iML: A Logic-based Framework for Constructing Graphical User Interface on Mobile Agents

Naoki Fukuta, Nobuaki Mizutani, Tadachika Ozono, and Toramatsu Shintani

Department of Intelligence and Computer Science,
Nagoya Institute of Technology,
Gokiso-cho, Showa-ku, Nagoya 466-8555, JAPAN
{fukuta,miztani,chika,tora}@ics.nitech.ac.jp

Abstract. Mobile agent technology is an emerging technology that allows easier design, implementation, and maintenance of distributed systems. Mobility enables agents to reduce network load, overcome network latency, and handle network disconnections. Few mobile agent systems support Prolog language, and no system supports GUI programming on Prolog language. Prolog's first order logic representation has benefits for constructing mobile agents. In this paper, we present a Prolog-based programming framework, *iML*, for constructing mobile agents with graphical user interfaces. On the *iML* framework, we provide three mechanisms: runtime environment for mobile agents, a Prolog-based mobile agent description language, and a visual tool to design GUI layouts. We evaluate the *iML* framework from two viewpoints: migration performance and the usability of the framework. Experimental results show that the *iML* method is 6.7 times smaller in size, and 6.0 times faster in migration speed in comparison with the Java serialization technique. We used the *iML* framework at our knowledge programming lecture. During the lecture, 60 students attended after which 45 students could create GUI-based mobile agents, and 16 students could create excellent applications by using the framework.

1 Introduction

Mobile agent technology is an emerging technology that allows easier design, implementation, and maintenance of distributed systems. Mobility enables agents to reduce network load, overcome network latency, and handle network disconnections[4]. Mobile agent systems allow users to implement mobile agent applications easily[4, 3, 6]. In many mobile agent systems, the mobility is implemented on a certain programming language (e.g., Java), or sometimes on a specialized language (e.g., *Telescript*[7]). Those mobile agent systems force programmers to use a certain programming language for developing mobile agent applications. According to the list of mobile agent systems[5], there are at least 70 mobile agent systems. Despite this, few systems support Prolog language, and no system supports GUI programming on Prolog language. Prolog's first order logic representation has two benefits for constructing mobile agents:

– *The powerful expression* The programmer can use powerful expression of first order logic to implement complex behaviors of the mobile agent.
– *The simplicity of their codes* The codes written using first order logic is simple and compact so that can reduce both of programmer's load and code-migration overhead.

The mobile agent technology has also benefits for implementing network-based distributed intelligent systems. We expect a synergy effect on combining prolog language and the mobile agent technology. We have presented a prolog-based mobile agent framework *MiLog* for this purpose. In our previous implementation, GUIs are implemented by using Java and Java to *MiLog* communication interface. There are two problems using this approach, the usability and the performance. On GUI programming, there are events established by the user's operations. The programmer may need some codes to handle and pass the event to appropriate prolog queries. At this situation, the system do not provide strong mobility on the Java programming even if the *MiLog* framework provides strong mobility on prolog programming. The state of GUIs are preserved by using the Java serialization technique. The problem is the performance of the Java serialization is low and some GUI components (especially built by Swing components) cannot preserved by Java serialization. In this paper, we present a Prolog based programming framework, *iML*, for constructing mobile agents with graphical user interfaces. On the it iML framework, all GUIs are represented by using first order logic in compact form. These represented GUIs are operated seamlessly by prolog programs. On the *iML* framework, we provide three mechanisms: a runtime environment for mobile agents, a Prolog-based mobile agent description language, and a visual tool to design GUI layouts. We evaluate the *iML* framework from two viewpoints, migration performance and the usability of the framework.

The following sections are organized as follows. In Section 2, we present the model and the structure of the framework. In Section 3, we show some implementation details of the framework. In Section 4, we show the user interface of the *iML* visual tools. In Section 5, we evaluate the framework in terms of its performance and usability. In Section 6, we make some concluding remarks.

2 The model and structure of *iML*

Our programming model is based on the *MiLog* mobile agent framework[2]. Figure 1 shows the agent migration model of the *MiLog* framework. On the *MiLog* framework, a mobile agent is an object which contains its own Prolog interpreter. Each mobile agent has an individual clause database (program code and data) and a thread (stack area). The *MiLog* framework supports strong mobility[1], which migrates the thread and the clause database of the agent.

On the *MiLog* framework, we simply added a migration predicate '`move/1`', which migrates the state of the interpreter to another runtime environment. The predicate '`move/1`' does not change any control flow of the Prolog program except for failure of the migration. Figure 2 shows a sample behavior of the migration predicate '`move/1`' on the code level. The left of the figure shows a sample migrating program. The right of the figure shows the trace of a query '`?- migrationSample.`' to the program on the left of the figure. Once the predicate '`move/1`' is executed, whole state of the program will be migrated to the target host. The predicate '`move/1`' does not backtrack but the program can be backtracked previous to the '`move/1`' predicate. Programmers have to learn only the behavior of the predicate '`move/1`' to understand the behavior and semantics of the agent migration. This approach is good for novice programmers

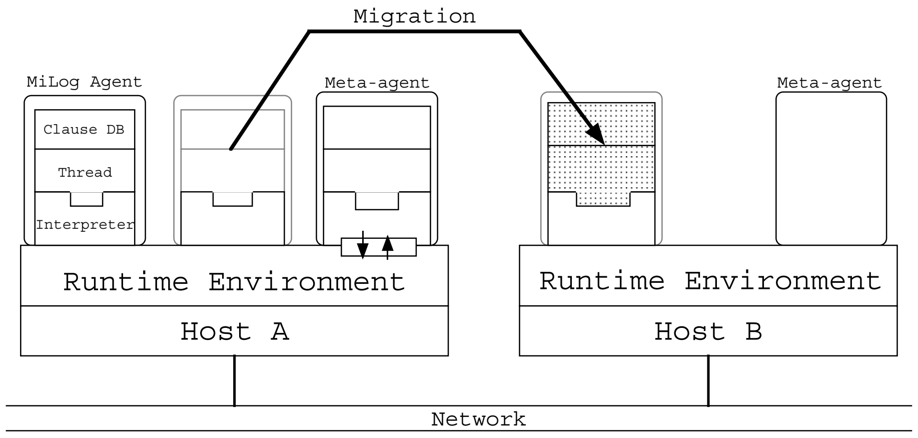


Fig. 1. Agent migration model on the *MiLog*

because the behavior is simple enough for novices to understand. This approach is also useful for the reuse of many legacy Prolog program codes.

On the *iML* framework, each *iML* agent is an extended mobile agent on the *MiLog* framework. The agent is extended to handle GUI components and events. The agent has also capability of preserving the state of the GUI components. Figure 3 shows the program code of GUI preservation on *iML* agents. The predicate '`move/1`' is overwritten by the shown code on *iML* agents. The predicate '`moveInterpreter/1`' has an equivalent function of the original '`move/1`' predicate. First, the predicate '`assertAllGUI/0`' is executed in order to preserve the current state of the GUI. Then, the predicate '`moveInterpreter/1`' is executed to migrate the interpreter state of the agent. If the migration is success, the predicate '`restoreAllGUI/0`' is executed to restore the preserved state of the GUI. In this stage, event handlers are also re-instantiated on the target host. If the migration is failure, then backtracking will occur to another '`move/1`' definition. Here, simply '`restoreAllGUI/0`' is executed on the original host. The mechanism is simple enough to preserve the GUI on the *iML* framework because of the strong migration technique.

3 Implementation

3.1 Overview

The *iML* framework is implemented by using Java and *MiLog*. The *MiLog* framework is implemented on Java. The *MiLog* framework is small enough (approximately 220Kbytes) that it can be downloaded and installed easily by many users. The Java serialization technique is used by many Java-based mobile agent systems. Due to the limitation of the Java implementation, capturing the thread state is not supported on the Java serialization. We realized thread state capture by implementing a new logic program engine on Java. On the architecture of our engine, the thread state of the Prolog

```
migrationSample :-
  member(HOST,[hostB,hostC]),
  move(HOST),
  myMasterIsHere.

member(X,[X|_]).
member(X,[_|Y]) :-
  member(X,Y).
```

A Sample Migrating Code on *MiLog*

```
CALL migrationSample
CALL  member(HOST1,[hostB,hostC])
EXIT  member(hostB,[hostB,hostC])
CALL move(hostB)
```

Executed on the hostA

```
EXIT move(hostB)
CALL myMasterIsHere
FAIL myMasterIsHere
REDO member(HOST1,[hostB,hostC])
CALl  member(HOST1,[hostC])
EXIT  member(hostC,[hostC])
EXIT member(hostC,[hostB,hostC])
CALl move(hostC)
```

Executed on the hostB

```
EXIT move(hostC)
CALL myMasterIsHere
EXIT myMasterIsHere
<<EXECUTION COMPLETE>>
```

Executed on the hostC

The Trace of the Query

Fig. 2. A Sample Migrating Code on *MiLog*

program is represented as a tree of Java objects which can be captured by using Java serialization.

3.2 GUI preservation

On the iML framework, all GUIs are constructed by iML *components*. Each *component* has a unique *ID* to identify the *component* on the *MiLog* program. Each *component* has corresponding *Java class* as the type of the *component*(e.g., `Java.awt.Button`). The state of the GUI is represented by *properties* and *inclusions*. The *properties* are properties of the *component* as the Java object. The *inclusions* are relations between *components*. *Properties* and *inclusions* of *components* are preserved on the clause database of the *MiLog* agent. Figure 4 shows an example representation of a simple GUI which contains a frame (window) with a button. The predicate `iml_cpp/3` represents the *ID*, corresponding *Java class*, and *properties* of the *component*. The predicate `iml_ctr/2` represents the *inclusion* of the *components*.

4 The user interface of the visual tool

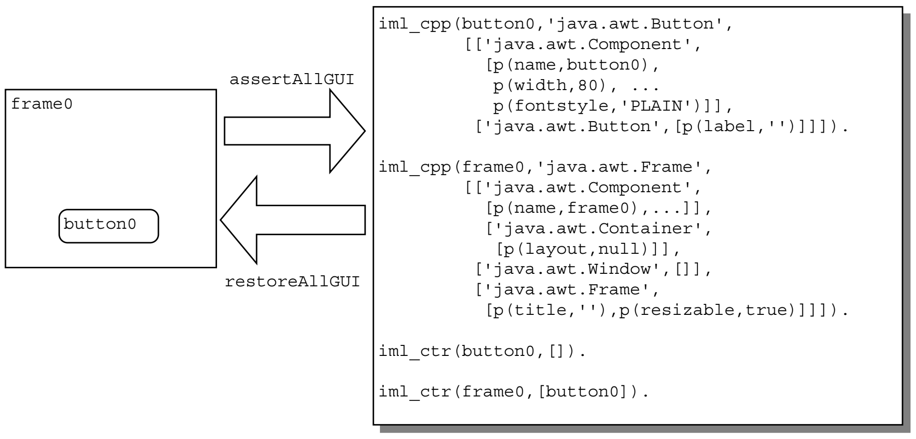
The *iML* visual tool has three major modes, *sketch mode*, *edit mode*, and *exec mode*.

On the *sketch mode*, the system provides users with an intuitive GUI outlook design workspace in which designers can place and move GUI components (e.g., Buttons, Panels, etc.). Figure 5 shows the user interface on the *sketch mode*. The window on the

```
move(X) :-
  assertAllGUI,
  moveInterpreter(X),
  restoreAllGUI,!.

move(X) :-
  restoreAllGUI.
```

Fig. 3. The program code of GUI preservation



```
iml_cpp(button0,'java.awt.Button',
        [['java.awt.Component',
          [p(name,button0),
           p(width,80), ...
           p(fontstyle,'PLAIN')]],
         ['java.awt.Button',[p(label,'')]]]).

iml_cpp(frame0,'java.awt.Frame',
        [['java.awt.Component',
          [p(name,frame0),...]],
          ['java.awt.Container',
           [p(layout,null)]],
         ['java.awt.Window',[]],
          ['java.awt.Frame',
          [p(title,''),p(resizable,true)]]]).

iml_ctr(button0,[]).

iml_ctr(frame0,[button0]).
```

Fig. 4. A Sample Clause Representation of a Preserved GUI

left is called the *inspector window*, and the window on the right is called the *layout design window*. Users can try any GUI layouts by using simple mouse operation (e.g., dragging the edge of the component to adjust the component size). The strict value of the components can be set by using the *inspector window*. The *layout design window* is also provided on the *edit mode*. The difference is that the *sketch mode* provides a more lightweight, abstract tool. Therefore, users can quickly build abstract design of the GUI with good response.

Edit mode provides a more detailed design tool in which designers can change the properties of a certain component. Figure 6 shows the user interface on the *edit mode*. Although the *layout design window* (the window on the right) looks like almost same on the *sketch mode*, the window shows the detailed appearance of the GUI by applying many properties (e.g., font size, label, color, etc.). Users can also add and modify event-handling codes to respond to the user's behavior (e.g., pressing a button).

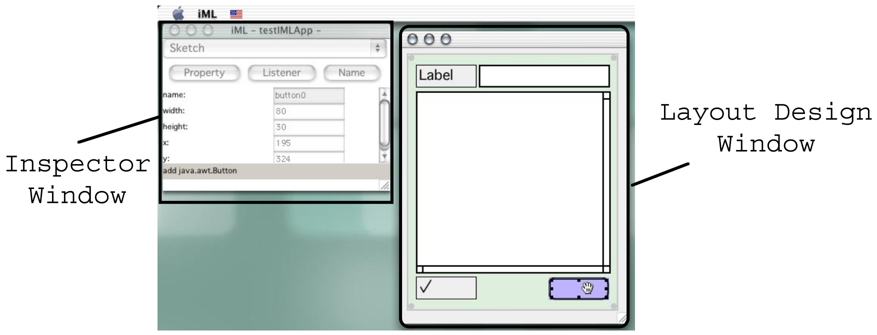
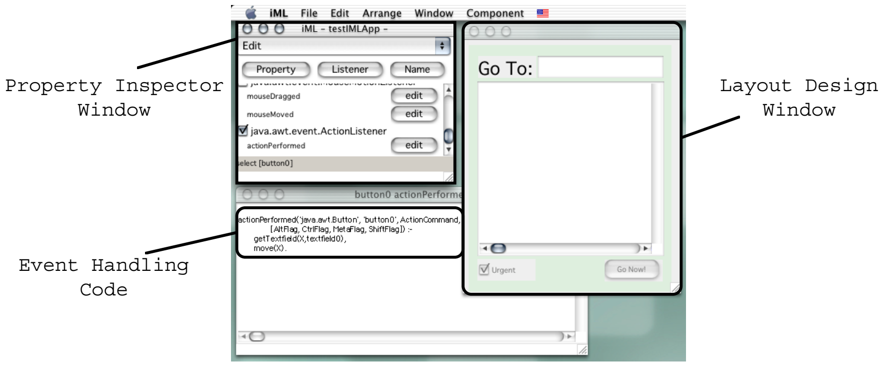


Fig. 5. User Interface on the *sketch mode*



```
actionPerformed('java.awt.Button', 'button0', ActionCommand,
        [AltFlag, CtrlFlag, MetaFlag, ShiftFlag]) :-
    getTextfield(X,textfield0),
    move(X).
```

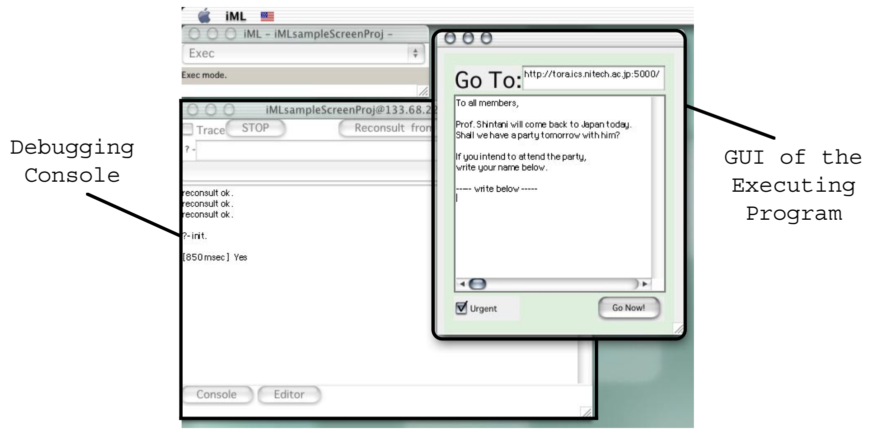
Fig. 6. User Interface on the *edit mode*

Exec mode provides a testing environment in which designers can test and verify the behavior of GUI components which is established by the designer on the *edit mode*. Figure 7 shows the user interface on the *exec mode*. The *debugging console* (the window on the left) is available to trace and debug the GUI code directly.

Finally, the designed application is compiled into a *MiLog* agent which can be run on the *MiLog* runtime environment.

5 Evaluation

We have evaluated the *iML* framework from two viewpoints: performance and usability. To measure the performance of the *iML*'s mobility, we compare it with the Java serialization technique. Use of the Java serialization technique is popular in many Java-based mobile agent systems. For this comparison, we have built a calendar application by using both Java and *iML*. This application is capable of using both the Java serialization and the *iML*'s clause representation on demand. We used Java2 SE 1.3.0 with HotSpot


Debugging Console
GUI of the Executing Program
iML
iML – iMLsampleScreenProj –
Exec
Exec mode.
iMLsampleScreenProj@133.68.2
Trace
STOP
Reconsult fron
?-
Console
Editor
Go To:
http://tora.ics.nitech.ac.jp:5000/
Urgent
Go Now!

```
reconsult ok.
reconsult ok.
reconsult ok.

?-init.

[850msec] Yes
```

```
To all members,

Prof. Shintani will come back to Japan today.
Shall we have a party tomorrow with him?

If you intend to attend the party,
write your name below.

----- write below -----
```

Fig. 7. User Interface on the *exec mode*

client VM on a 750MHz Windows laptop PC. The experimental results are shown in Table 1. The listed results are the average values per one migration. From this result, *iML*'s method is 6.7 times smaller in size, and 6.0 times faster in migration speed. Basically, the Java serialization mechanism is designed for general purpose use and is not tuned for maximum performance. So it is natural that the performance of Java serialization shows a low value. But we consider the performance of our method good enough for prototyping and teaching purposes.

Table 1. The performance of agent migration

	Java serialization	*iML*
transmitting data size [bytes]	59419	8876
elapsed time [msec]	4952	829

We used the *iML* framework at our knowledge programming lecture. On the lecture, 60 students attended, after while 45 students could create GUI-based mobile agents, and 16 students could create excellent applications by using the framework.

6 Conclusions

In this paper, we presented a Prolog-based programming framework for constructing graphical user interfaces for mobile agents. The performance and usability is sufficient for use in lecturers. Currently our implementation is available for Java2 SE, but not available for Java2 ME (especially for PDA and mobile phones). Our future work will focus on the implementation of a simplified framework for Java2 ME environments.

References

1. A.Fuggetta, G.P.Picco, and G.Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, may 1998.
2. N. Fukuta, T. Ito, and T. Shintani. A Logic-based Framework for Mobile Intelligent Information Agents. *Poster Proceedings of 10th International World Wide Web Conference(WWW10)*, pp.58–59, 2001.
3. T. Kawamura, N. Yoshioka, T. Hasegawa, A. Ohsuga, and S. Honiden. Bee-gent : Bonding and encapsulation enhancement agent framework for development of distributed systems. In *Proc. of the 6th Asia-Pacific Software Engineering Conference*, 1999.
4. D.B. Lange and M.Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
5. Mobile Agents List. http://inf.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal /preview/preview.html.
6. I. Satoh. Mobilespaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 161–168. IEEE Press, 2000.
7. J. E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 19, pages 437–472. AAAI Press/The MIT Press, 1997.