Vol. 47 No. 4 IPSJ Journal Apr. 2006

Regular Paper

Logic-Based Mobile Agent Framework with a Concept of "Field"

Shinichi Motomura,[†] Takao Kawamura[†]
and Kazunori Sugahara[†]

A new logic-based mobile agent framework named Maglog is proposed in this paper. In Maglog, a concept called "field" is introduced. By means of this concept, the following functions are realized: (1) agent migration, which is a function that enables agents to migrate between computers, (2) inter-agent communication, which is indirect communication with other agents through the field, (3) adaptation, which is a function that enables agents to execute programs stored in the field. We have implemented Maglog in a Java environment. The program of an agent, which is a set of Prolog clauses, is translated into Java source code by our Maglog translator, and is then compiled into Java classes by a Java compiler. The effectiveness of Maglog is confirmed through descriptions of two applications: a distributed e-learning system and a scheduling arrangement system.

1. Introduction

Mobile agent technology is attracting attention as a key technology for developing distributed systems. For realization of mobile agent systems, the following functions need to be implemented:

(1) Agents should be able to migrate from one computer to another with data and programs.

(2) Agents should be able to communicate with other agents.

(3) Agents should be able to adapt themselves to environments such as the computers they belong to. Such adaptation is accomplished by absorbing data and programs from their environments.

Accordingly, a concept called "field" is proposed as a means of realizing the above functions in a simple manner.

Agents communicate with other agents indirectly through a field and adapt themselves to the environment by importing data and programs stored in the field. The functions realized by the field can be summarized as follows:

(1) Migration: A function that enables agents to migrate between computers.

(2) Inter-agent communication: Indirect communication with other agents through the field. That is, an agent is able to import data or programs stored in the field by other agents.

(3) Adaptation: A function that enables agents to execute programs stored in the field.

To implement a mobile agent system with the concept of field, programs that describe the behavior of the agent are written in Prolog in our system. Since Prolog is a logic programming language and has a powerful pattern-matching mechanism, agents are able to search for data and programs stored in fields easily. This powerful pattern-matching mechanism of Prolog is called "unification." Unifications between computers are realized to construct a mobile agent system.

This paper proposes a mobile agent framework named Maglog that implements the above-mentioned functions in a Java environment. Java is adopted because of its huge class libraries for building network applications. It should also be noted that Java's goal of "write once, run anywhere" is desirable for mobile agent systems.

Several mobile agent frameworks have been realized as sets of class libraries for Java, such as Aglets[1)], MobileSpaces[2)], and Bee-gent[3)]. Each of them, when used in combination with a Prolog interpreter written in Java, such as NetProlog[4)] or Jinni[5)], has some similarity to Maglog. The main difference between these combinations and Maglog is the class of mobility. They have weak mobility, since only their clause databases are migrated. In Maglog, all of the execution state, including the execution stack, can be migrated. That is to say, Maglog has strong mobility, and consequently agents in Maglog can backtrack and unify variables across the network. That makes programs in Maglog simple and understandable.

† Tottori University

Except for Maglog, MiLog[6] is the only logic-based framework with strong mobility. However, it does not have a concept simular to the field presented in this paper.

Flage[7] uses a similar concept to Maglog's fields, but Flage's fields cannot be used as a medium of synchronous communication between agents. Furthermore, in Flage, unifications between two fields are not supported.

2. Overview of Maglog

Figure 1 shows an overview of a mobile agent system in Maglog. In the figure, two computers (hereafter referred to as hosts) are connected to a network and agent servers are running on each of them to activate agents and to provide fields for them.

The remainder of this section describes the three basic components of Maglog, namely, agent, agent server, and field.

2.1 Agent

An agent has the following functions:

(1) Execution of a program that describes the behavior of the agent,
(2) Execution of procedures stored in a field where the agent is currently located,
(3) Communication with other agents through a field,
(4) Creation of agents and fields,
(5) Migration to another host in a network.

An agent of Maglog executes its program sequentially. The class of agent migration is strong migration, which involves the transparent migration of an agent's execution state as well as its program and data. In order to realize unifications between computers, Maglog supports strong mobility.

For creation of a child agent, a parent agent executes the following built-in predicate:

```
    create(AgentID,File,Goal)
```

In this predicate, `File` corresponds to the file-name in which the behavior of the agent is described. If the execution of the predicate is successful, an agent is created and its globally unique identifier `AgentID` is returned. The created agent immediately executes the goal specified by the argument `Goal` and disappears when the execution is accomplished.

An agent can obtain its identifier by executing the following built-in predicate.

```
    get_id(Agent)
```

Each agent contains Prolog program and its interpreter. The initial behavior of the agent is described in the Prolog program given by `File` in the predicate of its creation. Since Prolog treats programs and data identically, the agent behavior might be modified during execution.

Figure 2 shows an example of an agent's behavior. The program of agentA is assumed to contain a clause `in((clause(p(x),Y),assert(p(X):-Y)), fieldA)`. The behavior of agentA can be described as follows:

(1) agentA enters fieldA.
(2) agentA executes a predicate `clause(p(X),Y)` and retrieves a clause whose head matches `p(X)` from fieldA as a result. Here `Y` is bound to `q(X),r(X)` which is the body of the clause.
(3) agentA executes a predicate `assert(p(X):-Y)`, and then a clause `p(X):-q(X),r(X)` is added to its own program.

That is to say, an agent is able to import clauses from fields so that it can change its behavior dynamically.

The built-in predicate `in/2` will be described in Section 3.1. Here the notation `Name/Arity`

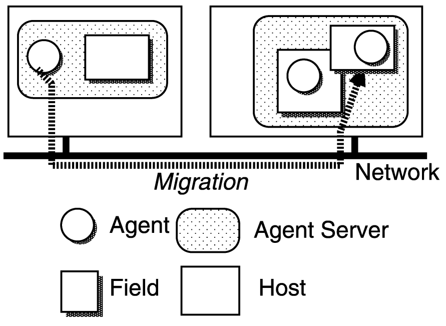


Fig. 1 Overview of a mobile agent system in Maglog.

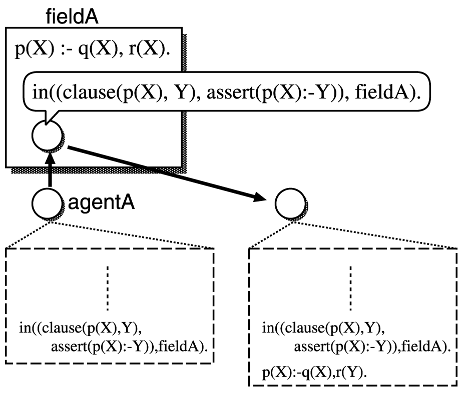

fieldA
p(X) :- q(X), r(X).
in((clause(p(X), Y), assert(p(X):-Y)), fieldA).
agentA
in((clause(p(X),Y), assert(p(X):-Y)),fieldA).
in((clause(p(X),Y), assert(p(X):-Y)),fieldA).
p(X):-q(X),r(Y).

Fig. 2 Dynamic change in a program that describes the behavior of the agent by asserting a new clause.

is the predicate indicator (hereafter referred as PredSpec) which refers to one or several predicates. `Name` and `Arity` correspond to the name of a predicate and its number of argument's respectively.

2.2 AgentServer

An agent server is a runtime environment for agents that provides required functions for agents. The above-mentioned predicates, such as `create/3` and `get_id/1`, are examples of functions.

An agent server creates and deletes agents. An agent server assigns an `AgentID` to the created agent. An `AgentID` consists of a host's IP address and the time at which the agent was created, and is thus globally unique. In addition, an agent server provides an agent migration function. When an agent migrates from hostA to hostB, the agent server on hostA suspends the agent's execution and transports the agent to hostB. After that, the agent server on hostB resumes execution of the agent.

An agent server also manages fields and provides functions that enable an agent to utilize them.

2.3 Field

A field is an object managed by an agent server to hold Prolog clauses, and is created when an agent executes the following built-in predicate:

```
    fcreate(Field)
```

If `Field` is an unbound variable, a field with a unique identifier is created, and its identifier is bound to the argument `Field`. If `Field` is a symbol, the action of this predicate depends on whether the field whose identifier is the symbol exists or not. If it does not exist, a field whose identifier is the symbol is created; otherwise, nothing is done.

Important features of Maglog realized through the concept of field will be described in the following section.

3. Features Realized through the Concept of Field

3.1 Predicate Library

An agent enters a field and executes a goal by using the following built-in predicate:

```
    in(Goal, Field)
```

The agent exits `Field` automatically whether the execution succeeds or not. This built-in predicate is re-executable; that is, each time it is executed, it attempts to enter the field and executes the next clause that matches `Goal`.

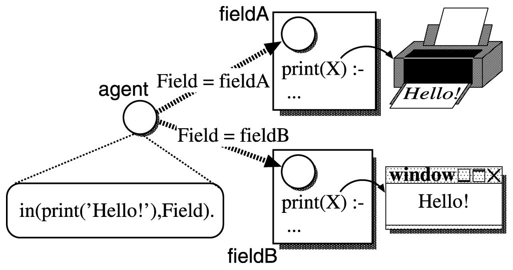


Fig. 3 Dynamic change in an agent's behavior according to the field.

When there are no more clauses to execute, this predicate fails.

When an agent enters a field, it imports the procedures of the field and combines them with its own procedures. Therefore, an agent does not need to contain all of the program by itself to solve a problem, but instead enters the fields that provide the necessary procedures. An agent can change its behavior dynamically according to the field it enters. In this way, an agent can adapt its behavior to its environment.

Figure 3 shows an example in which an agent executes different `print/1` predicates in fieldA and fieldB. The execution of the goal `print('Hello!')` sends the string "Hello!" to a printer when the agent is in fieldA; on the other hand, the same goal creates a new window containing the string "Hello!" when the agent is in fieldB, because fieldA and fieldB provide appropriate procedures for their output devices.

3.2 Inter-agent Communication

Agents entering the same field can be considered as forming a group. The procedures within the field are shared by the agents. Moreover, by adding or removing procedures within the field, agents can influence the behavior of other agents.

Updating of procedures in a field can be performed by means of the following built-in predicates:

```
    fasserta(Clause, Field)
    fassertz(Clause, Field)
    fretract(Clause, Field)
```

The first argument `Clause` of these predicates is a clause to be added or removed from the field specified by the second argument `Field`. `fasserta/2` inserts the clause in front of all the other clauses with the same functor and arity. Functor and arity mean the name of a predicate and its number of arguments, respectively. On the other hand, `fassertz/2` adds the clause after all the other clauses with the same functor

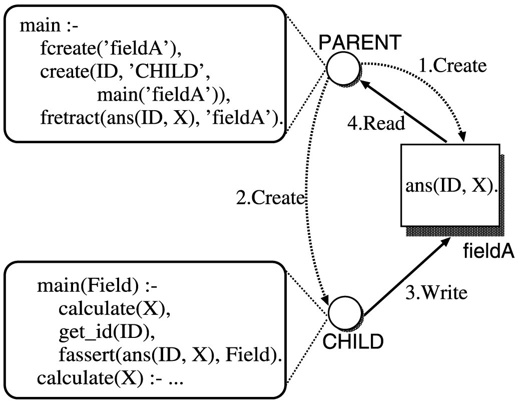


Fig. 4 Agents can communicate synchronously through a field.

and arity. `fretract/2` removes the next unifiable clause that matches the argument from the field. This built-in predicate is re-executable, that is, each time it is executed it attempts to remove the next clause that matches its argument. If there are no more clauses to remove, then this predicate fails.

By using these predicates, an agent can communicate with other agents not only asynchronously but also synchronously. An agent has two modes for execution of procedures stored in a field. In the fail mode, the execution fails when an agent attempts to execute or to retract a non-existent clause in a field. In the block mode, an agent that attempts to execute or to retract a non-existent clause in a field is blocked until another agent adds the target clause to the field. For agents in the block mode, a field can be used as a synchronous communication mechanism such as a tuple space in the Linda model[8)].

Figure 4 shows an example of synchronous inter-agent communication.

(1) `PARENT` creates `fieldA`.
(2) `PARENT` creates `CHILD` and makes it execute `main('fieldA')`. `PARENT` attempts to remove the clause that matches `ans(ID,X)` from `fieldA` and `PARENT` is blocked until a unifiable clause is added by `CHILD`.
(3) `CHILD` executes `calculate(X)` and the result is bound to `X`. The identifier of `CHILD` is bound to `ID` by the execution of the built-in predicate `get_id(ID)`. `CHILD` adds `ans(ID,X)` to `fieldA`.
(4) `PARENT` wakes up and removes `ans(ID,X)` from `fieldA`.

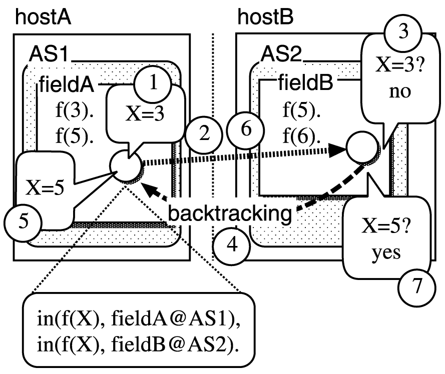


Fig. 5 Backtracking and unification between two hosts.

3.3 Agent Migration

Each agent server has a globally unique identifier composed of the server's IP address and defined name.

If the second argument of the predicates `in/2`, `fasserta/2`, `fassertz/2`, and `fretract/2` is specified in the form of `Field@ServerID`, the agent executing this predicate migrates to the host in which the agent server specified by `ServerID` runs, and enters `Field`. The agent returns to the host located before the migration automatically as it exits the field.

Figure 5 shows that the agent matches `f(X)` with clauses in two fields in hostA and hostB. As shown in Fig. 5, this attempt proceeds through performing the following steps and succeeds:

(1) An agent enters fieldA in hostA and executes the goal `f(X)`. Consequently, `X` is bound to 3, because `f(3)` is the first clause that matches `f(X)`.
(2) The agent migrates to hostB and enters fieldB.
(3) The agent executes the goal `f(3)`. This attempt fails, since there is no clause that matches `f(3)`.
(4) The agent returns to hostA and enters fieldA automatically.
(5) The agent attempts to execute the next clause that matches with `f(X)`. `X` is therefore bound to 5.
(6) The agent migrates to hostB and enters fieldB again.
(7) The agent executes the goal `f(5)`. This attempt succeeds, since the clause `f(5)` is in fieldB.

4. Implementation

We have implemented Maglog in a Java environment by extending PrologCafé[9], which is a Prolog-to-Java source-to-source translator system.

The program of an agent, which is a set of Prolog clauses, is translated into Java source code by our Maglog translator, and is then compiled into Java classes by a Java compiler. As mentioned in Section 2.1, an agent can import Prolog clauses from a field at run time. These clauses are interpreted by the Prolog interpreter included in an agent instead of being compiled into Java classes. An agent runs as a thread in a process named an agent server.

Agent servers have an XML-RPC interface, which is accessible from applications written in any other language with support for XML-RPC.

The following operations from other systems are available through XML-RPC:

(1) Create and kill agents,
(2) Create and delete fields,
(3) Assert clauses into fields and retract clauses from fields,
(4) Get a list of names of fields,
(5) Get a list of IDs of agents currently existing.

Figure 6 shows a screen-shot of the user interface program for manipulation of agent servers. It can create/kill agents and create/delete fields, and can browse both the contents of fields and the outputs of agents.

Implementation of Maglog features realized through the concept of field is described in the remainder of this section.

4.1 Predicate Library

As shown in **Fig. 7**, a field is implemented as a Java Hashtable; that is, procedures in a field are put into a hashtable. A key is PredSpec of a procedure, and the value is a set of objects representing the procedure whose predicate indicator is PredSpec.

When an agent executes a predicate in `in/2`, it searches for the predicate by specifying PredSpec from the hashtables of fields it is currently in, and interprets the located values.

In order to improve the execution rate, the concept of a static field is introduced into Maglog. It stores read-only procedures compiled into Java classes before the agent server to which the field belongs starts.

A static field is implemented as a Java Class Loader, which receives PredSpec and loads the bytecodes of the class for the corresponding procedure.

According to the experiments, an agent can execute a clause in a static field about 250 times faster than in an ordinary field.

4.2 Inter-agent Communication

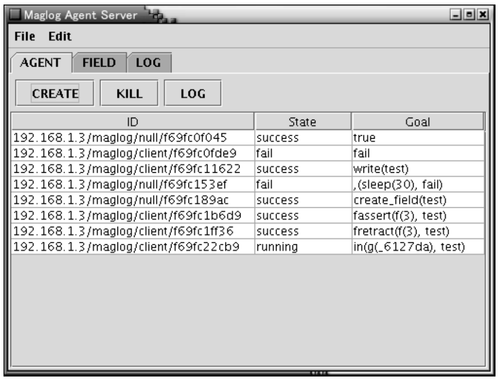
As mentioned in Section 3.2, an agent that attempts to execute or to retract a non-existent clause in a field simply fails in the fail mode, while an agent in the block mode is blocked by calling the Java `wait` method.

When another agent adds one clause to a field, the blocked agents in the field are woken up by the Java `notifyAll` method and try to execute their goals. The agents whose target clause has been added restart, while the remainder of the woken-up agents are blocked again.

4.3 Agent Migration

The migration of an agent is realized by using a Remote Procedure Call (RPC) as follows:

(1) The source agent server encodes the agent as the argument of an RPC.
(2) The source agent server obtains the serverID of the destination agent server from the second argument of the predicates `in/2`, `fasserta/2`, `fassertz/2`, and `fretract/2`.
(3) The source agent server sends an RPC



ID	State	Goal
192.168.1.3/maglog/null/f69fc0f045	success	true
192.168.1.3/maglog/client/f69fc0fde9	fail	fail
192.168.1.3/maglog/client/f69fc11622	success	write(test)
192.168.1.3/maglog/null/f69fc153ef	fail	,(sleep(30), fail)
192.168.1.3/maglog/null/f69fc189ac	success	create_field(test)
192.168.1.3/maglog/client/f69fc1b6d9	success	fassert(f(3), test)
192.168.1.3/maglog/client/f69fc1ff36	success	fretract(f(3), test)
192.168.1.3/maglog/client/f69fc22cb9	running	in(g(_6127da), test)

Fig. 6 GUI for an agent server.

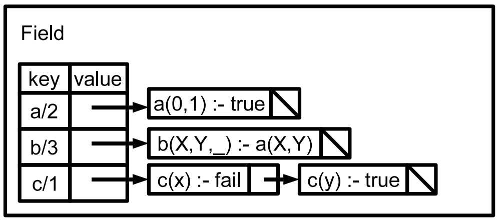


Fig. 7 Structure of a field.

request to the destination agent for invocation of the `receiveAgent` method.

(4) The destination agent server decodes the argument of the RPC and restarts the decoded agent.

Two mechanisms for RPC are implemented: RMI and XML-RPC. RMI is superior to XML-RPC from the viewpoint of the migration speed. On the other hand, XML-RPC is more firewall-friendly than RMI. HTTP connections used as the transport connections for XML-RPC are usually permitted through firewalls, while RMI connections are not usually permitted. In Maglog, both mechanisms are provided and users can choose whichever they prefer.

In order to reduce the traffic, a whole agent is not migrated initially. That is, Java classes compiled from Prolog predicates of an agent are transported on demand from the agent server on which the agent has been created.

5. Experiments

This section presents the experimental results for the execution time and amount of memory usage. In the experiments, two PCs with an Intel Xeon 3.4 GHz processor and 1 GB of RAM were connected via a 1000Base-T network. TurboLinux Server10 was used as the operating system. The version of the Java language runtime environment was 1.4.2. The performance of agents was examined from the following viewpoints:

(1) Creation of an empty agent,
(2) Migration of an empty agent,
(3) Reading 500 characters from a field,
(4) Writing 500 characters into a field.

Each experiment was repeated 100 times, and the average times are summarized in **Table 1**. **Table 2** shows the amount of memory usage of an agent server, of an empty agent, and of an agent with 60 clauses.

In **Fig. 8**, programs contained in agentA and in fieldA are shown. **Table 3** shows execution times in the case where fieldA is an ordinary field compared with the case where fieldA is a static field. The experiments were repeated 10,000 times and the total times were summarized. We can confirm that the agent execution time in a static field is much faster than in an ordinary field.

6. Applications

In this section, two applications are described to confirm the effectiveness of Maglog.

Table 1 Execution time of agent creation, agent migration, and reading/writing characters from/into a field.

Agent creation	Agent migration	Reading from a field	Writing into a field
7.30 msec	338.21 msec	0.05 msec	0.08 msec

Table 2 Amount of memory usage.

Agent server	Empty agent	Agent with 60 clauses
1,037 KB	413 KB	775 KB

```
main(N):-
  in(recursion(N),fieldA).
```

(a)

```
recursion(N):-
  N1 is N - 1,
  recursion(N1).
```

(b)

Fig. 8 Programs for comparison of the execution times in the case where fieldA is an ordinary field compared with the case where fieldA is a static field. (a) The program is contained in agentA; (b) the program is in fieldA.

Table 3 Comparison of the execution times in the case where fieldA is an ordinary field compared with the case where fieldA is a static field.

Ordinary field	9,341 msec
Static field	36 msec
Ratio (ordinary/static)	259

6.1 Distributed e-Learning System

A distributed e-learning system[10),11)] for asynchronous Web-based training was built using Maglog. This system allows students to study by themselves in their own time and following their own schedules, without any live interaction with a teacher.

Our distributed e-learning system consists of exercise agents and user interface programs. Each exercise agent includes not only exercise data but teacher’s functions for marking user’s answers, giving the correct answers, and showing some extra information. Every student’s computer receives some exercise agents from another computer when it joins the system and takes on the responsibility of sending appropriate exercise agents to requesting computers.

Figure 9 shows one part of the key codes in this application. This procedure is a part of an exercise agent. This is the procedure for

```
    loop:-
1:      fretract(request(Field,Host),fieldA),
2:      in(provide_exercise,Field@Host),
3:      loop.
```

Fig. 9 Procedure for providing an exercise for a remote user.

providing an exercise for a remote user. In executing this procedure, the following steps are performed.

(1) An agent retrieves a clause `request/2` which another agent added from `fieldA`. Here, `Host` and `Field` are the host name and field name of the student's computer.

(2) The agent migrates to `Host` and enters `Field`, and provides an exercise for the student. When the student finishes the exercise, the agent returns to the host it belongs to automatically.

(3) The agent recursively executes this procedure.

In this procedure, two types of field, fieldA and `Field` are used. `fieldA` in line 1 of Fig. 9 is used as a medium of asynchronous communication between agents, and `Field` in line 2 is used as an abstraction of migration.

6.2 Scheduling Arrangement System

The Mobile-Agent-Based Scheduling Arrangement System[12),13)] arranges a meeting schedule without human negotiations. It consists of negotiation agents and user interface programs. Once a convener convenes a meeting through the system, agents move around the meeting participants and negotiate with them semi-automatically. The distinguishing features of this system are as follows:

(1) Any user of this system can be a convener.

(2) The number of computers participating in this system can be changed flexibly.

(3) Neither the schedules of the participants nor the programs for negotiation are concentrated on a particular server. Instead, agents the collect schedules of the participants and negotiate with them.

Figure 10 shows one part of the key codes in this application. This procedure is a part of a negotiation agent which arranges a meeting schedule. An agent asks participants to open their schedules by executing this procedure during `Period`. In executing this procedure, the following steps are performed.

```
    negotiate(Period, [Field@Host|Tail]) :-
1:      fassert(request_open(Period),Field@Host),
2:      fretract(reply(X),Field@Host),!,
3:      X = ok,
4:      negotiate(Period, Tail).
```

Fig. 10 Procedure for asking participants to open their schedules during a particular period.

(1) A negotiation agent migrates to `Host` and enters `Field` to add the clause `request_open/1` to `Field`. An agent on `Host` retrieves this clause and asks the users to open their schedules.

(2) The former agent retrieves the negotiated result `reply/1`.

(3) The variable `X` is examined to recognize whether the negotiation succeeded or not. If the negotiation succeeded, step (4) will be executed; otherwise, this procedure fails.

(4) The agent recursively executes this procedure for the remaining participants.

As in the previous example in Section 6.1, two types of field are used in this procedure. `Field`, which appears in lines 1 and 2 of Fig. 10, is used as a medium of asynchronous communication and an abstraction of migration at the same time. In addition, unification across the network, which is one of the key features of Maglog, is presented. The variable `X` is bound to some value on `Host` and is unified with `ok` on another computer. In this example, if the unification fails, the whole procedure will fail, because of a cut operator in line 3 of Fig. 10. However, in the absence of the cut operator, the agent would return `Host` automatically and try to retrieve another clause matching `reply/1`. These features of unification and backtracking across the network simplify the control flows of an agent's program.

7. Conclusion

A new framework named Maglog for mobile agent systems was designed and developed in a Java environment. In Maglog, a concept called "field" is introduced, and is used to realize migration, inter-agent communication, and adaptation functions.

The effectiveness of the proposed framework was confirmed through descriptions of two applications: a distributed e-learning system and a scheduling arrangement system.

As regards the issue of error handling, Maglog currently handles only one type of error, which occurs when an agent intends to migrate to a host. Handling of errors after or during migration remains a task for the future. Security issues are indispensable problems for distributed applications using mobile agents. In Maglog, insufficient programmable security functions are provided, because security issues are vast. These functions will be added in the future. In addition, to make programs more practical, it is necessary to provide a program development environment, such as debugging and testing tools.

References

1) Lange, D.B. and Oshima, M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley (1998).
2) Satoh, I.: MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, *Proc. IEEE International Conference on Distributed Computing Systems*, IEEE Press, pp.161–168 (2000).
3) Kawamura, T., Hasegawa, T., Ohsuga, A. and Honiden, S.: Bee-gent: Bonding and Encapsulation Enhancement Agent Framework for Development of Distributed Systems, *Systems and Computers in Japan*, Vol.31, No.13, pp.42–56, John Wiley & Sons, Inc. (2000).
4) de Carvalho, C.L., Pereira, E.C. and da SilvaJulia, R.M.: NetProlog: A Logic Programming System for the Java Virtual Machine, *Proc. 1st International Conference on Enterprise Information Systems*, pp.591–598, Setubal, Portugal (1999).
5) Tarau, P.: Inference and Computation Mobility with Jinni, *The Logic Programming Paradigm: A 25 Year Perspective*, Apt, K., Marek, V. and Truszczynski, M. (Eds.), pp.33–48, Springer (1999).
6) Fukuta, N., Ito, T. and Shintani, T.: MiLog: A Mobile Agent Framework for Implementing Intelligent Information Agents with Logic Programming, *Proc. 1st Pacific Rim International Workshop on Intelligent Information Agents*, pp.113–123 (2000).
7) Kumeno, F., Ohsuga, A. and Honiden, S.: Flage: A Programming Language for Adaptive Software, *IEICE Trans.Inf.& Syst.*, Vol.E81-D, No.12, pp.1394–1403 (1998).
8) Carriero, N. and Gelernter, D.: Linda in Context, *Comm. ACM*, Vol.32, No.4, pp.444–458 (1989).
9) Banbara, M. and Tamura, N.: Translating a Linear Logic Programming Language into Java, *Proc. ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Carro, M., Dutra, I., et al. (Eds.), pp.19–39 (1999).
10) Kawamura, T. and Sugahara, K.: A Mobile Agent-Based P2P e-Learning System, *IPSJ Journal*, Vol.46, No.1, pp.222–225 (2005).
11) Motomura, S., Kawamura, T., Nakatani, R. and Sugahara, K.: P2P Web-Based Training System Using Mobile Agent Technologies, *Proc. 1st International Conference on Web Information Systems and Technologies*, pp.202–205, Miami, USA (2005).
12) Kinosita, S., Kawamura, T. and Sugahara, K.: Mobile Agent Based Schedule Arrangement System, *Proc. 5th IEEE Hiroshima Student Symposium (HISS)*, pp.205–206 (2003).
13) Motomura, S., Kagemoto, K., Kawamura, T. and Sugahara, K.: Meeting Arrangement System Based on Mobile Agent Technologies, *IPSJ Journal*, Vol.46, No.12, pp.3123–3126 (2005).

(Received July 11, 2005)
(Accepted January 6, 2006)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.2, pp.189–197.)

Shinichi Motomura was born in 1973. He received his B.Eng. and M.Eng. degrees in Computer Engineering from Toyohashi University of Technology, Japan, in 1995, 1997, respectively. He is currently a Ph.D. student in Tottori University. His research interests include multi-agent systems and distributed systems.

Takao Kawamura was born in 1965. He obtained his B.Eng., M.Eng. and Ph.D. degrees in Computer Engineering from Kobe University, Japan, in 1988, 1990 and 2002, respectively. Since 1994 he had been in Tottori University as a research associate and has been in the same University as an associate professor in the Faculty of Engineering since 2003. His current research interests include multi-agent systems and distributed systems. He is a member of IEICE, JSSST, and JSAI.

Kazunori Sugahara received the B.Eng. degree from Yamanashi University, Japan, in 1979 and M.Eng. degree from Tokyo Institute of Technology, Japan, in 1981. In 1989, he received the D.Eng. degree from Kobe University, Japan. From 1981 to 1994, he was on the staff of the Department of Electronic Engineering, Kobe City College of Technology. In 1994, he joined Tottori University as an associate professor of the Department of Electrical and Electronic Engineering and he is a professor of the department of Information and Knowledge Engineering. His current interest lies in the fields of computer architectures and hardware realizations of image processing algorithms. Dr. Sugahara is a member of IEEE and IEICE.