Proceedings of _____:
Conference Name
Date and Location

Reference ID

HIGHER-ORDER MOBILE AGENTS FOR CONTROLLING INTELLIGENT ROBOTS

Yasushi Kambayashi/Nippon Institute of Technology **Munehiro Takimoto/ Tokyo University of Science**

ABSTRACT

This paper presents a framework for controlling intelligent robots. This framework provides novel methods to control coordinated systems using higher-order mobile agents. Higher-order mobile agents are hierarchically structured agents that can contain other mobile agents. By using higher-order mobile agents, intelligent robots in action can acquire new functionalities dynamically as well as exchange their roles with other colleague robots.

1. INTRODUCTION

The traditional means to constructing intelligent robots is making large monolithic artificial intelligent software. Robotics has been considered as a part of artificial intelligence. ALVINN autonomous driving system is one of the most successful such developments [1].

Putting intelligence into robots is, however, not easy task. Intelligent robot that is able to work in the real world needs large scale knowledge base. The ALVINN system employs the neural networks to acquire the knowledge semi-automatically [2].

One of the limitations of neural networks is, however, to require the assumption that the system is used in the same environment as it is trained. When the intelligent robot is expected to work in unknown space or the extremely dynamic environment, it is not realistic to assume the neural network is fully trained. Indeed, some intelligent robots need a mechanism to adopt unknown environment.

On the other hand, multi-agent system in robotics is getting popular in RoboCup or MIROSOT recently [3]. In the traditional multi-agent systems, robots communicate each other to achieve cooperative behaviors. The Nerd Herd and ALLIANCE are successful examples of cooperative intelligent systems [4] [5]. They, however, cannot be extended dynamically after they start working. Further more, one agent program controls one robot, and there is no notion that a mobile agent migrates dynamically into a robot to extend the functionality of the robot. It is hard for the traditional multi-agent systems to adapt unknown environments.

In this paper, we propose a framework for constructing intelligent robots controlled by higher-order mobile agents. The higher-order property of the mobile agents enables them to be organized hierarchically and dynamically. Each mobile agent can be a container of other mobile agents and can migrate to other agents. Therefore the robots controlled by the mobile agents can acquire new functions by migration of other agents. The extended agent behaves as a single agent so that it can migrate to another agent with the containing agents.

In addition to the advantages described above, higher-order mobile agents require minimum communication. They only need connection being established when they perform migration [6]. This is useful for controlling robots working in a remote site.

The structure of this paper is as follows. The second section describes the higher-order mobile agents. The third section describes the dynamic extension feature of the mobile agent system. The dynamic extension is the key feature to add new functionalities to intelligent robots in action. The fourth section shows an example intelligent robot system in which robots play the game of TAG. Finally, the fifth section discusses future works and conclusive remarks.

NOMENCLATURE

Mobile agent, dynamic software extension, dynamic software composition, and intelligent robot control.

2. HIGHER-ORDER MOBILE AGENT

The mobile agent system we use to control robots is based on a mobile agent system, called MobileSpaces, developed by I. Sato [7] [8]. MobileSpaces is also based on the mobile ambients computational model proposed by L. Cardelli and A. D. Gordon [9]. MobileSpaces provide the basic framework for mobile agents. It is built on the Java virtual machine, and agents are supposed to be programmed in Java language.

Mobile agents can migrate place to place. When they migrate, not only the program code of the agent but also the state of the agent can be transferred to the destination. The higher-order mobile agents are mobile agents whose destination can be other mobile agents as well as places in traditional agent systems.

Two unique features are worth mentioning for our robot control system. 1) Each mobile agent can contain one or more mobile agents (hierarchical construction), and 2) Each mobile agent can migrate to any other mobile agent (inter-agent migration).

Thus migration to another agent results in a nesting structure of agents. Agents in the other agent are still autonomous agents that can behave own scenario. When an agent migrates to another agent, the coming agent is called the *child* agent, and the container agent is called *parent* agent. In the same sense, nested agents are called *descendent* agents and nesting agents are called *ancestral* agents. Parent agents give their resources and services to their child agents so that an agent or a group of agent can acquire whatever its parent provides. Figure 1 illustrates the situation that agent C migrates from agent A to agent B, and the child agent D also migrate from agent A to agent B. Serialization and other necessary processing to migrate are done by a special stationary agent MATP.

The first feature allows a mobile application to be constructed by organizing more than one agent. The second feature allows a group of agents to be treated as a single agent. By using these two features, we can construct a mobile application as the combination of mobile agents. We can send the base agent to remote site, and then we can add new features and functions to the base agent by sending other agents later, while the base agent is running.

We use a special addressing term like URL to specify agents in a hierarchical structure. In this paper, we call it URL too. Figure 2 depicts a situation that an agent alpha that contains the other agent beta is on a machine whose IP address is 012.345.678.901 and uses port number 5000. In this case, URL for agent beta is "://012.345.678.901:5000/alpha/beta."

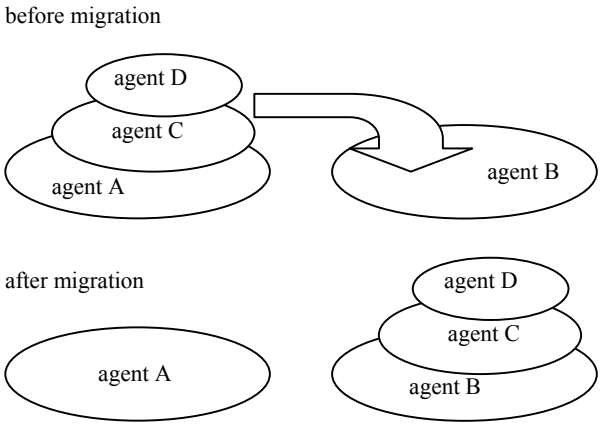


Figure 1. When agent C migrates from agent A to agent B, the contained agent D also migrates from A to B.

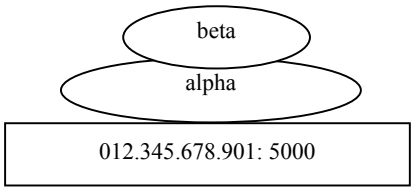


Figure 2. URL for agent *beta* is :// 012.345.678.901: 5000/alpha/beta

3. DYNAMIC EXTENSION

MobileSpace provides the basic mechanism for agent migration and remote method invocation. When an agent wants to invoke a method in another agent, the calling agent just needs to specify the called agent with URL and send a message object to the agent. It lacks, however, the feature that an agent can dynamically extend its functionality. For intelligent robot control, we add the dynamic extension feature to customize functions of robots while they are running.

Suppose an agent A is working somewhere and we want to extend its feature. One way is to replace that agent with a new agent B. On the other hand in our system, we only need to send an agent A' with the new feature to the agent A. While the agent A' is being the child of A, the agent A behaves with the extended feature. If the agent A' leaves the agent A, the agent A behaves with the original feature. All the other agents do not have to be aware of the change of the agent A. In Figure 3, after an agent A' migrates to an agent A, the other agent B still communicates to the agent A without knowing the migration of A'. The agents A and A' behave just as a single agent for the agent B.

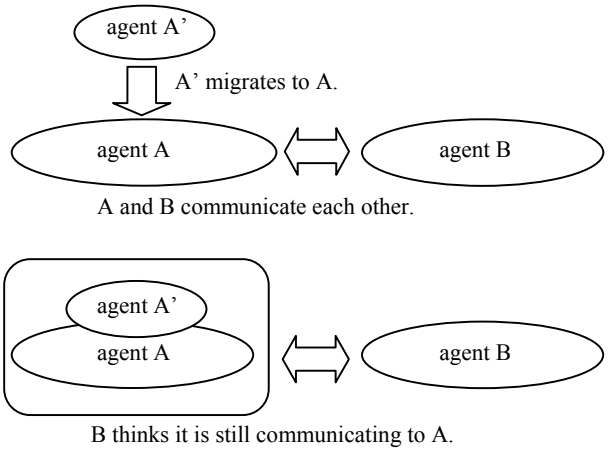


Figure 3. Dynamic extension by migration of agent with new features.

In order to extend the agent A, the agent A' only needs to have the difference (the new feature to be added). If the agents A and A' have methods with the same signature, the method in agent A' override the method with the same signature in the agent A.

For example, if *output* method in the agent A is needed to extend, the agent A' with new *output* method is to be sent into the agent A. The child agent A' intercepts all the incoming messages to A and passes through all the messages except for output request as shown in Figure 4. For output request, the agent A' uses its own output method instead of the output method in the parent agent A. Thus the agent migration achieves the same semantics with the dynamic inheritance [10].

The agent A' is also designed to communicate to B. Since A and A' are agents with their own threads, the agent A can still send messages to B after the arrival of A'. Therefore it is possible that the message sending of A interferes the message sending of A' as shown in Figure 5.

In order to ameliorate this problem, the extending agent A' have a mechanism to suppress its parent's message sending. It is hard for A to anticipate which of its messages may be suppressed in the future. Therefore the parent agent (that is extended) should not have any responsibility about it. On the other hand, it is easy for A' to have a list of messages to be suppressed (forbidden list). Therefore the child agent (that extends the parent) is given the forbidden list when it is created. The list consists of pairs of URL and method names that the parent agent is supposed to refrain to send.

When an agent tries to send a message to another agent, it checks the forbidden list in the child agent recursively. If the agent finds that the message it is trying to send is forbidden, it refrains to send it.

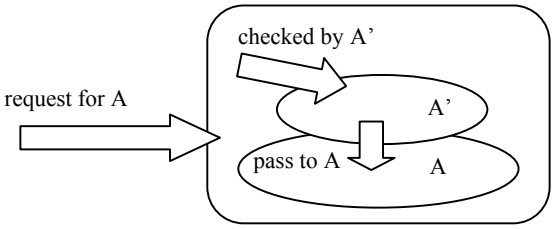
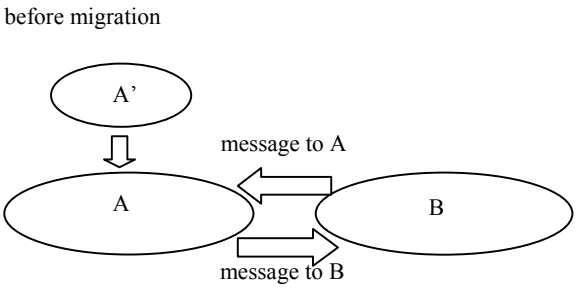


Figure 4. Every message to A is intercepted by A'. If the method corresponding to the request is not implemented in A', it is passed through A.



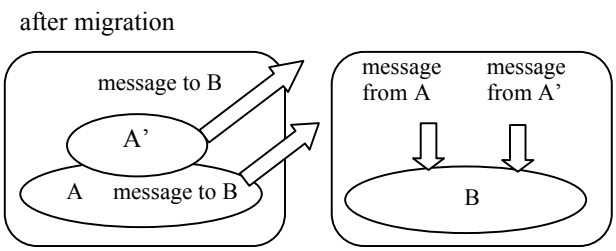


Figure 5. Agent A' migrates into A, and tries to represent for communication to B. Agent A can still send messages to B, and these may interfere the messages from A'.

The following is an example for dynamic extension. The sender agent (Figure 6) sends a message to the reporter agent (Figure 7) once a second. The *getService* method in *sender* agent sends the message which tells how much seconds is passed after the sender is created. The reporter agent just displays the given number in the message. When these agents start working, the method *output* in the reporter agent is called once a second and display the current content of *count*.

```
Public class sender extends Agent implements Runnable,
Serializable, StatusListner, StructureListner {
  Transient private Thread th;
        …
    setName("sender");
        …
  public void run() {
    int count=0;
    try {
      while (true) {
        count++;
        Thread.sleep(1000);
        Message msg = new Message("output");
        Msg.setArg(count);
        getService(new AgentURL("://reporter/"), msg);
      }
    } catch (Exception e) {System.out.println(e)}
```

Figure 6. The *sender* agent.

```
Public class reporter extends Agent implements
Runnable, Serializable, StatusListner,
StructureListner {
  Transient private Thread th;
        …
    setName("reporter");
        …
  public void output(int count) {
    System.out.println(count);
  }
```

Figure 7. The *reporter* agent.

Then we create an extending agent *reporter2* (Figure 8) and make it migrate into the reporter agent and override the *output* method. The difference between *reporter* and *reporter2* is the latter's *output* method displays annotated messages. The sender agent sends the same massage to the reporter agent, but now the arrived message is intercepted by the extending agent *reporter2* and *reporter2*'s *output* method is invoked so that annotated messages are displayed as shown in Figure 9.

Now we create another extending agent *sender2* that extends the sender agent. *Sender2* is the same as *sender* but sends a message once per ten seconds. When we make this

```
Public class reporter2 extends Agent implements
Runnable, Serializable, StatusListner,
StructureListner {
  Transient private Thread th;
        …
    setName("reporter2");
        …
  public void output(int count) {
    System.out.println(count+" seconds passed after
                       the sender created.);
  }
```

Figure 8. The *reporter2* agent.

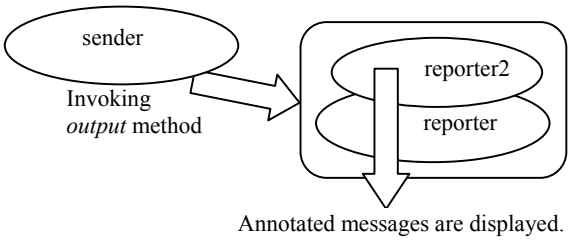


Figure 9. Upon arrival of reporter2, the new output method is used.

extending agent migrate into the sender agent, it starts to send messages once per ten seconds. However, the original sender *sender* is still working and issuing its message once a second.

This is not a favorable situation. I order to prevent this interference, *sender2* has to have the forbidden list that tells which methods in the parent agent are supposed to be suppressed. The *addRefusal* method invocation in *sender2* agent does this task (see Figure 10). The user of this agent is expected to set the URL and method name by using this method. Note that the URL is "`://reporter/`", not "`://reporter2`." *Sender2* does not know the reporter agent has been extended.

For example, once *sender2* have arrived in *sender* agent with the proper forbidden list, *sender*'s message (output invocation) is suppressed by *sender2* and the reporter agent received the output message once per ten seconds as shown in Figure 11.

After that, if the child agent of the reporter agent, *reporter2*, leaves from *reporter* and *reporter* has no child, the output request from *sender2* causes *reporter*'s *output* method invoked and non-annotated messages start to be displayed again. Then, the state of the target agent for *sender* and *sender2* (the reporter), is changed, but they need not know about it. What *sender* and *sender2* know is that they are sending messages to the reporter agent.

Note that the behavior of the sender agent before the reporter's extension is the same as that of after the reporter's extension, and that of after the extension is cancelled (when the child agent leaves).

Similarly, when the child agent of the sender agent *sender2* leaves from *sender*, *sender* agent is allowed to send messages, and reporter agent displays the output once a second.

```
addRefusal(new AgentURL("://reporter/"), "output",
           new Class[]{int.class});
```

Figure 10. *addRefusal* method invocation

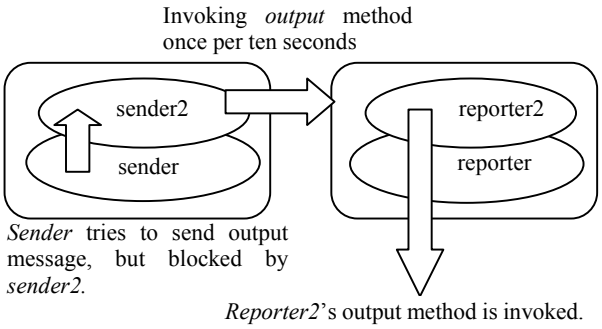


Figure 11. Upon arrival of *sender*2, *sende*r's *output* method is suppressed, and sender2's is used.

Thus the extension and restriction of agents can be achieved dynamically by the other agents' migration. It is not necessary to statically compose the agent program. It is even not necessary to stop the running agent. It should be a desirable feature for intelligent robot control program.

4. ROBOT CONTROL EXAMPLE

In this section, we demonstrate that the higher-order mobile agent with dynamic extension is suitable to compose software to control intelligent robot.

Intelligent robots are expected to work in distributed environment and communication is relatively unstable so that fully remote control is hard to achieve. Also we cannot expect that we know everything in the environment beforehand. Therefore intelligent robot control software needs to have the following features: 1) It should be autonomous in some extent. 2) It should be extendable to accommodate the working environment. 3) It should be replaceable as it is in action. Our higher-order mobile agent with dynamic extension satisfies all these desirable features.

Our control software consists of mobile agents which are autonomous in some extent. Once each agent migrates to a remote site, it requires minimum communication to the original site. Mobile agents are higher-order so that one can construct a larger agent by hierarchical composition of smaller agents. Finally, when we find that the constructed software has anomaly, we can replace the unsuitable component (an agent) with new component (another agent) by using agent migrations.

4.1 THE ROBOT

We employed Palm Pilot Robot Kit (PPRK) by ACRONAME Inc. as the platform for our prototype system [11]. Each robot has three servo motors with tires. The power is supplied by four AA batteries. It has a servo motor controller board that accepts RS-232 serial data from a host computer. We use Toshiba Libretto notebook computers for the host computers. Each robot holds one notebook computer as the host computer. Our control agents are supposed to migrate to the host computer by wireless LAN (see Figure 12).

4.2 THE CONTROLLER AGENTS

In the beginning, two agents are supposed to migrate to the host computer to give the basic behavior of the robot. One is o*perate* agent, and the other is w*all* agent.

Operate agent can read and write serial data, and behaves as the interface between PPRK on-board controller and the intelligent software agent. *Wall* agent receives sensor data from o*perate* agent, determines the basic behaviors of the robot based on that data, and sends messages such as go-forward, turn-left/right and stop to *operate* agent. And then o*perate* agent translates instructions corresponding to these messages into serial data and sends to the on-board controller.

In order to achieve these functions, *operate* agent has methods to obtain and to release a serial port, to read the sensor data as well as to instruct the robot movement. To read sensor data, an event listener is required. The registration of the event listener is done as a part of obtaining the serial port.

The sole task of *wall* agent is to avoid collisions. When it receives sensor data indicating something exists in front of it (wall or another robot), it issues instructions to turn around. This simple collision avoidance algorithm is implemented in *think* method in *wall* agent.

In order to give the robots more intelligent behavior,

Figure 12. Robot control agents are working on ACRONAME PPRK.

one is expected to extend this *wall* agent by the dynamic extension. In this example, two agents, *chase* and *escape*, extend *wall* agent. Robots with *escape* agent try to avoid the chaser as well the wall, while the robot with *chase* agent looks for other robots and tries to catch one of them. As a result, they play the game of TAG. Figure 13 shows the structure of the agents that control the robot.

The *escape* and *chase* agents have their own *think* methods so that they can extend *wall* agent to make the behaviors of the robots more intelligent. The *think* method of *escape* agents can distinguish the other robots from the wall, and instruct different behaviors. It can move as close as 30 cm if it finds the wall in front of it, but it should not move as close as 60 cm if it finds a robot in front of it, because the other robot may be the chaser.

The *chase* agent has two methods. One is its own *think* method and the other is *arrive* method. The *think* method of *chase* agents can also distinguish the other robots from the wall, and instruct different behaviors. If the robot with *chase* agent finds another robot, it chases the robot and tries to catch it. Since our robots do not have arms, if the chaser gets as close as 10 cm to another robot, the chaser judges the other robot to be caught. Figure 14 shows the *think* method of the *chase* agent.

When the robot with *chase* agent catches another robot, it migrates to the *wall* agent on the caught robot, so that the roles of chaser and escapee can be exchanged each other as shown in Figure 15. This is done by the *move* method

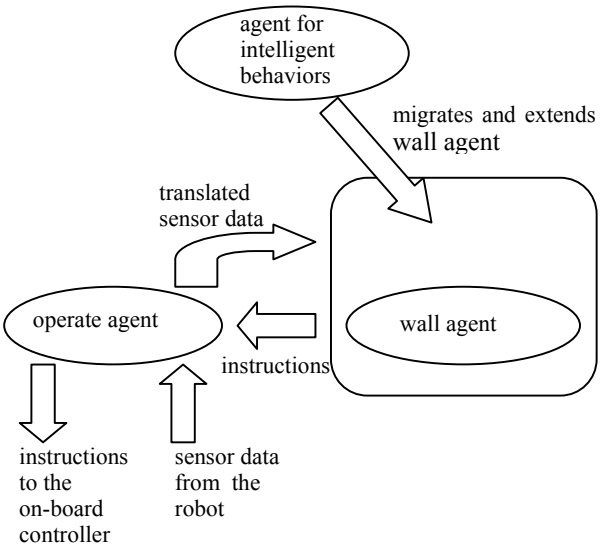


Figure 13. Structure of the controller agents. The base agent MATP is not shown.

```
public void think(int dist1, int dist2) {
  if (dist2<=10) {
    Context cx;
    cx = getContext();
    try {
      cx.move(new AgentURL(":://wall/chase/",
               new AgentURL(nextAddress));
    } catch(Exception e) {System.out.println(e);}
  }
  Message msg;
  if (dist2<100) {
    msg = new Message("forward");
  } else {
    msg = new Message("turn");
  }
  try {
    getService(new AgentURL(":://operate/", msg);
  } catch(Exception e) {System.out.println(e);}
}
```

Figure 14. *think* method in the *chase* agent

invocation in the *think* method of the *chase* agent as shown in Figure 14. Upon arrival in the *wall* agent on the caught robot, the *chase* agent make the *escape* agent migrate back to the *wall* agent on the robot which the *chase* agent existed previously. This is achieved by the *arrive* method in the *chase* agent. Figure 16 shows the *arrive* method.

The *chase* agent initiates and controls the exchange of the roles. Therefore other agents e.g. *escape* and *wall* agent, do not have to know anything about the role exchange. In addition, messages from the *operate* agent are intercepted by the child agents, i.e. the *escape* or *chase* agent, the *operate* agent do not need to know whether it communicates with the *wall* agent, the *escape* agent, or the *chase* agent. While *chase* and *escape* agents are migrating to swap themselves, the robot is controlled by the *wall* agent, which avoids collision. This simplicity is achieved by the framework of higher-order mobile agents.

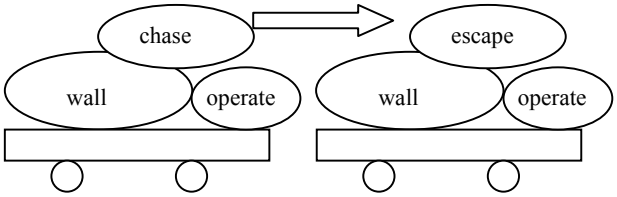
5. CONCLUSION AND FUTURE DIRECTION

We have presented a new framework for controlling intelligent robots. The framework helps users to construct intelligent robot control software by migration of mobile agents. Since the migrating agents are higher-order, the control software can be hierarchically assembled while they are running. Dynamically extending control software by the migration of mobile agents enables us to make base control software relatively simple, and to add functionalities one by one as we know the working environment.

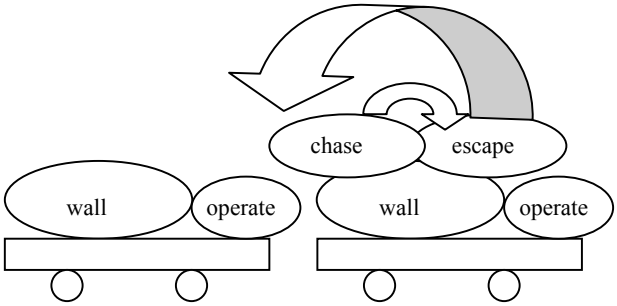
Thus we do not have to make the intelligent robot smart from the beginning or to make the robot learn by itself. We can send intelligence later as new agents.

We implemented a TAG playing robots to show the effectiveness of our framework.

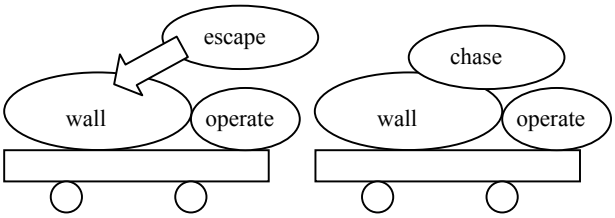
Even though our example is a toy program, our framework is scalable, and making a practical system is just



(a) When the chaser catches the escapee, the *chase* agent migrates to the *wall* agent on the escapee.



(b) Migrated *chase* agent makes the *escape* agent migrate back to where the *chase* agent came.



(c) The escape agent is moved by the chase agent, and the exchange of the roles is over.

Figure 15. The chaser catches the escapee and they swap their roles.

```
public void arrive(StructureEvent e) {
  Context cx;
  cx = getContext();
  try {
    cx.move(new AgentURL(":://wall/escape/",
              new AgentURL(nextAddress));
  } catch(Exception x) {System.out.println(x);}
}
```

Figure 16. *arrive* method in the *chase* agent

made by adding more mobile agents to the base system.

On the other hand, we are aware of the shortcomings of this prototype. The system lacks of any security features that are required for practical systems. Also the implementation of the dynamic extension is not as elegant as we wish. The current system use rewriting references and other rather brute force programming techniques. We plan to re-implement the system based on events and event listeners.

ACKNOWLEDGMENTS

Masato Kurio and Satoshi Kanesaka contributed in discussions and implementation of the system.

REFERENCES

[1] D. A. Pomerleau, "Defense and Civilian Applications of the ALVINN Robot Driving System", *Proceedings of Government Microcircuit Applications Conference*, 1994.

[2] D. A. Pomerleau, "ALVINN: An Autonomous Land Vehicle in a Neural Network", *Advances in Neural Information Processing System 1*, Morgan Kaufmann, 1989.

[3] R. R. Murphy, *Introduction to AI robotics*, MIT Press, 2000.

[4] M. Mataric, "Minimizing Complexity in Controlling a Mobile Robot Population", *Proceedings IEEE International Conference on Robotics and Automation*, 1992.

[5] L.E. Parker, "ALLIANCE: An Architecture for Fault Tolerant Multirobot Cooperation", *IEEE Transaction on Robotics and Automation, vol. 14, No. 2*. pp.20-240, 1998.

[6] W. Binder, J. G. Hulaas and A. Villazon, "Portable Resource Control in the J-SEAL2 Mobile Agent System", *Proceedings of International Conference on Autonomous Agents (AGENTS'01)*, pp.222-223, May 2001.

[7] I. Satoh, "Hierarchically Structured Mobile Agents and their Migration", *Workshop on Mobile Object Systems (MOS'99)*, July 1999.

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

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

[10] M. Abadi and L. Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.

[11] Acroname Inc. home page, http://www.acroname.com/