Selection of Mobile Agents

Ichiro Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: ichiro@nii.ac.jp

***Abstract*— When a task is assigned to mobile agents, ones suitable to perform the task need to be selected according to not only their application-specific behaviors but also their mobilities. The focus of current research, however, is on the development of execution platforms and applications for mobile agents and not on methodologies for the selection of mobile agents. This paper presents a general approach for selecting mobile agents according to their itineraries among multiple hosts. The approach offers a process algebra-based language for formally specifying the itineraries of mobile agents and an algebraic order relation between two itineraries specified as terms of the language. The relation can strictly decide whether or not the itineraries of mobile agents can satisfy the itinerary required by a given task, in the sense that the agents can migrate to all the hosts required by the task in a permissible order specified by the task. A prototype implementation of this approach was constructed on a Java-based mobile agent system. It enables each mobile agent to specify its itinerary as a term of the language and to migrate over a network according to only the itinerary. Also, when it receives a task request from its external environment, it can select a suitable mobile agent to perform the task by using the order relation. The paper also describes its prototype implementation and a practical application.**

I. Introduction

Mobile agents are software agents that can travel among computers under their own control. Mobile agent technology is being promoted as an emerging technology that makes it much easier to design, implement and maintain distributed systems. It may also be treated as a type of software agent technology, but it is not always required to offer such intelligent capabilities, for example reactive, pro-active, and social behaviors which are features of existing software agent technologies. This is because these capabilities tend to be large in scale and processing, where each mobile agent should not consume many computational resources, such as processors, memory, files, and networks, at its destinations. Also, each mobile agent must be made as small as possible because the size of a moving agent seriously affects the cost of migrating it over a network. Therefore, mobile agent-based distributed applications should offer various small agents specialized for supporting their particular tasks, rather than a few general-purpose agents for supporting various tasks and they should select suitable agents to perform the tasks requested by users.

For the same reason, it is difficult for each mobile agent to dynamically generate an efficient itinerary among multiple hosts, because both the cost of discovering such an itinerary and the size of its program tend to be large. This problem becomes more serious when mobile agents are used for network management, which is one of the most typical applications of mobile agent technology [11]. This is because network management systems must often handle networks that may have various malfunctions and disconnections and whose exact topology may not be known. Consequently, it is almost impossible for each mobile agent to discover its proper destinations on such networks. As a result, some existing mobile agent-based applications assume that their mobile agents are often launched with a set itinerary for greater of agent migration efficiency over the networks. Moreover, the itineraries of mobile agents must often be fixed to limit the ranges of free movement of the agents for the reason of security discussed in [4].

Nevertheless, current work on mobile agents focuses on the creation of an infrastructure, that among other tasks, provides functions and services that can be used by agents and a secure environment for both the mobile agents and their local execution environment. Therefore, the tasks of selecting mobile agents unfortunately have received little attention so far. That is, mobile agent technology lacks general methodologies for selecting mobile agents that can satisfy the itinerary required by given tasks. On the other hand, there have been many approaches for assigning tasks to non-mobile software agents (for example, [28], [6]) and some of the approaches may be available in the selection of mobile agents according to their behaviors. Mobile agents also need to be selected according to their itineraries among hosts in a network in addition to their behaviors.

The goal of this paper is to establish a general approach for selecting mobile agents according to their itineraries instead of their application-specific behaviors. To select suitable agents, we must analyze not only the itineraries that mobile agents can migrate along but also the itineraries that a given request requires candidates to migrate along. The both itineraries are various and complex. As a result, it is difficult and tedious to judge whether or not each of the candidates can satisfy the required itinerary. To solve this problem, the approach proposes a theoretical foundation for specifying and reasoning about the itineraries of mobile agents. It offers a process algebra-based specification language for the itineraries that mobile agents are able and required to migrate along and an algebraic relation for comparing the itineraries of mobile agents. It is not only a theoretical foundation but also an implementable mechanism for controlling and selecting mobile agents to efficiently perform a requested task. The current implementation of the approach is built on our Java-based mobile agent system, called MobileSpaces [20].

This paper is organized as follows: Section 2 presents the basic ideas behind the approach presented in this paper. Section 3 defines the process calculus for specifying mobile agents and algebraic relations over expressions written in the calculus. Section 4 presents the design and implementation of the approach. Section 5 presents some practical applications of the approach and Section 6 surveys related work. Section 7 briefly presents some future issues and Section 8 makes some concluding remarks.

II. Approach

This paper presents a general approach for selecting suitable and efficient mobile agents that can satisfy the requirements of a request from users, other agents, or external systems. Mobile agents should be selected according to two criteria: their application specific behaviors and their itineraries. Existing task assignment mechanisms for non-mobile software agents may be able to deal with the former criterion but cannot support the latter. Hence, the approach presented in this paper focuses on the selection of mobile agents according to their itineraries.

A. Agent Itinerary

The itinerary that a mobile agent is required to migrate along by a given task request is dependent on the request's kinds of applications. One of the most typical applications of mobile agent technology is remote searching and filtering, where mobile agents migrate among remote database servers to retrieve information and carry only relevant information over a network. If a searching agent gathers information from a database server and reflects the information on other database servers, its movement order among servers may affect the contents of the servers. Therefore, such an agent must migrate among the servers according to a specified itinerary. On the other hand, if a searching agent can travel among database servers to aggregate its interesting information from the servers without any writing on any database server, the order of its movement may be independent of its achievement. Moreover, an agent's itinerary is often dependent on the results of an agent's application-specific behavior. For example, such a searching agent can determine its destinations based on information it has acquired from the database servers that it has visited so far. However, there is a trade-off between the security advantages of fixed itineraries and the flexibility of free roaming. The approach presented in this paper allows each mobile agent to autonomously select one route from the candidates that are specified in the agent's itinerary.

B. Itinerary Specification Language

Since mobile agents' programs are written in general-purpose programming languages, such as Java, it is almost impossible to exactly extract only the itineraries of mobile agents from their programs. Therefore, our approach provides a specification language for the itineraries of mobile agents and assumes that each mobile agent explicitly specifies its own itinerary as a term of the language. To strictly select mobile agents according to their itineraries, the language is formulated as an extended process algebra with the expressiveness of agent movement. Since each mobile agent is disallowed to stray from its own itinerary, it can follow the itinerary and reflect the results of its processing on its destinations only when its itinerary permits it to migrate to the destinations. Furthermore, this approach assumes that the itinerary required by a task request from users, other agents, and external systems is written in the language. A given request may permit an agent to migrate along a traversal of all the specified hosts irrespective of arrival order, or a loose route, where a loose route means that some hosts may be omitted or visited any number of times. The language specifies such indefiniteness and agents' discretion by extending itself with non-deterministic operators.

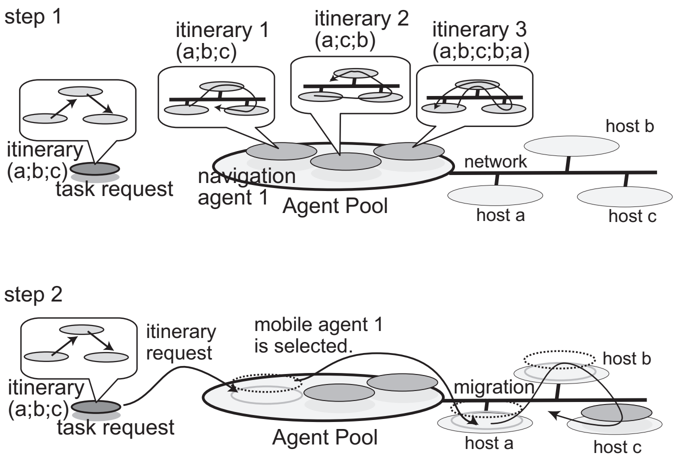

Fig. 1. System structure and its basic mechanism.

C. Algebraic Order Relation for Agent Selection

The selection of mobile agents is formulated based on an order relation over terms of the language. The relation is defined based on the notion of bisimulation and can compare between the possible itinerary of each mobile agent and the itinerary required by a task request. It allows us to strictly judge whether or not the former itinerary can satisfy the latter itinerary. We implemented the relation as a mechanism, called Agent Pool, for storing and selecting mobile agents, as shown in Figure 1. Each agent pool can be viewed as a broker or match-maker by users, other software agents, and external systems. When it receives a task request, it compares the itinerary of each of its stored agents with the itinerary required by the request by using the relation to select one of the most suitable mobile agent to accomplish the request. Note that the order relation selects a better itinerary from a finite set of given itineraries, but does not intend to generate the most efficient itinerary. Thus, the computational complexity of the relation is not large.

D. Remarks

Since the goal of this paper is to present a selection mechanism suitable for mobile agents according to their itineraries, this paper does not limit the types of tasks that mobile agents are requested to perform, except the order of movement to the hosts at which each task must be performed.

Some readers may think that simple executable languages, such as Lisp and Prolog, should be used to specify itineraries, but it is not easy to exactly verify whether or not itineraries written in such languages satisfies the itinerary required by a request. Also, the specification of each mobile agent should be independent of any particular implementations so that we can specify and select mobile agents, which are implemented in different mobile agent platforms, in a unified manner. That is, we need a platform-neutral approach for reasoning about mobile agents.

Although this paper addresses mobile agent technology, our approach itself makes several contributions to active network technologies, in particular active packets (also called a programmable capsule) [31], [9]. This is because the language that specifies itineraries of mobile agents can be directly used as a notation for describing routings of active packets. The language is simple and optimized for describing itineraries among multiple hosts. Therefore, programs written in the language are small enough to be embedded into packets and can be interpreted without consuming the computing power of hosts. Furthermore, existing active network technologies lack mechanisms for selecting active packets as mobile agent technology. Our selection mechanism is available in the selection of suitable active packets.

III. Formalizing Agent Itinerary

This section defines an executable specification language and an order relation as a theoretical basis for agent selection.

A. Agent Itinerary Specification Language

Our specification language is basically inherited from those of existing process algebras, for example CCS [15], π-calculus [16], and ACP [2], because process algebras provide well-studied foundations.

Definition 3.1: The set $\mathcal{E}$ of language expressions, ranged over by $E, E_1, E_2, \ldots$, is defined recursively by the following abstract syntax:

$$\begin{array}{lllllllll} E & ::= & \mathtt{0} & \mid & \ell & \mid & E_1 \,;\, E_2 & \mid & E_1 + E_2 \\ & \mid & E_1 \# E_2 & \mid & E_1 \% E_2 & \mid & E_1 \,\&\, E_2 & \mid & E^\star \end{array}$$

where $\mathcal{L}$ is the set of location names, ranged over by $\ell, \ell_1, \ell_2, \ldots$. We often omit $\mathtt{0}$. We describe a subset language of $\mathcal{E}$ as $\mathcal{S}$, when eliminating $E_1 \# E_2$, $E_1 \% E_2$, $E_1 \,\&\, E_2$, and $E^\star$ from $\mathcal{E}$. Let $S, S_1, S_2, \ldots$ be elements of $\mathcal{S}$. □

We describe itineraries required by tasks as terms of $\mathcal{E}$ and agents' itineraries as terms of S. The intuitive meanings of basic expressions in the language are as follows.

- $\mathtt{0}$ represents a terminated itinerary.
- ℓ represents agent migration to the host whose name or network address is ℓ.
- $E_1 \,;\, E_2$ denotes the sequential composition of two itineraries E_1 and E_2. If the migration of E_1 terminates, then the migration of E_2 follows that of E_1.
- $E_1 + E_2$ represents that an agent moves according to either E_1 or E_2. The selection can be explicitly performed by the presence of E_1 or E_2.
- $E_1 \# E_2$ means that an agent can select either E_1 or E_2 according to its internal computation independently of the presence of E_1 or E_2.
- $E_1 \# E_2$ means that an agent can follow either E_1 before E_2 or E_2 before E_1 as its itinerary.
- $E_1 \,\&\, E_2$ means that two itineraries E_1 and E_2 can be performed asynchronously.[1]
- $E^\star$ is a transitive closure of E and means that an agent can move along E an arbitrary number of times.

We believe that readers can mostly understand the expressiveness and usage of the language from the above intuitive meanings without reading mathematical definitions presented in this section. On the other hand, our approach aims at providing a theoretical and practical foundation for reasoning about agent mobility so that the remainder of this section defines the language in a formal manner.

The operational semantics of the language is based on the concept of interleaving semantics and defined as two layers of labeled transition rules: *migrant transition*, written as $\xrightarrow{\ell}$ ($\longrightarrow \subseteq \mathcal{E} \times \mathcal{L} \times \mathcal{E}$), and *non-deterministic transition*, written as $\xrightarrow{\tau}$ ($\longrightarrow \subseteq \mathcal{E} \times \{\tau\} \times \mathcal{E}$).

Definition 3.2: The language is a labeled transition system $\langle\, \mathcal{E}, \mathcal{L} \cup \{\tau\} \{ \xrightarrow{\alpha} \subseteq \mathcal{E} \times \mathcal{E} \mid \alpha \in \mathcal{E} \cup \{\tau\} \} \,\rangle$. The transition relation $\longrightarrow$ is defined by two kinds of axioms or induction rules as given below:

$$\frac{-}{\ell \xrightarrow{\ell} \mathtt{0}} \qquad \frac{E_1 \xrightarrow{\ell} E_1'}{E_1 \,;\, E_2 \xrightarrow{\ell} E_1' \,;\, E_2}$$

$$\frac{E_1 \xrightarrow{\ell} E_1'}{E_1 + E_2 \xrightarrow{\ell} E_1'} \qquad \frac{E_2 \xrightarrow{\ell} E_2'}{E_1 + E_2 \xrightarrow{\ell} E_2'}$$

$$\frac{E_1 \xrightarrow{\ell} E_1'}{E_1 \,\&\, E_2 \xrightarrow{\ell} E_1' \,\&\, E_2} \qquad \frac{E_2 \xrightarrow{\ell} E_2'}{E_1 \,\&\, E_2 \xrightarrow{\ell} E_1 \,\&\, E_2'}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 \,;\, E_2 \xrightarrow{\tau} E_1' \,;\, E_2} \qquad \frac{-}{E_1 \# E_2 \xrightarrow{\tau} E_1} \qquad \frac{-}{E_1 \# E_2 \xrightarrow{\tau} E_2}$$

$$\frac{-}{E_1 \% E_2 \xrightarrow{\tau} E_1 \,;\, E_2} \qquad \frac{-}{E_1 \% E_2 \xrightarrow{\tau} E_2 \,;\, E_1}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 + E_2 \xrightarrow{\tau} E_1'} \qquad \frac{E_2 \xrightarrow{\tau} E_2'}{E_1 + E_2 \xrightarrow{\tau} E_2'}$$

$$\frac{E_1 \xrightarrow{\tau} E_1'}{E_1 \,\&\, E_2 \xrightarrow{\tau} E_1' \,\&\, E_2} \qquad \frac{E_2 \xrightarrow{\tau} E_2'}{E_1 \,\&\, E_2 \xrightarrow{\tau} E_1 \,\&\, E_2'}$$

where $\mathtt{0} \,;\, E$ is treated to be syntactically equal to E and $E^\star$ is recursively defined as $\mathtt{0} \# (E \,;\, E^\star)$. We often abbreviate $E_0 \xrightarrow{\tau} E_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} E_{n-1} \xrightarrow{\tau} E_n$ to $E_0 (\xrightarrow{\tau})^n E_n$. □

In Definition 3.2, the ℓ-transition defines the semantics of an agent's mobility. For example $E \xrightarrow{\ell} E'$ means that the agent moves to a host named ℓ and then behaves as E'. Also, if there are two possible transitions $E \xrightarrow{\ell_1} E_1$

[1]In process algebras, & is an operator for specifying parallel executions. The operational semantics of the language is an interleaving model in the literature of process algebras and each agent migration is an atomic action.

and $E \xrightarrow{\ell_2} E_2$ in an agent, the processing of the agent chooses one of the destinations ℓ_1 or ℓ_2. On the other hand, the τ-transition corresponds to a non-deterministic choice in an agent's itinerary. For example, if an agent has two transitions $E \xrightarrow{\tau} E_1$ and $E \xrightarrow{\tau} E_2$, then one implementation of it can follow E_1 and another can follow E_2.

To demonstrate the expressiveness of our language, we describe three agent migration patterns studied in [1]. To simplify our discussion hereafter, we introduce three macros, corresponding to the patterns, for example, *Travel*, *Star*, and *Turn*. These macros do not extend the language because they are mapped into $\mathcal{E}$. We describe a list of host names as $[\ell_1, \ell_2, \ldots, \ell_n]$, where $\ell_1, \ldots, \ell_n \in \mathcal{L}$. Let [] be an empty list, $car(X)$ be the top element of a list X, i.e., ℓ_1 and $cdr(X)$ be the remaining list of X except for the top element, i.e., $[\ell_2, \ldots, \ell_n]$.

$$\begin{aligned}
Travel(\$(\mathrm{X})) &\stackrel{\mathrm{def}}{=} car(\$(\mathrm{X})) \ ; \ Travel(cdr(\$(\mathrm{X})))\\
Travel([]) &\stackrel{\mathrm{def}}{=} 0\\
Star(\$(\mathrm{X})|h) &\stackrel{\mathrm{def}}{=} (car(\$(\mathrm{X})) \ ; \ h) \ ; \ Star(cdr(\$(\mathrm{X}))|h)\\
Star([]|h) &\stackrel{\mathrm{def}}{=} 0
\end{aligned}$$

Let h be an element of $\mathcal{L}$ and X be a list of host names in $\mathcal{L}$. For example, $Travel(\$(\texttt{SNMP-AGENT}))$ allows an agent to travel around the hosts specified in the SNMP-AGENT list consisting of database server names on a sub-network. $Star(\$(\texttt{SNMP-AGENT})|h)$ corresponds to a star-shaped route, which allows an agent to go back and forth between the destinations specified in the SNMP-AGENT list and a given base host specified as h as the order of the list. To illustrate the transition defined in Definition 3.2, we show a transition of $Star(\$(\texttt{SNMP-AGENT})|h)$ in $\$(\texttt{SNMP-AGENT}) = [a, b, c]$ as follows.

$$\begin{aligned}
Star(\$(\texttt{SNMP-AGENT})|h) \quad & is \quad Star([a,b,c]|h)\\
& \stackrel{\mathrm{def}}{=} (a \ ; \ h) \ ; \ Star([b,c]|h)\\
& \xrightarrow{a} h \ ; \ Star([b,c]|h)\\
& \xrightarrow{h} Star([b,c]|h)\\
& \stackrel{\mathrm{def}}{=} (b \ ; \ h) \ ; \ Star([c]|h)\\
& \xrightarrow{b} h \ ; \ Star([c]|h)\\
& \xrightarrow{h} Star([c]|h)
\end{aligned}$$

In this framework, itineraries required by task requests from its external environment are written as terms of the language in $\mathcal{E}$. The terms of $\mathcal{E}$ are not executable, unlike those of $\mathcal{S}$, but can specify discretionary or loose itineraries, where the agent, to which a task is assigned, can omit or repeat visits to some destinations, by using non-deterministic operators. We show some itineraries as follows:

$$\begin{aligned}
Tour(\$(\mathrm{X})|h) &\stackrel{\mathrm{def}}{=} Tour'(\$(\mathrm{X})) \ ; \ h\\
Tour'(\$(\mathrm{X})|h) &\stackrel{\mathrm{def}}{=} car(\$(\mathrm{X})) \ \% \ Tour'(cdr(\$(\mathrm{X}))|h)\\
Tour'([]) &\stackrel{\mathrm{def}}{=} 0\\
Hub(\$(\mathrm{X})|h) &\stackrel{\mathrm{def}}{=} (car(\$(\mathrm{X})) \ ; \ h) \ \% \ Hub(cdr(\$(\mathrm{X})))\\
Hub([]|h) &\stackrel{\mathrm{def}}{=} 0
\end{aligned}$$

where $Tour(\$(\mathrm{X})|h)$ is a route among the hosts specified in list $\$(\mathrm{X})$ but does not require any movement order. When a task has $Tour(\$(\mathrm{X})|h)$ as its required itinerary, the agent, by which the task is carried, is required to visit and perform the task at all the hosts specified in $Tour(\$(\mathrm{X})|h)$ in any movement order.

B. Algebraic Order Relation

Next, we formulate an algebraic order relation that is suitable for selecting one of the agents whose itineraries can satisfy the requirement of a given task request. It is based on the concept of bisimulation [15].

Definition 3.3: A binary relation $\mathcal{R}^n$ ($\mathcal{R} \subseteq (\mathcal{E} \times \mathcal{S}) \times \mathcal{N}$) is an n*-itinerary* prebisimulation, where $\mathcal{N}$ is the set of natural numbers, if whenever $(E, S) \in \mathcal{R}^n$ where $n \geq 0$, then the following hold for all $\ell \in \mathcal{L}$ or τ.

(i) if $E \xrightarrow{\ell} E'$ then there is an S' such that $S \xrightarrow{\ell} S'$ and $(E', S') \in \mathcal{R}^{n-1}$

(ii) $E \, (\xrightarrow{\tau})^* \, E'$ and $(E', S) \in \mathcal{R}^n$

(iii) if $S \xrightarrow{\ell} S'$ then there exist E', E'' such that $E \, (\xrightarrow{\tau})^* E' \xrightarrow{\ell} E''$ and $(E'', S') \in \mathcal{R}^{n-1}$

where $E \sqsupseteq_n S$ if there exist some n-itinerary prebisimulations such that $(E, S) \in \mathcal{R}^n$. We call $\sqsupseteq_n$ n*-itinerary* order. □

Here we briefly explain the above definition. (i) means that E requires S to migrate to all the hosts specified in E in the same arrival order specified in E. If E has selective branches, such as $E_1 + E_2$, then S has the same branches and all its branches satisfy their corresponding branches in E in the same way. (ii) means that if E contains non-deterministic branches, such as $E_1 \# E_2$ and $E_1 \% E_2$, then S satisfies at least one of them. (iii) means that all the migrations specified in S must be the itinerary of E. That is, the informal meaning of $E \sqsupseteq_n S$ is that S is included in one of the permissible itineraries specified in E.

$\mathcal{R}^n$ is a family of relations indexed by a non-negative time value n. That is, in $E \sqsupseteq_n S$, n corresponds to the number of movements of the agent that can satisfy E. We show several algebraic properties of the order relation below. Several papers on performance evaluations of mobile agents [8], [10] have reported that the cost of migrating a Java-based mobile agent between hosts connected through a network faster than 10 Mbps Ethernet is dependent on the (un)installation at the source host and the destination host, including the (de)serialization of the agent, rather than the latency of transmitting the serialized agent over a network. That is, the number of agent migrations greatly affects the overall cost of mobile agent-based computing. As a result, if one or more mobile agents can satisfy the itinerary required by a given task request, we should select one of the most efficient agents according to the number of agent migration over a network. Hence, we need to select S_i whose n is the least among all S_j, which can hold $E \sqsupseteq_n S_j$.

When the domain of $\sqsupseteq_n$ is limited to $\mathcal{S} \times \mathcal{S}$, we can directly obtain that $S \sqsupseteq_n S$, and if $S_1 \sqsupseteq_n S_2$ and $S_2 \sqsupseteq_n S_3$ then $S_1 \sqsupseteq_n S_3$. Hence $\sqsupseteq_n$ is a preorder relation.

Proposition 3.4: Let $n_1 \leq n_2$. If $E \sqsupseteq_{n_1} S$, then $E \sqsupseteq_{n_2} S$ □

Proposition 3.5: Let $E_1, E_2 \in \mathcal{E}$, $S_1, S_2 \in \mathcal{S}$ and $E_1 \sqsupseteq_{n_1} S_1$ and $E_2 \sqsupseteq_{n_2} S_2$. Then we have $E_1 \mathrel{;} E_2 \sqsupseteq_{n_1+n_2} S_1 \mathrel{;} S_2$ and $E_1 + E_2 \sqsupseteq_{max(n_1,n_2)} S_1 + S_2$. □

The above properties mean that if the itineraries of an agent can satisfy the requirements of a task, then their combination by using ; or + can still satisfy the requirements. They are important because they can reduce the cost of comparing itineraries. There are some basic examples of $\sqsupseteq_n$ below.

$$(a \mathbin{\%} b \mathbin{\%} c) \mathrel{;} h \quad \sqsupseteq_4 \quad c \mathrel{;} a \mathrel{;} b \mathrel{;} h$$

where the right side requires an agent to migrate among three hosts a, b, c in indefinite order and then return to host h. When the left side is changed to $a \mathrel{;} b \mathrel{;} c \mathrel{;} h$, the relation is still preserved, but when the left side becomes $a \mathrel{;} b \mathrel{;} h$ or $a \mathrel{;} h \mathrel{;} b \mathrel{;} h \mathrel{;} c \mathrel{;} h$, the relation is not preserved.

$$((a \mathrel{;} b \mathrel{;} c) \mathbin{\&} h^{\star}) \mathrel{;} h \quad \sqsupseteq_6 \quad Star([a,b,c]|h)$$

In the above inequality, the left side allows an agent to drop in at host h during the itinerary $a \mathrel{;} b \mathrel{;} c$ and then finish its movement at host h. The right is a star-shaped route between three destinations $[a,b,c]$ and host h can satisfy the left side. Note that the left side can permit $Travel([a,b,c]) \mathrel{;} h$ but not $Star([b,a,c]|h)$.

IV. Design and Implementation

This section presents a prototype implementation of our approach. We tried to keep the implementation within the approach as much as possible. The current implementation is built on MobileSpaces.[2] The system can be run on any computer with a JDK 1.2-compatible Java runtime system that can migrate agents over a network using a TCP-based agent migration protocol. To make mobile agents as small as possible, this approach delegates the selection of mobile agents to a mechanism, called AgentPool, deployed at more than one host on a network.

A. Mobile Agents

Each mobile agent is implemented as an instance of a subclass of abstract class `ItineraryAgent`.

```
public class ItineraryAgent
  extends MobileAgent {
  // registering the itinerary specified as r
  void setRoute(Route r) throws
    IllegalSyntaxException .. { ... }
  // migrating to the host specified as h
  void moveTo(Host h) throws
    IllegalHostException,
      NoSuchHostException  ... { ... }
  // migrating to the next host specified in
  // its itinerary
  void moveToNext() throws
    MultiplePossibleHostsException,
      NoSuchHostException  ... { ... }
  // asking the possible destinations in
  // the next migration
  Host[] getPossibleHosts() ... { ... }
  ...
}
```

[2]Details about MobileSpaces are given in [20]. The approach presented in this paper, except for the application of the approach discussed in Section V, is independent of the system and can thus work with other Java-based mobile agent systems.

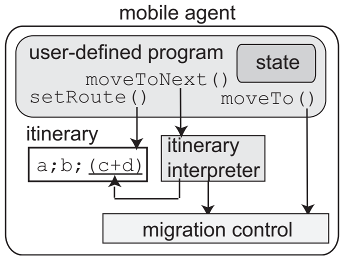

Fig. 2. Structure of a mobile agent.

Figure 2 shows the basic structure of a mobile agent. The itinerary of each mobile agent is denoted as a term of $\mathcal{S}$. It can be statically defined in the agent by invoking the `setRoute` method, or be dynamically given by an agent host, in which the agent is stored, as an explicit parameter.

```
setRoute(new Route("a;b;(c+d)"));
```

where `a;b;(c+d)` is an itinerary attached to the mobile agent and means that the agent migrates to host `a` and then to host `b`. Next, the agent can select either host `c` or `d` according to the result of its own processing. This approach restricts mobile agents from straying from their itinerary they registered with themselves. Each mobile agent can migrate itself over a network by using the following two approaches.

- The first approach allows each agent to move along the itinerary registered with itself. Each agent has a lightweight interpreter for the language in $\mathcal{S}$. When the agent invokes the `moveToNext()` method, the interpreter evaluates the agent's next destination from the itinerary and automatically moves the agent to the destination. However, if the itinerary contains one or more candidate destinations combined by the selective operator +, the invocation of the method throws a `MultiplePossibleHostsException`. The agent gets all the destinations that it can move to at the next hop by invoking the `getPossibleHosts()` method and moves to one of them by invoking the `moveTo(dst)` method with the selected destination specified as `dst`. For example, suppose that an agent registers `a;b;(c+d)` as its own itinerary. As shown in Fig. 3, it performs the `moveToNext()` method twice for two hops; from the current host to `a` and then from host `a` to `b`. Next, it can select either `c` or `d`, after which it performs the `moveTo(dst)` method with the name of the selected destination as the method's argument.
- The second approach permits an agent to control its mobility within its itinerary. That is, an agent decides its next destination and then migrates itself to the destination

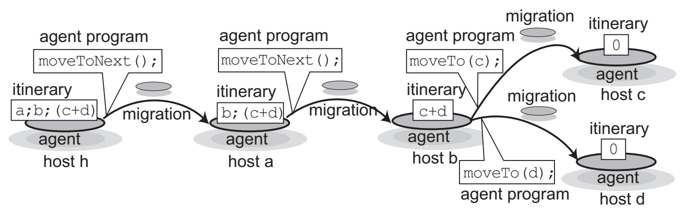

Fig. 3. Following-itinerary movement of a mobile agent with itinerary specified as a;b;(c+d).

by invoking the moveTo(dst) method where dst corresponds to the destination, only if the destination is specified in its itinerary. Otherwise, the method throws an exception, named IllegalHostException. For example, an agent whose itinerary is a;b;(c+d) can invoke the moveTo() method with a and then b to move to host a and then to b as shown in Fig. 4. Next, it can invoke the same method with either c or d.

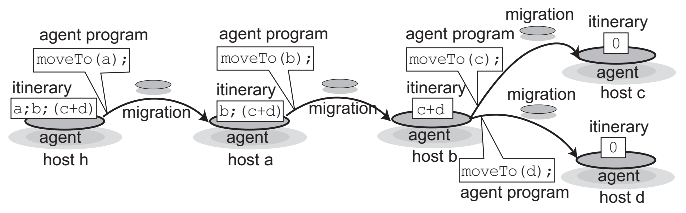

Fig. 4. Autonomous movement of a mobile agent whose itinerary is specified as a;b;(c+d).

In both the above approaches, when the movement of a mobile agent deviates from the itinerary registered by invoking the setRoute() method, it is constrained and the IllegalHostException is thrown to the agent. Each agent can explicitly change its itinerary by invoking the setRoute() method while it is moving, but the new itinerary becomes available after it returns to a agent pool.

B. Agent Pool

As mentioned previously, the key idea of the approach is to provide a variety of small mobile agents specialized for their particular itineraries and tasks rather than a few general-purpose agents for the reason of performance and security. Each agent pool is a place for storing and selecting idle mobile agents, as shown in Fig. 5. It is also responsible for receiving a task request from its external environment and then assigning the task to a suitable mobile agent. Tasks requested from the external environment are various, but the order of movement to the hosts at which each task must be performed is always written in $\mathcal{E}$.

Some readers may think that mobile agents should maintain their itineraries in explicit parameters, which can be overwritten by the external systems, and they should be dynamically given their itineraries by agent pools. However, our approach permits mobile agents to statically have their own itineraries as well as to be dynamically defined by themselves or the agent pools. This is because the approach should not assume the implementations of mobile agents. For example, some mobile agents may follow their own mobility control so that their itineraries should be treated as just specifications about their mobilities. The current implementation of agent pools stores idle mobile agents as well as only their itineraries.

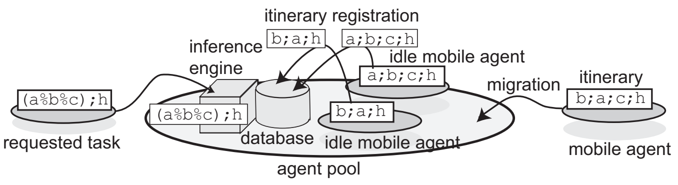

Fig. 5. Agent pool.

Here, we explain the selection algorithm of the current implementation[3]. Each agent pool maintains a repository database about itineraries. When it receives a task request from its external environment, each agent pool extracts the required itinerary from the request and selects one of the agents whose itineraries can satisfy the required itinerary among the itineraries stored inside the pool. To do this, the agent pool compares the required itinerary written in $\mathcal{E}$ with each of the itineraries of the agents written in $\mathcal{S}$ by directly using the order relation $\sqsupseteq_n \subseteq \mathcal{E} \times \mathcal{S}$ in Definition 3.3. First, it transforms each of its stored agent itineraries into a transition tree whose arcs are the labeled transitions in Definition 3.2. Also, in the same way it transforms the required itinerary into a transition tree whose arcs correspond to ℓ-transitions or τ-transitions in Definition 3.2. Next, it judges whether or not the former tree can satisfy the latter tree by matching the two trees according to the definition of the order relation as follows:

(1) if each node in one of the two trees has arcs corresponding to ℓ-transitions, then the corresponding node in the other tree can have the same arcs and the sub-nodes derived through the two trees' matching arcs can still satisfy either (1) or (2).
(2) if each node in the tree derived from the required itinerary has one or more arcs corresponding to τ-transitions, then at least one of the nodes derived through the arcs and the corresponding node in the tree derived from the agent's itinerary can still satisfy (1) or (2).
(3) if neither (1) nor (2) is satisfied, the agent pool backtracks from the current nodes in the two trees and tries to apply (1) or (2) to their two backtracked nodes.

Here, we illustrate a matching between a;((b;(c+d))#d) in E and a;b;(c+d) in S in Figure 6. Then, the agent pool assigns the request to the agent whose itinerary can satisfy the above conditions. If more than one agent satisfies the required itinerary, it selects the agent with the least number of agent migrations over a network, which is n of $\sqsubseteq_n$ in Definition 3.3. The current implementation can expand $E^\star$

[3]The current implementation was not optimized for performance, but it can handle the example itineraries presented in this paper within a few milliseconds.

by using the structural congruence presented in Definition 3.5 as lazily as possible. All the itineraries written in $\mathcal{E}$ or $\mathcal{S}$ can be transformed into finite trees, called image-finite in the literature of process algebras [5], [15].

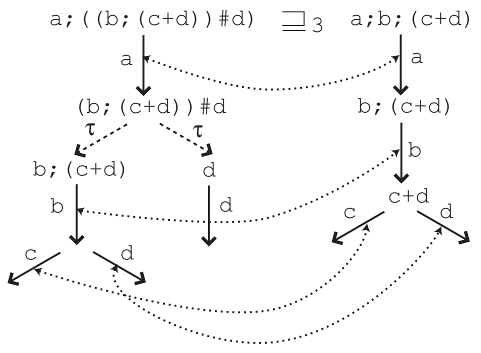

Fig. 6. Matching two terms `a;((b;(c+d))#d)` and `a;b;(c+d)`.

V. Applications

The approach presented in this paper can select mobile agents according to their itineraries, instead of any of their application-specific behaviors. However, this limitation is not serious in the development of typical applications of mobile agents, such as remote information retrieval and filtering and network management tasks, where a mobile agent contains code to define its application-specific task to be performed whenever it arrives at one of its destinations. That is, mobile agents, which itinerate among multiple hosts, often execute the same code at each of the hosts that they visit. This means that the separation of concerns studied in the literature of aspect-oriented programming [12] is effective in the development of of mobile agents.

In our previous papers [23], [25], we proposed a methodology for composing a mobile agent from two parts: application-specific and mobility control. The former part defines its own application-specific task to be performed at each of the hosts it visits. The latter part defines a particular itinerary on its target network, so that it can efficiently travel among its multiple destinations. However, since the previous papers aimed at applying the separation of concerns into mobile agent technology, they did not provide any mechanisms for matchmaking task agents and carrier agents.

This section presents that the approach is useful as such a mechanism. This implementation constructed the two parts as mobile agents, called *task* agents and *carrier* agents. Both agents were implemented as hierarchical mobile agents in MobileSpaces [20], which can hierarchically organize multiple mobile agents. In the MobileSpaces system, a mobile agent can dynamically contain other mobile agents and can migrate to other mobile agents as a whole with all its inner agents. Each carrier agent is a container of more than one task agent and carries its task agents over a network according to its own itinerary specified as a term of $\mathcal{S}$. On the other hand, each *task* agent defines its own application-specific task to be performed at each of the hosts it visits. It also has an attached term of $\mathcal{E}$ to specify the hosts at which application-specific task should be performed. Therefore, the former agent can be reused in any application and the latter can be used in any network. Since a carrier agent can be optimized to a particular itinerary on its target network, it can efficiently navigate its task agents among the hosts that the agents must visit. It is independent of any application-specific tasks.

Our approach provides Java-based abstract classes, called `TaskAgent` and `CarrrierAgent`, that allow us to easily define advanced task agents and carrier agents by extending the classes.

```
public class TaskAgent extends MobileAgent {
  // registering its requiring itinerary
  void setRoute(Route r)
    throws IllegalSyntaxException ... { ... }
  // callback method invoked after the agent
  // arrives at one of its destinations.
  void arrivedAt(Host here);
  // callback method invoked before the agent
  // leaves from the current host.
  void depaturingFor(Host dst);
  // callback method invoked after the agent
  // visits all the hosts in its itinerary
  void finished(Route r);
  ...
}
```

Each agent defines its task in the `arrivedAt()` method. When arriving at an agent pool, the task agent gives the pool the required itinerary along which a carrier agent is required to carry itself by performing the `setRoute()` method with an itinerary specified in $\mathcal{E}$. The agent pool selects a suitable carrier agent and then migrates the task agent into the selected agent. Upon arrival at a host, the carrier agent invokes the `arrivedAt()` method of its task agent to instruct it to do something for a given time period at the host. After receiving a certain event from all the task agents or after the period has elapsed, the carrier agent invokes the `depaturingFor()` method with the address of the next host and then moves itself and its task agents to the next destination on its itinerary. For reasons of security, all agents must be authenticated by the agent pool of a sub-network on behalf of the sub-network. This is helpful in network management systems whose hosts may have limited CPU power and memory. Since a sub-network may explicitly prohibit any task agent from visiting its hosts, task agents must be carried by a carrier agent managed by the agent pool of the sub-network. Therefore, a task agent alone cannot migrate to all the hosts, even if it knows the addresses of its target hosts in the sub-network.

We must offer a variety of carrier agents specific to their own itineraries in more than one agent pool on the target network. However, due to the lack of space, this section illustrates only two carrier agents, defined by `CarrierAgent1` and `CarrierAgent2` classes respectively.

```
public class CarrierAgent1
  extends CarrierAgent {
  public CarrierAgent1() {
    // registering its possible itinerary
    setRoute(new Route("h;a;b;c;d;h"));
  }
  // invoked at the completion of the task
  // agent's processing at the current host
  public void done() throws
    MultiplePossibleHostsException .. {
    moveToNext();
  }
  ...
}
```

`CarrierAgent1` can travel along a tour route, `h;a;b;c;d;h` written in $\mathcal{S}$ and `CarrierAgent2` can move along a star-shaped route, `h;a;h;b;h;c;h;d;h`. Figure 7 shows the two agents' itineraries.

```
public class CarrierAgent2
  extends CarrierAgent {
  public CarrierAgent2() {
    setRoute(
      new Route("h;a;h;b;h;c;h;d;h"));
  }
  public void done() throws
    MultiplePossibleHostsException .. {
    moveToNext();
  }
  ...
}
```

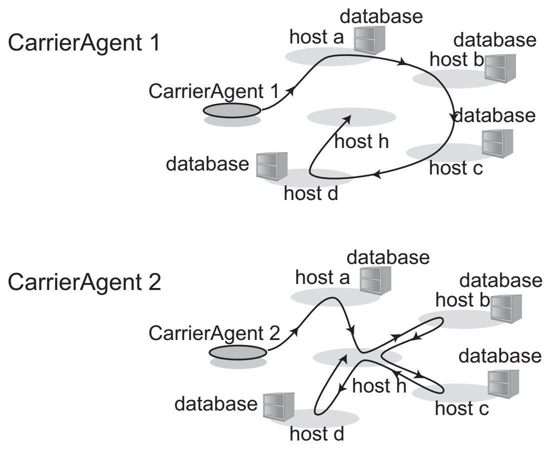

Fig. 7. Itineraries of two carrier agents (CarrierAgent1 and CarrierAgent2).

Next, suppose that a task agent has its required itinerary specified as `h;((a%b%c%d)&h^*);h`, where `h^*` denotes h^* in the language $\mathcal{E}$.

```
  setRoute(new Route("h;((a%b%c%d)&h^*);h"));
```

When an agent pool receives the task agent, it selects a suitable carrier agent whose itinerary can satisfy `h;((a%b%c%d)&h^*);h` among the idling agents storing inside it, as shown in Figure 8. In the above example, the two carrier agents can satisfy the required itinerary of the task agent. Since `CarrierAgent1` has fewer agent migrations than `CarrierAgent2`, the agent pool selects the former carrier agent and moves the task agent into it.

After receiving the task agent, the carrier agent carries it from host to host according to its own itinerary. When it arrives at one of the destinations, it issues certain events to invoke the `arrived()` method of the task. The task agent performs its application-specific task, such as information searching and filtering from the database on its visiting host. When it finishes the task that should be performed at the current host, it invokes the `done()` method to instruct the carrier agent to carry it to the next destination. The approach presented in this paper can strictly select one of the most suitable carrier agents, since it provides a theoretical and practical mechanism for comparing itineraries of the carrier agents.

We have obtained a preliminary measurement of the cost of migrating a carrier agent over a sub-network of the cluster system. Note that the system is just a prototype implementation; hence it is not optimized for efficient agent migration. Actually, the total size of the carrier agent containing one of the task agents is about 8 KB (zip-compressed) and it is only 20 percent greater than the size of a self-contained task agent that controls its own itinerary. This is a small increase in size if we take into account the amount of data such agents can collect from clusters. The cost of detecting a carrier agent in an agent pool is less than 10 msec, although the current algorithm for agent selection in agent pools was not optimized for performance.[4] The total cost of management depends on application-specific tasks performed at clusters rather than agent migration. After receiving a task agent at the agent pool of the sub-network, the carrier agent travels straightly around four clusters and then returns to the agent pool of the sub-network, where the clusters and the pool are Pentium III-800 MHz computers connected using a 100-Mbps Ethernet. The itinerary of the carrier agent is statically defined and corresponds to five hops. The round-trip time of the agent is about 480 msec. where the per-hop latency of agent migration for the task agent using the carrier agent is at most 25 percent greater than the per-hop latency of a self-contained task agent.

VI. Related Work

Many mobile agent systems have been developed over the last few years, for example, Aglets [13], Telescript [32], and MobileSpaces [20]. Several researchers have explored approaches for dynamically assigning tasks to non-mobile multi-agents, for example, contract-net protocol [28] and KQML [6]. Since most of the existing approaches can select agents suitable to perform tasks based on their application-specific behaviors, they cannot be directly applied to mobile agents, because not only the application-specific behaviors of mobile agents but also their itineraries may seriously affect their success and efficiency. However, since mobile agents are often treated as just an implementation of distributed systems, there have been few attempts to select mobile agents. Among them, Plangent [17] is a mobile agent system. In Plangent an agent can dynamically generate a plan for acquisition of the knowledge that users need and then it migrates and executes its application-specific actions according to the plan. When the agent cannot gain the knowledge because the plan is invalid, it generates new plans by using new knowledge at the

[4]The cost includes communication between the agent pool and mobile agents. It was measured when the agent host runs on Pentium III-800MHz and Windows 2000.

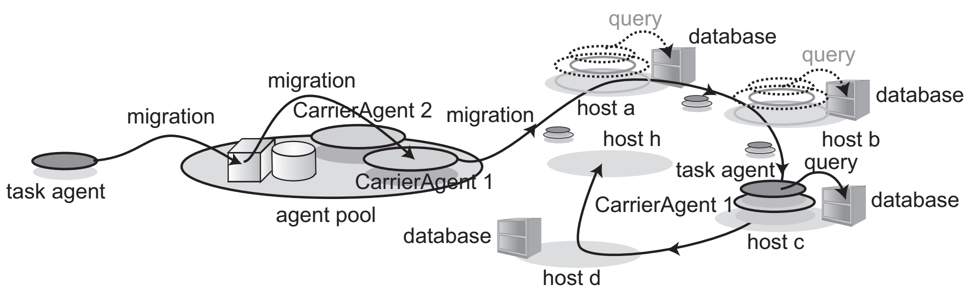

Fig. 8. Selection of a carrier agent (CarrierAgent1).

current hosts. However, the planning functionality of Plangent does not target the mobility of agents and cannot always generate valid plans. On the other hand, our approach offers a theoretical foundation for the the selection of mobile agents and allows us to strictly judge whether or not the movements of agents can satisfy the users' requirements.

Several papers also introduce the notion of an itinerary pattern to shift the responsibility for navigation from an application-specific agent to framework libraries or meta-level specifications, for example see [1] and [30]. The notion is based on design patterns studied in the literature of software engineering. Since it should be used in the development of agent software, it does not offer any dynamic selection mechanisms for selecting mobile agents.

Several papers have explored theoretical models for reasoning about mobile agents, for example, Mobile UNITY [14], the Join calculus [7], Ambient calculus [3], Distributed π-calculus [18], and Nomadic π-calculus [29]. Mobile UNITY is an extension of UNITY, which is an existing formal model for specifying distributed systems, with the expressiveness of the movement of components, including mobile computers and mobile software. Since it is designed for specifying variable and conditional assignment statements in programs by incorporating UNITY, it cannot extract and reason about only the itineraries of mobile components.

Most existing formal models for mobile agents are based on process calculi (or called process algebras), like ours. Ambient calculus [3] allows mobile agents (called ambients in the calculus) to contain other agents and to move with all its inner ambients. The calculus must always model the mobility of agents as a navigation along a hierarchy of agents, whereas itineraries of real mobile agents may be complicated. The join-calculus [7] also introduces the notion of named locations which form a tree and the mobility of an agent is modeled as a transformation of subtrees from one part of the tree to another. Distributed π-calculus and Nomadic π-calculus are extensions of π-calculus with the notion of locations. Existing process calculus-based models are just theoretical frameworks for reasoning about the whole computation of mobile agents. As far as the author knows, no existing calculi provide any preorder relations for the selection of mobile agents.

Although MobileSpaces [20], which serves as the basis for the framework presented in this paper, can dynamically adapt its functions and structures to changes in the environments, its goal is to provide a general platform for executing and migrating distributed applications. We also presented an architecture for building several agent migration protocols in our previous papers [21], [22]. That architecture is hierarchically organized like the notion of a protocol stack in existing data transmission protocols. It can customize network processing for agent migration embedded in a mobile agent runtime system. We presented a mobile agent-based framework for network management in other previous papers [23], [25]. That framework divides a mobile agent into two parts: its mobile control component and its application-specific component. Although it offers a basis for the separation of mobile agents presented in Section V, its purpose is to enhance the reusability of mobile agents. It is specific only to network management. These previous papers do not present any mechanisms for selecting mobile agents suitable to perform tasks, unlike this paper.

VII. Future Work

There are several open issues. Finally, we would like to mention some future research directions. The specification language presented in this paper addresses the movement of agents, but we are interested in extending the language with the expressiveness of application-specific behaviors, locations, the cost of agent migration by incorporating our previous process algebra for distributed systems studied in [19]. The performance of the current implementation of our agent selection algorithm presented in Section IV on the order relation is not yet satisfactory and we believe that existing optimizations for bisimulation, for example see [5], can be easily applied to the relation presented in this paper. This paper does not discuss any coordination among multiple mobile agents, but we are interested in developing a mechanism for assigning a task to one or more mobile agents. Also, we plan to establish an axiomatic system based on the order relation for the performance improvement of the agent selection. The approach presented in this paper was initially designed as a policy-based control system for mobile agents. We are interesting in applying the framework to our mobile agent-based systems, for example, a location-aware infrastructure for ambient intelligence studied in our previous paper [24] and a software testing framework for networked mobile computing in our another paper [27].

VIII. Conclusion

This paper presented a general approach to selecting mobile agents suitable to perform tasks according to their itineraries. The approach offers a process algebra-based language and an algebraic order relation between terms of the language as a theoretical foundation for the selection of mobile agents. The language can strictly specify the itineraries that mobile agents can migrate along and are required to migrate along. The relation can decide whether or not the possible itinerary of each mobile agent can satisfy the itinerary required by a requested task. A prototype implementation of the approach has been built on a Java-based mobile agent system, called MobileSpaces. Each mobile agent is implemented as a collection of Java objects with its own itinerary written in the language and can travel from host to host along the itinerary. Agent selection provides a mechanism for storing idle agents and selecting one of the most suitable and efficient agents when it receives a task request written in the language from its external environment. The approach presented in this paper focuses on a serious problem of existing mobile agent technology. We believe that the approach provides a general solution to this problem and enables us to strictly specify and select suitable mobile agents according to their mobilities. Since the approach itself is designed independently of any mobile agent platforms, it can easily specify and select other existing mobile agents.

Acknowledgements: We are grateful to the anonymous reviewers for many significant comments.

References

[1] Y. Aridor, and D.B. Lange, Agent Design Patterns: Elements of Agent Application Design, Proceedings of Second International Conference on Autonomous Agents (Agents'98), pp. 108-115, ACM Press, 1998.

[2] J. C. M. Beaten and J. A. Bergstra, Process Algebra, Cambridge University Press, 1990.

[3] L. Cardelli and A. D. Gordon: Mobile Ambients, Proceedings of Foundations of Software Science and Computational Structures, LNCS, vol. 1378, pp. 140-155, 1998.

[4] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, G. Tsudik, Itinerant Agents for Mobile Computing, IEEE Personal Communications, vol.2, no.5, pp.34-49, 1995.

[5] R. Cleaveland and O. Sokolsky, Equivalnce and Preorder Checking for Finite-State Systems, in Handbook of Process Algebra (eds. J. A. Bergstra, A. Ponse, S. A. Smolka), pp.391-424, North-Holland, 2001.

[6] T. Finin, Y. Labrou, and J. Mayfield, KQML as An Agent Communication Language, in Software Agents, MIT Press, 1997.

[7] C. Fournet, G. Gonthier, J. Levy, L. Marnaget, and D. Remy: A Calculus of Mobile Agents, Proceedings of CONCUR'96, LNCS, vol. 1119, pp.406-421, Springer, 1996.

[8] R. S. Gray, et al, Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task, Proceedings of International Conference Mobile Agents (MA'2001), LNCS vol.2240, pp.198-212, Springer, December, 2001.

[9] C. A. Gunter, S. M. Nettles, and J. M. Smith, The SwitchWare Active Network Architecture, IEEE Network, special issue on Active and Programmable Networks, vol. 12, no. 3, 1998.

[10] D. Hagimont and L. Ismail, A Performance Evaluation of the Mobile Agent Paradigm, Proceedings of International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99), pp.306-313, ACM Press, 1999.

[11] A. Karmouch, Mobile Software Agents for Telecommunications, IEEE Communication Magazine, vol. 36 no. 7, 1998.

[12] G. Kiczales, et al, Aspect-Oriented Programming Proceeding of European Conference on Object-Oriented Programming (ECOOP'97), LNCS, vol. 1241, Springer, 1997.

[13] B. D. Lange and M. Oshima: Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.

[14] P.J. McCann, and G.-C. Roman, Compositional Programming Abstractions for Mobile Computing, IEEE Transaction on Software Engineering, vol. 24, no.2, 1998.

[15] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[16] R. Milner, Communicating and mobile systems: the π-calculus, Cambridge University Press, 1999.

[17] A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden, PLANGENT: An Approach to Making Mobile Agents Intelligent, IEEE Internet Computing, vol.1, no.4, pp.55-57, 1997.

[18] J. Riely and M. Hennessy, Distributed Processes and Location Failures, ICALP'97, LNCS, vol. 1256, pp.471-481, Springer, 1997.

[19] I. Satoh and M. Tokoro, Time and Asynchrony in Interactions among Distributed Real-Time Objects, Proceedings of European Conference on Object-Oriented Programming (ECOOP'95), LNCS vol.952, pp.331-350, Springer, 1995.

[20] I. Satoh, MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, Proceedings of International Conference on Distributed Computing Systems (ICDCS'2000), pp.161-168, IEEE Computer Society, April, 2000.

[21] I. Satoh, Adaptive Protocols for Agent Migration, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2001), pp.711-714, IEEE Computer Society, April, 2001.

[22] I. Satoh, Network Processing of Mobile Agents, by Mobile Agents, for Mobile Agents, Proceedings of Workshop on Mobile Agents for Telecommunication Applications (MATA'2001), LNCS, vol.2146, pp.81-92, Springer, 2001.

[23] I. Satoh, A Framework for Building Reusable Mobile Agents for Network Management, Proceedings of Network Operations and Managements Symposium (NOMS'2002), pp.51-64, IEEE Communication Society, April, 2002.

[24] I. Satoh, Physical Mobility and Logical Mobility in Ubiquitous Computing Environments, to appear in Proceedings of International Conference on Mobile Agents (MA'2002), LNCS, Springer, October, 2002.

[25] I. Satoh, Building Reusable Mobile Agents for Network Management, IEEE Transaction on Systems, Man and Cybernetics, vol.33, no.3(C), pp.350-357, 2003.

[26] Ichiro Satoh, Reusable Mobile Agents for Cluster Computing, Proceedings of IEEE International Conference on Cluster Computing (Cluster'2003), pp.270-279, IEEE Computer Society, December 2003.

[27] I. Satoh, A Testing Framework for Mobile Computing Software, IEEE Transaction on Software Engineering (accepted), vol.29, No. 12, 2003.

[28] R. G. Smith, The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, IEEE Transactions on Computers, 1104-1113, 1980.

[29] P. Swell, P. T. Wojciechowski, and B. C. Pierce, Location-Independent Communication for Mobile Agents: A Two-Level Architecture, Workshop on Internet Programming Languages, LNCS, vol. 1686, Springer, 1998.

[30] Y. Tahara, A. Ohsuga, and S. Honiden: Agent System Development Method Based on Agent Patterns, Proceeding of International Conference on Software Engineering (ICSE'99), pp.356-367. IEEE Computer Society, 1999.

[31] D. L. Tennenhouse et al., A Survey of Active Network Research, IEEE Communication Magazine, vol. 35, no. 1, 1997.

[32] J. E. White, Telescript Technology: Mobile Agents, General Magic, 1995.