A Framework to Mitigate Debugging Difficulty on Agent Migration

Shin Osaki[1], Masayuki Higashino[2], Kenichi Takahashi[1], Takao Kawamura[1] and Kazunori Sugahara[1]
[1]*Department of Information and Electronics , Graduate School of Engineering, Tottori University, Tottori, Japan*
[2]*Organization for Regional Industrial Academic Cooperation, Tottori University, Tottori, Japan*
{*s092018, s032047, takahashi, kawamura, sugahara*}*@ike.tottori-u.ac.jp*

Keywords: Mobile Agent System, Debug, Distributed System, Migration, Interaction.

Abstract: A mobile agent is an autonomous software module that can work on different computers and migrate among these computers. The characteristics of a mobile agent, migration and interaction, are helpful to implement distributed systems. In the real world, however, a mobile agent is not widely used because its migration makes debugging distributed systems difficult. Therefore, in this paper, we discuss problems in debugging a mobile agent system and propose a framework that includes a search, a single-step execution, and a reproduction function to help programmers debug a mobile agent system. Results from our experiments on debugging test applications show that our framework is helpful to support programmers and help them to debug. This reduces the number of keystrokes by 41% and number of clicks by 24%.

1 INTRODUCTION

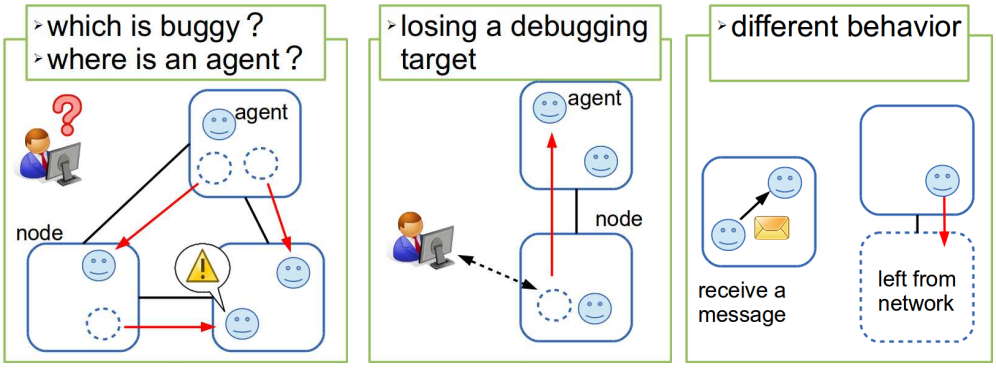
A mobile agent system, which is constructed from many mobile agents, can migrate among computers to achieve its tasks. Hence, we can use mobile agent migration instead of clientserver communications. Because the mobile agent can continue its work over the computers, we do not need to elaborate a client- and server-side program pair. We can implement a server- and client-side program as one mobile agent, thus enabling us to implement a network-based system without being aware of communication APIs and protocols. Such a mobile agent system is attractive for designing, implementing, and maintaining a distributed system (Hurson et al., 2010; Outtagarts, 2009); however, a mobile agent is not widely used in the real world because the migrations of mobile agents make it difficult to debug mobile agent systems.

To debug a mobile agent system, it is important to grasp their behavior. However, this is difficult because there are many mobile agents that migrate among computers autonomously. In general, a debugger provides remote debugging functions, such as a breakpoint function and a single-step execution function, to check the details of running programs; however, it does not expect the migration of programs. Thus, we cannot debug mobile agents continuously because they migrate to other computers even while programmers are debugging. Another problem is that the behavior of a mobile agent is affected by other mobile agents. Furthermore, network conditions can affect the migration of a mobile agent. Therefore, we need to consider not just one but multiple mobile agent states and network situations in debugging. There are some standards organization for agents such as FIPA(FIPA, 2014) and MASIF(Milojicic et al., 1998). However, they do not care about the previously mentioned Problems. In this paper, we discuss their problems and propose a debugger that has a search function, a single-step execution function, and a reproduction function for mobile agents.

This paper is organized as follows. Section 2 discusses problems in debugging mobile agents. Section 3 discusses a framework to solve these problems. Section 4 discusses the implementation and evaluation of our framework. Section 5 shows related works, and we conclude the paper in Section 6.

2 PROBLEMS IN DEBUGGING MOBILE AGENT SYSTEMS

Many mobile agents work on many computers and migrate autonomously. This characteristic enables us to reduce network traffic and helps us to construct more robust and faulttolerant distributed systems. However, it is difficult to debug a mobile agent system because of the following reasons.



(a) Autonomous migration. (b) Migration on debugging. (c) Influence from outside.

Figure 1: Problems on debugging a mobile agent system.

Problem 1. Autonomous Migration

In general systems, we usually debug one program. However, there are many mobile agents (programs) in a mobile agent system. Furthermore, mobile agents disperse to many computers because each mobile agent migrates among computers autonomously (Figure 1(a)). Therefore, programmers face difficulty grasping the behavior of mobile agents, thus making it difficult to determine which mobile agent is causing a bug. To debug a mobile agent system, we need a search function that enables us to find out the mobile agent that is causing a bug.

Problem 2. Migration During Debugging

To debug a program on a remote computer, we often use a remote debugger such as breakpoint function and single-step execution function. Remote debugger need to connect the remote computer where the target is running. A mobile agent, however, migrates among computers during debugging. A debugger loses the mobile agent when it migrates because the mobile agent is not running on remote connected computer yet (Figure 1(b)). To debug a mobile agent continuously, we need to change the connection automatically according to the mobile agents migration.

Problem 3. Outside Influences

To specify the cause of a bug, we have to reproduce the bug. On a mobile agent system, the behavior of a mobile agent is affected by the computer environment that the agent is running on and by other mobile agents. Furthermore, a computer may leave the network. These reasons make it difficult to reproduce a bug (Figure 1(c)). Therefore, we need a function that enables us to reproduce the bug.

3 DEBUGGING FUNCTIONS FOR A MOBILE AGENT SYSTEM

In this section, we design a framework to help in debugging a mobile agent system. The framework has three functions: search, single-step execution, and reproduction. The search function helps us to locate a mobile agent in the whole system. The single-step execution function changes a connection automatically according to mobile agent migration. The reproduction function records the state of a agent and the outside effects from other agents and enables programmers to reproduce a bug.

3.1 Search Function

There are many mobile agents working in a mobile agent system. To debug the mobile agent system, programmers need to determine the mobile agent that is causing the bug. Therefore, we propose a search function that helps us find a mobile agent that may be causing a bug.

Typical bugs on a mobile agent system are caused by the migration of a mobile agent. For example, a mobile agent migrates to unexpected computers, does not migrate, or migrates frequently. These migrations cause a defective state in the mobile agent migration route and/or intervals of mobile migration. Therefore, to find such a mobile agent, we prepare a migration log. The migration log consists of the following contents.

node_src address of the computer

node_dest address of the destination computer

staying_time running time on the computer

agent_name role of this mobile agent

agent_id identifier of this mobile agent

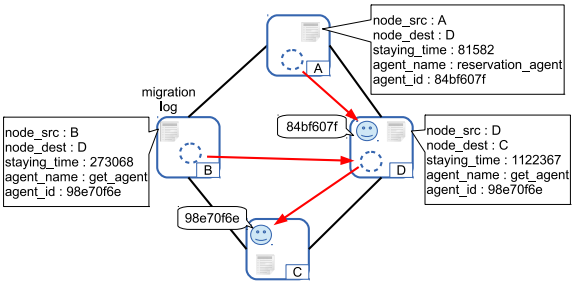


Figure 2: Recording migration logs.

The bugs related to migration are judged from *node_src*, *node_dest*, and *staying_time*.When a mobile agent arrives from a source computer, the search function records the address of the source computer as *node_src*. When a mobile agent departs to a destination computer, the search function records the address of the destination computer as *node_dest*. By tracing these logs, we can determine the migration route of the mobile agent.

staying_time records the difference between agent arrival time and departure time. This enables us to know the defect situation related to a mobile agents migration. *agent_name* is a name and role that the programmer specifies. *agent_id* is a universally unique identifier)(Leach et al., 2005) for identifying each agent. The search function records these logs when mobile agents migrate (Figure 2). *agent_name* and *agent_id* are used for showing a programmer the search results and selecting the target for debugging.

3.2 Single-step Execution Function

To debug a program, it is necessary to see the state of the program in detail. In general, a programmer debugs a program by using a single-step execution function. However, It is difficult to perform this debugging on a mobile agent system because a mobile agent migrates to remote computers. When the mobile agent migrates to other computers during debugging, the general debugger loses the mobile agent. Therefore, to debug a mobile agent continuously, a singlestep execution function for a mobile agent system has to change the connection according to the migration of the mobile agent when it migrates.

To change this connection, the singlestep execution function receives the notification of a migration when the mobile agent migrates to another computer. Then the singlestep execution function changes the connection according to this notification. Here, the single-step execution function might receive multiple notifications concurrently when a mobile agent migrates frequently because network speeds are different in each computer. Therefore, the single-step execution function needs to know the order of notifications to avoid losing the mobile agent. Thus, a notification consists of *agent_id*, which is an identifier for the agent; *node_dest*, which is the IP address of the destination computer; and *id_cnt*, which is the number of migrations.

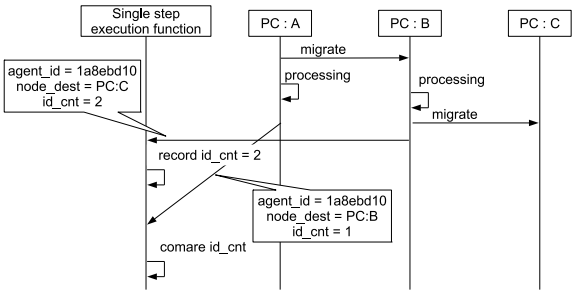


Figure 3: Migration sequence for the single-step execution function.

Figure 3 illustrates the sequence of the single-step execution function. When the mobile agent migrates from PC:A to PC:B, the mobile agent sends the following notification.

- *agent_id* = 1a8ebd10
- *node_dest* = PC:B
- *id_cnt* = 1

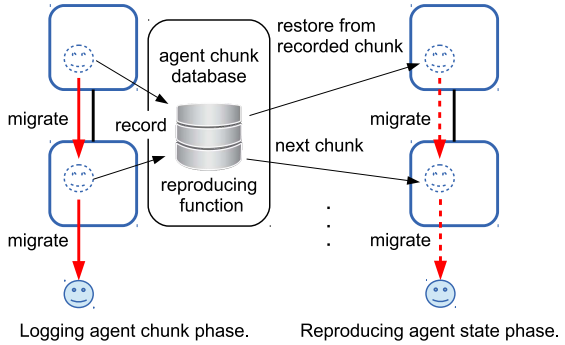
Subsequently, the mobile agent migrates from PC:B to PC:C and sends the following notification.

- *agent_id* = 1a8ebd10
- *node_dest* = PC:C
- *id_cnt* = 2

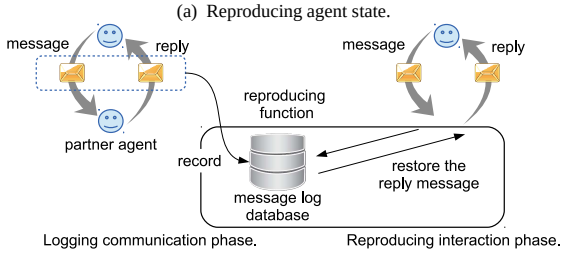
If the later notification arrives at the single-step execution function first, the single-step execution function records *node_dest* with $id\_cnt = 2$. After this process, the former notification arrives at the single-step execution function; the single-step execution function then compares the received *id_cnt* and recorded *id_cnt*; then, it ignores the notification because the received *id_cnt* is smaller than the recorded *id_cnt*. Thus, a single-step execution function can debug a mobile agent even when the agent frequently migrates during debugging.

3.3 Reproduction Function

To discover the cause of a bug, we must execute a program repeatedly to narrow down the cause. However, the migration and interaction characteristics of a mobile agent make it difficult to find the cause because the result of migration and interaction is different according to each situation. Therefore, we propose a reproduction function, which records the behavior of



(a) Reproducing agent state.



(b) Reproducing interaction.

Figure 4: Reproducing agent state and interaction.

a mobile agent and reproduces a past agent behavior. Using this information, programmers can confirm the behaviors as often as they want without executing a program repeatedly, thus helping them to identify the causes of bugs.

3.3.1 Migration

To reproduce a bug, it is necessary to reproduce the state of the debugging program. However, the migration of a mobile agent is affected by the condition of the network at that time. To avoid this, we use the characteristic of a mobile agent. The migration of a mobile agent is implemented by the following steps (Satoh, 2006):

Step 1. The runtime system on the source computer suspends the execution of the agent.

Step 2. It marshals the agent into a bit chunk that can be transmitted over a network.

Step 3. It transmits the chunk to the destination computer through the underlying network protocol.

Step 4. The runtime system on the destination computer receives the chunk.

Step 5. It unmarshals the chunk into the agent and resumes the agent.

Here, the chunk, which contains the state of the mobile agent inside, is created (marshaled) when a mobile agent migrates. Therefore, we record the chunk and use the recorded chunk when a mobile agent migrates. This enables us to avoid the effect of the network condition.

3.3.2 Interaction

Mobile agents interact with other mobile agents. Therefore, to discover the cause of a bug, the debugger needs to reproduce the interactions that led to the problem. To reproduce the interactions, we need to record the behavior of all partner agents.

The difficulty in doing this reproduction is that many mobile agents are running on a system. Therefore, we record only the effect from partner agents. Here, the interactions of mobile agents are realized by message passing. Therefore, we record a *sent message*, which is the message the agent sends to the partner, and *reply message*, which is the message to which the partner agent replies as a key-value pair. When the reproduction function reproduces the interaction, the function uses recorded messages instead of partner agents.

3.3.3 Reproducing Behaviors

A reproduction function reproduces the behaviors of a mobile agent. The reproduction function builds a virtual agent runtime environment for executing mobile agents. The function restores the mobile agent from a recorded chunk and executes the mobile agent on the virtual agent runtime environment. The virtual agent runtime environment has only one computer and one mobile agent that the function has recorded. Thus, it reproduces migration and interactions virtually using recorded chunks and messages.

When the reproduction function reproduces the migration of a mobile agent, the function replaces the state of the mobile agent from the current chunk to the next chunk (Figure 4(a)). Thus, we can reproduce the migration irrespective of network conditions.

When an agent tries to interact with a partner agent, the reproduction function steals the *sent message* without sending it to the partner. The function obtains the *reply message* which is the value of the *sent message* by searching with the *sent message* from recorded messages. The function hands this *reply message* to the agent instead of replying the message from partner agent (Figure 4(b)). Thus, the reproduction function can reproduce the interaction.

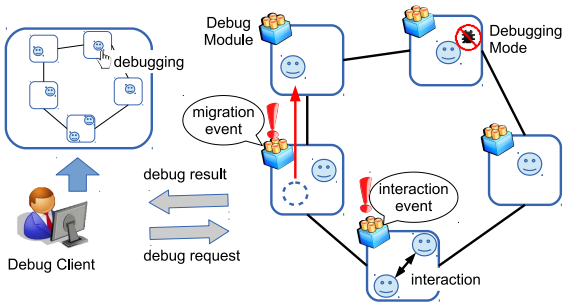


Figure 5: Overview of the debugger.

4 IMPLEMENTATION AND EVALUATION

We have implemented a prototype of our debugger on a mobile agent framework called Maglog (Kawamura et al., 2005). To evaluate our proposed debugger, we have experimented by using two applications that were developed on Maglog.

4.1 Implementation of the Debugger

Maglog is implemented by extending PrologCafé (Banbara et al., 2005), which can run on JVM. PrologCafé has a Prolog-to-Java source-to-source translator and a Prolog interpreter. The mobile agent of Maglog is implemented by extending the Prolog class, which has a Prolog engine. When a mobile agent migrates, the agent runtime environment converts a mobile agent state to a chunk by java object serialization (Oracle Corporation, 2014) and transmits it to another computer.

4.1.1 Construction of the Debugger

Our proposed debugger consists of a *debug modules* located on each computer and a *debug client* located on the programmer's computer (Figure 5). A *debug module* records the event logs of agent behaviors, such as agent migrations of mobile agent behaviors running on its computer. A *debug client* provides a user interface to debug for programmers and communicates with *debug modules*. Both *debug module* and *debug client* have an HTTP server. The *debug client* sends debug requests to *debug modules*. A *debug module* sends a debug result to the *debug client*.

4.1.2 Searching Methods

Our proposed search function has two searching methods. One is the flooding method, which sends the search query through every outgoing link except the one on which it arrived. The other is the forwarding method, which tracks the route of mobile agent migration by sending the search query to the destination of the mobile agent migration by using the migration log.

The flooding method is used for searching many mobile agents or computers, such as searching *agent_name*, because this method can spread the search query among many computers. Searching the migration route is realized by tracing the migration log by the forwarding method. The programmer can thereby check an agent and its route.

4.1.3 Single-step Execution Function

To use the single-stepping execution function and the breakpoint function, we find the mobile agent to debug by the search function. When a programmer sets a breakpoint to a mobile agent, a *debug client* sends a request to set the breakpoint to the *debug module* on which the mobile agent is running. The *debug module* sets the breakpoint on the mobile agent. When the mobile agent executes the breakpoint, the *debug module* interrupts the execution of the mobile agent and sends its state to the *debug client*. The *debug client* shows the programmer the mobile agents state. Similarly, when a programmer performs the mobile agent single-step execution, the *debug client* sends a request, the *debug module* interrupts the execution of the mobile agent, and replies with the state of the mobile agent.

Figure 6 shows the user interface of the single-stepping function. The top of the window shows the IP address in which the mobile agent is running. The right side of the window shows the current predicate that the mobile agent is executing. The left side of the window shows the code of the mobile agent.

This shows the situation where the mobile agent migrates from 192.168.132.56 to 192.168.132.55. At the left, a programmer sets the breakpoint to *go*, which is a built-in predicate for migration in Maglog, and the mobile agent is interrupted there. The window after the single-step execution is at the right in the figure. The current IP address changes to 192.168.132.55 and the current predicate changes to *retract*. Therefore, the programmer can continue the single-step execution function even if a mobile agent migrates to another computer during debugging.

4.1.4 Reproduction Function

When a programmer notices that something is wrong in the system, the programmer sets a mobile agent to

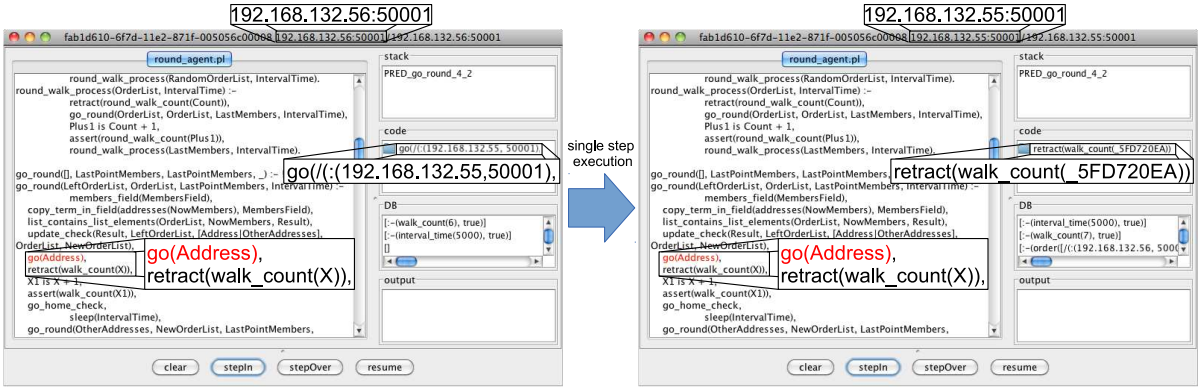


Figure 6: User Interface of the single-stepping function.

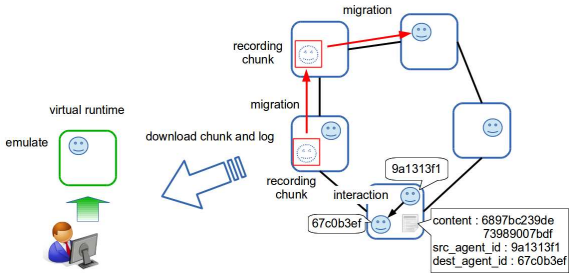


Figure 7: Overview of reproduction.

debug. *Debug module* records the chunk and messages of the mobile agent. *Debug client* has a virtual agent runtime environment that a mobile agent executes. *Debug client* restores the chunk and executes the mobile agent on the virtual agent runtime. environment (Figure 7).

When a programmer sets the breakpoint for the mobile agent, *debug client* downloads the chunk and messages from the *debug module*. *Debug client* restores the past state of the mobile agent and executes it. Reproduction is realized as follows.

- When the mobile agent tries to migrate, the reproduction function downloads the next chunk and recorded messages from the destination *debug module*.
- When the mobile agent tries to interact, the reproduction function hands a recorded message to the mobile agent.
- When the mobile agent executes the breakpoint, the *debug client* interrupts the execution of the mobile agent.

4.2 Evaluation

We implemented two applications for the evaluation of our debugger. The first application is a distributed hash table (DHT) and the second application is a hotel reservation application (HOTEL). Table 1 shows the bugs that occurred in the implementation of each application.

When we were developing the DHT application, it failed to obtain a value with a key from the table. In this DHT application, a mobile agent named "get_agent" migrated from one computer to other computers to search for a value related with a key. After "get_agent" finished searching for the value, the agent showed its result to us. However, we could not obtain the result. Therefore, we noticed that the application has bugs.

On this debugging, "get_agent" did not come back to our computer. We guess that the cause of it is that the agent died or failed to migrate on the destination computer. Therefore, we need to confirm the following:

- Whether the agent is alive or not;
- Where the agent is running on.

In order to confirm this, we searched a location of "get_agent" by the search function. The result of the search is that this agent was running on an unexpected computer. We noticed that the agent failed to migrate. Next, we should guess the cause of it. In this application, "get_agent" read a destination address from the address list managed by "network_agent". Therefore, we should confirm following:

- Whether the address in the address list is correct;
- Whether the process after reads is succeeded.

Therefore, we restored the previous state, in which the agent migrated to the unexpected computer, by a reproduction function, and confirmed the migration by the single-step execution function. This caused us to notice that "get_agent" read a wrong destination address and a list of addresses did not have a correct one. Thus, we noticed that this bug was caused by "network_agent".

Table 1: Bugs and the useful debugging functions.

	bugs	search	stepping	reproduction
DHT	fail to store value	✓	✓	✓
	fail to get value	✓	✓	✓
	fail to join network		✓	✓
HOTEL	fail to initialize hotel information		✓	
	lost a part of information	✓	✓	
	search wrong data	✓	✓	✓

Table 2: Number of keystrokes and clicks.

logs	keystrokes	clicks
without our debugger	869	167
with our debugger	514	128
reduction rate	41%	24%

While confirming the locations of mobile agents without our proposed debugger, we need to insert print statements and confirm each log dispersed. While finding defective functions and confirming the values of an instance, migrations of a mobile agent make it difficult to debug even when using an ordinary remote debugger because we lose a mobile agent when it migrates. Furthermore, the same bugs may not even occur if we deploy the same agent because the situation may change.

In contrast, when debugging with our proposed debugger, programmers can obtain the locations of mobile agents easily using the searching function. Because the single-step execution function can grasp the behaviors of a mobile agent, a programmer can be debugging even when the mobile agent migrates to other computers. The reproduction function can help us to confirm the causes of bugs by restoring a past state of a mobile agent.

Table 2 shows the number of keystrokes and clicks on debugging. As shown in Table 2,the keystroke count was reduced by 41% and the click count was reduced by 24% by our proposed debugger. Thus, our proposed debugger can effectively debug a mobile agent system.

LAM (Logical Agent Mobility)

5 RELATED WORKS

There are description languages to define interactions and scenarios among agents. If the definitions of interactions and scenarios are valid logically, native source codes generated from these definitions do not have any bugs. Mobile Object-Z (MobiOZ) (Taguchi and Song Dong, 2002), LAM (Logical Agent Mobility) (Xu et al., 2003) are description languages that define interactions and scenarios among agents for a mobile agent system. These languages verify the model of a mobile agent application by the Simple Process meta language Interpreter model checker (Ben-Ari, 2008) and the Construction and Analysis of Distributed Processes toolbox (Garavel et al., 2011). However, To apply these approaches to an existing application, a programmer must design and implement it again for these languages. Furthermore, these approaches are not realistic in practice because the costs increase with an increasing scale and complexity of the applications. Furthermore, some applications are not compatible with these approaches.

Several researchers have proposed a tool to visualize interactions among agents (Ndumu et al., 1999; Lam and Barber, 2005; Padgham et al., 2005; Vigueras and Botia, 2008; Cabac et al., 2009), but these tools do not focus on the mobility of agents. JADE (Bellifemine et al., 2010) and Agent Factory (Collier, 2007; Brazier et al., 2002) is a platform for multi-agent systems that includes a debugger. These platforms support the mobility of agents; nevertheless, their debuggers do not support and focus on the mobility of agents.

(Lynch and Rajendran, 2007) suggested requirements for integrated development environments (IDEs) to support the construction of multi-agent systems, but they do not focus on agent mobility. MiLog (Fukuta et al., 2000) is a framework (platform) including an IDE for mobile agent systems that can visualize the locations of agents in a network and dump logs of each agent. However, it supports a debugging only on the visualization of agent locations.

6 CONCLUSION

In this paper, we discussed problems in debugging a mobile agent system, and proposed the debugging functions: searching, single-step execution, and reproduction function. Experimental results show that our proposed functions can mitigate debugging difficulty on agent migration.

REFERENCES

Banbara, M., Tamura, N., and Inoue, K. (2005). Prolog cafe : A prolog to java translator system. In *Proc. of the 16th International Conference on Applications of Declarative Programming and Knowledge Management*, pages 1–11.

Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2010). *JADE PROGRAMMERS GUIDE*.

Ben-Ari, M. (2008). *Principles of the Spin Model Checker*. Springer London.

Brazier, F. M. T., Overeinder, B. J., van Steen, M., and Wijngaards, N. J. E. (2002). Agent factory: generative migration of mobile agents in heterogeneous environments. In *Proc. of the 2002 ACM symposium on Applied computing*, pages 101–106.

Cabac, L., Dörges, T., Duvigneau, M., and Moldt, D. (2009). Requirements and tools for the debugging of multi-agent systems. In *Proc. of the 7th German conference on Multiagent system technologies*, pages 238–247.

Collier, R. (2007). Debugging agents in agent factory. In *Proc. of the 4th international conference on Programming multi-agent systems*, pages 229–248.

FIPA (2014). FIPA. Web. http://www.fipa.org.

Fukuta, N., Ito, T., and Shintani, T. (2000). Milog: A mobile agent framework for implementing intelligent information agents with logic programming. In *Pacific Rim International Workshop on Intelligent Information Agents*.

Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2011). CADP 2010: a toolbox for the construction and analysis of distributed processes. In *Proc. of the 17th international conference on Tools and algorithms for the construction and analysis of systems*, pages 372–387.

Hurson, A. R., Jean, E., Ongtang, M., Gao, X., Jiao, Y., and Potok, T. E. (2010). Recent Rdvances in Mobile Agent-Oriented Applications. In *Mobile Intelligence: Mobile Computing and Computational Intelligence*, Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 1st edition.

Kawamura, T., Motomura, S., and Sugahara, K. (2005). Implementation of a logic-based multi agent framework on java environment. In *Proc. of IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pages 486–491.

Lam, D. N. and Barber, K. S. (2005). Debugging agent behavior in an implemented agent system. In *Proc. of the Second international conference on Programming Multi-Agent Systems*, pages 104–125.

Leach, P., Mealling, M., and Salz, R. (2005). A Universally Unique IDentifier (UUID) URN Namespace. Request for Comments 4122.

Lynch, S. and Rajendran, K. (2007). Breaking into industry: tool support for multiagent systems. In *Proc. of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 136:1–136:3.

Milojicic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S., and White, J. (1998). Masif: The omg mobile agent system interoperability facility. *Personal Technologies*, pages 117–129.

Ndumu, D. T., Nwana, H. S., Lee, L. C., and Collis, J. C. (1999). Visualising and debugging distributed multi-agent systems. In *Proc. of the third annual conference on Autonomous Agents*, pages 326–333.

Oracle Corporation (2014). Object Serialization. Web. http//docs.oracle.com/javase/tutorial/jndi/objects/serial.html.

Outtagarts, A. (2009). Mobile Agent-based Applications : a Survey. *International Journal of Computer Science and Network Security (IJCSNS)*, pages 331–339.

Padgham, L., Winikoff, M., and Poutakidis, D. (2005). Adding debugging support to the prometheus methodology. *Eng. Appl. Artif. Intell.*, 18(2):173–190.

Satoh, I. (2006). Mobile agents. In Scerri, P., Vincent, R., and Mailler, R., editors, *Coordination of Large-Scale Multiagent Systems*, pages 231–254. Springer US.

Taguchi, K. and Song Dong, J. (2002). An Overview of Mobile Object-Z. In *Proc. of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM-2002)*, pages 144–155.

Vigueras, G. and Botia, J. A. (2008). Tracking causality by visualization of multi-agent interactions using causality graphs. In *Proc. of the 5th international conference on Programming multi-agent systems*, pages 190–204.

Xu, D., Yin, J., Deng, Y., and Ding, J. (2003). A formal architectural model for logical agent mobility. *IEEE Trans. on Softw. Eng.*, 29(1):31–45.