名古屋工業大学博士論文
甲第420号(課程修了による)
平成１５年3月２５日授与

Studies on Applying π-Calculus to Formalizing Multi-Agent Systems

by
Kazunori Iwata

Dissertation submitted in partial fulfillment
for the degree of Doctor of Engineering

Under the supervision of
Professor Naohiro Ishii

Nagoya Institute of Technology
Nagoya, Japan

January 2003

Abstract

This thesis studies formalizing agents' behaviors and communications by use of π-calculus. We consider a model and a protocol for multi-threaded processes with choice to avoid deadlocks in using π-calculus. Further, we design a new language to implement π-calculus on a computer more easily than mathematical notations of it, and implement the system by using the model and protocol. Next, we develop a description for agents' plans and methods having dynamically changing structures. Moreover, we develop a protocol for agents to cooperate each other by sharing their actions. Finally, we develop the description to easily find the sharable actions, and the description of agents' actions including communications.

This thesis is divided into 5 chapters.

In Chapter 1, we state multi-agent systems, introduce modeling and formalizing agents.

In Chapter 2, we develop a model and a protocol for multi-threaded processes with choice written in π-calculus. In the model and the protocol, we assign each process in π-calculus to a thread, and define the three elements: Processes, Communication Managers(CMs) and Choice Managers(CHMs). A process is a basic unit to control concurrently executed threads in our concurrent and distributed system. CMs manage communication requests along channels from processes, and CHMs manage choice processes in processes. Moreover, we show why the protocol frees the processes from the deadlock. Finally, we design a primitive language instead of using the mathematical notations of π-calculus, and implement the system.

Chapter 3 considers an agent model to use descriptions for agents' plans and methods based on π-calculus. First, we develop a description for agents' plans. The plans described in π-calculus can be changed dynamically while it is executing, because π-calculus provides dynamically changing structures. Secondly, we develop a description for agents' methods that refer their parameters through channels in π-calculus. Thus, agents can dynamically change the references from the methods to the parameters. Next, we develop an agent model to use the descriptions. The agents can, therefore, change their plans and methods to adapt to the environment around them by themselves by using these property. Finally, we show two experiments: a tracing problem and a fire-world problem.

In Chapter 4, we develop a protocol for agents to cooperate each other by sharing their actions, and define the similar intention focused on agents'actions. In the protocol, agents build a group with agents having the same intentions or the similar intentions by sharing their actions. However, it is hard to find the common sharable actions among actions, since there is much combination in these actions. Thus, we develop the description to easily find such sharable actions, in order to solve the problem above, and explain how the processes make the target sets Moreover, we develop the description of agents' actions including communications, and show the calculations of the desciption by using the operational semantics of π-calculus.

Finally, Chapter 5 summarizes the results derived in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Agent Systems and Multi-Agent Systems are the subfields of Artificial Intelligence that aim to provide both principles for construction of complex systems involving multiple agents and mechanisms for coordination of independent agents' behaviors.

The term "agent" is difficult to define. Agents are often described as entities with attributes considered useful in a particular domain. This is the case with intelligent agents, where agents are seen as entities that emulate mental processes or simulate rational behavior; personal assistant agents, where agents are entities that help users perform a task; mobile agents, where entities that are able to roam networking environments to fulfill their goals; information agents, where agents filter and coherently organize unrelated and scattered data; and autonomous agents, where agents are able to accomplish unsupervised actions. A multi-agent system is a loosely coupled network of problem-solver entities that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity and an environment. More recently, the term "multi-agent system" has been given a more general meaning, and it is now used for all types of systems composed of multiple autonomous components.

When a group of agents in a multi-agent system share common long-term goal, they can be said to from a *team*. Team members(or teammates) coordinate their behaviors by adopting compatible cognitive processes and by directly affecting each other's inputs including via communicative actions. Other agents in the environment that have goals opposed to the team's long-term goal are the team's *adversaries*. Therefore, modeling and formalizing agents in Agent Systems and Multi-Agent Systems are important issues to help to analyze agents' behaviors and their interactions, to design and to implement intelligent agents. However, these are also difficult issues, since they must consider complex, real-time, noisy, collaborative and adversarial multi-agent environments..

In this thesis, we formalize agents' behaviors and communications by use of π-calculus [Mil91, KM01, FG99, Mun98], since it is a process calculus describing dynamically changing networks of concurrent processes. In addition, π-calculus provides a sound foundation to concurrent computations and a communication among parallel processes. These properties are useful to describe agents in complex, real-time, noisy, collaborative and adversarial multi-agent environments.

1.1 Modeling and Formalizing Agents

Agents need plans to satisfy their goals. A *plan* for an agent consists of a serial of processes that the agent will execute. In other words, an agent executes processes derived from the *plan* (called *executing plan*). *Planning* means that an agent constructs a plan for agents. The planning is an important problem for autonomous agent systems. The existing *plans* in Artificial Intelligence are made according to the current environment and future one forecasted by the agent. Thus, these plans in a dynamically changing environment are not always successfully executed due to unforecastable changes in the environment [JZ97, SN98]. If the environment is changed by some factors that are not forecasted by the agent, the *plan* becomes unexecutable and the agent should reform the *plan* according to the new environment. It is an important property that the agent can reform plans by himself. This property is called *reflection*. Therefore, we design and implement a lower-level primitive language based on π-calculus to provide the property *reflection* for agents.

π-calculus is a process calculus describing dynamically changing networks of concurrent processes and has a property choosing one process from them. The property provides a flexibility to agents by choosing the most appropriate process from some concurrent processes[Mun98]. However, calculating processes in π-calculus is complicated and causes a deadlock, because the process that is chosen between them needs to get a lock and block the other processes[BD95]. Thus, we divide the conditions of process communications into a condition with choice processes or without them and design π-model to adjust these divided conditions. In addition to this, we design the language to implement π-calculus on a computer more easily than mathematical notations of it. We implement the π-calculus processes in the model as multi-threaded processes. Thus, agents implemented by use of the model can easily control multi-threaded processes without deadlocks and use the language describing plan based on π-calculus with the property *reflection*.

We use the π-model system to control agents' communications in a multi-agent system. Agents build groups and collaborate each other to improve the performance of the system. Thus, we define agents' collaboration by sharing common actions among agents that have similar intentions [HCY99, FG99, Nak99, CLS97]. However, it is hard to find the common actions among their actions, since there is much combination in these actions and agents find the common actions through a network. We, then, describe actions based on π-calculus to find common actions among agents. In the description, an agent sends action sets to other agents through a network and easily performs actions and communication only by evaluating them. Moreover, an agent parallel calculates action sets to find common actions, because actions are described as parallel processes through a network.

1.2 Outline of Thesis

This section describes the outline of this thesis. This thesis is divided into Introduction, Chapter 2–4, Conclusions and Bibliography.

In Chapter 2, we develop a model and a protocol for multi-threaded processes with choice written in π-calculus. In the model and the protocol, we assign each process in π-calculus to a thread, and define the three elements: Processes, Communication Managers(CMs) and Choice Managers(CHMs). A process is a basic unit to control concurrently executed threads in our concurrent and distributed system. CMs manage communication requests along channels from processes, and CHMs manage choice processes in processes. Moreover, we show why the protocol frees the processes from the deadlock. Finally, we design a primitive language instead of using the mathematical notations of π-calculus, and implement the system.

Chapter 3 considers an agent model to use descriptions for agents' plans and methods based on π-calculus. First, we develop a description for agents' plans. The plans described in π-calculus can be changed dynamically while it is executing, because π-calculus provides dynamically changing structures. Secondly, we develop a description for agents' methods that refer their parameters through channels in π-calculus. Thus, agents can dynamically change the references from the methods to the parameters. Next, we develop an agent model to use the descriptions. The agents can, therefore, change their plans and methods to adapt to the environment around them by themselves by using these property. Finally, we show two experiments: a tracing problem and a fire-world problem.

In Chapter 4, we develop a protocol for agents to cooperate each other by sharing their actions, and define the similar intention focused on agents' actions. In the protocol, agents build a group with agents having the same intentions or the similar intentions by sharing their actions. However, it is hard to find the common sharable actions among actions, since there is much combination in these actions. Thus, we develop the description to easily find such sharable actions, in order to solve the problem above, and explain how the processes make the target sets Moreover, we develop the descriptions of agents' actions including communications, and show the calculations of the descriptions.

Finally, Chapter 5 summarizes the results derived in this thesis.

Chapter 2

A Protocol for Multi-Threaded Processes with Choices

We use π-calculus to implement agents' multi-threaded processes, but calculating processes in π-calculus is complicated and causes deadlocks. Therefore, this chapter considers a model and a protocol for multi-threaded processes with choices to avoid deadlocks. The model and protocol distinguish between choice processes and normal processes, and divide and manage the choice processes. They make processes free from deadlocks and we use the processes to describe agents' multi-threads. In addition to this, we design the language to implement π-calculus on a computer more easily than mathematical notations of it, and implement the system by using the model and protocol.

2.1 Introduction

We developed a model and a protocol for multi-threaded processes with choices written in π-calculus. In the model and protocol, we assign each process in π-calculus to a thread. π-calculus is a process calculus which can describe a channel-based communication among distributed agents. Agents communicate each other in 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[Mun98 BD95, Bag89, BA89, DD92]

In π-calculus agents have a property to choose one process from concurrent processes. An agent gets a mutex-lock and executes the process to choose one process. The other processes are blocked by the lock and will be stopped, if the chosen process will be successfully executed. The process is easily executed, if the agent executes concurrent processes without communicating. However, when these processes are executed by the communication with agents, the conditions of mutex-lock is too complex for agents to avoid deadlocks. Hence, we adjust the conditions of the communication and define a model and a protocol to avoid deadlocks[Mun98, GA83, C.A85, EK96b, EK96a, EK97]. In addition, we design the language to implement π-calculus on a computer more easily than mathematical notations of it, and implement the system called PiEngine.

2.2 π-Calculus

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

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

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 $\overline{x}y.P_1 \mid x(z).P_2$, is thus synchronized on x, and reduced 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.

The output and input primitives of π-calculus are monadic: exactly one channel is exchanged during each communication. Polyadic π-calculus is a useful extension of

π-calculus which allows many channels exchanged during each communication[Mil91]. The syntax of polyadic π-calculus resembles that of monadic. It is as follows:

$$
\begin{array}{rcll}
P & ::= & \overline{x}[\overrightarrow{y}].P & \text{/* Output */} \\
 & | & x(\overrightarrow{z}).P & \text{/* Input */} \\
 & | & P \mid Q & \text{/* Parallel composition */} \\
 & | & (\nu x_1 \ldots x_n)P & \text{/* Restriction */} \\
 & | & P + Q & \text{/* Summation */} \\
 & | & 0 & \text{/* Nil */} \\
 & | & !P & \text{/* Replication */} \\
 & | & [x = y]P & \text{/* Matching */}
\end{array}
$$

Only output and input are different from those of monadic. $\overrightarrow{y}$ and $\overrightarrow{z}$ stand for vectors of channels including empty. The output process $\overline{x}[\overrightarrow{y}].P_1$ sends a vector of channels, $\overrightarrow{y}$, along x and then, after the output has completed, continues to be as P_1. Conversely, the input process $x(\overrightarrow{z}).P_2$ waits until a vector of values is received along x, substitutes them for the vector of bound variable $\overrightarrow{z}$. When it receives channels $\overrightarrow{y}$, it continues to be as $P_2\{\overrightarrow{y}/\overrightarrow{z}\}$. The reduction relation of π-calculus, denoted as $\rightarrow$, over processes is the least relation satisfying the following rules:

$$\text{COMM} : (\cdots + \overline{x}[\overrightarrow{y}].P) \mid (\cdots + x(\overrightarrow{z}).P) \quad \rightarrow \quad P \mid Q\{\overrightarrow{y}/\overrightarrow{z}\}$$

$$\text{PAR} : \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad\qquad \text{RES} : \frac{P \rightarrow P'}{(\nu\ x)P \rightarrow (\nu\ x)P'}$$

$$\text{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

2.3 A Model and a Protocol for Multi-Threaded Processes

In this section, we developed a model and a protocol for multi-threaded processes with choices. The processes concurrently communicate each other. First, we explain the situations of the process communication.

2.3.1 Situations of Processes' Communication

The situations of the processes' communication are divided into the two situations. One of them is that a process is now communicating with another process by using a channel. The other is that a process is now choosing one process among external choice(summation) processes.

In the former case, there are many communications among the processes and they are executed concurrently. Each communication depends on only the channel, and it is independent of the other communications executed through the other channels. Hence, it is easy to model this situation. In the later case, on the other hand, it is difficult to model the situation. Here, a process should choose one process among many processes and determines which process should be chosen by synchronizing a communication. However to synchronize a communication, a process should block other processes. For example, the process $(overlinex[1].P_1 + overlinex[2].P_2) \mid (x(z).Q_1 + x(w).Q_2)$ can reduce to either $P_1 \mid Q_1\{1/z\}$ or $P_1 \mid Q_2\{1/w\}$ or $P_2 \mid Q_1\{2/z\}$ or $P_2 \mid Q_2\{2/w\}$. If it reduces to $P_1 \mid Q_1\{1/z\}$, the process $\overline{x}[1].P_1$ should block the process $\overline{x}[2].P_2$ and $x(w).Q_2$ to communicate with the process $x(z).Q_1$. However, if the process $x(w).Q_2$ would like to communicate with the process $\overline{x}[2].P_2$, the process $\overline{x}[1].P_1$ and $x(z).Q_1$ should be blocked. Hence, the process $\overline{x}[1].P_1$ and $x(w).Q_2$ block each other. Adjusting the order of blocked processes is needed to avoid the situations above.

2.3.2 π-Model

We design the π-model that manages communications among processes and consider the orders of the processes being blocked. The π-model has elements shown below:

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

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

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

The π-model is shown in Figure 2.1.

2.3.3 The Outline of the Protocol

We give an outline of the protocol, before giving a precise of it.

Outline of Processes' Behavior: Processes perform a output process (resp. an input process) through CMs. To perform a output process (resp. an input process), processes send the information to CMs and wait for the answer from CMs.

When processes encounter a choice operator, they send the information about it to CHMs and assign each process in choice for a new thread. Each thread interacts CMs independently each other. One of these thread is selected by CHM, and the others are stopped.

Outline of Communication Managers' Behavior: Each CM has a queue that stores the communication requests from processes. CMs divide the communication requests according to the channel name. When the output request and the input request are gathered on the same channel, CMs send processes the results of communication.

Outline of Choice Managers' Behavior: Each CHM controls the choice operator. It receives the information from the processes and interact CMs to decide which thread should be remained. It send the result of selection to processes and CMs.

2.3.4 The Behavior of a Process

Conditions of a Process: A process have the variable to store its condition. The set of possible conditions of process is {*init, wait-CMout, wait-CMin, wait-CHMid, wait-CHM, wait-Res, done, stopped*}.

Figure 2.2 shows the relations among the conditions.

Transition Rules of a Process: In Table 2.1, process means the current process of the process, *aid* denotes the id of the process, *Cid* denotes the id of CHM ,*pid* denotes the id of choice process and 0 means the process is not produced from the choice (e.g. $P + Q + R$: P has id 1, Q has id 2 and R has id 3).

If processes do not meet choice, they send the information about processes to CMs(see R1, R2, R5, and R6 in Table 2.1). On the other hand, if processes meet choice, they send the information about processes to CHMs and are divided into new threads that take over the conditions (see R4 in Table 2.1). The new threads, then, wait for the answer from CHMs (see R7, R8 and R11 in Table 2.1). If one of them receives the *(resume)*, it continues the process and the other processes are stopped.

2.3.5 The Behavior of a CM

A CM manages the communications among processes on one channel. It checks output processes and input processes, and stores them to two queues. If the both queues store processes, it permits two processes, which are the top of each queue, to communicate each other. The precise transition rules of a CM is described in Table 2.2 and 2.3.

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

Transition Rules of a CM: A CM stores the requests from processes, and if the output requests and the input requests are gathered on the same channel without the choice, it permits processes to communicate(see R1–R4 in Table2.2).

The requests with the choice lead the complex rules denoted in R5–R8, R11 and R12 in Table2.2 and 2.3. If opponent to communicate dose not exist, the request is stored in the queue(see R5 and R6 in Table2.2). In the other cases, CMs ask CHMs whether the request is executable. If CHMs answer *yes*, then CMs continue the communication, otherwise CMs ignore the input. $(execute, aid, pid)$ means the process with *aid* and *pid* is executed, and $(suspend, aid, pid)$ means the process is suspended.

The requests without the choice but the opponent with choice, the rules are constructed as well as R5–R8, R11 and R12. In these cases, CMs ask CHMs whether the request is executable. If CMs receive answer *yes*, they continue the communication, otherwise CMs discard the opponent of the request and apply the rules to the request(see R9 and R10 in Table 2.2).

2.3.6 The Behavior of a CHM

Conditions of a CHM: A CHM has variables named "flag" and "queue". The variable flag stores the conditions of one choice process, it has {*suspend, try, done*}. The condition *suspend* means the choice process is suspended now, the condition *try* means the choice process is now being tried to be executed, and the condition *done* means the choice process has been already executed. The variable queue stores the processes of the choice process that are tried to execute by CMs.

Transition Rules of a CHM: CHMs manage choice processes. The basic concept is that a CHM manages one choice process. A CHM receives a choice process and numbers the processes in the choice process(see R1 in Table 2.4). After numbering the processes, the CHM waits for requests from CMs as the first phase(see R2, R3 or R7 in Table 2.4). If the CHM receives the requests from CMs, the CHM behaves as follows:

if the flag is *suspend*.

The CHM allows the requested process to be executed and sends the permission to execute it (sends the signal "*yes*"). It, then, changes the flag to *try* (see R2 in Table 2.4).

else if the flag is *try*.

The CHM suspends the process and stores it to the queue. It, then, sends the CM the condition that other CM is now trying to execute the choice process (see R3 in Table 2.4).

else

The CHM sends the CM the signal to stop the process (see R7 in Table 2.4).

After sending the signal "*yes*" to the CM, the CHM waits for the result of executing the process in the CM and behaves as follows:

if the CHM receives the signal that the CM succeeds to execute the process.

The CHM sends that the process can be executed and the other processes in the choice process should be stop, then, changes the flag to *done* (see R6 in Table 2.4).

else

The CHM suspends the process. If the queue is not empty, the CHM checks the top process in the queue and sends the signal "*yes*" to the CM that owns the process. (see R4 and R5 in Table 2.4).

Figure 2.3 shows the relations among the conditions of a CHM, Table 2.4 describes the transition rules of a CHM, and Figure 2.4 shows the behavior of a CHM. The top figure in Figure 2.4 shows the CHM numbers the processes. The "Request" and "Answer" mean the first phase (R2, R3 and R7 in Table 2.4), and the "Result" means the second phase(R4, R5 and R6 in Table 2.4) in the bottom figure in Figure 2.4.

2.4 Freeing Processes from Deadlocks

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

We consider four cases to show the freedom from the deadlocks,

1. There is no choice process and only one paired processes(input and output) use a channel.
2. There is no choice process and processes use channels. It means some input and output processes 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.

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

1. By R2 in Table 2.2, the id of A_{in} is registered for `in-x`, and then the condition of A_{in} is changed to *wait-CMin*.
2. If A_{out} is registered for a CM, by R3 in Table 2.2, the values in A_{out} are output to the process indicated by the id in the top of `in-x`. Then, A_{in} receives the values from the CM and executes next process by R9 in Table 2.1. The condition of A_{out} receives the results from the CM and executes next process by R12 in Table 2.1.

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 for CMs.

(a) If m == 1 (the length of `in-x` is 1) then

This condition is same as the case 1. Thus, 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 Table 2.2, the values in the output process are sent to the process indicated by `id` in `in-x`. The output process proceeds to the next process through the condition *wait-CMout* by R12 in Table 2.1. The process, which receives the values by R9 in Table 2.1, proceeds to the next process.
The process in the top on `in-x` and the output process communicates 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 for a CHM, by R2 and R3 in Table 2.1. Each process proceeds independently but has the condition *wait-CHM*. The choice process $B_1 + B_2 + \cdots + B_n$ is executed like as the choice process $A_1 + A_2 + \cdots + A_n$, but uses a different CHM. There are many combination to execute these processes. Before explaining it, we explain the actions of CHMs.

A CHM receives the processes and commits them to memory (see R1 in Table 2.4). It has no knowledge of a channel used by the processes and checks the process that is requested to execute by CMs (see R2 and R3 in Table 2.4). The requests means that the process would like to use the channel and can be executed if the CHM answers *yes*. When the CHM receives the first request, it returns the answer *yes*(see R2 in Table 2.4). When the CHM receives the second request or more requests, it store the requests in the queue and checks the head request in the queue if the first request cannot be executed (see R3, R4 and R5 in Table 2.4).

We consider the cases that the process A_1 and B_1 communicate each other and the other processes are discarded. 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 each other.

   We explain the first case in them.

   The process A_1 registers for a CM by R5 or R6 in Table 2.2. The process B_1 registers for the CM and the CM requests CHMs to execute A_1 and B_1 by

R11 or R12 in Table2.3. CHMs answer *yes* to the CM, because the requests is the first request for each CHM(see R2 in Table2.4).

The CM permits A_1 and B_1 to communicate by R11 or R12 in Table2.3 and CHMs send *stop* to the other processes by R6 in Table2.4.

If the other processes register for a CM before receiving the signal *stop*, CHMs answer *no* to the CM by R7 in Table2.4.

2. $P_{AB} \rightarrow A_1 \rightarrow P_{A'B'} \rightarrow B_1 \rightarrow$ the other processes

   or

   $P_{AB} \rightarrow B_1 \rightarrow P_{A'B'} \rightarrow A_1 \rightarrow$ the other processes

   where $P = \{A_i, B_j | 2 \leq i, j \leq n\}$, $P_{AB} = \{A_i, B_j | A_i, B_j \in P \text{ and } i \neq j\}$, $P_{A'B'} = \{A_i, B_j | A_i, B_j \in P - \text{pair}(P_{AB})\}$ and pair(P) means the set of processes that can communicate with the processes in the set P. For example, let A_1 and A_2 communicate with B_1 and B_2, respectively, pair$(\{A_1, A_2\}) = \{B_1, B_2\}$

   We explain the first case in them. It means the some processes registered before the process A_1 registering. Then, when the process B_1 registers for a CM before the pair to the processes that have already registered, the CM requests to CHMs to execute it and CHMs answer *yes* to the CM. When the CM receives the answer *yes*, it sends CHMs the signal of execution(see R11 and R12 in Table 2.3). Under this condition, if the pair to the processes registers for a CM before CHMs receive the signal from the CM, the CM requests to CHMs and the CHM blocks this process(see R4 in Table 2.4). The processes A_1 and B_1 communicate each other in spite of blocking other processes, because the CM has already sent the signal of the execution to CHMs and CHMs send the permeations to execute to the processes (in this case A_1 and B_1).

   In this case CHMs block a process, but it does not generate a deadlock, The blocked processes have no chance to be execute and the blocks have no influence, because blocking the processes are generated by determining which processes should be executed.

In this case, we consider only two choice processes using different channels. However, if there are many choice processes, they do not generate a deadlock. Because the CM considers the processes according to their channels and one CHM manages one choice process, and it blocks the processes only if the CHM determines which process is executed.

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 the suspension(see R11 and R12 in Table 2.3).

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

If all processes have finished to register for CHMs, the conditions that CMs send CHMs the signal of the suspension are generated by some orders of requests.

The orders which generate the suspensions are as follows:

The CM sends the requests to execute two pairs $\{A_1, B_1\}$ and $\{B_n, N\}$(see R11 and R12 in Table 2.3). Under this condition, if the CHM managing $A_1 + A_2$ permits to execute A_1 and the CHM managing $B_1 + B_n$ permits to execute B_n, and the communication between B_n and N succeeds. Thus, executing the process B_1 is impossible and the CHM answers *no* to the CM(see R11 and R12 in Table 2.3), and then the CM sends the signal of the suspension to the CHM managing $A_1 + A_2$. The CHM removes the process A_1 from the queue and waits for a new request from CMs.

In this case, the process A_1 blocks other process A_2 in the same choice process, but the process A_1 releases the block if the process B_1 cannot communicate.

Therefore, it does not generate a deadlock.

If the number of the process in one choice process, CHMs consider only the first process in the queue storing the requests from CMs. The condition is, therefore, the same as this condition.

We consider the all possible condition and show the protocol frees choice processes from deadlocks. Hence, by using the protocol, we avoid to deadlocks in π-calculus.

2.5 Implementing the Interpreter for π-Calculus

In order to implement an interpreter for π-calculus more easily on computers, we design a primitive language instead of using the mathematical notations of π-calculus.

2.5.1 The Syntax of our Language

Table 2.5 shows the syntax of definitions of a processes, and its corresponding the definitions in π-calculus.

Example 2.5.1 *We show simple π-calculus expressions and the program in our language corresponding to the expressions:*

π-calculus expression	**Program**
$A \mid B$	`A	B`
$A \stackrel{def}{=} K + L + M$	`A` $::=$ $K + L + M$
$B \stackrel{def}{=} X + Y + Z$	`B` $::=$ $X + Y + Z$
$K \stackrel{def}{=} \overline{a}[b]$	`K` $::=$ (`out` $a\ b$).@
$L \stackrel{def}{=} c(d)$	`L` $::=$ (`in` $c\ d$).@
$M \stackrel{def}{=} \overline{e}[f]$	`M` $::=$ (`out` $e\ f$).@
$X \stackrel{def}{=} a(x)$	`X` $::=$ (`in` $a\ x$).@
$Y \stackrel{def}{=} \overline{c}[y]$	`Y` $::=$ (`out` $c\ y$).@
$Z \stackrel{def}{=} e(z)$	`Z` $::=$ (`in` $e\ z$).@

The result to execute this program is shown in the next subsection.

2.5.2 The Implementation of the Interpreter

We implement an interpreter for our Language in JAVA, called PiEngine by using the π-model and the protocol. In this subsection, we verify the correctness of the interpreter by running Example 2.5.1.

This interpreter is implemented in JAVA, because JAVA provides multi-threaded programming. The interpreter assigns each process in our language to one threaded and executes these processes concurrently. Each channel is regarded as one instance of a class accessed by some processes. Each threaded process can access concurrently some instance of the class. Hence, JAVA is an appropriate platform to implement the interpreter for our language.

Before executing the above example, we illustrate the actions of the program in Figure 2.5.

Figure 2.6 and 2.7 show that the messages on windows are the results executing the program of Example 2.5.1. We show only two cases of the results, because the choice process make many different results.

In Figure 2.6 and 2.7, each window shows one threaded process in multi-threaded processes. The message "ok $\langle A \rangle \ldots$" means the process named "A" is executed and "ok $\langle M \rangle \ldots$" means the process "A" chooses the process named "M".

In the case 1, the process A chooses the process M and the process B chooses the process Z. In the case 2, the process A chooses the process K and the process B chooses the process X. Therefore, the interpreter system named PiEngine can execute programs and can simulate the choice process of π-calculus.

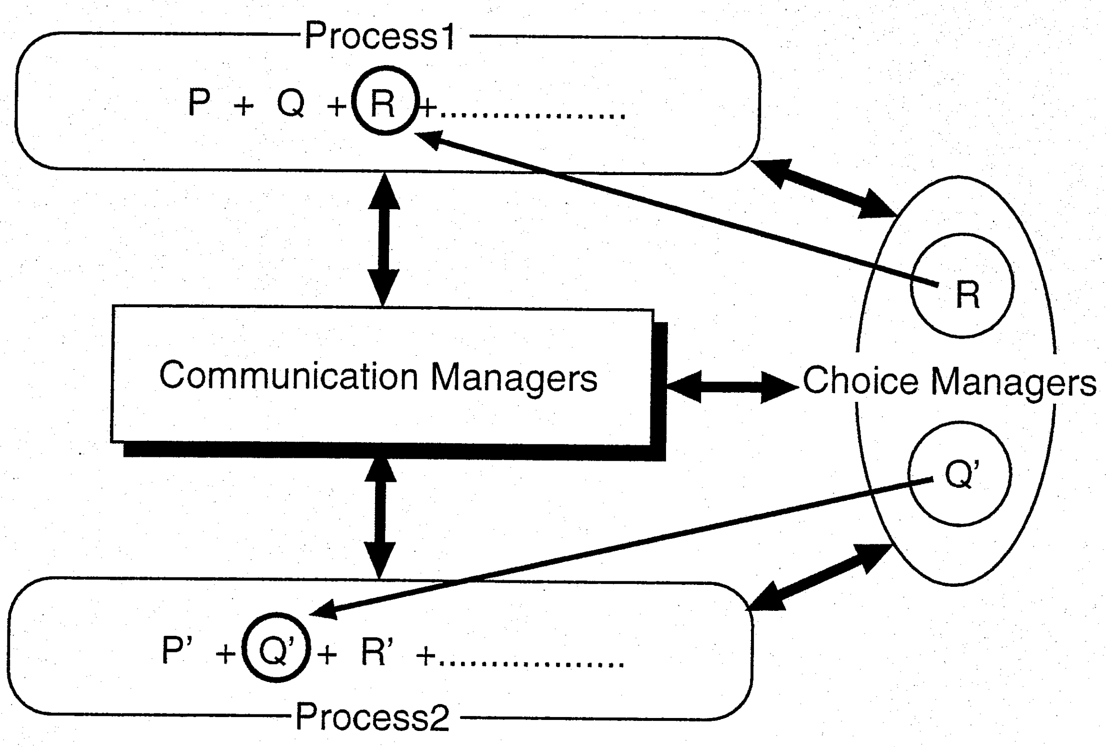
2.6 Conclusions

We have developed the model named π-model and the protocol for multi-threaded processes with choices written in π-calculus. In the model and protocol, we have assigned each process in π-calculus to a thread, and we have defined the three elements:

Processes, Communication Managers(CMs) and Choice Managers(CHMs). A process is a basic unit to control concurrently executed threads in our concurrent and distributed system. CMs manage communication requests along channels from processes, and CHMs manage choice processes in processes.

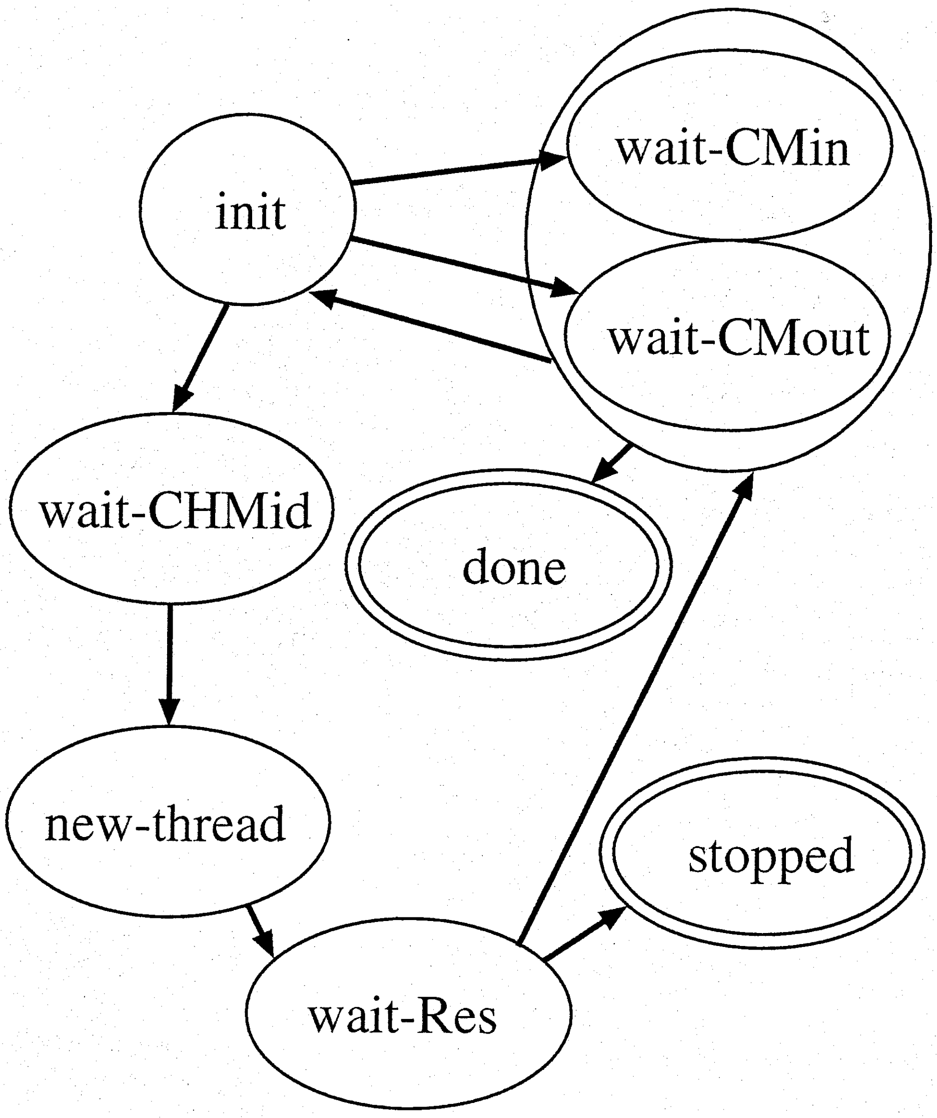
We have shown why the protocol frees the processes from the deadlocks. If the processes have no choice process, any process do not be blocked. If the processes have choice processes, CHMs order the requests from CMs and manage the block to avoid deadlocks. Hence, the protocol frees the processes from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

We have designed a primitive language instead of using the mathematical notations of π-calculus, and implemented the system in JAVA called PiEngine by using the π-model and the protocol.


Process1
P + Q + R +..................
R
Communication Managers
Choice Managers
Q'
P' + Q' + R' +..................
Process2

Each element communicates with each other. If there is no choice process in the processes, the processes and CMs don't communicate with CHMs.

Figure 2.1: π-Model



The arrows mean the transitions.The circles mean the conditions. The *init* is the first condition. The *done* or *stopped* is the final condition.

Figure 2.2: The Conditions of a Process

Table 2.1: The Transition Rules of a Process

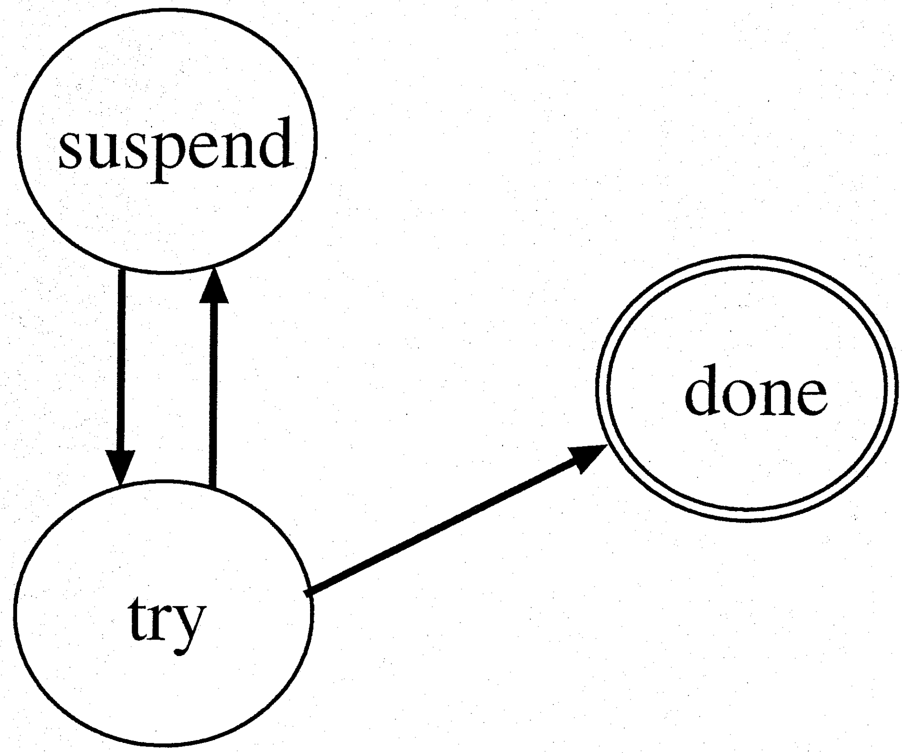
Rules	Conditions	Inputs	Processes	Next Conditions	Outputs	Other Actions
R1	*init*	-	$\overline{x}[\overrightarrow{y}]$	*wait-CMout*	Sending $(in, x, \overrightarrow{y}, aid, 0, 0)$ to a CM.	-
R2	*init*	-	$x(\overrightarrow{z})$	*wait-CMin*	Sending $(out, x, \overrightarrow{z}, aid, 0, 0)$ to a CM.	-
R3	*init*	-	$P \equiv Q + \ldots$	*wait-CHMid*	Sending (P, aid) to a CHM.	The current process is not changed.
R4	*wait-CHMid*	Receiving (Cid) from a CHM.	$P \equiv Q + \ldots$	*wait-CHM*	-	Each process in the choice is divided into one process and each process has the condition *wait-CHM.*
R5	*wait-CHM*	-	$\overline{x}[\overrightarrow{y}]$	*wait-Res*	Sending $(in, x, \overrightarrow{y}, aid, Cid, pid)$ to a CM.	-
R6	*wait-CHM*	-	$x(\overrightarrow{z})$	*wait-Res*	Sending $(out, x, \overrightarrow{z}, aid, Cid, pid)$ to a CM.	-
R7	*wait-Res*	Receiving $(resume)$ from a CHM.	$\overline{x}[\overrightarrow{y}]$	*wait-CMout*	-	This thread is selected to be executed.
R8	*wait-Res*	Receiving $(resume)$ from a CHM.	$x(\overrightarrow{z})$	*wait-CMin*	-	This thread is selected to be executed.
R9	*wait-Res*	Receiving $(stop)$ from a CHM.	-	*stopped*	-	This thread is stopped.
R10	*wait-CMout*	Receiving $(output)$ from a CM.	$\overline{x}[\overrightarrow{y}]$	*done* : if there is no next process *init* : otherwise	-	The condition *done* means the process is finished.
R11	*wait-CMin*	Receiving $(\overrightarrow{v})$ from a CM.	$x(\overrightarrow{z})$	*done* : if there is no next process *init* : otherwise	-	The condition *done* means the process is finished.

Table 2.2: The Transition Rules of a CM – 1/2

Rules	Conditions	Inputs	Next Conditions	Outputs
R1	in-x $= \emptyset$	Receiving $(out, x, \vec{z}, aid, 0, 0)$ from a process.	out-x $=$ out-x $+ (aid, \vec{z}, 0, 0)$	-
R2	out-x $= \emptyset$	Receiving $(in, x, \vec{y}, aid, 0, 0)$ from a process.	in-x $=$ in-x $+ (aid, 0, 0)$	-
R3	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow} (aid', 0, 0)$ $+$ in-x'	Receiving $(out, x, \vec{z}, aid, 0, 0)$ from a process.	in-x = in-x'	Sending $(output)$ to aid and $(\vec{z})$ to aid'.
R4	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow} (aid', \vec{z}, 0, 0)$ $+$ out-x'	Receiving $(in, x, \vec{y}, aid, 0, 0)$ from a process.	out-x = out-x'	Sending $(output)$ to aid' and $(\vec{z})$ to aid.
R5	in-x $= \emptyset$	Receiving $(out, x, \vec{z}, aid, Cid, pid)$ from a process.	out-x $=$ out-x $+ (aid, \vec{z}, Cid, pid)$	-
R6	out-x $= \emptyset$	Receiving $(in, x, \vec{y}, aid, Cid, pid)$ from a process.	in-x $=$ in-x $+ (aid, Cid, pid)$	-
R7	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow} (aid', 0, 0)$ $+$ in-x'	Receiving $(out, x, \vec{z}, aid, Cid, pid)$ from a process.	Sending (aid, pid) to Cid and if receiving yes from Cid then:	
			in-x = in-x'	Sending $(output)$ to aid $(\vec{z})$ to aid' $(execute, aid, pid)$ to Cid.
			if receiving no from Cid then:	
			Ignore this input.	-
R8	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow} (aid', \vec{z}, 0, 0)$ $+$ out-x'	Receiving $(in, x, \vec{y}, aid, Cid, pid)$ from a process.	Sending (aid, pid) to Cid and if receiving yes from Cid then:	
			out-x = out-x'	Sending $(output)$ to aid' $(\vec{z})$ to aid' $(execute, aid, pid)$ to Cid.
			if receiving no from Cid then:	
			Ignore this input.	-
R9	in-x $\neq \emptyset$ in-x $\overset{div}{\rightarrow} (aid', Cid, pid)$ $+$ in-x'	Receiving $(out, x, \vec{z}, aid, 0, 0)$ from a process.	Sending (aid, pid) to Cid and if receiving yes from Cid then:	
			in-x = in-x'	Sending $(output)$ to aid $(\vec{z})$ to aid'.
			if receiving no from Cid then:	
			in-x = in-x' Apply these rules again.	-
R10	out-x $\neq \emptyset$ out-x $\overset{div}{\rightarrow} (aid', \vec{z}, Cid, pid)$ $+$ out-x'	Receiving $(in, x, \vec{y}, aid, 0, 0)$ from a process.	Sending (aid, pid) to Cid and if receiving yes from Cid then:	
			out-x = out-x'	Sending $(output)$ to aid' $(\vec{z})$ to aid.
			if receiving no from Cid then:	
			out-x = out-x' Apply these rules again.	-

Table 2.3: The Transition Rules of a CM – 2/2

Rules	Conditions	Inputs	Next Conditions	Outputs
R11	in-x $\neq \emptyset$ in-x $\xrightarrow{div}$ (aid', Cid', pid') + in-x'	Receiving $(out, x, \vec{z}, aid, Cid, pid)$ from a process.	Sending (aid, pid) to Cid and (aid', pid') to Cid' and if receiving *yes* from Cid and Cid'then:	
			in-x = in-x'	Sending $(output)$ to aid $(\vec{z})$ to aid' $(execute, aid, pid)$ to Cid $(execute, aid', pid')$ to Cid'.
			if receiving *yes* from Cid and *no* from Cid' then:	
			in-x = in-x' Applying these rules again.	-
			if receiving *no* from Cid and *yes* from Cid' then:	
			Ignoring this input.	Sending $(suspend, aid', pid')$ to Cid'.
			if receiving *yes* from Cid and *try* from Cid' then:	
			Waiting for a while and applying these rules again.	Sending $(suspend, aid, pid)$ to Cid.
			if receiving *try* from Cid and *yes* from Cid' then:	
			Waiting for a while and applying these rules again.	Sending $(suspend, aid', pid')$ to Cid'.
			if receiving *no* from Cid and Cid' then:	
			in-x = in-x' Ignoring this input.	-
R12	out-x $\neq \emptyset$ out-x $\xrightarrow{div}$ (aid', Cid', pid') + out-x'	Receiving $(in, x, \vec{y}, aid, Cid, pid)$ from a process.	Sending (aid, pid) to Cid and (aid', pid') to Cid' and if receiving *yes* from Cid and Cid' then:	
			out-x = out-x'	Sending $(output)$ to aid' $(\vec{z})$ to aid $(execute, aid, pid)$ to Cid $(execute, aid', pid')$ to Cid'.
			if receiving *yes* from Cid and *no* from Cid' then:	
			out-x = out-x' Applying these rules again.	-
			if receiving *no* from Cid and *yes* from Cid' then:	
			Ignoring this input.	Sending $(suspend, aid', pid')$ to Cid'.
			if receiving *yes* from Cid and *try* from Cid' then:	
			Waiting for a while and applying these rules again.	Sending $(suspend, aid, pid)$ to Cid.
			if receiving *try* from Cid and *yes* from Cid' then:	
			Waiting for a while and applying these rules again.	Sending $(suspend, aid', pid')$ to Cid'.
			if receiving *no* from Cid and Cid' then:	
			out-x = out-x' Ignoring this input.	-

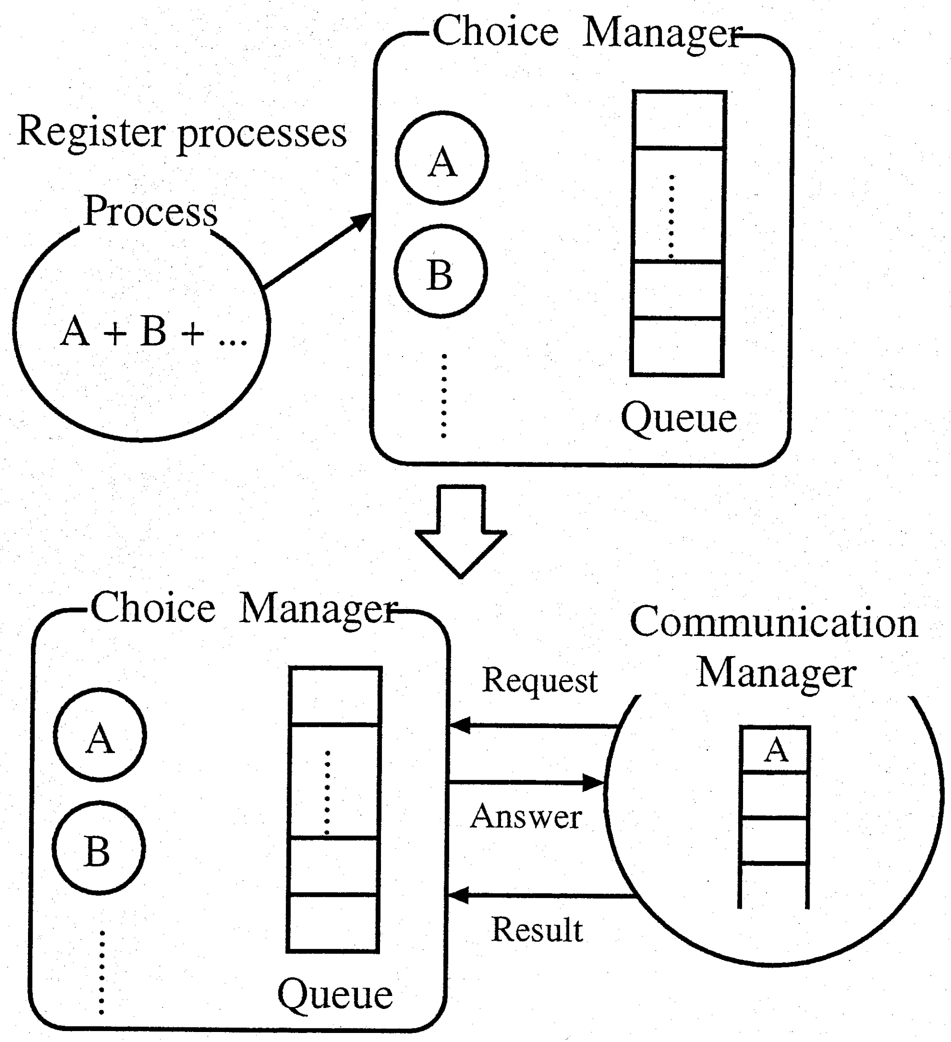


The last condition *done* is reached through the condition *try*.

Figure 2.3: The Conditions of a CHM

Table 2.4: The Transition Rules of a 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)	Send *try* to CM.	-
R4	flag = *try* queue = $\emptyset$	Get $(suspend, aid, pid)$ from CM.	flag = *suspend*	-	-
R5	flag = *try* queue $\overset{div}{\to} (aid', pid')$ +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.	-

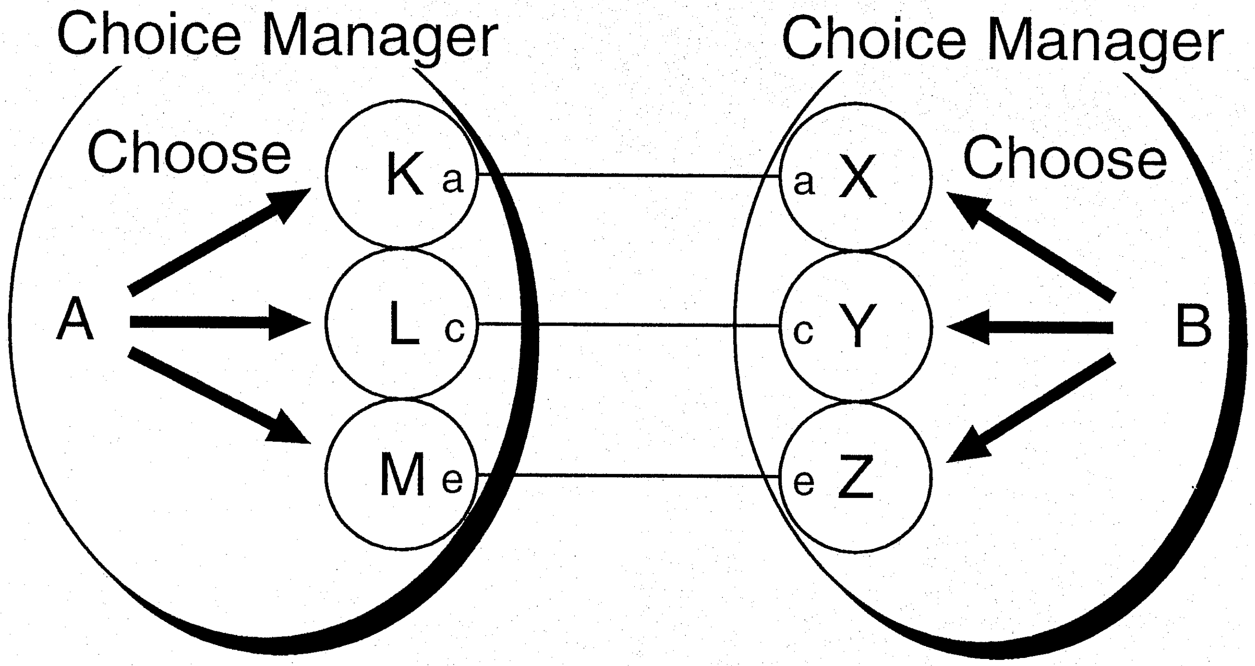


The choice process $A + B + \ldots$ is registered for the Choice Manager, and it communicates with the Communication Manager.

Figure 2.4: The Behavior of a CHM

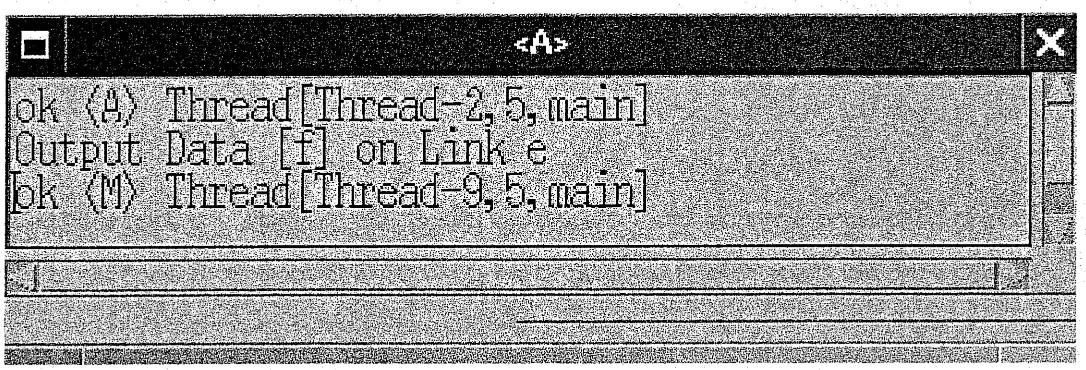
Table 2.5: Comparing π-Calculus with Our Language

Functions	Polyadic π-calculus	Our Language
Output	$\overline{x}[y_1 \ldots y_n].P$	$(\texttt{out}\ x\ y_1 \ldots y_n).P$
Input	$x(z_1 \ldots z_n).P$	$(\texttt{in}\ x\ z_1 \ldots z_n).P$
Parallel composition	$P_1 \mid \ldots \mid P_n$	$P_1 \mid \ldots \mid P_n$
Restriction	$(\nu\ x_1 \ldots x_n)P$	$(\texttt{new}\ x_1 \ldots x_n).P$
Summation	$P_1 + \cdots + P_n$	$P_1 + \cdots + P_n$
Nil	0	@
Replication	!P	$!P$
Matching	$[x = y]P$	$[x = y].P$

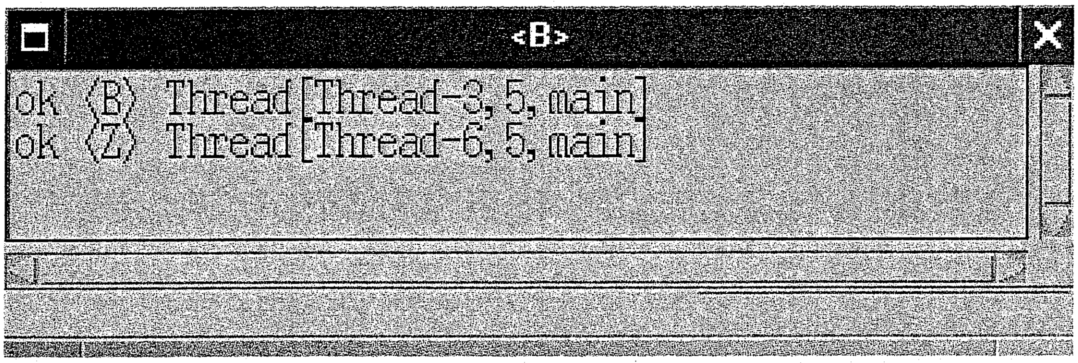


The process A has the choice process $K + L + M$, and the process B has the choice process $X + Y + Z$.The choice managers choose the pair (K, X) or (L, Y) or (M, Z).

Figure 2.5: The Choice Processes in Example 2.5.1



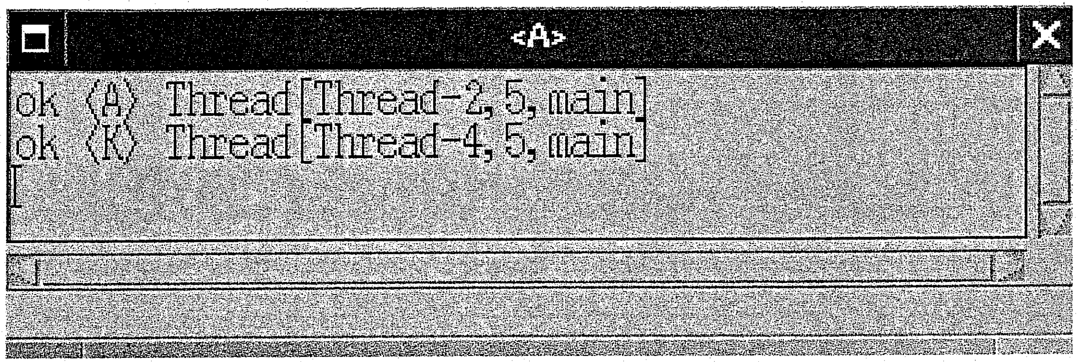
```
ok (A) Thread[Thread-2,5,main]
Output Data [f] on Link e
ok (M) Thread[Thread-9,5,main]
```



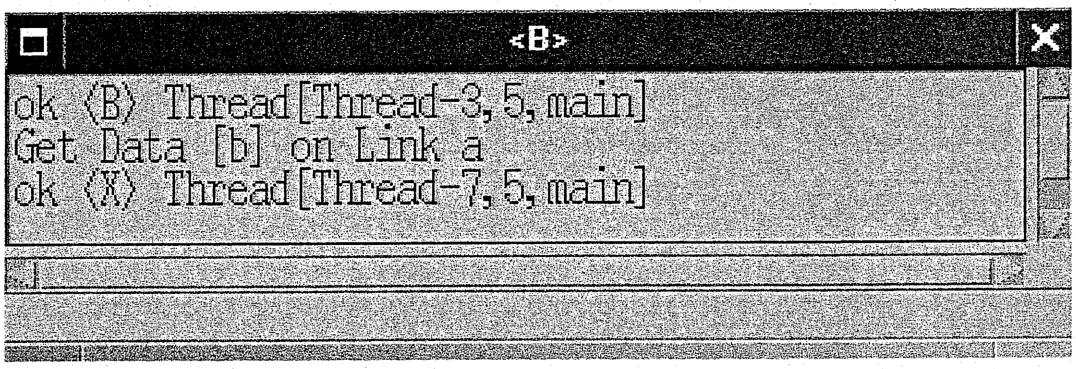
```
ok (B) Thread[Thread-3,5,main]
ok (Z) Thread[Thread-6,5,main]
```

It shows the process M is chosen from the process A and the process Z is chosen from the process B.

Figure 2.6: The Results of Executing Processes in Example 2.5.1



```
ok <A> Thread[Thread-2,5,main]
ok <K> Thread[Thread-4,5,main]
```



```
ok <B> Thread[Thread-3,5,main]
Get Data [b] on Link a
ok <X> Thread[Thread-7,5,main]
```

It shows the process K is chosen from the process A and the process X is chosen from the process B.

Figure 2.7: The Results of Executing Processes in Example 2.5.1

Chapter 3

An Agent Model for a Dynamically Changing Environment

We develop an agent model to use descriptions for agents' plans and methods based on π-calculus[Mil91, MPW92, BD95]. First, we develop a description for agents' plans. The plans described in π-calculus can be changed dynamically while it is executing, because π-calculus provides dynamically changing structures. Secondly, we develop a description for agents' methods that refer their parameters through channels in π-calculus. Thus, agents can dynamically change the references from the methods to the parameters. Next, we develop an agent model to use the descriptions. The agents can, therefore, change their plans and methods to adapt to the environment around them by themselves by using these property. This property is important to agents, called *reflection*. Finally, we show two experiments: a tracing problem and a fire-world problem.

3.1 Introduction

Agents mean the object which has some of the following properties:

- Autonomy : Agents decide to act according to their knowledge.
- Proactivity : Agents begin to act by themselves.

- Sociality : Agents communicate with the other agents or human to act co-operatively.

- Reactivity : Agents change rapidly their action to adapt to the change of their environment.

The autonomy and proactivity are more important than the others to define agents [Sho93, INH+98, IIDI99b, IIDI99a].

In order to act autonomously and proactively, agents need to observe their environment and make the plan to satisfy their goals. A plan for an agent consists of a series of processes that the agent will execute. The one process in the plan is called "method". In other words, an agent executes methods according to the *plan* (called *executing plan*). *Planning* means that an agent constructs a plan for agents. The planning is an important problem for autonomous agent systems. However, the environment around agents is not the static environment and dynamically changed in real time. Hence, agents in the environment have the following problems:

1. The environment is the real time world.
   The delay to calculate the parameters, which are referred to by methods, causes the delay to execute the methods.

2. The environment is the dynamic world.
   The agents need function to change the reference to parameters, and cannot always succeed in executing plans due to unforecastable changes in the environment.

The solution for the first problem is that agents should concurrently and independently execute the calculation of the parameters and the methods. The second problem is difficult, because formalizing methods that can change the reference to parameters is difficult and the agents should dynamically reform the plans to succeed in executing them in the new environment [JZ97, SN98, TK95, AT97, Mor97]. We called these property called *reflection* and the agent with reflection are called reflective agent.

For example, assume that there are two agents in a room. One agent (called agent1) traces another agent (called agent2). Each agent has the same ability about the movement. Agent1 can check the position of agent2 by using the sensor and form the plan to trace agent2. Agent2 moves to the position which is away from the agent1 at the rate X, otherwise to the random position. If agent1 can use sensor for all times, it can choice the best route to trace agent2. But if agent1 only uses the sensor to form the plan, this plan is not suitable to trace agent2. Hence, agent1 should have the function *reflection*.

Our solution of these problems is based on π-calculus. π-calculus is a process calculus, which is able to describe dynamically changing networks of concurrent processes.

First, we develop a description for agents' plans by using π-calculus. The agents can change the plans described in π-calculus even when they are executing the plans. Secondly, we develop a description for agents' methods. In the description, the parameters and the methods are regarded as the processes in π-calculus. Hence, the parameters and the methods are concurrently and independently executed. The references from the methods to the parameters are regarded as the networks in π-calculus. Hence, the references can be changed dynamically. Next, we define an agent model to use the descriptions for plans and methods. It has the processor for π-calculus(PiEngine in Chapter 2) and the functions to behave like an agent. Finally, we make two experiments. One of them is that an agent traces and catches another agent like as the above example. Another agent is that an agent searches the route to escape from the fire-world. In the fire-world, the fire randomly moves and spreads, agents can not predict how the environment changes.

3.2 Dynamically Changing Plans in π-calculus

In this section we develop a description for agents' plans by using polyadic π-calculus. The description is basically a set of list-typed expressions. Most importantly, the descriptions of a plan can be modified dynamically by changing the connections of the list according to the requests of an agent.

In a static environment, an agent can succeed in executing plans. However, in a dynamically changing environment, the plans become unexecutable. Because the plans are usually not changed until the executing them is finished. Thus, they should be changed in executing them. In order to provide this function for plans, an agent should have the function *reflection*. Therefore, we develop the plans written in polyadic π-calculus.

We define the process for calling the method from π-calculus in Definition 3.2.1.

Definition 3.2.1 Process of Calling Method

$$Call \overset{def}{=} \mathtt{exec}(\mathrm{method}, \mathrm{method}_{\mathrm{end}}).Call(method).\overline{\mathtt{method}_{\mathrm{end}}}$$

$$Call(method) = \begin{cases} \text{It doesn't call no method} & \text{If } method == \text{“nil”} \\ \text{It calls the method named } method & \text{Otherwise} \end{cases}$$

where the `typewritten fonts` words represent the channel name; and *method* represents the name of the method to be executed; and $method_{end}$ represents the name which checks the finishing to execute the method; and the name `nil` denotes the end of plan and doesn't call any methods.

Definition 3.2.2 Basic element

$$\begin{aligned} &\mathrm{Element}(\mathrm{previous}, \mathrm{method}, \mathrm{next}) \\ &\overset{def}{=} \ \underline{\mathtt{previous}}. \\ &\quad (\overline{\mathtt{change}}[\mathrm{method}].\mathtt{change}(\mathrm{method}).\overline{\mathtt{exec}}[\mathrm{method}, \mathrm{method}_{\mathrm{end}}].\mathtt{method}_{\mathrm{end}}.\overline{\mathtt{next}} \\ &\quad + \ \mathtt{execute}.\overline{\mathtt{exec}}[\mathrm{method}, \mathrm{method}_{\mathrm{end}}].\mathtt{method}_{\mathrm{end}}.\overline{\mathtt{next}}) \end{aligned}$$

where the name “previous” and “next” represent channels to the other elements.

We define a plan by using the definition 3.2.2. On the assumption that method_1, method_2, …, method_{n-1}, method_n should be executed, the description of the plan is as follows:

Definition 3.2.3 Definition of a plan

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{Plan}(\mathrm{method}_1, \mathrm{method}_2, \ldots, \mathrm{method}_{n-1}, \mathrm{method}_n)} \\
\overset{def}{=} & (\nu\ \mathrm{n}_1\ \mathrm{n}_2\ \ldots \mathrm{n}_{n-1}\ \mathrm{n}_n) & \mathrm{Element}(\mathrm{start}, \mathrm{method}_1, \mathrm{n}_1) \\
 & & \mid\ \mathrm{Element}(\mathrm{n}_1, \mathrm{method}_2, \mathrm{n}_2) \\
 & & \vdots \\
 & & \mid\ \mathrm{Element}(\mathrm{n}_{i-1}, \mathrm{method}_i, \mathrm{n}_i) \\
 & & \vdots \\
 & & \mid\ \mathrm{Element}(\mathrm{n}_{n-2}, \mathrm{method}_{n-1}, \mathrm{n}_{n-1}) \\
 & & \mid\ \mathrm{Element}(\mathrm{n}_{n-1}, \mathrm{method}_n, \mathrm{nil}) \\
 & & \mid\ \mathtt{nil}\ \mid\ \mathrm{Call}
\end{array}
$$

where the last element `nil` denotes the end of plan.

3.2.1 Executing and Changing a Plan

An agent should execute the methods in a plan at a suitable timing. First, we, therefore, define the process changing the plan into an executable condition. Secondly, we define the process requesting to execute the method in a plan.

Definition 3.2.4 Requesting the start of a plan

$$\mathrm{Start} \overset{def}{=} \overline{\mathtt{start}}$$

Definition 3.2.5 Requesting the execution of a method

$$\mathrm{Execute} \overset{def}{=} \overline{\mathtt{execute}}$$

This definition is simple. This process named "Execute" requests a plan to execute one method. Therefore, if a plan has n methods, n processes named "Execute" are needed to finish the plan. However, this property is useful to change a plan.

Next, we define three processes to change a plan: "Substitute", "Add" and "Delete".

Definition 3.2.6 Requesting the substitution of a method

$$\mathrm{Substitute}(\mathrm{method}') \overset{def}{=} \mathtt{change}(\mathrm{method}).\overline{\mathtt{change}}[\mathrm{method}']$$

Definition 3.2.7 Requesting the addition of a method

$$\mathrm{Add}(\mathrm{method}') \overset{def}{=} \mathtt{change}(\mathrm{method}).\overline{\mathtt{exec}}[\mathrm{method}', \mathrm{method}'_{\mathrm{end}}].\mathtt{method}'_{\mathrm{end}}.\overline{\mathtt{change}}[\mathrm{method}]$$

Definition 3.2.8 Requesting the deletion of a method

$$\mathrm{Delete} \overset{def}{=} \mathtt{change}(\mathrm{method}).\overline{\mathtt{change}}[\mathrm{nil}]$$

The usages of them is same as the request named "Execute". This property simplifies to implement the system.

3.3 The Properties of a Plan

We show four properties as Theorems 3.3.1, 3.3.2, 3.3.3 and 3.3.4, by using Definitions 3.2.5, 3.2.6, 3.2.7 and 3.2.8,

Theorem 3.3.1 Execution of method

If $\mathbf{Plan}(\mathbf{method}_1, \ldots, \mathbf{method}_n)$ *exists and it is performed in the parallel composition with the process named "*Execute*" then: the method named method*$_1$ *is called and the plan is changed to* $\mathbf{Plan}(\mathbf{method}_2, \ldots, \mathbf{method}_n)$.

Executing the plan shown in Definition 3.2.3 is as follows:

$$\mathrm{Plan}(\mathrm{method}_1, \ldots, \mathrm{method}_n) \mid \mathrm{Execute} \mid \overline{\mathtt{start}}$$

$$\rightarrow \quad \mathrm{Plan}(\mathrm{method}_2, \ldots, \mathrm{method}_n) \mid \mathit{Call}(\mathit{method}_1).\overline{\mathtt{method}_{1_{\mathrm{end}}}} \mid \mathtt{method}_{1_{\mathrm{end}}}.\overline{\mathtt{n}_1}$$

If $\mathbf{Call}(\mathbf{method}_1)$ *is successfully executed, then*

$$\mathrm{Plan}(\mathrm{method}_2, \ldots, \mathrm{method}_n) \mid \mathit{Call}(\mathit{method}_1).\overline{\mathtt{method}_{1_{\mathrm{end}}}} \mid \mathtt{method}_{1_{\mathrm{end}}}.\overline{\mathtt{n}_1}$$
$$\rightarrow \quad \mathrm{Plan}(\mathrm{method}_2, \ldots, \mathrm{method}_n) \mid \overline{\mathtt{n}_1}$$

Theorem 3.3.2 Substitution of method

If $\mathbf{Plan}(\mathbf{method_1}, \ldots, \mathbf{method_n})$ *exists and it is performed in the parallel composition with the process named "*$\mathbf{Substitute}(\mathbf{method_{new}})$*" then: the method named* $\mathbf{method_{new}}$ *is called and the plan is changed to* $\mathbf{Plan}(\mathbf{method_2}, \ldots, \mathbf{method_n})$.

The substitution of a method for the method in the plan in Definition 3.2.3 is as follows:

$$\begin{aligned} &\mathrm{Plan}(\mathrm{method_1}, \ldots, \mathrm{method_n}) \mid \mathrm{Substitute}(\mathrm{method_{new}}) \mid \overline{\mathtt{start}} \\ \rightarrow\ &\mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \mathit{Call}(\mathit{method_{new}}).\overline{\mathtt{method_{1_{end}}}} \mid \mathtt{method_{1_{end}}}.\overline{\mathtt{n_1}} \end{aligned}$$

if $\mathrm{Call}(\mathbf{method_{new}})$ *is successfully executed, then*

$$\begin{aligned} &\mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \mathit{Call}(\mathit{method_{new}}).\overline{\mathtt{method_{1_{end}}}} \mid \mathtt{method_{1_{end}}}.\overline{\mathtt{n_1}} \\ \rightarrow\ &\mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \overline{\mathtt{n_1}} \end{aligned}$$

Theorem 3.3.3 Addition of method

If $\mathbf{Plan}(\mathbf{method_1}, \ldots, \mathbf{method_n})$ *exists and it is performed in the parallel composition with the process named "*$\mathbf{Add}(\mathbf{method_{new}})$*" then: the methods named* $\mathbf{method_{new}}$ *and* $\mathbf{method_1}$ *are called and the plan is changed to* $\mathbf{Plan}(\mathbf{method_2}, \ldots, \mathbf{method_n})$.

Adding a method to the plan in Definition 3.2.3 is as follows:

$$\begin{aligned} &\mathrm{Plan}(\mathrm{method_1}, \ldots, \mathrm{method_n}) \mid \mathrm{Add}(\mathrm{method_{new}}) \mid \overline{\mathtt{start}} \\ \rightarrow\ &\quad \mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \mathit{Call}(\mathit{method_{new}}).\overline{\mathtt{method_{new_{end}}}} \\ &\mid \quad \mathtt{method_{new_{end}}}.\overline{\mathtt{change}}[\mathrm{method_1}] \\ &\mid \quad \mathtt{change}(\mathrm{method_1}).\overline{\mathtt{exec}}(\mathrm{method_1}, \mathrm{method_{1_{end}}}).\mathtt{method_{1_{end}}}.\overline{\mathtt{n_1}} \end{aligned}$$

If the process Call($\mathrm{method_{new}}$) *is successfully executed, then*

$$\begin{aligned} & \mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \mathit{Call}(\mathit{method_{new}}).\overline{\mathtt{method}_{\mathtt{new_{end}}}} \\ \mid\ & \mathtt{method}_{\mathtt{new_{end}}}.\overline{\mathtt{change}}[\mathrm{method_1}] \\ \mid\ & \mathtt{change}(\mathrm{method_1}).\overline{\mathtt{exec}}(\mathrm{method_1}, \mathrm{method_{1_{end}}}).\mathtt{method}_{\mathtt{1_{end}}}.\overline{\mathtt{n_1}} \\ \rightarrow\ & \mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \mathit{Call}(\mathit{method_1}).\overline{\mathtt{method}_{\mathtt{1_{end}}}} \mid \mathtt{method}_{\mathtt{1_{end}}}.\overline{\mathtt{n_1}} \end{aligned}$$

If Call($\mathrm{method_1}$) *is successfully executed, then*

$$\begin{aligned} & \mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \mathit{Call}(\mathit{method_1}).\overline{\mathtt{method}_{\mathtt{1_{end}}}} \mid \mathtt{method}_{\mathtt{1_{end}}}.\overline{\mathtt{n_1}} \\ \rightarrow\ & \mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \overline{\mathtt{n_1}} \end{aligned}$$

Theorem 3.3.4 Deletion of method

If **Plan($\mathbf{method_1}, \ldots, \mathbf{method_n}$)** *exists and it is performed in the parallel composition with the process named "***Delete***" then: no method is called and the plan is changed to* **Plan($\mathbf{method_2}, \ldots, \mathbf{method_n}$)**.

Deleting a method from the plan in Definition 3.2.3 is as follows:

$$\begin{aligned} & \mathrm{Plan}(\mathrm{method_1}, \ldots, \mathrm{method_n}) \mid \mathrm{Delete} \mid \overline{\mathtt{start}} \\ \rightarrow\ & \mathrm{Plan}(\mathrm{method_2}, \ldots, \mathrm{method_n}) \mid \overline{\mathtt{n_1}} \end{aligned}$$

We show the examples of Theorem 3.3.1, 3.3.2,3.3.3 and 3.3.4 in Example 3.3.5, 3.3.6, 3.3.7 and 3.3.8, respectively.

Example 3.3.5 *Let the plan be Plan($method_1, method_2, method_3$), the processes of the plan with the requests named "Execute" are shown in Figure 3.1. The figure illustrates the methods named $method_1$, $method_2$ and $method_3$ are executed.*

Example 3.3.6 *Let the plan be Plan($method_1, method_2, method_3$), the processes of the plan with the requests named "Execute" and "Substitute" are shown in Figure 3.2. The figure illustrates the method $method_4$ is substituted for the method $method_2$.*

Example 3.3.7 *Let the plan be $Plan(method_1, method_2, method_3)$, the processes of the plan with the requests named "Execute" and "Add" are shown in Figure 3.3. The figure illustrates the method $method_4$ is added before the method $method_2$.*

Example 3.3.8 *Let the plan be $Plan(method_1, method_2, method_3)$, the processes of the plan with the requests named "Execute" and "Delete" are shown in fig 3.4. The figure illustrates the method $method_2$ is deleted from the plan $Plan(method_1, method_2, method_3)$.*

3.4 Describing Agents' Methods in π-calculus

In this section we develop a description for agents' methods and parameters, by using polyadic π-calculus. In the description, the method refers the parameters through the channels in π-calculus. Hence, the relation among the method and the parameters can be changed dynamically. Each parameter is described as the process in π-calculus. Hence, it is calculated independently of the method.

First, we define the process to store the parameter values before defining the method.

Definition 3.4.1 Process to Store Parameter Value

$$Param(a_i) \overset{def}{=} (a'_i(v_{a_i}).Param(a_i) \ + \ \overline{a_i}[v_{a_i}].Param(a_i))$$

where a'_i means the name used to change the parameter value, and v_{a_i} means the default value of the parameter.

Secondly, we define the method satisfying the following properties:

- The delay of calculating the parameter value is considered.
- The relation among the method and the parameters can be changed dynamically.

Definition 3.4.2 Description of Method

We define the method named "M" referring $a_1 \cdots a_n$ as the parameters.

$$\begin{aligned} M(a_1, \cdots, a_n) &\overset{def}{=} !\Big(M.a_1(v_{a_1}). \cdots .a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \\ &\quad + M(a_1 \cdots a_n).a_1(v_{a_1}). \cdots .a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]\Big) \mid \\ &\quad Param(a_1) \mid \cdots \mid Param(a_n) \end{aligned}$$

where $\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]$ means the information about the method is sent from the processor for π-calculus to outside unit and the method named M is executed with the parameter values $v_{a_1}, \cdots, v_{a_n}$ in the unit.

Finally, we define the method referring n parameters in Definition 3.4.2 and the process changing the parameter value in Definition 3.4.3.

Definition 3.4.3 Process to Change Parameter Value

We define the process changing the parameter value in a_i.

$$renew(a_i, v) \overset{def}{=} \overline{a'_i}[v]$$

where a'_i means the name used to change the parameter value, and v means the new parameter value as Definition 3.4.1.

3.5 The Properties of Methods

3.5.1 Executing Methods

We show how to execute the method described in Definition 3.4.2.

Theorem 3.5.1 Executing Method

The output process in π-calculus having the same name of the method should be given to the process of a method to execute the method. When calculating new parameter values are not finished ,the method refers the default values of the parameters.

We show the example of Theorem 3.5.1 in Example 3.5.2.

Example 3.5.2 Executing Method

In order to execute $M(a_1, \cdots, a_n)$, $\overline{M}$ should be given. We show calculating the process of the method execution below:

$$\begin{aligned}
&\overline{M} \mid M(a_1, \cdots, a_n) \\
&\qquad\downarrow \\
&\qquad\vdots \\
&\qquad\downarrow \\
&\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \mid !\Big(M.a_1(v_{a_1}). \cdots .a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \\
&\qquad + M(a_1 \cdots a_n).a_1(v_{a_1}). \cdots .a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]\Big) \\
&\mid \ Param(a_1) \mid \cdots \mid Param(a_n)
\end{aligned}$$

The results $\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]$ show the method M is executed with the parameter values $v_{a_1}, \cdots, v_{a_n}$.

3.5.2 Changing Parameter Values and References

We show how to change the parameter values.

Theorem 3.5.3 Changing Parameter Values

The parameter value in a_i is changed to v by executing the parallel composition with the process $renew(a_i, v)$.

We show the example in which the parameter value in a_1 is changed to v.

Example 3.5.4 Changing Parameter Values

The parameter value can be changed by executing the parallel composition with $M(a_1, \cdots, a_n)$ and $renew(a_i, v)$. We show calculating the process changing the parameter value below:

$$\begin{array}{l}
renew(a_i, v) \mid M(a_1, \cdots, a_n) \\
\qquad\qquad \downarrow \\
\qquad\qquad \vdots \\
\qquad\qquad \downarrow \\
!\Big(M.a_1(v_{a_1}).\cdots.a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \\
\qquad\qquad + M(a_1 \cdots a_n).a_1(v_{a_1}).\cdots.a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]\Big) \\
\mid \quad Param(a_1) \mid \cdots \mid Param(a_i)\{v/v_{a_1}\} \mid \cdots \mid Param(a_n)
\end{array}$$

If the method M is executed, then

$$\begin{array}{l}
\overline{M} \mid !\Big(M.a_1(v_{a_1}).\cdots.a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \\
\qquad\qquad + M(a_1 \cdots a_n).a_1(v_{a_1}).\cdots.a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]\Big) \\
\mid \ (a'_1(v_{a_1}).Param(a_i) \ + \ \overline{a_1}[v_{a_1}].Param(a_i)) \mid \cdots \\
\mid (a'_i(v).Param(a_i)\{v/v_{a_1}\} \ + \ \overline{a_i}[v].Param(a_i)\{v/v_{a_1}\}) \mid \cdots \\
\mid (a'_n(v_{a_n}).Param(a_n) \ + \ \overline{a_n}[v_{a_n}]..Param(a_n)) \\
\qquad\qquad \downarrow \\
\qquad\qquad \vdots \\
\qquad\qquad \downarrow \\
\overline{M_{out}}[v_{a_1}, \cdots, v, \cdots, v_{a_n}] \quad \mid \ !(M.a_1(v_{a_1}).\cdots.a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \\
\qquad\qquad + M(a_1 \cdots a_n).a_1(v_{a_1}).\cdots.a_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}]\Big) \\
\mid \quad Param(a_1) \mid \cdots \mid Param(a_i)\{v/v_{a_1}\} \mid \cdots \mid Param(a_n)
\end{array}$$

The results shows that the method M is executed with the parameter values v_{a_1}, $\cdots$, v, $\cdots$ and v_{a_n}. It means that the parameter value in a_i is changed to v.

By using Theorems 3.5.1 and 3.5.3, the method is executed independently of the delay of calculating the parameters.

We show how to change the references to parameters in Theorem 3.5.5.

Theorem 3.5.5 Changing References to Parameters

The references to parameters are changed by executing the parallel composition with the output process having the name of the method and new references.

We show the example in which the method "$M(a_1, \cdots, a_n)$" are changed the parameters to $b_1, \cdots, b_n$.

Example 3.5.6 Changing References to Parameters

The references to parameters can be changed by executing the parallel composition with $M(a_1, \cdots, a_n)$ and $\overline{M}(b_1, \cdots, b_n)$. We show calculating the process changing the references to parameters below:

$$\begin{array}{c} \overline{M}[b_1 \cdots b_n] \mid M(a_1, \cdots, a_n) \\ \downarrow \\ \vdots \\ \downarrow \\ b_1(v_{a_1}). \cdots .b_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \mid M(a_1, \cdots, a_n) \end{array}$$

where in order to use the values from the the variables "$b_1 \cdots b_n$", this processes need to execute the parallel composition with the processes $Param(b_1) \mid \cdots \mid Param(b_n)$, then

$$\begin{array}{l} b_1(v_{a_1}). \cdots .b_n(v_{a_n}).\overline{M_{out}}[v_{a_1}, \cdots, v_{a_n}] \mid M(a_1, \cdots, a_n) \mid (b'_1(v_{b_1}).Param(b_1) \\ + \overline{b_1}[v_{b_1}].Param(b_1)) \mid \cdots \mid (b'_n(v_{b_n}).Param(b_n) + \overline{b_n}[v_{b_n}].Param(b_n)) \\ \qquad\qquad\qquad\qquad \downarrow \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad \downarrow \\ \overline{M_{out}}[v_{b_1}, \cdots, v_{b_n}] \mid M(a_1, \cdots, a_n) \mid (b'_1(v_{b_1}).Param(b_1) + \overline{b_1}[v_{b_1}].Param(b_1)) \mid \cdots \\ \mid (b'_n(v_{b_n}).Param(b_n) + \overline{b_n}[v_{b_n}].Param(b_n)) \end{array}$$

The results shows that the method M is executed with the parameter values $v_{b_1}, \cdots, v_{b_n}$. It means that the references to parameters are changed.

3.6 Agent Model

The definitions in the section 3.2 and 3.4 provide the important properties shown in the section 3.3 and 3.5. The properties show that the plan and the parameters of a method can be dynamically changed. However, it is difficult to use directly these definitions on the computer and to provide the functions needed by agents. Hence, we define an agent model to use the definitions and to provide the functions. We show the agent model in Figure 3.5.

In this model, the agent has many units. First, we explain the functions of the units.

Sensor Unit It receives the information from the environment and selects the information needed by the agent, and then sends the selected information to "Plan Generation Unit". Moreover, it sends the requests to change the plans and the parameters of the methods to "PiEngine".

Execution Unit It receives the information about the methods and executes the method according to the information from "PiEngine", and then sends the results of the execution to the environment. Moreover, it sends the requests to change the plans and the parameters of the methods to "PiEngine".

Plan Generation Unit It receives the information selected by "Sensor Unit" and generates plans.

PiEngine It receives the plans from "Plan Generation Unit" and the requests to change the plans and the parameters of the methods from "Sensor Unit" and "Execution Unit" Moreover, it sends the information about the methods with the parameters to "Execution Unit".

Secondly, we show the step to make a plan.

1. "Sensor Unit" receives the information from the environment.
2. "Sensor Unit" sends "Plan Generation Unit" the information which is useful for the agent.
3. "Plan Generation Unit" makes a plan according to the information from "Sensor Unit".

The agent acts by executing the methods in the plan. Under the condition, the agent can execute the methods which are dynamically changed. Thirdly, we explain how to execute and change the methods in the plans.

1. "Sensor Unit" and "Execution Unit" send the requests , which execute the methods and change the plan and the parameter of the methods, to "PiEngine".
2. "PiEngine" calculates the information about the methods by using the requests form the units and sends the information to"Execution Unit". If necessary, the plans and the methods are changed at the calculation.
3. "Execution Unit" executes the methods according to the information from "PiEngine" and sends the results to the environment.

"Sensor Unit" and "Execution Unit" play a part of the wrapper adjusting the information to the environment like as the environmental agent EAMMO [INH+98].

3.7 The Experiments by Using the Dynamically Changing Plans

We explain the experiments by using the description of the plan defined in Section 3.2 and the model in Figure 3.5. These experiments are similar to the example in Section 3.1.

The purpose of these experiments is to check whether the plans and the model are useful in a dynamically changing environment.

3.7.1 The Environment of the Experiments

We explain the environment of the experiments in Table 3.1.

3.7.2 The Ways of the Experiments

We have three experiments in the environment in Table 3.1.

Optimal plan Using the sensor at every time and forming the plan according to the information. This environment is regarded as a static environment.

Normal plan Forming the plan according to the information from the sensor and a new plan is formed after finishing the old plan. This environment is regarded as a dynamically changing environment.

Reflective plan Forming the plan according to the information from the sensor and it is changed according to the executions of the methods. This environment is regarded as a dynamically changing environment.

3.7.3 The Results of the Experiments

We execute 10,000 experiments for each way of the experiments and calculate the average of each 100 experiment. Hence, we get the 100 average of each way. This is shown in Figure 3.6 and Table 3.2. The results in Figure 3.6 show the average steps of each plan in the world which is 50 $\times$ 50 lattice(N =50), and in Table 3.2 show the average steps in N = 50, N = 75 and N = 100.

From the results of the experiments the reflective plan has the advantage of the step as compared with the normal plan. The results of the experiments show that reflective plan is useful in the dynamically changing environment.

3.8 The Experiments by Using the Dynamically Changing Methods

We explain the experiments to show efficiency of the definitions in Section 3.4. The experiments are that the agent searches the route to escape from the fire-world. In the fire-world the fire randomly moves and spreads and the agent can not predict how the environment changes. Thus, the environment is regarded as dynamically changing world.

3.8.1 The Environment of the Experiments

We explain the environment of the experiments. In the fire-world, there are four basic elements to compose the environment.

Floor It is a basic element in the fire-world. The agent can pass it freely. However if it burns fiercely, the agent can not pass it.

Wall It is the partition of the floors. The agent can not pass it. However if it is burnt out, the agent can pass it.

Door It is the conjunctive element between the rooms. If the agent want to move next room, the agent pass it.

Exit It is the goal for the agents. It is basically the same "Door". The difference between the two is that it is not burnable.

The image of the environment is shown in Figure 3.7

3.8.2 The Change of the Environment

The change of the environment is determined by spreading the fire. Hence, we assume the four properties of the element to determine how to spread the fire.

Power It shows the current heating power of the element. The endurance of the element is reduced according to its value, If the degree of the power of an element is higher, the surrounding elements easily start to burn and the agent can not pass it.

Max Power It shows the max value of "Power".

Flammability It shows the flammability of an element. If the degree of its flammability has the high value, it easily starts to burn.

Endurance It shows the endurance of an element. When the degree of its endurance becomes zero, the condition means it burns out.

The environment is changed according to the assumptions. The next environment is calculated by using the current environment and each element. The environment is changed as soon as the calculation is finished. Hence, the agent cannot forecast when the information about the new environment is given.

3.8.3 The Ways of the Experiments

The agent receives the information around it from the environment. However, this information is not periodically provided. Hence, the agent can not forecast when the information of the environment is given. Under the condition, the agent should concurrently and independently perform the calculation for the parameters and the execution of the method. Moreover, the agent should change the reference to the parameters.

We execute the experiment with two kinds of the agents. One of them has the all function in Figure 3.5, the others don't have "PiEngine".

3.8.4 The Results of the Experiments

We execute 10,000 experiments for each agent. We calculate the average time and the success rate to escape from the fire-world. This calculation is executed by each 100 experiment. Hence, we get 100 average times and 100 success rates. We show the average time in Figure 3.8 and the success rate in Figure 3.9 and the total results in Table 3.3.

These results mean that the agent with "PiEngine" can escape from the fire-world in shorter time and with higher rate than without it. Hence, the descriptions are useful in a real time, dynamically changing environment.

3.9 Conclusions

We have developed the agent model to use descriptions for agents' plans and methods based on π-calculus. The model and the descriptions provide the functions which an agent needs to act in a real time, dynamically changing environment.

We have shown the properties of the descriptions as theorems and shown the examples. These properties have shown that the plan can be dynamically changed while it is being executed and the method can be executed independently of the delay to calculate the parameters and can dynamically change the reference to the parameters.

We have shown two experiments. One of them is a tracing problem and another experiment is a fire-world problem. In the tracing problem, we have executed three types experiments. One of them is in static environment and the others are in a dynamically changing environment. The former is used for reference. The latter is used to compare the reflective plan with the normal plan. The results of the experiments have shown the reflective plan is useful in a dynamically changing environment. In the fire-world problem, we have executed two types experiments. One agent is an agent with the unit called "PiEngine" and another agent is an agent without it. The results of the experiments have shown the developed descriptions of the method are useful in a real time, dynamically changing environment.

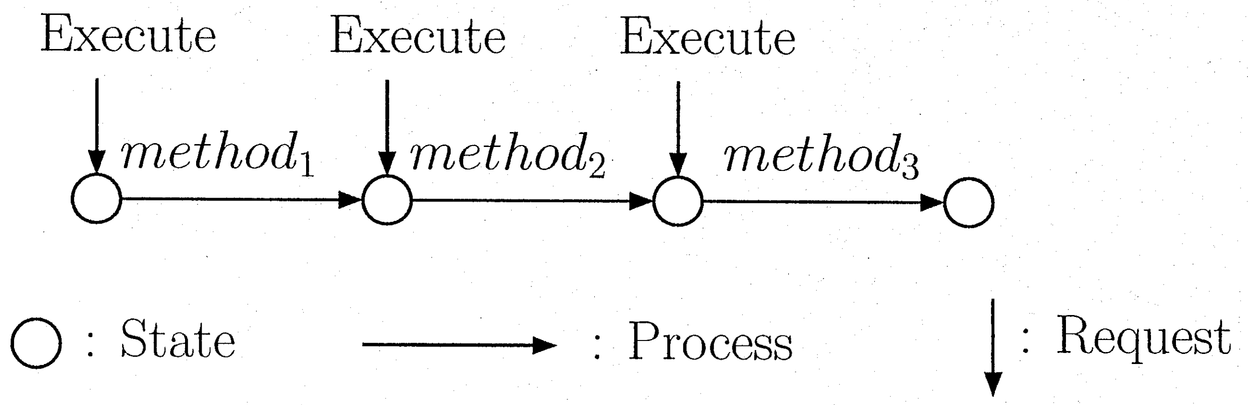


Figure 3.1: Executing $Plan(method_1, method_2, method_3)$

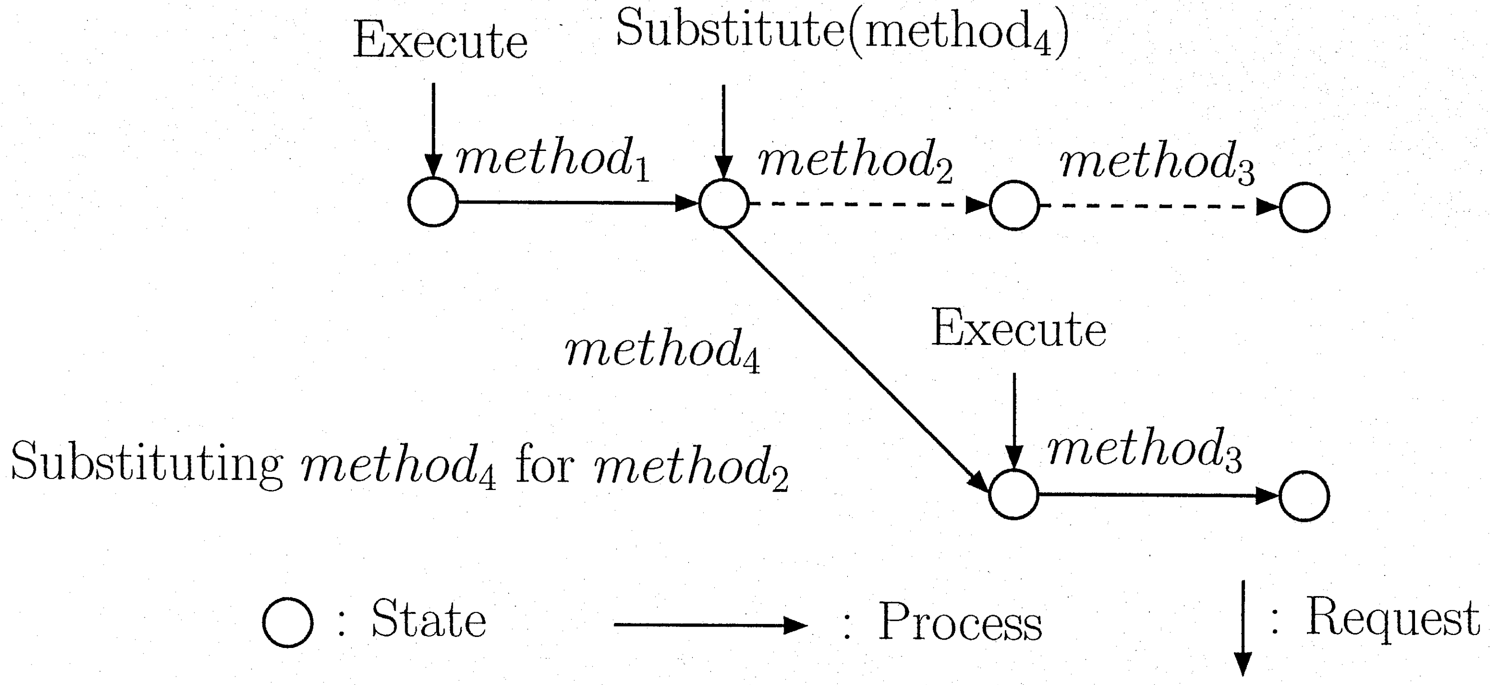


Figure 3.2: Substituting $method_4$ for $method_2$ in $Plan(method_1, method_2, method_3)$

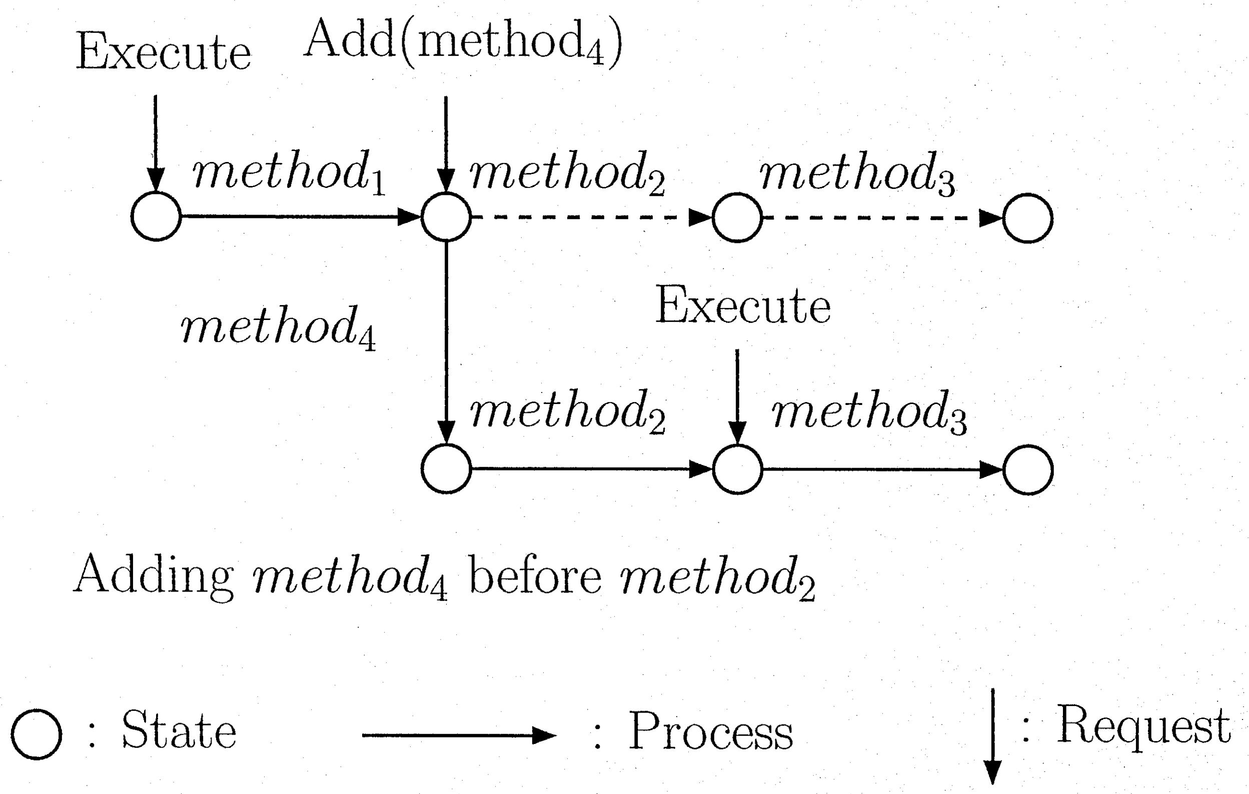


Figure 3.3: Adding $method_4$ before $method_2$

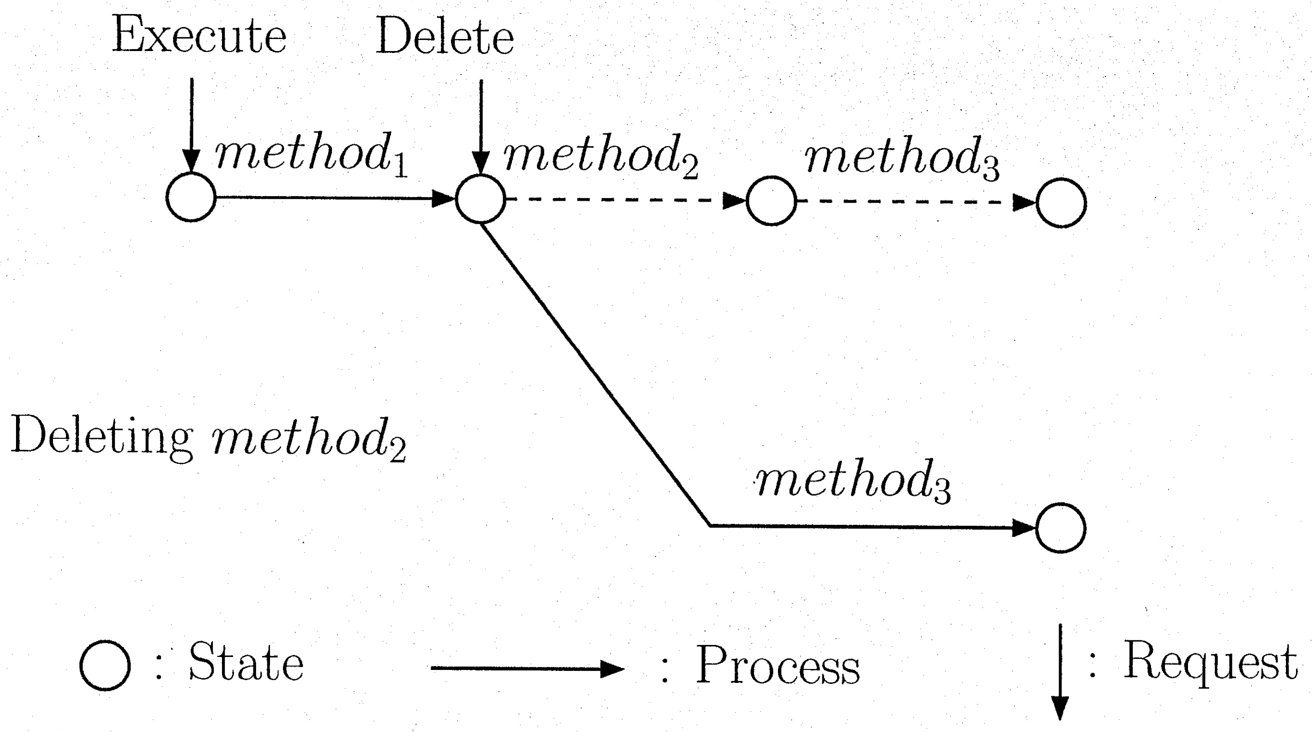


Figure 3.4: Deleting $method_2$ from $Plan(method_1, method_2, method_3)$

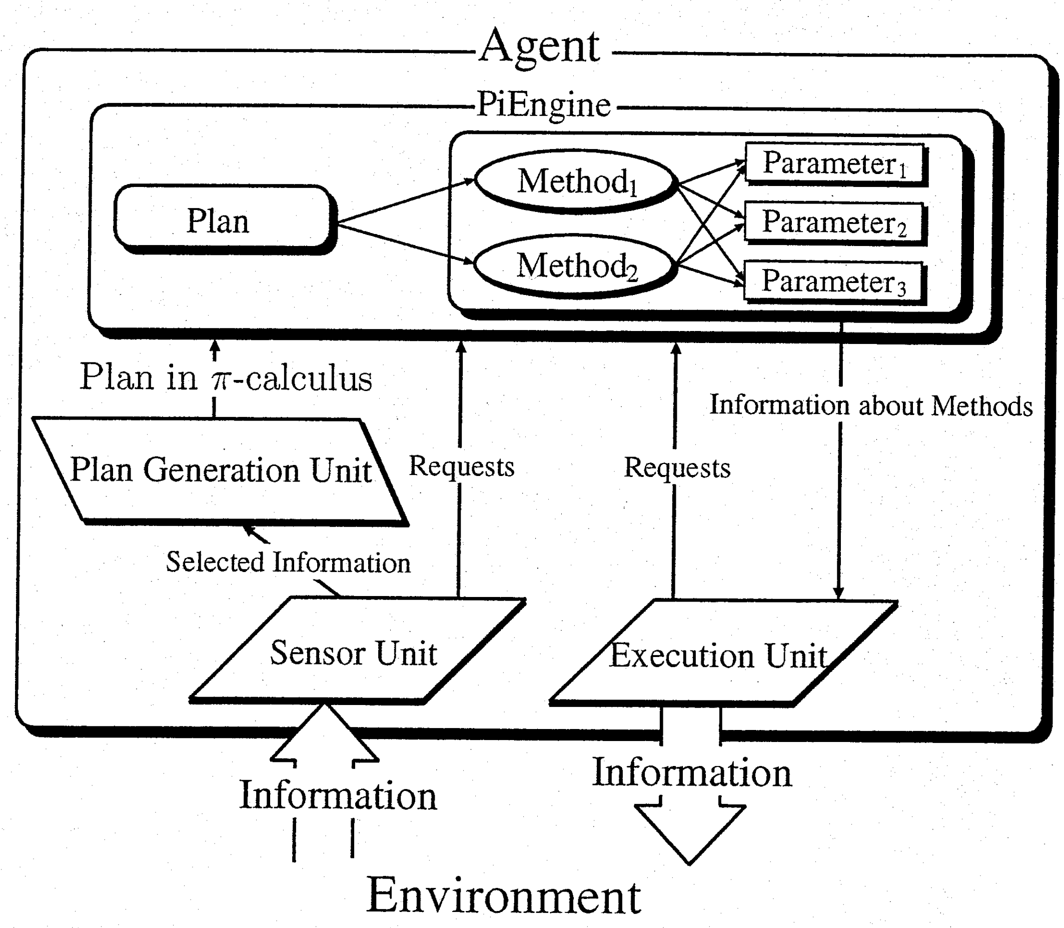
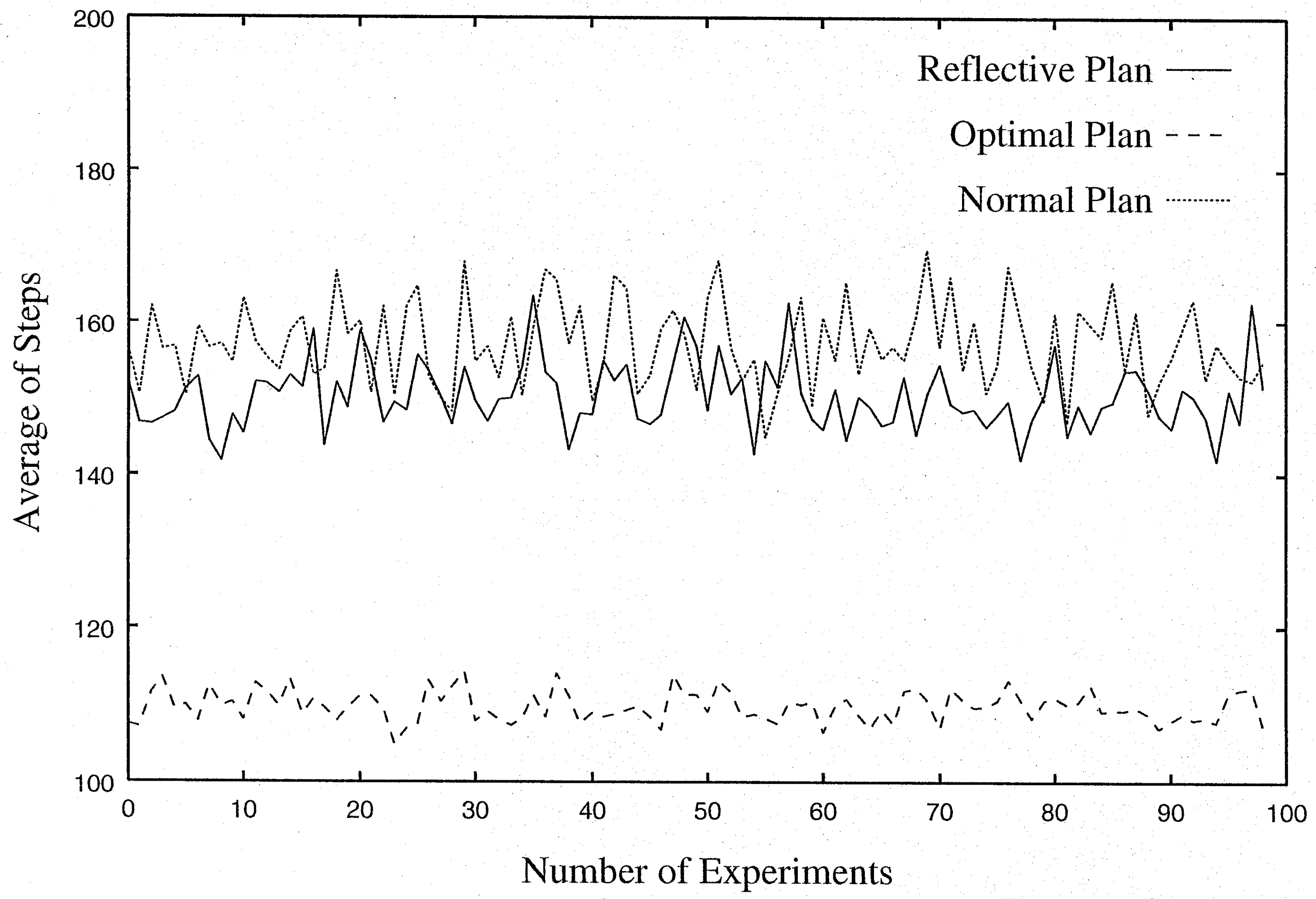


Figure 3.5: Agent Model

Table 3.1: The Environment of the Tracing Problem

Kinds	Values
The Casts of the World	Two agents: agent1 and agent2
The Purpose of the Agent1	To trace the agent2 and catch it
The Area of the World	N×N lattice
The Ability of the Agents	To move the contiguous lattice
The Ability of the Agent1	To receive the position of the agent2 by using the sensor
The Time of the Experiments	To count the number of the movements of the agent1



Each graph shows the number of the average steps till the agent1 catches the agent2.

Figure 3.6: The Results of the Experiments

Table 3.2: The Results of the Tracing Problem
the results show the average step of each type of agents.

Kind	Average		
	N = 50	N = 75	N = 100
Optimal Plan	109.784	160.6075	211.6293
Normal Plan	157.1423	216.8295	275.0708
Reflective Plan	150.2967	209.4437	265.9149

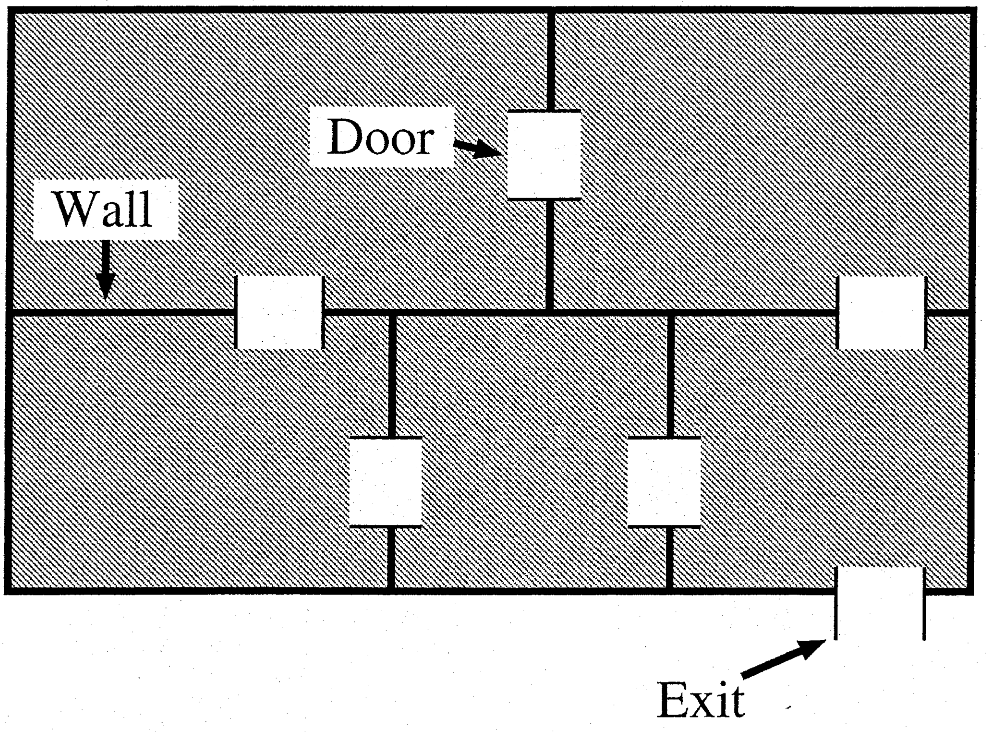


Figure 3.7: The Environment of the Fire-World

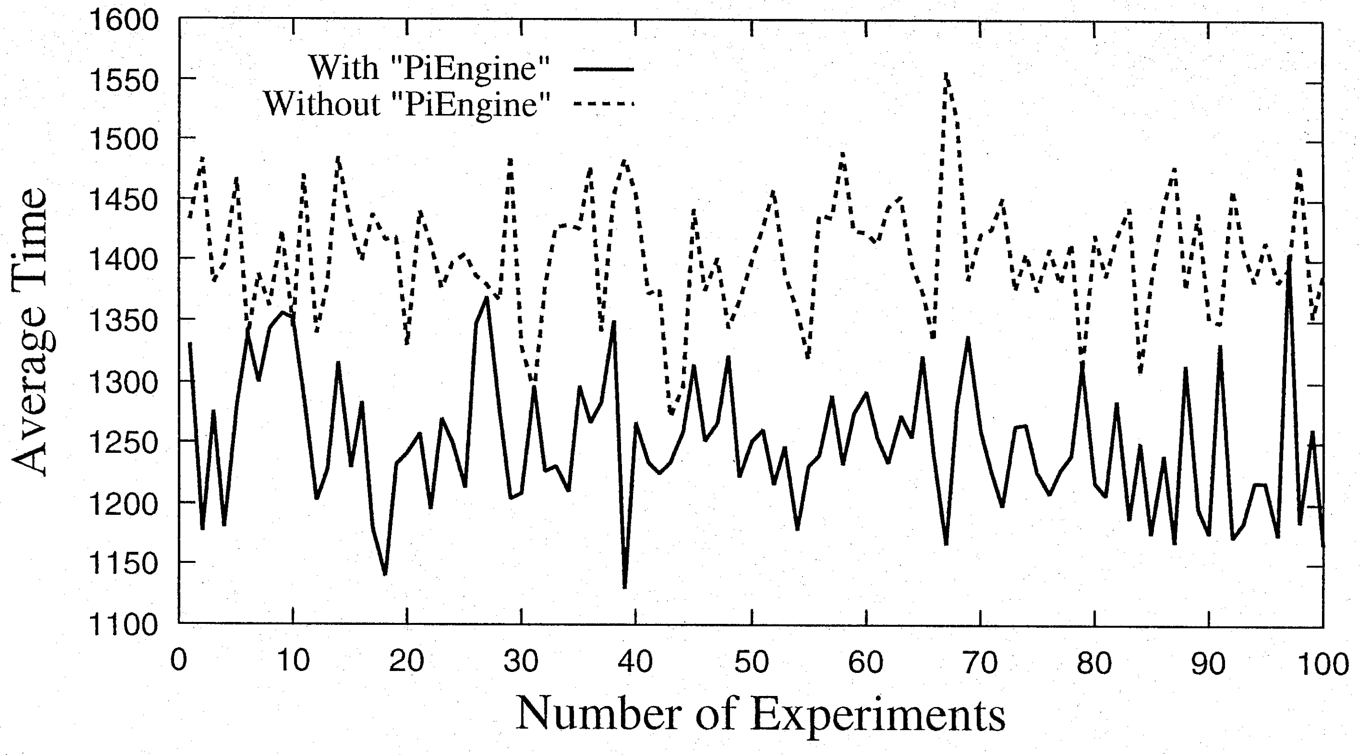


Figure 3.8: The Average Time to Escape from the Fire-World

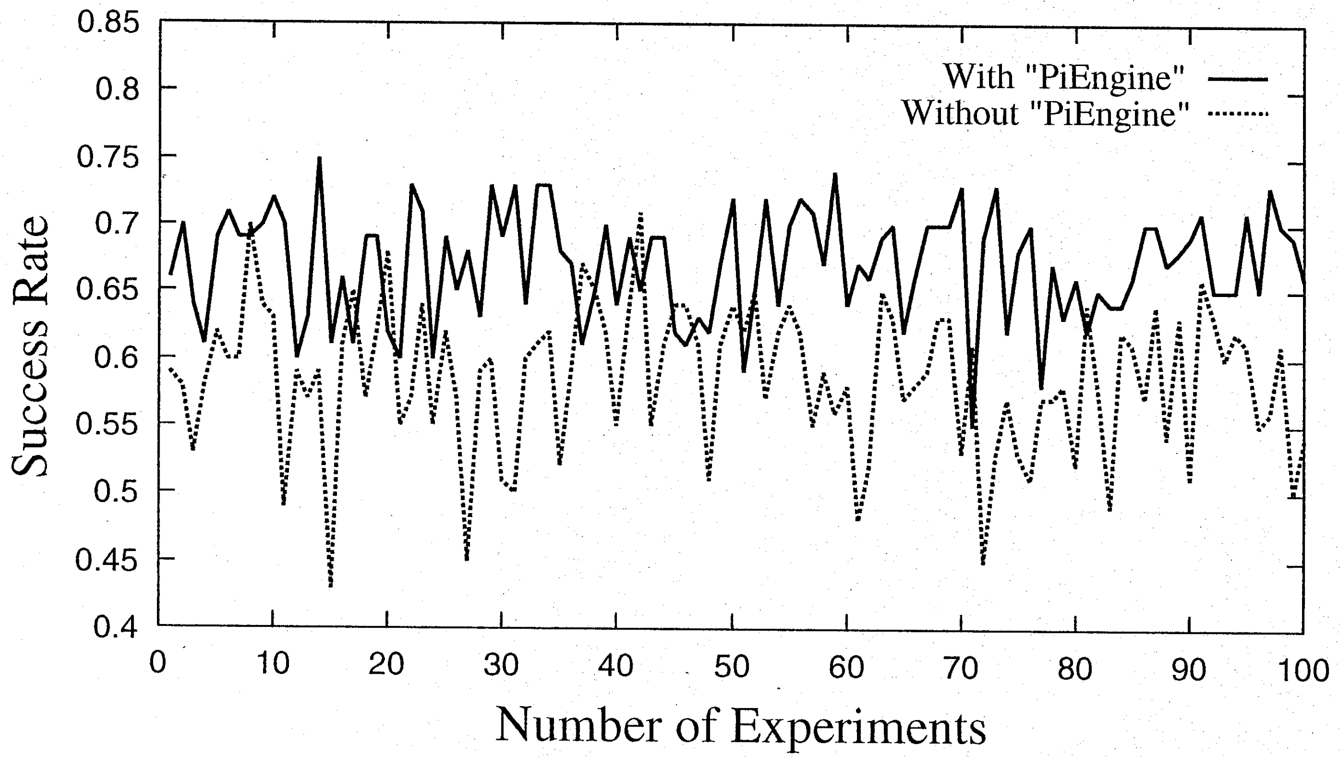


Figure 3.9: The Rate of Succeeding in Escaping from the Fire-World

Table 3.3: The Total Results of Escaping from the Fire-World

Kind of Agent	Average Time	Success Rate
With PiEngine	1249.88	67.02
Without PiEigine	1403.94	58.57

Chapter 4

Descriptions for the Collaboration of Agents by Sharing Actions

We develop a protocol for agents to collaborate with each other by sharing their actions, and define the similar intention focused on these actions. In the protocol, agents build a group with other agents having the same or similar intentions by sharing their actions. We consider that an intention derives actions and the similar intentions derive the common sharable actions. However, it is hard to find the common actions among actions from many intentions, since there is much combination in these actions. Thus, we develop the description to easily find such sharable actions, in order to solve the problem above. Moreover, we develop the description of agents' actions including the communication function. The descriptions are based on π-calculus[Mil91] and extended π-calculus[KM01] which provide a sound foundation for concurrent computation and communication among parallel processes[FG99, Mun98]. Thus, we use processes in π-calculus to define an action, then the action is described as a parallel process which can be transmitted through a network, and it can construct the actions including the communication function.

4.1 Introduction

The performance of a multi-agent system is poor when the agents within they independently perform actions. Although the agents may communicate with each other in order to improve the performance, this may not be the result as the communication can lead to ineffective actions [HCY99] In order to avoid this situation, an agent has to communicate with other agents having similar intentions[CLS97].

In a BDI(belief, desire and intention) model, agents focus on their intentions and collaborate with each other. Although these concepts have been proposed, there are no clear descriptions of how such a model could be used and effectively work [CLS97, Nak99].

First, we define sharable actions and similar intentions focused on agents' actions and develop a new protocol for agents to collaborate with each other. Secondly, we develop the easier way to find common actions by distributed agents and explain how the definitions make the target action sets. Finally, we develop the description of agents' actions including the communication function. The descriptions are based on π-calculus[Mil91] and extended π-calculus[KM01] which provide a sound foundation to concurrent computations and a communication among parallel processes[FG99, Mun98]. Thus, we use processes π-calculus to define an action, then the action is are described as a parallel process which can be transmitted through a network and lead to some action sets, and it can construct the actions including the communication function.

4.2 Sharable Actions

"Sharable Action" means the action which can be performed by every agent in a group. We define the rule to decide "Sharable Action" by denoting unsharable actions which are able to be performed by an only agent. The definition of unsharable actions is as follows:

Definition 4.2.1 Unsharable Actions

- *An action influences an agent, who would perform it, is unsharable, like as a walk and a turn, etc.*
- *An action influences agents in a group is unsharable, like as a communication with neighbor agents, etc.*
- *An action which other agents cannot perform is unsharable, like as taking the neighbor object which adjoins a agent.*

The first rule means that if the action which is significant to an only agent and influences himself is performed by other agent. It causes waste of time. For instance, when an agent would like to move to a position and it performed by other agent, they have to go back to previous position.

The second rule means that if the action which influences agents in the same group is performed by other agents in some other group, the results is undesirable. For instance, when an agent would like to communicate with neighbor agents and it performed by other agent, the agent, who built the action, cannot receive necessary information to fail to communicate.

The third rule means that some agents can perform the action and the others cannot perform it. For instance, when an agent would like to clear rubble around him and it would be performed by other agent, if he does not adjoin the rubble the other agent cannot clear the rubble.

Thus, we call an action sharable, if it does not suit with any rules for the unsharable actions.

4.3 Similar Intentions

We define similar intentions to easily build agents' groups. Agents build groups according to their own intentions. The simplest way to construct a group and to share actions is to build a group containing agents having the same intentions. However, this condition is so difficult that agents cannot build many groups under the condition. Thus, we define similar intentions to simplify the condition.

Before defining similar intentions, we explain what an intention means in this chapter. An intention leads to actions according to an agent's environment and goal. In other words, an agent selects actions according to an intention using the information of his environment and goal.

Definition 4.3.1 An Intention

$Intention : G \times E \rightarrow A$

where "G" denotes the current goal of the agent, "E" denotes the information including the environment around the agent, and "A" denotes the actions having a layered structure.

Similar intentions are defined by a relation among intentions. We define similar intentions as those that load to several common sharable actions as a result of their maps.

Definition 4.3.2 The Similar Intentions

$SIntentions : I^n \times G^n \times E^n \rightarrow B$

where "B" denotes a boolean value(true or false) and "I" denotes an Intention function as in Definition 4.3.1.

If n equals 2, the $SIntention$ function is as follows:

$SIntentions(I_1, G_1, E_1, I_2, G_2, E_2)$

$$\equiv \begin{cases} \text{if} & |Set(I_1(G_1, E_1)) \cup Set(I_2(G_2, E_2))| > \theta \quad \text{then} \quad 1 \\ \text{if} & |Set(I_1(G_1, E_1)) \cup Set(I_2(G_2, E_2))| \leq \theta \quad \text{then} \quad 0 \end{cases}$$

where I_n denotes the $Intention$ function, Set denotes the function from a layered structure to a set as $Set : A \rightarrow \{a | a \in A\}$ and θ is a threshold.

4.4 Agents' Groups

An agent communicates with other agents in order to build a group containing agents having the same or similar intentions. If an agent communicates with all other agents in the system, the amount of communication among agents proliferates. However, an agent only needs to communicate with his surrounding agents, because the probability that the surrounding agents have sharable actions is high. For instance, if an agent would like to move a block and collaborate with other agents being are physically distant, they are unable to find the block or move near it. Thus, any communication among them wastes time.

By considering the previous situation, we can simplify Similar Intentions in Definition 4.3.2, because agents that are physically close to one anther have similar environmental information. The definition simplifying *SimilarIntentions* is as follows:

Definition 4.4.1 Simplified Similar Intentions

$SIntentions' : I^n \times G^n \times E_o \to B$

If n equals 2, the *SIntention'* function is as follows:

$SIntentions(I_1, G_1, E_o, I_2, G_2)$

$$\equiv \begin{cases} \text{if} & \left|Set(I_1(G_1, E_o)) \cup Set(I_2(G_2, E_o))\right| > \theta \text{ then } 1 \\ \text{if} & \left|Set(I_1(G_1, E_o)) \cup Set(I_2(G_2, E_o))\right| \leq \theta \text{ then } 0 \end{cases}$$

where I_n denotes the *Intention* function, *Set* denotes the function from a layered structure to a set as $Set : A \to \{a|a \in A\}$ and θ is threshold. However, it is difficult to find common actions, since there is much combination among sets. Hence, we develop the easier way in Section 4.6.

4.4.1 Building Groups

An agent builds a group with his surrounding agents due to the reasons given previously. When an agent does not belong to a group and would like to collaborate with other agents, it builds a group according to the following steps:

1. The agent, named L, asks each surrounding agent whether he belongs to any groups.
2. In answer to this query, each agent sends his group identification to L otherwise he sends his identification.
3. L sends his current goal to the agents who sent their identification. The amount of communications in one communication decreases compared with Definition 4.3.2, because the leader only needs to send his goal without the information of his environment.
4. Each agent checks the similarity between the intentions defined in Definition 4.4.1. Then, agents send the results of the check.
5. L chooses the members of a new group according to the results from the agents. The criterion for choosing a member is the similarity between the intentions.
6. L sends the new group identification to the chosen agents.

4.4.2 The Collaboration in a Group

The agent that sent the group's identifications to the other agents is the leader of the group. In Subsection 4.4.1, agent L is the leader of the group.

The roles of the leader are the management of the members and the allocation of the sharable actions to the members. The details of the collaboration and the leader's roles are as follows:

1. A member of the group sends his identification to the leader if he wants to continue belonging to the group, otherwise he informs the leader that he secedes from the group.

2. The leader orders the members to make the process according to their own actions.
3. The members simplify the processes and send them to the leader (details in Figure 4.1 in Section 4.6)
4. The leader sends the action sets to all members including himself by using the descriptions in Section 4.7.
5. The members independently perform the action set. They then send the results to the leader when performing their actions are completed.
6. The leader sends his next goal to the members. Each member agent checks the similarity between the intentions. Then, they send the results of the check.
7. The leader checks the results and decides the next member of the group.
8. When the number of member is less than some number γ, the leader informs all members that the group is dissolved and the work of the group is finished. Otherwise, the leader asks the other agents to send him their identifications, and we return to step one.

4.5 Layered Structures

In this section, we explain the layered structure of actions and how the actions are shared among agents. The layers we refer to express the the relationships among the actions. The upper layer specifies the preconditions that must be satisfied in order to perform lower layers. Actions within the same layer can be performed independently of one another. Figure 4.2 shows the actions with two layers.

In Figure 4.2, "a" and "a_i $\{1 \leq i \leq n\}$" are in the upper layer and "b" and "b_i $\{1 \leq i \leq m\}$" are in the lower layer. Moreover, actions "$a_1 \ldots a_n$" (or "$b_1 \ldots b_m$") can be performed in any order, hence actions in the same layer can be performed in parallel. The leader of a group can distribute actions in the same layer to the members of the group.

The four examples in Figure 4.2 are divided into two cases according to the number of actions in the upper layer. The first case is when the upper layer consists of one action, which is shown as the left two diagrams in Figure 4.2. The other case is shown in the right two diagrams where the upper layer consists many actions. In the former case, actions b and b_i $\{1 \leq i \leq m\}$ are performed after action a is performed. Therefore, the agent performing the action a should just tell other agents to perform the actions b and b_i $\{1 \leq i \leq m\}$. In the latter case, when all actions named a_i $\{1 \leq i \leq n\}$ are finished, the action b and b_i $\{1 \leq i \leq m\}$ can be performed. In this case, it is necessary to confirm that all actions in a precondition are finished before any actions in the corresponding lower layer are performed. The leader of a group analyzes the actions it receives from the members and connects them by sharing the same actions. The general style for connecting actions is shown in Figure 4.3.

We abbreviate "the action set A" to "A", and use the same abbreviation for the other sets. The notation "A-X" indicates the difference set between the sets "A" and "X".

The intersections among action sets A, B, C and D which are not described in Figure 4.3 do not contain any elements. For instance, the case $A \cap C = X \neq \emptyset$ implies $A \cap C = X \neq \emptyset$, $A \cap D = \emptyset$, $B \cap C = \emptyset$, $A \cap B = \emptyset$, and $C \cap D = \emptyset$.

1. For the case $A \cap C = X \neq \emptyset$, preconditions have common actions described by X, and hence X must contain the preconditions for B and D. If A equals C, then A contains all the preconditions for B and D.

2. For the case $A \cap D = Y \neq \emptyset$, Y contains the preconditions for B, but C must be finished before perform Y can be performed. If A equals D, then A is performed after finishing C. In this situation, the actions are said to be grafted. Moreover, there is the symmetric case $B \cap C = Y' \neq \emptyset$.

3. For the case $B \cap D = Z \neq \emptyset$, Z requires A and C as preconditions. If B equals D, then B is performed after finishing A and C.

4. The case $A \cap C = X \neq \emptyset$, $A \cap D = Y \neq \emptyset$ and $X \cap Y = \emptyset$ is composed of cases one and two.

5. The case $B \cap C = W \neq \emptyset$, $B \cap D = Z \neq \emptyset$ and $W \cap Z = \emptyset$ is composed of case three and the inverse of case two.

However, it is hard to find common actions as X, Y and so on, since there is much combination among sets. Hence, we develop the way to find common actions by distributed agents.

4.6 Dividing Action Sets

In this section, we define the way to divide action sets into the common elements and the others. For instance, if one agent has action set $A = \{a_1, \ldots a_n\}$, another agent has action set $B = \{b_1, \ldots b_m\}$ and the common action set between A and B is C, then the target sets are $A - C$, $B - C$ and $C = A \cap B$. Then, if $C_1 \cup C_2 = C$ and $C_1 \cap C_2 = \emptyset$, one agent will perform $A' = A - C$ and C_1 and another agent will perform $B' = B - C$ and C_2. This situation means the useless actions which would be performed two times are reduced to be performed by either of agents. Figure 4.4 shows the example.

4.6.1 The Processes Related to Actions

We assign each action to a process written in π-calculus. The definition of the process related to actions is as follows:

Definition 4.6.1 The Process Related to One Action

Let a_i be one element of A, the process related to a_i is as follows:

$Initial(A, a_i) \overset{def}{=} P(A, a_i) + \overline{a_i}[A] + \overline{set}[A, a_i]$

$P(A, a_i) \equiv (\nu\ X)a_i(X).(P(A.X, a_i) + \overline{set}[A.X, a_i])$

where A.X means the concatenation of names defined in Definition 4.6.2 below.

Definition 4.6.1 means that one process, which relates to a_i, divides the action set A into the common action set $A.X$ or $A \leftarrow A - X$. The notation $A.X$ means that X is connected with A, when X differs from A. The definition of it is as follows:

Definition 4.6.2 Concatenating the Names of Action Sets

Let A_1, A_2, $\cdots$, A_{n-1} and A_n mean the different names of the action sets:

$$A_1.A_2.\cdots.A_{n-1}.A_n.X \overset{def}{=} \begin{cases} A_1.A_2.\cdots.A_{n-1}.A_n & if\ A_i = X \\ A_1.A_2.\cdots.A_{n-1}.A_n.X & otherwise \end{cases}$$

where no special order is observed.

For instance, $A.B.C.A$, $A.B.C.B$ and $A.B.C.C$ are translated into $A.B.C$.

The process dividing an action set into the common action sets consists of the parallel composition among the processes in Definition 4.6.1. For instance, when an agent has an action set $A = \{a_i, \ldots a_n\}$, he makes the processes :

$$Initial(A, a_1) \mid Initial(A, a_2) \mid \cdots \mid Initial(A, a_{n-1}) \mid Initial(A, a_n)$$

He sends the processes to other agents or receives processes from other agents, then these processes are calculated and resulted into the simplified processes which bring new action sets.

Before showing the transition of the process in Definition 4.6.1, we define an additional operation in terms of the process *add_element*: $\overline{add\_element}[set\_name, element]$ which means the element named *element* is added into the set named *set_name*. The definition of the process is as follows:

Definition 4.6.3 Add Element to Set

$\overline{add\_element}[set\_name, element] \overset{def}{=} set\_name \leftarrow set\_name \cup \{element\}$

The processes in Definition 4.6.1 can reduce common processes to one process, then the processes in Definition 4.6.3 can make new action sets according to the reduced processes. The procedure of reducing process equals to find common elements among action sets. The algorithm of these processes is shown in Figure 4.1.

1. Each agent makes the parallel composed processes.
   For instance, an agent who has an action set named $A = \{a_1, \ldots, a_n\}$ makes the processes $Initial(A, a_1) \mid \cdots \mid Initial(A, a_n)$
2. Some agents are selected to calculate such processes by their leader. Then, they receive the processes from the other agents and calculate them through the parallel composition. The example of the calculation is as follows:
   $Initial(A, a_1) \mid \cdots \mid Initial(A, a_n) \mid Initial(B, b_1) \mid \cdots \mid Initial(B, b_m) \mid \cdots$
3. If they have enough time at finishing the calculation, they return to the previous step. Otherwise they send the results of the calculation to their leader.
4. The leader makes new action sets by calculating the duplication of the process in Definition 4.6.3 with the results of previous calculations. The example of the calculation is as follows:
   $!(set(set\_name, element)\ .\overline{add\_element}[set\_name, element]) \mid \cdots$ (the results of the previous calculation)

Figure 4.1: Algorithm of Reducing Common Actions

4.6.2 The Examples of the Calculation

In this subsection, we show the example how to find common actions by using the descriptions.

Let one agent have the action set named $A = \{a_1, a_2\}$, one more agent have the action set named $B = \{b_1, a_2\}$ and other agents have the action sets $C = \{c_1, a_2\}$ and $D = \{b_1\}$.

At the first step in the algorithm, the agents having $A = \{a_1, a_2\}$ and $B = \{b_1, a_2\}$ make the following processes:

One agent has the process	$Initial(A, a_1) \mid Initial(A, a_2)$
The another agent has the process	$Initial(B, b_1) \mid Initial(B, a_2)$

At the second step in the algorithm, one of the agents calculates the processes:

$$Initial(A, a_1) \mid Initial(A, a_2) \mid Initial(B, b_1) \mid Initial(B, a_2)$$

The processes are reduced to(the details are in Appendix 1)

$$P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B, c_1) + \overline{b_1}[B] + \overline{set}[B, b_1]$$
$$\mid P(B.A, a_2) + \overline{set}[B.A, a_2]$$

At the third step in the algorithm, if other agents makes the processes:

$$P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1] \mid P(C, a_2) + \overline{a_2}[C] + \overline{set}[C, a_2]$$
$$\mid P(D, c_1) + \overline{b_1}[D] + \overline{set}[D, b_1]$$

These processes are made from the agents having $C = \{c_1, a_2\}$ and $D = \{b_1\}$. They make the following processes:

$$Initial(C, c_1) \mid Initial(C, a_2) \mid Initial(D, b_1)$$

Then, the agents return to the second step, one of the agents calculates the processes:

$$P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1]$$
$$\mid P(B.A, a_2) + \overline{set}[B.A, a_2] \mid P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1]$$
$$\mid P(C, a_2) + \overline{a_2}[C] + \overline{set}[C, a_2] \mid P(D, b_1) + \overline{b_1}[D] + \overline{set}[D, b_1]$$

The processes are reduced to(the details are in Appendix 2)

$$P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B.D, b_1) + \overline{set}[B.D, b_1]$$
$$\mid P(B.A.C, a_2) + \overline{set}[B.A.C, a_2] \mid P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1]$$

At the fourth step in the algorithm: The agent calculates the processes:

$$!(set(set\_name, element).\overline{add\_element}[set\_name, element])$$
$$\mid P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B.D, b_1) + \overline{set}[B.D, b_1]$$
$$\mid P(B.A.C, a_2) + \overline{set}[B.A.C, a_2] \mid P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1]$$

The processes are reduced to(the details are in Appendix 3)

$$!(set(set\_name, element).\overline{add\_element}[set\_name, element])$$
$$\mid \overline{add\_element}[A, a_1] \mid \overline{add\_element}[B.D, b_1] \mid \overline{add\_element}[B.A.C, a_2]$$
$$\mid \overline{add\_element}[C, c_1]$$

Then, the process $\overline{add\_element}[A, a_1]$ and the others make new action sets:

$$A = \{a_1\}, B.D = \{b_1\}, B.A.C = \{a_2\} \text{ and } C = \{c_1\}$$

The results mean that the action sets A, B, C and D are divided into the four action sets A, $B.D$, $B.A.C$ and C.

The set $B.D$ and $B.A.C$ mean the intersection of each set.

The number of the actions in the reduced sets is 4, which is less than that of the initial state 7($A = \{a_1, a_2\}$, $B = \{b_1, a_2\}$, $C = \{c_1, a_2\}$ and $D = \{b_1\}$). Then, the agent who have A(B, C and D) in initial state will perform only the action a_1(a_2, c_1 and b_1, respectively). This example is shown in Figure 4.5.

The left figure in Figure 4.5 means that the agent named "Agent1" can perform only the action named a_1 and a_2. The other agents are in the same situations. The action named a_1 and b_1 can be shared by three agents and two agents, respectively. The right side case means that the agent named "Agent2" takes the action named a_2 and the agent named "Agent4" takes the action named b_1.

4.7 Sharing Actions

In this section, we develop the actions according to sharing actions described in Section4.5. We use π-calculus and extended π-calculus [Mil91, KM01] to implement the transparent communication. Moreover, we describe the calculations of the descriptions by using the operational semantics of π-calculus. The calculations of the previous definitions use the basic definitions of π-calculus with an additional operation.

4.7.1 Describing Actions in π-calculus

First, we define the actions of the leader and the member of a group. They are the basic processes for agents to share actions.

Definition 4.7.1 The Leader's Actions

$Leader \overset{def}{=} !\big(Leader(x).(eval\ \ x)\big)$

where Leader is the port name to receive a process and $(eval\ \ x)$ denotes the self-interpreter like 'eval' in Lisp.

Definition 4.7.2 The Member's Actions

$Agent_n \overset{def}{=} !\big(Agent_n(x).(eval\ \ x)\big)$

where $Agent_n$ is the agent identification and $(eval\ \ x)$ denotes the self-interpreter like 'eval' in Lisp.

Definitions 4.7.1 and 4.7.2 mean an agent receives processes through a network and performs the processes by evaluating them.

Definition 4.7.3 The Action Set Sent to the Members

$Actions(t_1, t_2, \ldots t_{n-1}, t_n) \overset{def}{=} Code(P)$

$P \equiv \overline{system}[t_1].\overline{t_1} \mid \cdots \mid \overline{system}[t_n].\overline{t_n} \mid t_1.\ \cdots\ .t_n.\overline{Leader}[Code(Q)]$

$Q \equiv \overline{t_1} \mid \cdots \mid \overline{t_n}$

In Definition 4.7.3, $Code(P)$ denotes a coded version of the process P to transmit it through a network. Thus, two processes P and $(eval\ Code(P))$ are equivalent.

$\overline{system}[t_i]$ denotes accessing the action named t_i in the system. The details of this process are in the next subsection.

Definitions 4.7.4, 4.7.5, 4.7.6, 4.7.7 and 4.7.8 correspond to the connecting actions 1, 2, 3, 4 and 5 in Section 4.5, respectively. These defined processes are used by the leader agent according to the action structures. In these definitions, A_i denotes the action set $\{a_k, a_{k+1}, \cdots, a_{l-1}, a_l\}(k \leq l)$ assigned to $Agent_i$. Similarly, B_i, C_i, D_i, W_i, X_i, Y_i and Z_i are defined in the same way.

For instance, Definition 4.7.4 means that the leader assigns the action sets A_i, C_i and X_i to $Agent_i$ at first, then after action sets X and A are finished, he assigns the action sets B_i to $Agent_i$ or finishing the action sets X and C he assigns the action set D_i to $Agent_i$. We will explain the calculations in detail in the next section.

Definition 4.7.4 Common Preconditions

$$Common \overset{def}{=} x_1.x_2 \cdots x_{k-1}.x_k.\big((a_i. \ \cdots \ .a_j.P) \mid (c_x. \ \cdots \ .c_y.Q)\big) \mid R$$
$$P \equiv \overline{Agent_1}[Actions(B_1)] \mid \cdots \mid \overline{Agent_n}[Actions(B_n)]$$
$$Q \equiv \overline{Agent_1}[Actions(D_1)] \mid \cdots \mid \overline{Agent_n}[Actions(D_n)]$$
$$R \equiv \overline{Agent_1}[Actions(A_1C_1X_1)] \mid \cdots \mid \overline{Agent_n}[Actions(A_nC_nX_n)]$$

where $X = A \cap C$, $A = \bigcup_{i=1}^{n} A_i$ *and* $A_i \cap A_j = \emptyset (i \neq j)$, *and the other sets are the same.*

Definition 4.7.5 Merged Preconditions

$$Merged \overset{def}{=} (a_i. \ \cdots \ .a_j.y_t. \ \cdots \ .y_s.P) \mid (c_x. \ \cdots \ .c_y.Q) \mid R$$

$$P \equiv \overline{Agent_1}[Actions(B_1)] \mid \cdots \mid \overline{Agent_n}[Actions(B_n)]$$
$$Q \equiv \big(\overline{Agent_1}[Actions(D_1)] \mid \cdots \mid \overline{Agent_n}[Actions(D_n)]$$
$$\mid \overline{Agent_1}[Actions(Y_1)] \mid \cdots \mid \overline{Agent_n}[Actions(Y_n)]\big)$$
$$R \equiv \overline{Agent_1}[Actions(A_1C_1)] \mid \cdots \mid \overline{Agent_n}[Actions(A_nC_n)]$$

where $X = A \cap D$. A_i, B_i, C_i, D_i *and* Y_i *are the same as above.*

Definition 4.7.6 Common Posterior Actions

$$CommonPost \overset{def}{=} \big(a_i. \cdots .a_j.(\overline{a'} \mid P)\big) \mid \big(c_x. \cdots c_y.(\overline{c'} \mid Q)\big) \mid \big(a'.c'.R\big) \mid S$$

$$P \equiv \overline{Agent_1}[Actions(B_1)] \mid \cdots \mid \overline{Agent_n}[Actions(B_n)]$$
$$Q \equiv \overline{Agent_1}[Actions(D_1)] \mid \cdots \mid \overline{Agent_n}[Actions(D_n)]$$
$$R \equiv \overline{Agent_1}[Actions(Z_1)] \mid \cdots \mid \overline{Agent_n}[Actions(Z_n)]$$
$$S \equiv \overline{Agent_1}[Actions(A_1C_1)] \mid \cdots \mid \overline{Agent_n}[Actions(A_nC_n)]$$

where $Z = B \cap D$. A_i, B_i, C_i, D_i and Z_i are the same as above.

Definition 4.7.7 Common Preconditions & Merged Preconditions

$$Common\&Merged$$
$$\overset{def}{=} x_1. \cdots .x_k.\Big((a_i. \cdots .a_j.y_t. \cdots .y_s.P) \mid (c_x. \cdots .c_y.Q)\Big) \mid R$$

$$P \equiv \overline{Agent_1}[Actions(B_1)] \mid \cdots \mid \overline{Agent_n}[Actions(B_n)]$$
$$Q \equiv \big(\overline{Agent_1}[Actions(D_1)] \mid \cdots \mid \overline{Agent_n}[Actions(D_n)]$$
$$\mid \overline{Agent_1}[Actions(Y_1)] \mid \cdots \mid \overline{Agent_n}[Actions(Y_n)]\big)$$

$$R \equiv \overline{Agent_1}[Actions(A_1C_1X_1)] \mid \cdots \mid \overline{Agent_n}[Actions(A_nC_nX_n)]$$

where $X = A \cap C$ and $Y = A \cap D$. A_i, B_i, C_i, D_i, X_i and Y_i are the same as above.

Definition 4.7.8 Common Posterior Actions & More Actions

$$CommonPost\&More$$
$$\overset{def}{=} \big(a_i. \cdots .a_j.(\overline{a'} \mid P)\big) \mid \big(c_x. \cdots c_y.(\overline{c'} \mid Q)\big) \mid \big(a'.c'.(R \mid S)\big) \mid T$$

$$P \equiv \overline{Agent_1}[Actions(B_1)] \mid \cdots \mid \overline{Agent_n}[Actions(B_n)]$$
$$Q \equiv \overline{Agent_1}[Actions(D_1)] \mid \cdots \mid \overline{Agent_n}[Actions(D_n)]$$
$$R \equiv \overline{Agent_1}[Actions(W_1)] \mid \cdots \mid \overline{Agent_n}[Actions(W_n)]$$

$S \equiv w_1. \cdots .w_n.(\overline{Agent_1}[Actions(Z_1)] \mid \cdots \mid \overline{Agent_n}[Actions(Z_n)])$
$T \equiv \overline{Agent_1}[Actions(A_1C_1)] \mid \cdots \mid \overline{Agent_n}[Actions(A_nC_n)]$

where $W = B \cap C$ *and* $Z = B \cap D$. A_i, B_i, C_i, D_i, W_i *and* Y_i *are the same as above.*

4.7.2 The Calculations of the Descriptions

The calculations of the descriptions in Subsection 4.7.1 use the basic definitions of π-calculus and extended π-calculus with an additional operation in terms of the output process *system*: $\overline{system}[a]$ which means accessing the action named a in the system. This process can be performed without a symmetric input process denoted such as $system(x)$.

The operation rule of the process extends the reduction rule of π-calculus as follows:

SYSTEM : $\overline{system}[a].P \xrightarrow{a} P$

The reduction mark $\xrightarrow{a}$ means that after observing the system performs the process named "a", the process is then reduced to P. However, this rule has no influence upon calculating processes in π-calculus.

First, we give the calculation of Definition 4.7.4. Suppose the action set A_i equals $\{a_i\}$ to simplify the explanation. We can also assume the other action sets are defined similarly. Thus, the leader has the following processes:

$Leader \equiv\ !(Leader(x).(eval\ x)) \mid x_1. \cdots .x_n((a_1. \cdots .a_n.P) \mid (c_1. \cdots .c_n.Q)) \mid R$
$P \equiv \overline{Agent_1}[Actions(b_1)] \mid \cdots \mid \overline{Agent_n}[Actions(b_n)]$
$Q \equiv \overline{Agent_1}[Actions(d_1)] \mid \cdots \mid \overline{Agent_n}[Actions(d_n)]$
$R \equiv \overline{Agent_1}[Actions(a_1c_1x_1)] \mid \cdots \mid \overline{Agent_n}[Actions(a_nc_nx_n)]$

The member numbered i has the following processes:

$Agent_i \equiv !(Agent_i(x).(eval\ x))$

The names *Leader* and $Agent_i$ denote the communication ports to the leader and agent named i, respectively.

In the first step, the leader performs the processes $\overline{Agent_i}[Actions(a_ic_ix_i)]$ $\{1 \leq i \leq n\}$. Each agent named i receives $Actions(a_ic_ix_i)$ and evaluates it by the process $(eval\ Actions(a_ic_ix_i))$. These operations are as follows:

$$\begin{array}{ll} Leader & !\big(Leader(x).(eval\ x)\big) \mid x_1.\cdots.x_n\big((a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\big) \mid R \\ \quad\longrightarrow & !\big(Leader(x).(eval\ x)\big) \mid x_1.\cdots.x_n\big((a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\big) \end{array}$$

$$\begin{array}{ll} Agent_i & !\big(Agent_i(x).(eval\ x)\big) \\ \quad\longrightarrow & !\big(Agent_i(x).(eval\ x)\big) \mid (eval\ Actions(a_ic_ix_i)) \end{array}$$

The next step is that each agent performs his actions by $(eval\ Actions(a_ic_ix_i))$:

$$\begin{array}{ll} & (eval\ Actions(a_ic_ix_i)) \\ \longrightarrow & \overline{system}[a_i].\overline{a_i} \mid \overline{system}[c_i].\overline{c_i} \mid \overline{system}[x_i].\overline{x_i} \mid a_i.c_i.x_i.\overline{Leader}[Code(S_i)] \\ \xrightarrow{a_i,c_i,x_i} & \overline{a_i} \mid \overline{c_i} \mid \overline{x_i} \mid a_i.c_i.x_i.\overline{Leader}[Code(S_i)] \\ \longrightarrow & \overline{c_i} \mid \overline{x_i} \mid c_i.x_i.\overline{Leader}[Code(S_i)] \\ \longrightarrow & \overline{x_i} \mid x_i.\overline{Leader}[Code(S_i)] \\ \longrightarrow & \overline{Leader}[Code(S_i)] \end{array}$$

The leader receives $Code(S_i) \equiv Code(\overline{a_i} \mid \overline{c_i} \mid \overline{x_i})$ through the port *Leader*. The next operation is as follows:

$$\begin{array}{ll} & !\big(Leader(x).(eval\ x)\big) \mid (eval\ Code(S_i)) \\ & \mid x_1.\cdots.x_n\big((a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\big) \\ \longrightarrow & !\big(Leader(x).(eval\ x)\big) \mid \overline{a_i} \mid \overline{c_i} \mid \overline{x_i} \mid x_1.\cdots.x_n\big((a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\big) \end{array}$$

When $Agent_1$ finishes his actions and the leader receives the process $Code(\overline{a_1} \mid \overline{c_1} \mid \overline{x_1})$ from the agent named $Agent_1$ then

$$\begin{array}{ll} \longrightarrow & !\big(Leader(x).(eval\ x)\big) \\ & \mid \overline{a_1} \mid \overline{c_1} \mid \overline{x_1} \mid \overline{a_i} \mid \overline{c_i} \mid \overline{x_i} \mid x_1.\cdots.x_n\big((a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\big) \\ \longrightarrow & !\big(Leader(x).(eval\ x)\big) \\ & \mid \overline{a_1} \mid \overline{c_1} \mid \overline{a_i} \mid \overline{c_i} \mid \overline{x_i} \mid x_2.\cdots.x_n\big((a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\big) \end{array}$$

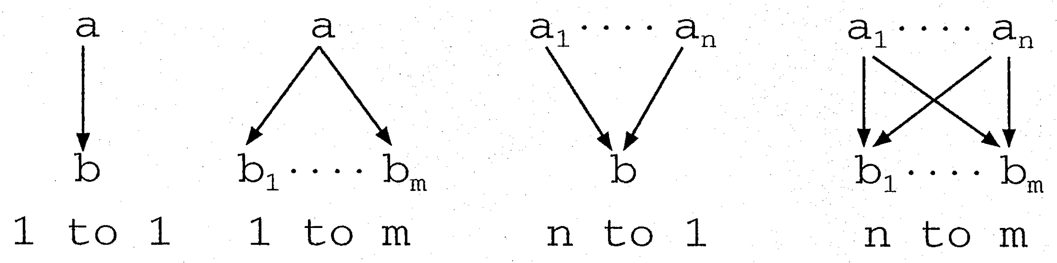
When the other agents, $Agent_j$ $(2 \leq j \leq i-1$ or $i+1 \leq j \leq n)$ finish their actions and the leader receives the processes $Code(\overline{a_j} \mid \overline{c_j} \mid \overline{x_j})$ $(2 \leq j \leq i-1$ or $i+1 \leq j \leq n)$ then

$$\begin{aligned}
&\rightarrow\ !\big(Leader(x).(eval\ x)\big) \mid \overline{a_1} \mid \overline{c_1} \mid \cdots \mid \overline{a_n} \mid \overline{c_n} \mid (a_1.\cdots.a_n.P) \mid (c_1.\cdots.c_n.Q)\\
&\rightarrow\ !\big(Leader(x).(eval\ x)\big) \mid P \mid Q\\
&\rightarrow\ !\big(Leader(x).(eval\ x)\big) \mid \overline{Agent_1}[Actions(b_1)] \mid \cdots \mid \overline{Agent_n}[Actions(b_n)]\\
&\qquad \mid \overline{Agent_1}[Actions(d_1)] \mid \cdots \mid \overline{Agent_n}[Actions(d_n)]
\end{aligned}$$

In the above operations, the agents have performed actions a_i, c_i and x_i. After these operations are finished, the leader sends $Actions(b_i)$ and $Actions(d_i)$ to $Agent_i$. Thus, $Agent_i$ performs the actions b_i and d_i as described above. The results of the operations are as follows: the actions A, C and X are performed first, then B and D are performed. The operations satisfy the conditions for the actions. The other actions can also be performed as described above, but we abbreviate the details here.

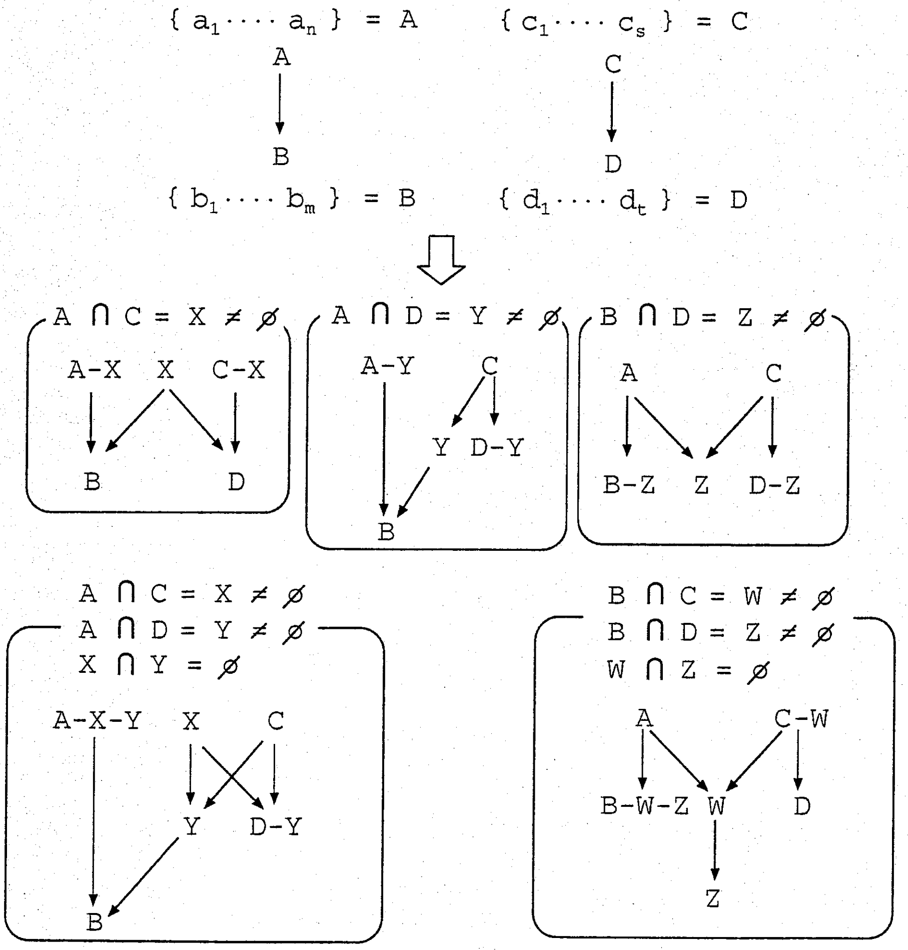
4.8 Conclusions

In this chapter, we have developed a protocol for agents to collaborate with each other by sharing their actions, and defined the similar intention focused on agents' actions. An agent builds a group which consists of other agents having the same or similar intentions by sharing their actions. We have developed the description to easily find such sharable actions, and the description of agents' actions including the communication function. Thus, we have used processes in π-calculus to define an action, then the action is described as a parallel process which can be transmitted through a network, and the processes have been able to construct the actions including the communication function. Moreover, we have explained how the processes make the target sets and have shown the calculations of the descriptions.



The actions a and $a_1, \ldots, a_n$ mean the preconditions of the actions b and $b_1, \ldots, b_m$.

Figure 4.2: Layered Structures of Actions



$\{ a_1 \cdots a_n \} = A$
$\{ c_1 \cdots c_s \} = C$
A
B
C
D
$\{ b_1 \cdots b_m \} = B$
$\{ d_1 \cdots d_t \} = D$
$A \cap C = X \neq \emptyset$
A-X X C-X
B D
$A \cap D = Y \neq \emptyset$
A-Y C
Y D-Y
B
$B \cap D = Z \neq \emptyset$
A C
B-Z Z D-Z
$A \cap C = X \neq \emptyset$
$A \cap D = Y \neq \emptyset$
$X \cap Y = \emptyset$
A-X-Y X C
Y D-Y
B
$B \cap C = W \neq \emptyset$
$B \cap D = Z \neq \emptyset$
$W \cap Z = \emptyset$
A C-W
B-W-Z W D
Z

These show the patterns of connecting actions.

Figure 4.3: Connecting Actions

`Agent1 has action set:` $A = \{a_1 \dots a_n\}$

`Agent2 has action set:` $B = \{b_1 \dots b_m\}$

$$\Downarrow$$

`The results of calculation:`

$$C = A \cap B, A' = A - C, B' = B - C$$

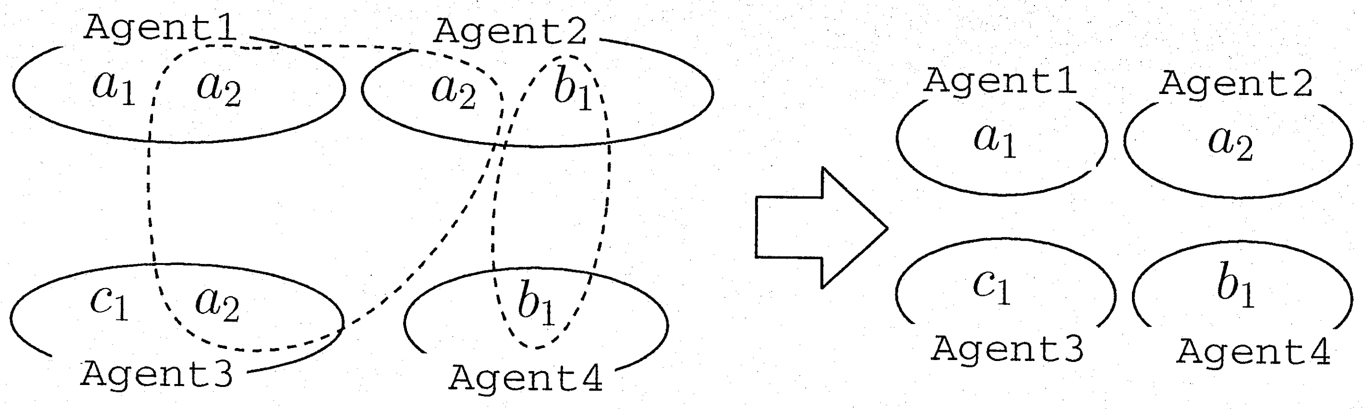
`If` $C_1 \cup C_2 = C, C_1 \cap C_2 = \emptyset$ `then:`

`Agent1 will perform action set:` A' `and` C_1

`Agent2 will perform action set:` B' `and` C_2

This transitions show how to share common actions.

Figure 4.4: Common Actions



Agents share the actions a_2 and b_1.

Figure 4.5: The Example of Sharing Actions

Appendix 1

In this appendix, we show the details of reducing the processes
$Initial(A, a_1) \mid Initial(A, a_2) \mid Initial(B, b_1) \mid Initial(B, a_2)$.

$$\begin{aligned}
& Initial(A, a_1) \mid Initial(A, a_2) \mid Initial(B, b_1) \mid Initial(B, a_2) \\
\rightarrow\ & P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(A, a_2) + \overline{a_2}[A] + \overline{set}[A, a_2] \\
& \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1] \mid P(B, a_2) + \overline{a_2}[B] + \overline{set}[B, a_2] \\
\rightarrow\ & P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \\
& \mid (\nu X)a_2(X).\ (P(A.X, a_2) + \overline{set}[A.X, a_2]) + \overline{a_2}[A] + \overline{set}[A, a_2] \\
& \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1] \\
& \mid (\nu X)a_2(X).\ (P(B.X, a_2) + \overline{set}[B.X, a_2]) + \overline{a_2}[B] + \overline{set}[B, a_2]
\end{aligned}$$

The processes have two cases of the calculation under the condition. We show the detail of only one case, because these cases resemble each other.

The first case is as follows:

If the processes $\overline{a_2}[A]$ and $a_2(X)$ would like to communicate each other:

$$\begin{aligned}
\rightarrow\ & P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid \overline{a_2}[A] \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1] \\
& \mid a_2(X).(P(B.X, a_2) + \overline{set}[B.X, a_2]) \\
\rightarrow\ & P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1] \\
& \mid (P(B.X, a_2) + \overline{set}[B.X, a_2])\{A/X\} \\
\rightarrow\ & P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1] \\
& \mid P(B.X, a_2)\{A/X\} + \overline{set}[B.X, a_2]\{A/X\} \\
\rightarrow\ & P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B, b_1) + \overline{b_1}[B] + \overline{set}[B, b_1] \\
& \mid P(B.A, a_2) + \overline{set}[B.A, a_2]
\end{aligned}$$

The following is the result of the second case:

If the processes $a_2(X)$ and $\overline{a_2}[C]$ would like to communicate each other:

$$\begin{aligned}
\rightarrow\quad & P(A,a_1) + \overline{a_1}[A] + \overline{set}[A,a_1] \ \mid a_2(X).(P(A.X,a_2) + \overline{set}[A.X,a_2]) \\
& \mid P(B,b_1) + \overline{b_1}[B] + \overline{set}[B,b_1] \mid \overline{a_2}[B] \\
\rightarrow & \cdots \rightarrow \\
\rightarrow\quad & P(A,a_1) + \overline{a_1}[A] + \overline{set}[A,a_1] \ \mid P(A.B,a_2) + \overline{set}[A.B,a_2] \\
& \mid P(B,b_1) + \overline{b_1}[B] + \overline{set}[B,b_1]
\end{aligned}$$

The processes $P(B.A,a_2)+\overline{set}[B.A,a_2]$ equal the processes $P(A.B,a_2)+\overline{set}[A.B,a_2]$, because of Definition 4.6.2.

Appendix 2

There are many routes reducing the processes :

$$\begin{aligned}&P(A,a_1)+\overline{a_1}[A]+\overline{set}[A,a_1] \mid P(B,b_1)+\overline{b_1}[B]+\overline{set}[B,b_1]\\ &\mid P(B.A,a_2)+\overline{set}[B.A,a_2] \mid P(C,c_1)+\overline{c_1}[C]+\overline{set}[C,c_1]\\ &\mid P(C,a_2)+\overline{a_2}[C]+\overline{set}[C,a_2] \mid P(D,b_1)+\overline{b_1}[D]+\overline{set}[D,b_1]\end{aligned}$$

However, all routes bring the same results like Appendix 1. Hence, we show the one route of reducing the processes.

$$\begin{aligned}&P(A,a_1)+\overline{a_1}[A]+\overline{set}[A,a_1] \mid P(B,b_1)+\overline{b_1}[B]+\overline{set}[B,b_1]\\ &\mid P(B.A,a_2)+\overline{set}[B.A,a_2] \mid P(C,c_1)+\overline{c_1}[C]+\overline{set}[C,c_1]\\ &\mid P(C,a_2)+\overline{a_2}[C]+\overline{set}[C,a_2] \mid P(D,b_1)+\overline{b_1}[D]+\overline{set}[D,b_1]\\ \rightarrow\ &P(A,a_1)+\overline{a_1}[A]+\overline{set}[A,a_1]\\ &\mid (\nu X)b_1(X).\ (P(B.X,b_1)+\overline{set}[B.X,b_1])+\overline{b_1}[B]+\overline{set}[B,b_1]\\ &\mid (\nu X)a_2(X).\ (P(B.A.X,a_2)+\overline{set}[B.A.X,a_2])+\overline{set}[B.A,a_2]\\ &\mid P(C,c_1)+\overline{c_1}[C]+\overline{set}[C,c_1] \mid P(C,a_2)+\overline{a_2}[C]+\overline{set}[C,a_2]\\ &\mid P(D,b_1)+\overline{b_1}[D]+\overline{set}[D,b_1]\end{aligned}$$

The process $\overline{a_2}[C]$ is chosen from the processes $P(C,a_2)+\overline{a_2}[C]+\overline{set}[C,a_2]$, and the process $\overline{b_1}[D]$ is chosen from the processes $P(D,b_1)+\overline{b_1}[D]+\overline{set}[D,b_1]$:

$$\begin{aligned}\rightarrow\ &P(A,a_1)+\overline{a_1}[A]+\overline{set}[A,a_1] \mid b_1(X).(P(B.X,b_1)+\overline{set}[B.X,b_1])\\ &\mid a_2(X).\ (P(B.A.X,a_2)+\overline{set}[B.A.X,a_2])\\ &\mid P(C,c_1)+\overline{c_1}[C]+\overline{set}[C,c_1] \mid \overline{a_2}[C] \mid \overline{b_1}[D]\end{aligned}$$

The process $\overline{a_2}[C]$ communicates with the process $a_2(X)$, and the process $\overline{b_1}[D]$ communicates with $b_1(X)$:

$$\begin{aligned}\rightarrow\ &P(A,a_1)+\overline{a_1}[A]+\overline{set}[A,a_1] \mid (P(B.X,b_1)+\overline{set}[B.X,b_1])\{D/X\}\\ &\mid (P(B.A.X,a_2)+\overline{set}[B.A.X,a_2])\{C/X\} \mid P(C,c_1)+\overline{c_1}[C]+\overline{set}[C,c_1]\end{aligned}$$

$$\begin{aligned}
\rightarrow\quad & P(A,a_1) + \overline{a_1}[A] + \overline{set}[A,a_1] \;\mid P(B.X,b_1)\{D/X\} + \overline{set}[B.X,b_1]\{D/X\} \\
& \mid P(B.A.X,a_2)\{C/X\} + \overline{set}[B.A.X,a_2]\{C/X\} \\
& \mid P(C,c_1) + \overline{c_1}[C] + \overline{set}[C,c_1] \\
\rightarrow\quad & P(A,a_1) + \overline{a_1}[A] + \overline{set}[A,a_1] \;\mid P(B.D,b_1) + \overline{set}[B.D,b_1] \\
& \mid P(B.A.C,a_2) + \overline{set}[B.A.C,a_2] \;\mid P(C,c_1) + \overline{c_1}[C] + \overline{set}[C,c_1]
\end{aligned}$$

Appendix 3

There is only one route reducing the processes :

$$\begin{aligned} &!\big(set(set\_name, element).\overline{add\_element}[set\_name, element]\big) \\ &\mid P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \mid P(B.D, b_1) + \overline{set}[B.D, b_1] \\ &\mid P(B.A.C, a_2) + \overline{set}[B.A.C, a_2] \ \mid P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1] \end{aligned}$$

First, the processes $set(set\_name, element).\overline{add\_element}[set\_name, element]$ are replicated by the operator !:

$$\begin{aligned} &!\big(set(set\_name, element).\ \overline{add\_element}[set\_name, element]\big) \\ &\mid P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \ \mid P(B.E, b_1) + \overline{set}[B.E, b_1] \\ &\mid P(B.A.C, a_2) + \overline{set}[B.A.C, a_2] \ \mid P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1] \\ \rightarrow\ &!\big(set(set\_name, element).\ \overline{add\_element}[set\_name, element]\big) \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1] \ \mid P(B.E, b_1) + \overline{set}[B.E, b_1] \\ &\mid P(B.A.C, a_2) + \overline{set}[B.A.C, a_2] \ \mid P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1] \end{aligned}$$

The processes $\overline{set}[A, a_1]$, $\overline{set}[B.E, b_1]$, $\overline{set}[B.A.C, a_2]$ and $\overline{set}[C, c_1]$ are chosen from the processes $P(A, a_1) + \overline{a_1}[A] + \overline{set}[A, a_1]$, $P(B.E, b_1) + \overline{set}[B.E, b_1]$, $P(B.A.C, a_2) + \overline{set}[B.A.C, a_2]$ and $P(C, c_1) + \overline{c_1}[C] + \overline{set}[C, c_1]$, respectively:

$$\begin{aligned} \rightarrow\ &!\big(set(set\_name, element).\ \overline{add\_element}[set\_name, element]\big) \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid set(set\_name, element).\ \overline{add\_element}[set\_name, element] \\ &\mid \overline{set}[A, a_1] \mid \overline{set}[B.E, b_1] \ \mid \overline{set}[B.A.C, a_2] \mid \overline{set}[C, c_1] \end{aligned}$$

The processes $\overline{set}[A, a_1]$, $\overline{set}[B.E, b_1]$, $\overline{set}[B.A.C, a_2]$ and $\overline{set}[C, c_1]$ independently communicate with the four processes $set(set\_name, element)$:

$$\begin{aligned}
\rightarrow\ & !\big(set(set\_name, element).\ \overline{add\_element}[set\_name, element]\big) \\
& \mid \overline{add\_element}[set\_name, element]\ \{A/set\_name, a_1/element\} \\
& \mid \overline{add\_element}[set\_name, element]\ \{B.E/set\_name, b_1/element\} \\
& \mid \overline{add\_element}[set\_name, element]\ \{B.A.C/set\_name, a_2/element\} \\
& \mid \overline{add\_element}[set\_name, element]\ \{C/set\_name, c_1/element\} \\
\rightarrow\ & !\big(set(set\_name, element).\ \overline{add\_element}[set\_name, element]\big) \\
& \mid \overline{add\_element}[A, a_1]\ \mid \overline{add\_element}[B.E, b_1] \\
& \mid \overline{add\_element}[B.A.C, a_2]\ \mid \overline{add\_element}[C, c_1]
\end{aligned}$$

Chapter 5

Conclusions

This thesis has studied formalizing agents' behaviors and communications by use of π-calculus. We have considered the model and the protocol for multi-threaded processes with choice to avoid deadlocks in using π-calculus. Further, we have designed the language to implement π-calculus on a computer more easily than mathematical notations of it, and have implemented the system by using the model and protocol. Next, we have developed the description for agents' plans and methods having dynamically changing structures. Moreover, we have developed the protocol for agents to cooperate each other by sharing their actions. Finally, we have developed the description to easily find the sharable actions, and the description of agents' actions including communications.

In Chapter 2, we have developed the model named π-model and the protocol for multi-threaded processes with choice written in π-calculus. We have assigned each process in π-calculus to a thread in the model, and defined the three elements: Processes, Communication Managers(CMs) and Choice Managers(CHMs). A process is a basic unit to control concurrently executed threads in our concurrent and distributed system. CMs manage communication requests along channels from processes, and CHMs manage choice processes in processes. Moreover, we have shown why the protocol frees the processes from the deadlock. Finally, we have designed a primitive language instead of using the mathematical notations of π-calculus, and implemented the system in JAVA called PiEngine by using the π-model and the protocol.

In Chapter 3, we have developed the agent model to use descriptions for agents' plans and methods based on π-calculus. The model and the descriptions have provided the functions which an agent needs to act in a real time, dynamically changing environment. Moreover, we have shown the properties of the descriptions as theorems and shown the examples. These properties have shown that the plan can be dynamically changed while it is being executed and the method can be executed independently of the delay to calculate the parameters and can dynamically change the reference to the parameters. Further, we have shown the experiments. One of them is a tracing problem and another experiment is a fire-world problem. In the tracing problem, we have executed three types experiments. One of them is in static environment and the others are in a dynamically changing environment. The former is used for reference. The latter is used to compare the reflective plan with the normal plan. The results of the experiments have shown the reflective plan is useful in a dynamically changing environment. In the fire-world problem, we have executed two types experiments. One agent is an agent with the unit called "PiEngine" and another agent is an agent without it. The results of the experiments have shown the developed descriptions of the method are useful in a real time, dynamically changing environment.

In Chapter 4, we have developed the protocol for agents to cooperate each other by sharing their actions, and defined the similar intention focused on agents' actions. In the protocol, agents have built a group with agents having the same intentions or the similar intentions by sharing their actions. We have developed the description to easily find such sharable actions, and the description of agents' actions including communications. We have assigned a process to an action in the description, then the actions have been described as the parallel processes by agents through networks and bring some action sets ,and it can construct the actions including communications. Moreover, we have explained how the processes make the target sets and have shown the calculations of the descriptions by using the operational semantics of π-calculus.

It has been shown in this thesis that it is effective to apply π-calculus to formalizing multi-agent systems. However, if agents in a multi-agent system would increase in number, it would be important to improve the performance of the system for π-calculus. Further, the learning mechanism would be needed in more complex environments, then, the system should take in the mechanism.

Acknowledgments

I would like to appreciate Professor Naohiro Ishii, my supervisor, for his continuous encouragement and support.

I wish to thank the members of this thesis reviewing committee: Professor Hidenori Itoh and Professor Toramatsu Shintani for their careful review of this dissertation.

I am heartily grateful to Professor Xiaoyong Du of the Renmin University of China and Assistant Professor Nobuhiro Ito of Nagoya Institute of Technology for their constant guidance, encouragement and suggestions throughout this work.

I wish to thank the Japan Society for the Promotion of Science for its financial supports for my study.

Furthermore, I would like to thank all members of Ishii Laboratory for their helping, and owe them a great deal for comfortable and pleasant school life at the laboratory.

Finally, I wish to thank my family for their mental and various supports.

Bibliography

[AT97] Antonio Moreno and Ton Sales. Limited logical belief analysis. In Anand Rao Lawrence Cavedon and Wayne Wobcke, editors, *Intelligent Agent Systems*, number 1209 in LNAI, pages 104–118. Springer, 1997.

[BA89] M. Ben-Ari. Principles of Concurrent and Distributed Programing. In *Prentice-Hall International(UK) Limited*, 1989.

[Bag89] R. Bagrodia. Synchronization of asynchronous processes in CSP. In *ACM Transaction on Programming Languages and Systems*, volume 11, No. 4, pages 585–597, 1989.

[BD95] Benjamin C. Pierce and David N. Turner. Concurrnet objects in a process calculus. LNCS 907, pp.187–215, TPPP'95, 1995.

[C.A85] C.A.R. Hoare. Communicating sequential processes. In *Communications of the ACM*, volume 21, No.8, pages 666–677, 1985.

[CLS97] P. Cohen, H. Levesque, and I. Smith. On team formation. 2(3):87–114, 1997.

[DD92] R.Milner D. Berry and D.N. Turner. A semantics for ML concurrency primitive. In *POPL'92*, pages 119–129, 1992.

[EK96a] 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.

[EK96b] 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.

[EK97] 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.

[FG99] Jacques Ferber and Olivier Gutknecht. Operational semantics of multi-agent organizations. In *Agent Theories, Architectures, and Languages*, pages 205–217, 1999.

[GA83] 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.

[HCY99] S. Hayden, C. Carrick, and Q. Yang. Architectural design patterns for multiagent coordination, 1999.

[IIDI99a] Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, and Naohiro Ishii. Applying π-calculus to planning for reflective agents. In *IEEE INTERNATIONAL CONFERENCE ON INFORMATION INTELLIGENTCE AND SYST EMS*, pages 622–629, October 31–November 3, 1999.

[IIDI99b] Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, and Naohiro Ishii. Agent model for dynamically-changing plan in π-calculus. In *IASTED Software Engineering and Applications*, pages 159–163, October 6–8, 1999.

[INH$^+$98] Nobuhiro Ito, Kouichi Nakagawa, Takahiro Hotta, Xiaoyong Du, and Naohiro Ishii. Eammo: An environmental agent model for multiple objects. *Information and Software Technology*, 40(7):397–404, 1998. ELSEVIER.

[JZ97] John Bell and Zhisheng Huang. Dynamic goal hirarchies. In Anand Rao Lawrence Cavedon and Wayne Wobcke, editors, *Intelligent Agent Systems*, number 1209 in LNAI, pages 88–103. Springer, 1997.

[KM01] Yoshinobu Kawabe and Ken Mano. Executing coded π-calculus processes. In *Proceedings of the ACIS 2nd International Conference on Softwre Engineering, Artificial Intelligence, Networking & Parallel/Distributed computing(SNPD'01)*, pages 25–32, August 2001.

[Mil91] Robin Milner. Polyadic π-calculus:a tutorial. LFCS Report Series ECS-LFCS-91-180, Laboratory for Foundation of Computer Science, 1991.

[Mor97] David Morley. Semantics of bdi agents and their environment. In Anand Rao Lawrence Cavedon and Wayne Wobcke, editors, *Intelligent Agent Systems*, number 1209 in LNAI, pages 119–134. Springer, 1997.

[MPW92] Milner, R., Parrow, J.G., and Walker, D.J. A calculus of mobile processes. In *Information and Computation,100(1)*, pages 1–77, 1992.

[Mun98] Munindar P. Singh. Applying the Mu-Calculus in Planning and Reasoning about Action. *Journal of Logic and Computation*, 8:425–445, 1998.

[Nak99] Yasuo Nakayama. Communication and attitude change. In *The Second International Conference on Cognitive Science and The 16th Annual Meeting of the Japanese Cognitive Science Society Joint Conference (ICCS/JCSS99)*, 1999.

[Sho93] Y. Shoham. Agent-oriented programming. In *Artif. Intell.*, volume 60, pages 51–92, 1993.

[SN98] S. Au and N.Parameswaran. Plan execution in a dynamic world. In *International Conference on Computing and Information 98*, pages 331–338, 1998.

[TK95] Honda, K. Takeuchi, K. and Kudo, M. An interaction-based language and its typing system. LNCS 817, pp.398–413, PARALE'95, 1995.

List of Publications

1. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Agent Model for Dynamically-changing Plan in π-calculus"
   Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications – SEA'99, Scottsdale, Arizona, USA, October 6–8, 1999 pp.159–163

2. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Applying π-calculus to Planning for Reflective Agents"
   Proceedings 1999 IEEE International Conference on Information Intelligence and Systems – ICIIS'99 Bethesda, Maryland, USA, October 31–November 3, 1999 pp.622–629

3. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Applying π-calculus to dynamically changing plan for agent model"
   International Journal on Artificial Intelligence Tools vol. 9,No.3(2000) pp.377–395, World Scientific Publishing Co. Pte. Ltd.

4. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Formalization for the agent method by using π-calculus"
   6th Pacific Rim International Conference on Artificial Intelligence – PRICAI2000 Melbourne, Australia, August/September 2000
   Lecture Notes in Artificial Intelligence 1886 pp.819

5. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Agent Model for Dynamically-changing Plan in π-calculus"
   International Journal of Computers and Applications, ACTA Press Vol. 23, No 3, 2001, pp.166–172

6. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Using π-calculus to Formalize Agent Plan"
   JPSJ Journal Vol.42 No.9 Sep 2001 pp.2213–2220(in Japanese)

7. Kazunori Iwata, Shingo Itabashi, Naohiro Ishii
   "A Protocol for Multi-Threaded Processes with Choice"
   International Conference on Computational Science - ICCS 2001 San Francisco, CA, USA, May 28–30, 2001
   Lecture Notes in Computer Science, Springer, No. 2074 pp.138–147

8. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Design of the Agent Model for Multi-Threaded Processes"
   Proceedings of the ACIS 2nd International Conference Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing – SNPD'01 August 2001, pp.287–293,

9. Kazunori Iwata and Naohiro Ishii
   "Implementation of the Protocol to Avoid Deadlocks in π-Calculus"
   Knowledge-Based Intelligent Information Engineering System & Allied Technologies
   Frontiers in Artificial Intelligence and Applications Volume 69, IOS Press September 2001, pp.642–646

10. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
"Design of the Agent Model for Multi-Threaded Processes"
International Journal of Computer and Information Science(IJCIS), ACIS press
Vol. 3, No 1, 2002, pp.21–30

11. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
"Description for Sharable Action Discovery to Coordinate Collaborative Agents"
3rd International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing – SNPD'02 pp.123–130 Madrid, Spain, Jul 26–28, 2002

12. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
"Finding and Sharing Common Actions for Distributed Agents Collaboration"
Proceedings of 2002 IEEE International Conference on Industrial Technology – ICIT'02 pp.1254–1259 Shangri-La Hotel, Bangkok, Thailand, Dec 11–14, 2002

Chapter 2

1. Kazunori Iwata, Shingo Itabashi, Naohiro Ishii
   "A Protocol for Multi-Threaded Processes with Choice"
   International Conference on Computational Science - ICCS 2001 San Francisco, CA, USA, May 28–30, 2001
   Lecture Notes in Computer Science, Springer, No. 2074 pp.138–147

2. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Design of the Agent Model for Multi-Threaded Processes"
   Proceedings of the ACIS 2nd International Conference Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing – SNPD'01 August, 2001, pp.287–293

3. Kazunori Iwata and Naohiro Ishii
   "Implementation of the Protocol to Avoid Deadlocks in π-Calculus"
   Knowledge-Based Intelligent Information Engineering System & Allied Technologies
   Frontiers in Artificial Intelligence and Applications Volume 69, IOS Press September, 2001, pp.642–646

4. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Design of the Agent Model for Multi-Threaded Processes"
   International Journal of Computer and Information Science(IJCIS), ACIS press Vol. 3, No 1, 2002, pp.21–30

Chapter 3

1. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Agent Model for Dynamically-changing Plan in π-calculus"
   Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications – SEA'99, Scottsdale, Arizona, USA, October 6–8, 1999 pp.159–163

2. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Applying π-calculus to Planning for Reflective Agents"
   Proceedings 1999 IEEE International Conference on Information Intelligence and Systems – ICIIS'99 Bethesda, Maryland, USA, October 31–November 3, 1999 pp.622–629

3. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Applying π-calculus to dynamically changing plan for agent model"
   International Journal on Artificial Intelligence Tools vol. 9,No.3(2000) pp.377–395, World Scientific Publishing Co. Pte. Ltd.

4. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Formalization for the agent method by using π-calculus"
   6th Pacific Rim International Conference on Artificial Intelligence – PRICAI2000 Melbourne, Australia, August/September 2000
   Lecture Notes in Artificial Intelligence 1886 pp.819

5. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Agent Model for Dynamically-changing Plan in π-calculus"
   International Journal of Computers and Applications, ACTA Press Vol. 23, No 3, 2001, pp.166–172

6. Kazunori Iwata, Nobuhiro Ito, Xiaoyong Du, Naohiro Ishii
   "Using π-calculus to Formalize Agent Plan"
   JPSJ Journal Vol.42 No.9 Sep 2001 pp.2213–2220(in Japanese)

Chapter 4

1. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Description for Sharable Action Discovery to Coordinate Collaborative Agents"
   3rd International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing – SNPD'02 pp.123–130 Madrid, Spain, Jul 26–28, 2002

2. Kazunori Iwata, Nobuhiro Ito, Naohiro Ishii
   "Finding and Sharing Common Actions for Distributed Agents Collaboration"
   Proceedings of 2002 IEEE International Conference on Industrial Technology – ICIT'02 pp.1254–1259 Shangri-La Hotel, Bangkok, Thailand, Dec 11–14, 2002