A Logic-based Framework for Mobile Multi-Agent Systems

Takao KAWAMURA, Shin KINOSHITA, and Kazunori SUGAHARA
Department of Information and Knowledge Engineering, Faculty of Engineering, Tottori University
4–101, Koyama-Minami, Tottori 680–8552, JAPAN
+81 857 31 5217
{kawamura,kinosita,sugahara}@ike.tottori-u.ac.jp

Tsuyoshi KUWATANI
Small Business Innovation Research, Industrial Research Institute, Tottori Prefecture
7–1–1,Wakabadai-Minami, Tottori, 689–1112, JAPAN
+81 857 38 6214
kuwatani@e-tottori.com

Abstract—*In this paper, we present a Prolog-based framework for building mobile multi-agent systems. In this framework, a new concept called "field" is introduced. A field provides data and procedures which are written in forms of Prolog program. An agent can "enter" into a field. Multiple agents can enter into one field, and the data and procedures of the field are shared by the agents. Therefore, agents in a field can communicate each other through the field. Furthermore, a field is located on a computer in a network, and an agent entering in a field migrate to a computer on which the field locate automatically. An implementation of this framework has been built on Java environment. In this paper, example programs running on our system are included to demonstrate the effectiveness of the proposed framework.*

1. Introduction

Multi-agent system is drawing attention as a structural model for many software systems including distributed systems and artificial intelligence systems[1]. In a multi-agent system, a number of autonomous agents cooperates mutually and achieves given tasks. Each agent is generated according to a given task and can have its own situation and operates under the situation. Situation consists of states and procedures where both of them can be dynamically changed in general. Therefore, it becomes necessary for the agent to dynamically hold the states and procedures (hereafter we refer them as a knowledge base). Moreover, when a number of agents are cooperating, it is necessary for them to share knowledge bases and to conduct knowledge communications between agents. In addition, mobility of agents is important in multi-agent system because of not only reducing network latency but simplifying architecture of software systems[2]. Although some frameworks for multi-agent systems have been proposed[3, 4, 5, 6, 7, 8, 9, 10], no framework is provided for constructing distributed knowledge sharing mobile agents.

In this research, an object called "field" is proposed which can contain a knowledge base. An agent can "enter" into a field. Multiple agents can enter into one field, and also an agent can enter into multiple fields. An agent, by entering a field, can refer to the knowledge base of the field as if it were its own. Other agents in the same field share the knowledge base of the field. Fields provide not only knowledge base but also a communication mechanism for agents, i.e., agents can communicate with each other synchronously and asynchronously through fields. Furthermore, a field is located on a computer in a network, and an agent entering in a field migrate to a computer on which the field locate automatically.

The proposed framework for mobile multi-agent systems, Maglog is based on Prolog, and has features of creating and controlling both agents and fields. We implemented Maglog on Java environment so that it is multi-platform and agents in Maglog can make or use any Java object.

This paper is organized in 5 sections. We describe the architecture of our framework in Section 2. In Section 3, we describe a implementation of the framework. In Section 4, we present example programs running on the system. Finally, in Section 5, we describe some concluding remarks.

2. Architecture of Maglog

The proposed framework is based on Prolog and each agent can conduct autonomous actions. A field provides an environment for agents. Agents which belong to the same field can be considered of forming a group. Information exchange between agents in the same field is conducted through the knowledge base within the field.

Agent

An agent has the following functions.

1. Execution of a program defining actions of the agent,
2. Execution of a program included in a field where the agent currently belongs,

3. Communication with other agents through a field,
4. Creation of agents and fields,
5. Entering into a field. That often implying the agent migrates to another computer in a network. The model of agent state migration in Maglog is strong migration which involves the transparent migration of agent's execution state as well as its program and data[11].

An agent is created when another agent executes the following predicate.

```
create(Agent, Agent_file, Goal)
```

An agent of the given `Agent_file` which contains Prolog program is created and its identifier is returned to the argument `Agent`. The created agent executes `Goal` and disappears when the execution of `Goal` is accomplished.

By executing the following predicate, the agent of the identifier `Agent` disappears anytime.

```
kill(Agent)
```

An agent can obtain his own identifier by executing the following predicate.

```
get_id(Agent)
```

As shown in Figure 1, an agent consists of Prolog program and Prolog engine. Prolog engine is a Prolog interpreter which executes the Prolog program. Prolog program is initially the program given from `Agent_file` . However since there is no distinction between program and data in Prolog, the agent program might be modified during execution. For example, as shown in Figure 2, agent A obtains a clause "`p(X):-q(X),r(X).`" from field A. Leaning of agents can be realized by this function.

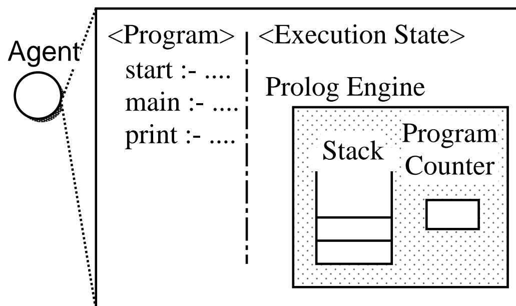

Figure 1 - An agent is composed of Prolog program and Prolog engine.

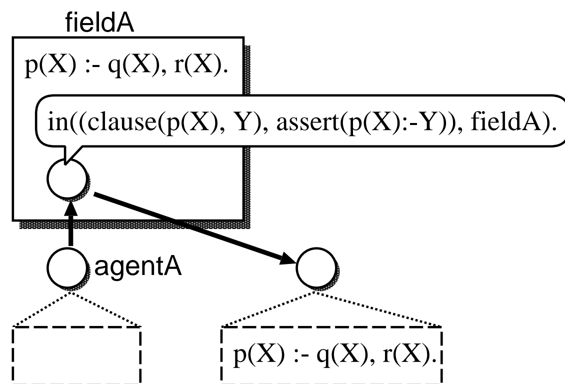

Figure 2 - Dynamic change of agent's knowledge base by asserting a new clause.

Field

A knowledge base can be stored within a field. The expression formats of both knowledge bases and agent programs are the same, i.e. Prolog clauses.

A field is created when an agent executes the following predicate.

```
fcreate(Field)
```

If `Field` is an unbound variable, a field which has an unique identifier is created and its identifier is bound to the argument `Field`. If `Field` is a symbol(atom), the function 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 which is created by this predicate, otherwise nothing is done.

An agent enters a field and execute a goal by the following predicate.

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

When an agent enters into a field, it can use combined knowledge of itself and the field. Therefore, an agent needs not to hold all the knowledge by itself to solve a problem, but rather entering to appropriate fields which provide necessary knowledge.

As shown in Figure 3, an agent in a field can access its own knowledge base and the field's knowledge base as if they were combined into a single knowledge base. The search order of the knowledge bases is set up so that the agent's own knowledge base is searched first, and then the field's knowledge base is searched. An agent can reside in multiple fields, and the search is executed in the reverse order of entering to the fields, in this case. If multiple knowledge bases have the same procedure, the last entered field's procedure shadows the other's procedures. They are not merged into one procedure. For example, `f(fieldA)` in Figure 3 shadows `f(fieldB)` and `f(agent)`.

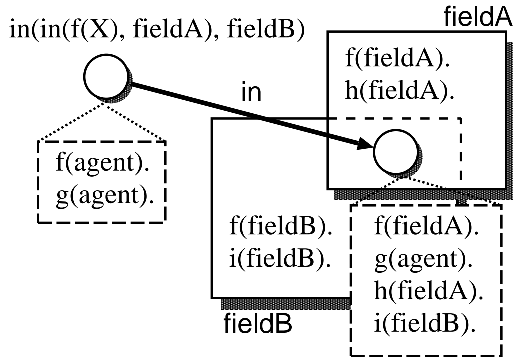

Figure 3 - An agent combines its own knowledge base with the knowledge base of the fields.

An agent can change its behavior dynamically through entering a field. Figure 4 shows an example. 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.

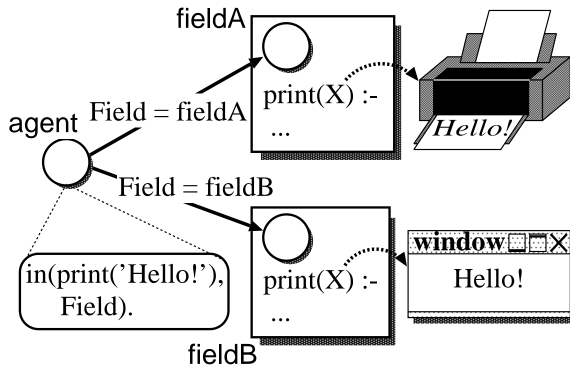

Figure 4 - Dynamic Change of agent's behavior through entering a field.

Agents belonging to the same field can be considered of forming a group. The knowledge within the field is shared by the agents. Moreover, by changing the knowledge within the field, agents can influence the actions of other agents.

Updating knowledge base in a field can be done by the following predicates.

```
fassert(Clause, Field)
fretract(Clause, Field)
```

The first argument `Clause` of both the predicates is a clause to be added or deleted from the field specified by the second argument `Field`.

Through these predicates, agents can communicate with other agents also synchronously. An agent has three mode for execution of clauses included in a field. Firstly, in the error mode, an agent is halted permanently when it intends to execute a non-existent clause in a field. Secondly, in the fail mode, it is failed as an ordinary Prolog interpreter under the same condition. Finally, in the block mode, it is blocked until the target clause is added to the field by another agent. For agents in the block mode, a field can be used as a synchronous communication mechanism such as a tuple space in Linda model[12].

Figure 5 shows an example that an agent generates a child agent and waits until the child returns the result.

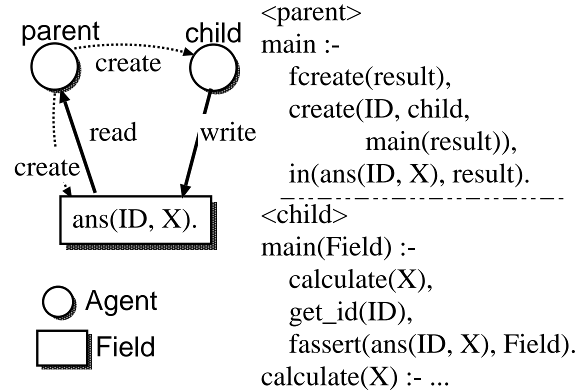

Figure 5 - Agents can communicate synchronously through a field.

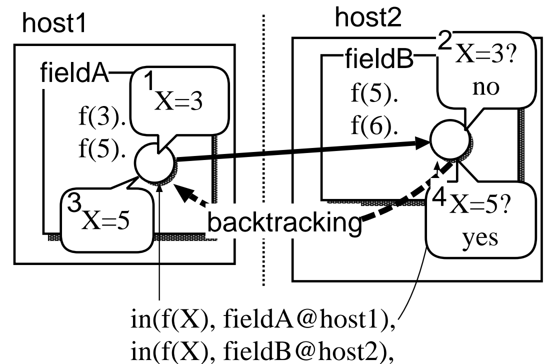

Figure 6 - Backtracking and unification between two computers.

3. Realization of Maglog

Maglog is realized on Java environment (JDK1.2 or later version) using Prolog Café[13] and run on any computer with a JDK 1.2-compatible runtime.

As shown in Figure 7, each computer in a network runs a server process for Maglog. Any agent runs on the server as a thread. An agent migrates from one server to another server through Java RMI technology.

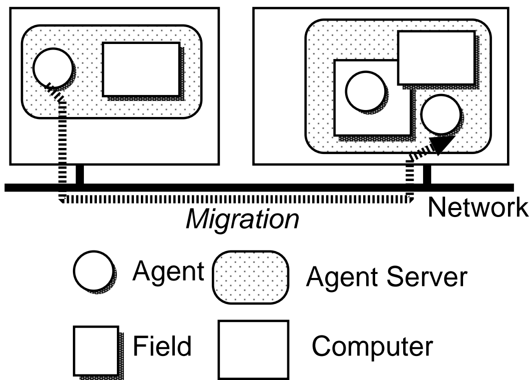

Figure 7 - Structure of Maglog

We have classified fields into static fields and dynamic fields. The former is compiled into Java classes before the server process which own the field starts. Thereby static fields can not be changed after the server process starts. On the other hand, dynamic fields can be changed in any time through the predicates described in section . An agent can execute a clause in a static field about 250 times faster than in a dynamic field.

Since Maglog is implemented on Java, any Java class library can be used from Maglog programs.

The implementation provides graphical user interfaces for operating agents and fields as shown in Figure 8.

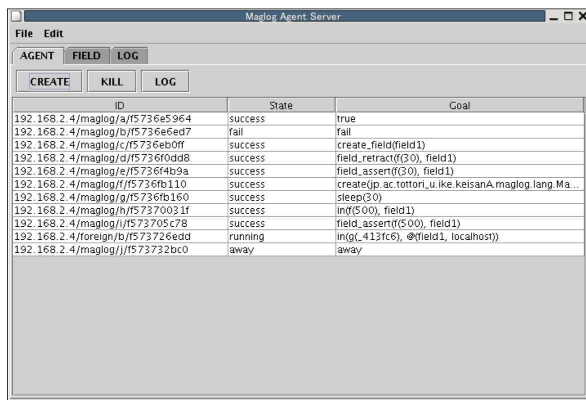

Figure 8 - The control window of the Maglog server

Basic Performance of Maglog

Table 1 shows data for a simple agent for evaluation of Maglog that makes 100 round trips between two computers in Maglog and Aglets[3]. Those computers are connected directly via 100Base-TX Ethernet.

The program size of the agent in Maglog is smaller than one in Aglets because Maglog supports strong agent mobility while Aglets only supports weak agent mobility. For same reason, the agent size in Maglog is larger than one in Aglets. However, the migration of the agent in Maglog is faster than one in Aglets.

Table 1 - Comparison between Maglog and Aglets

	Maglog	Aglets
program size(lines)	**5**	41
program size(bytes)	**93**	1047
agent size(bytes)	**7764**	767
migration time(msec)	**12089**	14281

Test program in Maglog and Aglets are shown in the appendix A and B, respectively.

4. Application Examples

Contract Net Protocol

The Contract Net Protocol (CNP) [14] assigns subtasks to agents which are involved in multi-agent problem solving. Appendix C presents a sample program of CNP in Maglog. The “contract” field not only provides utility procedures for CNP but also realizes communication between the manager agent and the worker agents.

Search for Books

Suppose a searcher agent wants to know the ISBN code of a book of which the title and the author are known. the searcher agent first searches the bookshelf in his own room, then if the desired item is not found, he goes around to a number of libraries until the desired book is found. A bookshelf or a library are expressed by a field, and book databases are stored in each respective field. Ordinarily, in one’s own room, every title on the bookshelf must be checked. In a library, a card file or a relational database is used to search for books. Therefore, the search method differs depending on the field. However, if search methods are defined under the same name (for example, `search`) the searcher agent can use them without being aware of these differences. A sample program is shown in the appendix D.

5. Conclusion

In this paper, we present a new framework for mobile multi-agent systems, Maglog. A mechanism called “field” which holds a knowledge base is introduced.

By using this concept,

1. Knowledge base can be shared by agents,
2. Situations of agents can be expressed,
3. Strong migration of agents can be realized, and
4. Mutual cooperation between agents distributed on a computer network can be realized.

We realized Maglog on Java environment and confirmed its effectiveness through experiments.

ACKNOWLEDGEMENTS

We would like to thank Prof. Yukio Kaneda, Prof. Naoyuki Tamura, Dr. Katumi Inoue, Dr. Hisasi Tamaki, and Dr. Mutsunori Banbara for many fruitful discussions.

REFERENCES

[1] Weiss, G.(ed.): *Multi-Agent Systems: A Modern Approach to Artificial Intelligence*, MIT Press (2000).

[2] Lange, D. B. and Oshima, M.: Seven good reasons for mobile agents, *Communications of the ACM*, Vol. 42, No. 3, pp. 88–89 (1999).

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

[4] Osuga, A., Nagai, Y., Irie, Y., Hattori, M. and Honiden, S.: Plangent: An Approach to Making Mobile Agents Intelligent, *IEEE Internaet Computing*, Vol. 1, No. 4, pp. 50–57 (1997).

[5] Wong, D., Paciorek, N., Walsh, T. and Dicelie, J.: Concordia: An Infrastructure for Collaborating Mobile Agents, *Proceedings of the First International Workshop on Mobile Agents*, Vol. 1219, Springer-Verlag, pp. 86–97 (1997).

[6] Kumeno, F., Ohsuga, A. and Honiden, S.: Flage: A Programming Language for Adaptive Software, *IEICE Transactions of Information & System*, Vol. E81–D, No. 12, pp. 1394–1403 (1998).

[7] Tarau, P.: Inference and Computation Mobility with Jinni, *The Logic Programming Paradigm: a 25 Year Perspective* (Apt, K., Marek, V. and Truszczynski, M.(eds.)), Springer, pp. 33–48 (1999).

[8] Fukuta, N., Ito, T. and Shintani, T.: MiLog: A Mobile Agent Framework for Implementing Intelligent Information Agents with Logic Programming, *Proceedings of the 1st Pacific Rim International Workshop on Intelligent Information Agents*, pp. 113–123 (2000).

[9] Satoh, I.: MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System, *Proceedings of IEEE International Conference on Distributed Computing Systems*, IEEE Press, pp. 161–168 (2000).

[10] Suri, N., Bradshaw, J. M., Breddy, M. R., Groth, P. T., Hill, G. A., Jeffers, R., Mitrovich, T. S., Pouliot, B. R. and Smith, D. S.: NOMADS: toward a strong and safe mobile agent system, *Proceedings of the fourth international conference on Autonomous agents*, pp. 163–164 (2000).

[11] Ghezzi, C. and Vigna, G.: Mobile Code Paradigms and Technologies: A Case Study, *Proceedings of the First International Workshop on Mobile Agents*, pp. 39–49 (1997).

[12] Carriero, N. and Gelernter, D.: Linda in Context, *Communications of the ACM*, Vol. 32, No. 4, pp. 444–458 (1989).

[13] Banbara, M. and Tamura, N.: Translating a linear logic programming language onto Java, *Proceedings of ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pp. 19–39 (1999).

[14] Smith, R.: The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, *IEEE Transactions on Computers*, Vol. C-29, No. 12, pp. 1104–1113 (1980).

A TEST PROGRAM IN MAGLOG

```
main(_, 0):-!.
main(Dest, N):-
    in(true, test@Dest),
    N1 is N - 1,
    main(Dest, N1).
```

B TEST PROGRAM IN AGLETS

```
public class HelloAglet extends Aglet {
    SimpleItinerary si = null;
    String home = null;
    String dest = null;
    int count = 100;
    public boolean
    handleMessage(Message msg) {
        if (msg.sameKind("atHome"))
            atHome(msg);
        else if (msg.sameKind("atDest"))
            atDest(msg);
        else
            return false;
        return true;
    }
    public void onCreation(Object init) {
        si = new SimpleItinerary(this);
        home = getAgletContext().
            getHostingURL().toString();
    }
    public void atHome(Message msg) {
```

```
        if(count > 0)
            try {
                si.go(dest, "atDest");
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        dispose();
    }
    public void atDest(Message msg) {
        try {
            count--;
            si.go(home, "atHome");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    public void setDest(String dest) {
        this.dest = dest;
    }
}
```

C Program of the Contract Net Protocol

```
%%%%%%%%%% contract.pl %%%%%%%%%%
:-public delegate/3, get_goal/2,
        propose/2, inform/2.
delegate(Goal, Result, Time) :-
    fcreate(cfp),
    get_id(ID),
    fassert(cfp(ID, Goal), cfp),
    sleep(Time),
    noblock(findall(
        proposal(Wkr, Cost),
        fretract(proposal(Wkr,ID,Cost),
                 cfp),
        Proposals)),
    fretract(cfp(ID, Goal), cfp),
    sort(Proposals, [Best|Reject]),
    fassert(reply(ID, Best, accept),cfp),
    reject(Reject, cfp),
    fretract(inform(ID, Result), cfp),
    Result == yes.
get_goal(Mngr, Goal) :-
    fcreate(cfp),
    in(cfp(Mngr, Goal), cfp).
propose(Mngr, Cost) :- get_id(ID),
    fassert(proposal(ID, Mngr, Cost),
            cfp),
    fretract(reply(Mngr,ID,R), cfp),
    R == accept.
inform(Mngr, Res):-
    fassert(inform(Mngr, Rel), cfp).
reject([], _) :-!.
reject([proposal(Wkr,Cost)|T], Field) :-
    get_id(ID),
    fassert(reply(ID,Wkr,reject),Field),
    reject(T, Field).
%%%%%%%%%% worker.pl %%%%%%%%%%
main :-
    in(get_goal(Mngr, Goal), contract),
    calculate_cost(Goal, Cost),
    in(propose(Mngr, Cost), contract),
    process(Goal, Res),
    in(inform(Mngr, Res), contract).
process(Goal, yes) :- Goal, !.
process(Goal, no).
%%%%%%%%%% manager.pl %%%%%%%%%%
main(Goal) :-
    in(delegate(Goal, Result, 3),
       contract).
```

D Program of Search for Books

```
%%%%%%%%%%%% agent.pl %%%%%%%%%%%%
book_search(Author, Title, ISBN) :-
    bs(Author, Title, ISBN,
       [fieldA@hostA,
        fieldB@hostB,
        fieldC@hostC]).
bs(Author, Title, ISBN, [Field|Tail]) :-
    in(search(Author, Title, ISBN),
       Field), !.
bs(Author, Title, ISBN, [_|Tail]) :-
    bs(Author, Title, ISBN, Tail).
%%%%%%%%%%%% fieldA.pl %%%%%%%%%%%%
:- public search/3.
search(Author, Title, ISBN) :-
    book(Author, Title, ISBN).
book(author0,title0,'ISBN0-0000-0000-0').
book(author0,title1,'ISBN1-1111-1111-1').
%%%%%%%%%%%% fieldB.pl %%%%%%%%%%%%
:- public search/3.
search(Author, Title, ISBN) :-
    data(Author, List),
    s(Title, ISBN, List).
s(Title, ISBN, [book(Title, ISBN)|_]).
s(Title, ISBN, [_|Tail]) :-
    s(Title, ISBN, Tail).
data(author0,
     [book(title2,'ISBN2-2222-2222-2'),
      book(title3,'ISBN3-3333-3333-3')]).
data(author1,
     [book(title4,'ISBN4-4444-4444-4')]).
%%%%%%%%%%%% fieldC.pl %%%%%%%%%%%%
:- public search/3.
search(Author, Title, ISBN) :-
    java_constructor('example.SQL', SQL),
    java_method(SQL,
              query(Author,Title),ISBN).
```