A Protocol for Multi-threaded Processes with Choice in π-Calculus

Kazunori Iwata, Shingo Itabashi, Naohiro Ishii

Dept. of Intelligence and Computer Science, Nagoya Institute of Technology,
Gokiso-cho, Showa-ku, Nagoya, 466-8555, Japan
kazunori@egg.ics.nitech.ac.jp
shingo@egg.ics.nitech.ac.jp
ishii@egg.ics.nitech.ac.jp

Abstract. We have proposed a new protocol for the multi-threaded processes with choice written in π-calculus[1,2]. This protocol frees the multi-threaded processes from deadlock. It has been defined as transition rules. We have shown why the protocol avoids the deadlock in the multi-threaded processes.

1 Introduction

We propose a new protocol for multi-threaded processes with choice written in π-calculus. π-calculus is a process calculus which can describe a channel-based communication among distributed agents. Agents communicate with each other in π-calculus according to the following rules:

1. A message is successfully delivered when two processes attempt an output and an input at the same time.
2. Agents are allowed to attempt outputs and inputs at multiple channels simultaneously, with only one actually succeeding.

This process of communication has been identified as a promising concurrency primitive[3,4,5,6,7].

In π-calculus agents have a property to choose one process from concurrent processes. In order to choose one process, agents get a mutex-lock and execute the process. The other processes are blocked by the lock and will be stopped, if the process will be successfully executed. This chosen process may be easily executed, if agents execute concurrent processes only by itself. However, these processes are executed by the communication among agents. The mutex-lock is very complex to avoid deadlock. Hence, we adjust the situations in the communication and define the protocol to avoid deadlock[7,8,9,10,11,12]. In the protocol, we appropriate each process in π-calculus to the thread.

2 π-Calculus

π-calculus is a process calculus which is able to describe dynamically changing networks of concurrent processes. π-calculus contains just two kinds of entities: process and channels. Processes, sometimes called agents, are the active components of a system. The syntax of defining a process is as follows:

$$
\begin{array}{rlll}
P ::= & \overline{x}y.P & /\text{* Output */} \\
| & x(z).P & /\text{* Input */} \\
| & P \mid Q & /\text{* Parallel composition */} \\
| & (\nu x)P & /\text{* Restriction */} \\
| & P \ + \ Q & /\text{* Summation */} \\
| & 0 & /\text{* Nil */} \\
| & !P & /\text{* Replication */} \\
| & [x = y]P & /\text{* Matching */}
\end{array}
$$

Processes interact by synchronous rendezvous on channels, (also called names or ports). When two processes synchronize, they exchange a single value, which is itself a channel.

The output process $\overline{x}y.P_1$ sends a value y along a channel named x and then, after the output has completed, continues to be as a new process P_1. Conversely, the input process $x(z).P_2$ waits until a value is received along a channel named x, substitutes it for the bound variable z, and continues to be as a new process $P_2\{y/z\}$ where, y/z means to substitute the variable z in P_2 with the received value y. The parallel composition of the above two processes, denoted as $\overline{x}y.P_1 \mid x(z).P_2$, may thus synchronize on x, and reduce to $P_1 \mid P_2\{y/z\}$.

Fresh channels are introduced by restriction operator ν. The expression $(\nu x)P$ creates a fresh channel x with scope P. For example, expression $(\nu x)(\overline{x}y.P_1 \mid x(z).P_2)$ localizes the channel x, it means that no other process can interfere with the communication between $\overline{x}y.P_1$ and $x(z).P_2$ through the channel x.

The expression $P_1 + P_2$ denotes an external choice between P_1 and P_2: either P_1 is allowed to proceed and P_2 is discarded, or converse case. Here, external choice means that which process is chosen is determined by some external input. For example, the process $\overline{x}y.P_1 \mid (x(z).P_2 + x(w).P_3)$ can reduce to either $P_1 \mid P_2\{y/z\}$ or $P_1 \mid P_3\{y/w\}$. The null process is denoted by $\mathbf{0}$. If output process (or input process) is $\overline{x}y.0$ (or $x(z).0$), we abbreviate it to $\overline{x}y$ (or $x(z)$).

Infinite behavior is allowed in π-calculus. It is denoted by the replication operator $!P$, which informally means an arbitrary number of copies of P running in parallel. This operator is similar to the equivalent mechanism, but more complex, of mutually-recursive process definitions.

π-calculus includes also a matching operator $[x = y]P$, which allows P to proceed if x and y are the same channel.

$\rightarrow$.

3 The Protocol for Multi-threaded Processes with Choice

In this section, we propose a protocol for multi-threaded processes with choice. The processes concurrently communicate each other. First, we introduce basic concepts concerning the communication.

3.1 Basic Concepts

Agent: Agents are units of concurrent execution of our concurrent and distributed system. Agents are implemented as threads. If agents meet the choice process, they make new threads for each process in choice process.

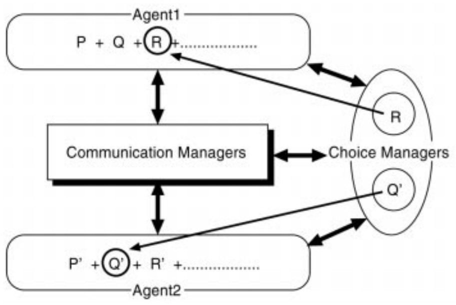
Agent1
P + Q + R + ...
R
Communication Managers
Choice Managers
Q'
P' + Q' + R' + ...
Agent2

Fig. 1. Relationship

Communication Manager: Communication Managers(CMs) manage communication requests on channels from agents. They make possible for agents to communicate with one another. They have queues which consist of the communication requests from agents.

Choice Manager: Choice Managers(CHMs) manage choice processes on agents. They observe the threads made from the choice, and decide which process should be remained.

The relationship among these elements(Agents, CMs,CHMs) is in Fig. 1.

3.2 The Behavior of Agents

States of an Agent: An agent have the variable to store its state. The set of possible states of agent is {*init, wait-CMout, wait-CMin, wait-CHMid, wait-CHM, wait-Res, done, stopped* }.

Tab. 1 describes, in terms of state transition rules, the behavior of the agent.

3.3 The Behavior of CHMs

States of a CHM: CHMs have variables named flag and queue. The variable flag stores the status of one choice process, *suspend* means the choice process is suspended now, *try* means the choice process is tried to execute, *done* means the choice process is executed. The variable queue stores the processes in choice process that are tried to execute by CMs.

Tab. 2 describes the behavior of the CHM.

3.4 The Behavior of CMs

States of a CM: Each CM has two queues named in-*xxx* and out-*xxx* (*xxx* means arbitrary strings) The queue store the request from agents according to the kind of process. $\overset{div}{=}$ denotes the process which divides the queue into the first element and the others.

Tab. 3, 4 describes the behavior of the CM.

4 Free Processes from Deadlock

The processes with choice are nondeterministic, thus the executions have various results. Hence, if the executions are in deadlock, it is difficult to find the cause. In this section, we show why the protocol frees the processes from the deadlock.

To show the freedom from the deadlock, we consider four cases.

1. There is no choice process and the only one paired agents(input and output) use a channel.
2. There is no choice process and the agents use channels. It means some input and output agents use the same channel.
3. The first process, which a CHM determine to execute, in the choice process can be executed.
4. The first process, which a CHM determine to execute, in the choice process cannot be executed.

Table 1. Transition Rules of an Agent

Rule	State	Input	Process	Next State	Output	Other Actions
R1	*init*	-	$\overline{x}[\overrightarrow{y}]$	*wait-CMout*	Send $(in, x, \overrightarrow{y}, aid, 0, 0)$ to CM.	-
R2	*init*	-	$x(\overrightarrow{z})$	*wait-CMin*	Send $(out, x, \overrightarrow{z}, aid, 0, 0)$ to CM.	-
R3	*init*	-	$P \equiv Q + \ldots$	*wait-CHMid*	Send (P, aid) to CHM.	The current process is not changed.
R4	*wait-CHMid*	Get (Cid) from CHM.	$P \equiv Q + \ldots$	*wait-CHM*	-	Each process in choice is divided into one agent and each agent has the state *wait-CHM*.
R5	*wait-CHM*	-	$\overline{x}[\overrightarrow{y}]$	*wait-Res*	Send $(in, x, \overrightarrow{y}, aid, Cid, pid)$ to CM.	-
R6	*wait-CHM*	-	$x(\overrightarrow{z})$	*wait-Res*	Send $(out, x, \overrightarrow{z}, aid, Cid, pid)$ to CM.	-
R7	*wait-Res*	Get $(resume)$ from CHM.	$\overline{x}[\overrightarrow{y}]$	*wait-CMout*	-	This thread is selected to execute.
R8	*wait-Res*	Get $(resume)$ from CHM.	$x(\overrightarrow{z})$	*wait-CMin*	-	This thread is selected to execute.
R9	*wait-Res*	Get $(stop)$ from CHM.	-	*stopped*	-	This thread is stopped.
R10	*wait-CMout*	Get $(output)$ from CM.	$\overline{x}[\overrightarrow{y}]$	*init* if there are more processes otherwise *done*.	-	The state *done* means the process is finished.
R11	*wait-CMin*	Get $(\overrightarrow{v})$ from CM.	$x(\overrightarrow{z})$	*init* if there are more processes otherwise *done*.	-	The state *done* means the process is finished.

Table 2. Transition Rules of an CHM

Rule	State	Input	Next State	Output	Other Actions
R1	-	Get (P, aid) from Agent.	flag = *suspend*	Send (Cid) to *aid*.	Numbering each process.
R2	flag = *suspend*	Get (aid, pid).	flag = *try* from CM	Send *yes* to CM.	-
R3	flag = *try*	Get (aid, pid) from CM.	queue = queue + (aid, pid)	-	-
R4	flag = *try* queue = $\emptyset$	Get $(suspend, aid, pid)$ from CM.	flag = *suspend*	-	-
R5	flag = *try* queue $\overset{div}{\rightarrow}$ (aid', pid') +queue'	Get $(suspend, aid, pid)$ from CM.	queue = queue' + (aid, pid)	Send *yes* to CM which sent (aid', pid').	-
R6	flag = *try*	Get $(executed, aid, pid)$ from CM.	flag = *done*	Send $(resume)$ to *aid* with *pid* and $(stop)$ to *aid* without *pid*.	-
R7	flag = *done*	Get (aid, pid) from CM.	-	Send *no* to CM.	-

Table 3. Transition Rules of a CM(1)

Rule	State	Input	Next State	Output
R1	in-x = $\emptyset$	Get $(out, x, \overrightarrow{z}, aid, 0, 0)$ from agent.	out-x = out-x + $(aid, \overrightarrow{z}, 0, 0)$	-
R2	out-x = $\emptyset$	Get $(in, x, \overrightarrow{y}, aid, 0, 0)$ from agent.	in-x = in-x + $(aid, 0, 0)$	-
R3	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow}$ $(aid', 0, 0)$ + in-x'	Get $(out, x, \overrightarrow{z}, aid, 0, 0)$ from agent.	in-x = in-x'	Send $(output)$ to *aid* and $(\overrightarrow{z})$ to *aid'*.
R4	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow}$ $(aid', \overrightarrow{z}, 0, 0)$ + out-x'	Get $(in, x, \overrightarrow{y}, aid, 0, 0)$ from agent.	out-x = out-x'	Send $(output)$ to *aid'* and $(\overrightarrow{z})$ to *aid*.
R5	in-x = $\emptyset$	Get $(out, x, \overrightarrow{z}, aid, Cid, pid)$ from agent.	out-x = out-x + $(aid, \overrightarrow{z}, Cid, pid)$	-
R6	out-x = $\emptyset$	Get $(in, x, \overrightarrow{y}, aid, Cid, pid)$ from agent.	in-x = in-x + (aid, Cid, pid)	-
R7	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow}$ $(aid', 0, 0)$ + in-x'	Get $(out, x, \overrightarrow{z}, aid, Cid, pid)$ from agent.	Send (aid, pid) to *Cid* and if get *yes* from *Cid* then:	
			in-x = in-x'	Send $(output)$ to *aid* $(\overrightarrow{z})$ to *aid'* $(execute, aid, pid)$ to *Cid*.
			if get *no* from *Cid* then:	
			Ignore this input.	-
R8	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow}$ $(aid', \overrightarrow{z}, 0, 0)$ + out-x'	Get $(in, x, \overrightarrow{y}, aid, Cid, pid)$ from agent.	Send (aid, pid) to *Cid* and if get *yes* from *Cid* then:	
			out-x = out-x'	Send $(output)$ to *aid'* $(\overrightarrow{z})$ to *aid'* $(execute, aid, pid)$ to *Cid*.
			if get *no* from *Cid* then:	
			Ignore this input.	-

Table 4. Transition Rules of an CM(2)

Rule	State	Input	Next State	Output
R9	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow} (aid', Cid', pid')$ + in-x'	Get $(out, x, \overrightarrow{z}, aid, Cid, pid)$ from agent.	Send (aid, pid) to Cid and (aid', pid') to Cid' and if get *yes* from Cid and Cid' then:	
			in-x = in-x'	Send $(output)$ to aid $(\overrightarrow{z})$ to aid' $(execute, aid, pid)$ to Cid $(execute, aid', pid')$ to Cid'.
			if get *yes* from Cid and *no* from Cid' then:	
			in-x = in-x' Apply these rules again.	-
			if get *no* from Cid and *yes* from Cid' then:	
			Ignore this input.	Send $(suspend, aid', pid')$ to Cid'.
			if get *no* from Cid and Cid' then:	
			in-x = in-x' Ignore this input.	-
R10	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow} (aid', Cid', pid')$ + out-x'	Get $(in, x, \overrightarrow{y}, aid, Cid, pid)$ from agent.	Send (aid, pid) to Cid and (aid', pid') to Cid' and if get *yes* from Cid and Cid' then:	
			out-x = out-x'	Send $(output)$ to aid' $(\overrightarrow{z})$ to aid $(execute, aid, pid)$ to Cid $(execute, aid', pid')$ to Cid'.
			if get *yes* from Cid and *no* from Cid' then:	
			out-x = out-x' Apply these rules again.	-
			if get *no* from Cid and *yes* from Cid' then:	
			Ignore this input.	Send $(suspend, aid', pid')$ to Cid'.
			if get *no* from Cid and Cid' then:	
			out-x = out-x' Ignore this input.	-
R11	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow} (aid', Cid, pid)$ + in-x'	Get $(out, x, \overrightarrow{z}, aid, 0, 0)$ from agent.	Send (aid, pid) to Cid and if get *yes* from Cid then:	
			in-x = in-x'	Send $(output)$ to aid $(\overrightarrow{z})$ to aid'.
			if get *no* from Cid then:	
			in-x = in-x' Apply these rules again.	-
R12	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow} (aid', \overrightarrow{z}, Cid, pid)$ + out-x'	Get $(in, x, \overrightarrow{y}, aid, 0, 0)$ from agent.	Send (aid, pid) to Cid and if get *yes* from Cid then:	
			out-x = out-x'	Send $(output)$ to aid' $(\overrightarrow{z})$ to aid.
			if get *no* from Cid then:	
			out-x = out-x' Apply these rules again.	-

Case 1 Let the input process be A_{in} and the output process A_{out}. Each process uses the same link. We consider the process A_{in} is registered to CMs before A_{out}.

1. By R2 in Tab. 3, the id of A_{in} is registered to `in-x`. Then, the state of A_{in} is changed to *wait-CMin* by R2 in Tab. 1
2. If A_{out} is registered to a CM, by R3 in Tab. 3, the value in A_{out} is output to the process indicated by the id in the top of `in-x`. Then, A_{in} gets the value from the CM and executes next process by R11 in Tab. 1. The state of A_{out} gets the results from the CM and executes next process by R10 in Tab. 1.

Hence, the process A_{in} and A_{out} can communicate each other.

Case 2 Let the nth input and the nth output processes which use the same channel exist.

1. The mth input processes have already registered to CMs.

   a) If m == 1 (the length of `in-x` is 1) then
   This condition is same as the case 1. Hence the communication succeeds.

   b) Assuming that the communications succeed on m == k (the length of `in-x` is k) then considering the condition as m == k + 1 (the length of `in-x` is k + 1) :

   When the condition on m == k + 1,

      i. Let the next registered process be the output process.

      By R3 in Tab. 3, the value in the output process is sent to the process indicated by `id` in `in-x`. The output process proceeds to the next process through the state *wait-CMout* by R10 in Tab. 1. The process, which gets the value by R11 in Tab. 1, proceeds to the next process.
      The process in the top on `in-x` and the output process communicates with each other. The length of `in-x` is changed to m - 1, that means m == k.

      ii. Let the next registered process be the input process.

      The length of `in-x` is changed to m + 1, then the communication succeeds by the previous case.

   Then by the assumption of the induction(b), the communication succeeds in any cases.

Case 3 We consider about the choice process $A_1 + A_2 + \cdots + A_n$ and $B_1 + B_2 + \cdots + B_n$.

Let the process A_1 be able to communicate with B_1 and the process A_2 be able to communicate with B_2 and so on. It means the different process uses a different channel.

The choice process $A_1 + A_2 + \cdots + A_n$ is divided into the process A_1, A_2, ... and A_n and are registered to a CHM, by R3 and R4 in Tab. 1. Each process proceeds independently but has the state *wait-CHM*. The choice process $B_1 + B_2 + \cdots + B_n$ is done like as the choice process $A_1 + A_2 + \cdots + A_n$, but different CHM.

There are many combination to execute these processes. Before explaining it, we explain the actions of CHMs.

A CHM gets the processes and commits them to memory (see R1 in Tab. 2). It does not know the channel in the processes and checks the process which is requested to execute by CMs (see R2 and R3 in Tab. 2). The requests means a possibility to use the channel which the process want to use and the process can be executed if the CHM answers *yes*. When the CHM gets the first request, it returns the answer *yes*(see R2 in Tab. 2). When the CHM gets the second request or more requests, it store the requests in the queue and check the head request in the queue if the first request cannot be executed (see R3, R4 and R5 in Tab. 2).

We consider only the case that the process A_1 and B_1 can communicate with each other. However, there are many cases on the communication. These cases are distinguished by the order of registration to CMs. The kind of order is as follows:

1. $A_1 \rightarrow B_1 \rightarrow$ the other processes
   or
   $B_1 \rightarrow A_1 \rightarrow$ the other processes

   These cases means the process A_1 and B_1 registered before the others, and communicate with each other.
   We explain the first case in them.
   The process A_1 registers to a CM by R5 or R6 in Tab. 3. The process B_1 registers to the CM and the CM requests CHMs to execute A_1 and B_1 by R9 or R10 in Tab.4. CHMs answer *yes* to the CM, because the requests is the first request for each CHM(see R2 in Tab.2).
   The CM allow to communicate A_1 and B_1 by R9 or R10 in Tab.4 and CHMs send *stop* to the others by R6 in Tab.2. The other processes which get stop the execution.
   If the other processes register to a CM (before getting the signal *stop*), CHMs answer *no* to the CM by R7 in Tab.2.
2. Some processes do not include the pair A_i and $B_i \rightarrow A_1 \rightarrow$ Some processes do not include the pair to the processes which have already registered $\rightarrow B_1 \rightarrow$ the others
   or
   Some processes do not include the pair A_i and $B_i \rightarrow B_1 \rightarrow$ Some processes do not include the pair to the processes which have already registered $\rightarrow A_1 \rightarrow$ the others

   These cases means the some processes registered before the process A_1(or B_1) registering. When the process B_1 (or A_1) registers to a CM before the pair to the processes which have already registered registers, The CM requests to CHMs to execute it and CHMs answer *yes* to the CM. When the CM gets the answer *yes*, it sends CHMs the signal of execution(see R9 and R10 in Tab. 4). In this condition, if the pair to the processes registers to a CM before CHMs get the signal from the CM, the CM requests to CHMs and the CHM blocks this process(see R4 in Tab. 2). However, whether this process is blocked by the CHM or not, the process A_1 and B_1 communicate with each other. Because the CM has already sent the signal of execution to CHMs and CHMs send which process should be executed to Agent (in this case A_1 and B_1).

In this case CHMs block a process, but it does not generate a dead lock. Because blocking the process is generated by the determination which process should be executed. Hence, the blocked process has no chance to be execute and the block has no influence.

In this case, we consider only two choice processes in which each process uses a different channel. However, if many choice processes in which each process uses same channel, they do not generate a dead lock. Because, the CM considers the processes according to their channel and one CHM manage one choice process and the CHM blocks a process only if the other process which is managed by the CHM can communicate.

Case 4 We consider about these choice processes $A_1 + A_m$, $B_1 + B_n$.

In this case, we consider the condition that CMs sends CHMs the signal of suspension(see R9 and R10 in Tab. 4).

Let the process A_1 be able to communicate with B_1 and the process A_m and B_n be able to communicate other processes(M and N).

If all processes have finished to register CHMs, the condition that CMs sends CHMs the signal of suspension is generated by the order of requests to CHMs from CMs and answers from CHMs.

The order is as follows:

The CM sends the requests to execute the pair A_1 and B_1 B_n and N(see R9 and R10 in Tab. 4). In this condition, if the CHM which manages $A_1 + A_2$ arrows to execute A_1 and the CHM which manages B_1+B_n arrows to execute B_n, the communication between B_n and N succeed. Then B_1 cannot be executed and the CHM answers *no* to the CM(see R9 and R10 in Tab. 4). The CM sends the signal of suspension to the CHM which manages $A_1 + A_2$ The CHM removes the process A_1 from the queue and waits for new request from CMs.

In this case, the process A_1 blocks other processes in the same choice process. However, the process A_1 releases the block if the process B_1 cannot communicate.

If the number of the process in one choice process, CHMs consider only the first process in the queue which stores the requests from CMs. Hence, the condition is the same as this condition.

We consider the all possible condition and show the freedom from deadlock. Hence, by using the protocol, we avoid to deadlock in π-calculus.

5 Conclusion and Future Work

In this paper, we have proposed a new protocol for multi-threaded processes with choice written in π-calculus. In the protocol, we have defined the three elements: agents, Communication Managers(CMs) and Choice Managers(CHMs). Agents are units of concurrent execution of our concurrent and distributed system. CMs manage communication requests on channels from agents. CHMs manage choice processes on agents.

We have shown why the protocol frees the processes from the deadlock. If the agents have no choice process, any process do not be blocked. Hence, the deadlock is avoided. If the agents have choice processes, CHMs order the requests from CMs and manage the block to avoid deadlocks. Hence, the protocol frees the processes from the deadlock.

One of the future works is to implement the compiler for π-calculus and build it into any system like as agent systems, distributed object systems and so on.

6 Acknowledgement

A part of this research result is by the science research cost of the Ministry of Education.

References

1. R. Milner. Polyadic π-calculus:a Tutorial. LFCS Report Series ECS-LFCS-91-180, Laboratory for Foundation of Computer Science, 1991.
2. Milner, R., Parrow, J.G., and Walker, D.J. A calculus of mobile processes. In *Information and Computation,100(1)*, pages 1–77, 1992.
3. R. Bagrodia. Synchronization of asynchronous processes in CSP. In *ACM Transaction on Programming Languages and Systems*, volume 11, No. 4, pages 585–597, 1989.
4. M. Ben-Ari. Principles of Concurrent and Distributed Programing. In *Prentice-Hall International(UK) Limited*, 1989.
5. R.Milner D. Berry and D.N. Turner. A semantics for ML concurrency primitive. In *POPL'92*, pages 119–129, 1992.
6. Benjamin C. Pierce and David N. Turner. Concurrnet Objects in a Process Calculus. LNCS 907, pp.187–215, proc. TPPP'95, 1995.
7. MUNINDAR P. SINGH. Applying the Mu-Calculus in Planning and Reasoning about Action. *Journal of Logic and Computation*, 8:425–445, 1998.
8. G.N.Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. In *ACM Transactions on Programming Languages and Systems*, volume 5, No.2, pages 223–235, 1983.
9. C.A.R. Hoare. Communicating sequential processes. In *Communications of the ACM*, volume 21, No.8, pages 666–677, 1985.
10. E. Horita and K. Mano. Nepi: a network programming language based on the π-calculus. In *Proceedings of the 1st International Conference on Coordination Models, Languages adn Applicationos 1996*, volume 1061 of *LNAI*, pages 424–427. Springer, 1996.
11. E. Horita and K. Mano. Nepi^2: a two-level calculus for network programming based on the π-calculus. In *IPSJ SIG Notes, 96-PRO-8*, pages 43–48, 1996.
12. E. Horita and K. Mano. Nepi^2: a Two-Level Calculus for Network Programming Based on the π-calculus. ECL Technical Report, NTT Software Laboratories, 1997.