[l(\lambda).l) \uparrow ; P(l)] \\ Q(r) = r[r(x).r \uparrow .Q(r)] \\ SYNC'(l,r) = (\bar{l}\langle\lambda\rangle \mid \overline{r}\langle d\rangle); SYNC'(l,r) \end{array}$$

At this point both of the high-level actions, *write* and *take*, have started their low-level interaction with the channel process. The channel is now ready to end both actions at the same time.
We now perform two obvious steps, we let the remaining π-calculus send and receive actions to be executed:

$$\xrightarrow{Reaction:(\bar{l}\langle\lambda\rangle, l(\lambda))\ ;\ (\overline{r}\langle d\rangle, r(x))}_{7,8}$$
$$\begin{array}{l} S = new(l,r)\ (\ P(l) \mid Q(r) \mid SYNC'(l,r)) \\ P(l) = l[l \uparrow ; P(l)] \\ Q(r) = r[r \uparrow .Q(r)] \\ SYNC'(l,r) = SYNC'(l,r) \end{array}$$

The value is now finally transmitted from the producer to the consumer process. We now let the producer and consumer process *disconnect* from their ends:

$$\xrightarrow{l\uparrow;r\uparrow}_{9,10}$$
$$S = new(l,r)\ (\ P(l)\ |\ Q(r)\ |\ SYNC'(l,r)\ |\ \mathcal{R}^l\ |\ \mathcal{R}^r\ )$$
$$P(l) = P(l)$$
$$Q(r) = Q(r)$$
$$SYNC'(l,r) = SYNC'(l,r)$$

The *disconnect actions* results in the return of the channel-end resources $\mathcal{R}^l$ and $\mathcal{R}^r$. At this point, we are back at the specification which we started with.

Producer/Producer and Consumer/Consumer

Instead of the typical producer/consumer scenario, We can use two source- and two sink-end channel types to coordinate two producers or two consumers.

If we use a channel with two source-ends we can compose two producer processes together. For example, using the synchronous drain channel type we get:

$$S \stackrel{def}{=} new(l,r)\ (\ P(l)\ |\ P(r)\ |\ SYNCDRAIN(l,r)\ )$$
$$P(e) \stackrel{def}{=} new\ d\ (\ e\downarrow.e!\langle d\rangle.e\uparrow);P(e)$$

On the other hand, if we use a two sink-end channel type we can compose two consumer processes together. For example, using the asynchronous spout channel type we get:

$$S \stackrel{def}{=} new(l,r)\ (\ Q(l)\ |\ Q(r)\ |\ ASYNCSPOUT(l,r)\ )$$
$$Q(e) \stackrel{def}{=} e\downarrow.e?(x).e\uparrow.Q(e)$$

Competing Producers and Consumers

Without having to change the specification of either the producer, the consumer, or the synchronous channel, we can add more producer and consumer processes to our initial example. The producers then automatically compete against each other to obtain the resource of the source channel-end, and the consumers compete for the resource of the sink-end. An example:

$$S \stackrel{def}{=} new(l,r)\ (\ P(l)\ |\ P(l)\ |\ P(l)\ |\ Q(r)\ |\ Q(r)\ |\ SYNC(l,r)\ )$$
$$P(e) \stackrel{def}{=} new\ d\ (\ e\downarrow.e!\langle d\rangle.e\uparrow);P(e)$$
$$Q(e) \stackrel{def}{=} e\downarrow.e?(x).e\uparrow.Q(e)$$

We initialize the system and re-write its specification in the same way we have done previously. This time we concentrate at the producer processes and, therefore, we omit the body of the consumers and the synchronous channel. We index the identity of the processes for convenience:

$$
\begin{array}{ll}
S = new(l,r) & (\ P_1(l) \mid P_2(l) \mid P_3(l) \mid Q_1(r) \mid Q_2(r) \mid \\
 & SYNC'(l,r) \mid \mathcal{R}^l \mid \mathcal{R}^r\ ) \\
\multicolumn{2}{l}{P_1(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_1(l)} \\
\multicolumn{2}{l}{P_2(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_2(l)} \\
\multicolumn{2}{l}{P_3(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_3(l)}
\end{array}
$$

At this point resource $\mathcal{R}^l$ is available and any of the three producers can take it by performing a connect operation ($l \downarrow$). If all three try to execute the connect operation at the same time they automatically compete among each other and only one non-nondeterministically succeeds to perform the operation. Suppose that process P_3 “wins”:

$$
\begin{array}{l}
S = new(l,r)\ (\ P_1(l) \mid P_2(l) \mid P_3(l) \mid Q_1(r) \mid Q_2(r) \mid SYNC'(l,r) \mid \mathcal{R}^r\ ) \\
P_1(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_1(l) \\
P_2(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_2(l) \\
P_3(l) = new\ \ d\ \ (\ l[l!\langle d\rangle.l \uparrow);P_3(l)]
\end{array}
$$

Producer P_3 succeeds to connect to channel-end l and therefore the resource $\mathcal{R}^l$ is removed from the specification, so that no other producer can connect to the same channel-end anymore. P_3 can now transfer data with a consumer as described before. After which it disconnects from the source-end:

$$
\begin{array}{ll}
S = new(l,r) & (\ P_1(l) \mid P_2(l) \mid P_3(l) \mid Q_1(r) \mid Q_2(r) \mid \\
 & SYNC'(l,r) \mid \mathcal{R}^l \mid \mathcal{R}^r\ ) \\
\multicolumn{2}{l}{P_1(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_1(l)} \\
\multicolumn{2}{l}{P_2(l) = new\ \ d\ \ (\ l \downarrow .l!\langle d\rangle.l \uparrow);P_2(l)} \\
\multicolumn{2}{l}{P_3(l) = P_3(l)}
\end{array}
$$

After disconnecting, the resource $\mathcal{R}^l$ becomes available so that the producers can compete among each others again trying to connect and get exclusive access to channel-end l.

4.5.2 Mobile Phones

The mobile phones example is presented in [Mil99] to show how well the π-calculus deals with mobility. This is a representative example for the kind of systems that are easy to implement with the MoCha middleware. Therefore, in this section we present the same example but now using the MoCha-π calculus.

In this example *cars* are moving around while their passengers make calls using on-board mobile phones at arbitrary times. For this purpose each car is connected to a nearby *transmitter*. However, if a car gets too far from this transmitter it is switched to another more nearby transmitter. The coordination of all transmitters is done by a *control* unit.

For simplicity, just like in[Mil99], we consider only one car and two transmitters. In Figure 4.2 we show how the car switches from one transmitter to the other. The car is linked to the transmitters by two channels, an outgoing channel *talk* and an incoming channel *listen* (from the point of view of the car). In this example, we use a handy notation where we denote the source-end of a *channel* as *channel* and the

sink-end of the same channel as $\ddot{channel}$. The car is connected to the source-end $\dot{talk}$ of channel *talk* and to the sink-end $\ddot{listen}$ of channel *listen*. The transmitter is connected to the other ends $\ddot{talk}$ and $\dot{listen}$. The specification of the car is:

$$\begin{aligned} Car(\dot{talk}, \ddot{listen}) \stackrel{def}{=}\ & new\ d_1\ (\dot{talk} \downarrow .\dot{talk}!\langle d_1 \rangle.\dot{talk} \uparrow \\ & + \ddot{listen} \downarrow .\ddot{listen}?(d_2).\ddot{listen} \uparrow);Car(\dot{talk}, \ddot{listen}) \end{aligned}$$

The car can either talk or listen when connected to the appropriate channel-end. In contrast with the π-calculus example, our car does not (need to) receive any new channel-ends for communication from the transmitter after a switch. Our car does not even know that a switch is taking place nor with which transmitter it is communicating.

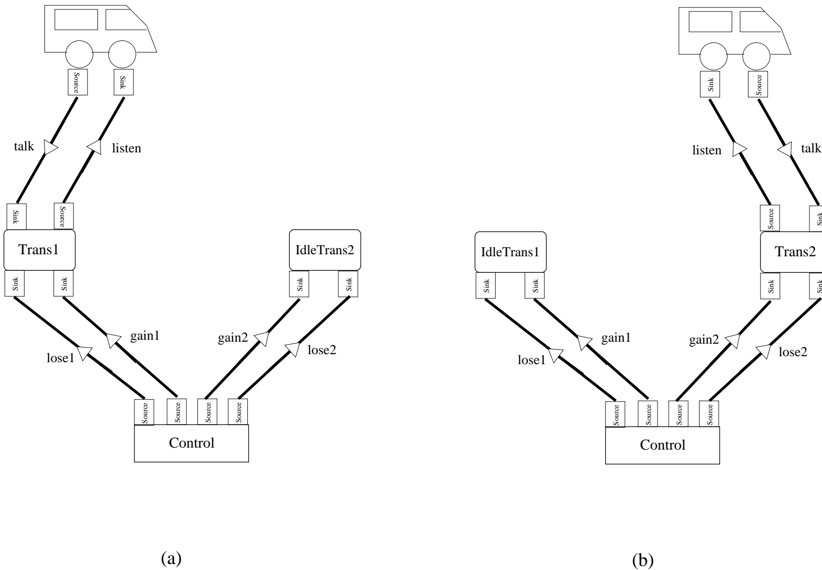


Figure 4.2: Example, Calling Mobile from a Car

The transmitters have two incoming channels *gain* and *lose*. They are connected to their sink channel-ends $\ddot{gain}$ and $\ddot{lose}$. Initially a transmitter is *idle*, but it becomes active when it receives the ends $\ddot{talk}$ and $\dot{listen}$ through the channel *gain*. After activation the transmitter starts the communication with the car. If an *active transmitter* receives the same two channel-ends through the channel *lose*, it terminates the communication and becomes idle. Observe, that we don't really need the two channels *gain* and *lose*, only one channel is sufficient. However, we want to stay close to the original example. Therefore, we model the two channels instead of just

one. Here is the specification of the transmitter, where $i = 1, 2$:

$$
\begin{aligned}
&\mathit{IdleTrans}_i(\ddot{\mathit{gain}}_i, \ddot{\mathit{lose}}_i) \stackrel{def}{=} \ddot{\mathit{gain}}_i \downarrow .\ddot{\mathit{lose}}_i \downarrow .\ddot{\mathit{gain}}_i?(\ddot{\mathit{talk}}) \\
&\quad .\ddot{\mathit{gain}}_i?(\dot{\mathit{listen}}).\ddot{\mathit{talk}} \downarrow .\dot{\mathit{listen}} \downarrow \\
&\quad .\mathit{Trans}_i(\ddot{\mathit{gain}}_i, \ddot{\mathit{lose}}_i, \ddot{\mathit{talk}}, \dot{\mathit{listen}}) \\
&\mathit{Trans}_i(\ddot{\mathit{gain}}_i, \ddot{\mathit{lose}}_i, \ddot{\mathit{talk}}, \dot{\mathit{listen}}) \stackrel{def}{=} \\
&\quad \mathit{new}\ d_2\ (\ddot{\mathit{talk}}?(d_1) + \dot{\mathit{listen}}!\langle d_2 \rangle) \\
&\quad ;\mathit{Trans}_i(\ddot{\mathit{gain}}_i, \ddot{\mathit{lose}}_i, \ddot{\mathit{talk}}, \dot{\mathit{listen}}) \\
&\quad +\ddot{\mathit{lose}}_i?(\ddot{\mathit{talk}}).\ddot{\mathit{lose}}_i?(\dot{\mathit{listen}}).\ddot{\mathit{talk}} \uparrow .\dot{\mathit{listen}} \uparrow \\
&\quad .\mathit{IdleTrans}_i(\ddot{\mathit{gain}}_i, \ddot{\mathit{lose}}_i)
\end{aligned}
$$

The control unit receives at its initialization the source-ends of each *gain* and *lose* channel. The control process then connects to these channel-ends. Besides these ends, control also receives as parameters the channel-ends $\ddot{\mathit{talk}}$ and $\dot{\mathit{listen}}$. It first writes these ends to the $\dot{\mathit{gain}}_1$ channel-end, so that transmitter 1 can start interacting with the car. This is the situation in figure 4.2(a). At some point in time, it writes the ends to $\dot{\mathit{lose}}_1$, making transmitter 1 idle again. Fortunately, afterward, it writes the ends to the $\dot{\mathit{gain}}_2$ channel-end. Now transmitter 2 becomes active and takes over the interaction with the car. This is the situation in Figure 4.2(b). After completing the switch, the control unit disconnects from all connected channel-ends and terminates. We give the specification:

$$
\begin{aligned}
&\mathit{Control}(\dot{\mathit{gain}}_i, \dot{\mathit{lose}}_i, \ddot{\mathit{talk}}, \dot{\mathit{listen}}) \stackrel{def}{=} \dot{\mathit{gain}}_i \downarrow .\dot{\mathit{lose}}_i \downarrow \\
&\quad .\dot{\mathit{gain}}_1!\langle \ddot{\mathit{talk}} \rangle.\dot{\mathit{gain}}_1!\langle \dot{\mathit{listen}} \rangle \\
&\quad .\dot{\mathit{lose}}_1!\langle \ddot{\mathit{talk}} \rangle.\dot{\mathit{lose}}_1!\langle \dot{\mathit{listen}} \rangle.\dot{\mathit{gain}}_2!\langle \ddot{\mathit{talk}} \rangle \\
&\quad .\dot{\mathit{gain}}_2!\langle \dot{\mathit{listen}} \rangle.\dot{\mathit{gain}}_i \uparrow .\dot{\mathit{lose}}_i \uparrow \quad \text{(where } i = 1, 2\text{)}
\end{aligned}
$$

Finally, we now must set up the system. The system process creates all others including the channel processes. It is this process that initially distributes all channel-ends.

$$
\begin{aligned}
&\mathit{Sys} \stackrel{def}{=} \mathit{new}(\dot{\mathit{talk}}, \ddot{\mathit{talk}}, \dot{\mathit{listen}}, \ddot{\mathit{listen}}, \dot{\mathit{lose}}_i, \ddot{\mathit{lose}}_i, \dot{\mathit{gain}}_i, \ddot{\mathit{gain}}_i) \\
&\quad (\ \mathit{SYNCHRONOUS}(\dot{\mathit{gain}}_i, \ddot{\mathit{gain}}_i) \mid \\
&\quad \mathit{SYNCHRONOUS}(\dot{\mathit{lose}}_i, \ddot{\mathit{lose}}_i) \mid \mathit{FIFO}(\dot{\mathit{talk}}, \ddot{\mathit{talk}}) \mid \\
&\quad \mathit{SYNCHRONOUS}(\dot{\mathit{listen}}, \ddot{\mathit{listen}}) \mid \\
&\quad \mathit{Control}(\dot{\mathit{gain}}_i, \dot{\mathit{lose}}_i, \ddot{\mathit{talk}}, \dot{\mathit{listen}}) \mid \\
&\quad \mathit{IdleTrans}_1(\ddot{\mathit{gain}}_1, \ddot{\mathit{lose}}_1) \mid \mathit{IdleTrans}_2(\ddot{\mathit{gain}}_2, \ddot{\mathit{lose}}_2) \mid \\
&\quad \mathit{Car}(\dot{\mathit{talk}}, \ddot{\mathit{listen}})\ ) \quad \text{(where } i = 1, 2\text{)}
\end{aligned}
$$

Observe that, in contrast with the π-calculus example, we can change the behavior of the system by simply choosing other types of channels between the processes.
To make the example more realistic we could introduce more cars than just one. In the original π-calculus example this leads to changing the specification of all processes and introducing new links. In MoCha-π adding more cars is very easy. We just add more *Car* processes with parameters $\dot{\mathit{talk}}$ and $\ddot{\mathit{listen}}$. These processes then automatically compete among themselves to gain access to the channel-end pair. No other changes are required.

4.5.3 Mobile Agent

In Section 2.3, we illustrated the usage and benefits of mobile channels by giving an example that involves mobile Internet components. We discussed a scenario where a *mobile agent* queries several *XML information sources* in search for MoCha-bean prices, and sends these results to an interested component *U*. In particular, we discussed the movement of the agent from the location of XML component A to the one of component B.

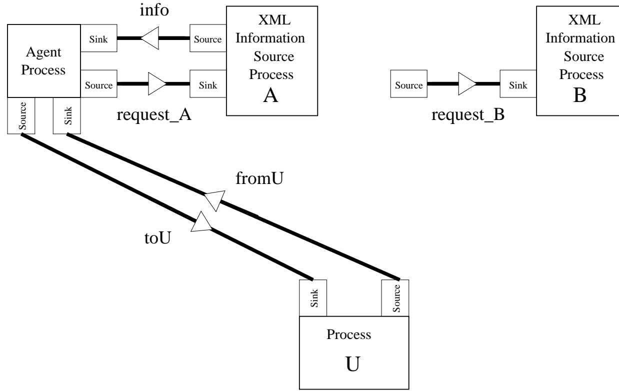


Figure 4.3: Mobile Agent Example with MoCha-π

We now give a possible implementation using our MoCha-π calculus. Figure 4.3 shows our particular implementation approach of the mobile agent example (the original example is given in Figures 2.4 and 2.5). All the components (*XML information sources*, *U*, and *the mobile agent*) are now MoCha-π processes. We took the locations from the original figures away, since in MoCha-π the locations of processes are implicitly determined by the channel connections that they have with other processes. Furthermore, we gave names to the channels that we use (which we will gradually introduce).

We start by giving the MoCha-π specification of the XML processes. Just like in the mobile phones example, we use a convenient notation where we denote the source-end of a *channel* as $cha\underset{\cdot}{n}nel$ and the sink-end of the same channel as $cha\ddot{n}nel$.

$$
\begin{aligned}
&XML_A(requ\ddot{e}st_A) \stackrel{def}{=} new\ data_A\ (requ\ddot{e}st_A \downarrow \\
&\quad .requ\ddot{e}st_A?(source).source \downarrow .source!\langle data_A\rangle.source \uparrow \\
&\quad .requ\ddot{e}st_A \uparrow);XML_A(requ\ddot{e}st_A)
\end{aligned}
$$

The XML process A has an incoming channel *request_A*. The sink-end of this channel is given to it as a process parameter. Our process A connects to this channel-end and waits for a request to arrive (by taking a value from the sink-end). A request comes together with a source-end where to write the requested information to. Process A connects to this source-end, writes the requested information to it, and then disconnects from it. Then, it finishes its execution by disconnecting from the $requ\ddot{e}st_A$ sink-end. Afterward, it invokes itself and the cycle is repeated. The

specification of XML information source B is equal to process XML_A, except that we replace all "$_A$" by "$_B$".

The process U has an incoming channel toU and a outgoing channel fromU. In this particular implementation, process U knows the source channel-ends of the request channels (which are given as parameters). It sends these channel-end references to the agent each time it wants to acquire MoCha-beans price information from these particular information sources. We give its specification:

$$
\begin{aligned}
&U(\dot{\mathit{fromU}}, \ddot{\mathit{toU}}, \dot{\mathit{request_A}}, \dot{\mathit{request_B}}) \stackrel{def}{=} \dot{\mathit{fromU}} \downarrow .\ddot{\mathit{toU}} \downarrow \\
&\quad .\dot{\mathit{fromU}}!\langle \dot{\mathit{request_A}} \rangle.\ddot{\mathit{toU}}?(\mathit{data}) \\
&\quad .\dot{\mathit{fromU}}!\langle \dot{\mathit{request_B}} \rangle.\ddot{\mathit{toU}}?(\mathit{data}) \\
&\quad .\dot{\mathit{fromU}} \uparrow .\ddot{\mathit{toU}} \uparrow
\end{aligned}
$$

Initially, process U connects to the given channel-ends. After which, it starts a request for information by writing the $\dot{\mathit{fromU}}$ channel-end to the source-end of the $\mathit{request_A}$ channel. It then waits for the result by trying to take from the $\ddot{\mathit{toU}}$ sink channel-end. Afterward, it repeats this scheme for getting information from XML information source B. Finally, process U terminates by disconnecting from the $\dot{\mathit{fromU}}$ and the $\ddot{\mathit{toU}}$ channel-ends.

The *agent* is linked with process U by two channels (toU and fromU). It receives the source-end of the first ($\dot{\mathit{toU}}$) and the sink-end of the second ($\ddot{\mathit{fromU}}$) as process parameters. We give its specification:

$$
\begin{aligned}
&\mathit{Agent}(\dot{\mathit{toU}}, \ddot{\mathit{fromU}}) \stackrel{def}{=} \mathit{new}(\dot{\mathit{info}}, \ddot{\mathit{info}})(\dot{\mathit{toU}} \downarrow .\ddot{\mathit{fromU}} \downarrow \\
&\quad .\ddot{\mathit{fromU}}?(\mathit{source}).\mathit{source} \downarrow .\mathit{source}!\langle \dot{\mathit{info}} \rangle.\mathit{source} \uparrow \\
&\quad .\ddot{\mathit{info}} \downarrow .\ddot{\mathit{info}}?(\mathit{data}).\dot{\mathit{toU}}!\langle \mathit{data} \rangle.\ddot{\mathit{info}} \uparrow \\
&\quad .\dot{\mathit{toU}} \uparrow .\ddot{\mathit{fromU}} \uparrow \ \mid \mathit{FIFO}(\dot{\mathit{info}}, \ddot{\mathit{info}}));\mathit{Agent}(\dot{\mathit{toU}}, \ddot{\mathit{fromU}})
\end{aligned}
$$

At initialization, the agent connects to the given channel-ends. It then acquires a source channel-end by taking it from the $\ddot{\mathit{fromU}}$ sink channel-end. This source is either the $\dot{\mathit{request_A}}$ or $\dot{\mathit{request_B}}$ channel-end send by process U. The agent connects to this source and writes the source-end of a FIFO channel that the agent created itself. After disconnecting from this source-end, it connects to the sink-end of the FIFO channel to await the requested data. After receiving the data via channel-end $\ddot{\mathit{info}}$, it writes it to channel-end $\dot{\mathit{toU}}$. The agent terminates by disconnecting from all connected channel-ends. Afterward, it invokes itself and the cycle is repeated.

Finally, we set up the system. The system process creates all other processes including the channels (with exception of the channel created by the agent process). We give its specification:

$$
\begin{aligned}
&\mathit{Sys} \stackrel{def}{=} \mathit{new}(\dot{\mathit{request_A}}, \ddot{\mathit{request_A}}, \dot{\mathit{request_B}}, \ddot{\mathit{request_B}}, \\
&\quad \dot{\mathit{toU}}, \ddot{\mathit{toU}}, \dot{\mathit{fromU}}, \ddot{\mathit{fromU}}) \\
&\quad (\ \mathit{FIFO}(\dot{\mathit{request_A}}, \ddot{\mathit{request_A}}) \mid \mathit{FIFO}(\dot{\mathit{request_B}}, \ddot{\mathit{request_B}}) \mid \\
&\quad \mathit{FIFO}(\dot{\mathit{toU}}, \ddot{\mathit{toU}}) \mid \mathit{FIFO}(\dot{\mathit{fromU}}, \ddot{\mathit{fromU}} \mid \\
&\quad \mathit{XML_A}(\ddot{\mathit{request_A}}) \mid \mathit{XML_B}(\ddot{\mathit{request_B}}) \mid \\
&\quad U(\dot{\mathit{fromU}}, \ddot{\mathit{toU}}, \dot{\mathit{request_A}}, \dot{\mathit{request_B}}) \mid \mathit{Agent}(\dot{\mathit{toU}}, \ddot{\mathit{fromU}}))
\end{aligned}
$$

Observe that, just like in the previous examples, we can change the behavior of the system by simply choosing other types of channels between the processes.

4.6 Conclusions and Related Work

In this chapter we presented MoCha-π, an exogenous coordination calculus based on mobile channels. A novelty of our calculus is in the fact that channels are not just links but special kinds of processes. This allows us to have user defined channel types without having to change the rules of the calculus itself. Our calculus provides anonymous communication; the communicating processes do not know each other. This combined with the fact that we can specify our own channel types, gives MoCha-π the property of placing any type of channel between processes without them knowing how different channel types affect their behavior; yielding exogenous coordination. Another novelty is the fact that our calculus treats channels as resources. Processes must compete with each other in order to gain access to a particular channel. This makes MoCha-π a more realistic model of existing systems.

The channels of MoCha-π are very general in the sense that we can specify channels with a user defined number of ends and corresponding user defined number of resources. Furthermore, the ends of a channel can be input, output, or both. All of this is specified in the definition of a particular channel type. For the MoCha framework, however, the channels are too general. That is why, in Section 4.4, we introduced a design pattern for specifying channels that are compatible with those of the framework. These are channels as specified in Chapter 2. They have exactly two ends, exactly two resources, and an end is either input or output to the channel but not both.

Besides MoCha-π there are other calculi that model distributed systems; see [FPT00] for an overview. Two well-known calculi are the *Distributed π-calculus* [HR98] and the *Ambient* [CG00] calculus. The *Distributed π-calculus* is an extension of the π-calculus with an explicit notion of location. Channel communication is synchronous and local; the processes involved in the communication must reside at the same location. In the *Ambient* calculus there is a message-driven communication that always takes place locally within a single ambient. An ambient is a bounded environment where processes cooperate. Both these calculi are good candidates to follow for extending the MoCha-π calculus if an explicit notion of location is desired.

MoCha-π is based on the *mobile channel* coordination primitive. The *KLAIM kernel* [NFP98] calculus is an asynchronous high-order process calculus that is based on the *Linda* [CG90] coordination paradigm. In KLAIM processes anonymously communicate via a shared multi-set of tuples. It is certainly possible to model all different MoCha channel types with this calculus. However, this cannot be done in an exogenous way; meaning that, the communicating processes of KLAIM do not have the option of leaving the desired coordination behavior up to the internals of a user-defined channel. Instead, they must implement such behavior themselves. Another modeling language for distributed systems based on channels is presented in [WS99].

In MoCha-π we don't explicitly model *process mobility*. Since we focus on coordination, we model only channel-end mobility. However, it would be nice to extend the calculus with the possibility of sending processes though channels, and therefore

explicitly model process mobility. One interesting approach that we can use is given in the work of *Barnes* and *Welch* about a model for mobile processes in *OCCAM-π* [BW04]. OCCAM-π is a small language that models processes that communicate through channels. It has many features of CSP [Hoa85] and the π-calculus, however, it is not really a calculus but more a programming language. It even has its own compiler, see [WMBW00]. In [BW04] the semantics for process mobility is:

- $c?x$, mobile process x arrives
- $x(...)$, process $x(...)$ runs from *somewhere* to *somewhere*
- $d!x$, mobile process x departs

where *somewhere* is either a process *start*, a *suspension-point*, or *termination*. We are interested in extending MoCha-π with these semantics. We are also interested whether it is possible to implement our calculus in the *OCCAM-π* language.

Chapter 5

Channel-based Semantics for Component Based Software

In the two previous chapters we defined semantics from the point of view of mobile channels. In this chapter, we give a semantics from the point of view of the components that use them. We do this by presenting a coordination model for component-based software systems based on the notion of mobile channels and define it in terms of a compositional trace-based semantics. 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, independent and exogenous manner.

5.1 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* [BRJ99] 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[RSRS99], which is an integration of the architectural description language *ROOM*[SGW94] into UML.

In this chapter we present and advocate a coordination model for component-based software that is based on mobile channels and describe it in terms of a transition system. 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.

This chapter is organized as follows. In Section 5.2 we discuss components, their interface, several coordination mechanisms for their composition, and present our rationale for a model based on channels. In Section 5.3, we give a compositional trace-based semantics for our model. Finally, in Section 5.4, we end with a discussion. We leave the conclusions and related work for after the introduction of the Java implementation in Chapter 8 (see Section 8.3).

5.2 Components and their Composition

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

5.2.1 Component-Based Software

In **Component-Based Software** systems are built out of *components* and *connections* among them. In Figure 5.1 an example of such a system is given. It consists of four components and six (one-way) connections among them (arrows). This simple system allows users to obtain data from a database. Component A collects user-requests, and sent them to the **control**-component, which, in turn, checks whether the user can have access to it. If access is denied, the component sends an error message to the **user interface**-component, otherwise it sends the request to the **database**-component. This last component sends its data to the **user interface**-component. If necessary, the data is sent first through the **translation**-component. Component D must then somehow know which connection to use. A nice solution here is to use a mobile channel. Another solution is letting component C tell the **database**-component which connection to use.

Why use component-based software? There are several advantages, all due to the fact that the implementation of a component is not relevant for the functionality of the system; only its interface is. The most important advantages are:

- **Easy building**. To build a system, it is enough to specify the interfaces of the components and the connections among them. Then, the components can be implemented, and/or reused from other systems, and/or bought from vendors.

  - **Reuse**. Components created for other systems can be reused, without trouble, for a new system when they have the same required interface. There is no need to implement them again.
  - **Buying components**. Components can be bought from vendors to save time. Many standard components can be bought these days. Also buying tailor made components can be a good idea when creating them is more costly.

- **Adaptability**. Components with new features can easily replace existing ones, making the system easy to adapt. For example, in the system of Figure 5.1, component A could be replaced by a new component E that provides a better user interface with new features, without having to change major parts of the system.

- **Fast time-to-market**. Because of the advantages above, systems (and improved versions of the system) can be developed more rapidly. This is perhaps, the most important reason for companies to use components.

5.2.2 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.

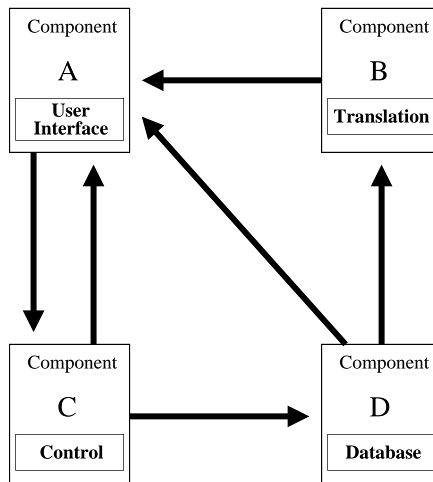


Figure 5.1: A Component-Based System.

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 5.3 we express the external observable behavior of a component in terms of a compositional trace-based semantics.

5.2.3 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* [And91].

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*) [JMQ], a package based on the Java Message Service (*JMS*) [JMS] open standard. The Microsoft Message Queuing Services [Hay99] for COM+ [COM+], 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* [JB], 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* [CG90], 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 [FHA99], a powerful Jini service from Sun, is an example of a shared data space that is being used for components. *Lime* [MPR03] (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 2.1, is a one-to-one connection that offers exactly two ends to components: usually $\langle$*source*, *sink*$\rangle$ for most common channel-types, but also $\langle$*source*, *source*$\rangle$ and $\langle$*sink*, *sink*$\rangle$ for special types (see Section 2.4). A component can write by inserting values to the *source*-end, and take 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* [HBB00], *CSP for Java* [Wel01], both based on the *CSP* model [Hoa85], and *Pict* [PT97], 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* [ABBG02] (see Chapter 6), *JCSP.net* [VW02, AFW02] (see Section 6.5.1) and *Nomadic Pict* [WS99], 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 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).

5.3 A Semantic Approach

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 Chapter 8.

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 [ABB00a]. Our one-to-one channels can still be composed into many-to-many connectors, while preserving these prerequisites for compositionality [Arb04, Arb02].

Physical movement of channel-ends, see Section 2.2.2, 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 in the implementation given in Section 8.2 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 of its 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.

5.3.1 Component Transition System

Definition 5.3.1 *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* $e \in Source \cup Sink$.

5.3.2 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.

5.3.3 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`

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

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`

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

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`

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

where $\sigma' = \sigma[c/id]$, $\gamma' = \gamma[\perp/s][\perp/t]$ and $\langle s,t\rangle \notin \mathit{Chan}$ and $\mathit{Chan}' = \mathit{Chan} \cup \{\langle s,t\rangle\}$. 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`

$$\frac{\sigma(id) \xrightarrow{\nu\langle \mathit{Comp}_i, K\rangle} c}{\langle \sigma, \gamma, \mathit{Chan}\rangle \xrightarrow{\langle id,id',K\rangle} \langle \sigma', \gamma, \mathit{Chan}\rangle}$$

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.

5.3.4 Trace Semantics

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

$$\{\theta \mid \langle a_0, K\rangle \xrightarrow{\theta}\!\!\!\!\rightarrow\},$$

where a_0 denotes the initial (abstract) state of *Comp* and $\xrightarrow{\theta}\!\!\!\!\rightarrow$ 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*:

$$\begin{array}{lll} P(e \downarrow id.\theta, id, K) & = e \downarrow .P(\theta, id, K \cup \{e\}) & \\ P(e \downarrow id'.\theta, id, K) & = P(\theta, id, K) & id \neq id' \end{array}$$

- *disconnect*:

$$\begin{array}{lll} P(e \uparrow id.\theta, id, K) & = e \uparrow .P(\theta, id, K \setminus \{e\}) & \\ P(e \uparrow id'.\theta, id, K) & = P(\theta, id, K) & id \neq id' \end{array}$$

- *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*:

$$P(\langle s, t, v, ¿\rangle.\theta, id, K) = \begin{cases} s!v.P(\theta, id, K) & s \in K \\ t¿v.P(\theta, id, K) & t \in K \\ P(\theta, id, K) & s, t \notin K \end{cases}$$

- *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 5.3.1 *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 [Hen88].

5.4 Discussion

In this chapter we used a simple labelled transition system to model the observable interaction between the components and the channels of a system. The difference with the MoCha-π calculus of Chapter 4 is that we take a higher level of abstraction concerning the mobile channels. In MoCha-π we focus on mobile channels by explicitly describing their internal behavior as π-calculus processes. In the semantics presented in this chapter we abstract away from this internal behavior and focus more on components and the actions they perform on channel-ends. From the point of view of a component any interaction with a channel is always synchronous, therefore, it was sufficient for us to introduce only the synchronous channel type in our semantics. Other types can be constructed by using the synchronous type in combination with components that implement certain desired channel behavior. In MoCha-π, we also use the synchronous (π-calculus) channel type to construct other types. The difference is that instead of hiding the extra behavior in a component, MoCha-π explicitly describes this behavior as a process.

Our component based software 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.

In Chapter 8, we continue with our component coordination model by discussing its implementation in the Java language. At the end of that chapter we also discuss other work related to our approach.

Part III

Implementation

Chapter 6

The MoCha Middleware: API and Applications

In this chapter we present the MoCha middleware: a middleware for distributed communication and coordination of components. With this middleware we implement the mobile channels of the MoCha framework and their MoCha-π semantics. We have divided the explanation of the middleware in two chapters. In this first chapter, we take the point of view of a distributed system developer who wants to use the middleware but does not want to know anything about its internal implementation details. We leave these technical details for Chapter 7. Therefore, in this chapter, we discuss the main features of the Application Programming Interface of the MoCha middleware. We provide examples of how to use the middleware by giving simple producer/consumer components that are coordinated in various ways. We discuss several applications of the middleware: Component Based Software (like In-Home networks), Web Services, and Peer-to-Peer networks (like P2P file-transfer applications). Finally, we conclude with a survey on middleware software related to MoCha.

6.1 Introduction

A *distributed system* is a collection of autonomous computers linked by a network and equipped with distributed system software [CDK94]. The distributed system software enables the comprising computers to coordinate their activities and to share system resources. A well-developed distributed system software provides the illusion of a single and integrated environment although it is actually implemented by multiple computers at different locations [TB00].

Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system. In doing so, it provides a higher-level building block for programmers than, for example, low-level sockets that are provided by the operating system [Bak05].

We implemented the mobile channels of the MoCha framework by developing such a middleware. We call this distributed software package *MoCha*: a middleware for distributed communication and coordination of components. The MoCha middleware comes in three different flavors: *MoCha*, *easyMoCha*, and *chocoMoCha*. Distributed applications can use any of the three MoCha middleware versions. We can see in Figure 6.1 that these three versions are build on top of a *MoCha core* layer. This core layer itself is build on top of Java *RMI* [RMI]. We leave the details of the two lower layers for our implementation topic in Chapter 7.

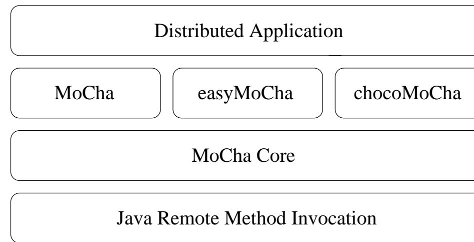


Figure 6.1: MoCha and its Three Different Flavors

The “plain” *MoCha* version is the most basic one of all three. Except for the connect and disconnect operations (see Section 2.2.1) it provides all the features needed to properly work with mobile channels. The MoCha user interface is small but powerful. However, it is meant for expert programmers since it is so compact and, one could say, low-level for object-oriented standards. In this version there is no automatic internal update of channel-end references. This means that, when a component physically moves a channel-end from one location to another in the network, it gets a new reference that points to the moved channel-end. However, all “old” references that other components may posses become invalid, or dangling. We don’t update the “old” references for two reasons. The first reason is security, we can assume that the component moving the channel-end may want to have exclusive knowledge of it. Otherwise, it can spread the new reference around to the other components. The second reason is that, updating the references that components may have is a costly operation. Furthermore, MoCha does not know anything about the components that uses a particular reference. Therefore, the components themselves are responsible for implementing an efficient protocol that updates these dangling references, if desired.

The *easyMoCha* version offers all the functionality of the “plain” MoCha version plus a richer and easier to use interface. Furthermore, this version has a build-in protocol for taking care of invalid dangling channel-end references; this means that components don’t have to worry about a channel-end reference becoming invalid, once a component gains a reference to a channel-end it will always remain valid. This version is especially made for those that either find it difficult to work with the “plain” MoCha version, want a more object-oriented friendly interface, or don’t want to be concerned with dangling references.

The *chocoMoCha* version, which stands for *ch*annel c*o*nne*c*ti*o*n, has the same functionality as the previous version with the addition that it implements the connect

and disconnect operations as well. Components have to successfully connect to a channel-end first before being able to use it. We chose to give it the same rich user interface as the easyMoCha version, plus the automatic internal reference update protocol. We think that this brings the middleware at the right level of programming abstraction. Furthermore, with the first two versions the communication is *many-to-many*, while with chocoMoCha we can have *one-to-one* communication due to the fact that components have exclusive access to channel-ends. Naturally, in the "plain" MoCha version there one-to-one communication is also possible by moving a channel-end and keeping its identity secret to other components. However, this exclusive communication is not guaranteed like in chocoMoCha. A malicious component may secretly obtain the reference to a channel-end and start using it at any time.

The chocoMoCha version is the one that adheres more to the theory of mobile channels that is introduced in this thesis. We shall, therefore, concentrate on and mainly present this version.

Next, in Section 6.2, we discuss the Application Programming Interface of the chocoMoCha middleware. In Section 6.3, we give some examples of how to use the middleware. In Section 6.4, we discuss some applications of chocoMoCha. And finally, in Section 6.5, we conclude with related work.

6.2 The Application Programming Interface (API)

We present and discuss the Application Programming Internal (API) of choco-MoCha; i.e. its user interface. Our aim is to give an overview of the main features and explain the choices that we made, rather than presenting the full technical details of the API. Therefore, we concentrate on the important features and abstract away from the less relevant ones. For full details and more information about the API, we refer to the MoCha middleware manual [Gui05].

6.2.1 Location and Keys for Components

The MoCha middleware doesn't know anything about its environment. It does not know anything about the notion of *location* that is used in the distributed system it is running. It does not know anything about the security or priority policies the network may have on using channel-ends. It does not even know what kind of entities are using the channel-ends; whether they are components, threads, processes, active objects, etc. Therefore, the middleware provides certain constructs for everything it needs to know itself. The chocoMoCha version needs to know what a *location* is, and some way of identifying the owner of a channel-end. For this purpose, the middleware offers two classes: `MoChaLocation` and `ComponentKey`.

MoChaLocation

The MoCha middleware provides the class `MoChaLocation` to the user in order to identify specific logical execution spaces. A reference to an instance of this class is, thus, a *location reference* that the MoCha middleware understands and can work with. Such a reference can be passed on to others through channels, if desired.

method	parameters	return
Class:**MoChaLocation**		
constructor	()	new instance
equals	(MoChaLocation loc)	boolean
createChannel	(String type)	ChannelEnd[]
Class:**ComponentKey**		
constructor	()	new instance
equals	(ComponentKey key)	boolean

Table 6.1: Locations and Keys

The user is free in defining what a location means in his distributed system. He can either: (1) create one instance of `MoChaLocation` per Java Virtual Machine (JVM) [Java]. (2) He can create many instances of `MoChaLocation` per JVM. Or (3), he can create and share one instance of `MoChaLocation` among many JVM's. The first is the standard way of using the `MoChaLocation` instances in a distributed system. The second way is useful, for example, for testing a system in one single JVM. The third is useful when dealing with unstable locations that often suddenly disconnect from the network. The middleware puts critical internal objects only at the JVM's where a location has actually been created. By sharing a `MoChaLocation` with an unstable location we make sure that the middleware is not affected by a suddenly disconnect (or crash) by this location.

As shown in Table 6.1, the `MoChaLocation` class has one constructor and two public methods: `createChannel` and `equals`. To create an instance of this class we use the constructor:

```
MoChaLocation loc = new MoChaLocation();
```

We can check whether two locations are equal by using the `equals` method:

```
boolean result;
MoChaLocation loc = new MoChaLocation();
MoChaLocation loc2 = new MoChaLocation();
result = loc.equals(loc);
result = loc.equals(loc2);
```

The first comparison evaluates to `true` and the second to `false`. Usually, a component uses this method to check if a given location is equal to one it already has.

It is possible to create a new channel by using the `createChannel` method. In Section 6.2.2, we show a much nicer way of creating channels. We also list there the different channel types that the middleware implements. We give an example of how to create a new channel using an instance of `MoChaLocation`:

```
MoChaLocation loc = new MoChaLocation();
ChannelEnd[] channel = loc.createChannel("Synchronous");
```

The method returns an array of `ChannelEnd`, a class we define in Section 6.2.4. The length of this array is two, with `chan[0]` being the first channel-end and `chan[1]` the second one.

ComponentKey

The MoCha middleware provides the class `ComponentKey` to the user in order to identify components. With an instance of the class `ComponentKey` the components can identify themselves to channel-ends in order to get access to their I/O operations, move operation, and for the connect and disconnect operations (operations which we discuss further on).

The user is free in defining what a component is; e.g. a thread, a group of threads, active objects in a package, etc. Once he defines what a component is, he is also free in giving more than one key to one component, or share one key among many components. The first case, for example, is useful when a component has multiple interfaces. Each interface represents another aspect that we want to separate from the rest by giving another access-key to its channel-ends. The second case, for example, is useful when we have a group of collaborating components that all require simultaneous access to the shared channel-ends. However, in most of the cases, the standard thing to do is to give each component exactly one key. This way we also guarantee the *one-to-one* communication property between components. Nevertheless, with this second technique we can also have a *one-to-many*, *many-to-one*, or a *many-to-many* communication, if desired. An example of this is given in Section 6.3.3.

As shown in Table 6.1, the `ComponentKey` class has one constructor and one public method: `equals`. To create an instance of this class we use the constructor:

```
ComponentKey key = new ComponentKey();
```

We can check whether two keys are identical by using the `equals` method:

```
boolean result;
ComponentKey key = new ComponentKey();
ComponentKey key2 = new ComponentKey();
result = key.equals(key);
result = key.equals(key2);
```

The first comparison evaluates to `true` and the second to `false`. Usually, a component uses this method to check if a given key is equal to one it already has.

6.2.2 Mobile Channels: Creation and Types

In Section 6.2.1 we explained how to create a new channel using an instance of the class `MoChaLocation`. However, the easiest and more convenient way of creating a new channel in chocoMoCha is by using an instance of the class `MobileChannel`. Before explaining all the details of this class, we first list all the eleven channel types that the middleware offers to its users. All MoCha versions implement these types.

We already explained their behavior in Section 2.4. Therefore, we just list the types and their names in the MoCha middleware (which is a string):

- *Synchronous channel.* "Synchronous".
- *Lossy synchronous channel.* "LossySynchronous".
- *Filter (synchronous) channel.* "Filter [pattern]".
  Example: "Filter java.lang.Integer java.lang.Double mypackage.myclass".
- *Synchronous drain channel.* "SynchronousDrain".
- *Synchronous spout channel.* "SynchronousSpout".
- *Asynchronous unbounded FIFO channel.* "FIFO".
- *Asynchronous bounded FIFO (FIFO n) channel.* "FIFOn [number]".
  Examples: "FIFOn 1", "FIFOn 10", and "FIFOn 11031974".
- *Asynchronous drain channel.* "AsynchronousDrain".
- *Asynchronous spout channel.* "AsynchronousSpout".
- *Drain channel.* "Drain".
- *Spout Channel.* "Spout".

Interesting to notice are the string names of the Filter and the FIFO n channel types. The first channel type allows the user to specify a filter pattern. This can be any Java class including non-standard ones made by users. If at a given location, one of the classes specified in the pattern does not exist, the channel simply discards this class from the filter for that particular location. The second channel type allows the user to specify a capacity. This capacity is a non-zero positive integer with a maximum specified by the Java `Integer` class constant `MAX_VALUE`. Currently this value is $2^{31}-1$. We chose to include both the pattern and the capacity as part of the string name of the channel type. We did this for uniformity with the other channel types. Since almost everything is easily convertible to a string in the Java language, this does not impose any problems for the user. The middleware then parses the string and automatically extracts the pattern or capacity out of it.

Class:**MobileChannel**		
Attributes: ChannelEnd ce1, ChannelEnd ce2		
method	**parameters**	**return**
constructor	(MoChaLocation loc, String type)	new instance
constructor	()	new instance
equalsChannel	()	boolean

Table 6.2: Mobile Channels

The `MobileChannel` class has two constructors and one method `equalsChannel` (see Table 6.2). The first constructor is meant for creating new channels. The type of

the channel is given as a string. As mentioned above, patterns and capacities are also specified in this string. After creation, the public attributes of type `ChannelEnd` refer to the two ends of the new created channel. We discuss the abstract class `ChannelEnd` in Section 6.2.4. We give an example of channel creation:

```
MoChaLocation my_computer = new MoChaLocation();
MobileChannel channel = new MobileChannel(my_computer, "FIFOn 1");
```

Afterward, the attribute `chan.ce1` refers to the source-end of the channel and `chan.ce2` to its sink-end. Unlike with the `MoChaLocation` class, where we get an array of `ChannelEnd` back after creating a new channel, with the `MobileChannel` class there is no need to work with arrays and we immediately get a nice place holder for the ends of the channel we just created.

The second constructor is meant for creating only a place holder for channel-ends. This constructor creates an instance of `MobileChannel`, but does not create a new channel. Instead, the attributes are initially `null`. The user, then, can assign received channel-end references to them.

Therefore, it makes sense to provide a method that checks whether the ends represented by the attributes belong to the same channel. This is the purpose of the `equalsChannel` method. For example:

```
boolean result;
MobileChannel channel = new MobileChannel();
channel.ce1 = received_Source;
channel.ce2 = received_Sink;
result = channel.equalsChannel();
```

In this case the ends are received from the outside, via channels, so there is no a priori way of knowing what the result is after the execution of the method `equalsChannel`.

6.2.3 Source & Sink Channel-End

The `SourceEnd` and the `SinkEnd` are the most important classes of the MoCha middleware. They implement all the conceptual operations we discussed in Section 2.2.1. These operations are divided in three groups: the *topology operations*, the *I/O operations* , and the *inquiry operations.*

Notice that, neither of the two classes have a constructor. This is done on purpose. From our point of view, an end always belongs to a channel. Therefore, we cannot allow users to create single loose channel-ends that are not conceptually, and internally, related to a channel. The only way to create a new channel-end is by creating a complete new channel using either the class `MoChaLocation` or `MobileChannel`.

The methods that implement the topology and the I/O operations have *time-outs*[1]. This means that a component can define a certain length of time it is willing to wait until an operation succeeds. If an operation does not succeed within this

[1]except the method that implements the disconnect operation.

given time, it gets canceled. In Chapter 8 we explicitly use these time-outs in our channel-based component model. However, in this chapter, we do not consider them. This omission simplifies all the explanations and examples of the MoCha middleware. We implicitly assume that all the topology and I/O operations have a time-out that is set to *infinity*; i.e. components wait until an operation succeeds, not matter how long this takes.

The Topology Operations

Class:**SourceEnd**, **SinkEnd**		
method	**parameters**	**return**
connect	(ComponentKey key)	void
disconnect	(ComponentKey key)	void
move	(MoChaLocation loc, ComponentKey key)	void

Table 6.3: SourceEnd & SinkEnd, the Topology Operations

Table 6.3 gives an overview of the *topological operations*. Before being able to "use" a channel-end, a component must first successfully `connect` to this particular channel-end. The method terminates if no other component is currently connected to this channel-end, or this component is already connected to this end. In all other cases the component must wait until this channel-end becomes available and, therefore, the method suspends until this is the case. For identifying a component chocoMoCha uses a special *key* as explained in Section 6.2.1. Once connected, a component can perform I/O operations and the topological `move` operation. For query operations a component needs not be connected to a channel-end. We decided not to require a component to be connected to a particular channel-end, because a component may need to perform a query operation on a channel-end before wanting to connect to it. Also, not connected parties may need actual information about a particular channel-end.
A typical example of `connect/disconnect` usage:

```
source.connect(my_ID);
source.operation1();
source.operation2();
...
source.disconnect(my_ID);
```

The `move` operation *physically* moves a channel-end from one location to the given `MoChaLocation` in the network. Conceptually, this operation is not needed since a reference to a channel-end never becomes invalid in chocoMoCha, even if the target channel-end is in another `MoChaLocation` than the component performing an operation on it. However, it makes sense to move the channel-end for efficiency reasons. For example, for trying to keep the channel-end at the location it currently is most intensively used, or preventing a channel-end to execute in a location with low resource capacity.
An example of moving a channel-end is:

```
sink.connect(my_ID);
sink.move(my_Computer, my_ID);
sink.operation();
...
sink.disconnect(my_ID);
```

The I/O Operations

Table 6.4 gives an overview of the *I/O operations*. The most important I/O methods are `write`, `take` and `read`, where `read` is the non-destructive version of `take`. Besides writing, reading and taking *data objects*, components can also send MoCha *channel-ends* and *locations* to each other through channels themselves. To ensure the strong encapsulation of components (as explained in Section 8.1), no object from outside should be able to refer to an object inside a component. Therefore, no object that is transmitted through MoCha channels refers to objects inside components. To accomplish the MoCha middleware makes a so called deep copy [Eck98] of every object written to a source-end. This deep copy ensures that not only the written object is copied, but also every other object it may refer to is also copied. Therefore, a whole tree of objects is copied if necessary. The middleware takes care of possible cycles in these trees. For this purpose, we require every written object to be of the special Java type `Serializable`; this means that, the object can be transformed into a string for communication through networks. Naturally, the MoCha channel-ends and locations are the only objects that are not (deep) copied. Otherwise, we would have implicit channel(-end) and location creation as a side effect.

Class:**SourceEnd**		
method	**parameters**	**return**
write	(Serializable object, ComponentKey key)	void
writeBoolean	(Boolean ob, ComponentKey key)	void
writeByte	(Byte ob, ComponentKey key)	void
writeChannelEnd	(ChannelEnd ce, ComponentKey key)	void
writeCharacter	(Character ob, ComponentKey key)	void
writeDouble	(Double ob, ComponentKey key)	void
writeFloat	(Float ob, ComponentKey key)	void
writeInteger	(Integer ob, ComponentKey key)	void
writeLong	(Long ob, ComponentKey key)	void
writeMoChaLocation	(MoChaLocation ob, ComponentKey key)	void
writeShort	(Short ob, ComponentKey key)	void
writeString	(String ob, ComponentKey key)	void

Table 6.4: SourceEnd, the I/O Operations

The `write`, `take`, and `read` methods can be seen as untyped, because in the Java language every object is an instance of the basic class `Object`. This means that, the MoCha middleware does not care about the type of the data it transports. The components themselves are responsible for passing on the right types of data, and to figure out what data types they receive. We chose to do this because most distributed systems work with dynamic data types; these are types that are

dynamically created during the runtime of the system and are not fixed a priori. For example, it is often the case that, when a computer joins a specific truly distributed network it receives class definitions for specifying the type of the data it needs to communicate with others in the network. In Java RMI [RMI], they have an automatic feature for this called *dynamic class loading*.
An example of untyped `write` and `take` operations is:

Writer:
```
source.connect(writer_ID);
source.write(new Integer(777), writer_ID);
source.disconnect(writer_ID);
```
Taker:
```
Object result;
Integer int;
sink.connect(taker_ID);
result = sink.take(taker_ID);
if (result instanceof Integer) { int = (Integer) result}
sink.disconnect(taker_ID);
```

In this simple example, the `taker` is responsible for checking whether the object `result` is of type `Integer`, before it can safely use it.

Class:**SinkEnd**		
method	**parameters**	**return**
take	(ComponentKey key)	Object
read	(ComponentKey key)	Object
takeBoolean	(ComponentKey key)	Boolean
takeByte	(ComponentKey key)	Byte
takeChannelEnd	(ComponentKey key)	ChannelEnd
takeCharacter	(ComponentKey key)	Character
takeDouble	(ComponentKey key)	Double
takeFloat	(ComponentKey key)	Float
takeInteger	(ComponentKey key)	Integer
takeLong	(ComponentKey key)	Long
takeMoChaLocation	(ComponentKey key)	MoChaLocation
takeShort	(ComponentKey key)	Short
takeString	(ComponentKey key)	String

Table 6.5: SinkEnd, the I/O Operations

However, for some distributed systems it makes sense to work with typed data objects for static verification purposes. Therefore, chocoMoCha also provides typed `write` and `take` methods. These methods internally call the basic ones. The user can always make his own typed methods by wrapping the basic ones as well. The behavior of the typed `take` method is that, if the taken data from the sink channel-end is not of the right type, it loses the data and waits for the next available one.
An example of typed `write` and `take` operations follows:

Writer:
```
source.connect(writer_ID);
source.writeInteger(new Integer(333), writer_ID);
source.disconnect(writer_ID);
```
Taker:
```
Integer int;
sink.connect(taker_ID);
int = sink.takeInteger(taker_ID);
sink.disconnect(taker_ID);
```

In this simple example, the `taker` is sure to always get an instance of the class `Integer` from the sink channel-end.

The Inquiry Operations

Table 6.6 lists the *inquiry operations* of both the `SourceEnd` and the `SinkEnd` classes. As mentioned above, components need not be connected to a particular channel-end to perform an inquiry operation.

Class:**SourceEnd, SinkEnd**		
method	**parameters**	**return**
empty	()	boolean
equals	(ChannelEnd ce)	boolean
equalsChannel	(ChannelEnd ce)	boolean
full	()	boolean
getStatus	()	Boolean[]
isSinkEnd	()	boolean
isSourceEnd	()	boolean

Table 6.6: SourceEnd & SinkEnd, the Inquiry Operations

The result of the `empty` or `full` method depends on the type of the channel and the end that it belongs to. For example, with a *synchronous* channel type a source-end always returns `true` on the `empty` method; when we are able to query the source-end there is no writer pending on a write operation. The sink-end returns `true` on the `full` method if there is a *write* operation pending on the source-end of the same channel; i.e. there is a value available from this channel. Another example, with the *FIFO n* channel type a source-end returns `true` on the `full` method if the channel has reached its capacity. The sink-end returns `true` on the `empty` method when there are no elements stored in the channel. The details regarding the other types are given in the manual [Gui05].

The `getStatus` method provides an atomic way of inquiring a channel-end. For example, if we want to know the `full` and `empty` status of a particular channel-end we can either call the two methods separately or call the `getStatus` method. The difference is that, with the first the status of the channel-end may change between the two method calls, while with the second the inquiry operations are performed at the same time preventing the channel-end from changing its status in the meanwhile. This atomicity is useful, for example, in those cases where many components are

querying (or using) the same channel-end at a given time. These components can cause such big delays between the separate queries of one particular component, that the status of the channel-end is more likely to have changed when the component finally succeeds in executing all of its query operations.

The `equals` method checks if a given channel-end is equal to `this` channel-end. Naturally, if we compare a sink with a source channel-end we get `false` as result. The `equalsChannel` method checks if the given end belongs to the same channel as `this` end.

We discuss the methods `isSinkEnd` and `isSourceEnd` in Section 6.2.4.

6.2.4 Channel-End

Sometimes it is convenient not to directly refer to a source or sink channel-end but to a more general class that does not specify the type of the end. Therefore, we provide an `abstract` class called `ChannelEnd`. An `abstract` class is a base class that presents only an interface for its derived classes. No `abstract` class can be instantiated. However, unlike the Java `interface` class, it can implement methods. For more information about `abstract` classes we refer to [Eck98]. The `ChannelEnd` class offers all the methods of the source and sink channel-ends (see Tables 6.4, 6.5, 6.3, and 6.6).

Our `MobileChannel` class (see Section 6.2.2) conveniently uses two attributes, `ce1` and `ce2`, of type `ChannelEnd`. Since the type of the channel determines whether we get two distinct or two equal channel-end types upon creation of a channel instance, it is not possible to make `ce1` of type `SourceEnd` and `ce2` of type `SinkEnd`, or vice versa. However, now with the `ChannelEnd` class we don't have this problem anymore. The attributes can both be of type `SourceEnd` or `SinkEnd`.

Using the `ChannelEnd` class is useful for cases when we are not interested in the precise type of the end, because the operations that we want to perform on them are implemented by both the sink and the source channel-ends. For example, imagine that our job is just to receive channel-ends and move them to Amsterdam.

```
while (true) {
   received_end = input_end.takeChannelEnd(my_ID);
   received_end.connect(my_ID);
   received_end.move(Amsterdam, my_ID);
   received_end.disconnect(my_ID);}
```

In this example, we don't care about the type of the channel-end.

In the cases that we use specific methods of either `SourceEnd` or `SinkEnd`, like `write` and `take`, we can use the `isSourceEnd` and `isSinkEnd` method to determine the right channel-end type:

```
received_end = input_end.takeChannelEnd(my_ID);
   received_end.connect(my_ID);
if (received_end.isSourceEnd()) {received_end.write("Source", my_ID)}
if (received_end.isSinkEnd()) {result = received_end.take(my_ID)}
   received_end.disconnect(my_ID);
```

It is also possible to cast the channel-end to the right type:

```
received_end = input_end.takeChannelEnd(my_ID);
if (received_end.isSourceEnd()) {
   received_end.connect(my_ID);
   source = (SourceEnd) received_end;
   source.write("Hello", my_ID);
   source.disconnect(my_ID);}
```

The difference is that once casted, we don't have to check the type anymore. The user is free in selecting his favorite way of working with instances of `ChannelEnd`.

6.3 Examples

After introducing the API in the previous section, we now give some small examples of how to use the MoCha middleware. Each example highlights a certain feature of the middleware. We show some Java code for almost every example. However, we omit most of the technical details, for we are only interested in showing the usage of the chocoMoCha middleware. For the working implementation of these examples and the technical details, we refer to the electronic manual [Gui05].

6.3.1 Producer/Consumer

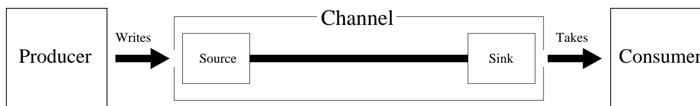


Figure 6.2: Producer/Consumer Example

We start with a simple system consisting of a producer and consumer linked together by a channel. We give the architecture of this system in Figure 6.2. The producer creates integer values and writes them to the source channel-end, while the consumer takes integers from the sink-end of the same channel. Our purpose with this first example is to show the exogenous coordination property of the middleware.

We implement the producer as a simple Java `thread` (see Figure 6.3). Upon initialization, the producer needs the following parameters: a source channel-end (`ChannelEnd so`), the number of writes to this source-end (`int writes`), and the length of time in between writes (`int wait`). The producer checks whether the given channel-end is a source-end, if not it terminates the execution by giving an error. For identification purposes, the constructor creates an instance of the chocoMoCha `ComponentKey class`. During execution, the producer connects to the given source-end, writes an integer to it as many times as specified by the `writes` parameter, and ends by disconnecting from the channel-end.

The consumer is also implemented as a Java `thread` (see Figure 6.4). Upon initialization, the consumer needs the following parameters: a sink channel-end

```
import cwi.sen3.chocoMoCha.*;
/** Producer, produces an Integer
* @author Juan Vicente Guillen-Scholten
*/
public class Producer extends Thread {
   ChannelEnd source;
   ComponentKey key;
   int times;
   int waittime;
   /** Constructor.
   * @param so the source-end to write to.
   * @param writes times that the producer writes a value.
   * @param wait time to wait in between writes (1000 = 1 second).
   */
   Producer (ChannelEnd so, int writes, int wait) {
      if (so.isSourceEnd())
      {source = so;} else {throw new ErrorException();}
      times = writes;
      waittime = wait;
      key = new ComponentKey();
   }// constructor
   public void run(){
      source.connect(key);
      for (int i = 0; i < times; i++) {
         sleep(waittime);// sleep between writes
         source.writeInteger(new Integer(i), key);
      }// done
      source.disconnect(key);
   }//run
} // end Producer
```

Figure 6.3: A Basic Producer

(`ChannelEnd si`), the number of takes from this sink-end (`int takes`), and the length of time in between takes (`int wait`). The consumer, also, checks whether the given channel-end is a sink-end, if not it terminates the execution by giving an error. For identification purposes, the constructor creates an instance of the choco-MoCha `ComponentKey class`. During execution, the consumer connects to the given sink-end, takes an integer from it as many times as specified by the `takes` parameter, and ends by disconnecting from the end.

We implemented this example in such a way that the channel type is given as a parameter while starting the system in the *command-line*. This is done to show that we can change the behavior of our system by simply choosing another type of channel in between the threads, without having to re-compile them or any other part of the application.

In Figure 6.5 we show the result of executing our example using three different types of channels in between the producer and the consumer. We delay the execution of the producer for one second after every successful write, and we delay the execution of the consumer for three seconds after every successful take. In the first run, we use a *synchronous* channel type. We see that both writes and takes are synchronized and happen atomically. In this case, the network is so fast that the times also coincide.

```
import cwi.sen3.chocoMoCha.*;
/** Consumer, takes an Integer.
* @author Juan Vicente Guillen-Scholten
*/
public class Consumer extends Thread {
   ChannelEnd sink;
   ComponentKey key;
   int times;
   int waittime;
   /** Constructor.
   * @param si the sink-end to take from.
   * @param takes times that the consumer takes a value.
   * @param wait time to wait in between takes (1000 = 1 second).
   */
   Consumer (ChannelEnd si, int takes, int wait) {
      if (si.isSinkEnd())
      sink = si; else {throw new ErrorException();}
      times = takes;
      waittime = wait;
      key = new ComponentKey();
   }// constructor
   public void run() {
      Integer res;
      sink.connect(key);
      for (int i = 0; i < times; i++) {
         sleep(waittime); // sleep between takes.
         res = sink.takeInteger(key);
      }// done
      sink.disconnect(key);
   }//run
} // end Consumer
```

Figure 6.4: A Basic Consumer

This is not always the case, because there is a delay possible between the write and the take operations due to the distance between the different machines in the network. However, this is fine as long as the operations succeed atomically; i.e. in this case, no write or take operation can succeed after its previous write/take pair finishes.

In the second run, we use an unbounded *FIFO* channel type. We see indeed that the write and take operations happen asynchronously from each other. This is not entirely the case in the third run. There we choose a FIFO channel type with a buffer capacity of three values. Since the producer is three times faster than the consumer, we see that the first three integers are asynchronously written every second into the channel. However, after this, the buffer is full and the next writes must wait for a take to happen. We see that the next writes coincide with the first three takes.

6.3.2 Producer/Producer and Consumer/Consumer

Sometimes, we are not interested in transferring data between components but only in coordinating them. For example, when we cannot change the components them-

```
Output listing.
Producer writes every second,
Consumer takes every 3 seconds.

With a Synchronous channel.
Producer:
Producer p1 writes Integer 0 to the channel [time 17:26:12].
Producer p1 writes Integer 1 to the channel [time 17:26:15].
Producer p1 writes Integer 2 to the channel [time 17:26:18].
Producer p1 writes Integer 3 to the channel [time 17:26:21].
Producer p1 writes Integer 4 to the channel [time 17:26:24].
Consumer:
Consumer c1 takes Integer 0 from the channel [time 17:26:12].
Consumer c1 takes Integer 1 from the channel [time 17:26:15].
Consumer c1 takes Integer 2 from the channel [time 17:26:18].
Consumer c1 takes Integer 3 from the channel [time 17:26:21].
Consumer c1 takes Integer 4 from the channel [time 17:26:24].

With an unbounded FIFO channel.
Producer:
Producer p1 writes Integer 0 to the channel [time 17:56:32].
Producer p1 writes Integer 1 to the channel [time 17:56:33].
Producer p1 writes Integer 2 to the channel [time 17:56:34].
Producer p1 writes Integer 3 to the channel [time 17:56:35].
Producer p1 writes Integer 4 to the channel [time 17:56:36].
Consumer:
Consumer c1 takes Integer 0 from the channel [time 17:56:34].
Consumer c1 takes Integer 1 from the channel [time 17:56:37].
Consumer c1 takes Integer 2 from the channel [time 17:56:40].
Consumer c1 takes Integer 3 from the channel [time 17:56:43].
Consumer c1 takes Integer 4 from the channel [time 17:56:46].

With a FIFO-2 channel.
Producer:
Producer p1 writes Integer 0 to the channel [time 18:15:53].
Producer p1 writes Integer 1 to the channel [time 18:15:54].
Producer p1 writes Integer 2 to the channel [time 18:15:55].
Producer p1 writes Integer 3 to the channel [time 18:15:58].
Producer p1 writes Integer 4 to the channel [time 18:16:1].
Consumer:
Consumer c1 takes Integer 0 from the channel [time 18:15:55].
Consumer c1 takes Integer 1 from the channel [time 18:15:58].
Consumer c1 takes Integer 2 from the channel [time 18:16:1].
Consumer c1 takes Integer 3 from the channel [time 18:16:4].
Consumer c1 takes Integer 4 from the channel [time 18:16:7].
```

Figure 6.5: Exogenous Coordination.

selves but still want to exogenously coordinate them. For this purpose we have the *coordination only* channel types (which we introduced in Section 2.4.2).

Analogous to our producer/consumer example, we can use these two source- and two sink-end channel types to coordinate two producers or two consumers, as shown in Figure 6.6. The Java implementation of both threads remains the same (as given in figures 6.3 and 6.4).

We run two experiments: one with two producers and one with two consumers. Each time one of the two threads acts every second, and the other one every three seconds. We show the results of the executions in Figure 6.7. Between the two producers we place an instance of a *synchronous drain* channel type. Therefore, both producers write every three seconds; the slowest producer determines the frequency in this case. Between the two consumers we place an instance of a *spout* channel

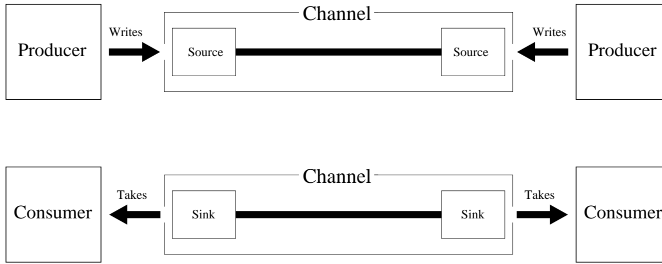


Figure 6.6: Producer/Producer and Consumer/Consumer Example

```
Output listing.
With a Synchronous Drain channel.
Producer p1 [writes every second]:
Producer p1 writes Integer 0 to the channel [time 21:55:39].
Producer p1 writes Integer 1 to the channel [time 21:55:42].
Producer p1 writes Integer 2 to the channel [time 21:55:45].
Producer p1 writes Integer 3 to the channel [time 21:55:48].
Producer p1 writes Integer 4 to the channel [time 21:55:51].
Producer p2 [writes every three seconds]:
Producer p2 writes Integer 0 to the channel [time 21:55:39].
Producer p2 writes Integer 1 to the channel [time 21:55:42].
Producer p2 writes Integer 2 to the channel [time 21:55:45].
Producer p2 writes Integer 3 to the channel [time 21:55:48].
Producer p2 writes Integer 4 to the channel [time 21:55:51].

With a Spout channel.
Consumer c1 [takes every second]:
Consumer c1 takes Integer 1251778893 from the channel [time 21:59:47].
Consumer c1 takes Integer 1498255357 from the channel [time 21:59:48].
Consumer c1 takes Integer 659978419 from the channel [time 21:59:49].
Consumer c1 takes Integer 695311685 from the channel [time 21:59:50].
Consumer c1 takes Integer 1839671779 from the channel [time 21:59:51].
Consumer c2 [takes every three seconds]:
Consumer c2 takes Integer 839736381 from the channel [time 21:59:49].
Consumer c2 takes Integer 1640308734 from the channel [time 21:59:52].
Consumer c2 takes Integer 1243628531 from the channel [time 21:59:55].
Consumer c2 takes Integer 1207696563 from the channel [time 21:59:58].
Consumer c2 takes Integer 307909386 from the channel [time 22:0:1].
```

Figure 6.7: Exogenous Coordination with Coordination-only Channel Types.

type. We see that indeed the writes are independent of each other. Observe that this channel type produces random integers that are different for each end. This causes the strange integer output listed in the figure.

6.3.3 Competing Producers and Consumers

In a distributed system several components may have a reference to a particular channel-end. This means that they are all allowed to use this end. However, usually, we like to have a *one-to-one* channel communication between the components, where no other component can interfere with the communication between the two components currently using a particular channel. Thankfully, in chocoMoCha we require components to first successfully *connect* to a channel-end before being able

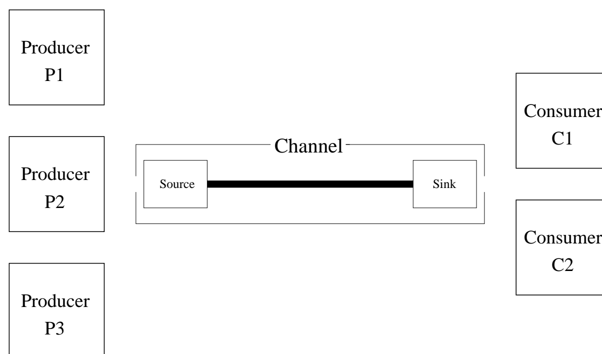


Figure 6.8: Competing Producers and Consumers Example.

to use it (see Section 6.2.3). This way, a component gains exclusive access to a particular channel-end until it *disconnects* from it. Therefore, taking the producer and consumer of the previous examples and creating several instances of them, like in Figure 6.8, is not a problem. At any particular point in time, at most one producer gets to perform its write operations, and at most one consumer gets to perform its take operations.

Naturally, in some cases a *one-to-many*, *many-to-one*, or a *many-to-many* communication is desired. This is also possible in chocoMoCha by letting the components share their *component key* (as explained in Section 6.2.1). For this purpose, we need to change the code of the producer and the consumer (see figures 6.3 and 6.4) so that they are able to share their instance of `ComponentKey`.

6.3.4 Untyped Producer/Consumer

In the previous examples, the producers and the consumers use the typed methods `writeInteger` and `takeInteger`. However, as we discussed in Section 6.2.3, most distributed systems work with dynamic data types. In this example, we show the usage of the generic untyped chocoMoCha `write` and `take` methods.

For the producer (see Figure 6.3) the only thing that changes is the replacement of the `source.writeInteger(new Integer(i), key)` statement for the untyped statement `source.write(new Integer(i), key)`. For the consumer (see Figure 6.4) we need to determine the type of the data that it takes. Suppose that the consumer can deal with instances of the `Double`, `Integer`, and `String` classes. Then, we change the implementation code of the `for`-loop into:

```
for (int i = 0; i < times; i++) {
    sleep(waittime); // sleep between takes.
    res = sink.take(key);
    if (res instanceof Double) { }
    if (res instanceof Integer) { }
    if (res instanceof String) { }
    }// done
```

Naturally, in this way we can make a consumer that deals with many data types. It is even possible to dynamically add new types to the consumer by using the `isInstance` method of the special system Java class `Class` [Eck98]. However, since each type needs to be dealt with in a different manner it is better to either replace the consumer for an updated version, or delegate the data to some other consumer that can deal with it. In our next example we discuss this last scenario.

6.3.5 Crazy Producer and Cooperating Consumers

In this more elaborated example we show how components distribute the knowledge of channel-ends through channels themselves, as well as movement of the channel-ends to other locations in the system, and the difference between the *read* and *take* operations.

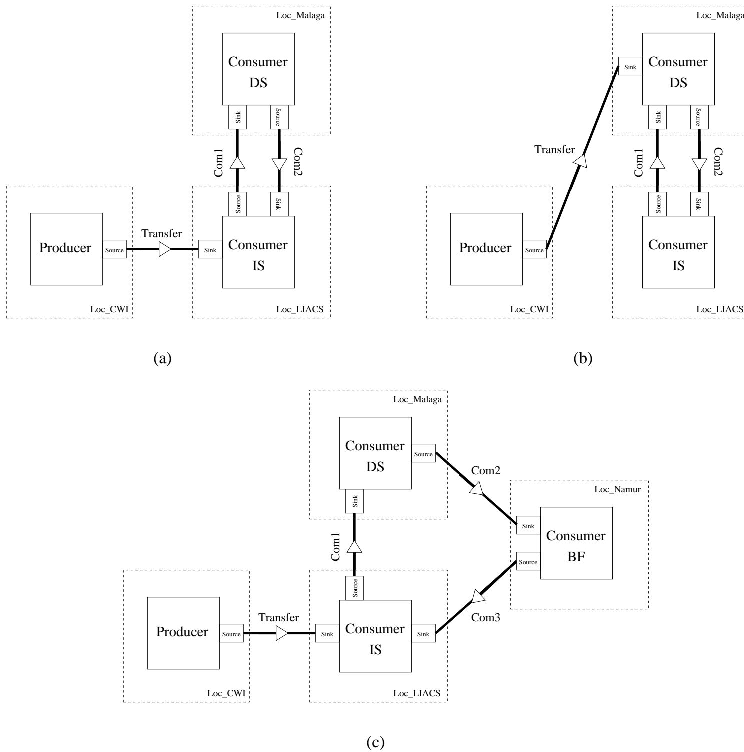


Figure 6.9: Crazy Producer and Cooperating Consumers Example.

Our example consists of one crazy producer and two consumers. We give the architecture of this example in Figure 6.9(a). The dashed lines denote a specific *location* in the network. The crazy producer randomly writes values of type `Integer`, `Double`, or `String` to the source-end of the *transfer* channel. The consumers read and take these values from the sink-end of this channel. However, each separate consumer is not able to deal with all the data types. Consumer *IS* handles data

of types `Integer` and `String`, while consumer *DS* handles data of types `Double` and `String`. Therefore, the consumers need to cooperate with each other to deal with all the data types that the crazy producer is writing to the *transfer* channel. For this purpose, the two consumers are linked to each other by the channels *com1* and *com2*. The idea is that if a consumer reads data of a known type it takes it out of the channel, otherwise it delegates the task to the other consumer.

The implementation of the producer is the same as that of Figure 6.3, except that we substitute the single `write` statement of the `for`-loop with a random choice out of three writing possibilities:

```
for (int i = 0; i < times; i++) {
    sleep(waittime); // sleep between writes.
    choice = (int) ((Math.random () * 3) + 1);
    if (choice == 1) { source.write(new Integer(i), key);}
    if (choice == 2) { source.write(new Double(i), key);}
    if (choice == 3) { source.write(new String(String.valueOf(i)), key);}
    }// done
```

Notice that instead of one crazy producer we could also take three producers where each of them writes a different data type. These producers, then, would automatically compete among themselves to write to the source-end of the *transfer* channel (as explained in the last example). However, for entertainment purposes we chose to use a crazy producer instead in this example.

The implementation of the two consumers is identical from the point of view of the chocoMoCha middleware. The differences are in the way the consumers deal with the data they know the type of. In Figure 6.10 we give the implementation of consumer *IS*. The implementation of consumer *DS* is the same except for replacing the `Integer` data type check with a `Double` data type check.

Upon initialization, the consumer needs the following parameters: the sink-end (`SinkEnd si`) of the *transfer* channel, the source-end (`SourceEnd communicationSource`) of the *com1* channel, the sink-end (`SinkEnd communicationSink`) of the *com2* channel, and the length of time in between takes (`int wait`). For identification purposes, the constructor creates an instance of the chocoMoCha `ComponentKey class`. We also let the consumer create its own chocoMoCha location, but as an alternative it could also be a parameter, if desired.

When starting the execution, the first thing that the consumer does is to connect to the ends of the *com1* and *com2* channels. At its termination the consumer disconnects from these ends. For simplicity, we omit the details of a termination protocol for the consumers by letting the consumer go on forever in a `while(true)`-loop. Therefore, the disconnect statements are not reachable in this example. Naturally, an easy way for termination is to let the producer send a special termination token when it finishes writing. Then all consumers can read this token and terminate. However, we don't want to unnecessarily complicate the example.

In the `while(true)`-loop, the consumer checks whether it has a valid reference to the sink-end of the *transfer* channel; this is the case when `transSink` is not `null`. In Figure 6.10(a), the consumer *IS* has a valid reference to the sink-end, which is given at initialization. If the reference is valid, the consumer reads a value from the channel. This means, that it gets a copy of the next value that the channel is going to output. However, the original value remains in the channel until removed

```
import cwi.sen3.chocoMoCha.*;
/** Consumer, takes Integer and String.
* @author Juan Vicente Guillen-Scholten
*/
public class ConsumerIS extends Thread {
   ChannelEnd transSink;
   SourceEnd comOutSource;
   SinkEnd comInSink;
   ComponentKey key;
   MoChaLocation loc_LIACS;
   int waittime;
   /** Constructor.
   * @param si the sink-end of the transfer-channel.
   * @param communicationSource the source-end of the com1-channel.
   * @param communicationSink the sink-end of the com2-channel.
   * @param wait time to wait in between takes (1000 = 1 second).
   */
   Consumer (SinkEnd si, SourceEnd communicationSource,
   SinkEnd communicationSink, int wait) {
      waittime = wait;
      transSink = si;
      comOutSource = communicationSource;
      comInSink = communicationSink;
      key = new ComponentKey();
      MoChaLocation loc_LIACS = new MoChaLocation();
      if (transSink != null) { transSink.connect(key);}
   }// constructor
   public void run() {
      Object value;
      comOutSource.connect(key);
      comOutSink.connect(key);
      while(true) {
         sleep(waittime); // sleep between takes.
         if (transSink == null) {
            transSink = (sinkEnd) comInSink.take(key);
            transSink.connect(key);
            transSink.move(loc_LIACS);
         }// fi
         value = transSink.read(key);
         if (value == instanceof Integer || value == instanceof String) {
            value = transSink.take(key);
         } else {
            comOutSource.write(transSink, key);
            transSink.disconnect(key);
            transSink = null;
         }// fi
      }// done
      comOutSource.disconnect(key);
      comOutSink.disconnect(key);
   }//run
} // end ConsumerIS
```

Figure 6.10: The Consumer IS

by a `take` operation. After reading the value, the consumer checks if it is a known type. If it is, the consumer takes the value out of the channel, does something with it, and the cycle repeats. If the value is of an unknown type, the consumer delegates the value to some other consumer. It does this by writing the `transSink` reference to the source-end of the outgoing communication channel (`comOutSource`). Then, it disconnects from the `transSink` channel-end so that other consumers can connect to it, and sets the reference to `null` so that it knows that it cannot use this end anymore.

If the reference `transSink` is `null`, as with the consumer *DS* of Figure 6.10(a), then the consumer performs a take operation on the sink-end of the incoming communication channel (`comInSink`). Only the sink-end of the *transfer* channel is passed through the communication channels, therefore, when the operation succeeds the consumer acquires a reference to this sink-end. It then connects to it, and moves the channel-end to its location for efficiency reasons. Now it is ready to use it as described above. This is the case in Figure 6.10(b), where the sink-end of the *transfer* channel moves from the *IS* to the *DS* consumer.

Observe that, with this protocol it is easy to add new consumers without having to change the code of the producer and other consumers. For example, in Figure 6.10(c) we added a new consumer *BF* that handles data of type `Boolean` and `Float`. For this purpose, we created a new communication channel *com3* and linked the new consumer to the first one. We also linked the *DS* consumer to the new one. Thus we create a chain of cooperating consumers. Other new consumers are added to this chain in the same way.

6.4 Applications

There are many applications that benefit from using the MoCha middleware. In this section, we discuss three application areas: *Component Based Software*, *Web Services*, and *Peer-to-Peer networks.* Notice that, these areas are not disjoint. Web Services are an instance of Component Based Software, and Component Based Systems can have a Peer-to-Peer architecture. However, for example, not all Peer-to-Peer networks are Component Based.

6.4.1 Component Based Systems

The MoCha middleware is especially suitable for Component Based Systems. These are distributed systems consisting of *components* and *connections* among them. In Chapter 5, we already discussed a channel-based model for component based software. Basically, a component is a *black-box*, whose internal implementation is hidden from the outside world. Components offer an *interface* that describes their externally observable behavior. We defined such an interface in Section 5.2.2 as a set of channel-ends plus semantics describing this observable behavior. With this model we are able to use the MoCha middleware for communication and coordination of components. There are many benefits for doing this, we discuss five:

First, the channels of the middleware respect the *black-box* encapsulation property of components by prohibiting objects that are transmitted through channels to refer to any internal objects of these components (see Section 6.2.3).

Second, the *anonymous* aspect of communication through channels provides a decoupling that makes it possible to dynamically plug in and remove components from the system. The components have no references to each other and perform I/O network operations only on channels. This means that a component can be updated or replaced during runtime by another one without the component at the other end of the channel having any knowledge of any of this happening.

Third, the basic *exogenous coordination* that channels provide makes it possible to change the behavior of component based systems without having to change

the components themselves (as we showed in the examples of Section 6.3). Moreover, with mobile channels we can make a clear separation of concerns between the computational and the coordination part of these systems.

Fourth, the *mobile* property of channel-ends allows dynamic reconfiguration of channel connections among the components in a system. An example of how this is done with the middleware is given in Section 6.3.5. This property is very useful when components themselves are mobile (laptops, mobile phones, mobile Internet agents, etc.). For when such a component moves it can take the channel-ends it is currently connected to with him to the new location. In Section 8.2.5, we give an example of a mobile agent.

Fifth, and last, the MoCha middleware is easy to use in combination with, and to incorporate into, several other Java technologies that are being used for component based systems. Examples are: *JavaBeans* [JB], *RMI* [RMI], *Corba* [TB00] (for Java), *JavaSpaces* [FHA99], *Jini* [Jini, ASSWW99], and *Enterprise JavaBeans* [EJB].

A representative example of Component Based Systems and Java technology are *In-Home Networks* [Sto02]. These are small networks at people's houses, that aim at connecting each device with each other device. We can think of devices such as televisions, video-recorders, coffee machines, microwaves, lamps, etc. Besides these static devices, there are also mobile ones like laptops, mobile phones, cleaning robots, remote controls, etc. The *Jini* technology, among others, is being used for these networks. Since Jini uses *RMI* as its underlying communication protocol, it is easy to incorporate the MoCha middleware so that *In-Home Networks* benefit from all the advantages described above.

6.4.2 Web Services

Web Services [BHMNCFO04] are actually an instance of Component Based Systems. They consist of a set of components which are invoked, and whose interface descriptions are published and discovered. These components are called Web Services, which can consist of many sub-services. The main requirement is that these services have well-defined interfaces and are accessible to humans, other services, and software components in general. In terms of e-business and business processes, we can see a Web Service as a reusable piece of business logic that an organization exposes to other organizations through the World Wide Web.

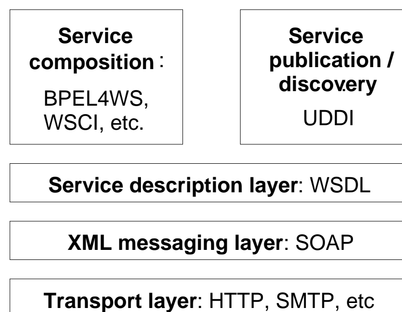


Figure 6.11: Layered Overview of Web Services

Figure 6.11 gives an overview of the main ingredients of the Web Services standard in a layered fashion. The *transport layer* has the responsibility of transporting messages between a service provider and a requester. The *XML messaging layer* has the responsibility of encoding the communication messages using a common XML format such as the SOAP protocol [XPG03], so that both sides have a common language for communication. The *service description layer* describes the public interface of a Web Service using the Web Services Description Language (WSDL) [CCMW01]. The *service publication and discovery* allows publishing and searching for Web Services using the Universal Description, Discovery and Integration (UDDI) [OAS04]. The SOAP, WSDL, and UDDI specifications constitute the core of the WS standard. These specifications have reached a mature state wherein many major software vendors have committed to incorporating Web Services into the basic infrastructure of their products. The *Web Services composition layer* facilitates the construction of new Web Services that are constituted out of simpler ones. At present, the industry has not agreed upon a common specification for service composition. Business Process Execution Language for Web Services (BPEL4WS) [CGKLRTW03] and the Web Service Choreography Interface (WSCI) [W3C02] constitute two examples of candidates for a service composition specification standard.

The most common and standard Web Services example is the travel agent scenario [HHO04]. In this scenario a travel agent books vacations for its customers. The agent makes use of several Web Services provided by airlines, hotels, bus companies and car rental companies, for booking respectively a plane, hotel, bus tickets, or a car. These Web Services themselves make use of other Web Services provided by credit card companies to guarantee payments made by the customers. The main purpose of this example is to show the need for coordination of the different services. For example, a customer buys a vacation only if he can get a seat on the plane, a room in a hotel, and a car at his destination. If any of these three bookings fail, the customer does not want this particular vacation. Booking each constituent separately is not a good idea for we need to cancel everything each time one part fails. Therefore, we need to synchronously coordinate these Web Services so that they are handled at the same time to ensure the vacation for the customer.

This need for Web Services “orchestration” has inspired the coordination community to produce several models for this purpose. For example, in the work of *Bocchi* [Boc04] the asynchronous π-calculus is used for Web Service composition based on the notion of long running transactions. As another example, in the work of *Diakov* and *Arbab* [DF04] a model for coordinating Web Services using the Reo coordination paradigm [Arb04] is presented. In this last work, the MoCha middleware is used for implementation of the coordination model. Therefore, we briefly discuss this implementation as an example of how to use our middleware for Web Services coordination.

In Reo, complex coordinators, called *connectors*, are defined and constructed out of simpler ones, where the most simple connector is a mobile channel. The idea of *Diakov* and *Arbab* is to provide a specification for service composition (see Figure 6.11) that is based on Reo. Besides giving a composition specification they also want to really coordinate Web Services using a coordination middleware that is based on mobile channels. One of their approaches for doing this is to define a *Reo transport layer* for Web Services technology. This way, both new Reo Web Services

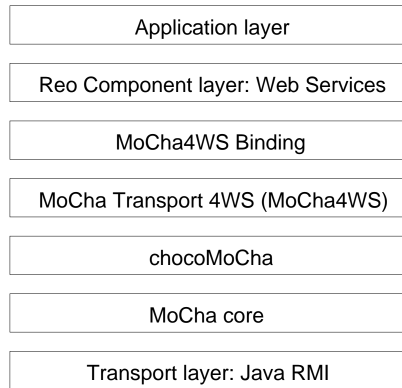


Figure 6.12: MoCha for Web Services

and existing non-Reo Web Services are all coordinated by the middleware through this transport layer.

In Figure 6.12, we give an overview of the implementation scheme. The *transport layer* is now Java RMI, which makes it possible to use the MoCha middleware. The Web Services standard separates the definition of messages exchanged with a Web Service from the way the distributed environment communicates to and from a Web Service. Therefore, *Diakov* and *Arbab* defined a *MoCha transport for Web Services* layer on top of the *chocoMoCha* middleware layer. On top of this transport layer, there is a *MoCha for Web Services Binding* layer. Bindings define how a service provider sends messages with a particular transport, which in this case is the MoCha middleware. Since MoCha has no notion of channel composition nor components, an extra *Reo Web Services* layer (*Reo component layer* in the figure) is added that provides these extra features to the applications.

Thanks to the added Web Services layers we are able to coordinate Web Services using the MoCha middleware. No changes to the middleware are needed since the Web Services standard allows the use of Java RMI as a lower transport layer. Non-trivial exogenous coordination is provided by the *Reo Web Services* layer. Alternatively, we could define a Web Services layer for the MoCha channel composition model of Chapter 9 (which implements a subset of Reo).

6.4.3 Hybrid and Pure P2P Networks

Today, a big percentage of the Internet traffic is generated by file-sharing applications. Most applications of this kind are based on a so called peer-to-peer (P2P) network. P2P networking refers to a class of systems, applications and architectures that employ distributed resources to perform any kind of task in a decentralized and self organizing way [Sch01]. The popularity of P2P networks originates from the introduction of the Napster [Shi01] application in the year 2000 and it is continued by many other P2P file-sharing applications like Kazaa [Sha03], BitTorrent [Coh03], and many clients of the Gnutella network [Rip01].

P2P networks are often put in contrast with *client/server networks*. That is because in many network architectures each process on the network is either a client or a server: servers are processes dedicated to specific tasks like managing of disk drives, printers, or network traffic, whereas clients are processes that rely on servers for resources. The clients themselves do not share any resources. In a *peer-to-peer* architecture each node is both a client and a server at the same time. Therefore, the nodes are said to be *equal*. They have equivalent responsibilities, enabling applications that focus on collaboration and communication in a decentralized and self organizing way. Features of a peer-to-peer architecture include a better distributed network control, high availability through the existence of multiple peers in a group, and the possibility of dynamic exchange of information about the network topology.

The flexibility of P2P network architectures is increased by infrastructures that (1) allow making connections between distributed nodes across several heterogeneous platforms and operating systems, (2) enable nodes to establish anonymous connections among them, (3) provide some kind of mechanism for easy dynamic reconfiguration of the network topology, (4) provide exogenous coordination by letting the creator of the connection choose between a synchronous or an asynchronous type, and (5) offer a clear and easy high-level API for P2P applications.

The MoCha middleware offers such an infrastructure. Next, we discuss the two current architecture types for P2P networks. These are the *hybrid* and the *pure* P2P network architectures as defined in [Sch01]. We explain how to implement both P2P architectures using mobile channels by giving an example for each of them. Our purpose is to show the advantages of using the MoCha middleware for P2P applications.

In a *hybrid* P2P network there is always a *central entity* necessary that provides parts of the offered network services. Such a central entity is often regarded as a server in the traditional way. However, the definition of a hybrid P2P architecture is not the same as the one of the client/server architecture; All the nodes of the first potentially share (their local) resources, while the clients of the second do not.

Figure 6.13 shows a *hybrid* P2P network that uses mobile channels for connections between its nodes. This network example is similar to the one of the Napster application[Shi01]. Each *application node* has a set of resources to share among the other nodes of the network. However, an application node does not know any other nodes, nor the resources these other nodes are sharing. Instead, an application node connects to a *central index server* that contains a list of all the resources available from all the nodes connected to it. Once a node receives a list of resources from the *server* and requests a particular resource, the *server* arranges a connection between the requesting- and the providing-node.

Implementing this example in chocoMoCha is fairly easy. In Figure 6.14 we show the most important methods of a possible implementation. Figure 6.13 shows a snap-shot of our example network. The server has several *request channels*. The sink-ends of these channels are kept private by the server and are meant for reading requests. However, the source-end references are known to all the application nodes in order for them to write requests to these channels. Since, in our example, the server is always on-line the nodes get a source-end reference at their creation; see their `constructor` method. Each application node has a *connection channel* meant for receiving data from the outside world, it does so by reading from the sink-end of

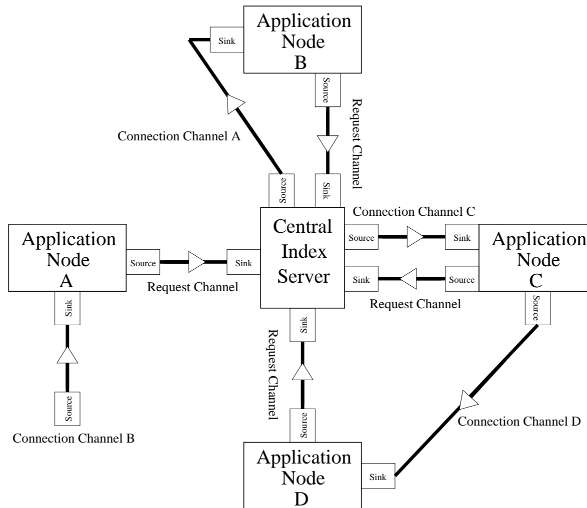


Figure 6.13: A Hybrid P2P Network.

this channel. The nodes spread the reference of the source-end to the server when connecting to it, as specified in the method `connect`. Suppose that node A is in the process of connecting to the server, then node B represents the resulting state. The server moved the source-end to its location and wrote an acknowledgment message back.

At some point in time node B requests a resource; see the `getResource` method. The server, in response, reads the request but it does not take it out of the channel; see the `performGetResource` method. Instead, the server looks randomly for a node that has the requested resource and moves the *connection source channel-end* to the found application node. This is the state represented by the nodes D and C. Node C receives a resource request from the server that was written by node D. The request remained in the channel unaffected by the channel-end move and without node D begin aware of it. Since the request also contains the target source-end, node C writes the data to it. However, it does not know that node D is receiving the data, nor does node D know that it is getting the requested data from node C. Therefore, the connection is *completely anonymous*.

To illustrate the advantage of exogenous coordination: we chose the types of the *request channels* and the *connection channels* to be respectively *asynchronous FIFO* and *synchronous*. However, we could choose other channel types as well, if desired. For example, we can make the *request channels* to be of type *synchronous*. This way, we get a different system behavior with the big advantage of not having to change, nor re-compile, the code of the application nodes. Moreover, the nodes don't even know what channel types they deal with.

A *pure* P2P network has no *central entity*, it is completely decentralized. Figure 6.15 shows a *pure* P2P network that uses mobile channels for connections between its nodes. This network example is similar to, but not entirely the same as the Kazaa network[Sha03]. Actually, the implementation of this example is also similar to the

```
class P2PApplicationNode
   P2PApplicationNode (SourceEnd ce)
      requestSource = ce;
      location = new MoChaLocation();
      connection = new MobileChannel(location,"Synchronous");
      key = new ComponentKey();
      connection.ce2.connect(key);
   connect()
      // connection.ce1 = source-end
      // sharelist = list of resources we share.
      if (!connection.ce1.full()) {
         Message msg = new Message("Joining network", connection.ce1, shareList);
         requestSource.connect(key);
         requestSource.write(msg, key); }
         requestSource.disconnect(key);
      else { // find another server or try later.  }
   getResource(String resource)
      // connection.ce2 = sink-end
      Message msg = new Message("Request", resource, connection.ce1);
      requestSource.connect(key);
      requestSource.write(msg, key);
         requestSource.disconnect(key);
      while(!finished) {
         msg = connection.ce2.take(key);
         result.add(msg); } //done
class centralizedIndexServer
   centralizedIndexServer()
      location = new MoChaLocation();
      request = new MobileChannel(location, "FIFO 100");
      key = new ComponentKey();
      request.ce2.connect(key);
   void performConnect()
      msg = request.ce2.take(key); // request.ce2 = sink-end
      shareList.add(msg.shareList, msg.source);
      msg.source.connect(key);
      msg.source.move(location, key); // move conn.  chan.  source-end to us.
      msg.source.write(new Message("Connected to server"), key);
      msg.source.disconnect(key);
   void performGetResource()
      msg = request.ce2.read(key); // request.ce2 = sink-end
      Node tmp = shareList.getRandomNodeWith(Resource);
      msg.source.connect(key);
      msg.source.move(node.location, key); // move conn.  chan.
         // source-end to resource node.
      tmp.source.write(msg, key); // msg already contains target SourceEnd.
      msg.source.disconnect(key);
```

Figure 6.14: Partial Abstract Java Code of a Hybrid P2P Network.

one of the *hybrid* network example. That is why we do not present any code for this example. Most of the functionality is already given in Figure 6.14.

Instead of having a fixed central server, we now have *supernodes*. A *supernode* is a normal application node, but at the same time it performs some of the tasks of the *server* in the *hybrid* example; it keeps a resource-list of the connected clients, and it arranges connections between the different connected nodes in the same manner as the server did. For legal reasons, the nodes cannot share any resources of the supernode they are connected too, and vice-versa.

A supernode itself is a normal node connected to another supernode. Any node can become a supernode and back to normal depending on the network state and heuristics. This means that nodes need a dynamic list of supernode source-ends,

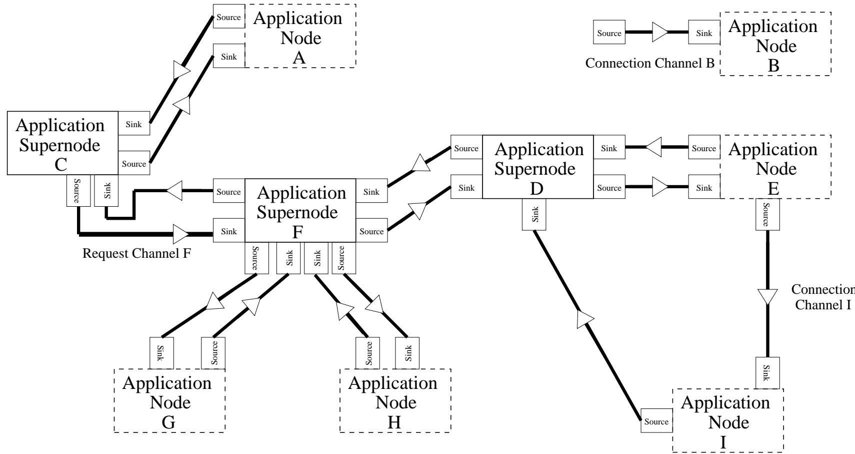


Figure 6.15: A Pure P2P Network.

for when the supernode they are connected to becomes unavailable. To keep its list up to date a node can request source-end references of neighbor nodes from its supernode. However, to keep network traffic down, a node can connect to only one supernode at a time.

In Figure 6.15 nodes E and I are connected to supernode D, nodes C, D, G and H are connected to supernode F, node A is connected to supernode C, and node B is not connected to any supernode. In this snap-shot nodes E and I are the only ones involved in resource transfer; node E is writing data to the connection channel of node I. The anonymous connection between the two nodes is made by supernode D in the same way as the index server in the hybrid network example.

In this *pure* P2P network the topology changes more than that of the previous example. This more dynamically changing network example clearly shows the benefits of the mobility feature of MoCha's channels. Instead of creating and deleting channels every time a topological change occurs, we just simply move channel-ends to other nodes. When moving one end, the node using the other end of the channel is not even aware of the channel-end movement.

Just like in the previous example, we can change the network's behavior by choosing different types for the channels between the nodes. All of this is done in an exogenous way.

6.5 Related Work

In this section we discuss and compare related middleware systems with our MoCha middleware. For convenience, we divide the section in three parts: channel-based middleware, coordination middleware, and P2P middleware.

6.5.1 Channel-Based Middleware

We discuss four middleware systems that are based on the notion of channels.

CTJ, CTC, CTC++

Communicating Threads for Java (*CTJ*) [HBB00] is based on the CSP paradigm [Hoa85]. In CSP (*Communicating Sequential Processes*) a system is described as a number of processes and channel connections among them. These processes operate independently (in parallel) and communicate with each other using channels. The CTJ package is an implementation of processes and channels in the Java language. There is also a C (CTC) and a C++ (CTC++) version available. The packages/libraries are developed to make parallel programming easier in these programming languages.

The channels of CTJ, (CTC, and CTC++), provide anonymous communication (between processes) and can be used in distributed environments. These are the similarities with the MoCha middleware. However, there are quite some differences. We discuss four of them. First, CTJ works with channels as a whole and no channel-ends. Second, the CTJ channels are exclusively synchronous. Even if the work in [HBB00] suggests that there are buffered channels available, we found no trace of asynchronous channels in the package itself. Third, the channels are not mobile. No dynamic reconfiguration of channel-connections in the system is possible. Fourth the channels of CTJ are not primarily developed to work in distributed environments. Therefore, distribution is indirectly supported through a plug-in structure that must be provided by the user.

JCSP

CSP for Java (*JCSP*) [Wel01] is also based on the CSP paradigm [Hoa85]. The authors of JCSP argue that the *monitor-threads* model provided by Java, while easy to understand, proves very difficult to apply *safely* in any system above a modest level of complexity. This makes parallel programming in Java very hard. However, parallel composition of CSP processes is easier to apply, is mathematically clean, yields no engineering surprises and scales well with system complexity. This is the reason they developed JCSP.

JCSP is a Java class library providing a base range of CSP primitives plus a rich set of extensions. It provides processes and channels in Java. Like the MoCha middleware, JCSP provides anonymous communication. The differences are almost the same has for CTJ (see above): no channel-ends, exclusively synchronous channels, and no mobility.

JCSP Network Edition

The *JCSP network edition* (JCSP.net for short) extends the JCSP middleware (see above) with distributed channels. JCSP.net seems closely related to MoCha since it shares many of its features. We now extensively discuss this middleware and the main commonalities and differences with our middleware. Unlike JCSP, JCSP.net is a commercial product. Unfortunately, we were not able to get a version of the middleware for testing purposes. Therefore, we are limited in our comparison on the

work presented in [VW02, AFW02] and the free (short) documentation available on line at [JCSP.net].

JCSP.net views a system consisting of processes that run on nodes. Several processes can run on a single node. A node is associated with a Java virtual machine (JVM) [Java] and represents a particular location in the network. Thus, all processes running on the same node have the same location. This is different in MoCha, where a location is a logical notion that is decoupled from the underlying JVM. In fact, a JVM can have many MoCha locations, and a location can involve many JVM's (see Section 6.2.1).

JCSP.net provides two kinds of channels. *Local JCSP channels* for processes on the same node, and *network channels* for communication between processes on different nodes. The local channels pass objects by reference, and the network channels pass the objects by copying (using Java serialization). The MoCha channels pass objects exclusively by copying and are used for both local and remote communication. This is all done internally, the fact whether the communication is local or remote is hidden from the processes that use MoCha channels.

Channels that are created between nodes are virtual in JCSP.net. Internally, between two nodes a bi-directional *link* is created that multiplexes the data from the various virtual channels. This is different in MoCha, where for each channel that is created by processes, a physical implementation independent of other channels is created internally.

We concentrate on the *network channels* since these types provide the distributed communication. Network channels are either *to-the-net* or *from-the-net*. Actually, we can regard these channels as implicit *channel-ends*. In MoCha a *to-the-net* channel corresponds to a *source* channel-end, and a *from-the-net* channel corresponds to a *sink* channel-end. Network channels may have *one* or *any* number of application processes attached to them. This gives us the set {*One2NetChannel*, *Any2NetChannel*, *Net2OneChannel*, *Net2AnyChannel* }. Thus, the choice of whether the communication is *one-to-one*, *many-to-many*, etc., is statically determined. In MoCha, this choice is made dynamically and can change during the life time of the channel. We accomplish this through the `ComponentKey` identification class and the connect/disconnect operations (see Sections 6.2.1 and 6.2.3).

There are two ways of setting up channels between processes in JCSP.net, either through a *Channel Name Server* (CNS) or *Anonymously*. The first is the standard way, the second is an alternative that is more related to MoCha but has some restrictions. We discuss both schemes.

With the first approach, a writing process creates a channel-end and registers a particular unique name with the CNS. For example:

```
Net2OneChannel source = new Net2OneChannel ("cwi.channel");
```

At the other side, a taking process creates a channel-end and registers it to the same CNS. For example:

```
One2NetChannel sink = new One2NetChannel ("cwi.channel");
```

The CNS, then, relates the two channel-ends and creates a link between them so that the processes can start using them. We see that the processes create the

channel-ends separately and explicitly relate them by using the same name, which they somehow must know beforehand. In MoCha, instead, a channel as a whole is created, where the channel-ends are internally related and given to the creator. Such a channel creator can be a third party.

With the second approach, the networked channels do not get registered to the CNS, instead they are constructed anonymously. The input-end (source-end in MoCha) is then send to other processes via some other existing channel. An example of the taker side:

```
Net2OneChannel sinkCom = new Net2OneChannel ();
NetChannelLocation sinkComLocation = sinkCom.getLocation ();
SourceExisting.write (sinkComLocation);
```

The writer, then, receives the location of the created sink-end and is able to create a matching source-end. For example:

```
NetChannelLocation sinkComLocation =
(NetChannelLocation) SinkExisting.read ();
One2NetChannel sourceCom =
new One2NetChannel (sinkComLocation);
```

Unlike in MoCha, channel-ends are not directly sent through channels themselves. Instead, the locations of these ends are send. We could not find out whether these locations are unique for each end or not. In the second case, the question “what happens if two channel-ends are at the same location?” arises. Furthermore, it is only possible to send *from-the-net* channels (sink channel-ends) through other channels.

Regarding the exclusive access to particular channel-ends, in MoCha we have the connect/disconnect operations for this purpose. However, in JCSP.net there is no such mechanism. If we want to have a private channel communication between two processes we must either: (a) make sure that they are the only ones that know the CSN name, or (b) with the anonymous creation approach, make sure that we pass the sink-end to the right process, and make sure that this process does pass this end to its friends.

A last comparison involves the different channel types. Next to the primary synchronous channel, there are also buffered channels available in JCSP.net. We were unable to determine how these types are selected at creation. However, this implies some sort of coordination like in the MoCha middleware. Most probably, this coordination is not exogenous since it seems that the processes involved in the communication themselves need to agree on the type of the channel they want to use.

Pict and Nomadic Pict

Pict [PT97] is a concurrent programming language based on the π-calculus. The π-calculus is a model for describing concurrent computation as systems of communicating processes. In the computational world modeled by the π-calculus there are two entities: processes and mobile channels. Processes, are the active components of

a system; they interact by synchronous rendezvous on mobile channels, also called names or ports. Pict includes a Pict-to-C compiler, reference manual, language tutorial, numerous libraries, and example programs.

Similar to the MoCha middleware, Pict supports *logical mobility* as defined in Section 2.2.2, i.e. the spreading of channel knowledge through the system by allowing to send channel identities through channels. However, since pict does not support distributed environments, it also does not support *physical mobility* (as defined in the same section), i.e. physically moving a channel(-end) from one location to another in the network. Furthermore, all channels in Pict are synchronous. Some form of asynchronous output is allowed, but the receiver must always send an explicit acknowledgment back to the sender.

Nomadic Pict [WS99] is the distributed version of Pict that supports both types of mobility as described above. The difference with the MoCha middleware, then, is the fact that there are no explicit channel-ends present, and there is no exogenous coordination since there is only one channel type (synchronous).

6.5.2 Coordination Middleware

We discuss and relate three middleware systems that are, like MoCha, explicitly made for coordination purposes.

JavaSpaces

The *JavaSpaces* [FHA99] technology is a Jini [Jini, ASSWW99] service from Sun Microsystems that is based on the notion of shared data spaces. In a *shared data space*, all components read and write values, usually *tuples* like in *Linda* [CG90], 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.

A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism; instead of communicating directly, they coordinate by exchanging objects through spaces. Processes perform simple operations to *write* new objects into a space, *take* objects from a space, or *read* (make a copy of) objects in a space. The persistent property of the space means that a collection of data remains intact even if its source is no longer attached to (or temporarily disconnected from) the network. The processes use the RMI technology to access the spaces. These space repositories are not distributed themselves but are centralized in one network location (one machine, one computer, etc.), making all operations on spaces remote (except if the process performing the operation happens to be at the same location).

The *JavaSpaces* coordination model is quite different than the one of MoCha. In Section 5.2.3, we already discussed the main differences. We summarize: shared data spaces are easy to use and even useful for network architectures like blackboard systems. In contrast, mobile channels are more *efficient* to implement for most distributed systems, provide more *security*, are more *architecturally expressive*, and provide transparent *exogenous coordination*.

Lime

The *Linda in a Mobile environment* (*Lime*) [MPR03] is a Java-based middleware that is also based on the notion of shared data spaces. In Linda, processes communicate by writing, reading, and removing data from a tuple space that is assumed to be persistent and globally shared among all processes. *Lime* adapts this notion to a mobile environment by breaking up the notion of a global tuple space, and distributing its contents across multiple mobile components. These components share the content of their local tuple spaces when either they are on the same host, or communication is available between mobile hosts that contain components. This way, they form a federated tuple space. The content accessible through such virtual tuple space changes from time to time according to the current connectivity pattern. *Lime* also introduces the notions of tuple location, for querying a partition of the federated tuple space, and of reactive programming, for allowing actions to be performed with varying degrees of atomicity upon insertion of a tuple.

The *Lime* middleware implementation is kept independent from both the underlying support for mobile agents and the underlying tuple space implementation. In the first case, an adaptation layer is provided in *Lime* that allows the integration of the mobile agent system. For this purpose, the current release provides an adapter for the μCode mobile code system [Pic98]. In the second case, *Lime* uses the LighTS package [Pic05], a tuple space implementation that offers an adaptation layer allowing one to use a different tuple space implementation without changing the interface.

MANIFOLD

Manifold [Arb96a] is a coordination language for writing program modules (coordinator processes) to manage complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes that comprise a single application. The conceptual model behind Manifold is based on *IWIM* (Idealized Worker Idealized Manager) [Arb96b]. The basic concepts in the IWIM model (and thus also in MANIFOLD) are processes, events, ports, and channels (in MANIFOLD called streams). A process is a black box that exchanges units of information with the other processes in its environment through its input and output ports, by means of standard I/O primitives analogous to read and write. The interconnections between the ports of processes are made through directed channels. Independent of channels, there is an event mechanism for information exchange in IWIM. Events are broadcast by their sources, yielding event occurrences. Processes can tune in to specific event sources, and react to event occurrences.

In IWIM, a process can be regarded as a worker process or a manager (or coordinator) process. An application is built as a (dynamic) hierarchy of worker and manager processes. Lowest in the hierarchy are pure worker processes that do not do any coordinating activities. Highest in the hierarchy are pure coordinators. A process between the lowest and highest level may consider itself a worker doing a task for a manager higher in the hierarchy, or a manager coordinating processes lower in the hierarchy.

A Manifold application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts, some of which may be parallel systems.

Processes in the same application may be written in different programming languages and some of them may not know anything about Manifold, nor the fact that they are cooperating with other processes through Manifold in a concurrent application.

Both MoCha and MANIFOLD aim at a separation of concerns between the coordination and the computational aspects of systems by providing exogenous coordination. In MoCha, this is done by selecting different types of channels between two communicating components. In MANIFOLD, this is done by always letting a third party arrange a channel between two communicating processes. The big differences between the two middleware systems is that: (1) MoCha does not have a manager/worker architecture as MANIFOLD does; in MoCha all the components/processes are equal. (2) MoCha works with channel-ends, not channels as a whole. (3), and last, the channels of MoCha have different types for exogenous coordination of the components involved in the communication.

6.5.3 P2P Middleware

In the P2P examples of Section 6.4.3 we showed how P2P systems benefit from the MoCha middleware. Especially P2P systems where coordinated anonymous exogenous connections are desired. However, our middleware provides only a coordination mechanism (mobile channels) and does not provide certain P2P services like *searching for particular data*, *load balancing*, and *indexing*. The second generation of P2P middleware offers a complete package for such systems. Well-known P2P middleware systems are *Chord* [SMKKB01], *Pastry* [RD01], *Tapestry* [ZKJ01], and *CAN* [RFHKS01]. They all provide means for locating nodes and data in the network, as well as efficient and scalable routing protocols for messages. However, they do not provide explicit (exogenous) coordination between the nodes. Therefore, designers using these middleware systems can still profit from MoCha by making prototypes of their systems using mobile channels to explicitly show the coordination aspects of these systems. Later they can implement MoCha's mobile channels in these second generation P2P middleware systems, if desired.

JXTA

The project JXTA [Gon01] provides a set of protocols that have been designed for *ad hoc*, pervasive, and multi-hop peer-to-peer network computing. These protocols establish a virtual network overlay on top of the Internet and non-IP networks, allowing peers to directly interact and self-organize independently of their network connectivity. Multiple ad hoc virtual networks can be created and dynamically mapped into one physical network unleashing a richer multi-dimensional virtual network world. The JXTA technology runs on any device, including cell phones, PDAs, two-way pagers, electronic sensors, desktop computers, and servers.

The JXTA protocol most closely related to MoCha is the *Pipe Binding Protocol*. In contrast to MoCha channels, pipes provide the illusion of a virtual in and out mailbox that is independent of any single peer location, and network topology (multi-hops route). MoCha does not provide middleware services as JXTA does, only communication mechanisms. Services like *searching*, *indexing*, and *authentication* have to be built on top of MoCha, if desired.

Chapter 7

The MoCha Middleware: Implementation Details

In this chapter we discuss the implementation details of the MoCha middleware. We discuss these details at an appropriate level of abstraction, where we give an overview of the main concepts of our implementation. In particular, we focus on the *Java RMI* layer, the *P2P mobile architecture* we build upon it, and the implementation of the *mobile channels*. We also introduce many experiments to determine the performance of the MoCha middleware, and discuss their various results.

7.1 Introduction

In the last chapter, we took the point of view of a distributed system's developer to explain the MoCha middleware. Therefore, we presented its Application Programming Interface (API), showed some examples of usage, discussed applications, and related our middleware with others. There are three versions of the MoCha middleware available. These versions are: "plain" *MoCha*, *easyMoCha*, and *choco-MoCha*. For the developer, each version is a separate package, or library, where he is free in choosing any of the three for his distributed application. These packages are implemented above a *MoCha core* layer, that itself is built on top of Java RMI (see Figure 6.1).

From the point of view of the middleware itself, the easyMoCha, and choco-MoCha packages are built upon the "plain" MoCha package, as illustrated in Figure 7.1. The MoCha package is built upon Java *RMI*, and consists of a *MoCha core* and an API layer. The MoCha core layer coincides with the one presented in Figure 6.1. Internally, it is divided into a *Channels* and a *P2P Mobile Architecture* layer. The first, contains the implementations of the eleven channel types that the middleware currently supports. The second layer, extends the Java RMI architecture in such a way that we obtain a peer-to-peer mobile architecture for the channels to be implemented in.

In this chapter, we discuss the implementation details of the MoCha middleware. In particular, we focus on the *Channels*, *P2P Mobile Architecture*, and *Java RMI*

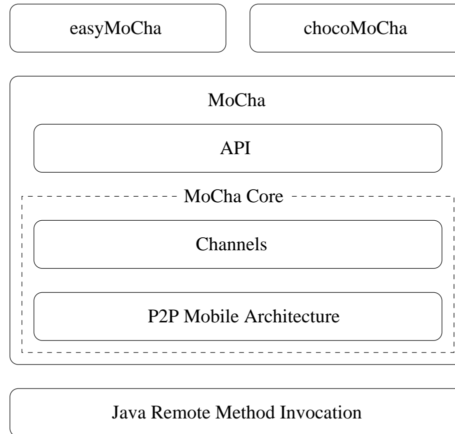


Figure 7.1: The Internal Overview of MoCha

layers.

We begin, in the next section, by motivating our choice for the Java RMI infrastructure. In Section 7.3, we explain the basics of Java RMI. In Section 7.4, we discuss the mobile P2P architecture layer. In Section 7.5, we discuss the implementation of MoCha's channels. Finally, in Section 7.6 , we run several experiments and present the measurements to show the performance of the middleware.

7.1.1 Figure Legend

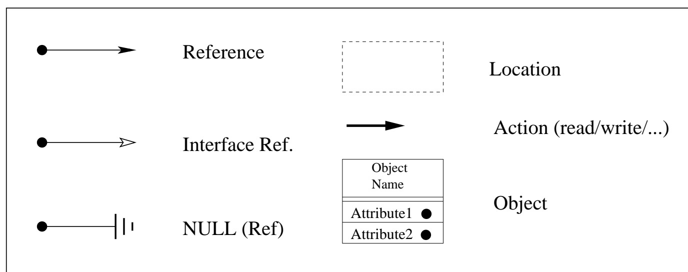


Figure 7.2: Figure Legend

To explain the implementation details we use many figures. In Figure 7.2 we give the legend for the symbols in these figures. The *references* either point to local or to remote entities, denoted by thin arrows within a location or crossing its border, respectively. An *interface reference* is a reference from a component to a channel-end, restricting the access of the component to only pre-defined operations on the channel. A reference to `NULL`, is an attribute that is set to `null`. By *location* we mean a *logical address space* where processes, threads, objects, or components

execute. *Actions* are performed by either components or objects. We represent an *object* by giving its `class` name and its attributes (which are references to other objects).

7.2 Choosing the Right Infrastructure

One of the first choices we had to make, in order to find a good infrastructure to implement MoCha, was choosing the right programming language. Coming from a C++ [Str91] background, this language seemed to be a good choice. However, we decided to implement the middleware in the, back then strong upcoming, Java [Java] language. Java has several benefits, *Gosling* and *McGilton* describe its benefits by using the following buzzwords: *Architecture neutral*, *distributed*, *dynamic*, *interpreted*, *high performance*, *multi-threaded*, *portable*, *robust*, *secure*, and *simple*. For an extensive explanation of these buzzwords, the interested reader is referred to their work in [GM96]. We just briefly discuss those that influenced our choice.

Simplicity is in the eye of the beholder. However, if you are from a C++ background then you will find Java simple to use, for it offers a cleaned-up version of the syntax of C++ and is certainly more programmer friendly. Java is an *interpreted* language, the compiler generates *architectural neutral* code that can run on several machines with different operating systems (OS). Especially this feature is appealing to us, since it means that we just have to compile the middleware once instead of having to produce several versions for different platforms. Java has a nice integrated *multi-threading* system that, in contrast with other programming languages, is easy to use. Interesting for us are the extensive libraries that Java provides for *distributed* environments. Especially for the Internet (which is the biggest distributed system on earth). The Java platform is heavily network oriented. A Java Virtual Machine (JVM), used to execute Java object code, can download classes located on remote hosts and execute them. For example, this feature is used in web browsers to download and execute small pieces of Java code, called *applets*.

Another appealing feature of the Java language is that it is freely, and fully, available at Sun's Java website [Java]. Sun frequently updates Java by creating new libraries, and updating existing ones, in order to support new standards and technologies. Furthermore, most of the applications for which MoCha can be used, are using Java technology. We gave examples of these applications in Section 6.4. Probably, this is also the main reason for other recent coordination middleware systems, like LIME [MPR03], to use Java.

The most primitive and basic distributed communication framework in Java are *sockets* [Bre01]. Virtually any other distributed high-level computing paradigm eventually invokes the socket API to perform the actual distributed communication. Java sockets are directly implemented on top of the socket primitives provided by the underlying OS, and therefore, provide the fastest form of distributed communication in Java. Furthermore, Java sockets support the frequently used TCP/IP [Ste95] communication semantics. For example, the HyperText Transfer Protocol (HTTP) [FGMFB96] is based on the TCP/IP protocol and is usually implemented using sockets.

Java sockets seemed to be a good choice for implementing MoCha. However, there are many drawbacks for doing this. Java sockets are needed if we want to have

full control over the protocol used between the communicating processes. In any other case having to deal with sockets is too cumbersome [Bre01]. The services they provide are aimed solely at transferring and receiving data successfully. Any other service or protocol, like marshalling [Eck98] or remote references, must be build from scratch. Due to their low-level aspect, it is also easy to make mistakes when implementing protocols using sockets. Therefore, developing the MoCha middleware directly on top of sockets would make it an extensive time consuming activity, which is not feasible within the duration of a PhD.

Instead, we looked at *distribution middleware systems* [SS01]. These middleware systems define higher-level distributed programming models whose reusable APIs and components automate and extend the native OS network programming capabilities encapsulated by host infrastructure middleware. Distribution middleware enables clients to program distributed applications much like stand-alone applications, i.e., by invoking operations on target objects without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, and hardware. We considered three such middleware systems: *Corba*, *JMQ*, and *Java RMI*.

The OMG's *Common Object Request Broker Architecture* (CORBA) [TB00], is an architecture and specification for creating, distributing, and managing distributed application objects in a network. CORBA provides platform and location transparency for sharing well-defined objects across a distributed computing platform. The middleware uses an interface definition language (IDL) to specify the interfaces that objects present to the world. CORBA, then, specifies a "mapping" from IDL to a specific implementation language like C++ or Java. In the implementation, CORBA uses an *Object Requested Broker* (ORB), which functions as a broker between clients and servers. ORB provides services as object registration, location, activation, parameter marshalling and un-marshalling, discovery, dynamic invocations, persistent naming, etc.

Java Message Queue (JMQ) [JMQ] enables the transmission of messages between application processes in a distributed environment. With JMQ, processes running in different architectures and operating systems simply connect to the same *virtual network* to send and receive information. JMQ also handles all data translation between application processes. The middleware supports two communication models: the *point-to-point model*, and the *publish-and-subscribe* model. In the first model, a sending process addresses the message to the queue that holds the messages for the intended receiving process. In the second model, a sending process addresses (publishes) the message to a *topic* to which multiple processes can be subscribed.

The *Remote Method Invocation* (RMI) [RMI, HC00] middleware implements the *remote procedural calls* protocol [RPC] for Java objects. The caller must first acquire a reference to the remote object, for example, by looking it up in the RMI bootstrap naming service or by receiving a reference as an argument or a method return value. Using the object reference, a call can be made on the remote server object. The server can in turn be a client of other remote objects. RMI technology uses object serialization to marshal and unmarshal parameters between method calls.

JMQ does not fully adhere to the definition of a distribution middleware, for it does not provide invoking operations on remote objects. On the other hand, it does offer a high-level model that abstracts away from the underlying distributed

platform. However, we didn't choose this middleware for two reasons. First, if we work at the level of messages we might as well work at the level of sockets. With JMQ we need to implement protocols like remote synchronous method calls from scratch. Second, we have no control over the way the middleware sends messages around; if this is done in an inefficient way it would slow down our own middleware.

Both CORBA and RMI are good candidates to implement our coordination middleware with. We chose RMI for several reasons. First, since we already decided to implement our middleware in the Java programming language, we don't need the separation that CORBA offers between the specification layer (IDL) and the implementation layer. Second, we are interested in a simple distribution middleware that remains close to the sockets level but yet offers a higher-level interface. The RPC feature of RMI is sufficient for our purposes, and it seems to be closer to the TCP/IP level than CORBA's ORB. CORBA includes many other mechanisms in its standard (such as a standard for Transaction Processing monitors) none of which is part of Java RMI. There is also no notion of an "object request broker" in Java RMI. Finally, RMI integrates well with other frequently used Java middleware software, for example the Jini technology [Jini].

7.3 Remote Method Invocation

The basic idea of Java's *Remote Method Invocation* (RMI) is quite simple. It makes remote objects appear as if they were local. From the point of view of the user, all method calls on objects are local regardless of whether these objects are on his machine or elsewhere in the network. Naturally, if the method call is on a remote object, somehow the method parameters must be shipped to the other machine, the remote object must be informed to execute the method, and the return value must be shipped back. RMI automatically handles all these details. In this section, we discuss Java's Remote Method Invocation middleware. We give a general overview without going too much into the technical details. The interested reader can find more in depth details about RMI in the tutorials given in [Bre01, HC00, RMI].

In Figure 7.3, we give an overview of the main entities of RMI and how they communicate with each other. RMI is based on a Client/Server architecture [Sch01] (see Sections 2.2.2 and 6.4.3). The straight lines denote the local method calls and the dashed lines denote the actual remote network calls. The object making the method call is the *client object* in RMI terminology. The object executing the requested method is the *server object*. The computer executing the client object is called the *client*, and the computer executing the server object is called the *server*. When the client object wants to invoke a method on a remote object, internally it actually calls a local method of a surrogate object called a *stub*. The stub object acts as a replacement for the actual remote object, for it implements the same interface. Its main functionality is to: (1) initiating a connection with the remote machine containing the remote object. (2) marshalling and transmitting the parameters to the remote machine; *parameter marshalling* is converting the given parameters into a format suitable for transport through a network. (3) waiting for the result of the method invocation. (4) unmarshalling the return value or exception; *unmarshalling* is the reverse action of marschalling. Finally, (5), returning the value to the client object. The stub hides the serialization of parameters and the network-level com-

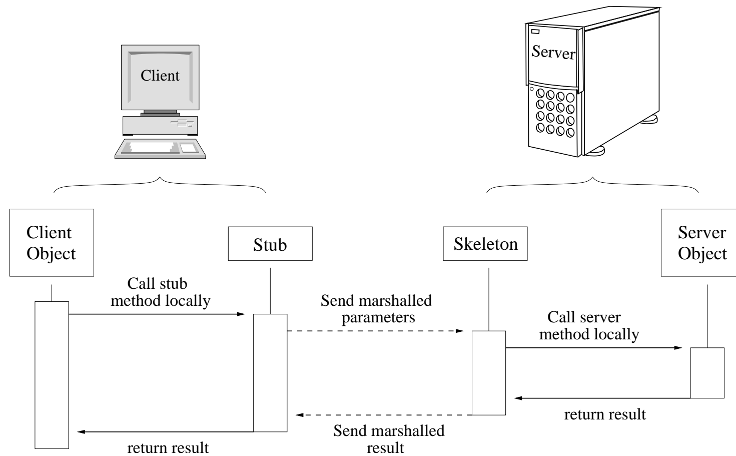


Figure 7.3: Remote Method Invocation Overview

munication in order to present a simple invocation mechanism to the caller. On the remote machine, each remote object has a corresponding *skeleton* object. This skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following: (1) it unmarshals the parameters for the remote method; (2), it invokes the method on the actual remote object implementation and waits for the result; and (3), marshals and transmits the result back to the stub object.

This stub/skeleton model is valid for all Java RMI versions. However, from Java version *v.1.2* (*Java 2 SDK, Standard Edition*) and up no more actual stubs and skeletons are created. Instead a generic object is produced that when executing on the client side it behaves as a stub, and when executing on the server side behaves like a skeleton. To make things confusing Java calls this generic object also a *stub*.

7.4 MoCha's Mobile P2P Architecture

In this section, we describe MoCha's P2P mobile architecture. We do this in two steps, first we present the P2P architecture and then we extend this architecture with support for mobility.

7.4.1 Using RMI for P2P Networking

As described in the previous section (Section 7.3), RMI implements a Client/Server architecture. The JVM containing *client objects* is the *client*, and the JVM containing *server objects* is the *server*. As illustrated in Figure 7.4(a), a server can have many clients, whose client objects perform remote method calls on the available server objects. Servers are processes dedicated to specific tasks like managing of disk drives, printers, or network traffic, whereas clients are processes that rely on servers for resources. The clients themselves do not share any resources. For

example, we can think of the lightweight Java *applets* client objects that run on web browsers and access remote *database* server objects. Regarding the MoCha middleware, the problem with this architecture is that all the resources are *centralized* on the servers. Implementing our point-to-point mobile channels in such an architecture makes the middleware inefficient. Instead, we need a P2P framework to implement MoCha's mobile channels. In a P2P (*peer-to-peer*) architecture each node is both a client and a server at the same time (for more details about P2P architectures see Section 6.4.3).

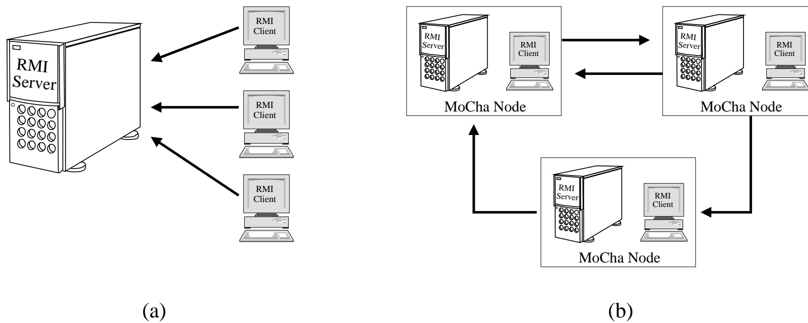


Figure 7.4: Constructing a Peer-to-Peer Architecture

In RMI it is possible to define objects that are both a client and a server object at the same time (although when designing RMI it was not meant to in the first place). We call such objects *node objects*. We extend the RMI architecture by converting all remote objects into standard node objects. As illustrated in Figure 7.4(b), all the JVMs that use the MoCha middleware are considered to be *MoCha nodes*. The node objects of these MoCha nodes may have references to other remote objects on other nodes and call their methods. Naturally, not all node objects need to have references to each other. Also, in this P2P architecture the node objects may pass third party remote references to each other in order to dynamically change the remote reference connections within the network, i.e. spreading the knowledge of remote references. By passing a remote reference we mean that we copy and pass stub objects that themselves refer to remote objects. However, this passing of remote references causes a problem in some cases. Next, we discuss this problem and how we extended the architecture in order to fix this.

The Distributed Garbage Collector Problem

In the Java language there are no methods for explicit memory management like, for example, in C++. There is no need for allocating memory to objects, freeing memory by explicitly destroying objects, and keeping track of what memory can be freed when needed. Instead Java has a *garbage collector* that does this automatically and implicitly in the background. Once an object is created the run-time system keeps track of the object's status and automatically reclaims memory when it is no longer in use (and the memory space is needed). To determine whether an object is no longer in use, the garbage collector keeps track of all the references to each

object in the system. When an object has no more references pointing to it by other objects it is no longer in use and, therefore, candidate for garbage collection.

The distributed RMI garbage collector is the union of all the local garbage collectors of each JVM plus a mechanism to determine whether a remote reference is still valid. A remote object is garbage collected when there are no local nor remote references pointing to it. For each *server object* there is a *counting mechanism* to detect whether there are still *client objects* referring to it. When a client object obtains a remote reference, RMI sends a `dirty()` message to the remote server object, which gives the client a *lease* for this object. The lease expires after a specific time, so the client JVM has to automatically renew the lease by periodically sending `dirty()` messages to the server object. If the client object drops the reference, the client JVM sends a `clean()` message to indicate this loss. If the server object receives a `clean()` message or if a `dirty()` message does not arrive (on time) due to network problems, its reference count is decreased by one. If the reference count reaches zero, then RMI indicates to the local garbage collector that the server object is ready for garbage collection.

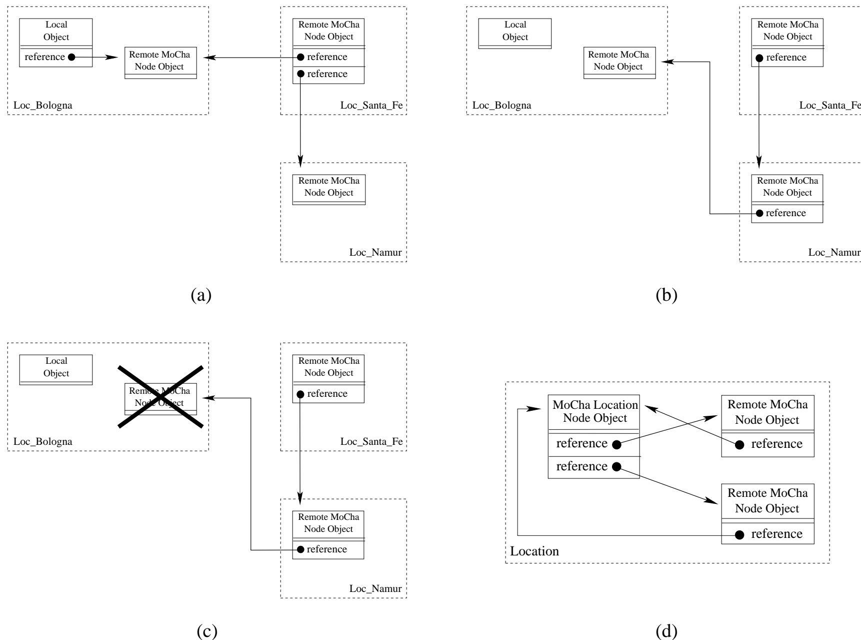


Figure 7.5: The Distributed GC Problem

The *lease* time mechanism of RMI works well in Client/Server architectures. Since the server objects have all the resources, they usually have local references pointing to them and are, therefore, never illegitimately garbage collected. In our MoCha P2P framework extension we can have node objects at a certain JVM that are referred to only by remote references. If the `dirty()` messages do not arrive on time due to network problems, these objects are illegitimate garbage collected, causing an internal MoCha error when trying to access their methods afterward.

Network problems are not the only possibility for the late arrival of these messages. In our P2P architecture it is possible to pass on third party references, i.e. passing stub objects to other node objects in the network. If while we are passing a particular stub no other stubs exists that send `dirty()` messages, then the remote node object may also get illegitimately garbage collected. We illustrate this situation in Figure 7.5(a). The dashed square denotes a location, the lines denote a remote reference, i.e. the object holding the remote reference has a local reference to a stub that refers to the remote object. In this particular example we have three locations. We focus on the remote MoCha node object in Bologna. This object is referred to by two other objects: a local object, and a node object at the Santa Fe location. This last object also has a remote reference to a node object at the Namur location. Now, suppose that at some point in time the local object drops its reference to our node object. Then, we must hope that the distance between Bologna and Santa Fe is not too long for the `dirty()` messages to arrive on time to our node object. Suppose that this goes well and that now the node object in Santa Fe sends the stub pointing to our object to the node object in Namur. Furthermore, after doing this, the Santa Fe node object also drops its reference to our object. Then, there are two possible outcomes. If we are lucky, we get the situation of Figure 7.5(b), where the Namur node object has a remote reference to our still existing Bologna node object. However, we can also get the situation of figure 7.5(c), where due to the distance or traffic between Santa Fe and Namur, before the stub gets to Namur and the JVM there is able to send a `dirty()` message, our Bologna object got garbage collected.

To solve the problem of illegitimate garbage collection of MoCha node objects, we need an object that locally refers to these node objects all the time. Thankfully, at each JVM there is always at least one internal MoCha node object present: the *location object*. This object is explicitly created by the components of a particular JVM and is used for channel creation and movement of channel-ends (see Section 6.2.1). We extend our architecture by letting location objects to locally refer to the internal MoCha node objects (see Figure 7.5(d)). At the same time, all MoCha node objects have a reference to their current location (object). At channel creation time, the channel-ends and other created internal objects (if present) report themselves to the (local to them) location object so that it can refer to them. We want these references to always be local, therefore, when a channel-end moves, it and all necessary internal objects check themselves out of current local location object and reports themselves to the location object at the new destination (we discuss object movement in Section 7.4.2). In this way, using the location objects we prevent illegitimate garbage collection.

7.4.2 Mobility on top of RMI

The channels of the MoCha middleware are *mobile* (see Sections 2.2.2 and 2.3). This means, (1) spreading the knowledge of channel-end references in the system through channels themselves, and (2) physically moving a channel-end from one location to another location in a distributed system. The first property is handled by channels themselves and is already supported by our RMI MoCha P2P platform. However, mobility of objects is not a feature of RMI, therefore, we extend our architecture to support object mobility.

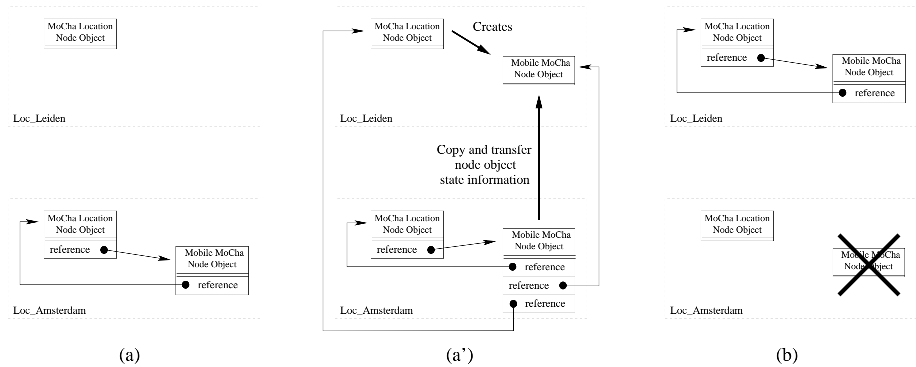


Figure 7.6: Moving a MoCha Node Object

Not only channel-ends can move in MoCha, but depending on the type of the channel, internal data structures associated with particular channel-ends may also move when necessary. Thus, the extension we make is not only meant for channel-ends but for any MoCha node object in general. We specify a moving protocol where we use the location node object. It makes sense to use this object, since a channel-end always has a reference to a local instance of such an object, and since the move operation requires the target location as a parameter (see Section 6.2.3), we have a reference to the new location available during the movement process.

We illustrate our protocol in Figure 7.6. Suppose we have a mobile node object that wants to move from Amsterdam to Leiden. Naturally, in Amsterdam it has a reference to a local location object, that in turn refers back to our mobile node object (as explained in Section 7.4.1 and illustrated in Figure 7.6(a)). Due to the move operation, our mobile object acquires a reference to the target remote location object. This enables our object to ask this location to create a new object of the same type as his. After creation, the target location passes on the reference of the "new" node object to the "old" one. Our Amsterdam object, then, invokes a special method that copies its current object state and transfers it to the Leiden object. The result is an identical copy of the original mobile node object (see Figure 7.6(a')). To complete the process, both the location object and the "old" node object at Amsterdam drop their mutual references (see Figure 7.6(b)). To facilitate garbage collection of other objects, our "old" mobile object drops all references it has to any object in the system. Our Amsterdam mobile node object is now ready to get garbage collected. After this, finally, the "new" mobile node object in Leiden reports itself to the new location so that both objects refer to each other as was the case with the old object at the (initial) Amsterdam location but now at Leiden.

Coping with Dangling and Invalid References

Channel-end movement leads to dangling and invalid references. For example, suppose that the Amsterdam mobile MoCha node object in Figure 7.6(b) is a channel-end, and that there are still components that have references pointing to it. Then, these references are either *invalid* if the channel-end is still not garbage collected, or

dangling if this channel-end was garbage collected. In both cases, the components get an exception error from the *"plain"* MoCha middleware.

However, the *easyMoCha* and *chocoMoCha* middleware versions internally update their channel-end references so that when a component acquires a reference to a particular channel-end it always remains valid (see Chapter 6). To accomplish this, we make a small change in the protocol. If the mobile node object is a channel-end, then, we don't entirely leave it behind for garbage collection. Instead we make it drop all references it may have to any object, except to the new copied object (see Figure 7.6a'). At the same time, we give it a tag to denote that this particular channel-end is invalid. Components do not directly refer to a channel-end but to a *local MoCha proxy*, that in turn refers to the "real" channel-end (see Section 7.5.1). Therefore, when a component tries to access an invalid channel-end, the local proxy notices that the channel-end is invalid, asks and gets the reference to the "new" channel-end, updates its own reference, and continues with the operation that the component wants to perform. Only when all local proxies update their references to the "new" channel-end, the "old" one is ready to get garbage collected.

Naturally, if a channel-end moves a lot, a large chain of invalid channel-ends may emerge if components have references to the "old" instances of this end but never use them. However, the invalid channel-ends are lightweight objects that don't consume much local resources. So even if we get big chains of invalid channel-ends, this won't affect the performance of the JVMs that much. Nevertheless, we encourage components to drop their channel-end references if they don't need them anymore.

7.5 MoCha's Channels

Now that we constructed a P2P mobile architecture on top of Java RMI, we are ready to implement our mobile channels on top of it. In this section, we first describe the general internal overview of a channel. We then give the implementation of the FIFO channel type as an example. Finally, we discuss several small but interesting implementation issues regarding channels.

7.5.1 General Mobile Channel Implementation Overview

In Figure 7.7 we show how a mobile channel is realized in the MoCha middleware. A channel consists of exactly two distinct ends, which are either of type *source* or *sink*. There is no *channel object* that relates the two ends. Introducing such an object gives problems such as: at which location do we put this object? It also makes the internal communication less efficient since the ends need to indirectly communicate with each other through this *channel object* (which can be at a different location as the two ends). Instead, in MoCha the two ends directly refer to each other (the `other_e_rf` reference). In this more efficient way, the channel as a whole is just a concept, while internally the channel-ends are two separate objects that are related to each other. Depending on the type of the channel, the channel-ends *may* refer to a (distributed or local) buffer. For example, this is the case for the asynchronous FIFO channel types, but not for the synchronous types. That is why in the figure these references are dashed. Naturally, channel-ends may refer to other internal (MoCha) objects depending on their channel type.

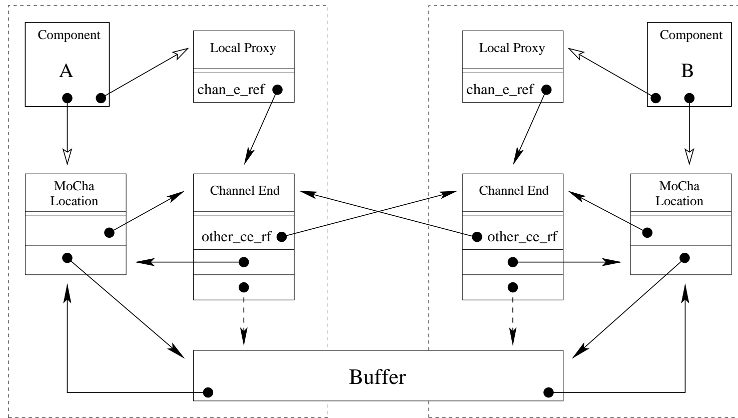


Figure 7.7: A Channel in MoCha

Channel-ends and other internal objects, like buffers, are (mobile) MoCha node objects. Therefore, they are all referred to by a MoCha location object, and they all refer back to such an object (as explained in the previous section). Components also work and know MoCha location objects (as explained in Section 6.2.1). They have at least one interface reference to such an object. Obviously, components also know and work with channel-ends. However, they do not directly refer to “real” channel-ends. Instead, they have interface references to *local proxies*, which internally refer to the channel-ends. As shown in Figure 7.8, a channel-end can have many local proxies, but a proxy refers to exactly one channel-end. Many components at the same or different locations can have references to the local proxies of a particular channel-end. These components don’t know that they are referring to such proxies; as far as they are concerned they refer to local channel-end objects. The fact that the channel-end operations that these components perform may involve remote communication is internally taken care of by the MoCha middleware.

Our motivation for introducing local proxies for channel-ends is to reduce the internal network communication. For example, there are some *inquiry operations* (see Section 6.2.3) that are locally handled by the proxy without needing to access the (remote) channel-end. Such operations include, for example, `isSourceEnd()`, `equals(ChannelEnd ce)`, and `equalsChannel(ChannelEnd ce)`. Also, local proxies provide load balancing of `connect` operations. If we allow components to directly perform these operations on channel-ends, then eventually the channel-ends need to keep a big queue of component identities (and references) waiting to get access to them. The bigger this waiting queue is, the more internal network communication we get (for example, due to RMI internal update messages). To reduce this communication, the local proxies implement their own local waiting queues. All the components at a particular location performing `connect` operations on the same channel-end wait on this local proxy queue. The proxy, then, selects one component at the time that goes through to the waiting queue of the channel-end. The proxy selects another component to go through only when the previous component gets channel-end access and (after a while) releases it by performing a `disconnect` op-

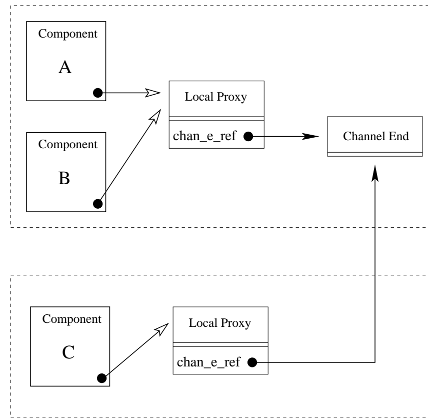


Figure 7.8: Local Proxies for Channel-ends

eration. This protocol considerably reduces internal network communication due to smaller waiting queues on channel-ends.

7.5.2 Example: FIFO Channel Implementation

In Figure 7.7 we gave the general overview of how channels are implemented in the MoCha middleware. We now present a more detailed example of one of these channels: the *unbounded FIFO* channel type (see Section 2.4). With this channel type, the I/O operations performed on the two channel-ends succeed asynchronously. Values written into the source channel-end are stored in the channel in a FIFO (First In, First Out) distributed buffer until taken from the sink-end.

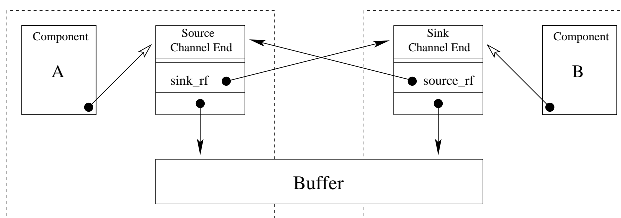


Figure 7.9: FIFO Channel Type Overview

To simplify our explanation, we abstract away from *MoCha location* and *local proxy* objects, in order to focus on the details of the FIFO channel implementation algorithms. We give a simplified overview in Figure 7.9, where we pretend that components have a direct interface reference to channel-ends. With this channel type, there is a *sink* and a *source* channel-end present that refer to each other (the *sink_rf*- and *source_rf*-fields in the figure). There is also a distributed buffer present, whose implementation consists of a chain of local FIFO buffers that are distributed

over the system. These local buffers, and other fields, are gradually introduced in our explanation.

Our approach is to describe the abstract operations: *create channel*, *write*, *take*, and *move*. We divide our explanation of the I/O operations into two parts: with and without mobility. Afterward, we discuss details like heuristics and clean-up of buffers. We do not explain other operations like *connect*, *read*, *empty*, etc., because the operations that we selected are representative enough to give a good intuition of how this channel type is implemented.

The interested reader can find more about the implementation details of this channel type, as well as about other channel types, in the abstract algorithms of Appendix A.

Towards an Efficient Implementation

In a distributed environment, data-transfer between different locations is more costly with respect to time than data-transfer within a location. Therefore, an efficient implementation must reduce the amount of non-local data-transfer. Trivial implementations of FIFO mobile channel types in distributed environments, such as a centralized buffer for which (virtually) every *take* and *write* operation is non-local, clearly do not meet this demand. Our implementation reduces non-local data-transfers by allowing the distribution of data all over the system, and avoiding any kind of centralized control. The movement of data is minimized by making sure that:

- Every write is always a local operation.
- A take is mostly a local operation. We try to reduce the number of non-local take operations by using heuristics.
- Moving the channel-ends does not involve any transfer of data elements at all.

Create Channel

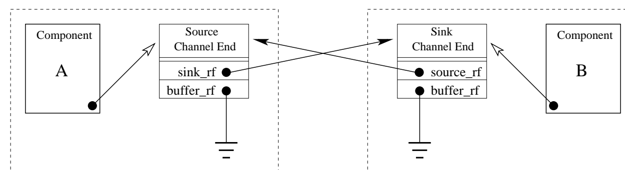


Figure 7.10: FIFO Channel Creation

Upon creation of a new channel, its channel-ends (its *sink* and its *source*) are created at two given locations, which need not be distinct. A reference to each channel-end is then returned. We sometimes say “a component holds a channel-end” when it has access to this channel-end.

An example of creation is shown in Figure 7.10. In this figure, components A and B need not be distinct, because the same component can hold both channel-ends. At creation, the channel-end fields *sink_rf* and *source_rf* are set in the proper

way (pointing to each other), and the *buffer_rf*-fields are set to `NULL`. The reason for doing this is that there is no guarantee that the component that initially holds a channel-end, actually takes or writes any data before the channel-end is moved to another location. Therefore, creating a buffer is done only when really needed, which is more efficient (examples follow).

Write (Without Mobility)

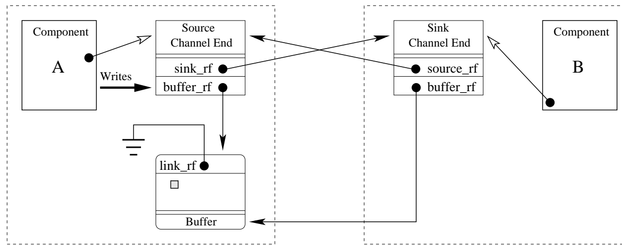


Figure 7.11: First Write Action

In Figure 7.11, component A starts to write for the first time. Because its *buffer_rf*-field is `NULL` the source creates a local unbounded FIFO buffer. The buffer has a field called *link_rf*, which is a reference to another buffer. Because there is just one buffer now, this field is set to `NULL`.

For its internal consistency, MoCha requires an *invariant* to hold on the *buffer_rf*-fields of the two ends of a channel: they must be either both `NULL` or both `non-NULL`. In Figure 7.10 the channel is just created and the *buffer_rf*-fields are, initially, `NULL`. Therefore, after the creation of the first buffer of the channel, they must be both `non-NULL`. The source notifies the sink about the new buffer, and the sink updates its *buffer_rf*-field to point to this new buffer in order to satisfy the invariant.

The first buffer of a channel is always created by its source-end. After the first write action, there is always a local buffer at the same location as the source.

Take (Without Mobility)

In Figure 7.10 both *buffer_rf*-fields are `NULL`. If component B wants to take, the sink-end will notice its *buffer_rf*-field is `NULL` and conclude that there are no elements in the channel. It then waits for a first write.

In Figure 7.11 the *buffer_rf*-field of the sink is not `NULL` anymore, but it points to a non-local buffer. If B attempts to take now the sink creates a new local buffer. The *link_rf*-field of this buffer is set to point to the old buffer (where the *buffer_rf*-field of the sink points to), and the *buffer_rf*-field is updated to point to the new local buffer. The result is shown in Figure 7.12. If A and B are at the same location, then no new local buffer is created, because the *buffer_rf*-field of the sink already points to a local one. After the first take action, there is always a local buffer at the same location as the sink.

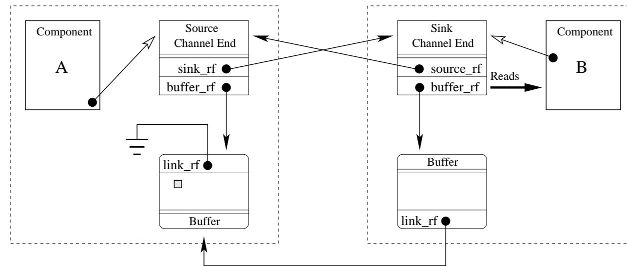


Figure 7.12: First Take Action

Move

We already discussed how a channel-end moves from one location to another in Section 7.4.2. For the FIFO channel type this involves additional considerations. When a channel-end moves to a component in a different location, its *buffer_rf*-field reference changes from local to non-local. Because a channel-end can move among several components before one of them actually uses it, no buffers are moved or created in MoCha by move operations. A buffer is created in MoCha only when necessary: when a component actually starts to read or write. After a move, a channel-end notifies the other end of the channel of its new location. Examples of move follow.

Write (with mobility)

When writing, a local buffer is created at the location of the source if its *buffer_rf*-field is either `NULL` or pointing to a non-local buffer. If the *buffer_rf*-field is `NULL`, the source notifies the sink about its newly created local buffer (as explained before).

Figure 7.13 is an example of writing after the source end of a channel is moved to a new location. At some point in time component A has the source channel-end and writes some elements into the channel. Then the channel-end moves to component B in another location, at which time no buffer is created or moved. Then component B starts to write (Figure 7.13a). The *buffer_rf*-reference of the source is non-local, therefore a new local buffer is created. Both the *buffer_rf*-field, and the *link_rf*-field of the old buffer (the buffer where the *buffer_rf*-field points to) are changed to point to the new buffer. The result is Figure 7.13b. The data written by component B can now be inserted into this new local buffer.

Take (with mobility)

When taking, a local buffer is created at the location of the sink if its *buffer_rf*-field points to a non-local buffer. If the *buffer_rf*-field points to a local buffer, then taking can proceed. If it is `NULL`, then no elements exist in the channel and the sink waits.

Figure 7.14 shows an example of taking after the sink end of a channel is moved to a new location. Component C holds the sink channel-end right after the move, the sink moved. In Figure 7.14 a component C starts to take. The *buffer_rf*-reference of the sink is non-local, therefore a new local buffer is created. The *link_rf*-field of the

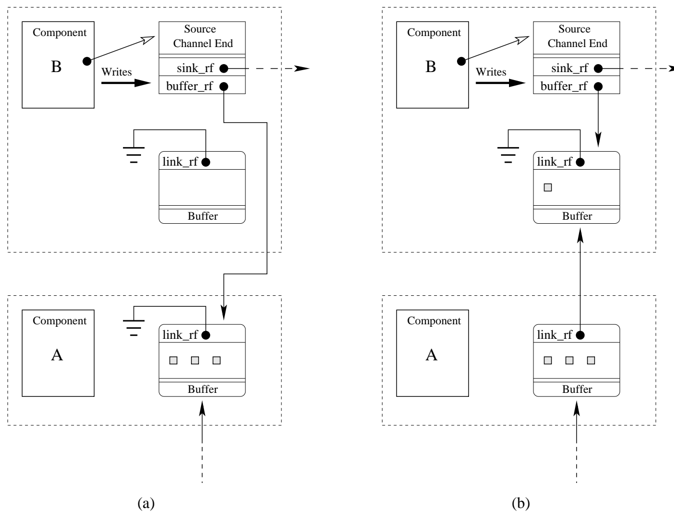


Figure 7.13: Writing after Movement of Source-end

new buffer is changed to point to the old buffer (the buffer where the *buffer_rf*-field points to), and the *buffer_rf*-field is changed to point to the new buffer. The result is shown Figure 7.14b. The taking can now begin. More details on take follow.

Chain of local buffers

The mobility of channel-ends in combination with the actions write and take can lead to a *chain* of buffers that is distributed over many locations. Figure 7.15 shows the general case. The components need not be distinct and the *buffer_rf*-field references need not be local. The *first* buffer of the chain is the one where the *buffer_rf*-field of the sink points to, it contains the oldest elements inserted in the channel. The *last* buffer is the one where the *buffer_rf*-field of the source points to, it contains the most recent elements inserted in the channel. In this way the FIFO structure of the channel is preserved.

In Figure 7.16 a particular instance of the general case (Figure 7.15) is given. There are two buffers at the location of component B, because the source-end moved twice between the locations of component B and A. Component B has no access or knowledge of the buffers (only channel-ends can access the buffers). At the location of component F there are also two buffers. This situation can exist only if a buffer, between the two now present, has been garbage collected (we explain buffer garbage collection shortly).

How data elements move

We showed how local buffers are created as a consequence of taking and writing. For writing this is sufficient because this action just creates a local buffer, sets ups the references, and fills it with elements. For taking, however, we still need to explain

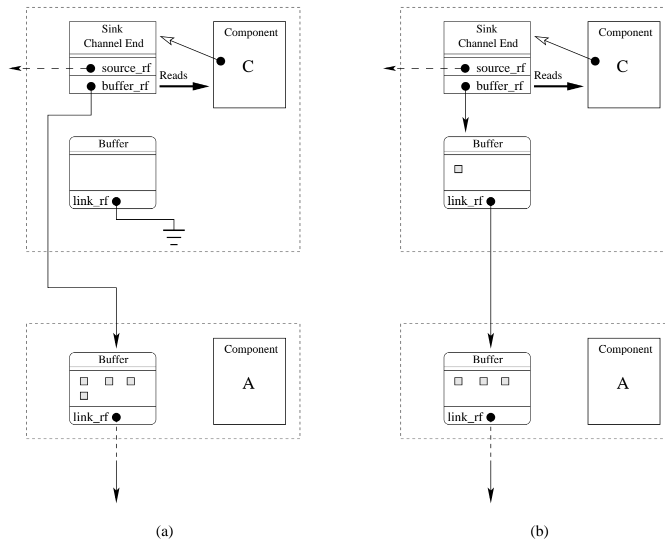


Figure 7.14: Taking after Movement of Sink-end

how elements are moved through the chain of buffers.

The buffers in MoCha support additional functionality beyond the typical FIFO buffers. A buffer is either a *sink-buffer*, or a normal MoCha buffer. A sink-buffer is one that is *local* to the sink-end of a channel. A channel can have at most one sink-buffer at any given time, but any buffer in its chain can become its sink-buffer at some point in time.

When reading, the sink asks the sink-buffer for an element. If the sink-buffer contains at least one element, it gives an element to the sink. If it is empty, it asks the next buffer in the chain (if any) for a certain number of elements. This number is determined by the take heuristics (which we explain next). The sink-buffer receives from this next buffer either (1) the requested number of, (2) fewer or, (3) no elements. The sink-buffer also receives a reference that is either:

- `NULL`, which indicates that the buffer is either not empty after it provided these elements, or it is the last buffer of the chain. When receiving a `NULL` reference, the sink-buffer does not modify its *link_rf*-field.

- `non-NULL`, which indicates that the buffer is now (or was) empty. The reference points to the next buffer in the chain, the sink-buffer receiving a `non-NULL` reference changes its *link_rf*-field to point to this new value.

Take Heuristics

For efficiency, sink-buffers use certain heuristics to determine the number of elements they request from their next buffers in their chains. For this purpose, the sink channel-end has an extra field called *consumed*, that keeps track of how many data

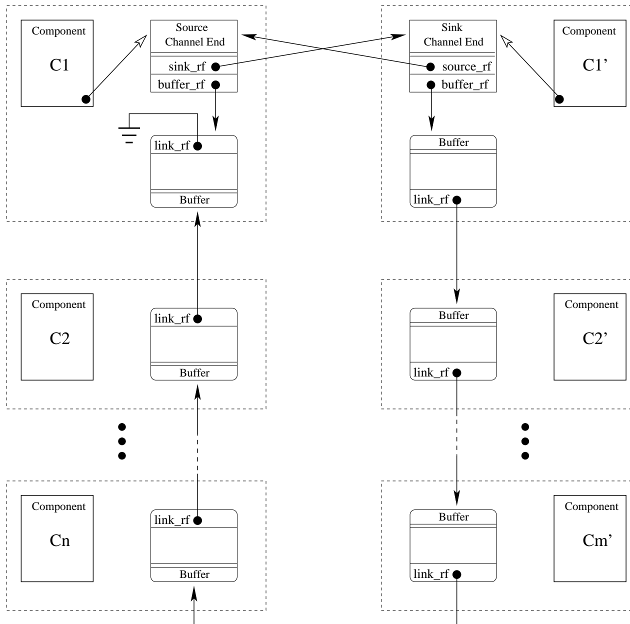


Figure 7.15: Chain of Buffers

elements the current component holding the sink has already consumed. Initially, and every time the channel-end moves, this variable is set to zero.

Knowing nothing of the behavior of the system it is reasonable to assume that the component performing a take is going to consume (in total) *twice* the amount it has already consumed. This is similar to a good heuristic rule used in dynamic memory management. By this heuristic, the number of elements to be requested is *consumed + 1*.

Figure 7.17 shows an example. Component C holds the sink channel-end and has already consumed 3 elements. It now asks for another element. The sink-buffer is empty, therefore it requests elements from the next buffer in the chain. We assume that the component is going to consume a total of 7 elements ($2*3 + 1$), therefore the sink-buffer requests 4 elements (`consumed + 1`). In Figure 7.17b the 4 elements of the non-local buffer are transfered to the sink-buffer. An element is then given to the component by the sink, which is not shown in the figure.

The rationale for asking for more data elements than needed (just 1) is the fact that the cost of communication in a distributed system is not really dependent on the amount of transfered data, but rather, on the number of exchanged messages. Consequently, within reasonable bounds, it costs the same to exchange a short message as to exchange a long one. Furthermore, the majority of the exchanged messages in a system are well below those "reasonable bounds" and leave some considerable amount of free bandwidth that can be used to transfer additional data at no extra cost.

Asking for more data items than it needs, the sink-buffer informs the remote

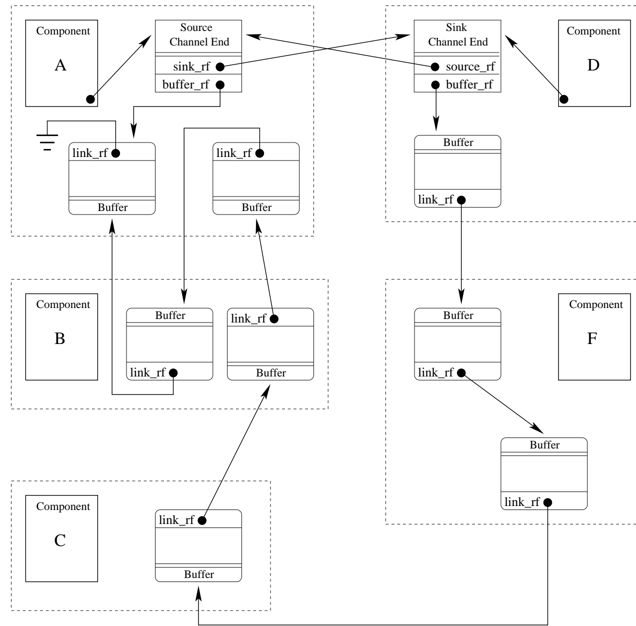


Figure 7.16: An instance of the General Case

buffer of the likelihood that it will consume those additional elements in the near future. The remote buffer, then, determines the maximum number of data items, up to the requested amount, that fit in the free bandwidth of one message and packs and sends at most that many, if it indeed has that many.

Observe that it makes no sense for the remote buffer to try to send more data elements that it actually contains, by obtaining them from its next buffer in the chain. Also, it is not necessarily a good idea for the remote buffer to ignore the (heuristically determined) number of requested data items and always "flush" itself sending its entire contents to the sink-buffer, because the sink-end may move to the same location as the remote buffer before all those elements are consumed.

Garbage Collection of Buffers

A buffer can become empty when elements are taken out of it. An empty buffer is no longer of use, unless it is the first or the last buffer of the chain. Therefore, an empty buffer prepares itself for garbage collection when it gives its *link_rf*-reference (the reference to the next buffer in the chain) to the sink-buffer, unless it is the last buffer of the chain (in which case its *link_rf*-reference is `NULL`). We already described the behavior of the buffers above.

In Figure 7.18, a buffer has become empty due to a take operation. The buffer detects this and gives the sink-buffer not only the requested data elements but also a reference to its next buffer in the chain. The sink-buffer updates its *link_rf*-field to this next buffer. The empty buffer, having passed its next-buffer reference to the

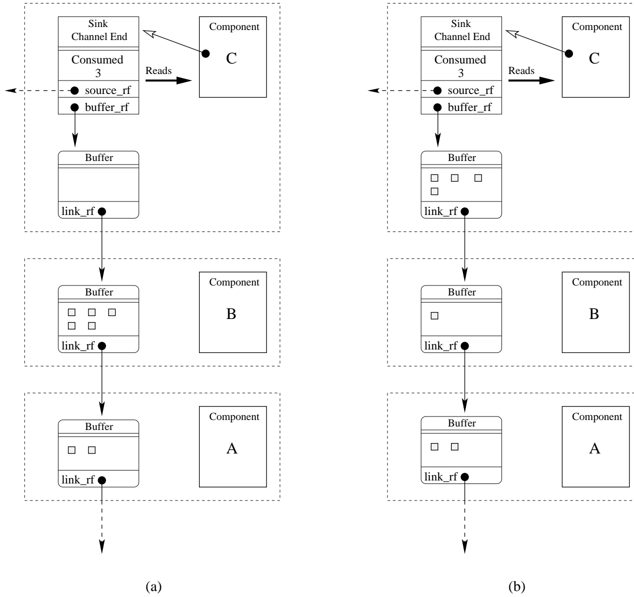


Figure 7.17: Heuristic of the Sink Buffer

sink-buffer, concludes that it can safely destroy itself because it is no longer part of the chain. It sets all its references to NULL and knowing that no other object refers to it, it can safely wait for garbage collection by the local JVM.

7.5.3 Channel Implementation Issues

While implementing the mobile channels of the MoCha framework in Java, we encountered several small interesting problems and issues. In this section, we describe three such problems and explain how we solved them. Our intention is to give the reader an idea of what kind of difficulties, other than the ones described in the previous sections, we had to face. Furthermore, the reader can benefit from the solutions we describe here for his own implementation purposes.

Forcing Local Deep Copy of Objects

Our main philosophy towards components is that no object from outside should be able to refer to an object inside a component. Therefore, every object that is transmitted through MoCha channels is deep copied [Eck98]. This deep copy ensures that not only the written object, but also every other object that it refers to, is copied. Therefore, a whole tree of objects is copied if necessary. The MoCha channel-ends and locations are the only objects that are not (deep) copied. Otherwise, we would have implicit channel(-end) and location creation as a side effect.

RMI automatically makes a deep copy of every object that is *mashalled* and sent to a remote object somewhere in the network. However, if the communication is not through the network, RMI *does not* copy the transmitted object. This gives

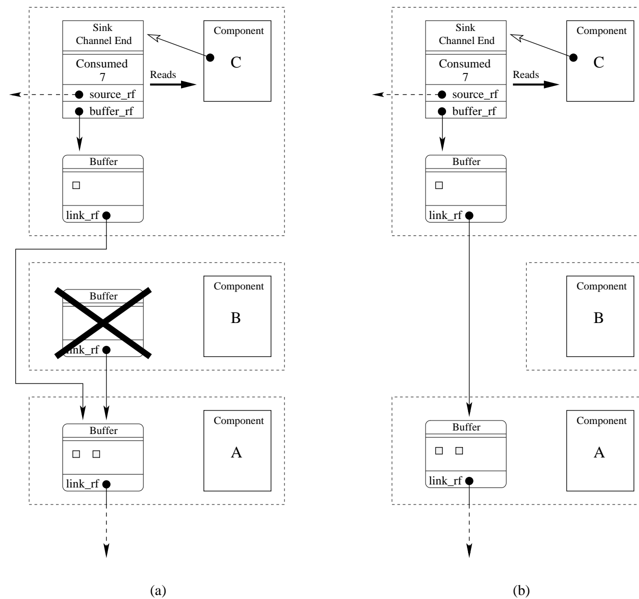


Figure 7.18: Garbage Collection of Buffers

problems in certain situations. For example, when two components and a channel between them are at the same location. If in this situation an object is sent, then it is by reference. Thus, we end up with a component that unintentionally refers to an internal object of another component. Another example is the implementation of the FIFO channel type (see the previous section). The source channel-end always writes its data to an internal local buffer. An object written into this buffer may refer to objects inside a component, until it is transmitted through the network to another buffer or the sink channel-end. To solve this problem, the MoCha middleware checks for situations where RMI does not deep copy the transmitted objects. If this is the case, MoCha enforces the deep copy by marshalling the objects itself. This way, we guarantee to always get a deep copy of the original object.

Recognizing Object Types

The *Filter channel* behaves like a *synchronous* type (see Section 2.4). However, values that do not match the channel's pattern are filtered out (lost). This pattern is given to the middleware as a string (see Section 6.2.2). For example, `"Filter java.lang.Integer java.lang.Double mypackage.myclass"`. The middleware, then, parses the string and automatically extracts the pattern out of it. In this case, the channel accepts objects of type `Integer`, `Double`, and `myclass` (which is a user-defined class). Thus, the pattern can be any Java class including non-standard ones made by users.

The filter channel type must *dynamically convert* the string into a valid class definition, and *dynamically check* if a given object matches the pattern. In a non-

interpreted programming language like C++ this is very difficult to do, because there is no static solution possible. Therefore, we need to (somehow) build an infrastructure to have type information available during runtime, and the possibility of comparing this information with class instances. The fact that we can specify user-defined classes in the pattern, makes it even more difficult for such programming languages.

Thankfully, the Java language is *interpreted* (see Section 7.1). The Java Virtual Machine (JVM) contains all the dynamic information that we need. Therefore, we implemented above pattern matching by using the special Java system class `Class` [Eck98] whose methods query the JVM for certain dynamic system information. In particular, the instances of this class represent classes and interfaces in a running Java application. We use the static method `forName` that takes a string as parameter and returns an instance of `Class` that is associated with the class or interface with the given string name. Then, we use the `isInstance` method, that takes an object and determines if this object is assignment-compatible with the object represented by a particular instance of `Class`. Naturally, MoCha takes care of situations where (a subset of) the classes given in the pattern do not exist at the current location. In this case, the filter channel type simply discards these classes from the filter for this particular location.

Adding Semaphores to Java

For the abstract channel algorithms (given in Appendix A) we use *semaphores* [Dijk68] as our synchronization primitive. For our Java implementation we also used such primitives to make the algorithms more straightforward to implement and relate with the Java code. However, Java does not provide semaphores. Instead, the language provides *monitors* [BG94]. Therefore, we had to make our own semaphores in Java. Afterward, several colleagues were interested in this result for use in their own projects. This motivates us to briefly explain how we implemented semaphores in Java by using monitors.

A semaphore is a non-negative shared integer variable with two operations, V and P. For a semaphore s, V(s) increments the variable with one, and P(s) tries to decrement the variable by one; if the variable is zero, however, it blocks until the variable becomes non-zero first. The two operations are indivisible, so if several processes (threads) simultaneously try to execute any of these two operations on the same semaphore, the operations are still executed one at a time.

A monitor is similar to an abstract type. It encapsulates shared data and operations on these data. The distinguishing feature of a monitor is the guarantee that at any given time only one process can be executing any of the operations in the monitor; when a process is inside a monitor, it can be certain it is the only one active there. It is not always possible for a process to leave a monitor on time so that other processes can get access to it. For example, it may wait for some data to arrive. Therefore, the monitor has a *condition variable* with two operations defined on it, *wait* and *signal*. *Wait* blocks the invoking process, and *signal* reactivates one process blocked in a *wait*. If the current active process in the monitor is blocked by a *wait*, then some other process is allowed to enter the monitor. Upon reactivation, by *signal*, of a process it has to wait until it gets exclusive access to the monitor

again before continuing its execution.

```
/** Semaphore
* @author Juan Vicente Guillen-Scholten */
class Semaphore
{
   /** The internal shared integer variable.*/
   private int locked;
   /** default constructor!
   Semaphore starts unlocked.*/
   Semaphore () {
      locked = 1;
   }//contructor

   synchronized void lock() {
      if (locked == 0) {wait();}
      locked--;
   }// lock

   synchronized void unLock() {
      if (locked == 0) {
         locked++;
         notify();
      }//fi
   }// unLock
}//Semaphore
```

Figure 7.19: A Binary Semaphore in Java

In Java, a monitor is a normal class whose methods have the syntactic keyword "*synchronized*" written in front of their statement. In Figure 7.19, we give our semaphore implementation (where we abstract away from details like exceptions). The `constructor` method sets the private variable `locked` to one, this indicates that the semaphore is initially free. Our semaphore has two methods, `lock` and `unLock`, that correspond to the semaphore operations described above, respectively with P and V. The fact that these methods are *synchronized* guarantees the atomicity of their execution. When a thread executes the `lock` method two things can happen. (1) The semaphore is free (the `locked` variable is one), so the thread locks the semaphore for others (by setting the `locked` variable to zero), and successfully terminates the method. (2) the semaphore is already taken by another thread (the `locked` variable is zero), then our thread performs a *wait* operation and places itself in the waiting queue of the monitor.

When a thread executes the `unLock` method two things can happen. (1) If the semaphore is currently taken (the `locked` variable is zero), it frees the semaphore (by setting the `locked` variable to one), and releases one thread from the waiting queue by performing a *signal* operation (in Java it executes the `notify` method). (2) The semaphore is currently free (the `locked` variable is one), then nothing happens, the thread must have made a mistake. Although the programmer is responsible for

making proper lock and unlock pairs, we prefer this behavior than returning an error.

7.6 Performance Measurements

To determine the performance of the MoCha middleware we run several experiments on a computer cluster within the *CWI institute*. MoCha is meant for running on the Internet and on Intranet networks. However, it is really difficult to run experiments on these networks in such a way that we get consistent results. Therefore, we use a computer cluster instead. The difference is that, because the computers of a cluster are all physically at the same location the communication between them is much faster than in a Internet/Intranet network. Nevertheless, the results from our experiments give a good indication of MoCha's performance because we can scale them to the target networks (in which the communication is much slower).

Our hardware configuration consists of 11 equally equipped computers with the following specifications: A *32/64bit Athlon 2.2Ghz* processor, *1 Gigabyte* of memory, *48 Gigabyte* of scratch disk, a *64-bit* communication port, and *one GigabitEthernet* card. Each computer runs the *Suse* Linux operation system version *9.3*, and the *Java* language platform version *"1.4.2_08" Standard Edition*.

For all our experiments we use instances of the Java class `Integer` (not to be confused with the Java integer primitive) as the elements that we send between the several computers. We use graphs for showing the various experimental results, where the x-axis represents the total number of `Integer` elements that are exchanged during the experiment, and the y-axis the communication time, i.e. the time that it takes to exchange these elements (from the point of view of a global clock).

7.6.1 RMI and MoCha

MoCha is implemented on top of *Java RMI* (see Section 7.3). Therefore, we created an experiment where we show the overhead of MoCha with respect to RMI. A producer and a consumer residing on two different computers send elements to each other starting from `10000` and up to `100000` incrementing the amount in each run by `1000` elements. We perform two series of tests, one using RMI and one using MoCha's *synchronous channel*. This is the most suitable channel type for the comparison since it, as RMI, provides synchronous communication.

Figure 7.20 shows the results of our experiment. As expected, RMI is faster due to the added coordination layer of the synchronous channel. We can see that the overhead for `10000`, or fewer, elements is smaller than for `100000` elements. For the more elements we send, the more overhead we get. However, the difference between the two lines is acceptable: proportionally, they more or less stay close to each other.

7.6.2 Comparing Channel Types

We compare the performance of MoCha's mobile channels with each other. We divide the experiment in two groups. The first group consists of the communication

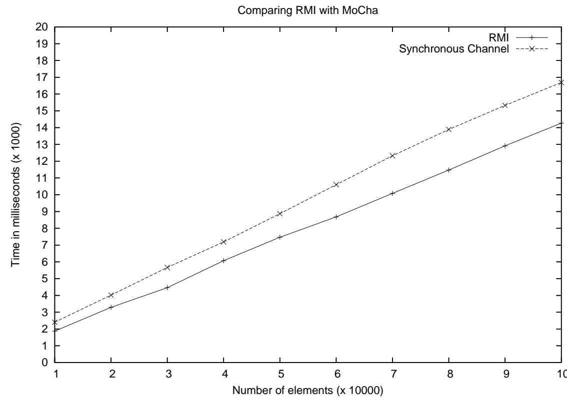


Figure 7.20: Comparing RMI and MoCha

and coordination channel types. The second group consists of the coordination only channel types.

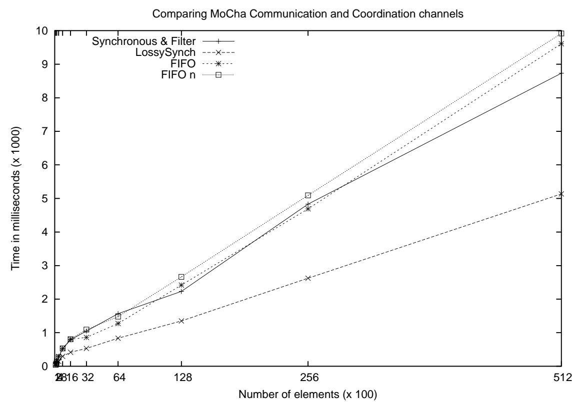


Figure 7.21: Communication and Coordination Channel Types (I)

For the first group, we set up a producer component on one computer, a consumer component on another one and put a channel in between the two. Initially, the producer and consumer exchange `100` elements. We double the number of elements with each test, until we finally arrive at `51200` elements. We do this, to see how well the MoCha channels scale with respect to the number of elements that they transmit. The channel types in the first group are: *synchronous*, *filter*, *lossy synchronous*, *FIFO (unbounded)*, and *FIFO n*. Figure 7.21 shows the result of our experiment. Since, the first part of the graph is difficult to read, we give a separate graph of this region in Figure 7.22 (`100 - 3200` elements).

We can see that the results from `10000` elements and up are quite stable and that the lines increment proportionally. For smaller number of elements, we are

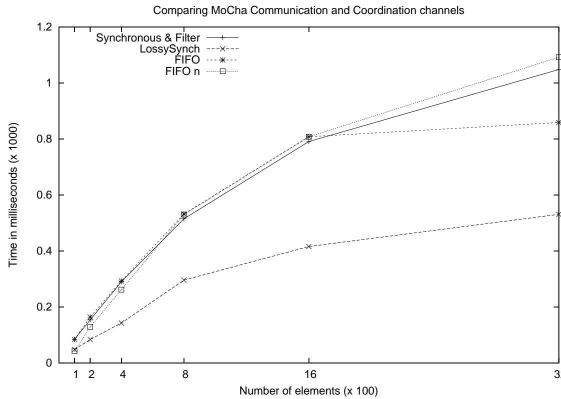


Figure 7.22: Communication and Coordination Channel Types (II)

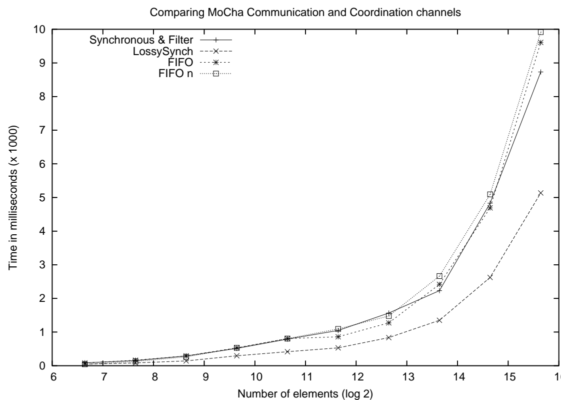


Figure 7.23: Communication and Coordination Channel Types (III)

dealing with Java virtual machine issues such as initialization procedures and initial internal optimizations that affect our results. For bigger numbers, we can ignore the influence of these issues. In this experiment we are not testing mobility, therefore, unbuffered communication is expected to be faster than buffered. This is the case in our graph, where the synchronous channels are faster than the asynchronous ones. The fastest channel type is the *lossy synchronous*. However, this is due to its lossiness property (see Section 2.4); elements get lost when the producer writes them while the consumer is not simultaneously taking them. Otherwise, with no elements getting lost, the performance of this channel type should be more or less equal to the one of the normal *synchronous* channel type. The slowest channel type is the *FIFO n* one. Although we give it a capacity higher than the number of elements that we send, this capacity check seems to slow down the communication with respect to the normal (unbounded) *FIFO* channel. In some cases, the asynchronous channels are

slightly faster than the synchronous ones. We expect this to be due to the internal heuristics as described in Section 7.5.2. However, the more elements we send, the less these heuristics seem to work; which makes sense since the heuristics are based on channel-end mobility, and without it there is not much to optimize after a certain number of elements. The *Filter* channel performs as well as a *synchronous* type one. This is no surprise, since checking whether data matches the channel's pattern is done locally and does not affect the communication at all (if all data elements match the pattern as is the case here). In Figure 7.23, we compute the logarithm ($\log_2$) of the number of elements to show that the MoCha channels scale quite nicely.

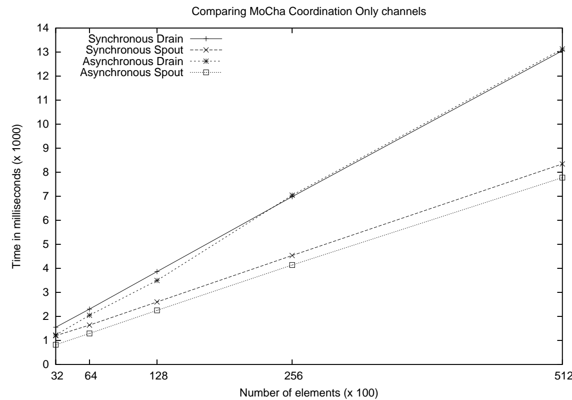


Figure 7.24: Coordination Only Channel Types

For the second group, we set up either two consumer or two producer components, depending whether the channel type has two source or two sink channel-ends. We then measure the time by taking the start time of the first starting component and the end time of the last ending component. With this kind of measuring procedure it is not possible to get accurate results for small numbers of elements. Therefore, we begin with `3200` and double this number each round until `51200`. The channel types in these experiment are: *synchronous drain*, *synchronous spout*, *asynchronous drain*, and *asynchronous spout*. Since the ends of the *drain* and *spout* channel types are independent of each other, we don't consider these types because we couldn't get any reliable results. Figure 7.24 shows the result of this experiment. To our surprise, the spout channels are faster than the drain ones. Considering that the internal synchronization structure between channel-ends for all types is more or less equivalent, we can only conclude that the channel *take* operation must be (somehow) faster than the *write*.

7.6.3 (Static) MoCha vs. LighTS

We compare the *peer-to-peer* architecture of MoCha against the *shared data spaces* architecture of the *LighTS* package (see Section 6.5.2). We add to LighTS an efficient RMI interface to make it a distributed application. Just like with the previous experiment, we use a producer and a consumer component that, in this case, either

use a MoCha channel or the distributed LighTS application for communication (and coordination). These components initially exchange `100` elements but double this number in each run until they reach `51200` elements. The communication between the components and the shared data space is asynchronous. Therefore, we compare LighTS with the *FIFO* channel type. To show that a peer-to-peer architecture is faster than a centralized one, we also tested a *synchronous* channel type as well.

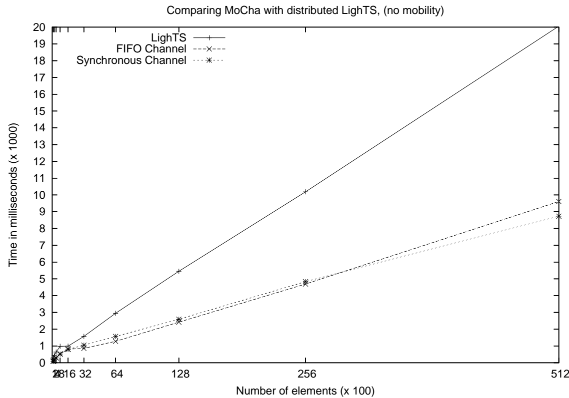


Figure 7.25: MoCha vs. LighTS (No Mobility) (I)

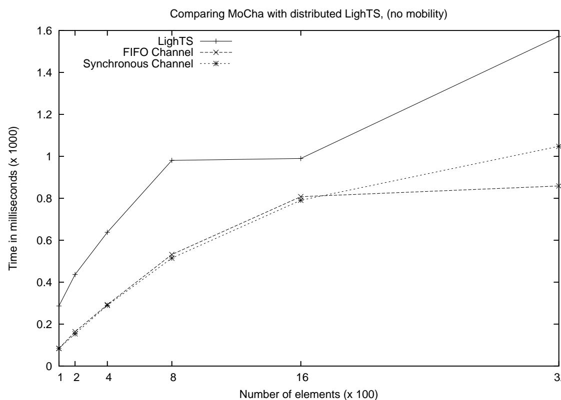


Figure 7.26: MoCha vs. LighTS (No Mobility) (II)

In Figure 7.25 we present the results of this experiment. In Figure 7.26 we enlarge the first region of the graph. As expected, the MoCha channels are much faster in all cases. Just like with the previous experiment, we see that the FIFO channel is more efficient than the synchronous ones in the beginning. However, beyond a certain number of elements, the heuristics don't work anymore and the synchronous channel becomes faster. In Figure 7.27, we compute the logarithm ($\log_2$) of the number of elements. Here we can clearly see that the more elements

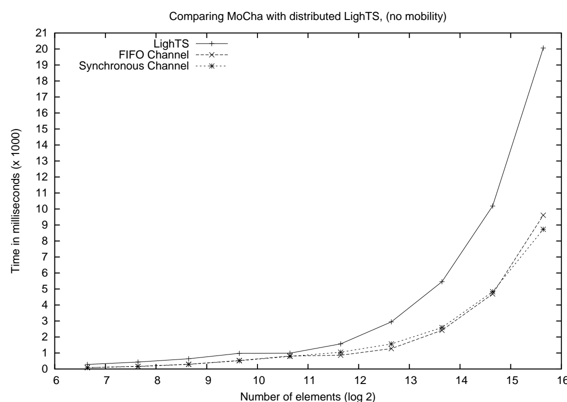


Figure 7.27: MoCha vs. LighTS (No Mobility) (III)

the components exchange, the bigger the difference between the two middleware systems becomes, and the faster this difference grows.

7.6.4 Movement of Channel-ends

In the previous experiments we did not consider (channel-end) mobility. In the mobility experiments we first measure the overhead of channel-end movement without considering data transfer or coordination. To measure the movement of channel-ends we created an experiment where we let a particular channel-end move `100` times between several computers. We then take the total amount of time and divide this by `100`. We tested all ends of each channel type, and calculated an average time for channel-end movement. We make a distinction between a source and a sink channel-end: the average time for source-end movement is `173.5` milliseconds (`0.1735` seconds), and the average time for sink-end movement is `173` milliseconds (`0.173` seconds). Thus, there is not much difference between how long it takes to move a source or a sink channel-end from one computer to another one.

7.6.5 Mobile Components (Moderated Mobility)

One of the major features of MoCha, besides the exogenous coordination of components, is that channel-end mobility allows dynamic reconfiguration of channel connections among the components in a (distributed) system. In Section 2.3, we show how useful this feature is in combination with mobile components. Therefore, we created an experiment where we use two mobile components: a producer and a consumer component. To make the movement of these components possible we developed a mobility framework based on Java RMI. This framework is independent of the MoCha middleware, no channels (nor any other MoCha entity) are used for the actual movement of components. Naturally, the producer and consumer are linked to each other by a mobile channel, and when they move to a new location (computer in this case) they take their corresponding channel-end with them. They do this by

performing a `move` operation on the end after they themselves moved to the new location first (to ensure independence).

We ran several tests using a *synchronous* and a *FIFO* channel type. In a previous experiment we compared (static) MoCha with LighTS. Therefore, we also perform tests using our distributed extension of this shared data space application (see Section 7.6.3). Naturally, we use the same mobile framework with LighTS for a fair comparison. We let the components initially exchange `20000` elements and double this number in each run until we get to `320000` elements. We let each component to move exactly `5` times from one computer to another (each time a different one). The number of `write` or `take` operations per location, is the total number of elements that have to be transmitted in a run divided by `5`. So, initially the producer or consumer writes or takes `5000` elements at each location and then moves to a new computer. In the last run (`320000` elements) the components write/take `64000` elements before moving to a new computer each time.

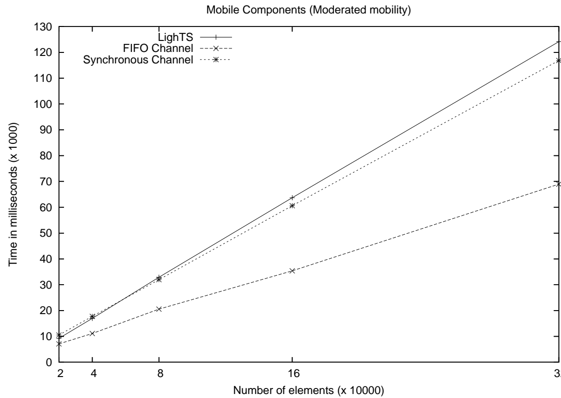


Figure 7.28: Mobile Components, Moderated Mobility (I)

In Figure 7.28, we present the results of this experiment. In Figure 7.29, we compute the logarithm ($\log_2$) of the x-axis to show how the middleware scales with respect to the number of elements. In contrast with the results of our internal experiment (see Section 7.6.2), the buffered communication of the FIFO channel is always much faster than the unbuffered one of the synchronous channel. To our surprise, there is not a substantial difference between the performance of LighTS and the synchronous channel. Moreover, in the beginning LighTS is faster. Afterward the synchronous channel slowly becomes faster. We expected the synchronous channel (due to its peer-to-peer architecture) to be much faster than LighTS. However, the communication in the computer cluster that we use goes so fast that only for big number of elements there is a significant difference. On the Internet, we expect the communication with the shared data space to be much slower and, thus, for LighTS to perform less well.

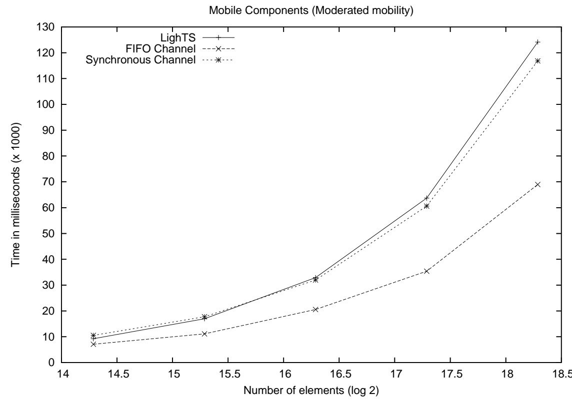


Figure 7.29: Mobile Components, Moderated Mobility (II)

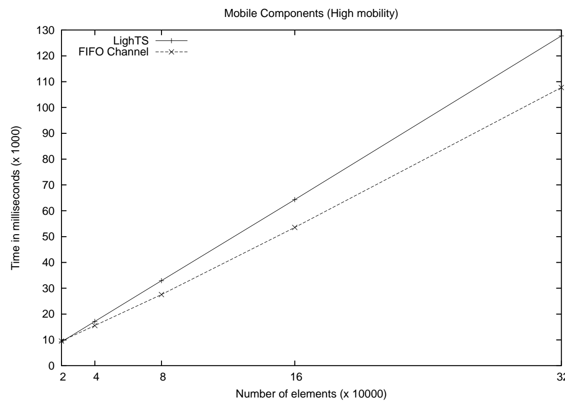


Figure 7.30: Mobile Components, High Mobility (I)

7.6.6 Mobile Components (High Mobility)

In the previous experiment, the producer and consumer components move modestly (5 times). We now repeat the same experiment where we considerably increase the mobility of these components. We also give the source and sink channel-ends a different moving frequency. We let the producer to write 4000 elements at each location before moving, and the consumer to take 1000 elements at each location. Thus, with 20000 elements the source-end moves 5 times and the sink-end 20 times. At the end, with 320000 elements, the source-end moves 80 times and the sink-end 320 times.

In Figures 7.30 and 7.31 we give the results of this experiment where we test the FIFO channel type and the LighTS shared data space. From our experiment in Section 7.6.4, we know that channel-end movement costs about 173 milliseconds. Thus, the more channel-ends we move the slower we expect our results to be. Indeed,

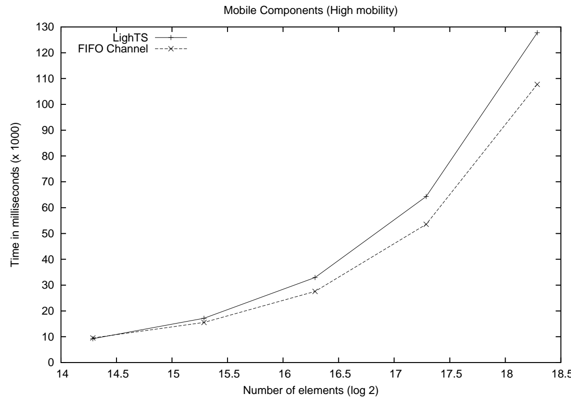


Figure 7.31: Mobile Components, High Mobility (II)

if we compare the results of the previous test (see Figure 7.28) with this one, we can see that the performance of the FIFO channel in this test is now much slower. The distributed LighTS is not that much affected by the increased component movement since it involves no channel-end movement in the first place. Nevertheless, our FIFO channel is still faster than LightTS. Only for small numbers of elements they perform equally well (see the `20000` elements point). Therefore, we can conclude that we have to be careful about the movement frequency. If we have a lot of channel-end movement but few elements to actually transfer then we better use a shared data space. In other cases (which are more common), MoCha performs better.

Chapter 8

An Implementation of the Channel-Based Component Model

In this chapter, we implement the coordination model for the component-based software that we introduced in Chapter 5. The Java implementation of this coordination model 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. At the same time it demonstrates that it is self-contained enough for developing component-based systems in object-oriented languages.

8.1 Integration of Components with Object-Oriented Technology

Regarding the integration of components with object-oriented technology, components adhere to the following object-oriented fundamental principles:

- system-wide 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 8.2, 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).

8.2 Implementation in Java

The coordination model we present in Chapter 5 can be implemented in any object-oriented programming language that supports distributed environments, like Java [Java], or C++ [Str91]. In this section we describe an implementation of our model in the Java language.

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

8.2.1 Components in Java

Usually, JavaBeans [JB] are used to implement components in Java. However, they do not comply with our definition of components (see Section 5.2.2) 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 5.2.3).

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 8.1). Therefore, we impose some restrictions that must be verified by our pre-compiler. 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 8.2.4), 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 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.

8.2.2 Implementation Overview

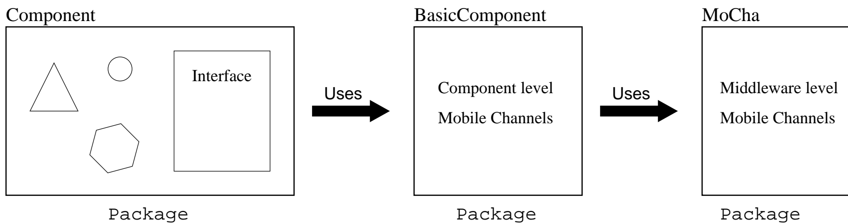


Figure 8.1: Implementation Overview

Figure 8.1 shows a general overview of the structure of our implementation. The component `package` uses, with the `import` feature of Java, our *BasicComponent* `package`. The BasicComponent `package` is an extra layer between the components and the mobile channels of MoCha. This layer is needed to translate the coordination operations of Section 8.2.4 to the middleware operations of Section 6.2. For example, within a component we don't need to specify a MoCha `ComponentKey` and pass it on as a parameter with every coordination operation. This is taken care of by the intermediate layer.

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. To accomplish this, the `package` provides methods that are `protected` and which only the coordination methods of the interface can use.

The BasicComponent `package` also provides a `public class CmpLocation` for components. This `class` 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. The usage of this `class` is analogous to the `MoChaLocation class` of Section 6.2.1.

8.2.3 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 8.2 shows the skeleton of a `.cmp` file for the interface. There is some syntactic sugar in this file that the *pre-compiler* 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 8.3). The pre-compiler 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 8.2: The `.cmp` Skeleton File for the Interface of a Component

The `public` part of the interface consists of three parts (see Figures 8.2 and 8.3): one or more constructors, a `getLocation` method, and a `finalize` method. The pre-compiler checks that 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 8.3) 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 8.1. 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 Chapter 5.

The `package private` part of the interface includes the coordination methods provided by the class `BasicInterface` (see Figure 8.3), channel-end variables, and all the other methods and variables in the interface that are not `public`. We explain the coordination methods in Section 8.2.4.

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

class BasicInterface
{
   BasicInterface(CmpLocation loc)
   public CmpLocation 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 8.3: 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.

8.2.4 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 coordination methods are equal to (a subset of) the methods that the exogenous coordination language *Reo* [Arb02] provides to components. The choice for defining the same methods allows the possibility to easily extend our component model to work with Reo instead of the MoCha middleware (for more complex and non-trivial exogenous coordination of components than with "just" the simple MoCha channels).

The coordination methods are listed in Figure 8.3. The threads cannot perform any operation directly on 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.

In Chapter 6 we mentioned the fact that the coordination operations have time-outs, but we chose not to consider them for simplicity reasons. In this chapter, we explicitly consider time-outs for they are an important concept in our component model. For every method containing a `timeout` parameter in Figure 8.3, 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` (if we create a ⟨*source*, *sink*⟩ channel type, otherwise we get either two source or two sink ends).

`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.
`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 they are 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.

8.2.5 A Small Example

We use a simple implementation of the mobile agent component of the example in Section 2.3, to show the utility of the coordination operations provided by our model. Figure 8.4 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.

8.3 Related Work and Conclusions

In this and in Chapter 5, we presented a coordination model for component-based software based on mobile channels. The motivation of these chapters comes from these works of *Arbab et al.*: [ABB00a] and [ABB00b]. The idea of using mobile channels for components comes from Reo [Arb04, Arb02].

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

```
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 8.4: Simple Implementation of The Mobile Agent

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 [ALSN01] 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* [SGW94] and *Darwin* [MDEK94], two architectural description languages (ADL), are related to our work. In ROOM com-

ponents 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 chapter, 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.

Finally, although it is not the main purpose of our work, the Java implementation presented in Section 8.2 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 our notion of components is possible in object oriented paradigms such as UML.

Part IV

Composition

Chapter 9

Composition of Mobile Channels

In this chapter, we present a model for composition of mobile channels based on the notion of *coordination components*. These are lightweight components that are meant for linking channels together according to some transparent coordination behavior. We introduce a basic set of such components, and give examples of compositions. For each component we give semantics by providing a MoCha-π specification and a Petri Net. We also discuss how to obtain the semantics of the compositions as well.

9.1 Introduction

Up to now, in this thesis, we exogenously coordinate components by using single mobile channels in between them. In this chapter we take the next logical step by composing mobile channels. Our motivation for doing the composition is that it brings two immediate advantages: the easy specification of more (complex) exogenous coordination than just the ones specified by a pre-defined set of mobile channels, and, the possibility of coordinating more than two components at the same time. The idea of composing mobile channels is not new and originally comes from *Reo* [Arb02, Arb04] (see Section 9.2).

Each of the mobile channel types we introduced in this thesis (see Section 2.4) provides a different exogenous coordination pattern for the components using it. Naturally, we hope that this particular set of mobile channels is enough to cover all exogenous coordination possibilities. However, this is far from the truth. Therefore, we need to introduce a new channel type for each new exogenous coordination pattern that is not already provided by an existing mobile channel type. The problem with this is that introducing a new channel type each time is a lot of work. We have to describe its behavior (see Chapter 2), specify it using our *MoCha-π* calculus (see Chapter 4), (alternatively) construct a *Petri Net* for it (see Chapter 3), and implement it in the *MoCha middleware* (see Chapters 6 and 7). Thankfully, with channel composition we have a much easier alternative. By regarding the existing mobile channel types provided by this thesis as basic blocks, we can build other (more com-

plex) types by "simply" composing the basic ones. Naturally, composed types can be composed again into a new type, and so on. This means that, both the *MoCha-π* and the *Petri Nets* specification of the composed types is just the composition of the ones of its constituents (as we shall see). The *MoCha middleware* implementation is realized by linking all the channels together using coordination components (see Section 9.4).

Single mobile channels coordinate only two components at the same time. However, in some situations it is desirable to relate and coordinate three or more components. The Reo model defines the useful notion of a *connector*. A connector is an exogenous coordination infrastructure that provides channel-ends to components. It can have many channel-ends that are of type *source* or *sink*. A single mobile channel is regarded to be a connector as well. A connector, whose internals is the composition of mobile channels (or/with other connectors), that consists of exactly two ends is regarded to be a mobile channel as well. However, connectors with either one or more than two channel-ends *are not* considered to be mobile channels. Connectors are able to coordinate more than two components at a time.

In Section 9.2, we discuss the already existing model for mobile channel composition Reo. In Section 9.4, we introduce a composition model that takes some of the ideas of Reo and realizes them by composing mobile channels using the notion of *coordination components*. We shall see that our model offers connectors at a lower level of abstraction that brings them closer to actual implementations in distributed systems. Moreover, our model easily implements a subset of Reo. In Section 9.5, we give an example of a component based system where the component instances are coordinated by connectors. This is also an example of a Reo application that we can we implement using our model. In Section 9.6, we end with a discussion and a comparison between the two models.

9.2 Reo

In this section we discuss *Reo*, a model for mobile channel composition created (and invented) by *Arbab* [Arb02, Arb04]. Reo is a channel-based exogenous coordination model wherein complex coordinators, *connectors*, are compositionally built out of simpler ones. The simplest connectors are mobile channels. The properties of these channels are equal to the ones of this thesis (see Section 2.2.2 for the properties).

In Reo, a connector is a set of channel-ends organized in a graph of *nodes* and edges such that:

- Zero or more channel-ends coincide on every node.
- Every channel-end coincides on exactly one node.
- There is an edge between two (not necessarily distinct) nodes if there is a channel of which one end coincides on each of those nodes.

A node is an important concept in Reo. Not to be confused with a location or a component, a node is a logical construct representing the fundamental topological property of coincidence of a set of channel ends, which has specific implications on the flow of data among and through those channel-ends.

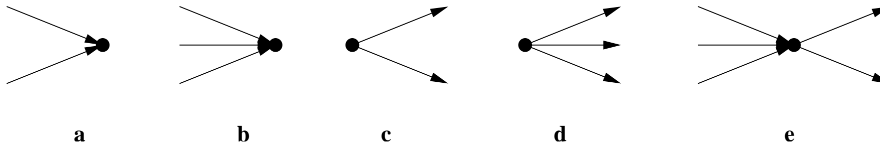


Figure 9.1: Nodes in Reo

The set of channel-ends coincident on a node A is disjointly partitioned into the sets $\mathsf{Src}(A)$ and $\mathsf{Snk}(A)$, denoting the sets of source and sink channel-ends that coincide on A, respectively. A node A is called a *source node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) = \emptyset$ (the node consists of only source-ends). Analogously, A is called a *sink node* if $\mathsf{Src}(A) = \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$ (the node consists of only sink-ends). A node A is called a *mixed node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$ (the node consists of both source- and sink-ends). Figures 9.1(a) and 9.1(b) show sink nodes with, respectively, two and three coincident channel-ends. Figures 9.1(c) and 9.1(d) show source nodes with, respectively, two and three coincident channel-ends. Figure 9.1(e) shows a mixed node where three sink and two source channel-ends coincide.

Reo enables components to connect to and perform I/O operations on source and sink nodes only; components cannot connect to, read from, or write to mixed nodes. At most one component can be connected to a particular (source or sink) node at a time.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel-ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a especial kind of *replicator*. Especial, because it needs to know whether all source-ends can accept the data before writing to them.

A component can obtain data items from a sink node that it is connected to through destructive (take) and non-destructive (read) input operations. A take operation succeeds only if at least one of the (sink) channel-ends coincident on the node offers a suitable data item; if more than one coincident channel-end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*.

A mixed node is a self-contained "pumping station" that combines the behavior of a sink node (merger) and a source node (replicator) in an atomic iteration of an endless loop: in every iteration a mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel-ends and replicates it into all of its coincident source channel-ends. A data item is suitable for selection in an iteration only if it can be accepted by all source channel-ends that coincide on the mixed node.

9.2.1 Composition of Connectors

Every mobile channel represents a (simple) connector with two nodes. More complex connectors are constructed in Reo out of simpler ones using its *join* operation. Joining two nodes destroys both nodes and produces a new node on which all of

their coincident channel-ends coincide. Reo also provides a *split* operation for decomposition of connectors. Splitting a node produces a new node and divides the set of channel-ends that coincide with the first node between the two. Both operations are performed by "external" components on Reo nodes and make it possible to dynamically reconfigure connectors. More extensive details about these operations are given in [Arb02].

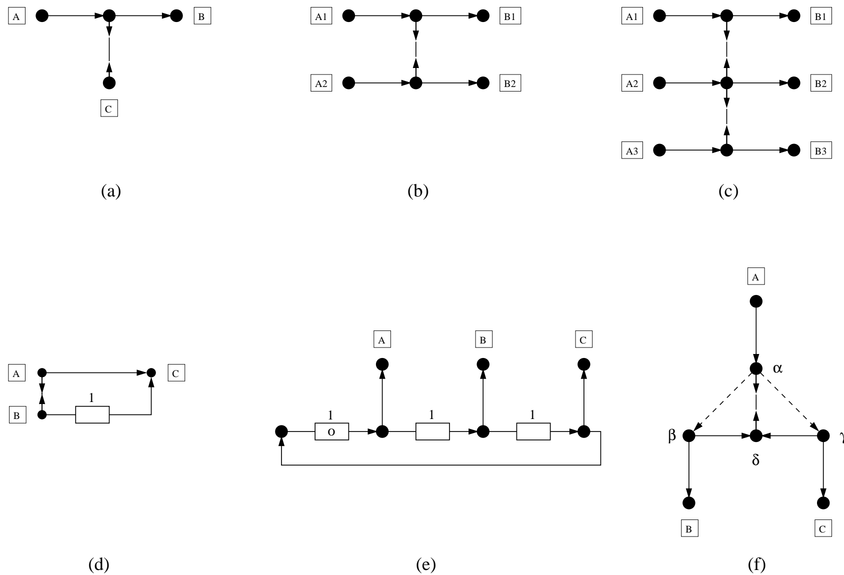


Figure 9.2: Reo Connectors

We give some examples of frequently used connectors in the Reo papers, where we add external taking and writing components for an easier explanation of their behavior. The *write-cue regulator* (see Figure 9.2(a)) is composed out of two synchronous and one synchronous drain channel. The mixed node of this particular connector can take a value from component A only when B is ready to take a value from the connector and C is ready to write one into it. Therefore, every write action of component A has to simultaneously coincide with a take action of B and a write action of C. In other words, with this connector component C regulates the flow of data between A and B.

The *barrier synchronizer* (see Figure 9.2(b)) is a simple extension of the previous connector. The synchronous drain channel ensures that a value passes from components A_1 to B_1 simultaneously with the passing of a value between A_2 and B_2. Therefore, all writes and takes on this connector are synchronized with each other. The barrier synchronizer can easily be extended to any number of component pairs A and B as shown in Figure 9.2(c).

The *ordering* connector (see Figure 9.2(d)) is composed out of one synchronous, one synchronous drain, and one FIFO 1 channel. The synchronous channel ensures that a write by component A simultaneously succeeds with a take by component C. The synchronous drain ensures that the write actions of components A and B are

synchronized. Therefore, the first value that component C gets is always written by A. The value of component B goes into the FIFO 1 channel. The next value that C gets is the one written by B, due to the fact that the FIFO 1 channel must be empty before components A and B can (synchronously) write again. The third value comes from A again, the fourth from B, and so on.

The *sequencer* (see Figure 9.2(e)), consists of FIFO 1 channels in series and an equal plus one number of synchronous channels (in the figure there are three FIFO 1 channels). The left most FIFO 1 channel is initialized with a “dummy” value in its buffer, as indicated by the presence of the symbol “o”. This connector ensures that the take operations of the components A, B, and C succeed only in the strict left to right order. The sequencer is easily extensible to any number of taking components, by just adding for each one an extra FIFO 1 and synchronous channel to the connector.

The *exclusive router* (see Figure 9.2(d)) is composed out of five synchronous channels, two lossy synchronous channels, and a synchronous drain. The behavior of this connector is that the values that are written by component A are taken by either components B or C. If both components want to simultaneously take a value, one of the two is non-deterministically selected. The internals of this simple connector are non-trivial. To simplify our explanation we marked some of the nodes in Figure 9.2(d) with a Greek letter. In the case that component A writes a value and only component B wants to take it, the written value of A flows through path (α, β) and gets lost through path (α, γ). If only component C wants to take the value written by A, then the value flows though path (α, γ) and gets lost through path (α, β). If both components simultaneously want to take the value, node δ acts as a merger and nondeterministically selects which path is lossy and which path is going to let the value through. Observe that, the writes of component A are synchronized with the takes of either component B or C; i.e. component A cannot write when there is no component to take.

9.2.2 More about Reo

The reader that wants to know more about Reo is kindly referred to these works of Arbab [Arb02, Arb04, Arb06]. More complex examples of Reo connectors in the context of e-commerce are given in [DZP05]. Examples of Reo connectors modeling biological processes are given in [CCA04].

There are also several kinds of semantics available for Reo. In [AR03], co-inductive calculus semantics are given for connectors and their composition. In [ABBR04], semantics for Reo are given using temporal logics. In [ABRS4], the semantics of Reo are defined using constraint automata. For a good understanding of the Reo connector semantics we refer to the work of Clarke and Costa about connector coloring [CCA05].

9.3 Issues Concerning the Distributed Implementation of Reo Connectors

In this section we discuss some of the main issues regarding the implementation of Reo connectors in distributed systems. Addressing these issues is important for understanding the differences between Reo and the MoCha model we will present in Section 9.4.

Reo uses the concept of a *node*, which is not a component but a logical construct. Therefore, the first step towards the implementation of a Reo connector is to identify what are its components. A trivial step is to implement each node as a component. However, from the point of view of efficiency, we may identify sub-parts of a connector as a component as well. For example, take a look at the sequencer connector of Figure 9.2(c). If we posses the knowledge that this particular connector is going to remain static during the lifetime of the system, it is a good idea to implement the series of FIFO 1 channels (and their corresponding nodes) as one component.

Another implementation issue involves the dynamic reconfiguration of connectors. Reo uses the *join* and *split* operations for this purpose. However, these are logical operations that work on graphs. Reo does not specify how these operations are actually (to be) implemented in distributed systems. Therefore, the components that we use (which we identified above) must implement some protocol to allow dynamic reconfiguration.

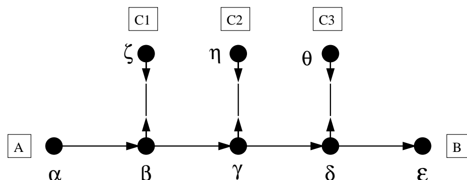


Figure 9.3: Synchronization of Channel-end Operations

The most difficult implementation issue involves the propagation of synchrony. With Reo we are able to create connectors where we can achieve atomic synchronization of channel-end operations that involve more than one Reo node; for example, the barrier synchronizer connector of Figure 9.2(c) where all (internal and external) write and take operations are synchronized. However, the nodes of these kind of connectors need to have *global* state connector information to make certain decisions. We explain this by looking at the Reo connector given in Figure 9.3. This connector consists of four synchronous channels and three synchronous drain channels. The behavior of this connector is that component B successfully takes one value written by component A each time that all components C write one "dummy" value to the connector. Furthermore, all the write and take operations are synchronized; i.e. atomically succeed. For this to be possible each node must know the state of the entire connector (or parts of it) and its boundary. For example, node ϵ needs to know if component B is ready to take a value before it is able to accept one itself. Node δ needs to know if node ϵ accepts a value and if node θ has one to offer before it can accept a value to replicate. Node γ needs to know if node δ accepts a value

and if node η has one to offer. And so on, until we get to node α. For this node to allow component A to successfully write to it, it must first (direct or indirectly) check the status of all the other nodes. A process that must be repeated for each value that flows from A to B.

The implementation of such a Reo connector is not an easy task. A truly distributed implementation prohibits the presence of a central point that contains the current global connector state for the components (which implement the Reo nodes) to query. Instead, all the information that is relevant for the local synchronization of nodes must be propagated through the different synchronous sub-sections of the connector. The challenge lies in finding an efficient implementation. For example, we can propagate the global state to each component for every channel-end operation that is performed on the connector. However, such a protocol is highly inefficient for it generates far more control messages than the amount of data that actually flows through the connector.

The work in [CCA05] discusses the requirements that a distributed implementation of Reo must fulfill. It also provides the first steps towards a non-trivial algorithm for implementing the Reo propagation of synchrony. This algorithm is still in an early phase to be able to determine whether its actual implementation will be efficient.

9.4 MoCha's Coordination Components Model

Inspired by Reo, we present a composition model that provides a lower level of abstraction which is closer to the actual implementation of connectors in distributed systems. We combine the results of the previous chapters of this thesis to create this model, where we take and implement some of the main ideas of Reo while we avoid or modify others. Thus, we can say that our model (easily) implements a subset of Reo. We proceed by presenting the main features of our model while briefly indicating the similarities and differences with Reo. In Section 9.6 we discuss the differences between the two models in more detail.

As with Reo, the major requirement we impose on our model is *compositionality*: one should be able to distinguish the individual components and channels in the composed system, and it must be easy to decompose and rearrange the system; i.e. to update and replace specific components and channels without having to change the rest of the system.

The aim of our model is to be able to easily implement its connector specifications in distributed systems. Therefore, instead of composing channels using Reo nodes we use a special kind of components, called *coordination components*, for this purpose. These are lightweight components that are meant for linking channels (or connectors) together according to some transparent coordination behavior. This behavior can be much more complex than the one of the Reo nodes. The coordination components follow the specification given in Chapter 8, where we state that a component's interface consists of a dynamic set of channel-ends. Figures 9.4(a) and 9.4(b) give examples of such a component. For convenience, we draw the required channel-ends on our components as ports. An input port is a required sink channel-end where the component takes values from. An output port is a required source channel-end, where the component writes values to.

Alike Reo, the connectors of our model are dynamically reconfigurable. For this purpose, Reo uses the logical operations *split* and *join*. We implement these operations by providing our coordination components with a protocol for changing their interface's set of channel-ends (or ports) dynamically. On top of this, the mobile channels that we use already provide dynamic reconfiguration of channel connections among the components in a system (see Chapter 4 and Chapter 6). Therefore, we are able to change the topology of our connectors dynamically; both the channel-end connections and the amount of incoming and outgoing channels of the coordination components can dynamically change during the runtime of a system.

In Section 9.3 we discussed the difficulties of implementing an efficient algorithm for the Reo propagation of synchrony. The nodes involved in this synchrony need to have *global* state connector information to make certain decisions. In contrast, in our MoCha model the coordination components make decisions strictly based on *local* state information. This makes it easier to implement the MoCha connectors in distributed systems.

Next, we introduce a basic set of coordination components: the *replicator*, the *non-deterministic multiplexer*, the *ordered multiplexer*, the *non-deterministic demultiplexer*, the *ordered demultiplexer*, the *write gate transistor*, the *take gate transistor*, the *write switch*, and the *take switch*. The choice for this particular set is based on examination of Reo examples in the literature. Moreover, the behavior of some of these components resemble the ones of existing basic Reo connectors. We shall indicate which ones, and make comparisons. For simplicity, for each coordination component we first introduce a static version, where the (amount of) channel-ends of its interface is fixed and given at creation time. Afterward, we give the generic dynamic version, where the (amount of) channel-ends of its interface change(s) during the runtime of the system. For both versions, we give the MoCha-π specification (see Chapter 4). For each of the static coordination component versions, we give a Petri Net specification (see Chapter 3). For each coordination component we give examples of simple connectors. We also show how to do the composition of the MoCha-π and Petri Nets semantics. All the components have been implemented and added to the MoCha middleware (see Chapter 6).

After the introduction of these coordination components, we give some further examples of useful connectors: the *sequencer*, the *ordered drain*, the *multi write gate transistor*, and the *n-to-n write switch*. Finally, we discuss the composition of connectors themselves.

9.4.1 Replicator

The *replicator* has one input channel and many output channels (see Figure 9.4(a)). This coordination component takes a value from the sink-end of the input channel and replicates it by writing the same value to all the source-ends of the output channels (in the figure the channel-ends are represented as ports). The replicator's behavior resembles to the one of the Reo source node (see Figures 9.1(c) and 9.1(d)), the difference is that the replicator does not require to know whether all output source-ends can accept the data before writing to them. Our replicator just writes the data to all source-ends and waits until all these operations are completed. We give its MoCha-π specification:

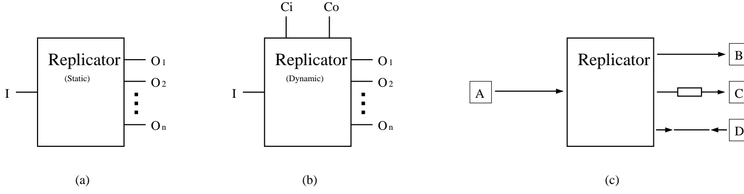


Figure 9.4: Replicator

$$Rep(i_{Sink}, \vec{o}) \stackrel{def}{=} i_{Sink} \downarrow .(o \downarrow .)^{\{\forall o \in \vec{o}\}}; Rep'(i_{Sink}, \vec{o})$$
$$Rep'(i_{Sink}, \vec{o}) \stackrel{def}{=} i_{Sink}?(data).(\ (o!\langle data \rangle \mid)^{\{\forall o \in \vec{o}\}}\ ); Rep'(i_{Sink}, \vec{o})$$

The replicator process *Rep* receives at creation the sink-end i_{Sink} of the incoming channel and a vector $\vec{o}$ with the source-ends of all the output channels. At initialization, the process connects to all channel-ends and it's then ready to receive a value. The actual replication is given by process Rep'. This process takes a value from the given sink-end and writes it in parallel to all the source-ends of vector $\vec{o}$. After all writes are completed, and not earlier, the replication process repeats itself.

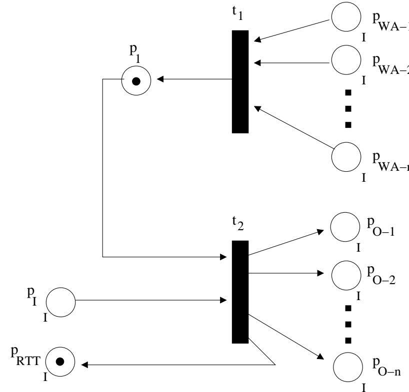


Figure 9.5: Replicator PN

In Figure 9.5 we give a Petri Net, from now on PN, that specifies the concurrent behavior of our replicator coordination component. This PN follows the same protocol we specified for components as explained in Section 3.3.2. The replicator PN has one input side (or taking side) that consists of the interface places $\{p_I, p_{RTT}\}$, and n output sides (or writing sides) that consists of the interface places $\{p_{O-k}, p_{WA-k}\}$ where $1 \leq k \leq n$. The initial configuration of our PN is $\{p_{RTT}, p_1\}$. Place p_{RTT} initially contains a token to indicate that the replicator component is ready to take a value. From the initial configuration nothing can happen. However, when composed with mobile channels, at some point in time the token of place p_{RTT} enters

the sink-end of a mobile channel. After which, sometime later, a token is placed in p_I to indicate that a value is available from this end. The replicator takes this value and writes it to all output places p_{O-k}. We give the sequential firing step: $\{p_I, p_1\}[t_2\rangle\{p_{RTT}, p_{O-1}, ..., p_{O-n}\}$. The source channel-ends take the token of their corresponding output place, and sometime later acknowledge their write by inserting a token in the corresponding p_{WA} place. When all writes are acknowledge, transition t_1 can fire and place p_1 is filled with a token again. We are then back at the initial configuration from where the next replication can take place. Observe, that we need not to always end in the initial configuration; before all writes are acknowledge, the sink-end may already have taken the token from place p_{RTT}. However, no next take (by the replicator) can happen until place p_1 contains a token again.

The generic version of the replicator, where the static version is a particular instance of, is the dynamic replicator. With this version the (number of) channel-ends change(s) dynamically. For this purpose, we add two extra incoming mobile channels to the static version (as illustrated in Figure 9.4(b)) and specify a protocol for adding and removing channel-ends. Through the channel C_i, the replicator receives sink channel-ends to take the input values from. If the given sink-end is distinct from the current input sink-end, the later is replaced by the new one. Otherwise, if the given sink-end is equal to the current one, the later is removed and the replicator ends with no sink-end to take from. The same protocol holds for adding and removing output source channel-ends that the replicator receives through channel C_o. We give the MoCha-π specification of this dynamic replicator (where we use if-statements to make the specification shorter):

$$
\begin{aligned}
&Rep(ci_{Sink}, co_{Sink}) \stackrel{def}{=} ci_{Sink}\downarrow.co_{Sink}\downarrow.Rep'(ci_{Sink}, co_{Sink}, \langle\rangle)\\
&Rep'(ci_{Sink}, co_{Sink}, \vec{o}) \stackrel{def}{=}\\
&\quad (ci_{Sink}?(si).si\downarrow.Rep''(ci_{Sink}, co_{Sink}, si, \vec{o}))\\
&\quad + (co_{Sink}?(so)\\
&\qquad .((so\downarrow.Rep'(ci_{Sink}, co_{Sink}, \langle o_1, ..., o_{|\vec{o}|}, so\rangle))\{if\ \ so \notin \vec{o}\}\\
&\qquad + (so\uparrow.Rep'(ci_{Sink}, co_{Sink}, \vec{o}\setminus so))\{if\ \ so \in \vec{o}\}))\\
&Rep''(ci_{Sink}, co_{Sink}, i_{Sink}, \vec{o}) \stackrel{def}{=}\\
&\quad (ci_{Sink}?(si)\\
&\qquad .((i_{Sink}\uparrow.si\downarrow.Rep''(ci_{Sink}, co_{Sink}, si, \vec{o}))\{if\ \ si \neq i_{Sink}\}\\
&\qquad + (i_{Sink}\uparrow.Rep'(ci_{Sink}, co_{Sink}, \vec{o})\ )\{if\ \ si = i_{Sink}\}))\\
&\quad + (co_{Sink}?(so)\\
&\qquad .((so\downarrow.Rep'(ci_{Sink}, co_{Sink}, i_{Sink}, \langle o_1, ..., o_{|\vec{o}|}, so\rangle))\{if\ \ so \notin \vec{o}\}\\
&\qquad + (so\uparrow.Rep'(ci_{Sink}, co_{Sink}, i_{Sink}, \vec{o}\setminus so))\{if\ \ so \in \vec{o}\}))\\
&\quad + (i_{Sink}?(data).((o!\langle data\rangle\ |\ )^{\{\forall o\in\vec{o}\}})\\
&\qquad ;Rep''(ci_{Sink}, co_{Sink}, i_{Sink}, \vec{o}))\{if\ |\vec{o}| \geq 1\}
\end{aligned}
$$

At creation, the replicator process *Rep* receives the sink-ends of the C_i and C_o channels and connects to them. After this, it calls process *Rep'*. Within this process either an output source-end is added or removed from/to the vector $\vec{o}$, or an input sink-end is received and process *Rep''* is called. Within this last process, there are three possibilities: (1) An input sink-end is replaced or removed. Or (2), an output source-end is added or removed. Or (3), a value is taken from the input sink-end and written in parallel to all output source-ends (if there are any source-ends available).

In Figure 9.4(c), we give an example connector that consists of one replicator, two synchronous channels, one FIFO and one synchronous drain channel. We give the MoCha-π specification of this connector using the static version of the replicator (we give an example of a connector using a dynamic coordination component upon the introduction of the next component in Section 9.4.2):

$$
\begin{aligned}
&Connector(so_a, si_b, si_c, so_d) \stackrel{def}{=} new(si_a, so_b, so_c, so_{d2})\\
&\quad (SYNCHRONOUS(so_a, si_a) \mid SYNCHRONOUS(so_b, si_b) \mid FIFO(so_c, si_c)\\
&\quad \mid SYNCHDRAIN(so_d, so_{d2} \mid Rep(si_a, \langle so_b, so_c, so_{d2}\rangle))
\end{aligned}
$$

We can see that the composition of the MoCha-π semantics of the replicator and each used channel type is just the process algebra parallel composition. For simplicity, we used the static version of the replicator. Naturally, for the dynamic version it is not a good idea to encapsulate the connector in a process for its configuration may change.

To construct the PN connector, we use the composition function σ that mergers, or concatenates, *interface places* (see Section 3.3.3). We take the replicator, the channels PN and use σ to obtain the connector. To compose the FIFO channel with the replicator we do: $Result = \sigma(Replicator, \{p_{O-3}, p_{WA-3}\}, FIFO, \{p_{Source}, p_{WA}\})$. To add the synchronous drain: $Result2 = \sigma(Result, \{p_{O-2}, p_{WA-2}\}, SynchDrain,$ $\{p_{Source2}, p_{WA2}\})$. To add the outgoing synchronous channel: $Result3 = \sigma(Result2,$ $\{p_{O-1}, p_{WA-1}\}, Synchronous, \{p_{Source}, p_{WA}\})$. Finally, we obtain the connector by adding the incoming synchronous channel: $Connector = \sigma(Result3, \{p_I, p_{RTT}\}, Synchronous,$ $\{p_{Sink}, p_{RTT}\})$.

Two writing components (A and D) and two taking components (B and C) are using our example connector. Every time component A writes a value to the connector, components B and C can take one from it and component D is allowed to write a value (this value gets lost). Before being able to write a next value, component A is forced to wait until component B takes the replicated value from the connector and component D writes one "dummy" value to it first. There is no need to wait for component C, since due to the *FIFO* mobile channel, it can take the values out of the connector at its own convenience. Observe, that synchronization is between the replicator and components A, B, and D. There is no "third party" synchronization between, for example, components A and B.

For completeness, we end the description of the replicator by adding the components to the MoCha-π specification as write and take processes:

$$
\begin{aligned}
&Writer(source) \stackrel{def}{=} source \downarrow .Writer'(source)\\
&Writer'(source) \stackrel{def}{=} new\ data\ (source!\langle data\rangle); Writer'(source)\\
&Taker(sink) \stackrel{def}{=} sink \downarrow .Taker'(sink)\\
&Taker'(sink) \stackrel{def}{=} sink?(data).Taker'(sink)\\
&System \stackrel{def}{=} new(so_a, si_b, si_c, so_d)\\
&\quad (Writer(so_a) \mid Writer(so_d) \mid Taker(si_b) \mid Taker(si_c)\\
&\quad \mid Connector(so_a, si_b, si_c, so_d))
\end{aligned}
$$

9.4.2 Non-Deterministic Multiplexer

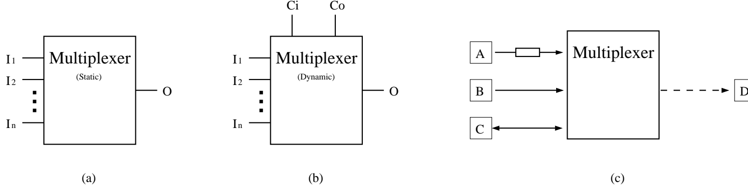


Figure 9.6: Multiplexer

The *non-deterministic multiplexer* has many input channels and one output channel (see Figure 9.6(a)). This coordination component non-deterministically takes a value from one of the sink-ends of the input channels that have a value to offer, and writes it to the source-end of the output channel. The behavior of our multiplexer is equal to the one of the Reo sink node (see Figures 9.1(a) and 9.1(b)). We give the MoCha-π specification of our multiplexer:

$$
\begin{aligned}
Mux(\vec{i}, o_{Source}) &\stackrel{def}{=} (i \downarrow .)^{\{\forall i \in \vec{i}\}}; o_{Source} \downarrow .Mux'(\vec{i}, o_{Source}) \\
Mux'(\vec{i}, o_{Source}) &\stackrel{def}{=} (i?(data)\ +)^{\{\forall i \in \vec{i}\}}; o_{Source}!\langle data \rangle .Mux'(\vec{i}, o_{Source})
\end{aligned}
$$

The multiplexer process receives at creation a vector $\vec{i}$ with the sink-ends of all the input channels, and the source-end o_{Source} of the output channel. At initialization, the process connects to all channel-ends. The multiplexing is done by process Mux'. This process non-deterministically chooses a sink-end and takes a value from it. Afterward, it writes this value to the source-end. When this last write action is completed, the process repeats the same pattern.

The MoCha-π specification is actually more general than the description of the non-deterministic multiplexer. The *Mux* process chooses any sink-end that *commits* to a take operation. However, it is not guaranteed that the channel of the chosen sink has (or will have) a value to give through this end. This leads to the deadlock of our multiplexer. To avoid this, we made all the MoCha-π mobile channels of Section 4.4.1 "Reo compliant"[1]; i.e. the sink-end of such a channel commits to a take operation only if it currently has a value to offer.

In Figure 9.7 we give the non-deterministic multiplexer PN. This PN has n input sides (or taking sides) that consists of the interface places $\{p_{I-k}, p_{RTT-k}\}$ where $1 \leq k \leq n$, and one output side (or writing side) that consists of the interface places $\{p_O, p_{WA}\}$. The initial configuration is $\{p_{RTT-1}, ..., p_{RTT-n}, p_1\}$. When composed with mobile channels, each sink-end of the input channels can take the token of place $\{p_{RTT-k}\}$ and put one back in the corresponding place $\{p_{I-k}\}$ to indicate that a value is available from this end. However, the multiplexer takes only one token

[1]An exception is the not so frequently used synchronous spout channel type, where only one of the two ends is able to commit when a value is available. Therefore, connecting two ends of the same synchronous spout channel instance to two multiplexer instances may lead to a deadlock situation.

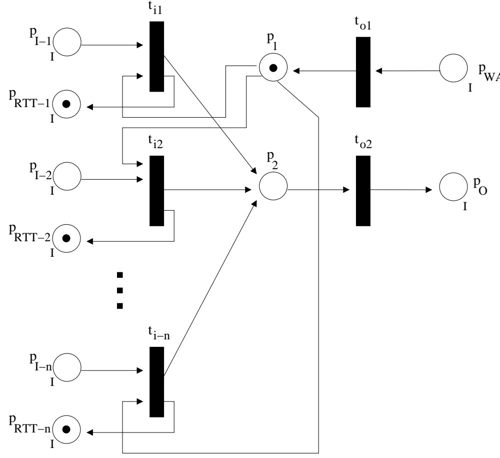


Figure 9.7: Non-Deterministic Multiplexer PN

(value) since the capacity of place p_2 is one and the token of place p_1 can only go to one input transition. For example, if the value of the first sink-end is selected the sequential firing step is $\{p_{I-1}, p_1\}[t_{i1}\rangle\{p_2, p_{RTT-1}\}$. After this, the value is written to the output place: $\{p_2, p_{RTT-1}\}[t_{o2}\rangle\{p_O, p_{RTT-1}\}$. The source-end of the output channel takes the token of the output place, and sometime later acknowledges the write by inserting a token in the p_{WA} place. Then, transition t_{o1} can fire and place p_1 is filled with a token again. We are then back at the starting configuration of our example, from where the next multiplexing can take place. Observe, that the function of place p_1 is to prevent the multiplexer from taking a new value before the write has properly succeeded.

For the dynamic non-deterministic multiplexer, we add two extra incoming mobile channels to the static version (as illustrated in Figure 9.6(b)) and specify a protocol for adding and removing channel-ends that is analogous to the one of the replicator component. Through the channel C_i the multiplexer receives sink channel-ends to take the input values from. Through the channel C_o the multiplexer receives source channel-ends to write the values to (only to one source-end at a time). We give the MoCha-π specification of the dynamic multiplexer, where we use if-statements to make the specification shorter (again):

$$
\begin{aligned}
&Mux(ci_{Sink}, co_{Sink}) \stackrel{def}{=} ci_{Sink} \downarrow .co_{Sink} \downarrow .Mux'(ci_{Sink}, co_{Sink}, \langle\rangle) \\
&Mux'(ci_{Sink}, co_{Sink}, \vec{i}) \stackrel{def}{=} \\
&\quad (co_{Sink}?(so).so \downarrow .Mux''(ci_{Sink}, co_{Sink}, \vec{i}, so)) \\
&\quad + (ci_{Sink}?(si) \\
&\qquad .((si \downarrow .Mux'(ci_{Sink}, co_{Sink}, \langle i_1, ..., i_{|\vec{i}|}, si\rangle))\{if\ \ si \notin \vec{i}\} \\
&\qquad + (si \uparrow .Mux'(ci_{Sink}, co_{Sink}, \vec{i} \setminus si))\{if\ \ si \in \vec{i}\}))
\end{aligned}
$$

$$
\begin{aligned}
&Mux''(ci_{Sink}, co_{Sink}, \vec{i}, o_{Source}) \stackrel{def}{=} \\
&\quad (co_{Sink}?(so) \\
&\qquad .((o_{Source} \uparrow .so \downarrow .Mux''(ci_{Sink}, co_{Sink}, \vec{i}, so))\{if\ \ so \neq o_{Source}\} \\
&\qquad + (o_{Source} \uparrow .Mux'(ci_{Sink}, co_{Sink}, \vec{i})\ )\{if\ \ so = o_{Source}\})) \\
&\quad + (ci_{Sink}?(si) \\
&\qquad .((si \downarrow .Mux'(ci_{Sink}, co_{Sink}, \langle i_1, ..., i_{|\vec{i}|}, si\rangle))\{if\ \ si \notin \vec{i}\} \\
&\qquad + (si \uparrow .Mux'(ci_{Sink}, co_{Sink}, \vec{i} \setminus si))\{if\ \ si \in \vec{i}\})) \\
&\quad + ((i?(data)\ +)^{\{\forall i \in \vec{i}\}}; o_{Source}!\langle data\rangle \\
&\qquad .Mux''(ci_{Sink}, co_{Sink}, \vec{i}, o_{Source}))\{if\ \ |\vec{i}| \geq 1\}
\end{aligned}
$$

At creation, the multiplexer process Mux receives the sink-ends of the C_i and C_o channels and connects to them. After this, it calls process Mux'. Within this process either an input sink-end is added or removed from/to the vector $\vec{i}$, or an output source-end is received and process Mux'' is called. Within this last process, there are three possibilities: (1) An output source-end is replaced or removed. Or, (2) an input sink-end is added or removed. Or, (3) a value is non-deterministically taken from one of the input sink-ends (if there is at least one available), and written into the output source-end.

In Figure 9.6(c) we give an example connector that consists of one non-deterministic multiplexer, one synchronous channel, one synchronous spout channel, and one lossy synchronous channel. The PN connector is composed using the σ function and its construction is analogous to the one of the replicator (see Section 9.4.1). We give the MoCha-π specification of the connector using the dynamic version of the multiplexer (an example of a specification using a static coordination component is given in Section 9.4.1):

$$
\begin{aligned}
&System \stackrel{def}{=} new(ci_{so}, ci_{si}, co_{so}, co_{si}, so_a, si_a, so_b, si_b, si1_c, si2_c, so_d, si_d) \\
&\quad (SYNCH(ci_{so}, ci_{si}) \mid SYNCH(co_{so}, co_{si}) \mid FIFO(so_a, si_a) \\
&\quad (SYNCH(so_b, si_b) \mid SYNCHSPOT(si1_c, si2_c) \mid LOSSYSYNCH(so_d, si_d) \\
&\quad \mid SYNCHDRAIN(so_d, so_{d2}) \mid Mux(ci_{si}, co_{si}) \\
&\quad \mid ci_{so} \downarrow .co_{so} \downarrow .co_{so}!\langle so_d\rangle.ci_{so}!\langle si_a\rangle.ci_{so}!\langle si_b\rangle.ci_{so}!\langle si_c\rangle.ci_{so}!\langle si1_c\rangle)
\end{aligned}
$$

This time we don't encapsulate the connector within a process so that we can easily dynamically change its configuration any time we wish to do so. We let the *System* process create all the entities and set them up. This process also configures the Mux by not only creating it, but also passing all the needed channel-ends to obtain the configuration of Figure 9.6(c). Naturally, this passing of channel-ends can also be delegated to some component if desired. Adding write and take processes to above specification is analogous to the way it is done in Section 9.4.1.

Two writing components (A and B) and two taking components (C and D) are using our example connector. Components B and C have to wait until there are selected by the multiplexer. If B is selected it writes a value that is multiplexed to component D. If C is selected it can take a value from the synchronous spout channel, another value coming from the other end of this channel is multiplexed to D. Component A needs not to wait for writing values to the connector, all its written values are stored in a FIFO channel until taken out by the multiplexer. Component

D is composed to the multiplexer with a lossy synchronous channel, therefore, every time that D does not perform a take operation a multiplexed value gets lost.

9.4.3 Ordered Multiplexer

The *ordered multiplexer* is a deterministic multiplexer that has many input channels and one output channel (see Figure 9.6(a)). This coordination component selects a value from the first sink-end of the input channels, then from the second one, from the third one, and so on. When it covers all sink-ends, it starts again with the first one. The behavior of this multiplexer resembles the one of the 2-input Reo ordering connector (see Figure 9.2(d)). The difference is that our coordination component can have many input channels. We give its MoCha-π specification:

$$
\begin{aligned}
Mux(\vec{i}, o_{Source}) &\stackrel{def}{=} (i \downarrow .)^{\{\forall i \in \vec{i}\}}; o_{Source} \downarrow .Mux'(\vec{i}, o_{Source}) \\
Mux'(\vec{i}, o_{Source}) &\stackrel{def}{=} (i_k?(data).o_{Source}!\langle data \rangle.)^{\{\forall i_k \in \vec{i}, k=1 to |\vec{i}|\}}; Mux'(i_{Sink}, \vec{o})
\end{aligned}
$$

The difference with the non-deterministic version is that process Mux', rather than choosing one sink-end non-deterministically each time, it sequentially takes a value from each sink-end in vector $\vec{i}$ starting with the first one and ending with the last one while writing the acquired values to the output source-end each time. When this sequence is done, the process repeats itself.

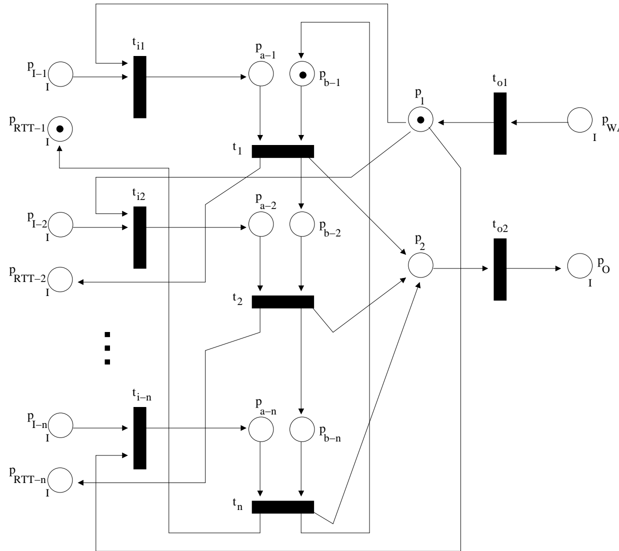


Figure 9.8: Ordered Multiplexer PN

In Figure 9.8 we give a PN that specifies the concurrent behavior of the ordered multiplexer. The initial configuration of this PN is $\{p_{RTT-1}, p_1, p_{b-1}\}$. The cyclic sequence of n places p_{b-k} (with $1 \leq k \leq n$) ensures that the first value is taken for the

first sink-end, then from the second one, from the third one, etc. For example, the firing sequence from the first multiplexed value is: $\{p_{I-1}, p_1, p_{b-1}\}[t_{i1}\rangle\{p_{a-1}, p_{b-1}\}[t_1\rangle$ $\{p_{RTT-2}, p_{b-2}, p_2\}[t_{to2}\{p_{RTT-2}, p_{b-2}, p_O\}$. The source-end of the output channel takes the token of the output place, and, sometime later, acknowledges the write by inserting a token in the p_{WA} place. Then, transition t_{o1} fires and place p_1 is filled with a token again. After this, the value of the next sink-end is ready to be multiplexed as soon as there is a token in place p_{I-2}. When the value from the last sink-end is multiplexed, the first sink-end is next again.

Analogous to the non-deterministic version, for the dynamic ordered multiplexer, we add two extra incoming mobile channels to the static version. The MoCha-π specification is the same except for the last line which we change into:

$$+ \ ((i_k?(data).o_{Source}!\langle data\rangle.)^{\{\forall i_k \in \vec{i}, k=1 to |\vec{i}|\}};Mux''(ci_{Sink}, co_{Sink}, \vec{i}, o_{Source}))$$

9.4.4 Non-Deterministic Demultiplexer

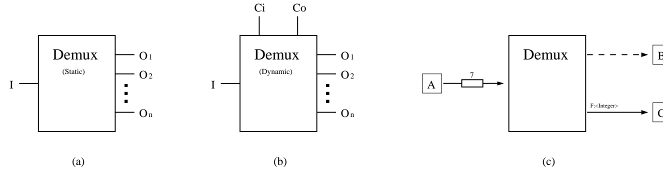


Figure 9.9: Demultiplexer

The *non-deterministic demultiplexer* (see Figure 9.9(a)) has one input channel and many output channels. This coordination component takes a value from the sink-end of the input channel and non-deterministically writes it to (only) one of the source-ends of the output channels. We give its MoCha-π specification:

$$Dmx(i_{Sink}, \vec{o}) \stackrel{def}{=} \ i_{Sink} \downarrow .(o \downarrow .)^{\{\forall o \in \vec{o}\}};Dmx'(i_{Sink}, \vec{o})$$
$$Dmx'(i_{Sink}, \vec{o}) \stackrel{def}{=} \ i_{Sink}?(data).(\ (o!\langle data\rangle \ + \ )^{\{\forall o \in \vec{o}\}} \ );Dmx'(i_{Sink}, \vec{o})$$

The process Dmx receives at creation the sink-end i_{Sink} of the incoming channel and a vector $\vec{o}$ with the source-ends of all the output channels. At initialization the process connects to all channel-ends. The actual demultiplexing is given by process Dmx'. This process takes a value from the sink-end and non-deterministically writes it to one of the source-ends. After the write action succeeds, the process repeats itself.

In Figure 9.10 we give a PN that specifies the concurrent behavior of the non-deterministic demultiplexer. This PN has one input side (or taking side) that consists of the interface places $\{p_I, p_{RTT}\}$, and n output sides (or writing sides) that consists of the interface places $\{p_{O-k}, p_{WA-k}\}$ where $1 \leq k \leq n$. The initial configuration of our PN is $\{p_{RTT}, p_1\}$. When composed with mobile channels, the sink-end of the input channels takes the token of place $\{p_{RTT}\}$ and puts one back in place p_I to indicate that a value is available from this end. The PN takes this value by

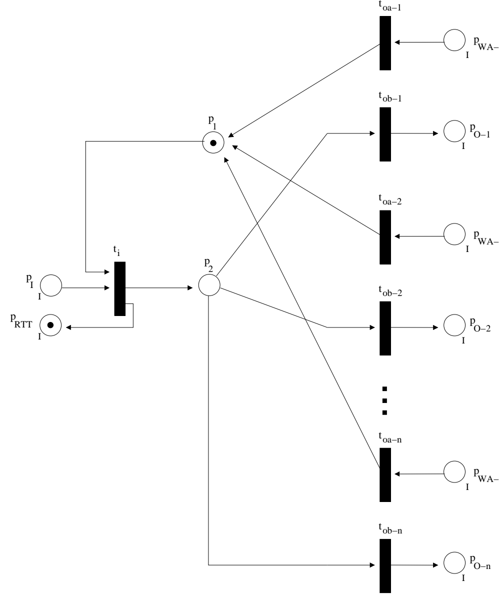


Figure 9.10: Non-Deterministic Demultiplexer PN

executing the sequential firing step: $\{p_I, p_1\}[t_i\rangle\{p_2, p_{RTT}\}$. Once a token is in place p_2, the demultiplexer non-deterministically chooses one output transition for the next step. For example, the second output transition: $\{p_2, p_{RTT}\}[t_{ob-2}\rangle\{p_{O-2}, p_{RTT}\}$. The source-end that is connected to this particular output takes the token of place p_{O-2}, and sometime later acknowledges the write by inserting a token in the p_{WA-2} place. If this is the case, then transition t_{oa-2} fires and fills place p_1 with a token again. In this new configuration, the demultiplexer is ready to take the next value.

Naturally, we also have a dynamic version of the non-deterministic demultiplexer (see Figure 9.9(b)). As with the previous coordination components we add two incoming channels C_i and C_o. The MoCha-π specification of this version is the same as the one for the dynamic replicator (see Section 9.4.1), except that we substitute the name process *Rep* to *Dmx* and we change the last line into:

$$+ \ (i_{Sink}?(data).(\ (o!\langle data\rangle\ +\ )^{\{\forall o \in \vec{o}\}}\ );Dmx''(ci_{Sink}, co_{Sink}, i_{Sink}, \vec{o}))$$

In Figure 9.9(c) we give an example connector that consists of one non-deterministic demultiplexer, one FIFO channel with a capacity of seven values, one lossy synchronous channel, and one filter channel with an Integer as its pattern. While introducing the previous components we already gave examples of how to construct connectors by composing the semantics of the MoCha-π calculus and the Petri Nets formalism (see Sections 9.4.1 and 9.4.2).

One writing component (A) and two taking components (B, and C) are using our example connector. Component A can write up to seven values into the connector. After this, the FIFO channel is full and A must wait until the demultiplexer takes out a value of the channel first. When this is the case, the value gets non-

deterministically demultiplexed to one of the three taking components. However, values can get lost with our connector. This is the case either, when component B is selected but this component is not currently taking, or when C is selected but the value is not of type `Integer`.

9.4.5 Ordered Demultiplexer

The *ordered demultiplexer* (see Figure 9.9(a)) is a deterministic demultiplexer that has one input channel and many output channels. This coordination component takes a value from the sink-end of the input channel and writes it to the first source-end of the output channels, then to the second one, the third one, etc. When it covers all source-ends, it starts again with the first one. We give its MoCha-π specification:

$$
\begin{aligned}
Dmx(i_{Sink}, \vec{o}) &\stackrel{def}{=} \quad i_{Sink} \downarrow .(o \downarrow .)^{\{\forall o \in \vec{o}\}}; Dmx'(i_{Sink}, \vec{o}) \\
Dmx'(i_{Sink}, \vec{o}) &\stackrel{def}{=} \quad (i_{Sink}?(data).o_k!\langle data \rangle.)^{\{\forall o_k \in \vec{o}, k=1 to |\vec{o}|\}}\ ); Dmx'(i_{Sink}, \vec{o})
\end{aligned}
$$

The difference with the previous version is that process Dmx', rather than choosing one source-end non-deterministically each time, it sequentially writes a value to each source-end in vector $\vec{o}$ starting with the first one and ending with the last one. When this sequence is done, the process repeats itself.

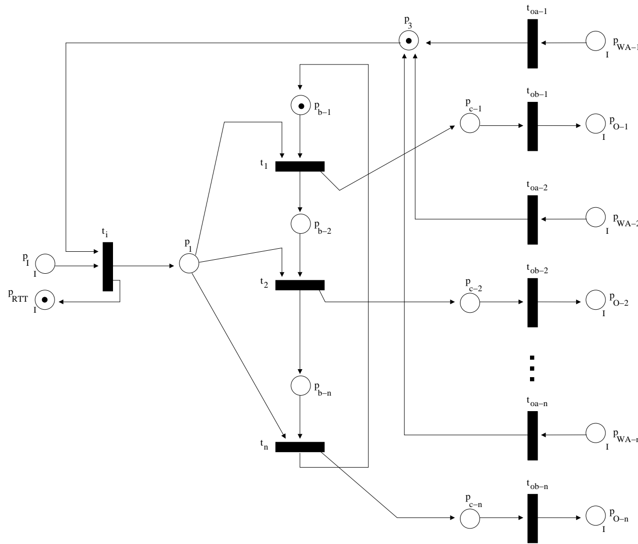


Figure 9.11: Ordered Demultiplexer PN

In Figure 9.11 we give a PN that specifies the concurrent behavior of the ordered demultiplexer. The initial configuration of this PN is $\{p_{RTT}, p_{b-1}, p_3\}$. There are two differences with the non-deterministic version. One is that place p_3 is now the one that ensures that the demultiplexer does not take a new value before the previous write action succeeds. And two, a cyclic sequence of n places p_{b-k} (where $1 \leq k \leq n$)

is added to ensure that the first value goes to the first source-end output, the second to the second one, and so on. The demultiplexer takes a value by performing the sequential firing step: $\{p_I, p_{b-k}, p_3\}[t_i\rangle\{p_1, p_{b-k}\}$. From this configuration, the value leads to an output place p_{O-k} depending on which place p_{b-k} currently contains a token.

Analogous to the non-deterministic version, for the dynamic ordered demultiplexer, we add two extra incoming mobile channels to the static version. The MoCha-π specification is the same except for the last line which we change into:

$$+ \; ((i_{Sink}?(data).o_k!\langle data\rangle.)^{\{\forall o_k \in \vec{o}, k=1 to |\vec{o}|\}} \;);Dmx''(ci_{Sink}, co_{Sink}, i_{Sink}, \vec{o}))$$

9.4.6 Write Gate Transistor

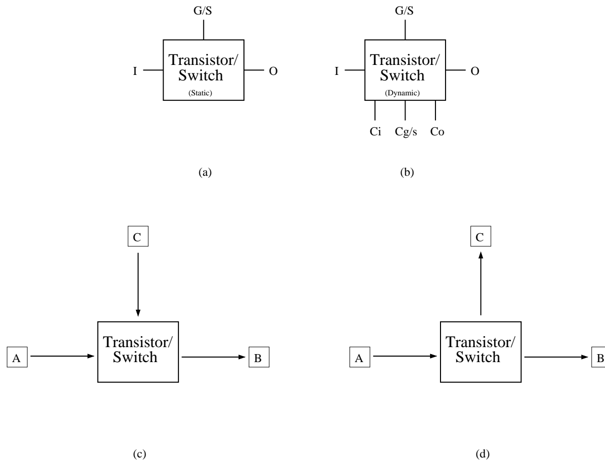


Figure 9.12: Transistors and Switches

The *write gate transistor* (see Figure 9.12(a)) has one input channel *I*, one output channel *O*, and one incoming gate channel *G*. Every time that this coordination component successfully takes any value from the sink-end of the gate channel, it takes one value from the sink-end of the input channel and writes it to the source-end of the output channel. The behavior of this transistor is equal to the one of the Reo write-cue regulator of Figure 9.2(d). We give MoCha-π specification of our transistor:

$$\begin{aligned}
&WGT(i_{Sink}, g_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad i_{Sink}\downarrow .g_{Sink}\downarrow .o_{Source}\downarrow .WGT'(i_{Sink}, g_{Sink}, o_{Source}) \\
&WGT'(i_{Sink}, g_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad g_{Sink}?(dg).i_{Sink}?(data).o_{Source}!\langle data\rangle.WGT'(i_{Sink}, g_{Sink}, o_{Source})
\end{aligned}$$

The transistor process initially connects to all the given channel-ends. After which, it repeats the following cycle: it takes a value from the gate sink-end g_{Sink} (the value

gets lost), it takes a value from the input sink-end i_{Sink}, and finally, it writes the obtained value to the output source-end o_{Source}.

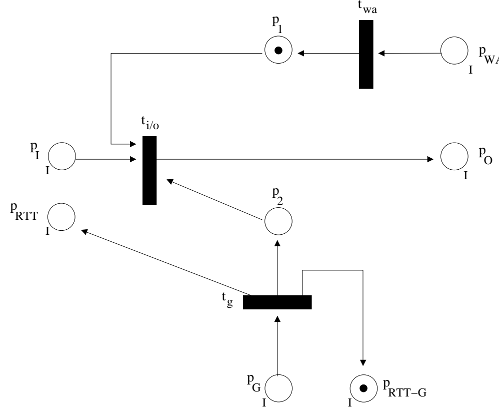


Figure 9.13: Write Gate Transistor PN

In Figure 9.13 we give the PN of the write gate transistor. The PN has one input side consisting of the (interface) places $\{p_I, p_{RTT}\}$, one output side consists of the places $\{p_O, p_{WA}\}$, and one gate side (from which the transistor takes from) consisting of the places $\{p_G, p_{RTT-G}\}$. The initial configuration is $\{p_1, p_{RTT-G}\}$, from which no value can "pass through" the transistor due to the fact that place p_2 is lacking a token for this to happen. However, if a write succeeds on the gate side (by firing transition t_g) the new configuration $\{p_1, p_2, p_{RTT-G}, p_{RTT}\}$ allows one value to pass through. We give the sequential firing step: $\{p_I, p_1, p_2, p_{RTT-G}\}[t_{i/o}\rangle\{p_O, p_{RTT-G}\}$. When the output write action gets acknowledge, place p_1 gets filled with a token again. This allows the next value to go through when another write succeeds on the gate side.

For the dynamic version of the write gate transistor (see Figure 9.12(b)) we add three incoming channels meant for adding and removing channel-ends (as specified by the protocol we previously described). Through channel C_i the transistor receives sink channel-ends to take the input values from. Through channel C_o the transistor receives source channel-ends to write values to. And, through channel C_g the transistor receives sink channel-ends to take values from the gate channel. We give the MoCha-π specification of the dynamic version. This specification follows the same pattern as the ones of the previous dynamic coordination components. The difference is that we have an extra incoming channel C_g. Therefore, we use the if-statements again to make the specification shorter:

$$
\begin{aligned}
&WGT(ci_{Sink}, cg_{Sink}, co_{Source}) \stackrel{def}{=} \\
&\quad new\ \epsilon\ (ci_{Sink} \downarrow .cg_{Sink} \downarrow .co_{Source} \downarrow \\
&\quad .WGT'(ci_{Sink}, cg_{Sink}, co_{Source}, \epsilon, \epsilon, \epsilon)) \\
&WGT'(ci_{Sink}, cg_{Sink}, co_{Source}, i_{Sink}, g_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad (ci_{Sink}?(si) \\
&\qquad .((i_{Sink} \uparrow .si \downarrow .WTG'(..., si, g_{Sink}, co_{Source}))\{if\ \ si \neq i_{Sink}\} \\
&\qquad + (i_{Sink} \uparrow .WTG'(..., \epsilon, g_{Sink}, co_{Source})\ )\{if\ \ si = i_{Sink}\})) \\
&\quad (cg_{Sink}?(si) \\
&\qquad .((g_{Sink} \uparrow .si \downarrow .WTG'(..., si, co_{Source}))\{if\ \ si \neq g_{Sink}\} \\
&\qquad + (g_{Sink} \uparrow .WTG'(..., \epsilon, co_{Source})\ )\{if\ \ si = g_{Sink}\})) \\
&\quad (co_{Sink}?(so) \\
&\qquad .((o_{Source} \uparrow .so \downarrow .WGT'(..., so))\{if\ \ so \neq o_{Source}\} \\
&\qquad + (o_{Source} \uparrow .WGT'(..., \epsilon)\ )\{if\ \ so = o_{Source}\})) \\
&\quad + (g_{Sink}?(dg).i_{Sink}?(data).o_{Source}!\langle data\rangle \\
&\quad .WGT'(...))\{if\ \ i_{Sink} \neq g_{Sink} \neq o_{Source} \neq \epsilon\}
\end{aligned}
$$

At creation, the transistor process WGT receives the sink-ends of the C_i, C_o, and C_g channels and connects to them. After this, it calls process WGT'. Within this process there are four possibilities after which the process repeats itself again. Either, (1) the input sink channel-end, (2) the gate sink channel-end, or (3) the output source channel-end, is added or replaced or removed. Or (4), the transistor takes a value from the gate sink-end, takes a value from the input sink-end, and then writes the last acquired value to the source-end (if all the ends are available).

In Figure 9.12(c) we give an example connector that consists of one write gate transistor, and three synchronous channels. Two writing components (A and C) and one taking component (B) are using our connector. Component B takes the values that A is writing to the connector. However, A is allowed to write only every time C does as well. Therefore, with this connector, component C regulates the communication between A and B.

9.4.7 Take Gate Transistor

The *take gate transistor* (see Figure 9.12(a)) has one input channel I, one output channel O, and one outgoing gate channel G. Every time that this coordination component successfully writes any value to the source-end of the gate channel, it takes one value from the sink-end of the input channel and writes it to the source-end of the output channel. The behavior of this transistor is based on the Reo *take-cue regulator* as given in [Arb02]. We give the transistor's MoCha-π specification:

$$
\begin{aligned}
&TGT(i_{Sink}, g_{Source}, o_{Source}) \stackrel{def}{=} \\
&\quad i_{Sink} \downarrow .g_{Source} \downarrow .o_{Source} \downarrow .TGT'(i_{Sink}, g_{Source}, o_{Source}) \\
&TGT'(i_{Sink}, g_{Source}, o_{Source}) \stackrel{def}{=} \\
&\quad new\ dg\ (g_{Source}!\langle dg\rangle.i_{Sink}?(data).o_{Source}!\langle data\rangle) \\
&\quad ;TGT'(i_{Sink}, g_{Source}, o_{Source})
\end{aligned}
$$

The transistor process initially connects to all the given channel-ends. After which, it repeats the following cycle: it writes a value to the gate source-end g_{Source}, it

takes a value from the input sink-end i_{Sink}, and finally, it writes the obtained value to the output source-end o_{Source}.

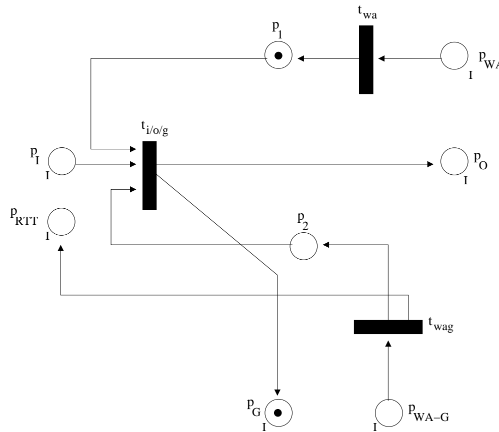


Figure 9.14: Take Gate Transistor PN

In Figure 9.14 we give the PN of the take gate transistor. The initial configuration is $\{p_1, p_{_G}\}$, from which no value can "pass through" the transistor due to the fact that place p_2 is lacking a token for this to happen. However, if the source-end of channel G takes the token from place $p_{_G}$, it eventually puts a token back in place $p_{_{WA-G}}$. After which, places p_2 and $p_{_{RTT}}$ get filled with a token and one input value is allowed to "go through" (due to the firing of transition t_{wag}). We give the sequential firing step: $\{p_{_I}, p_1, p_{2o}\}[t_{i/o/g}\rangle\{p_{_O}, p_{_G}\}$. When the output write action gets acknowledge, place p_1 gets filled with a token again. This brings the PN back to the initial configuration. The next value can go through the take gate transistor only when another take action succeeds on its gate side.

Analogous to the write gate transistor, for the generic dynamic take gate transistor, we add three extra mobile channels to the static version. The MoCha-π specification is the same except for that we replace all instances of g_{Sink} for g_{Source}, and the last line which we change into:

$$+ \ (new \ dg \ (g_{Source}!(dg).i_{Sink}?(data).o_{Source}!\langle data\rangle); WGT'(...)$$

In Figure 9.12(d) we give an example connector that consists of one take gate transistor, and three synchronous channels. One writing component (A) and two taking components (B and C) are using our connector. As with the write gate transistor connector example, component C regulates the communication between A and B. However, this time it does so by taking a value from the connector each time instead of writing to it.

9.4.8 Write Switch

The *write switch* (see Figure 9.12(a)) has one input channel I, one output channel O, and one incoming switch channel S. When the switch of the coordination component

is on, values are taken from the sink-end of the input channel and written to the source-end of the output channel. When the switch is off, nothing happens. The switch alternates between the on and off mode in reaction to successful take operations from the sink-end of the switch channel; i.e. some component externally writes a value to the switch channel. Initially, the switch is off. We give the MoCha-π specification:

$$
\begin{aligned}
&WS(i_{Sink}, s_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad i_{Sink} \downarrow .s_{Sink} \downarrow .o_{Source} \downarrow .\, WS'(i_{Sink}, s_{Sink}, o_{Source}) \\
&WS'(i_{Sink}, s_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad s_{Sink}?(switch).\, WS''(i_{Sink}, s_{Sink}, o_{Source}) \\
&WS''(i_{Sink}, s_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad (i_{Sink}?(data).o_{Source}!\langle data\rangle.\, WS''(i_{Sink}, s_{Sink}, o_{Source})) \\
&\quad +(s_{Sink}?(switch).\, WS'(i_{Sink}, s_{Sink}, o_{Source}))
\end{aligned}
$$

The switch process WS initially connects to all the given channel-ends. After which it operates in two modes, on and off, starting with the off mode. Process WS' specifies the off mode. This process takes a value from the sink-end of the switch channel, when this action succeeds the write switch goes into the on mode. This mode is specified by process WS''. In this process either a value is taken from the input sink-end and written to the output source-end, or a value from the switch sink-end is taken and the switch goes back into the off mode.

In Figure 9.15 we give the PN of the write switch. The initial configuration is $\{p_1, p_{RTT-S}\}$, where the switch is off. No value can be taken by the write switch since place p_3 is lacking a token. To turn on the switch, the sink-end of the switch channel writes a value by taking the token of place p_{RTT-S} and putting one back in place p_S. Then, the only thing that can happen is that transition t_{s-1} fires: $\{p_1, p_S\}[t_{s-1}\rangle\{p_1, p_3, p_4, p_{RTT}, p_{RTT-S}\}$. This step puts a token in the cyclic path that is constituted by places p_2 and p_3 and by transitions $t_{i/o}$ and t_c. This path is meant for enabling series of continuous takes and writes. For example, if the switch in on and there is an input available, we get the following firing sequence: $\{p_1, p_3, p_4, p_I, p_{RTT-S}\}[t_{i/o}\rangle\{p_O, p_2, p_4, p_{RTT-S}\}[t_c\rangle\{p_O, p_3, p_4, p_{RTT}, p_{RTT-S}\}$. From this last configuration, if the previous write gets acknowledge (place p_1 contains a token again) the next take/write series can take place. To turn the switch off, the sink-end of the switch channel writes another value. Due to the token in place p_4 the only possibility is to fire transition t_{s-2}. This transition takes the token out of the cyclic path so that no more series of take/write actions are possible. Observe, that as long as transition t_{s-2} does not fire, the write of the switch sink-end is still not finished. Meanwhile, several series of write/take actions may still occur.

For the dynamic version of the write switch (see Figure 9.12b) we add three incoming channels meant for adding and removing channel-ends. Through channel C_i the transistor receives sink channel-ends to take the input values from. Through channel C_o the transistor receives source channel-ends to write values to. And, through channel C_s the transistor receives sink channel-ends to take values from the switch channel. We give the MoCha-π specification:

$$
\begin{aligned}
&WS(ci_{Sink}, cs_{Sink}, co_{Source}) \stackrel{def}{=} \\
&\quad new\ \epsilon\ (ci_{Sink} \downarrow .cs_{Sink} \downarrow .co_{Source} \downarrow \\
&\quad .WS'(ci_{Sink}, cs_{Sink}, co_{Source}, \epsilon, \epsilon, \epsilon)) \\
&WS'(ci_{Sink}, cs_{Sink}, co_{Source}, i_{Sink}, s_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad (ci_{Sink}?(si) \\
&\qquad .((i_{Sink} \uparrow .si \downarrow .WTG'(..., si, s_{Sink}, co_{Source}))\{if\ \ si \neq i_{Sink}\} \\
&\qquad + (i_{Sink} \uparrow .WTG'(..., \epsilon, s_{Sink}, co_{Source})\ )\{if\ \ si = i_{Sink}\})) \\
&\quad (cs_{Sink}?(si) \\
&\qquad .((s_{Sink} \uparrow .si \downarrow .WTG'(..., si, co_{Source}))\{if\ \ si \neq s_{Sink}\} \\
&\qquad + (s_{Sink} \uparrow .WTG'(..., \epsilon, co_{Source})\ )\{if\ \ si = s_{Sink}\})) \\
&\quad (co_{Sink}?(so) \\
&\qquad .((o_{Source} \uparrow .so \downarrow .WS'(..., so))\{if\ \ so \neq o_{Source}\} \\
&\qquad + (o_{Source} \uparrow .WS'(..., \epsilon)\ )\{if\ \ so = o_{Source}\})) \\
&\quad + (s_{Sink}?(switch).WS''(...))\{if\ \ i_{Sink} \neq s_{Sink} \neq o_{Source} \neq \epsilon\} \\
&WS''((ci_{Sink}, cs_{Sink}, co_{Source}, i_{Sink}, s_{Sink}, o_{Source}) \stackrel{def}{=} \\
&\quad (i_{Sink}?(data).o_{Source}!\langle data\rangle.WS''(...)) \\
&\quad + (s_{Sink}?(switch).WS'(...))
\end{aligned}
$$

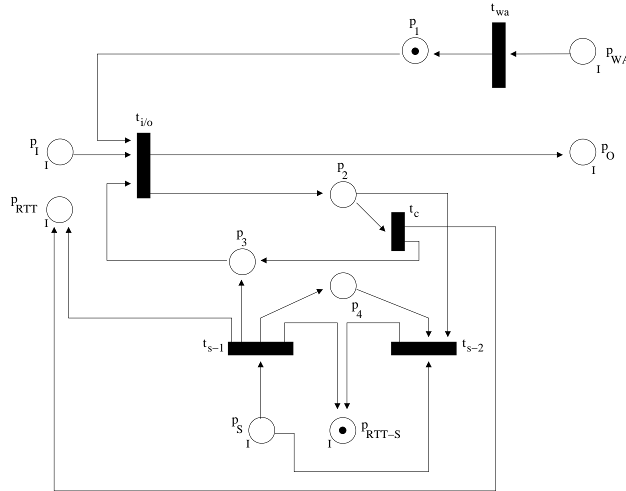


Figure 9.15: Write Switch PN

Just as with the static version, process WS' specifies the off mode. Within this process there are four possibilities. Either, (1) the input sink channel-end, (2) the switch sink channel-end, or (3) the output source channel-end, is added or replaced or removed. Or, (4), the switch is turned on by taking a value from the switch sink-end. The process WS'' specifies the on mode, where either a value is taken from the input sink-end and written to the output source-end, or a value from the switch sink-end is taken and the switch goes back into the off mode.

In Figure 9.12(c) we give an example connector that consists of one write switch,

and three synchronous channels. Two writing components (A and C) and one taking component (B) are using our connector. Component B takes the values that A writes to the connector. However, component C controls this flow of values, for with every successful write operation that is performs it it starts or stops the flow (initially there is no flow).

9.4.9 Take Switch

The *take switch* (see Figure 9.12(a)) has one input channel I, one output channel O, and one outgoing switch channel S. Analogous to the write switch, if the switch is on values are taken from the sink-end of the input channel and written to the source-end of the output channel. When the switch is off, nothing happens. The switch goes on and off when the coordination component successfully writes a value to the sink-end of the switch channel; i.e. some component externally takes a value from the switch channel. Initially, the switch is off. We give the MoCha-π specification:

$$
\begin{aligned}
&WS(i_{Sink}, s_{Source}, o_{Source}) \stackrel{def}{=} \\
&\quad new\ switch\ (i_{Sink} \downarrow .s_{Source} \downarrow .o_{Source} \downarrow .\,WS'(i_{Sink}, s_{Source}, o_{Source})) \\
&WS'(i_{Sink}, s_{Source}, o_{Source}) \stackrel{def}{=} \\
&\quad s_{Source}!\langle switch\rangle .\,WS''(i_{Sink}, s_{Source}, o_{Source}) \\
&WS''(i_{Sink}, s_{Source}, o_{Source}) \stackrel{def}{=} \\
&\quad (i_{Sink}?(data).o_{Source}!\langle data\rangle .\,WS''(i_{Sink}, s_{Source}, o_{Source})) \\
&\quad +(s_{Source}!\langle switch\rangle .\,WS'(i_{Sink}, s_{Source}, o_{Source}))
\end{aligned}
$$

The switch process initially connects to all the given channel-ends. After which it operates in two modes, on and off, starting with the off mode. Just like with the write switch, process WS' specifies the off mode and process WS'' the on mode. The difference is that WS'' writes a value to the sink-end of the switch channel instead of taking from it.

In Figure 9.16 we give the PN of the take switch. The initial configuration is $\{p_1, p_S\}$, where the switch is off. No value can be taken by the take switch since place p_3 is lacking a token. Basically, we use the same strategy as with the write switch. A cyclic path constituted by places p_2 and p_3 and by transitions $t_{i/o}$ and t_c ensures that once the switch is on, a token cycles around to allow series of take/write actions until the switch is turned off. The switch is turned on, when the source-end of the switch channel takes the token out of place p_S, puts one back in place p_{WA-S} and transition t_{s-1} fires. Due to place p_4 and transition t_{s-2}, the next write by the switch source channel-end turns the take switch off by taking the token out of the cyclic path. After which, the next write turns the switch on again, and so on.

Analogous to the write gate transistor, for the dynamic take gate transistor, we add three extra mobile channels to the static version. The MoCha-π specification is the same except for the last line which we change into:

$$+\ (new\ switch\ (s_{Source}!\langle switch\rangle .\,WS'(...))$$

In Figure 9.12(d) we give an example connector that consists of one take switch, and three synchronous channels. Two writing components (A and C) and one taking

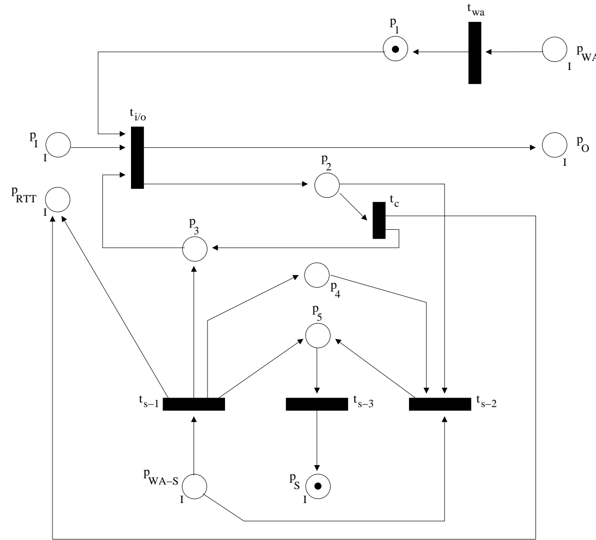


Figure 9.16: Take Switch PN

component (B) are using our connector. This time, component C controls the flow of values between A and B by taking from the connector (instead of writing as with the write switch).

9.4.10 Useful Connectors

In this section we give some examples of useful connectors. This in addition to the small connector examples we gave while introducing the coordination components. We list the connectors of this section in Figure 9.17. We use dashed boxes to denote the connectors, and add to each of them writing and taking components for an easier explanation of their behavior.

The *sequencer* connector (see Figure 9.17(a)) has n output sink channel-ends. This connector outputs values through each of its sink-ends. It does so in a sequential manner starting with the first sink-end and ending with the last one, after which, it starts the sequence again. No value is outputted before the previous one is taken out of the connector. So, component A_2 gets a value only after component A_1 gets one first and so on. To accomplish this behavior, the connector internally consists of one demultiplexer that has one incoming spout channel and n synchronous outgoing channels. The spout channel provides the coordination component with random values, while the synchronous channels ensure that no component is able to take a value before the previous one is finished taking his (by providing synchronization between the demultiplexer and the external taking component). The reader may recognize the similarity with the Reo sequencer connector (see Figure 9.2(e)).

The *ordered drain* connector (see Figure 9.17(b)) has n input source channel-ends. This connectors takes values through each of its source-ends and deletes them. It does so sequentially by taking from the first source-end, then the second one, then

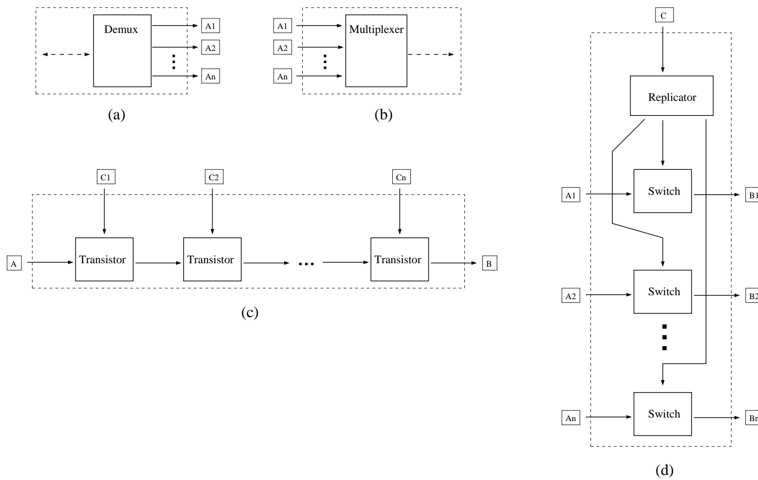


Figure 9.17: Useful Connectors

the third one, etc. When it reaches its last source-end, it starts the sequence again. To accomplish this behavior, the connector internally consists of one multiplexer that has n incoming synchronous channels and one lossy synchronous outgoing channel. As with the previous connector, the synchronous channels ensure that no component writes to the connector before the previous one finishes its write operation. Since there is no entity taking values from the lossy synchronous channel, all values written to it by the multiplexer get lost. Observe, that instead of a lossy synchronous channel we can also use a drain channel instead.

The *multi write gate transistor* connector (see Figure 9.17(c)) has $n + 1$ source-ends, and one sink channel-end. With this connector, component B is allowed to take one value written by A, when all components C have successfully written some value first. As one would expect, the internals of this connector consists of write gate transistors that are all in series. Each individual transistor is associated with one C component. So if there are n C components, then there are n transistors needed for the internals of this connector.

The *n-to-n write switch* connector (see Figure 9.17(d)) has $n + 1$ source, and n sink channel-ends. Every component A_k writes a value that is later on taken by the corresponding component B_k, where $1 \leq k \leq n$. However, they are not always allowed to do this since component C has the power to stop or enable this communication by alternating writes to the connector. Initially, no communication is allowed until C writes a value first. Upon the second written value the communication is stopped. With the third enabled again, and so on. To accomplish this behavior, the connector consists of write switch coordination components that are all lined up in parallel, and where each of them connects a A_k component with its corresponding B_k one. The values written by component C are replicated and given as input to all the switches. Observe, that there is no other synchronization between the switches, so when the communication is enabled the takes of the B components need not to synchronize.

The purpose of the MoCha model for composition is to build connectors that are used in distributed systems. Therefore, we must be cautious that we don't make connectors that are so big and complex that they consume a lot of resources. One strategy, is to use as much coordination components and as less mobile channels as possible. To accomplish this, we can encapsulate frequently occurring connectors (or sub-connectors) into coordination components. For example, it is a good idea to define a coordination component that implements the behavior of the n-to-n write switch connector. Unless, either we dynamically want to reconfigure the connector, or we want to achieve some kind of load balancing strategy in our system.

9.4.11 Composition of Connectors

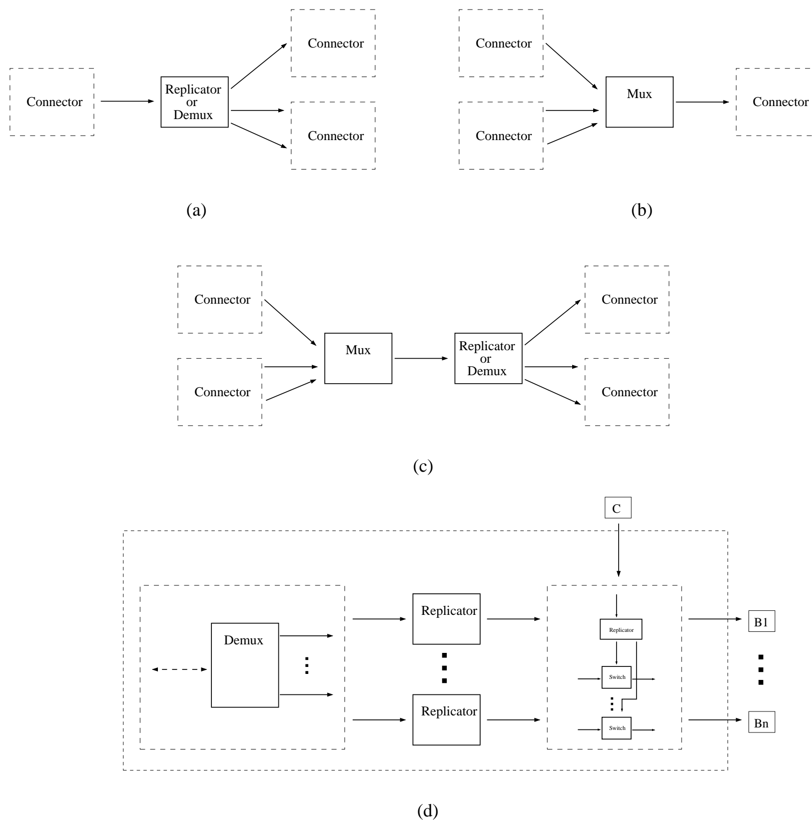


Figure 9.18: Composing Connectors

By using coordination components we can also compose already existing connectors together into new ones. Typically, we use a replicator or a (de) multiplexer for this purpose, however, other components may be used as well (if desired). In Figure 9.18(a) we demonstrate how to obtain a one-to-many connector composition by using either a replicator or a (non-)deterministic demultiplexer coordination

component. In Figure 9.18(b) we show how to obtain a many-to-one connector composition by using a (non-)deterministic multiplexer. In Figure 9.18(c) we combine a (non-)deterministic multiplexer with either a replicator or a (non-)deterministic demultiplexer to obtain a many-to-many connector composition. If we use a demultiplexer then a value is selected from one of the input connectors and given to (only) one of the output ones. If we use a replicator then the value gets replicated to all output connectors. The behavior of this last combination resembles the one of the Reo mixed node (see Figure 9.1(e)).

In Figure 9.18(d), we give an example of a composed connector. This connector outputs values to the n B external components in a sequential manner starting with the first component B and ending with the last one, after which, it starts the sequence again. Component C either enables or disables this process by performing alternating writes to the connector (initially the connector is disabled). We easily implement this connector by using a sequencer and a n-to-n write switch connector. We compose them together by using one-to-one replicators. Observe, that we can also use both one-to-one demultiplexers and one-to-one multiplexers in this case. However, due to the fact that the semantics of the replicator are simpler we use this coordination component instead.

9.5 Distributed Dining Philosophers

We give a small example of a component based system where the component instances are coordinated by connectors. For this example we look at the classical dinning philosophers problem as specified by Dijkstra in [Dijk71]. In [Arb06] a solution to a four philosophers version of this problem is given using Reo. The purpose of this solution is to show the significance of exogenous coordination in component based composition. We give an implementation of this solution using our MoCha coordination components model. This implementation is an example of the kind of Reo connectors that we can implement using MoCha (see Section 9.6.1 for a more detailed discussion about this). Furthermore, it also demonstrates that simple Reo connectors don't always have a trivial straightforward MoCha implementation.

In the dining philosophers problem there are four philosophers sitting around a circular table. Each of them has a plate of spaghetti in front of him and a fork at each side which they have to share with their neighbor (i.e. there are four forks in total). The life of a philosopher consists of periods of thinking and eating. When thinking a philosopher needs only it's brain and nothing else. However, when eating a philosopher requires to have a fork in each of his hands (we assume that our philosophers have good table manners). Philosophers pick up the two forks one at a time and all in the same manner: first the left fork and then the right one. After eating the philosophers release both forks and go back to their main activity (thinking) until they get hungry again. The problem consists of developing a solution to avoid *starvation* and *deadlock*. Deadlock occurs when each of the four philosophers has one fork and no one can get a second fork. Starvation occurs when a philosopher is unable to acquire both forks (might occur independently from deadlock).

The Reo solution of this problem comes in two versions (see [Arb06]): one version that avoids starvation but does not prevent deadlock, and one version that prevents both starvation and deadlock. Figure 9.19(a) shows the first version, it contains

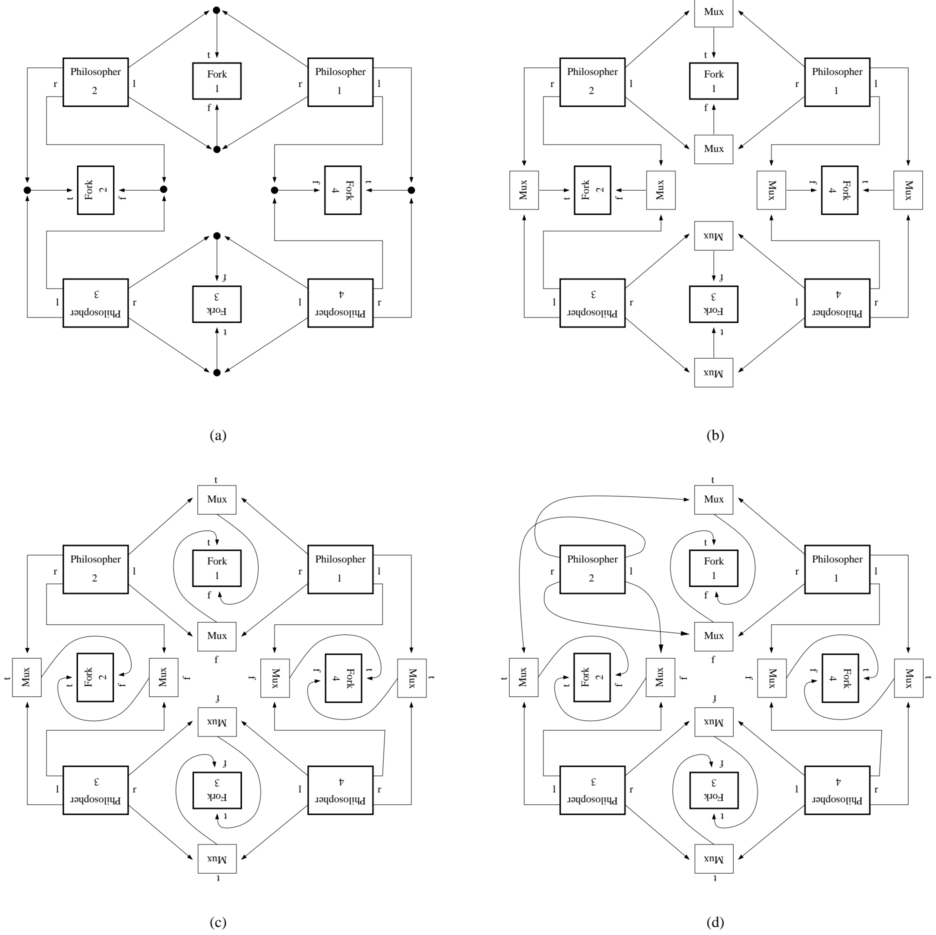


Figure 9.19: The Dining Philosophers

four instances of the philosopher component, four instances of the fork component, twenty four synchronous channels, and eight instances of the Reo mixed node. The behavior of the mobile channels and the mixed nodes are already previously defined in this thesis. However, we must define the formal behavior of the philosopher and the fork component. We start with the first one by giving it's MoCha-π specification:

$$
\begin{array}{ll}
Phil(lt, lf, rt, rf) \stackrel{def}{=} & lt\downarrow.lf\downarrow.rt\downarrow.rf\downarrow.Thinking(lt, lf, rt, rf) \\
Thinking(lt, lf, rt, rf) \stackrel{def}{=} & \tau.Eating(lt, lf, rt, rf) \\
Eating(lt, lf, rt, rf) \stackrel{def}{=} & new(token)(lt!\langle token\rangle.rt!\langle token\rangle.\tau \\
 & .lf!\langle token\rangle.rf!\langle token\rangle.Thinking(lf, lt, rt, rf))
\end{array}
$$

The philosopher process *Phil* receives at creation four source channel-ends, which are lt,lf,rt, and rf. At initialization it connects to these ends and goes to the thinking

mode by calling process *Thinking*. The process models the thinking of a philosopher by performing a τ action (the unobservable state). After thinking a philosopher is ready to eat and calls process *Eating*. A philosopher takes or drops a fork by successfully writing a token to one of the following source-ends: it writes to source-end *lt* to take the left fork, to *rt* to take the right fork, to *lf* to drop the left fork, and to *rf* to drop the right fork. The eating itself is given as a τ action. Thus, as specified by process *Eating* a philosopher picks up the left fork, the right fork, it then eats for a while, drops the left fork, drops the right fork, and finally goes back to the thinking mode again (by calling process *Thinking*).

A fork has two possible states: *taken* and *free*. Therefore, the behavior of the fork component is quite simple and is given by the following MoCha-π specification:

$$\begin{array}{lcl} Fork(t,f) & \stackrel{def}{=} & t\downarrow .f\downarrow .Fork'(t,f) \\ Fork'(t,f) & \stackrel{def}{=} & t?(token).f?(token).Fork'(t,f) \end{array}$$

The process *Fork* receives two source-ends (t and f) at its creation. A successful take from source-end t symbolizes that the fork is currently taken, a successful take from source-end f symbolizes that the fork is currently free. A fork is initially free (as we can see from process *Fork′*), because the only action that can happen is a take from t. After this action the fork is taken, because the only action that can happen next is a take from f. After the "free" action the cycle repeats itself.

For convenience, we annotated the left (l) and right (r) sides of the philosophers (they are facing the table) in the figure. We also indicate the channels that provide the t source-end (for taking the fork) to the forks with a *t*. We annotate the channels that provide the f source-end (free the fork) with a *f*. The channels that are connected to the philosophers at the outer-edge of the figure are used for taking the fork; they provide the *lt* and *rt* source-ends depending on which side of the philosopher the channels are. The channels closer to the center of the figure are used by the philosophers for dropping (or freeing) the fork; they provide the *lf* and *rf* source-ends.

Each fork can be taken and dropped by two philosophers, that is why the mixed nodes are used. Consider what happens in the node that is in between the "take" side of fork 1, philosopher 1, and philosopher 2. If the fork is free and ready to accept a token, as it initially is, whichever one of the two philosophers happens to write a token will succeed in taking the fork. Naturally, it is possible for both philosophers to attempt to take the fork at the same time. In this case, the definition of the mixed node (see Section 9.2) guarantees that only one of them non-deterministically succeeds; the write operation of the other philosopher remains pending until fork 1 is free again. Because a philosophers frees a fork only after it has taken it, there is never any contention at the node that is in between the same philosophers but at the "free" side of fork 1. The composition of channels in this Reo application enables philosophers to repeatedly go through their "eat" and "think" cycles at their leisure, resolving their contentions for taking the same forks. The possibility of starvation is ruled out because the nondeterminism of the mixed node is assumed to be fair (see [Arb06]).

To implement this Reo application of the dinning philosophers in our MoCha model, we must first identify the parts of the Reo connectors that we want to implement using coordination components. Usually these are the parts whose topology

will remain unchanged during the lifetime of the system (static). In this case it is easy, since all the connectors contain only one node each, we substitute each node for such a component. Considering that the nodes choose a value (token) from two inputs and pass it on to one output, then a logical choice is to use instances of the non-deterministic multiplexer. Figure 9.19(b) shows the result. Indeed, just like the Reo nodes, the multiplexers ensure that if one of the two philosophers adjacent to it writes a token, then this philosopher will succeed in taking the fork (if the fork is currently free). If both philosophers attempt to take the fork at the same time one of their tokens is non-deterministically chosen. Nevertheless, despite of this behavior, this particular MoCha implementation is wrong.

Consider what happens at the multiplexer on the "take side" of fork 1. Let's assume that the fork is currently taken by philosopher 1. If then philosopher 2 writes a token, the multiplexer takes this token and tries to write it to the fork (all through the synchronous channels). This last write does not succeed (since the fork is currently taken). However, the multiplexer has already taken the token from the philosopher and, therefore, made him "believe" that he has acquired the fork while this is not the case. The Reo node does not have this problem since it posses the non-local connector state information whether or not the fork is currently performing a take operation (it gains this information through the synchronous channel that connects it to the fork). This makes it possible for the Reo node to take a token only when the fork is currently free. In contrast, our multiplexer does not automatically posses this information since the coordination components (and the MoCha channels) don't propagate global state connector information.

One of the options, in order to fix the MoCha implementation, is to extend our connectors with a construction that explicitly propagates the fact that the fork is currently able to take a token (just as Reo does). Fortunately, in this case, we don't need to. It is sufficient to slightly change the topology of our connectors as given in Figure 9.19(c). We change all the outgoing synchronous channels of the multiplexers so that: all the multiplexers on the edge of the table are now connected to the "free" side of the fork, and all the multiplexers on the center of the table are connected to the "take" side of it.

With this new implementation we prevent the case where two philosophers are able to take the same fork at a same time. Suppose fork 1 is free, and either one philosopher (1 or 2) or both philosophers attempt to take it, then the multiplexer either takes the available token or non-deterministically selects the token of one of the two philosophers (as before). Suppose it is philosopher 1 (again) who succeeds in writing the token (as above), and thus acquiring the fork (from his point of view). The multiplexer then tries to write the acquired token to the "free" side of fork 1. However, this write does not succeed because the fork is waiting for a token at the "take" side (see its MoCha-π specification above). Therefore, the multiplexer is not able to take a new token from philosopher 2. After a while, philosopher 1 is done with eating and releases the fork by writing a "free" token. This token then goes to the "take" side of fork 1 and is taken by it. The fork then expects a token from its "free" side. Thus our multiplexer finally succeeds in writing the token he acquired from philosopher 1. After this write, the multiplexer gets back to the initial situation we began with. This behavior shows, that we correctly implemented the Reo application in our MoCha coordination components model.

As mentioned before, this version of the dining philosophers deadlocks. Namely, all the philosophers attempt to pick up their forks in the same order: left-first. If all forks are free and all philosophers attempt to take their left fork at the same time they will all succeed. This brings the system in a situation that there is no free fork at the right side of any philosopher to take. At the same time, no philosopher will relinquish its fork before it finishes its eating cycle. Therefore, the system deadlocks.

Reo is a exogenous coordination language. Therefore, it is easy to produce a deadlock-free version without having to modify the components, nor installing a central authority. It is sufficient to slightly change the topology of the connectors so that one philosophers left and right connections to its adjacent forks are flipped. Since MoCha also provides exogenous coordination, it is easy to implement this modified Reo version. In Figure 9.19(d) we have flipped the connections of philosopher 2. None of the components in the system are aware of this change (not even philosopher 2), nor is any of them modified in any way to accommodate it. The flipping of these connections is purely external to all components.

With this new version, if all philosophers attempt to take their left forks at a same time, philosopher 2 will actually reach for the one on its right side. Naturally he is not aware of the fact that he is aiming for the right fork instead of the left one as he intends to. In this case he competes with philosopher 3, which is also attempting to take its first fork. It makes no difference which one of the two actually acquires the fork. The fact that one of them is not able to acquire any fork ensures that, at all moments, there is always at least one philosopher that manages to acquire both forks. This philosopher is able to complete its eating cycle and will return both forks to the table. Therefore, deadlock never occurs.

More in depth details about the Reo applications we implemented can be read in [Arb06].

9.6 Comparisons and Conclusions

In this chapter, we presented the MoCha model for composition of mobile channels into connectors. This model is based on *coordination components*. These are lightweight components that are meant for linking channels (or connectors) together according to some transparent coordination behavior. This behavior is provided as a MoCha-π specification and a Petri Net (for the static version only). The actual implementation of the coordination components is provided by the MoCha middleware.

The MoCha model takes and implements some of the main ideas of Reo; the first model for mobile channel composition. In Section 9.4, we discussed some of the similarities and differences between the two models. In this section we continue this discussion. Afterward, we characterize a subset of Reo that can be implemented by MoCha.

The main difference between Reo and the MoCha composition model is that they serve different purposes. Reo aims to be a powerful exogenous coordination language based on a calculus of channels. It's primary goal is expressiveness and the easy specification of connectors. The purpose of our MoCha composition model is to provide connectors that are easily and immediately implementable in distributed

systems. Given this main difference it is easy to understand the motivation for all the other ones we explain next.

Reo composes mobile channels by using the concept of a node, which is a logical construct. We have seen in Section 9.3 that when implementing Reo connectors this involves (somehow) mapping the nodes (or a group of nodes and their adjacent channels) into components. In Section 9.5 we gave a simple example of such a mapping. However, this may not always be so clear and easy to do. Especially if we want to achieve an efficient implementation. In contrast, the MoCha model already uses components for channel composition. Therefore, given a connector we can immediately see all its components and distribute them among the locations of a system according to some placement strategy. Also, since with MoCha we are working at the level of components, we can already take efficiency into account when creating our connectors. Due to the fact that the coordination components can implement subparts of Reo connectors, their behavior can be much more complex than the one of Reo nodes.

Reo uses the *join* and *split* operations for the dynamic reconfiguration of connectors. However, these are logical operations that work on graphs. Reo does not specify how these operations are actually (to be) implemented in distributed systems. The MoCha model offers one possible implementation of these operations by making its coordination components dynamic; these components have a protocol for adding and removing channel-ends to/from their interfaces (more details about this protocol is given in Section 9.4).

In Section 9.3, we discussed the issue that Reo nodes need to have *global* state connector information to make certain decisions due to the propagation of synchrony. In Section 9.4, we showed and explained that the MoCha coordination components base their decisions strictly on *local* state connector information. In concrete, this difference means that MoCha components have no means of checking if their outgoing write actions are going to succeed or not, while the Reo nodes do. Therefore, a Reo (source or mixed) node is able to accept and take an incoming value only if all source channel-ends coincident on the node accept the value. In contrast, a MoCha replicator component initially always accepts and takes an incoming value. Then, it writes this value to all outgoing source channel-ends (without knowing if the channels are actually going to take the value). The replicator suspends until all the write actions succeed before taking the next value. Another concrete difference involves the non-deterministic taking of a value. A Reo (sink or mixed) node is able to know which incoming paths of synchronous channels have a value to offer. Therefore, it can safely non-deterministically choose the sink-end of such a path's last channel, even if this channel itself has no value to offer from its point of view. In contrast, the MoCha components cannot look at incoming paths of channels. Therefore, the MoCha multiplexer non-deterministically takes a value from the channels that, from their point of view, currently have one to offer.

We give an example to illustrate the consequences of above difference between Reo nodes and MoCha coordination components. In Figure 9.20(a) we show a Reo connector that composes two synchronous channels together (upper connector). The semantics of the Reo nodes are such that the external behavior of this connector is equivalent with the one below of it; namely, the writes of component A and the takes of component B are synchronized. In Figure 9.20(b) we show a MoCha connector

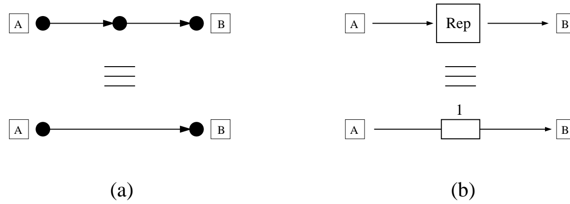


Figure 9.20: The Difference Between a Reo Node and a MoCha Component

that also composes two synchronous channels together by using a replicator. The semantics of the replicator are such that the external behavior of this connector is equivalent with the one below of it; namely, that components A and B communicate with each other using a FIFO 1 channel type. Comparing the two figures, we can see that Reo nodes don't buffer any values that pass through them (they don't need to), while MoCha components have a buffer of (at least) one value (they need to have this buffer). Observe, that instead of using a replicator we can also use a multiplexer for the MoCha connector of Figure 9.20(b).

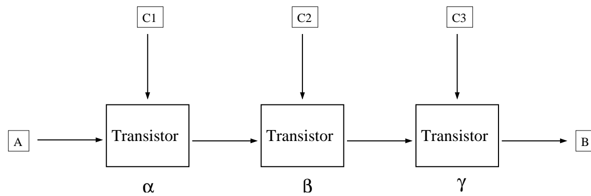


Figure 9.21: Multi Write Gate Transistor

We give another more elaborated example. We used the Reo connector in Figure 9.3 to explain the need for the propagation of connector state information through the Reo nodes. In Figure 9.21 we give a MoCha connector that resembles the behavior of the Reo connector. The behavior of both connectors is that component B successfully takes one value written by A each time that all components C write one "dummy" value to the connector. In the Reo connector, all the write actions and the take action are synchronized; i.e. atomically succeed. In the MoCha connector they are not synchronized. Each transistor coordination component locally checks if there is a value available at its input channel and writes it to its output channel when a value is written to it by its corresponding component C. This leads to cases where, for example, component A successfully writes a value to the connector but the value is pending on the input channel of transistor β because component C_2 didn't write a "dummy" value yet. Or to the case, where there is value pending on the input channel of transistor γ while another write of component A succeeds because component C_1 meanwhile also wrote a next value to the connector. However, the order in which component B receives the values is not altered. They arrive in the same order as component A wrote them into the connector (which is also the case with the Reo connector).

We already have seen in Section 9.4 that the choice for not having the propagation of synchrony makes MoCha connectors more easier to implement in truly distributed systems. Furthermore, as one would expect, this choice also leads to "simpler" and transparent compositional semantics; the compositional semantics of the MoCha connectors are "just" the parallel composition of the MoCha-π semantics of its constituents. In contrast, the need for *global* state connector information (needed for the propagation of synchrony) makes any compositional semantics for Reo far from trivial. For example, see the Reo constraint automata semantics in [ABRS4] or the connector coloring semantics in [CCA05].

9.6.1 Implementing a Subset of Reo

We can implement a subset of Reo using MoCha. The MoCha components already resemble the behavior of existing Reo nodes and connectors (on purpose). Table 9.1 gives an overview of the basic Reo nodes and connectors with their corresponding MoCha implementation that appeared in this chapter.

Reo	**Figure**	**MoCha**	**Figure**
Source node	9.1(c)	Replicator	9.4(b)
Sink node	9.1(a)	Non-determ. multiplexer	9.6(b)
Mixed node (one-to-many)	9.1(e)	Replicator	9.4(b)
Mixed node (many-to-one)	9.1(e)	Non-determ. multiplexer	9.6(b)
Mixed node (many-to-many)	9.1(e)	Replicator + multiplexer	9.18(c)
Ordering connector	9.2(d)	Ordered multiplexer	9.6(b)
Write-cue regulator	9.2(a)	Write gate transistor	9.12(b)
Sequencer	9.2(e)	Sequencer	9.17(a)

Table 9.1: Implementing Reo with MoCha

The subset that MoCha implements, as one would expect from the previous discussions, is the so-called *asynchronous Reo*. This is the Reo subset where the connectors don't need the propagation of synchrony. Typically these connectors consist of asynchronous channel types only. However, synchronous channel types are allowed as long as they don't produce any synchrony propagation. Due to the many similarities between Reo and MoCha, the implementation of the asynchronous Reo subset can be fully automated. Naturally, such an implementation may have less components and less channels than the original Reo connector depending on the implementation strategy and heuristics.

Some of the connectors that need the propagation of synchrony can still be implemented in MoCha by applying clever changes to the connector's topology; for example, see the MoCha implementation of the dining philosophers in Section 9.5. However, this probably only works for small connectors and can certainly not be (easily) automated. For the general case, we need to implement a protocol that takes care of the propagation. However, MoCha does not offer such a protocol yet.

Part V

Conclusion

Chapter 10

Conclusion

We conclude this thesis by giving a short summary of it, reflecting on its main question, and discussing future work.

10.1 A Short Summary

In Chapter 2, we gave an intuitive explanation of mobile channels. A channel consists of exactly two distinct ends that are either of type source or sink. Components may know and refer to channel-ends only, there are no references to channels as a whole. The ends of a channel are *mobile*, hence the name "mobile channels". Channel-end identities can be passed through channels to other components in the system. As well as, channel-ends can physically move from one location to another location in a distributed system. Another important feature of mobile channels is that they provide basic *exogenous coordination*. Channels allow several different types of connections among components without the components themselves knowing which channel types they are dealing with. Examples of channel types are synchronous, lossy synchronous, FIFO, asynchronous drain, etc.

In Chapter 3, we specified mobile channels using the Petri Nets formalism. We described the interface that these channel Petri Nets have towards components (that are also specified as Petri Nets), and showed how to compose them into one Petri Net model. We briefly discussed the analysis and simulation possibilities of these models.

In Chapter 4, we introduced an exogenous coordination calculus, called MoCha-π. A novelty of this calculus is in the fact that channels are not just links but special kinds of processes. This allows to have user defined channel types without having to change the rules of the calculus itself. Another novelty is the fact that our calculus treats channels as resources. Processes must compete with each other in order to gain access to a particular channel.

In Chapters 5 and 8, we presented a coordination model for component-based software systems based on the notion of mobile channels, defined it in terms of a compositional trace-based semantics, and described its implementation in the Java language. This model 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 than coordination of components.

In Chapter 6, we discussed the MoCha middleware, which implements the mobile channels we describe in this thesis. In our discussion, we took the point of view of a distributed system developer that wants to use the middleware but does not want to know anything about the internal implementation details of it. We introduced the main features of the middleware's Application Programming Interface, we provided examples of how to use the middleware, and looked at several applications of the middleware; like, for example, peer-to-peer file-transfer applications.

In Chapter 7, we discussed the implementation details for the MoCha middleware. We conceptually explained the many algorithms and the internal architecture of the middleware. In particular, we focused on the *Java RMI* layer, the peer-to-peer mobile architecture we build upon it, and the implementation of the mobile channels. We also provided performance measurements, by introducing all kinds of experiments and evaluating their results.

In Chapter 9, we investigated the composition of channels. We looked at Reo, the first model for composition of mobile channels into connectors, and implemented a subset of it. We did this by providing a model for composition of mobile channels that is conceptually closer to the actual implementation in distributed systems. Mobile channel composition in our model is done by coordination components. For each such component we gave semantics by providing a MoCha-π specification and a Petri Net. We implemented these components in the MoCha middleware.

10.2 Answering the Main Question of this Thesis

The main question of this thesis is, whether mobile channels are suitable as a communication and coordination mechanism for distributed systems. To answer this question, we throughly investigated different aspects of mobile channels. Next, we briefly go through the main chapters of this thesis and highlight the conclusions of our investigation.

The Petri Nets semantics of Chapter 3 clearly shows the benefits of the exogenous coordination feature that mobile channels provide. All the channel Petri Nets have the same interface towards components, making it for components impossible to recognize the type of the channel that they are using. This allows us to conveniently put any type of channel in between them. The Petri Net taker and writer components that we use in the examples of this chapter are always the same. However, by composing these components with different channel types each time we obtain different Petri Nets with each a different behavior.

The MoCha-π calculus of Chapter 4 confirms the benefits of exogenous coordination that is demonstrated by the Petri Nets semantics; the calculus allows the placement of any (user-defined) type of channel between processes without them knowing how these different channel types affect their behavior. MoCha-π also demonstrates that thanks to the mobility feature of channel-ends, we can easily reconfigure the channel connections among the processes of a system in a dynamical and transparent way; the MoCha-π mobile phones and mobile agent examples clearly show this.

The coordination model for component-based software systems of Chapters 5 and 8 shows that mobile channels promote and enhance the separation of concerns

between the coordination and the computational aspect of (distributed) systems. This makes it easy to develop, maintain and update the coordination part of a system independently of the computational part. In the case of component-based software systems, since components encapsulate computation, we can also develop, maintain and update the computational part independently of the coordination one. Mobile channels make this separation of concerns possible due to their anonymous communication and exogenous coordination features.

The examples in the MoCha middleware chapter (Chapter 6) show that mobile channels offer support for several architectural styles. This conclusion follows from the fact that we can implement the communication/coordination aspects of both centralized and decentralized architectures; in the examples, we compared and implemented a centralized client/server architecture with a decentralized peer-to-peer architecture. The examples also show how we can dynamically change the topology of the system by using channel mobility. In this case, by using the `move`, `connect` and `disconnect` middelware methods. Furthermore, the MoCha middleware illustrates how easy it is to exogenously coordinate components by just replacing the type of the channel that coordinates them with another one. The point to make here is that this replacement happens without having to change or modify the code of the components using this particular channel. The middleware also shows that mobile channels are architecturally very expressive. In the examples it is easy to see which components (potentially) exchange data with each other. This makes it easier to apply tools for analysis of the dependencies and data-flow.

The conclusion we can derive from the implementation of the MoCha middleware and its performance measurements (given in Chapter 7) is that mobile channels can efficiently be implemented in distributed systems. The internal structure of this implementation is such, that the MoCha middleware is able to efficiently implement both centralized and decentralized architectures as described above.

The results of Chapters 6 and 7 are very important, for they show that mobile channels are not just a theoretical concept, but that they can actually be implemented.

By looking at the various examples of connectors in Chapter 9 we can conclude that: the composition of channels is very important for obtaining exogenous coordination behavior that simple basic mobile channels cannot provide. An immediate example of this are connectors with more than two channel-ends. Moreover, a remarkable conclusion that Reo shows is that by defining a model for the composition of simple channels we get a powerful coordination language for the specification of connectors.

After examining different aspects of mobile channels by defining *semantics* for them, *implementing* them, and specifying *composition* of channels into connectors, we can finally conclude that mobile channels are very suitable as a communication and coordination mechanism for distributed systems. We summarize their major features and benefits:

- Anonymous communication.
- Basic exogenous coordination.
- Mobility.

- Support for several architectural styles.
- Efficient implementation in distributed systems.
- Architectural expressiveness.
- Separation of concerns between the coordination and the computational aspect of distributed systems.
- Complex and non-trivial exogenous coordination due to composition.

10.3 Future Work

The investigation we carried out in this thesis suggests many possibilities for future research. Next, we describe some projects that we want to carry out as future work.

We want to extend both the Petri Nets semantics and the MoCha-π calculus in order for them to model Reo. In both cases, we need to introduce the concept of a node and provide a protocol for propagating connector information that implements the operational semantics that Arbab describes in [Arb02]. Such an extension of the Petri Nets semantics can be used for making a theoretical comparison between Petri Nets and Reo. Furthermore, it can also be used for translating Reo to Petri Nets [1]. The extension of the MoCha-π calculus will provide process algebra semantics for Reo.

We also want to extend our model for composition of mobile channels so that it fully implements Reo instead of the subset that it currently covers. For this purpose, we need to add a protocol for the Reo propagation of synchrony in our model. We also need to efficiently implement this protocol in the MoCha middleware.

There is theory available on the verification of Java programming code, as well as enough tool support. For example, in [Abr05] and [BP02] the code of Java programs is annotated with predicates which should hold during program execution when the flow of control reaches the annotated point. A proof system takes these predicates and transforms them in the so-called verification conditions. These conditions are then verified by a theorem prover. Most of this process can be automated as shown by the tool given in [BP02]. We want to use this tool, or one alike, to verify the MoCha middleware. Due to the size of the middleware it is not feasible to verify all of its code. Verification tools are able to annotate some code automatically but only for standard methods and standard verifications. This means that for most of the methods of our middleware we would need to annotate them “by hand”. Therefore, we will primary focus on the Java code that implements the abstract algorithms of the mobile channels. We are interested in verifying the coordination behavior of each channel implementation and the occurrences of deadlock.

The component-based model that we give in this thesis is enough for components to be able to use mobile channels. However, we want to enrich this model by adding more higher programming structures than the basic operations that channels provide. For example, we want to introduce the notion of a service. The internal structure of a service consist of operations on channel-ends that are related to each other according to the some service semantics. The components themselves don't

[1] A translation from Petri Nets to Reo is already provided in [Arb06].

see these channel-end operations anymore, they just call a service and wait until they get their result. Internally, a service call and its parameters are translated into mobile channel operations. The result of these operations (usually the result of the take operations) are grouped together and returned to the calling component.

Bibliography

[ADO99] W.M.P. van der Aalst, J. Desel, A. Oberweis (Eds.), *Business Process Management: Models, Techniques, and Empirical Studies*, Lecture Notes in Computer Science, vol. 1806, Springer-Verlag, 1999.

[Abr05] E. Abraham, *An Assertional Proof System for Multithreaded Java - Theory and Tool Support*, PhD thesis, University of Leiden, ISBN 90-9018908-4, January 2005.

[ALSN01] 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.

[AFW02] J. Aldous, J. Foster, and P. Welch, *CSP Networking for Java (JCSP.net)*, Slides for ICCS 2002 (Global and Collaborative Computing), April 2002.

[And91] G. Andrews, *Paradigms for process interaction in distributed programs*, ACM Computing Surveys, Vol. 23, No. 1, pp. 49-90. March 1991.

[Arb96a] F. Arbab, *Manifold version 2: Language reference manual.* Technical report, CWI, 1996. Available at `http://www.cwi.nl/ftp/manifold/refman.ps.Z`

[Arb96b] F. Arbab, *The IWIM model for coordination of concurrent activities.* In P. Ciancarini and C. Hankin, editors, Coordination Languages and Models, volume 1061 of Lecture Notes in Computer Science, pages 34-56. Springer-Verlag, April 1996.

[Arb98] F. Arbab, *What Do You Mean, Coordination?*, Bulletin of the Dutch Association for Theoretical Computer Science, NVTI, pages 11-22, 1998.

[Arb02] F. Arbab, *A Channel-based Coordination Model for Component Composition*, Tech. Report, Centrum voor Wiskunde en Informatica, Amsterdam, 2002. Available at `http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0203.pdf`

[AR03] F. Arbab, J. Rutten, *A Coinductive Calculus of Component Connectors*, 16th International Workshop on Algebraic Development Techniques (WADT 2002), M. Wirsing, D. Pattinson and R. Hennicker (eds.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 2755, pp. 35-56, 2003.

[Arb04] 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.

[ABBR04] F. Arbab, C. Baier, F. de Boer, and J. Rutten, *Models and Temporal Logics for Timed Component Connectors*, Proceedings of the IEEE International Conference on Software Engineering and Formal Methods (SEFM '04), pp. 198-207, Beijing, China, 26-30 September 2004.

[ABRS4] F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani, *Modeling Component Connectors in Reo by Constraint Automata*, Proceedings of International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003), CONCUR 2003, Marseille, France, September 2003, Electronic Notes in Theoretical Computer Science, 97.22, Elsevier Science, July 2004.

[Arb06] F. Arbab, *A Behavioral Model for Composition of Software Components*, L'Objet, Lavoisier, vol. 12, no. 1, pp. 33-76, 2006.

[ABB00a] 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.

[ABB00b] 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.

[ABBG02] F. Arbab, F.S. de Boer, M.M. Bonsangue, and J.V. Guillen-Scholten, *MoCha: a Middleware Based on Mobile Channels*, proceedings of 26th Int. Computer Software and Application Conference (COMPSAC 02) IEEE Computer Society Press, 2002.

[ASSWW99] K. Arnold, B. O Sullivan, R.W. Scheifler, J.Waldo, and A.Wollrath, *The JiniTM Specification*, Addison-Wesley, 1999.

[Bak05] D. Bakken, *Middleware*, Chapter in Encyclopedia of Distributed Computing, J. Urban and P. Dasgupta, eds., Kluwer Academic Publishers, 2003.

[BG94] H.E. Bal, and D. Grune, *Programming language essentials*, Addison-Wesley, Reading, MA, 1994.

[BW04] F.R.M. Barnes and P.H. Welch, *Communicating Mobile Processes*, in I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, Communicating Process Architectures 2004, volume 62 of Concurrent Systems Engineering Series, pages 201-218, Amsterdam, The Netherlands, September 2004. IOS Press.

[BCCGKT05] T. Barnwell, M. Camacho, M. Cook, M. Gready, P. Kyme, and M. Tsouchlaris, *Platform Independent Petri-Net Editor 2 (PIPE)*, electronic manual, 2005. Available at `http://pipe2.sourceforge.net/` (last visited: September, 2006).

[BBBKS00] R. Bastide, D. Buchs, M. Buffo, F. Kordon, and O.Sy, *Questionnaire for a Taxonomy of Petri Net Dialects*, Online Report, May 2000. Available at http://www-src.lip6.fr/homepages/Fabrice.Kordon/PNSTD_WWW/pdf_result.pdf.

[Boc04] L. Bocchi, *Compositional Nested Long Running Transactions.* In Michel Wermelinger and Tiziana Margaria, editors, Proceedings of the 7th International Con- ference on Fundamental Approaches to Software Engineering (FASE 2004), volume 2984 of Lecture Notes in Computer Science, pages 195-208. Springer, 2004.

[BP02] F.S. de Boer, and C. Pierik, *Computer-aided specification and verification of annotated object-oriented programs*, A. Rensink and B. Jacobs, editors, proceedings of FMOODS 2002, pages 163-177, Kluwer Academic Publishers, 2002.

[BHMNCFO04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, *Web Services Architecture*, W3C Working Group Note 11, February 2004. Available at http://www.w3.org

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* , Addison-Wesley, Reading, Mass. USA, 1999.

[Bre01] F. Breg. *Java for High Performance Computing.* PhD thesis, University of Leiden, November 2001.

[BS01] 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.

[CG00] L. Cardelli and A. D. Gordon. *Mobile ambients*, Theoretical Computer Science, 240(1):177–213, June 2000.

[CG90] N. Carriero, D. Gelernter. *How to Write Parallel Programs: a First Course*, MIT press, 1990.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 2.0*, W3C, November 2001. Available at www.w3.org/TR/wsdl20/

[CCA04] D. Clarke, D. Costa, and F. Arbab, *Modeling Coordination in Biological Systems*, Proceedings of the International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), Paphos, Cyprus, 30 October - 2 November 2004.

[CCA05] D. Clarke, D. Costa, F. Arbab, *Connector Coloring I: Synchronization and Context Dependency*, 4th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2005), August 2005, San Francisco, California, USA; satellite workshop of CONCUR 2005.

[Coh03] B. Cohen, *Incentives Build Robustness in BitTorrent*, Technical Report, Bitconjurer.org, May 2003. Available at http://bitconjurer.org/BitTorrent/documentation.html

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design.* Addison-Wesley, Reading, MA, 1994.

[CGKLRTW03] F. Curbera, Y. Goland, J. Klein, F. Leyman, D. Roller, S. Thatte, and S. Weerawarana, *Business Process Execution Language for Web Services (BPEL4WS) 1.1*, May 2003. Available at `http://www.ibm.com/developerworks/library/ws-bpel/`

[DR98] J. Desel, and W. Reisig, *Place/Transition Petri Nets*, Lecture Notes in Computer Science, Vol. 1491: Lectures on Petri Nets I: Basic Models, pages 122-173. Springer-Verlag, 1998.

[DF04] N.K. Diakov, and F. Arbab, *Compositional Construction of Web Services Using Reo*, Proceedings of The Second International Workshop on Web Services (WSMAI'2004), INSTICC Press, Porto, April 2004.

[DZP05] N.K. Diakov, Z. Zlatev, and S. Pokraev, *Composition of Negotiation Protocols for E-Commerce Applications*, In the proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, William Cheung, Jane Hsu, IEEE Computer Society, pp.418-423, March 2005.

[Dijk68] E. W. Dijkstra, *Cooperating sequential processes*, In F. Genuys, editor, Programming Languages: NATO Advanced Study Institute, pages 43–112. Academic Press, 1968.

[Dijk71] E. W. Dijkstra, *Hierarchical ordering of sequential processes*, in Acta Informatica 1(2), pages 115-138, October, 1971.

[Eck98] B. Eckel, *Thinking in Java*, Prentice Hall PTR, Upper Saddle River, 1998.

[Eng91] J. Engelfriet, *Branching processes of Petri nets*, Acta Informatica Volume 28, Issue 6, pages 575 - 591, Springer-Verlag, 1991.

[Eng04] J. Engelfriet, *Private Correspondence*, LIACS, 2004.

[FPT00] G.L. Ferrari, R. Pugliese and E. Tuosto, *Foundational Calculi for Network Aware Programming*, Technical Report, Universita' di Firenze, c/o Dipartimento di Sistemi ed Informatica, 2000.

[FGMFB96] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext Transfer Protocol, HTTP/1.1*, Internet-Draft draft-ietf-http-v11-spec-07, HTTP Working Group, August 1996.

[Fok99] W. Fokkink, *Introduction to Process Algebra.* Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1999

[FHA99] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces TM Principles, Patterns, and Practice*, Chapter 1 of book, Addison-Wesley, September 1999.

[Gen87] H. J. Genrich, *Predicate/transition nets*, Advances in Petri nets 1986, part I on Petri nets: central models and their properties, pages: 207 - 247, Springer-Verlag, 1987.

[GV02] C. Girault and R. Valk, *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*, Springer-Verlag, 2002.

[Gui05] J.V. Guillen-Scholten, *MoCha, easyMoCha, chocoMoCha Electronic Manual beta version 0.96b*, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 2005.

[GABB02] 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.

[GABB05] 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.

[Gon01] L. Gong, *Project jxta: A technology overview.* Technical report, Sun Microsystems Inc., 2001.

[GM96] J. Gosling, and H. McGilton. *The Java Language Environment*, white Paper, Sun, 1996. Available at `http://java.sun.com/docs/white/langenv`

[HB03] R. Hamadi, B. Benatallah, *A Petri net-based model for web service composition*, Proceedings of the Fourteenth Australasian database conference on Database technologies 2003, Volume 17, pages: 191 - 200, Australia, 2003.

[Hay99] D. Hay, *COM+ Technical Series: Queued Components*, Microsoft Corporation, 1999.

[Hen88] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988

[HR98] M. Hennessy and J. Riely, *Resource Access Control in Systems of Mobile Agents*, HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998), U. Nestmann and B.C. Pierce, Eds. ENTCS 16.3, 1998.

[HHO04] H. He, H. Haas, and D. Orchard, *Web Services Architecture Usage Scenarios*, W3C Working Group Note 11, February 2004. Available at `http://www.w3.org`

[HBB00] G. Hilderink, J. Broenink, and A. Bakkers. *Communicating Threads for Java*, Technical report, The Netherlands, 2000. `http://www.rt.el.utwente.nl/javapp/information/CTJ/main.html`

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, London, UK, 1985.

[HC00] C.S. Horstmann, and G. Cornell, *Core Java 2; Volume II - Advanced Features*, Sun Microsystems Inc., Palo Alto, California, USA, 2000.

[Jen97a] K. Jensen, *A Brief Introduction to Coloured Petri Nets*, Tools and Algorithms for the Construction and Analysis of Systems. Proceeding of the TACAS'97 Workshop, Enschede, The Netherlands 1997, Lecture Notes in Computer Science Vol. 1217, Springer-Verlag 1997.

[Jen97b] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997. ISBN: 3-540-60943-1.

[Lak01] C.A. Lakos, *Object-Oriented Modelling with Object Petri Nets*, Lecture Notes in Computer Science 2001, Springer-Verlag, 2001.

[LO03] K. Lenz, A. Oberweis, *Inter-organizational Business Process Management with XML Nets*, Lecture Notes in Computer Science, Vol. 2472, Springer-Verlag, 2003.

[MDEK94] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*. In Proceedings of 5th European Software Engineering Conference, Spain, 1994.

[COM+] Microsoft Corporation. Home page of COM+, `http://www.microsoft.com/com/tech/complus.asp` (last visited: September, 2006).

[Mil99] R. Milner, *Communicating and Mobile Systems : The Pi-Calculus*, Cambridge University Press, May 20, 1999.

[Mil80] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol 92, Springer-Verlag 1980.

[MPR03] A.L. Murphy, G.P. Picco, and G.-C. Roman, *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.

[NFP98] R. De Nicola, G.L. Ferrari, and R. Pugliese, *KLAIM: A kernel language for agents interaction and mobility*, IEEE Transactions on Software Engineering, 24(5), pages 315-330, 1998.

[OAS04] OASIS group, *Universal Description, Discovery and Integration (UDDI) protocol 3.0*, November 2004. Available at `http://www.uddi.org/`

[Par01] J. Parrow, *An Introduction to the pi-Calculus*. In Handbook of Process Algebra, ed. Bergstra, Ponse, Smolka, pages 479-543, Elsevier 2001.

[Pet96] C.A. Petri, *Nets, Time and Space*. Theoretical Computer Science, 153:348, 1996.

[PNW05] Petri Nets World, *Petri Nets Tool Database*, online Database. Available at http://www.informatik.uni-hamburg.de/TGI/PetriNets/ (last visited: September, 2006).

[Pic98] G.P. Picco, *μCODE: A Lightweight and Flexible Mobile Code Toolkit*. In Proc. of the 2ed Int.Workshop on Mobile Agents, LNCS 1477. Springer, 1998.

[Pic05] G.P. Picco, *LighTS Web page*, 2005, http://lights.sourceforge.net (last visited: September, 2006).

[PT97] B. C. Pierce, D. N. Turner, *Pict: A Programming Language Based on the Pi-calculus*, Technical report, Computer Science Department, Indiana University, 1997. Available at http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html

[JCSP.net] Quickstone Technologies, *The JCSP Network Edition Web Page*, http://www.quickstone.com/ (last visited: September, 2006).

[RFHKS01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, *A Scalable Content-Addressable Network*, ACM SIGCOMM '01, San Diego, 2001.

[Rei98] W. Reisig, *Elements of Distributed Algorithms: Modeling and Analysis with Petri nets*, Springer-Verlag, 1998.

[RR98] W. Reisig. G. Rozenberg (Eds.), *Lectures on Petri Nets I: Basic Models*, Advances in Petri Nets, Lecture Notes in Computer Science, vol. 1491, Springer-Verlag, 1998.

[Rip01] M. Ripeanu, *Peer-to-Peer Architecture Case Study: Gnutella Network*, Technical Report, University of Chicago, 2001.

[Roz86] G. Rozenberg, *Behaviour of Elementary Net Systems*, Lecture Notes In Computer Science, Vol. 254, pages 60-94, Springer-Verlag, 1986

[RE98] G. Rozenberg and J. Engelfriet, *Elementary Net Systems*, Lecture Notes in Computer Science, v. 1491, Springer-Verlag, 12-121, 1998.

[RD01] A. Rowstron, P. Druschel, *Pastry: Scalable, Decentralized Object Location and Routing for LargeScale Peer-to-Peer Systems*, 18 Conference on Distributed Systems Platforms, Heidelberg (D), 2001.

[RSRS99] 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.

[SW01] D. Sangiorgi, and D. Walker, *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

[SS01] R. Schantz and D. Schmidt, *Middleware for Distributed Systems: Evolving the Common Structure for Networkcentric Applications*, The Encyclopedia of Software Engineering, J. Wiley & Sons, pp. 801-813, December 2001.

[Sch01] R. Schollmeier, *A Definition of Peer-to-Peer Networking towards a Delimitation Against Classical Client-Server Concepts*, Proceedings of EUNICE-WATM, pp. 131-138, Paris, France, September 3-5, 2001.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward, *Real-Time Object-Oriented Modelling*, John Wiley and Sons, Inc., 1994.

[Sha03] Sharman Networks, *Kazaa, Detailed On-line Guide*, Online Manual, 2003. Available at http://www.kazaa.com/us/help/guide.htm (last visited: September, 2006).

[Shi01] C. Shirky, *Listening to Napster*, in: Oram, A (ed.). Peer-to-Peer: Harnessing the Power of Disruptive Technologies, Sebastopol, O'Reilly, 2001.

[Ste95] W. R. Stevens, *TCP/IP Illustrated, Volume 1; The Protocols*, Addison Wesley, Boston, Massachusetts, 1995.

[SMKKB01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Diego, CA, August 2001, pp. 149-160.

[Sto02] P. van der Stok, *In-home middleware standards and interoperability*, Slides from the IPA Spring Days on Middleware, Heeze, April 2002.

[Sto98] H. Störrle, *An Evaluation of High-End Tools for Petri-Nets*, Technical Report No. 9802, University of Munich, June, 1998.

[Str91] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

[Java] Sun Microsystems Inc., *Home Page of Java*, http://java.sun.com (last visited: September, 2006).

[JB] Sun Microsystems Inc., *Home Page of JavaBeans*, http://java.sun.com/products/javabeans (last visited: September, 2006).

[EJB] Sun Microsystems Inc., *Home Page of Enterprise JavaBeans*, http://java.sun.com/products/ejb (last visited: September, 2006).

[Jini] Sun Microsystems Inc., *Home Page of Jini*, http://java.sun.com/products/jini (last visited: September, 2006).

[RMI] Sun Microsystems Inc., *Java Remote Method Invocation - Distributed Computing for Java*, white paper available at java.sun.com/rmi, 2004.

[RPC] Sun Microsystems Inc., *RPC: Remote procedure call protocol specification.* Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

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

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

[TS02] A.S. Tanenbaum, and M. van Steen, *Distributed Systems — Principles and Paradigms*, Prentice-Hall, Englewood Cliffs, NJ, U.S.A., 2002.

[TB00] Z. Tari and O. Bukhres, *Fundamentals of Distributed Object Systems: The CORBA Perspective*. John Wiley, 2000.

[VM94] B. Victor, F. Moller, *The Mobility Workbench - A Tool for the pi-Calculus*, Proceedings of the 6th International Conference on Computer Aided Verification, Lecture Notes In Computer Science Vol. 818, pages 428 - 440, Springer-Verlag, London, 1994.

[VW02] B.Vinter, and P.H.Welch, *Cluster Computing and JCSP Networking*, in Communicating Process Architectures 2002, vol. 60 Concurrent Systems Engineering, pp. 203-222, IOS Press, Amsterdam, September 2002.

[XML00] W3C, *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C recommendation, October 2000. Available at `http://www.w3c.org/XML/`

[W3C02] W3C, *Web Service Choreography Interface (WSCI) 1.0*, W3C Note, August 2002. Available at `http://www.w3.org/TR/wsci/`

[Wel01] 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/` (last visited: September, 2006).

[WMBW00] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood, *The KRoC Home Page*, 2000. Available at `http://www.cs.ukc.ac.uk/projects/ofa/kroc/` (last visited: September, 2006).

[WS99] P. Wojciechowski, and P. Sewell, *Nomadic Pict: Language and Infrastructure Design for Mobile Agents*, First International Symposium on Agent Systems and Applications (ASA'99)/(MA'99), Palm Springs, CA, USA, 1999.

[XPG03] XML Protocol Group, *SOAP 1.2*, W3C Recommendation, June 2003. Available at `http://www.w3c.org/2000/xp/Group/`

[YK98] A.V. Yakovlev, A.M. Koelmans, *Petri Nets and Digital Hardware Design*, Lecture Notes in Computer Science, Vol. 1492, Springer-Verlag, 1998.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, *Tapestry: An infrastructure for fault-resilient wide-area location and routing.* Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Appendix A

MoCha's Abstract Algorithms

In the MoCha middleware (see Chapters 6 and 7) we implemented eleven channel types. Before actually implementing them in the Java programming language, we first specified implementation language independent abstract algorithms for each channel type. In this appendix, we list the abstract algorithms of five representative channel types: *Synchronous*, *lossy synchronous*, *FIFO*, *synchronous drain*, and *asynchronous spout*. The abstract implementation of the remaining channels can easily be derived from the ones we give.

For each channel type we first give the data-structures that its algorithms use. We then give a set of abstract functions that describes the implementation. For each channel type we give the following major functions: *Create_Channel*, *Write*, *Take*, *Read*, *Move*, and *Destroy_Channel*. Naturally, channel types with two source-ends do not implement the take and read functions, and channel-types with two sink-ends do not implement the write function. Depending on the channel type, additional internal functions may be needed.

Distribution, Variables, and References

The mobile channels of MoCha are meant to be implemented in distributed systems. For efficiency reasons, we want this implementation to have a *decentralized* architecture (see Chapter 7). This means that, the algorithms are not allowed to use *shared data variables*. The use of globally accessible variables either imposes a centralized architecture, or the necessity for implementing an expensive protocol for automatic global variable update. Instead, our algorithms work with local variables. Therefore, all the necessary values have to be given as function parameters.

The references used by MoCha either point to a local or to a remote entity. A reference has a structure given in Figure A.1. It is a pointer with a flag that indicates whether it is local (L) or remote (R), so that the algorithms can test on this flag. The address-part of a reference not only specifies the memory-address of the entity but also its location. For the definition of location, it is sufficient to say that it is a *logical address space* where components and other entities run. The mapping of

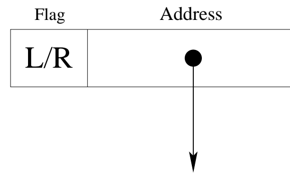


Figure A.1: A reference.

locations to physical implementations is irrelevant at this level of abstraction

Data-structures and References

The algorithms use a → notation to express that something is a reference to a particular data-structure type. "→data-structure" is the set of all references to this data-structure type. For example, in "Next_rf ∈ →Buffer", Next_rf is a reference to a Buffer data-structure type. The reference does not need to be valid, i.e. refer to an existing instance of the data-structure.

We use another notation for expressing the operation that takes a reference and gives the instance of the data-structure that it refers to. For this operation we use the symbol ↑. For example, "Next_rf↑.Vals := ⟨ ⟩", the Vals-field of the Buffer structure that Next_rf refers to, is set to empty. The algorithms have to make sure that the reference is valid before executing this operation in the first place.

Synchronization

Synchronization between the various processes that are executing the functions of a particular channel is achieved by using *binary semaphores*. These semaphores have two operations *Lock* and *Unlock*. Initially the semaphores are unlocked, unless otherwise specified. We assume that the waiting-queue where the blocked processes are waiting is a FIFO-queue, this is not necessary for a correct execution but it is assumed for fairness of the system.

A typical example of a semaphore we frequently use is the *CE_Lock*. This is the general channel-end lock that provides exclusive access to at most one process at a time.

Mapping to a Specific Programming Language

The abstraction level of the algorithms we give for each channel type is such that it is easy to implement them in the most common programming languages like C, C++, and Java. However, while mapping the algorithms to a specific language, some language dependent issues may arise. For example, when we implemented the algorithms in the Java language we had to translate the abstract functions in an object oriented setting. Also, we had to provide extra synchronization for accessing some variables (synchronization which was not needed in the abstract algorithms). Naturally, depending on the programming language other issues may arise.

A.1 Synchronous Channel

Data-structures

Source_Channel_End = ⟨Sink_rf, CE_Lock, Move_Lock, Wait_Lock, Write_Lock, Wait_Take, Sink_Wait_Read, Sink_Wait_Take, Current_Value⟩

- Sink_rf = A reference to the Sink_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Wait_Lock = A binary semaphore to wait when there is a write but the Sink is not ready to take a value.
- Write_Lock = A binary semaphore that is needed to give exclusive access to the booleans of the Source.
- Wait_Take = A boolean used to express that the Source is waiting until a take is performed by the Sink.
- Sink_Wait_Read = A boolean used to express that the Sink_Channel_End is waiting for value to read.
- Sink_Wait_Take = A boolean used to express that the Sink_Channel_End is waiting for a value to take.
- Old_Sink_Wait_Take = A boolean to remember the previous value of Sink_Wait_Take.
- Current_Value = A value that is stored for situations when the Source is locked under the Wait_Lock and the Sink request a value to read instead of to take.

Sink_Channel_End = ⟨Source_rf, CE_Lock, Move_Lock, Wait_Lock, Value_Read, Current_Value⟩

- Source_rf = A reference to the Source_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Wait_Lock = A binary semaphore used to lock when waiting for a value to be available at the Source_Channel_End, initially locked.
- Value_Read = A boolean used to express that the Sink already has read a value from the Source_Channel_End.
- Current_Value = A value that is stored for situations when it is acquired due to a read operation. Gets empty when a take operation is performed.

Initially the semaphores are unlocked, except for Wait_Lock and Value_Lock.

Functions, an Overview

- **Create_Channel** (*Loc* ∈ Location)
  returns ⟨*Source_rf* ∈ →Source_Channel_End, *Sink_rf* ∈ →Sink_Channel_End, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Write** (*Source_rf* ∈ →Source_Channel_End, *X* ∈ Value)
  returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Take** (*Sink_rf* ∈ →Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Read** (*Sink_rf* ∈ →Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Move** (*Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End}, *Target* ∈ Location)
  returns ⟨*New_Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End}, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Destroy_Channel** (*Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End})
  returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩

Algorithms

```
Create_Channel (Loc)
//
// Loc = The location where to create the Source_Channel_End and the Sink_Channel_End.
//
// returns <Source_rf ∈ →Source_Channel_End,
// Sink_rf ∈ →Sink_Channel_End,
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   Source_rf := new Source_Channel_End @ Loc;
   if ( Source_rf = ERROR ) then
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Sink_rf := new Sink_Channel_End @ Loc;
   if ( Sink_rf = ERROR ) then
      Delete(Source_rf);
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Source_rf↑.Wait_Take := FALSE;
   Source_rf↑.Sink_Wait_Read := FALSE;
   Source_rf↑.Sink_Wait_Take := FALSE;
   Sink_rf↑.Value_Read := FALSE;
   // Knowing each other.
   //
   Source_rf↑.Sink_rf := Sink_rf;
   Sink_rf↑.Source_rf := Source_rf;

   return <Source_rf, Sink_rf, SUCCESS>;
end
```

In case that the operation **new** does not succeed it will give an error and exit with or without throwing an exception (depends whether the target programming language can handle exceptions). The operation **new** returns a pointer with a local flag (reference).

```
Write (Source_rf, X)
//
// Source_rf ∈ →Source_Channel_End, X ∈ Value.
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Source_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;
   Lock(Source_rf↑.Write_Lock);

   Source_rf↑.Old_Sink_Wait_Take := Source_rf↑.Sink_Wait_Take;
   if ( Source_rf↑.Sink_Wait_Read ∨ Source_rf↑.Sink_Wait_Take ) then
      // Sink is waiting for a value due to a read or a take operation.
      // No need to lock the Sink since it is waiting.
      //
      Source_rf↑.Sink_rf↑.Current_Value := X;
      if (Source_rf↑.Sink_Wait_Take) then {Source_rf↑.Sink_Wait_Take = FALSE;}
      Unlock(Source_rf↑.Sink_rf↑.Wait_Lock);
   fi
   if ( ¬Source_rf↑.Old_Sink_Wait_Take ) then
      // There was no take operation performed on the Sink, Source must wait.
      //
      Source_rf↑.Current_Value := X;
      Source_rf↑.Wait_Take := TRUE;
      // Allow the Sink to enter the Source for reading/taking the value
      Unlock(Source_rf↑.Write_Lock);
      // Wait_Lock is initially locked!
      Lock(Source_rf↑.Wait_Lock);
      Lock(Source_rf↑.Write_Lock);
   fi

   // At this point Sink and Source have finished the write/take operation.
```

```
   Source_rf↑.Sink_Wait_Read := FALSE;

   Unlock(Source_rf↑.Write_Lock);
   Unlock(Source_rf↑.CE_Lock);
   return SUCCESS;
end


Take (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   if ( Value_Read ) then
      // Sink has read a value before, Current_Value is the right value.
      // unlock Source that is waiting.
      //
      Sink_rf↑.Source_rf↑.Wait_Take := FALSE;
      Unlock( Sink_rf↑.Source_rf↑.Wait_Lock);
      Sink_rf↑.Value_Read := FALSE;
   else
      // We need to consult the Source.
      // Prevent the Source from moving by locking own Move_Lock.
      Lock(Sink_rf↑.Move_Lock);
      // No need to check for Error, reference to Source should always be valid now.
      Lock(Sink_rf↑.Source_rf↑.Write_Lock);
      if ( Sink_rf↑.Source_rf↑.Wait_Take) then
         // The Source is already waiting for the Sink to take.
         Sink_rf↑.Current_Value := Sink_rf↑.Source_rf↑.Current_Value;
         Sink_rf↑.Source_rf↑.Wait_Take := FALSE;
         Unlock( Sink_rf↑.Source_rf↑.Wait_Lock);
         Unlock(Sink_rf↑.Source_rf↑.Write_Lock);
         Unlock(Sink_rf↑.Move_Lock);
      else
         // Tell the Source to inform us when a value is available.
         Sink_rf↑.Source_rf↑.Sink_Wait_Take := TRUE;
         // now Sink must wait.
         Unlock(Sink_rf↑.Source_rf↑.Write_Lock);
         Unlock(Sink_rf↑.Move_Lock);
         // Wait_Lock is initially locked!
         Lock(Sink_rf↑.Wait_Lock);
      fi
   fi
   Unlock(Sink_rf↑.CE_Lock);
   // At this point Current_Value is always filled with the right value.
   return <Sink_rf↑.Current_Value, SUCCESS>;
end


Read (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   if ( ¬Value_Read ) then
      // If Sink has read a value before, Current_Value is the right value.
      // This is not the case now, so we need to consult the Source.
      // Prevent the Source from moving by locking own Move_Lock.
      Lock(Sink_rf↑.Move_Lock);
      // No need to check for Error, reference to Source should always be valid now.
      Lock(Sink_rf↑.Source_rf↑.Write_Lock);
      if ( Sink_rf↑.Source_rf↑.Wait_Take) then
```

```
            // The Source is already waiting for the Sink to take.
            // Copy the value but do not unlock the Source.
            Sink_rf↑.Current_Value := Sink_rf↑.Source_rf↑.Current_Value;
            Unlock(Sink_rf↑.Source_rf↑.Write_Lock);
            Unlock(Sink_rf↑.Move_Lock);
        else
            // Tell the Source to inform us when a value is available.
            Sink_rf↑.Source_rf↑.Sink_Wait_Read := TRUE;
            // now Sink must wait.
            Unlock(Sink_rf↑.Source_rf↑.Write_Lock);
            Unlock(Sink_rf↑.Move_Lock);
            // Wait_Lock is initially locked!
            Lock(Sink_rf↑.Wait_Lock);
        fi
        Sink_rf↑.Value_Read := TRUE;
    fi
    Unlock(Sink_rf↑.CE_Lock);
    // At this point Current_Value is always filled with the right value.
    return <Sink_rf↑.Current_Value, SUCCESS>;
end

Move (Channel_End_rf, Target)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// Target = The location target.
//
// returns <New_Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End},
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
    ERROR := Lock(Channel_End_rf↑.CE_Lock);
    if ( ERROR ) then return <UNDEFINED,FAILURE>;

    if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
        // Lock Sink first, then Source.
        // There is no Unlock for the following Lock.
        //
        Lock(Channel_End_rf↑.Move_Lock);
        Lock(Channel_End_rf↑.Source_rf↑.Move_Lock);

        New_Channel_End_rf := new Sink_Channel_End @ Target;
        if ( New_Channel_End_rf = ERROR ) then
            Unlock(Channel_End_rf↑.Source_rf↑.Move_Lock);
            Unlock(Channel_End_rf↑.Move_Lock);
            Unlock(Channel_End_rf↑.CE_Lock);
            return <UNDEFINED, FAILURE>;
        fi

        Copy_Information(Channel_End_rf, New_Channel_End_rf);
        Channel_End_rf↑.Source_rf↑.Sink_rf := New_Channel_End_rf;
        Delete(Channel_End_rf);

        Unlock(New_Channel_End_rf↑.Source_rf↑.Move_Lock);

    else

        // Lock Sink first, then Source.
        //
        do
            ERROR := Lock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
        until ( ¬ERROR )

        // There is no Unlock for the following Lock.
        //
        Lock(Channel_End_rf↑.Move_Lock);
        New_Channel_End_rf := new Source_Channel_End @ Target;
        if ( New_Channel_End_rf = ERROR ) then
            Unlock(Channel_End_rf↑.Move_Lock);
            Unlock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
            Unlock(Channel_End_rf↑.CE_Lock);
```

```
            return <UNDEFINED, FAILURE>;
        fi
        Copy_Information(Channel_End_rf, New_Channel_End_rf);
        Channel_End_rf↑.Sink_rf↑.Source := New_Channel_End_rf;
        Delete(Channel_End_rf);

        Unlock(New_Channel_End_rf↑.Sink_rf↑.Move_Lock);
    fi
    return <New_Channel_End_rf, SUCCESS>;
end
```

We Always Lock the sink first. Locking the source first gives problems with the function *Read* while setting the field Wait_Read of the Source_Channel_End.

```
Destroy_Channel (Channel_End_rf)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
// There is no Unlock in this function
//
begin

    if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
        Sink_rf := Channel_End_rf;
    else
        // Check for dangling reference
        //
        ERROR := Lock(Channel_End_rf↑.CE_Lock);
        if ( ERROR ) then return FAILURE;
        Sink_rf := Channel_End_rf↑.Sink_rf;
        Unlock(Channel_End_rf↑.CE_Lock);
    fi

    // At this point Sink_rf ∈ Sink_Channel_End,
    // but needs not be valid anymore due to possible
    // movement of sink channel-end.

    // Lock Sink first, then Source.
    //
    ERROR := Lock(Sink_rf↑.CE_Lock);
    if ( ERROR ) then return FAILURE;

    do
        ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
    until ( ¬ERROR )

    Source_rf := Sink_rf↑.Source_rf;

    Delete(Source_rf);
    Delete(Sink_rf);
    return SUCCESS;
end
```

A.2 Lossy Synchronous Channel

Data-structures

The lossy synchronous channel uses the same data-structures as the synchronous channel type.

Functions, an Overview

The lossy synchronous channel has the same functions as the synchronous channel type.

Algorithms

The algorithms of this channel type are exactly the same as the ones of the synchronous channel type, except for the **Write** function which allows a write operation to succeed in cases when the sink channel-end is not waiting to take.

```
Write (Source_rf, X)
//
// Source_rf ∈ →Source_Channel_End, X ∈ Value.
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Source_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;
   Lock(Source_rf↑.Write_Lock);

   if ( Source_rf↑.Sink_Wait_Read ∨ Source_rf↑.Sink_Wait_Take ) then
       // Sink is waiting for a value due to a read or a take operation.
       // No need to lock the Sink since it is waiting.
       //
       Source_rf↑.Sink_rf↑.Current_Value := X;
       if (Source_rf↑.Sink_Wait_Take) then {Source_rf↑.Sink_Wait_Take = FALSE;}
       Unlock(Source_rf↑.Sink_rf↑.Wait_Lock);
       if ( Source_rf↑.Sink_Wait_Read ) then
           // There was a read operation performed on the Sink,
           // Source must wait now for a Take.
           //
           Source_rf↑.Current_Value := X;
           Source_rf↑.Wait_Take := TRUE;
           // Allow the Sink to enter the Source for taking the value
           Unlock(Source_rf↑.Write_Lock);
           // Wait_Lock is initially locked!
           Lock(Source_rf↑.Wait_Lock);
           Lock(Source_rf↑.Write_Lock);
       fi
   fi

   // At this point Sink and Source have either finished the write/take
   // operation or there has been a single write only.
   Source_rf↑.Sink_Wait_Read := FALSE;

   Unlock(Source_rf↑.Write_Lock);
   Unlock(Source_rf↑.CE_Lock);
   return SUCCESS;
end
```

A.3 Asynchronous FIFO Channel

Data-structures

Source_Channel_End = ⟨Buffer_rf, Sink_rf, CE_Lock, Move_Lock, Wait_Read⟩

- Buffer_rf = A reference to a Buffer-structure.
- Sink_rf = A reference to the Sink_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Wait_Read = A boolean used to express that the Sink_Channel_End is waiting for values.

Sink_Channel_End = ⟨Buffer_rf, Source_rf, Consumed, CE_Lock, Move_Lock, Read_Lock⟩

- Buffer_rf = A reference to a Buffer-structure.
- Source_rf = A reference to the Source_Channel_End of the same channel.
- Consumed = An integer that keeps track of the number of values consumed by the component(s) since the last move.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Read_Lock = A binary semaphore used to lock when waiting for values to be written into the channel (in case it was empty), initially locked.

Buffer = ⟨Vals, Link_rf, Buffer_Lock⟩

- Vals is a sequence of Value.
- Link_rf = A reference to a Buffer-structure.
- Buffer_Lock = A binary semaphore.

Initially all the semaphores are unlocked, except for the Read_Lock.

Functions, an Overview

- **Create_Channel** (*Loc1* ∈ Location, *Loc2* ∈ Location)
  returns ⟨*Source_rf* ∈ →Source_Channel_End, *Sink_rf* ∈ →Sink_Channel_End, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Write** (*Source_rf* ∈ →Source_Channel_End, *X* ∈ Value)
  returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Take** (*Sink_rf* ∈ →Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Read** (*Sink_rf* ∈ →Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Move** (*Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End}, *Target* ∈ Location)
  returns ⟨*New_Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End}, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Destroy_Channel** (*Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End})
  returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩

These five functions use the following functions that operate on the buffer data-structure:

- **Write** (*Buffer_rf* ∈ →Buffer, *X* ∈ Value)
- **Sink_Read** (*Buffer_rf* ∈ →Buffer, *ToBeAsked* ∈ Integer, Destructive_Read ∈ Boolean)
  returns ⟨*X* ∈ Value, *Success* ∈ $\{HAPPY, SAD\}$⟩
- **Buffer_Read** (*Buffer_rf* ∈ →Buffer, *Amount* ∈ Integer)
  returns ⟨*V* ∈ $Value^*$, *Next_Buffer_rf* ∈ →Buffer⟩

Algorithms

```
Create_Channel (Loc1 , Loc2)
//
// Loc1 = The location where to create the Source_Channel_End.
// Loc2 = The location where to create the Sink_Channel_End.
//
// returns <Source_rf ∈ →Source_Channel_End,
// Sink_rf ∈ →Sink_Channel_End,
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   Source_rf := new Source_Channel_End @ Loc1;
   if ( Source_rf = ERROR ) then
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Sink_rf := new Sink_Channel_End @ Loc2;
   if ( Sink_rf = ERROR ) then
      Delete(Source_rf);
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Source_rf↑.Buffer_rf := NULL;
   Source_rf↑.Wait_Read := FALSE;
   Sink_rf↑.Buffer_rf := NULL;
   Sink_rf↑.Consumed := 0;

   // Knowing each other.
   //
   Source_rf↑.Sink_rf := Sink_rf;
   Sink_rf↑.Source_rf := Source_rf;

   return <Source_rf, Sink_rf, SUCCESS>;
end

Write (Source_rf, X)
//
// Source_rf ∈ →Source_Channel_End, X ∈ Value.
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Source_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   if ( Source_rf↑.Buffer_rf = NULL ) then
      // Create first buffer of channel.
      //
      Source_rf↑.Buffer_rf := new Buffer;
      Source_rf↑.Buffer_rf↑.Link_rf := NULL;

      // Sink can not change its buffer field while it is NULL and
      // by locking the move of the source the sink can not move either.
      //
      Lock(Source_rf↑.Move_Lock);
      // Tell the other side.
      //
      Source_rf↑.Sink_rf↑.Buffer_rf := Source_rf↑.Buffer_rf;
      Unlock(Source_rf↑.Move_Lock);
   else
      if ( ¬Local(Source_rf↑.Buffer_rf) ) then
         // Create new local buffer
         //
         TempBuffer_rf := new Buffer;
         TempBuffer_rf↑.Link_rf := NULL;
         Source_rf↑.Buffer_rf↑.Link_rf := TempBuffer_rf;
         Source_rf↑.Buffer_rf := TempBuffer_rf;
      fi
   fi
```

```
   // At this point there is always a local buffer.
   //
   Write(Source_rf↑.Buffer_rf, X);
   if ( Source_rf↑.Wait_Read ) then
      // Sink is waiting for values
      //
      Unlock(Source_rf↑.Sink_rf↑.Read_Lock);
      Source_rf↑.Wait_Read := FALSE;
   fi

   Unlock(Source_rf↑.CE_Lock);
   return SUCCESS;
end


Write (Buffer_rf, X)
//
// Buffer_rf ∈ →Buffer, X ∈ Value.
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   Buffer_rf↑.Vals := Buffer_rf↑.Vals ∘ X;

   Unlock(Buffer_rf↑.Buffer_Lock);
end


Take (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   Success ∈ {HAPPY, SAD} := SAD;

   while ( Success ≠ HAPPY ) do
      if ( Sink_rf↑.Buffer_rf ≠ NULL ) then
         if ( ¬Local(Sink_rf↑.Buffer_rf) ) then
            // New local Buffer.
            //
            TempBuffer_rf := new Buffer;
            TempBuffer_rf↑.Link_rf := Sink_rf↑.Buffer_rf;
            Sink_rf↑.Buffer_rf := TempBuffer_rf;
         fi
         // At this point there is always a local buffer.
         //
         // Success indicates whether there is an element to read in the channel.
         //
         <X, Success> := Sink_Read(Sink_rf↑.Buffer_rf, Sink_rf↑.Consumed + 1, TRUE);
      fi

      // In case that there is no element in the channel "consult" the source
      //
      if ( Success ≠ HAPPY ) then
         do
            ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
         until (¬ERROR)
         if ( Sink_rf↑.Source_rf↑.Buffer_rf ≠ NULL ) ∧
            |Sink_rf↑.Source_rf↑.Buffer_rf↑.Vals| ≠ 0 ) then
            Unlock(Sink_rf↑.Source_rf↑.CE_Lock);
         else
            Sink_rf↑.Source_rf↑.Wait_Read := TRUE;
            Unlock(Sink_rf↑.Source_rf↑.CE_Lock);
            // Read_Lock is initially locked!
            //
```

```
            Lock(Sink_rf↑.Read_Lock);
         fi
      fi
   done

   Sink_rf↑.Consumed := Sink_rf↑.Consumed + 1;
   Unlock(Sink_rf↑.CE_Lock);
   return <X, SUCCESS>;
end
```

The while-loop above executes no more than 2 times under the assumption that the values inserted by the source-end never get deleted other than by taking elements from the buffers.

```
Read (Sink_rf)
//
// Sink_rf ∈ →Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
The code of this function is the same as the one for the take function with the
difference that the Sink_Read function is now called with Destructive_Read := FALSE.

   <X, Success> := Sink_Read(Sink_rf↑.Buffer_rf, Sink_rf↑.Consumed + 1, FALSE);

end


Sink_Read (Buffer_rf, ToBeAsked, Destructive_Read)
//
// Buffer_rf ∈ →Buffer
//
// ToBeAsked ∈ Integer, number of values to be asked
// to the next buffer in case Buffer_rf↑ is empty.
//
// Destructive_Read ∈ Boolean, is TRUE when this function is
// called due to a take operation.  In that case the element to be
// returned has to be removed from the buffer.
//
// returns <X ∈ Value, Success ∈ {HAPPY, SAD} >
//
// This function operates on the Buffer data-structure.
//
begin
    Lock(Buffer_rf↑.Buffer_Lock);

    V := < >; // a sequence of Value.
    Reference := Buffer_rf↑.Link_rf;

    // Execute loop while buffer is empty and
    // there is (still) a reference to another buffer.
    //
    while ( (|Buffer_rf↑.Vals| = 0) ∧ (Reference ≠ NULL) ) do
        // Buffer_Read returns, besides V, a reference to a
        // next buffer in the chain (if any exists) or NULL (if not).
        //
        <V, Reference> := Buffer_Read(Buffer_rf↑.Link_rf, ToBeAsked);
        if ( Reference ≠ NULL ) then
            Buffer_rf↑.Link_rf := Reference;
        fi
        Buffer_rf↑.Vals := V;
    done

    if (|Buffer_rf↑.Vals| ⟩ 0 ) then
        X := Head(Buffer_rf↑.Vals);
        if (Destructive_Read) then
            Buffer_rf↑.Vals := Tail(Buffer_rf↑.Vals);
        fi
        Success := HAPPY;
    else
```

```
        // No elements found in channel.
        //
        X := UNDEFINED;
        Success := SAD;
    fi

    Unlock(Buffer_rf↑.Buffer_Lock);

    return <X, Success>;
end
```

The while-loop executes no more than 3 times under the assumption that the values inserted by the source never get deleted other than by taking elements from the buffers.

```
Buffer_Read (Buffer_rf, Amount)
//
// Buffer_rf ∈ →Buffer
//
// Amount ∈ Integer, number of values requested.
//
// returns <V ∈ Value*, Next_Buffer_rf ∈ →Buffer>
//
// This function operates on the Buffer data-structure.
//
begin
   Lock(Buffer_rf↑.Buffer_Lock);

   V := < >;
   Destroy := FALSE;

   if ( |Buffer_rf↑.Vals| ≤ Amount ) then
      V := Buffer_rf↑.Vals;
      Buffer_rf↑.Vals := < >;
      Next_Buffer_rf := Buffer_rf↑.Link_rf;
      if ( Buffer_rf↑.Link_rf ≠ NULL ) then Destroy := TRUE; fi
   else
      for 1 to Amount do
         V := V o Head(Buffer_rf↑.Vals);
         Buffer_rf↑.Vals := Tail(Buffer_rf↑.Vals);
      od
      Next_Buffer_rf := NULL;
   fi

   // Unlocking before destroying would make
   // the algorithm unstable.
   //
   if ( Destroy ) then
      Delete(Buffer_rf);
   else
      Unlock(Buffer_rf↑.Buffer_Lock);
   fi

   return <V, Next_Buffer_rf>;
end
```

```
Move (Channel_End_rf, Target)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// Target = The location target.
//
// returns <New_Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End},
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   ERROR := Lock(Channel_End_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
      // Lock Sink first, then Source.
```

```
      // There is no Unlock for the following Lock.
      //
      Lock(Channel_End_rf↑.Move_Lock);
      Lock(Channel_End_rf↑.Source_rf↑.Move_Lock);

      New_Channel_End_rf := new Sink_Channel_End @ Target;
      if ( New_Channel_End_rf = ERROR ) then
         Unlock(Channel_End_rf↑.Source_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.CE_Lock);
         return <UNDEFINED, FAILURE>;
      fi

      Copy_Information(Channel_End_rf, New_Channel_End_rf);
      New_Channel_End_rf↑.Consumed := 0;
      Channel_End_rf↑.Source_rf↑.Sink_rf := New_Channel_End_rf;
      Delete(Channel_End_rf);

      Unlock(New_Channel_End_rf↑.Source_rf↑.Move_Lock);

   else

      // Lock Sink first, then Source.
      //
      do
         ERROR := Lock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
      until ( ¬ERROR )

      // There is no Unlock for the following Lock.
      //
      Lock(Channel_End_rf↑.Move_Lock);
      New_Channel_End_rf := new Source_Channel_End @ Target;
      if ( New_Channel_End_rf = ERROR ) then
         Unlock(Channel_End_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.Sink_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.CE_Lock);
         return <UNDEFINED, FAILURE>;
      fi
      Copy_Information(Channel_End_rf, New_Channel_End_rf);
      Channel_End_rf↑.Sink_rf↑.Source := New_Channel_End_rf;
      Delete(Channel_End_rf);

      Unlock(New_Channel_End_rf↑.Sink_rf↑.Move_Lock);
   fi
   return <New_Channel_End_rf, SUCCESS>;
end


Destroy_Channel (Channel_End_rf)
//
// Channel_End_rf ∈ {→Source_Channel_End, →Sink_Channel_End}
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
// There is no Unlock in this function
//
begin

   if ( Channel_End_rf↑ ∈ Sink_Channel_End ) then
      Sink_rf := Channel_End_rf;
   else
      // Check for dangling reference
      //
      ERROR := Lock(Channel_End_rf↑.CE_Lock);
      if ( ERROR ) then return FAILURE;
      Sink_rf := Channel_End_rf↑.Sink_rf;
      Unlock(Channel_End_rf↑.CE_Lock);
   fi

   // At this point Sink_rf ∈ Sink_Channel_End,
   // but needs not be valid anymore due to possible
```

```
   // movement of sink channel-end.

   // Lock Sink first, then Source.
   //
   ERROR := Lock(Sink_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   do
      ERROR := Lock(Sink_rf↑.Source_rf↑.CE_Lock);
   until ( ¬ERROR )

   Source_rf := Sink_rf↑.Source_rf;

   // Delete chain of buffers.
   //
   i := Sink_rf↑.Buffer_rf;
   While ( i ≠ NULL ) do
      j := i↑.Link_ref;
      Delete(i);
      i := j;
   done

   Delete(Source_rf);
   Delete(Sink_rf);
   return SUCCESS;
end
```

A.4 Synchronous Drain Channel

Data-structures

Passive_Source_Channel_End = ⟨Active_rf, CE_Lock, Move_Lock, Wait_Lock, Write_Lock, Passive_Wait, Active_Wait⟩

- Active_rf = A reference to the Active_Source_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Wait_Lock = A binary semaphore to wait when there is a write but the Active_Source is not ready to write on its side.
- Write_Lock = A binary semaphore that is needed to give exclusive access to the booleans of the Passive_Source.
- Passive_Wait = A boolean used to express that the Passive_Source is waiting until a write is performed on the Active_Source.
- Active_Wait = A boolean used to express that the Active_Source is waiting until a write is performed on the Passive_Source.

Active_Source_Channel_End = ⟨Passive_rf, CE_Lock, Move_Lock, Wait_Lock⟩

- Passive_rf = A reference to the Passive_Source_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Wait_Lock = A binary semaphore used to lock when waiting for a write operation to be performed at the Passive_Source_Channel_End, initially locked.

Initially all the semaphores are unlocked, except for Wait_Lock.

Functions, an Overview

- **Create_Channel** (*Loc* ∈ Location)
returns ⟨*Passive_rf* ∈ →Passive_Source_Channel_End, *Active_rf* ∈ →Active_Source_Channel_End, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Write** (*Passive_rf* ∈ →Passive_Source_Channel_End, *X* ∈ Value)
returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Write** (*Active_rf* ∈ →Active_Source_Channel_End, *X* ∈ Value)
returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Move** (*Channel_End_rf* ∈ {→Passive_Source_Channel_End, →Active_Source_Channel_End}, *Target* ∈ Location)
returns ⟨*New_Channel_End_rf* ∈ {→Passive_Source_Channel_End, →Active_Source_Channel_End}, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Destroy_Channel** (*Channel_End_rf* ∈ {→Passive_Source_Channel_End, →Active_Source_Channel_End})
returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩

Algorithms

```
Create_Channel (Loc)
//
// Loc = The location where to create the Passive_Source_Channel_End
// and the Active_Source_Channel_End.
//
// returns <Passive_rf ∈ →Passive_Source_Channel_End,
// Active_rf ∈ →Active_Source_Channel_End,
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   Passive_rf := new Passive_Source_Channel_End @ Loc;
   if ( Passive_rf = ERROR ) then
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Active_rf := new Active_Source_Channel_End @ Loc;
   if ( Active_rf = ERROR ) then
      Delete(Passive_rf);
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Passive_rf↑.Passive_Wait := FALSE;
   Passive_rf↑.Active_Wait := FALSE;
   // Knowing each other.
   //
   Passive_rf↑.Active_rf := Active_rf;
   Active_rf↑.Passive_rf := Passive_rf;

   return <Passive_rf, Active_rf, SUCCESS>;
end

Write (Passive_rf, X)
//
// Passive_rf ∈ →Passive_Source_Channel_End, X ∈ Value.
// Value X gets lost in this function...
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Passive_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;
   Lock(Passive_rf↑.Write_Lock);

   if ( Passive_rf↑.Active_Wait ) then
      // Active_end is waiting due to a write operation.
      // No need to lock the Active_end since it is waiting.
      //
      Passive_rf↑.Active_Wait = FALSE;
      Unlock(Passive_rf↑.Active_rf↑.Wait_Lock);
   else
      // There was no take operation performed on the Active_end,
      // the Passive_end must wait.
      //
      Passive_rf↑.Passive_Wait := TRUE;
      // Allow the Active_end to enter the Passive_end meanwhile
      Unlock(Passive_rf↑.Write_Lock);
      // Wait_Lock is initially locked!
      Lock(Passive_rf↑.Wait_Lock);
      Lock(Passive_rf↑.Write_Lock);
   fi

   // At this point the synchronous write operation is completed.

   Unlock(Passive_rf↑.Write_Lock);
   Unlock(Passive_rf↑.CE_Lock);
   return SUCCESS;
end
```

```
Write (Active_rf, X)
//
// Active_rf ∈ →Active_Source_Channel_End, X ∈ Value.
// Value X gets lost in this function...
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
begin
   ERROR := Lock(Active_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   // We need to consult the Source.
   // Prevent the Source from moving by locking own Move_Lock.
   Lock(Active_rf↑.Move_Lock);
   // No need to check for Error, reference to Passive_end should always be valid now.
   Lock(Active_rf↑.Passive_rf↑.Write_Lock);
   if ( Active_rf↑.Passive_rf↑.Passive_Wait) then
      // The Passive_end is already waiting.
      Active_rf↑.Passive_rf↑.Passive_Wait = FALSE;
      Unlock( Active_rf↑.Passive_rf↑.Wait_Lock);
      Unlock(Active_rf↑.Passive_rf↑.Write_Lock);
      Unlock(Active_rf↑.Move_Lock);
      else
      // Tell the Passive_end to inform us when Active_end can write.
      Active_rf↑.Passive_rf↑.Active_Wait := TRUE;
      // now the Active_end must wait.
      Unlock(Active_rf↑.Passive_rf↑.Write_Lock);
      Unlock(Active_rf↑.Move_Lock);
      // Wait_Lock is initially locked!
      Lock(Active_rf↑.Wait_Lock);
      fi
   fi
   Unlock(Active_rf↑.CE_Lock);
   return <SUCCESS>;
end

Move (Channel_End_rf, Target)
//
// Channel_End_rf ∈ {→Passive_Source_Channel_End, →Active_Source_Channel_End}
//
// Target = The location target.
//
// returns <New_Channel_End_rf ∈
// {→Passive_Source_Channel_End, →Active_Source_Channel_End},
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   ERROR := Lock(Channel_End_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;

   if ( Channel_End_rf↑ ∈ Active_Source_Channel_End ) then
      // Lock Active-end first, then Passive-end.
      // There is no Unlock for the following Lock.
      //
      Lock(Channel_End_rf↑.Move_Lock);
      Lock(Channel_End_rf↑.Passive_rf↑.Move_Lock);

      New_Channel_End_rf := new Active_Source_Channel_End @ Target;
      if ( New_Channel_End_rf = ERROR ) then
         Unlock(Channel_End_rf↑.Passive_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.Move_Lock);
         Unlock(Channel_End_rf↑.CE_Lock);
         return <UNDEFINED, FAILURE>;
      fi

      Copy_Information(Channel_End_rf, New_Channel_End_rf);
      Channel_End_rf↑.Passive_rf↑.Active_rf := New_Channel_End_rf;
      Delete(Channel_End_rf);

      Unlock(New_Channel_End_rf↑.Passive_rf↑.Move_Lock);
```

```
   else

       // Lock Active-end first, then Passive-end.
       //
       do
           ERROR := Lock(Channel_End_rf↑.Active_rf↑.Move_Lock);
       until ( ¬ERROR )

       // There is no Unlock for the following Lock.
       //
       Lock(Channel_End_rf↑.Move_Lock);
       New_Channel_End_rf := new Passive_Source_Channel_End @ Target;
       if ( New_Channel_End_rf = ERROR ) then
           Unlock(Channel_End_rf↑.Move_Lock);
           Unlock(Channel_End_rf↑.Active_rf↑.Move_Lock);
           Unlock(Channel_End_rf↑.CE_Lock);
           return <UNDEFINED, FAILURE>;
       fi
       Copy_Information(Channel_End_rf, New_Channel_End_rf);
       Channel_End_rf↑.Active_rf↑.Source := New_Channel_End_rf;
       Delete(Channel_End_rf);

       Unlock(New_Channel_End_rf↑.Active_rf↑.Move_Lock);
   fi
   return <New_Channel_End_rf, SUCCESS>;
end


Destroy_Channel (Channel_End_rf)
//
// Channel_End_rf ∈ {→Passive_Source_Channel_End, →Active_Source_Channel_End}
//
// returns Action_Status ∈ {SUCCESS, FAILURE}
//
// There is no Unlock in this function
//
begin

   if ( Channel_End_rf↑ ∈ Active_Source_Channel_End ) then
       Active_rf := Channel_End_rf;
   else
       // Check for dangling reference
       //
       ERROR := Lock(Channel_End_rf↑.CE_Lock);
       if ( ERROR ) then return FAILURE;
       Active_rf := Channel_End_rf↑.Active_rf;
       Unlock(Channel_End_rf↑.CE_Lock);
   fi

   // At this point Active_rf ∈ Active_Source_Channel_End,
   // but needs not be valid anymore due to possible
   // movement of sink channel-end.

   // Lock Active-end first, then Passive-end.
   //
   ERROR := Lock(Active_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   do
       ERROR := Lock(Active_rf↑.Passive_rf↑.CE_Lock);
   until ( ¬ERROR )

   Passive_rf := Active_rf↑.Passive_rf;

   Delete(Passive_rf);
   Delete(Active_rf);
   return SUCCESS;
end
```

A.5 Asynchronous Spout Channel

Data-structures

Passive_Sink_Channel_End = ⟨Active_rf, CE_Lock, Move_Lock, Current_Value⟩

- Active_rf = A reference to the Active_Source_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Current_Value = A value that is stored for situations when it is acquired due to a read operation. Initially it is a random Value and the value changes after every take operation. is performed.

Active_Sink_Channel_End = ⟨Passive_rf, CE_Lock, Move_Lock, Current_Value⟩

- Passive_rf = A reference to the Passive_Source_Channel_End of the same channel.
- CE_Lock = A binary semaphore.
- Move_Lock = A binary semaphore.
- Current_Value = A value that is stored for situations when it is acquired due to a read operation. Initially it is a random Value and the value changes after every take operation is performed.

Initially all the semaphores are unlocked, except for Wait_Lock.

Functions, an Overview

- **Create_Channel** (*Loc* ∈ Location)
  returns ⟨*Passive_rf* ∈ →Source_Channel_End, *Active_rf* ∈ →Sink_Channel_End, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Take** (*Passive_rf* ∈ →Passive_Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Take** (*Active_rf* ∈ →Active_Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Read** (*Passive_rf* ∈ →Passive_Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Read** (*Active_rf* ∈ →Active_Sink_Channel_End)
  returns ⟨*X* ∈ Value, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Move** (*Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End}, *Target* ∈ Location)
  returns ⟨*New_Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End}, *Action_Status* ∈ {SUCCESS, FAILURE}⟩
- **Destroy_Channel** (*Channel_End_rf* ∈ {→Source_Channel_End, →Sink_Channel_End})
  returns ⟨*Action_Status* ∈ {SUCCESS, FAILURE}⟩

Algorithms

```
Create_Channel (Loc)
//
// Loc = The location where to create the Passive_Sink_Channel_End
and the Active_Sink_Channel_End.
//
// returns <Passive_rf ∈ →Source_Channel_End,
// Active_rf ∈ →Sink_Channel_End,
// Action_Status ∈ {SUCCESS, FAILURE}>.
//
begin
   Passive_rf := new Passive_Sink_Channel_End @ Loc;
   if ( Passive_rf = ERROR ) then
```

```
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Active_rf := new Active_Sink_Channel_End @ Loc;
   if ( Active_rf = ERROR ) then
      Delete(Passive_rf);
      return <UNDEFINED, UNDEFINED, FAILURE>;
   fi

   Passive_rf↑.Current_Value := randomValue();
   Active_rf↑.Current_Value := randomValue();
   // Knowing each other.
   //
   Passive_rf↑.Active_rf := Active_rf;
   Active_rf↑.Passive_rf := Passive_rf;

   return <Passive_rf, Active_rf, SUCCESS>;
end

Take (Passive_rf)
//
// Passive_rf ∈ →Passive_Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Passive_rf↑.CE_Lock);
   if ( ERROR ) then return FAILURE;

   Value X := Passive_rf↑.Current_Value;
   Passive_rf↑.Current_Value := randomValue();

   Unlock(Passive_rf↑.CE_Lock);
   return <X, SUCCESS>;
end

Take (Active_rf)
//
// Active_rf ∈ →Active_Sink_Channel_End.
//
// returns <X ∈ Value, Action_Status ∈ {SUCCESS, FAILURE}>
//
begin
   ERROR := Lock(Active_rf↑.CE_Lock);
   if ( ERROR ) then return <UNDEFINED,FAILURE>;
   // No simultaneous takes on both ends at the same time,
   // therefore we lock the Passive-end until our take action is completed.
   do
      ERROR := Lock(Active_rf↑.Passive_rf↑.CE_Lock);
   until ( ¬ERROR )
   Value X := Active_rf↑.Current_Value;
   Active_rf↑.Current_Value := randomValue();

   Unlock(Active_rf↑.CE_Lock);
   Unlock(Active_rf↑.Passive_rf↑.CE_Lock);
   return <X, SUCCESS>;
end
```

The functions **Read** (*Active_rf* ∈ →*Active_Sink_Channel_End*) and **Read** (*Passive_rf* ∈ →Passive_Sink_Channel_End), are exactly the same as their corresponding take functions with the exception that they not update the `Current_Value` field. So the code line `Active_rf↑.Current_Value := randomValue();` and `Passive_rf↑.Current_Value := randomValue();` is omitted in these functions.

The functions **Move (Channel_End_rf, Target)** and **Destroy_Channel (Channel_End_rf)** are exactly the same as the ones for the *synchronous drain* channel type.

Summary

A distributed system is a collection of independent computers that appears to its users as a single coherent system. An example of such a system is the Internet, which is the biggest distributed system in the world. The independent computers of a distributed system are connected to each other through a network. On each of these computers there is at least one (software) component that needs to communicate with other components on remote computers to achieve some goal. Components can consist of threads, processes, databases, applications, etc. These components are not only distributed among the several computers of a network but they also run in parallel; i.e. the threads and processes of a component run in parallel. Therefore, besides communication, distributed systems need appropriate theory and infrastructures for the *coordination* of its concurrently running components.

In this thesis we present *MoCha*, a novel coordination framework for distributed systems that is based on the notion of mobile channels. A mobile channel is a coordination primitive that consists of (exactly) two ends, and can be regarded as a virtual path between components. This coordination primitive enables communication between and provides (basic) coordination among the components that use it.

The ends of the channels in the MoCha framework are *mobile*, hence the name "mobile channels". 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 (usually associated with a computer) to another. For example, a web service that migrates from one computer to another.

Mobile channels provide basic *exogenous coordination*. Channels allow several different types of connections among components without the components themselves knowing which channel types they are dealing with. Examples of channel types are synchronous, lossy synchronous, FIFO, asynchronous drain, etc. This makes it possible to coordinate components from the 'outside' (exogenous), and thus, change a distributed system's behavior without having to change its components.

In the thesis, we divide the presentation of the MoCha framework in three main parts: *semantics*, *implementation* and *composition*. Each of these parts offers different contributions and a different perspective on the framework.

In the semantics part, we define three different kinds of semantics: a translation of MoCha in *Petri Nets*, a description in process calculi thanks to an extension of the *π-calculus*, and compositional trace-based semantics. The translation in Petri nets

focuses on the concurrency aspect of mobile channels and allows the use of tools offered by the Petri net community. The description in MoCha-π, the process calculus, focuses on process interaction and allows to reason on programs and on their composition. Finally, the compositional trace-based semantics provide an operational perception of the framework that is more targeted at the notion of components that is used in component-based software.

In the implementation part, we discuss the MoCha middleware, which implements the MoCha framework. The middleware is presented from two points of view: the view of the user and the implementation level view. In the first view, we introduce the main features of the middleware's Application Programming Interface, we provide examples of how to use the middleware, and look at several applications of the middleware; like, for example, peer-to-peer file-transfer applications. In the second view, we discuss the implementation details of the MoCha middleware. We conceptually explain the many algorithms and the internal architecture of the middleware. In particular, we focus on the *Java RMI* layer, the peer-to-peer mobile architecture we build upon it, and the implementation of the mobile channels. We also provide performance measurements, by introducing all kinds of experiments and evaluating their results.

Finally, in the composition part, we discuss how to build complex channel behaviors from basic channels. We take some ideas of *Reo*, the first model for composition of mobile channels into connectors, and implement a subset of it. We do this by providing a model for composition of mobile channels that is conceptually closer to the actual implementation in distributed systems. Mobile channel composition in our model is accomplished by using coordination components. For each such component we give semantics by providing a MoCha-π specification and a Petri Net. The implementation of these components is realized in the MoCha middleware.

Samenvatting

Mobiele kanalen voor exogene coördinatie van gedistribueerde systemen

Een gedistribueerd systeem is een verzameling van onafhankelijke computers die naar de gebruikers toe één enkel coherent systeem lijkt te zijn. Een voorbeeld van een dergelijk systeem is het Internet, dat het grootste gedistribueerde systeem in de wereld is. De onafhankelijke computers van een gedistribueerd systeem zijn met elkaar verbonden via een netwerk. Op elk van deze computers is er minstens één (software) component die met andere componenten op andere computers in het netwerk moet communiceren om zo één gezamenlijke doel te bereiken. Deze componenten kunnen uit threads, processen, databanken, toepassingen, enzovoort bestaan. De componenten zijn niet alleen verdeeld over de verschillende computers van een netwerk maar ze executeren ook nog parallel van elkaar; d.w.z. de threads en processen waar een component uit bestaat executeren in parallel. Dit leidt tot de behoefte naar geschikte theorieën en infrastructuren voor de *coördinatie* van de in parallel executerende componenten van gedistribueerde systemen.

Dit proefschrift introduceert *MoCha*, een nieuwe coördinatie raamwerk voor gedistribueerde systemen die op het begrip van mobiele kanalen is gebaseerd. Een mobiele kanaal is een coördinatie primitieve die uit (precies) twee uiteinden bestaat, en dat als een virtuele pad tussen componenten kan worden beschouwd. Deze coördinatie primitieve regelt de (basis) coördinatie (en maakt de communicatie mogelijk) tussen de componenten die het gebruiken.

De uiteinden van de kanalen in MoCha zijn *mobiel*, vandaar de naam “mobiele kanalen”. Dit maakt het mogelijk om de kanaalverbindingen tussen de componenten van een systeem dynamisch te veranderen, een eigenschap dat zeer nuttig is voor gedistribueerde systemen en zelfs essentieel is voor systemen waar de componenten zelf mobiel zijn. Een component wordt aangeduid als “mobiel” wanneer het in staat is om van een locatie (in het netwerk) naar een andere te verhuizen. Een voorbeeld hiervan zijn webdiensten die van computer kunnen migreren.

Mobiele kanalen verstrekken basis *exogene coördinatie*. De kanalen staan het toe om verschillende soorten verbindingen tussen componenten tot stand te laten komen zonder dat de componenten zelf weten met welke kanaaltype ze te maken hebben. Voorbeelden van kanaaltypes zijn synchroon, verlieslijdende synchroon (lossy synchronous), FIFO, asynchrone afvoerkanaal (asynchronous drain), enzovoort. Exogene coördinatie maakt het mogelijk om componenten van “buitenaf” te coördineren (exogeen), met als voordeel het in staat zijn om het gedrag van een systeem te veranderen zonder zijn componenten zelf hiervoor te hoeven veranderen.

De presentatie van MoCha is in dit proefschrift in drie hoofddelen verdeeld:

semantiek, *implementatie* en *compositie*. Elk van deze delen bevat verschillende resultaten en bekijkt MoCha vanuit een ander perspectief.

In het semantiek gedeelte worden er drie soorten semantieken gedefinieerd: een vertaling (van MoCha) in *Petri Netten*, een beschrijving in proces calculi dankzij een extensie van de π-*calculus*, en een compositionele "trace-based" semantiek. De vertaling in Petri Netten concentreert zich op de "concurrency" aspect van mobiele kanalen en maakt het mogelijk om de tools te gebruiken van de Petri Netten gemeenschap. De beschrijving in *MoCha*-π, de proces calculus, concentreert zich op procesinteractie en staat het toe om te redeneren over programma's en over hun (parallelle) compositie. Tot slot, verstrekt de "trace-based" semantiek een operationele waarneming van MoCha dat meer aansluit bij het begrip van componenten zoals die in component-based software wordt gebruikt.

In het implementatie gedeelte wordt de MoCha middleware besproken. Deze middleware is de implementatie van het MoCha raamwerk. De middleware wordt vanuit twee oogpunten bekeken: die van de gebruiker en die van de ontwikkelaar die meer wilt weten over details van de implementatie. Vanuit het eerste oogpunt worden de belangrijkste features van de applicatie interface (Application Programming Interface) besproken, worden er voorbeelden gegeven over het gebruik van de middleware, en wordt er gekeken naar verschillende toepassingen van de middleware. Zoals, bijvoorbeeld, peer-to-peer bestanden uitwissel applicaties. Vanuit het tweede oogpunt worden de implementatie details van de MoCha middleware besproken. De interne architectuur van de middleware en de verschillende algoritmes worden op een conceptuele wijze weergegeven. In het bijzonder worden de volgende zaken besproken: de *Java RMI* laag, de peer-to-peer mobiele architectuur die boven de vorige laag is gebouwd, en de implementatie van de mobiele kanalen. Er worden ook allerlei experimenten en de resultaten die daar uit voortkomen besproken.

Ten slotte, in het compositie gedeelte, wordt er besproken hoe complexere vormen van kanaalgedrag te verkrijgen zijn door het samenstellen van basiskanalen. Hiervoor worden er sommige ideeën van *Reo* genomen, het eerste model voor compositie van mobiele kanalen in connectoren, en wordt er een implementatie van een deel van Reo gegeven. Dit wordt gedaan door een model voor compositie van mobiele kanalen te introduceren die dichter bij het implementatie niveau van gedistribueerde systemen ligt. De compositie van mobiele kanalen wordt in dit model verwezenlijkt door coördinatiecomponenten te gebruiken. Voor elk zo'n component wordt er een MoCha-π en een Petri Net specificatie gegeven. De implementatie van deze componenten wordt gerealiseerd in de MoCha middleware.

Curriculum Vitae

Juan Guillen Scholten werd geboren op 11 maart 1974 te Delft. Enkele jaren daarna, toen hij vijf was, verhuisde hij mee met zijn ouders naar Malaga (Spanje) waar hij het reguliere Spaanse basisonderwijs volgde. Bij zijn terugkomst naar Nederland, op vijftienjarige leeftijd, volgde hij voortgezet onderwijs op het Hugo Grotius te Delft. In 1994 behaalde hij daar met goed gevolg het VWO-diploma.

In het najaar van 1994 begon Juan met de studie Wijsbegeerte aan de Universiteit Leiden. In 1995 volgde hij wat bijvakken bij de studie Informatica aan het Leiden Institute for Advanced Computer Science (LIACS). Deze bijvakken vond hij zo leuk dat hij besloot om de gehele propedeuse Informatica te volgen. Na het behalen van beide propedeuses kwam hij echter tot de conclusie dat hij eigenlijk nog niet genoeg levenservaring had om de vele problematieken in de filosofie goed te kunnen doorgronden. Hierdoor besloot hij zich volledig te richten op de studie Informatica. In 2001 studeerde hij af in deze studie. Naast studeren was hij ook nog betrokken bij andere nevenactiviteiten. Zo is hij in 1996 voorzitter geweest van het Leids Filosofisch Dispuut. Bij het LIACS is hij student-assistent geweest voor verschillende vakken, is hij lid geweest van de instituutsraad, en heeft hij meegewerkt aan het (destijds) vernieuwende Haagz! Project.

In 2000 begon Juan een stage bij het Centrum voor Wiskunde en Informatica (CWI) te Amsterdam in het coördinatie talen thema-groep (SEN3) van prof. dr. Jaco de Bakker (later opgevolgd door prof. dr. Jan Rutten). Vanwege de goede resultaten van de stage besloot het CWI hem, na diens afstuderen in 2001, aan te nemen als Onderzoeker in Opleiding. Zowel zijn stage als zijn promotieonderzoek werd uitgevoerd onder de supervisie van prof. dr. Farhad Arbab, dr. Frank de Boer, en dr. Marcello Bonsangue. Tijdens zijn onderzoek werkte hij mee aan drie grote projecten: het nationale project ArchiMate over bedrijfsarchitectuur, het bilaterale NWO en DFG project Mobi-J over "assertional" methoden voor mobiele asynchrone kanalen in Java, en het interne MoCha project over mobiele kanalen. Zijn onderzoeksresultaten binnen deze projecten leidden tot verschillende publicaties in internationale conferenties en workshops. Naast onderzoeken heeft hij studenten en stagiaires begeleidt. In 2002 en 2003 heeft hij meegeholpen aan de lokale organisatie van het "Formal Methods for Components and Objects" symposium (FMCO). Van oktober 2003 tot oktober 2005 heeft hij de lokale administratie van het tweewekelijkse "Amsterdam Coordination Group (ACG)" colloquium verzorgd. In 2002 nam hij deel aan de zomerschool "Models, Algebras and Logic of Engineering Software" in Marktoberdorf (Duitsland).

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ.* Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support- .* Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21