REFLECTIVE AGENT PROGRAMMING ENVIRONMENT RXF AND ITS MULTI-AGENT TRACER

TADACHIKA OZONO and TORAMATSU SHINTANI
Department of Intelligence and Computer Science, Nagoya Institute of Technology
Address: Gokiso, Showa-ku, Nagoya, 466-8555, JAPAN
TEL: +81-52-744-3153 FAX: +81-52-745-5477
E-mail: {chika, tora}@ics.nitech.ac.jp
Keywords: Autonomous Systems, Intelligent Systems, Agent Programming, Agent Debugging

Abstract: In this paper, we describe a new programming environment, RXF, for building a multi-agent system (MAS), and its debugger. RXF is used for realizing intuitive development of an MAS that consists of reflective agents. The reflective agents are intelligent and autonomous agents based on a reflection. RXF provides the capabilities of a constraint-logic programming language and a reflection mechanism for realizing reflective agents. In constraint-logic programming, programmers can represent various data by using predicates as well as process numerical data. We also present a multi-agent tracer that is a debugging tool for developing MASs. We show the experimental result of the tracer and conclude that the debugger can be used for debugging MASs.

1 INTRODUCTION

We have developed RXF (Reflective Familiar) [1, 2, 3, 4], a new programming environment for building a multi-agent system (MAS). The main aim of this research is to create a programming environment for the efficient development of MASs instead of implementing an agent-oriented language [5]. In other words, we are interested in what types of functions are required when building MASs. An MAS in RXF consists of reflective agents, which are intelligent and autonomous agents based on reflection [6]. RXF employs constraint-logic programming (CLP) [7], multi-thread programming, and reflections [6]. The multi-thread programming and the reflections are used for concurrent programming [8] and meta-level programming [9], respectively. In CLP, we can represent various data by using logical predicates [10, 11, 12]. Agents in RXF run based on a thread process [1]; that is they can communicate with each other through ports (message-passing mechanisms) in RXF. The ports provide functions for using various communication protocols (e.g., TCP/IP, AppleTalk [13], etc). In addition, RXF provides an easy communication method for implementing MASs over networks. We have created RXF window interfaces that are effective environments for programming agents. RXF provides a multi-agent tracer that allows programmers to effectively debug MASs. A tracer for MASs can be considered to be a tracer for concurrent programs. The main problems associated with debugging concurrent programs are increased complexity, the "probe effect," and the lack of a synchronized global clock [14]. The "probe effect" [15] refers to the fact that any attempt to gain more information about the program may contribute to the difficulty of reproducing the erroneous behaviors. Many techniques for debugging concurrent programs have been proposed [14]; we have chosen tracing as our debugging technique. Tracing is a method to dynamically examine the state of programs running. We think tracing is good for interactive debugging because programmers can control the flow of programs as they are debugging. Our system is based on an interpreter, because we regard interaction during debugging to be important for rapid development. Generally speaking, tracing causes the "probe effect." Our tracer can reduce the "probe effect" when debugging MASs.

In this paper, we propose an RXF programming environment that allows us to program MASs efficiently. The paper is organized as follows: Section 2 presents the features of RXF. Section 3 presents an MAS debugging problem and its solution. Section 4 presents related work regarding RXF. Some concluding remarks are presented in Section 5.

2 RXF

2.1 SUBSYSTEMS

RXF is a programming environment for developing MASs. We implement RXF by using the C++ and Java [4] programming languages on a Macintosh. Figure 1 shows the three components of RXF: an agent operating system (AOS), the name manager, and the agent description language (ADL), which allows one to build agents. In RXF, ADL is based on a CLP language that has a reflection mechanism that can be used by the language. AOS provides various functions for building agents. AOS in RXF has a multi-thread and a port subsystem. The multi-thread subsystem provides functions that enable concurrent processing. The port subsystem is an I/O subsystem of ADL. For example, the port subsystem is used for implementing a file I/O system, a message-passing system, and a user interface system. The subsystems of AOS, the name manager, and ADL are implemented by directly using the Java language functions. The name manager manages relations between names (symbols) on ADL and subsystems of AOS. Specifically, ADL can access the subsystems with names that are managed by the name manager. Agents (including meta-agents) are built in ADL.

RXF can provide functionalities for constraint-logic programming, multi-thread programming, and reflection. CLP is a new class of programming languages combining the declarativity of logic programming with the efficiency of constraint solving [16]. In CLP, we can represent various data by using predicates and we can process numerical data. By using multi-thread programming, agents can run in parallel in RXF. In the agents, multiple threads run in parallel. For example, when a thread is running, other threads in the agent can process some tasks and events (e.g. interruption handling). Multi-thread programming enables concurrent programming. Reflection is used for meta-level programming. RXF can also provide an integrated development environment (IDE for RXF programs).

Agents can communicate with other agents by ports (message-passing mechanisms) in RXF. The ports provide functions for using various communication protocols. The state of an agent in a network environment changes dynamically in response to communication with other agents. A logic programming language deals with static programs and data. In order to implement agents using a logic programming method, we need to develop new methods for handling dynamic data. In implementing an RXF environment, we propose the port mechanism to deal with the dynamic data. An agent can only interact outside of his/her environment by using the ports. The ports translate low-level data such as bit sequences into logical predicates that are available in RXF. Agents have ports for message passing, generating meta-level expressions, and using a graphical user interface. A port for message-passing is called a message port, while a port for generating meta-level expressions is called an agent port. Generating meta-level expressions and modifying states of an agent are implemented by an agent port.

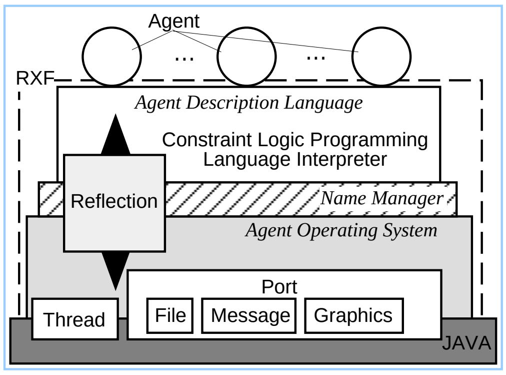


Figure 1: THE OUTLINE OF SUBSYSTEMS IN RXF

2.2 META-LEVEL PROGRAMMING FOR REFLECTIVE AGENTS

An agent in RXF consists of two components: a CLP language interpreter and an agent management system. The interpreter evaluates agent programs written in the CLP of RXF. The agent management system manages memories, files, messages, events, and interruptions for the agent programs. In RXF, the agent management system is also realized by an interpreter, as mentioned in the Introduction, in order to allow for efficient and simple system implementation, which in turn contributes to flexibility in building MASs.

Reflection [6] is used to realize an effective method for meta-level programming. The primary aims of implementing the reflection mechanism in RXF are (1) to enable an agent to manage his/her own states for problem solving, (2) to enable customization of the RXF language environment, (3) to clarify computation models by separating resource-management tasks from problem solving tasks, and (4) to enable the customization of computational states for an agent. In our approach to these aims, the agent ports are implemented for point (1). To implement point (2), we have created a meta-level execution mechanism, and achieving point (3) involves using agent ports and the meta-level execution mechanism. To achieve point (4), we designed a meta-agent that is used as an agent management system in the agent architecture. A reflective agent can dynamically adapt to its environment by using the reflection mechanism.

An agent port generates meta-level expressions that are representations of a base-level agent on a meta-level. For example, when an agent is evaluating a query "`a(X)`", a meta-level expression of query "`a(X)`" generated by an agent port for the agent is "`goal(a('X'))`". The quotations around X indicate that X is a base-level variable, not a meta-level variable.

```
(1) run_agent :-
      loop,
        event::in(Event,[wait]),
        do_attention(Event),
        fail.
(2) do_attention(message(F,T,M)) :-
      !,
      (message(F,T,M)
      ;base_message::put(message(F,T,M))),
      !,
      true.
(3) do_attention(Event) :-
      !,
      call(Event),
      !,
      true.
(4) query(X) :-
      base_eval::status = stop,!,
      base_eval::query(X).
(5) query(X) :-
      base_eval::thread_query(X).
```

Figure 2: AN EXAMPLE OF DESIGNING AN AGENT MANAGEMENT SYSTEM

2.3 IMPLEMENTING AN AGENT MANAGEMENT SYSTEM

In this section we show an example of RXF programming. Figure 2 shows an example for implementing an agent management system in RXF. The system manages the low-level processes of an agent. The low-level processes include message passing, interruptions, and performing tasks given by users. The program repeats two processes: (a) getting an event from an event port, and (b) handling the event properly. In process (a), the event port is a mechanism for getting events that are queries, messages, mouse clicks, etc. Process (b) handles an event that is obtained in the process (a). For example, if the agent management system obtains a message, the message is delivered into the message port. If the system obtains a query, the query is evaluated by a new thread.

In program (1) shown in Figure 2, the predicate "`run_agent`" is a program that repeats processes (a) and (b). The command "`event::in(Event,[wait])`" represents process (a), which is used for getting an event from the event-port "`event`" ("`[wait]`" is an instruction to block the process until an event is given). The predicate "`do_attention(Event)`" represents process (b) and gives an example of using the name manager. Specifically, in RXF, a command with an operator '`::`' ('`::`' is defined as an infix operator) is used for controlling the name manager.

In Figure 2, the predicates "`do_attention(message(F,T,M))`" indicated by (2) and "`do_attention(Event)`" indicated by (3) implement process (b). Program (2) processes messages, and program (3) processes events. If a message matches the name of an attention handler, program (2) evaluates that attention handler. If the evaluation of the attention handler fails, the command "`base_message::put(message(F,T,M))`" puts the message into the message port "`base_message`". Program (3) evaluates a goal that is obtained as an event.

In Figure 2, the predicates "`query(X)`" indicated by (4) and (5) evaluate events matching "`query(X)`". In program (4), the command "`base_eval::status = stop`" succeeds if the current state of the base-level interpreter is "`stop`". The state "`stop`" means that the interpreter evaluates nothing. Specifically, program (4) evaluates `X` by using "`base_eval::query(X)`" if the interpreter "`base_eval`" evaluates nothing. Program (5) evaluates the query if program (4) fails. Program (5) creates a new thread by using the command "`base_eval::thread_query(X)`", and lets the thread evaluate "`X`". Using the name manager, we can effectively design and program the low-level processes of an agent.

3 THE MULTI-AGENT TRACER

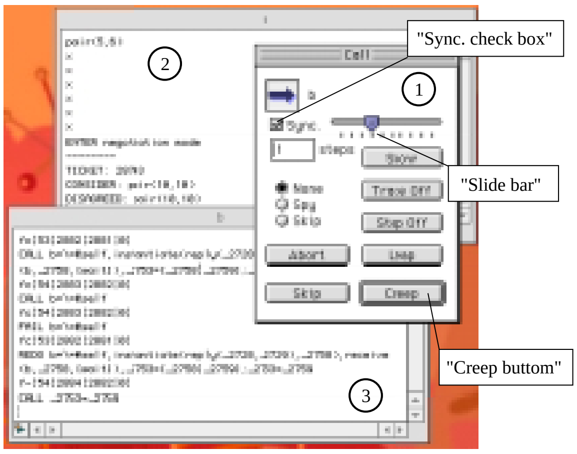
The multi-agent tracer is used for debugging MASs. A tracer for logic programming languages (e.g, Prolog) is a powerful tool for debugging programs in a single-threaded execution, but a simple tracer in a distributed environment is not as powerful as the single-threaded execution of logic programming languages.

RXF has a tracer based on the box model [17] to debug CLP programs. If a user uses the tracer to debug an agent in an MAS, the user needs to be aware that the agent being traced was suspended by the tracer but that other agents of the MAS are still running asynchronously. The asynchronous processing makes debugging difficult. We therefore needed a tracer that could delay the execution of all agents in an MAS while debugging occurs. The new RXF tracer can be used to approximately synchronize the execution of all agents in an MAS by delaying execution of the agents. We call this function the approximate synchronization facility.

3.1 THE INTERFACE

The tracer can delay the execution of agents in an MAS. Programmers can manually control the delay time in order to synchronize the agent programs are thereby effectively debug MASs.

Figure 3 is an illustrations of the graphical interface of the tracer. Figure 3 shows that two agents, "a" and "b," are running. Agent "a" is not being debugged. Agent "b" is now being debugged. There are three windows "a," "b," and "Call" in Figure 3. The dialog window labeled 1 shown in Figure 3 is a main interface of the tracer. Programmers can control the tracer through the dialog. The window labeled "2" and the



```
pair(5,6)
x
x
x
x
x
x
ENTER negotiation mode
----------
TICKET: 2070
CONSIDER: pair(10,10)
DISCARDED: pair(10,10)
```

```
[illegible]
```

Figure 3: THE ILLUSTRATIONS OF THE TRACER

window labeled "3" shown in Figure 3 are the interfaces of agents "a" and "b," respectively. We call each window an agent window. Agent window "a" displays the output of agent "a," while a agent window "b" displays a trace of agent "b" and the output of agent "b."

There are thirteen components in the dialog window that allow programmers to control the tracer. The components concerned with the approximate synchronization facility are the "Creep" button, the "Sync." check box, and the slide bar. The "Creep" button is for tracing entry into called goal. The button can be used to see what is happening, step by step. As soon as a programmer clicks the "Creep" button a new goal will be displayed in the agent window of a target agent. The slide bar can be used to manually control the delay time. The check box is the switch of the approximate synchronization facility. If the check box is on, the tracer delays the execution of nontarget agents.

3.2 EVALUATION

We believe that the tracer works well to debug MASs. We tested our system by tracing a multi-agent scheduling system we implemented. In multi-agent scheduling, agents who act autonomously in the network can schedule meetings on other individuals' behalf. The agents negotiate with each other by using a persuasion protocol [18] to reach an agreement.

The purpose of the test is to show that execution-speed ratios of the MAS with tracing and the MAS without tracing are similar. To compare the ratios, we measured the execution speeds while the tracing mechanism was active ("trace on") and inactive ("trace off"). The more similar the ratios are, the better the tracer is.

The test environment consisted of three computers connected with a 10 Mbps Ethernet. The computers were,an (a) Apple PowerMacintosh G3 233, (b) an Apple PowerMacintosh 7300/180, and (c) an Apple PowerMacintosh 7300/166. Each computer could communicate with the others using the network.

	A	B	C
trace off	1.00	0.603	0.554
trace on	1.00	0.597	0.511

Table 1: THE RATIO OF THE EXECUTION SPEEDS

Three agents, A, B, and C, belonged to the test MAS. A, B, and C were running on computers a, b, and c, respectively. A was a target agent; B and C were nontarget agents. We ran each test five times. First we examined the ratio of the agents in "trace off" mode. The average ratio was A : B : C = 1.00 : 0.603 : 0.554 (shown in table 1). Second we examined the ratio of the agents in "trace on" mode, and the average ratio was A : B : C = 1.00 : 0.597 : 0.511 (shown in table 1). Table 1 shows the result. This table indicates that we can consider the two ratios as nearly identical data. Therefore, we conclude that the tracer is practical enough to debug MASs.

4 DISCUSSIONS

We first describe related work on multi-agent system development.

The MadKit [19] is a multi-agent platform built on an organizational model. It provides general agent facilities (lifecycle management, message passing, distribution, ...) and allows high heterogeneity in agent architectures and communication languages.

The JAFMAS [20] is a Java-based framework for representing and developing cooperation knowledge and protocols in an MAS. The JAFMAS provides a generic methodology for developing speech-act-based MASs along with a set of classes to support implementing their agents in Java.

Fischer [21] has attempted to model autonomous resource-bound agents that interact with each other in dynamic multi-agent environments. InteRRaP implements a pragmatic belief desire-intention (BDI) architecture, where the agent's mental state is distributed over a set of layers. In RXF, functions of each layer of an agent are not defined as they are in InteRRaP. Because we believe that an optimal architecture for layers does not exist, in RXF programmers can decide upon and design architectures for specific kinds of problem-solving. The layers of InteRRaP cannot run in parallel. The layers of RXF can run in parallel based on the multi-thread mechanism.

Barbuceanu [22] has focused on developing "Agent Building Shell" that provides reusable languages and services for agent construction, relieving developers of the effort of building agent systems from scratch and guaranteeing that essential interoperation, communication, and cooperation services will always be there to support the application. The main difference between "Agent Building Shell" and RXF is that he latter is an agent architecture in which agents can be flexibly customized by meta-level programming based on a reflection.

We are not familiar with any debuggers for MASs. Instead, we describe research on debuggers for multi-thread programming.

Krinke [23] proposed a method for static slicing of threaded programs with interference. The method can analyze three dependence types: the control, data, and interference dependencies. Using Krinke's method, we can be aware of the dependencies on data and statements in programs, but this method is not suited for interactive debugging because the computation cost is high. On implementing interactive debuggers such as our system, we needed fast method, and our method is fast enough to allow programmers to interactively debug programs.

Stromme [24] developed a debugging and testing system. The software targeted by our testing technology is large-scale, distributed, concurrent software, with a significant real-time aspect, such as a digital switching system. Stromme's system can trace multi-threaded programs by using multiple tracers, and the tracers create a large amount of debugging information. Programmers cannot quickly understand the meaning of this much information. The debugging information produced by our system is little enough to understand instantly. We think that a single tracer is better than multiple tracers for interactive debugging of multi-threaded programs.

5 CONCLUSION

We have described a new programming environment for building reflective intelligent and autonomous agents. In order to realize an effective environment for building MASs, we have developed the RXF programming environment. RXF can be used for building flexible and extendable MASs. RXF is designed based on a constraint-logic programming language, and it has an effective mechanism of reflection to create autonomous agents. The reflection can be used to customize agents by themselves. The main purpose of this research is to create a programming environment for efficiently building multi-agent systems rather than implementing an agent-oriented language such as AGENT 0 [25]. Programmers implementing MASs need not only a development environment but also a debugger. In the past, programmers using a simple debugger for RXF have not been able to find bugs, but our new tracer resolves this problem, as it can control the execution speed of agents in an MAS. Our tests showed that the tracer can properly delay non-target agents. By using the tracer, we can effectively debug agents in an MAS.

References

[1] *Tadachika Ozono and Toramatsu Shintani, On Constraint Logic Programming Language RXF for Implementing Multiagent Systems, Trans.IPS.Japan*, 37(10), 1996, 1765–1772.

[2] *Tadachika Ozono and Toramatsu Shintani, Implementing a Reflective Mechanism in Constraint Logic Programming Language RXF, Trans.IPS.Japan*, 38(7), 1997, 1361–1369.

[3] *Tadachika Ozono and Toramatsu Shintani, On a Programming Environment for Building Reflective Agents, Proc. the IPSJ International Symposium on Information Systems and Technologies for Network Society*, 1997, 303–309.

[4] *Tadachika Ozono and Toramatsu Shintani: RXF : a Java-based Programming Environment for Building Multi-Agent System, PRICAI'98 Java-Based Intelligent Systems Workshop*, 1998, 51–60.

[5] *Y. Shoham, Agent-Oriented Programming, Artificial Intelligence*, 60(1), 1993, 51–92.

[6] *B. C. Smith, Reflection and Semantics in Lisp, Proc. 11th ACM Symposium on Principle of Programming Languages*, 1984, 23–35.

[7] *Joxan Jaffar and Jean-Louis Lassez, Constraint logic programming, Proc. 14th ACM Symposium on Principles of Programming Languagesm*, 1987, 111–119.

[8] *G. Agha, Concurrent Object-Oriented Programming, Comm. ACM*, 33(90), 1990, 125–141.

[9] *P. Maes, Issues in Computational Reflection, in Meta-Level Architecture and Reflection*, Elsevier Science, 1988, 21–35.

[10] *Tadachika Ozono and Toramatsu Shintani, The implementation of reflective constraint logic programming language RXF. Proc. 50th Annual Convention IPS Japan*, 2, 1995, 127–128.

[11] *Y. Lesperance, et.al, Foundations of a Logical Approach to Agent Programming, Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, Springer-Verlag, 1996, 331–346.

[12] *T. Mullen and M. P. Wellman, Some Issues in the Design of Market-Oriented Agents, Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, Springer-Verlag, 1996, 283–298.

[13] *Apple Computer, Inc., INSIDE MACINTOSH, Networking* (Addison-Wesley, 1994).

[14] *Charles E. McDowell and David P. Helmbold Debugging Concurrent Programs, ACM Computing Surveys*, 21(4), 1989, 593–622.

[15] *J. Gate, A debugger for concurrent programs, In Proceedings of Workshop on Parallel and Distributed Debugging*, ACM, 1988, 130–140.

[16] *Thom Fuhwirth, Alexander Herold, Volker Kuchenhoff, Thierry Le Provost: Pierre Lim, Eric Monfroy and Mark Wallace, Constraint Logic Programming, An Informal Introduction, technical report ECRC-93-5*, 1993.

[17] *W.F.Clocksin and C.S.Mellish, Programming in Prolog* (Springer-Verlag, 1981).

[18] *Takayuki Ito and Toramatsu Shintani, Persuasion among Agents : An Approach to Implementing a Group Decision Support System Based on Multi-Agent Negotiation, IJCAI-97*, 1997, 592–597.

[19] *Jacques Ferber and Olivier Gutknecht, A meta-model for the analysis and design of organization in multi-agent systems, ICMAS98*, 1998, 128–135.

[20] *Deepika Chauhan and Albert D. Baker, Developing Coherent Multiagent Systems using JAFMAS, ICMAS98*, 1998, 407–408.

[21] *Klaus Fischer, Jorg P. Muller and Markus Pischel, A Pragmatic BDI Architecture, Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, Springer-Verlog, 1996, 203–218.

[22] *Mihai Barbuceanu and Mark S. Fox, The Architecture of an Agent Building Shell, Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, Springer-Verlog, 1996, 235–250.

[23] *Jens Krinke, Static Slicing of Threaded Programs, ACM PASTE '98*,1998, 35–42.

[24] *Jon E. Stromme, Integrated Testing and Debugging of Concurrent Software Systems, INDC96*, 1996.

[25] *Y. Shoham, AGENT0: A simple agent language and its interpreter. AAAI-91*, 1991, 704–709.