MoCha-pi, an Exogenous Coordination Calculus based on Mobile Channels

Juan Guillen-Scholten,
Farhad Arbab, Frank de Boer,
Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands
{juan, farhad, frb}@cwi.nl

Marcello Bonsangue*
LIACS (Leiden University)
Niels Bohrweg 1, 2333 CA Leiden
The Netherlands
marcello@liacs.nl

ABSTRACT

In this paper we present MoCha-π, an exogenous coordination calculus that is based on mobile channels. A mobile channel is a coordination primitive that allows anonymous point-to-point communication between processes. Our calculus is an extension of the well-known π-calculus. 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. An immediate application of this calculus is the modeling of the MoCha middleware, a distributed system that coordinates components using mobile channels.

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: Semantics

Keywords

Calculus, Coordination, Distributed Mobile Channels

1. INTRODUCTION

In the MoCha Framework [6] components and processes are coordinated by *mobile channels*. A mobile channel is a coordination primitive that allows anonymous point-to-point communication, enables dynamic reconfiguration of channel connections in a system, and provides exogenous coordination.

Mobile channels are interesting for all kinds of entities that need to be coordinated, but they are specially interesting for Component Based Software. They provide a highly expressive data-flow architecture for the construction of complex coordination schemes, independent of the computation parts of components. This enhances the reusability 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: on the one hand, we can replace a component with another version without having to change any other component or the coordination scheme in the system. On the other hand, we can replace the coordination scheme with another one without changing the components of the system.

Currently, there is an implementation of the MoCha framework. The MoCha middleware [7] implements mobile channels for communication and coordination of (distributed) processes and components. However, it lacks a suitable logical model for specification and verification of systems in the software development phase. Therefore, in this paper we present MoCha-π, an exogenous coordination calculus based on mobile channels built on top of the π-calculus [11]. The novelty of our calculus is in the fact that channels have different types. These types are defined in terms of π-calculus processes, allowing the introduction of new channel types without having to change the rules of the calculus itself. Since the processes don't know the type of the channels they deal with, it is easy to coordinate them in an exogenous way: coordination from outside. Another novelty of our calculus is the fact that it views channels as resources where processes must compete with each other in order to gain access to a particular channel.

In the next section we introduce the general notion of a mobile channel while presenting the MoCha framework and middleware. In section 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, we give a design pattern for specifying channels that are compatible with the framework. Afterward, we show a representative example in section 5. We conclude with related and future work in section 6.

2. MOCHA

A channel in MoCha, see figure 1, consists of a pair of two distinct ends: usually (*source*, *sink*) for most common channel-types, but also (*source*, *source*) and (*sink*, *sink*) for special types. These channel-ends are available to the processes of an application. Processes can *write* by inserting

*The research of Dr. Bonsangue has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

values into the source-end, and *read* by removing values from the sink-end of a channel; the data-flow is locally *one way*: from a process into a channel or from a channel into a process.

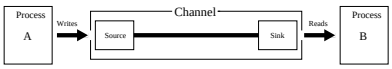

Figure 1: General View of a Channel.

Channels are *point-to-point*, they provide a directed virtual path between the (remote) processes involved in the connection. Therefore, using channels to express the communication carried out within an application is *architecturally very expressive*, because it is easy to see which processes (potentially) exchange data with each other. This makes it easier to apply tools for the analysis of dependencies and data-flow analysis in an application.

Channels provide *anonymous communication*. This enables processes to exchange messages with other processes without having to know *where* in the network those other processes reside, *who* produces or consumes the exchanged messages, and *when* a particular message was produced or will be consumed. Since the processes do not know each other, it is easy to update or exchange any one of them without the knowledge of the processes at the other side of the channels it is connected to. This provides a simple mechanism for composition of processes that are decoupled in space and time.

The ends of a channel are *mobile*. We introduce here two definitions of mobility: physical and logical. The first 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* wherein processes execute. The second, logical mobility, is typically defined in the π-calculus as the ability of passing channel(-end) identities through channels themselves to other processes in the application; i.e., spreading the knowledge of channel(-ends) references by means of channels. This is possible in MoCha. However, since we view channels as resources (see section 3.1) we define logical mobility as the changing of channel connections among processes in a system by means of *connect* and *disconnect* operations. Both physical and logical mobility are supported by MoCha.

Mobility allows dynamic reconfiguration of channel connections among the processes in an application, a property that is very useful and even crucial in systems where processes are mobile. A process is called mobile when, in a distributed system, it can move from one location (where its code is executing) to another. Because the communication via channels is also *anonymous*, when a channel-end moves, the processes at the other side of the channel are not aware nor affected by this movement.

Channels provide transparent *exogenous coordination*. Channels allow several different types of connections among processes without them knowing which channel types they deal with. Only the creator of the connection knows the type of the channel. This makes it possible to coordinate processes from the outside (exogenous), and thus, change an application's behavior without having to change the code of its processes.

2.1 Channel Types

Presently, MoCha supports eleven types of channels, with the same interface, but with different behavior. We give a short description of four representative channel types. For more details and the remaining channel types we refer to the MoCha middleware manual [7].

- *Synchronous channel*. The I/O operations on the two ends 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 *take* operation is the destructive version of the *read* operation.

- *Asynchronous unbounded FIFO channel*. The I/O operations performed on the two channel-ends succeeds asynchronously. Values written into the source channel-end are stored in the channel in a FIFO distributed buffer until taken from the sink-end.

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

- *Asynchronous spout channel*. The I/O operations performed on the ends of this channel succeed one at a time exclusively. This channel has two sink-ends. So the *take* operations on its two ends never succeed simultaneously. The channel produces random values to take.

2.2 Implementation

The MoCha framework is implemented in the *Java* language using the *Remote Method Invocation package* (RMI) [10]. This MoCha middleware can be used for both distributed and non-distributed applications. More details can be found in [7].

3. THE MOCHA-π CALCULUS

In this section we present the MoCha-π calculus. MoCha-π is an extension of the π-calculus [11, 12]. 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 MoCha-π's interface I/O actions. 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.

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 $c \in Links \subseteq \mathcal{N}$. For channel-ends we use $e \in ChannelEnds \subseteq \mathcal{N}$. Data is represented by $d \in Data \subseteq \mathcal{N}$. Observe that the sets $Links$,$ChannelEnds$, $Data$ are mutually exclusive. However for convenience, in the paper 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: $\{T, PRODUCER\}$ for *threads*, $\{K, FIFO\}$ for *channels*, $\{P, PROCESS\}$ for *runtime processes*. For *resources* we use a special notation given in definition 4. A system definition is given by $System = \langle D|S\rangle$, 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 $A \in Id$ 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 3.3, we can see that the creation of a process consists of substituting all free names $x_1, ..., x_n$ by the actual parameters $y_1, ..., y_n$

Definition 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.

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 2. A *Channel* is a user-defined process specification with grammar L^{ϑ} that has the following syntax:

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

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 3.2. 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.

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

Definition 3. 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:

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

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.

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. 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.

Each end e of a channel process K has a user defined number of resources $\mathcal{R}_1^e, \mathcal{R}_2^e, \mathcal{R}_3^e, ...$ We use $\mathcal{R}^e \in \{\mathcal{R}_1^e, \mathcal{R}_2^e, \mathcal{R}_3^e, ...\}$ to refer to any resource of a particular channel-end e.

3.2 Actions

We now define the actions of our calculus. For the MoCha-π channel-end actions we define a transition relation $\longrightarrow$.

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.

connect: $e\downarrow.P \ + \ Q \mid \mathcal{R}^e \longrightarrow e[P]$
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) each channel-end has exactly one resource. The success of a connect operation, therefore, guarantees exclusive channel-end access for its performing process P.

connect(2): $e[e\downarrow.P \ + \ Q] \longrightarrow e[P]$
If a process performs a connect action on a channel-end that it is already connected to, then the operation always succeeds. No resources are affected by this action.

disconnect: $e\uparrow P \ + \ Q \xrightarrow{e\uparrow} P$
This action has two cases: one where the process initially is connected to the channel, and one where it is not. In both cases process P will be disconnected from the channel-end after the operation. This rule applies if and only if P contains no original π-calculus actions $\overline{e}\langle x \rangle$ or $e(x)$ for any given x before the disconnect action takes place.

The action produces a label to determine what happens to the channel-end resource at a higher level. If P was connected to the channel-end, then the restriction rule (3), in section 3.4, dictates that a resource $\mathcal{R}^e$ becomes available to other processes. If P was not connected to the channel-end, then no resource becomes available. We assume that the label is implicitly dropped at the highest level.

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.

Write: $e[e!\langle a \rangle.P \ + \ Q] \longrightarrow e[\overline{e}\langle a \rangle.e(\lambda).P]$
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 paper 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 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 rule 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: $e[e?(b).P \ + \ Q] \longrightarrow e[\overline{e}\langle \lambda \rangle.e(b).P]$
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 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.

3.3 Structural Congruence

Before defining the reaction rules we need to 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 [11]. One might think that this congruence also holds for the MoCha-π calculus in general since the new communication actions can be translated into the original ones. However, this is not the case since we need to look at whether a process is connected to a particular channel-end before transforming a high-level interface action into a pair of π-calculus ones. This can only be done dynamically at execution time. Therefore, we need to re-define the notion of structural congruence for our calculus by adding an equation for the connected scope $e[\ ]$.

Definition 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
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 $x \notin fn(P)$
   $new\ x\ \mathbf{0} \equiv \mathbf{0}$, $new\ x\ y\ P \equiv new\ y\ x\ P$

$$\frac{P \longrightarrow P'}{P|Q \longrightarrow P'|Q}\ \text{P}arallel \qquad \frac{P \longrightarrow P'}{new\ x\ P \longrightarrow new\ x\ P'}\ \text{R}estriction(1)$$

$$\frac{P \longrightarrow P'}{e[P] \longrightarrow e[P']}\ \text{R}estriction(2) \qquad \frac{P \xrightarrow{e\uparrow} P'}{e[P] \longrightarrow P' \mid \mathcal{R}^e}\ \text{R}estriction(3)$$

$$\frac{P \longrightarrow P'}{Q \longrightarrow Q'}\ \text{S}tructural\ Rule \quad \text{if } P \equiv Q \text{ and } P' \equiv Q'$$

R*eaction*:
$(c(a).P + M) \mid (\overline{c}\langle b\rangle.Q + N) \longrightarrow \{^b/_a\}P \mid Q$
$e[(c(a).P + M)] + S \mid (\overline{c}\langle b\rangle.Q + N) \longrightarrow \{^b/_a\}e[P] \mid Q$
$(c(a).P + M) \mid e[(\overline{c}\langle b\rangle.Q + N)] + S \longrightarrow \{^b/_a\}P \mid e[Q]$

Figure 2: The Reaction Rules of MoCha-π

5. $e[P \mid Q] \equiv P \mid e[Q]$ if $e \notin n(P)$
   $e[g[P]] \equiv g[e[P]]$

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

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

Analogous to equation (4), our added equation (5) states that a process not containing e can be included or excluded from the scope $e[\ ]$ without changing the meaning of the process expression. 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.

3.4 Reaction and other Rules

We now define reaction and other support rules of MoCha-π. We take the π-calculus rules and add two extra restrictions and reactions. The rules are given in figure 2.

The reaction rule works at the π-calculus level and therefore is oblivious to whether or not processes are connected to particular channel-ends or not. However, we need to take into account that processes may be in the scope of a channel-end e, written as $e[\ ]$. The restriction rule (2) gives us the means to let a reaction happen within this scope. However, we need two additional rules for the cases where only one of the parallel processes is within the scope $e[\ ]$.

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 3.2) 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, $S \stackrel{def}{=} new\ (e_1, e_2, e_3, e_4)(\ K(e_1, e_2) \mid K(e_3, e_4)\ )$, where S creates two channel processes of type K is allowed, but not $S \stackrel{def}{=} new\ (e_1, e_2)(\ K(e_1, e_2) \mid K(e_1, e_2)\ )$, where S creates two channel processes that share their ends.

4.1 Channel Examples

We now give three channel type process specifications as examples of how to define channel processes that conform to the MoCha design pattern. We already explained their behavior in section 2. 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 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 $CE(l)$ to denote the channel-end that corresponds to link l.

Synchronous

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

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.

FIFO

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

This is the unbounded FIFO channel, with source-end $CE(l)$ and sink-end $CE(r)$, where we model the buffer as a sequence or vector of names $\vec{v}$ that is passed on as a parameter of the channel process. Observe, that if we want to model a LIFO channel type we merely need to take $v_{|\vec{v}|}$ out

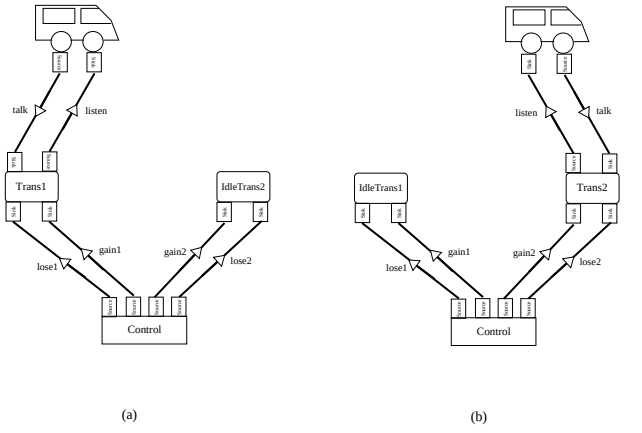

Figure 3: Example, Calling Mobile from a Car

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.

Synchronous Drain

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

This channel 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 writes on its source-ends.

All these channel types are implemented in the MoCha middleware [7]. Other channel types of the middleware are specified in the same way.

5. MOBILE PHONES EXAMPLE

The mobile phones example is presented in [11] 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[11], we consider only one car and two transmitters. In figure 3 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 $\dot{channel}$ 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:

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

The car can either connect to the channel-ends, disconnect from the ends, or either talk or listen when connected. 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.

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 $\ddot{listen}$ 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 $i = 1, 2$:

$$\begin{aligned}
&IdleTrans_i(\ddot{gain}_i, \ddot{lose}_i) \stackrel{def}{=} \ddot{gain}_i \downarrow .\ddot{lose}_i \downarrow .\ddot{gain}_i?(\ddot{talk}) \\
&\quad .\ddot{gain}_i?(\ddot{listen}).\dot{talk} \downarrow .\dot{listen} \downarrow \\
&\quad .Trans_i(\ddot{gain}_i, \ddot{lose}_i, \ddot{talk}, \ddot{listen}) \\
&Trans_i(\ddot{gain}_i, \ddot{lose}_i, \ddot{talk}, \ddot{listen}) \stackrel{def}{=} \\
&\quad new(d_2)\ (\ddot{talk}?(d_1) + \ddot{listen}!(d_2)) \\
&\quad .Trans_i(\ddot{gain}_i, \ddot{lose}_i, \ddot{talk}, \ddot{listen}) \\
&\quad + \ddot{lose}_i?(\ddot{talk}).\ddot{lose}_i?(\ddot{listen}).\dot{talk} \uparrow .\dot{listen} \uparrow \\
&\quad .IdleTrans_i(\ddot{gain}_i, \ddot{lose}_i)
\end{aligned}$$

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 $\dot{talk}$ and $\dot{listen}$. It first writes these ends to the $\dot{gain}_1$ channel-end, so that transmitter 1 can start interacting with the car. This is the situation in figure 3(a). At some point in time, it writes the ends to $\dot{lose}_1$, making transmitter 1 idle again. Fortunately, afterward, it writes the ends to the $\dot{gain}_2$ channel-end. Now transmitter 2 becomes active and takes over the interaction with the car. This is the situation in figure 3(b). After completing the switch, the control unit disconnects from all connected channel-ends and terminates. We give the specification:

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

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.

$$
\begin{aligned}
Sys \stackrel{def}{=}\ & new(\dot{talk}, \ddot{talk}, \dot{listen}, \ddot{listen}, \dot{lose}_i, \ddot{lose}_i, \dot{gain}_i, \ddot{gain}_i) \\
& (\ SYNCHRONOUS(\dot{gain}_i, \ddot{gain}_i) \mid \\
& SYNCHRONOUS(\dot{lose}_i, \ddot{lose}_i) \mid FIFO(\dot{talk}, \ddot{talk}) \mid \\
& SYNCHRONOUS(\dot{listen}, \ddot{listen}) \mid \\
& Control(\dot{gain}_i, \dot{lose}_i, \dot{talk}, \dot{listen}) \mid \\
& IdleTrans_1(\dot{gain}_1, \ddot{lose}_1) \mid IdleTrans_2(\ddot{gain}_2, \ddot{lose}_2) \mid \\
& Car(\dot{talk}, \ddot{listen})\ )\ \ (\text{where } i = 1, 2)
\end{aligned}
$$

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 $\dot{talk}$ and $\ddot{listen}$. These processes then automatically compete among themselves to gain access to the channel-end pair. No other changes are required.

6. CONCLUSIONS, RELATED AND FUTURE WORK

In this paper 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.

Besides MoCha-π there are other calculi that model distributed systems; see [5] for an overview. Two well-known calculi are the *Distributed π-calculus* [8] and the *Ambient* [2] 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* [4] calculus is an asynchronous high-order process calculus that is based on the *Linda* [3] 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 [13].

The MoCha framework strongly relates to Reo [1], a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. The MoCha-π calculus can be extended in order to support Reo. In Reo, channels are composed together into connectors by using *nodes*. Just like MoCha-π channels, a node can be modeled as a special kind of process with a specific communication pattern with its channels.

7. REFERENCES

[1] 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.

[2] L. Cardelli and A. D. Gordon. *Mobile ambients*, Theoretical Computer Science, 240(1):177–213, June 2000.

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

[4] 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.

[5] 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.

[6] J.V. Guillen-Scholten, F. Arbab, F. S. de Boer, M. M. Bonsangue, "A Channel-based Coordination Model for Components". Electr. Notes Theor. Comput. Sci. 68(3), Elsevier Science, 2003.

[7] J.V. Guillen-Scholten and F. Arbab, *MoCha, easyMoCha and chocoMoCha Manual v1.0*, CWI Technical Report, Amsterdam, 2004.

[8] 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.

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

[10] Sun Microsystems , *Java Remote Method Invocation - Distributed Computing for Java*, white paper available at `java.sun.com/rmi`, 2004.

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

[12] J. Parrow. *An Introduction to the pi-Calculus*. In Handbook of Process Algebra, ed. Bergstra, Ponse, Smolka, pages 479-543, Elsevier 2001.

[13] 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.