Bridging the Gap Between AUML and Implementation Using IOM/T

Takou Doi[1], Nobukazu Yoshioka[2], Yasuyuki Tahara[2], and Shinichi Honiden[1,2]

[1] University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan
[2] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
{tdoi, nobukazu, tahara, honiden}@nii.ac.jp

Abstract. Multi-agent systems are attractive means for developing complex software systems. However, multi-agent systems themselves tend to be complex, and certain difficulties exist in developing them. One of the difficulties is the gap between design and implementation especially for interaction protocols. In this paper, we propose a new interaction protocol description language called IOM/T. Interaction protocols described using IOM/T have clear correspondence with AUML sequence diagrams and the description can be consolidated into a single unit of IOM/T code. Then, we show how the process of implementing Java-based agent-platform code from AUML design can be carried out, and how IOM/T effectively bridges the gap between design and implementation.

1 Introduction

Multi-agent systems are attractive means for developing complex software systems. However, multi-agent systems themselves tend to be complex, and certain difficulties exist in developing them. One of the difficulties is the gap between design and implementation [3], [1]. Especially the gap about interaction protocols is large. In Analysis and Design phase, developers use an agent-oriented method such as Gaia[16]. In this phase, interaction protocols are considered as important as agents themselves. In the Implementation phase, however, developers use object oriented languages such as Java, and implement interaction protocols as agent capabilities. As a result, the actual implementation of interaction protocols becomes dispersed, and the correspondence between the messages transition and the message reception becomes unclear. Yet, interaction protocols of multi-agent system are quite complex, thus leading to problems such as the following:

1. Understanding the interaction protocols from code is difficult.
2. Maintenance of interaction protocols is difficult.

In this paper, we propose a new interaction protocol description language called IOM/T(Interaction Oriented Model by Textural representation). Using this language, we can separate interaction protocol code from agent code. Furthermore, we can implement an interaction protocol in a single unit of code. We also describe the development process using IOM/T and tools. During the process, we analyze and design multi-agent system

R.H. Bordini et al. (Eds.): PROMAS 2004, LNAI 3346, pp. 147–162, 2005.

on the basis of the interaction protocols. Then, we use a tool to generate a skeleton code of IOM/T for each interaction protocols. We refine the code by adding information of the implementation. Finally, we use a preprocessor to convert the IOM/T codes and the agents' codes into the Java codes for a certain agent platform. IOM/T and this development process will bridge the gap between the design and the implementation.

Below, this paper is structured as follows. In section 2, we propose IOM/T. Then, in section 3, we show the interaction based development process of multi-agent system. In section 4, we evaluate IOM/T by comparing it to an existing agent-platform. Related work is discussed in section 5, and some conclusions are presented in section 6.

2 IOM/T: Interaction Oriented Model by Textural Representation

In this section we present IOM/T and describe its syntactic rules.

2.1 Concepts

We think one of the factors which make the development of multi-agent systems difficult is that we can not deal with interactions in Implementation as same as in Design. We need a representation which holds information about both design and implementation. Furthermore interaction protocols are generic rules for communicating and do not depend on agents mental states. In other words, we can separate interaction code from agent code. This simplifies the representations of the interactions. Therefore we designed IOM/T on the basis of the following concepts:

1. Code describing the interaction protocols should have clear correspondence with the design.
2. Developers should be able to easily understand the interaction protocol flow.
3. Separation of Concerns should be achieved.
4. Interaction protocols should be allowed to be represented with a single unit of code.
5. Notation is based on Java.

2.2 Protocol Example

In the remainder of this paper, we will exploit a simple example to describe our language and methodology. We define the Iterated Ping protocol using an AUML[7] sequence diagram[8] as shown in Fig.1. Two roles, Sender and Receiver, participate in this protocol. The Sender sends a "ping" message to the Receiver, and the Receiver replies to that message. This sequence is repeated as long as the Sender wants to. The protocol is described using IOM/T as shown in Fig.2. This paper also includes an example code for the English Auction. A.1 includes the protocol representation in AUML sequence diagrams and A.2 includes the corresponding IOM/T code.

2.3 Definition of Protocols

In IOM/T a protocol is represented as a structure using the keyword $protocol$. The protocol structure has a unique identifier, and consists of definitions of roles and a interaction flow. Fig.2 defines an interaction protocol named PingProtocol.

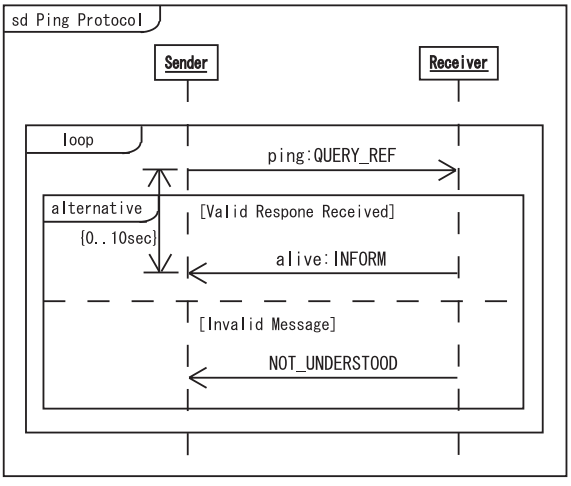


Fig. 1. Iterated Ping protocol represented using AUML sequence diagram

```
1: public protocol PingProtocol {
2:   player Sender {
3:     AID getTarget();
4:     isContinue();
5:     void knowAsDead();
6:     Date sendTime;
7:   }
8:   player Receiver {
9:   }
10:  interaction {
11:    while (Sender.isContinue()) {
12:      player Sender {
13:        ACLMessage ping = new ACLMessage();
14:        ping.setReceiver(getTarget());
15:        ping.setContent("(ping)");
16:        ping.setPerformative("QUERY_REF");
17:        sendAsync(ping);
18:        sendTime =
19:          Calender.getInstance().getTime();
20:      }
21:      player Receiver {
22:        ACLMessage ping = recvBlock();
23:        ACLMessage res = ping.createResponse();
24:        if (ping.getContent.equals("(ping)")) {
25:          res.setContent("(alive)");
26:          res.setPerformative("INFORM");
27:        } else {
28:          res.setContent(ping.getContent());
29:          res.setPerformative("NOT_UNDERSTOOD");
30:        }
31:        sendAsync(res);
32:      }
33:      player Sender() {
34:        while (true) {
35:          ACLMessage res =
36:            recvNonBlock();
37:          if (res != null &&
38:              res.getPerformative().equals("INFORM") &&
39:              res.getContent().equals("(alive)")) {
40:            break;
41:          }
42:          Date current =
43:            Calender.getInstance().getTime();
44:          if (current.getTime()
45:                  - sendTime.getTime()
46:                      >= 10 * 1000)) {
47:          knowAsDead();
48:          break;
49:        }
50:      }
51:    }
52:    player Sender {
53:      ACLMessage msg = new ACLMessage();
54:      msg.setReceiver(getTarget());
55:      msg.setContent("(end-of-loop)");
56:      msg.setPerformative("INFORM");
57:      sendAsync(msg);
58:    }
59:  }
60: }
61:
```

Fig. 2. Iterated Ping protocol described in IOM/T

The syntactic rule of protocol structures is as follows:

$$
\begin{array}{ll}
ProtocolDefinition & ::= \text{protocol } Identifier\ ProtocolBody \\
ProtocolBody & ::= \{\ ProtocolBodyDefinition\ \} \\
ProtocolBodyDefinition & ::= PlayerDefinitions\ InteractionDefinition \\
PlayerDefinitions & ::= PlayerDefinition \\
 & \quad |\ \ PlayerDefinitions\ PlayerDefinition
\end{array}
$$

2.4 Definitions of Roles

A role is defined as a structure using the keyword $player$. The symbol $*$ after the keyword $player$ represents the existence of several agents playing the role. A player structure has a unique identifier, and consists of definitions of sub-protocol, definitions of role functionality, and definitions of role information. When we have to deal with a large interaction protocol, we divide it into some sub-protocol and represent the whole interaction protocol by using them. Definition of sub-protocol specifies which sub-interaction the role participates in and what role it plays in the sub-protocol using the keyword $playing$. Definition of role functionality is specified using Java method definition notation. Definition of role information is specified using Java field definition notation. In Fig.2 the two roles, Sender and Receiver, are defined on lines 2 - 9. The code shows that the Sender has three functionalities and one variable for information. The functionality $getTarget()$ determines the target agent. The functionality $isContinue()$ is used to determine the end of loop. The functionality $knowAsDead()$ notifies the Sender agent that the target agent did not reply to the message. The information $sendTime$ is used to hold the time the Sender sent the message. On the other hand, the Receiver does not have any functionality and information.

The syntactic rule of player structures is as follows:

$$
\begin{array}{ll}
PlayerDefinition & ::= \text{player } *_{opt}\ Identifier\ PlayerBody \\
PlayerBody & ::= \{\ PlayerBodyDefinitions_{opt}\ \} \\
PlayerBodyDefinitions & ::= PlayerBodyDefinition \\
 & \quad |\ \ PlayerBodyDefinitions\ PlayerBodyDefinition \\
PlayerBodyDefinition & ::= UseProtocolDefinition \\
 & \quad |\ \ AgentFunctionDefinition \\
 & \quad |\ \ InformationDefinition \\
UseProtocolDefinition & ::= \text{playing } ProtocolName.PlayerName; \\
AgentFunctionDefinition & ::= ReturnType\ FunctionIdentifier(Arguments_{opt}); \\
InformationDefinition & ::= InformationType\ InfomationIdentifier;
\end{array}
$$

2.5 Definition of Interaction Protocol Flows

An interaction protocol flow is defined as an structure using the keyword $interaction$. An interaction structure consists of a combination of sequential flow and repeated flow. The sequence of the role actions represents the sequential flow, and "while" structure represents the repeated flow as in Java notation. There are restrictions on this structure because of the feasibility for parallel process. One is that the "while" condition predicate must describe a single functionality of a single role. The other is that the role which

```
public class ACLMessge {
    public void setSender(AID sender);
    public AID getSender(AID sender);
     ...
    public void setPerformative(String performative);
    public String getPerformative();
     ...
    ACLMessage createResponse();
}
```

Fig. 3. Outline of the ACLMessage class

determines the end of loop must send notification messages at the end of loop to other roles. A role action is defined as a block labeled with the role identifier. This block contains a role action defined using Java and some language enhancement, namely expressions for using role functionalities and information, expressions for dealing with FIPA-ACL message, and expressions for controlling protocols. In Fig.2 the Sender's action is described on lines 12 - 20, the Receiver's action on lines 21 - 32 and the Sender's action on lines 33 - 49 represent a sequential flow, and this sequential flow is structured as a repeated flow on lines 11 - 50. The Sender sends notification for the end of loop in the Sender's action on lines 52 - 58.

The syntactic rules for defining an interaction flow are as follows:

$InteractionDefinition$ ::= interaction $InteractionBody$
$InteractionBody$ ::= { $InteractionBlocks$ }
$InteractionBlocks$ ::= $InteractionBlock$
| $InteractionBlocks$ $InteractionBlock$
| while($WhilePredicate$) { $InteractionBlocks$ }
$InteractionBlock$ ::= player $PlayerIdentifier$ { $ActionBlock$ }
$WhilePredicate$ ::= $PlayerIdentifier.FunctionIdentifier(Args_{opt})$
| (PlayerIdentifier must specify a single role)

Expressions for Using Role Functionalities and Information. The use of a role functionality is represented as a method invocation in Java, and the use of a role information is represented as a field access in Java. In Fig.2 the functionality of Sender is used on line 11, 14 and 47. The information of Sender is used on line 18 and 45.

Expressions for Dealing with FIPA-ACL Message. In order to deal with FIPA-ACL messages, the following classes and functions are used:

1. AID : This class represents the agent identifier.
2. ACLMessage : This class represents FIPA-ACL messages and its outline is laid out on Fig3. It has getter/setter methods to handle the terms specified in the messages. This class also has a method to create a reply message as an instance of ACLMessage class.
3. send/receive functions : These functions are used to send or receive an instances of ACLMessage class. $sendSync$ function represents synchronous transmission, and $sendAsync$ function represents asynchronous transmission. $recvBlock$ function represents a blocking reception, and $recvNonBlock$ function represents a non-blocking reception.

```
public class MyAgent extends Agent {
    playing PingProtocol.Sender {
        getTarget = getPingAgent;
        isContinue = isPingContine;
        knowAsDead = knowAsDead;
    }
    AID getPingAgent() { ... }
    boolean isPingContinue() { ... }
    void knowAsDead() { ... }
}
```

Fig. 4. Agent code example using the Iterated Ping protocol

Expressions for Controlling Protocols. The following functions are used to control the protocols:

1. Begin sub-protocol : The function $beginProtocol$ is used to begin a sub-protocol. The next sentence means the beginning of a sub-protocol specified by the identifier $ProtocolIdentifier$ as the role specified by the identifier $RoleIdentifier$.

$$beginProtocol(ProtocolIdentifier.RoleIdentifier)$$

2. Terminate current protocol : The function $terminateProtocol$ is used to terminate the current protocol.

2.6 Using Interaction Protocols

Above, we show the description of interactions. We show the description of agents with a focus on the use of interaction protocols. The description of the other aspects are out of the scope of this paper. In order to use interaction protocols, functionality mappings are defined in the agent code. A functionality mapping represents how methods of each role functionality are implemented using the keyword $playing$. The function $beginProtocol$ is used to begin a sub-protocol. Fig.4 shows an example of using the Iterated Ping protocol.

The syntactic rules for mapping functionality are as follows:

$$
\begin{array}{ll}
PlayingDefinition & ::= \text{playing } ProtocolIdentifier.PlayerIdentifier \\
 & \quad | \quad PlayingDefinitionBody \\
PlayingDefinitionBody & ::= \{FunctionMappings_{opt}\} \\
FunctionMappings & ::= FunctionMapping \\
 & \quad | \quad FunctionMappings\ FunctionMapping \\
FunctionMapping & ::= NormalFunctionMapping \\
 & \quad | \quad ProtocollFunctionMapping \\
NormalFunctionMapping & ::= FunctionIdentifier\ =\ MethodIdentifier \\
ProtocolFunctionMapping & ::= ProtocolSpecifiers.FunctionIdentifier \\
 & \quad | \quad =\ MethodIdentifier \\
ProtocolSpecifiers & ::= ProtocolSpecifier \\
 & \quad | \quad ProtocolSpecifiers\ ProtocolSpecifier \\
ProtocolSpecifier & ::= ProtocolIdentifier.PlayerIdentifier
\end{array}
$$

2.7 AUML and IOM/T

The current version of IOM/T can not fully represent all the interactions that can be described with AUML sequence diagrams. An AUML sequence diagram corresponds to an protocol structure and a time-line corresponds to a role. However, IOM/T can not represent every kind of CombinedFragment[8]. Only CombinedFragments whose interaction operators are “Loop” can be represented by “while” structure. We have to describe the meaning of the other interaction operators in role actions. Although we can extend the language adding new structures to deal with these interaction operators, it is hard to generate codes which them such as alternative, option, parallel and so forth. This extension is included in our future work. In fact, we aim at providing the semantics and proving the equivalence with AUML sequence diagrams.

3 Development Process

In this section we provide an overview of our development process based on interaction protocols. The process is roughly divided into two phases. In the Analysis and Design phase, developers extract requirements and functionalities, and design the interaction protocols and agents in the system. Then, in the Implementation phase, developers implement the interaction protocols and agents on the basis of on the design decisions in the previous phase. This phase includes testing. We show the development process focusing on the Implementation phase due to limitations of space.

3.1 Analysis and Design

The first phase is the Analysis and Design phase. In this phase we extract the system requirements, and decide how these requirements are realized.

The products of this phase are the design of interaction protocols and agents in the system. Each interaction protocol consists of roles and message sequence. A role contains functionalities to fulfill the interaction. A message sequence is represented using an AUML sequence diagram. Each agent is endowed with roles that it has to play.

3.2 Implementation

The second phase is the Implementation. We implement interactions using IOM/T and agents using APIs that depend on the agent-platform. In this paper we concentrate on the implementation of interactions.

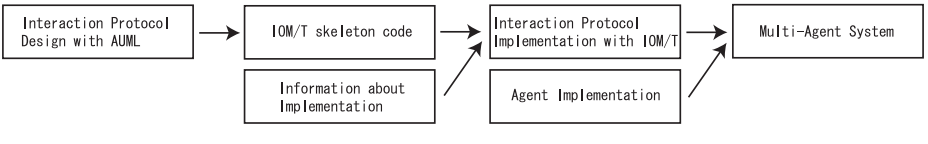


Fig. 5. Overview of Implementation Process

In the Analysis and Design phase, we acquired the interaction protocols and represented them using AUML sequence diagrams. Here, we show the process of implementation using AUML sequence diagrams. The overview of this process is as follows: First, we use tools to automatically generate skeleton codes of IOM/T from AUML sequence diagrams. Next, we add detail information of the implementation to the code. Then we implement agents and add the mappings of the roles' functionalities. Finally, we use a preprocessor to generate Java code from these codes for the target agent-platform. The overview of the Implementation phase is shown in Fig.5.

AUML to IOM/T. Interaction protocols described in IOM/T have clear correspondence with AUML sequence diagrams. For example Fig.6 shows the correspondence in the Ping protocol case. Thus, we can convert AUML sequence diagrams into skeleton codes of IOM/T based on the following simple algorithms:

1. Each time-line is extracted as a role.
2. A single role action is extracted from a set of zero or more continuous message reception and one or more continuous message transmission within the same time-line.
3. Loop structures are extracted as "while" structures.

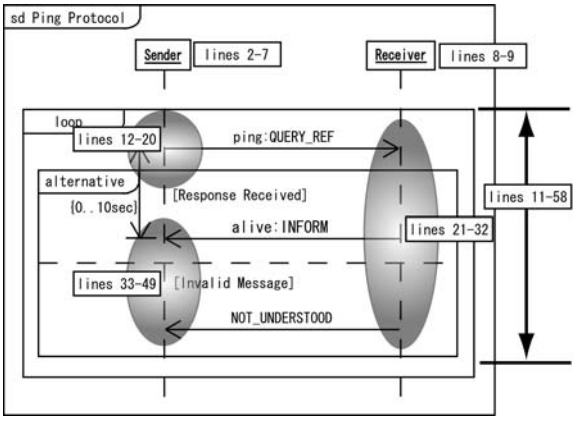


Fig. 6. Correspondence of AUML sequence diagrams to IOM/T

Implementation in IOM/T. The generated skeleton codes of the interaction protocol do not have sufficient information for the implementation. For instance, constraints in AUML sequence diagrams may not contain complete information for execution. Furthermore, AUML sequence diagrams represent the order of messages, but they may not represent the contents of messages. They also lack the information about how the contents are prepared. We have to make a decision on the precise meaning of the constrains and the contents of the messages. So, we add detailed information of the implementation

in the code. We refine roles by adding roles' functionalities and information. We refine the flow of the interaction protocol by adding the conditions of "while" structure. Then, we also refine the roles' actions by using Java notation.

In the example of the Ping protocol, we add three functionalities and one variable to the Sender role as shown on lines 3-6 in Fig.2. We add the condition of the "while" structure on line 11. We refine the roles' actions as shown on lines 24-30 and 43-48.

IOM/T to Agent-Platform Specific Java Codes While the interaction protocol is implemented using IOM/T, role functionalities i.e. agent functionalities depend on the specific agent-platform. Codes is merged and converted into target agent-platform Java codes as follows:

1. Generating a FSM model
   A FSM model is generated for each role assuming the role action as a state. State transitions occur on the basis of the order of the role actions and the "while" structures. Furthermore, for roles that do not determine an end of loop, an action is added for that purpose.
2. Implementing a FSM model on the target agent-platform.
   The generated FSM model is converted into the agent-platform specific Java codes.
3. Implementing Role Actions
   Portions implemented in Java are achieved directly. Only special extensions must be converted into agent-platform specific forms. For each role, an interface for accessing the role functionalities is generated. The using of role functionalities is converted into the method invocation through the interface. For each agent, the preprocessor generates helper class which implement interfaces rolls that it play. The helper class delegates role functionalities to the agent class according to the functionality mappings. The extension for FIPA ACL is converted into agent-platform specific representations. The beginning of a protocol is also converted into agent-platform specific representations.

Testing. We can apply unit test to the interaction protocols and the agents. The agent unit test is similar to one for objects. We prepare the test cases for verifying the functionalities of the roles that the agents play. We prepare the test case agents for the unit test for interaction protocols. For each role, we implement test case agents who have the role functionalities. Then, we prepare the test cases that contain the combination of the test case agents and the corresponding result of the interaction. The traditional unit test can check whether each agent is implemented as expected. However, it seems to be hard to check whether the agents can communicate with each other as expected. The unit test for interaction using IOM/T will support this verification. Furthermore, providing the finite ranges of values of the role functionalities, we can generate all possible test case agents. So we can verify the invariants of the interaction automatically.

4 Evaluation

In this section, we evaluate the effectiveness of IOM/T comparing an interaction protocol code described using IOM/T with the same protocol described using JADE[5], a Java based agent-platform. In JADE, interaction protocols are described in the classes which are inherited from the Behaviour class. Until the $done()$ method returns true, the $action()$ method is executed repeatedly. Fig.7 shows the Ping protocol described using JADE.

Firstly, we compare the development where the skeleton code of JADE are directly generated from AUML sequence diagrams to the development where we use IOM/T as an intermediate language. As Huget described in [9], the skeleton code of JADE can be generated directly. We think we have to modify the generated codes by hand in order to

```
 1: class PingSenderBehaviour extends Behaviour {
 2:
 3:   private boolean finished = false;
 4:   private int state = 0;
 5:
 6:   public PingSenderBehaviour(Agent a) {
 7:     super(a);
 8:   }
 9:   public void action() {
10:     switch (state) {
11:     case 0: {
12:       ACLMessage msg = new ACLMessage();
13:       msg.setPerformative(ACLMessage.QUERY_REF);
14:       msg.setReciver(
15:         ((PingSender)myAgent).getTarget());
16:       msg.setContent("(ping)");
17:       myAgent.send(msg);
18:       state = 1;
19:       break;
20:     }
21:     case 1: {
22:       ACLMessage msg =
23:           myAgent.blockingReceive(10 * 1000);
24:       if (msg == null) {
25:         ((PingSender)myAgent).knowAsDead();
26:       } else {
27:         String content = msg.getContent();
28:         if (msg.getPerformative()
29:               != ACLMessage.INFORM ||
30:           !content.equals("(alive)") {
31:           ((PingSender)myAgent).knowAsDead();
32:         }
33:       }
34:       state = 2;
35:       break;
36:     }
37:     case 2: {
38:       if (((PingSender)myAgent).isContinue()) {
39:         state = 0;
40:       } else {
41:         finished = true;
42:       }
43:         break;
44:       }
45:     }
46:   }
47:
48:   public boolean done() {
49:     return finished;
50:   }
51: }
52:
53: class PingReceiverBehaviour extends Behaviour {
54:
55:   private boolean finished = false;
56:
57:   public PingReceiverBehaviour(Agent a) {
58:     super(a);
59:   }
60:
61:   public void action() {
62:     ACLMessage  msg = myAgent.blockingReceive();
63:     if(msg != null){
64:       ACLMessage reply = msg.createReply();
65:       if(msg.getPerformative()
66:            == ACLMessage.QUERY_REF){
67:         String content = msg.getContent();
68:         if (content != null &&
69:             content.equals("(ping)")){
70:           reply.setPerformative(
71:                             ACLMessage.INFORM);
72:           reply.setContent("(alive)");
73:         } else {
74:           reply.setPerformative(
75:             ACLMessage.NOT_UNDERSTOOD);
76:           reply.setContent(msg.toString());
77:         }
78:       } else {
79:         reply.setPerformative(
80:                     ACLMessage.NOT_UNDERSTOOD);
81:         reply.setContent(msg.toString());
82:       }
83:       myAgent.send(reply);
84:     }
85:   }
86:
87:   public boolean done() {
88:     return finished;
89:   }
90: }
91:
```

Fig. 7. Iterated Ping protocol represented in JADE

represent the information which is showed as notes or constraints in AUML sequence diagrams in either case, although we tend to cause a mistake in this operation. For example, in Fig.1, there is a constrain, "[Valid Response Received]". In JADE code, this constraint is represented on lines 27-30 in Fig. 7, where we check the content of ACL message. The creation of this message is represented on lines 70-72. Although these two parts have strong relationship, they completely exist in separate code. On the other hand, in IOM/T, the constraint is represented on lines 37-39 in Fig. 2, and the creation of message is represented on lines 25-26. In this case, these two parts exist in a single unit of code. Furthermore, the role actions that include these parts are closed by. Therefore, it seems to be better to modify the skeleton codes of IOM/T rather than the skeleton codes of JADE.

Next, we consider the *understandability* . If we compare the IOM/T code with the JADE code, there are two significant differences. One difference is the code scattering for an interaction protocol. If we use JADE, we describe a code of the interaction protocol for each role. The code contains the only own part. Then, the codes for interaction protocol dispersed. If we use IOM/T to describe, the interaction protocol is described in a single unit of code. As a result, the correspondence between the message transmission and the message reception is clear. In Fig.7, the correspondence of line 17 to line 64 can easily be confused. However, In Fig.2, the same correspondence is represented on line 17 and 22. It is easy to understand this correspondence. The other difference is the representation of the interaction protocol flow. In the JADE code, we have to realize state transitions by assuming integers as states since Java has no notation for representing state transition. We can describe complex state transitions with this method. However it is not easy to understand the interaction protocol flow from the code, since we have to understand each state and each transition. Especially, loops are not clear since they are represented as recursive structures as shown on line 39. On the other hand, in the IOM/T code, the flow of the interaction protocol is represented in natural manner. Each role action corresponds to a state. The state transitions are represented on the basis of the order of the occurrences of the role actions, and the loops are represented explicitly using "while" structures. So we can intuitively understand the flow. Indeed we can represent more complex state transitions with the former method, but we mostly do not need to describe very complex state transitions. In this paper, we do not show the details of Analysis and Design phase. We think we can get the designs of interactions that are relatively simple through the phase. We can build up the system by using sub-interactions. So, our current achievement is to represent interaction protocols based on relatively simple AUML sequence diagrams.

IOM/T also improves the *maintainability* . IOM/T enhances the understandability of the interaction protocol as mentioned above. So, when we have to commit some changes, we can easily understand where and how we have to change. For example, suppose there was a change such as the "$alive$" message is replaced with the "$pong$" message in the Ping protocol. In Fig.7 the changes will be applied to line 30 in the $PingSenderBehaviour$ class and to line 72 in the $PingReceiverBehaviour$ class. These changes are not obvious. On the other hand, the same modification is easily achieved in the IOM/T code. We will change the lines 25 and 39 in Fig.2. Since the role actions that contain these lines are close by, the correspondence of the code is obvious.

In other words, since our development process holds *traceability* , the maintenabilty is improved. An AUML sequence diagram corresponds to a single unit of IOM/T code. A message in the diagram corresponds to role actions which are closed by. Then, we can found out the corresponding part of the code from the changes in the design model. So, we can deal with the changes easily.

5 Related Work

So far, several interaction protocol description languages such as AgenTalk[12], Q[11], COOL[2], COSY[4], micro-protocol[15] have been proposed. Each of these languages has specific interesting features. Micro-protocol is a language designed to improve reusability. This language is not considered here since we mainly consider languages for implementation. AgenTalk, COOL, COSY and Q are languages for implementation. However, with these languages, an interaction protocol is dispersed and represented within the scenario of each agent. Our aim is to bridge the gap between the AUML design and the implementation, by treating the interaction protocol as an independent module in the implementation.

In the area of interaction protocols, several results have been achieved on the basis of the formal methods. Huget have provided the methods for verifying AUML sequence diagrams by using Spin[10]. Paurobally have proposed diagrams as counterpart of formal methods in order to bridge the gap between formal specification and intuitive development[13]. These endeavors have relationship with our research. However, in this paper, we proposed a novel language and implementation process.

There are several methods for generating skeleton code from graphical representations. Huget shows some elements for generating code from AUML sequence diagrams[9]. Tahara provides the IPEditor[14] which generates skeleton codes of agents from AUML based graphical notation. Dinkloh has developed tools for generating skeleton codes of Jade from AUML sequence diagrams[6]. These works provide methods for generating object oriented programming code from design. Our approach deals with interaction protocols as it is in the Implementation phase.

6 Conclusion

In this paper, we proposed a new interaction protocol description language, IOM/T. We also showed how the implementations are conducted based on the AUML sequence diagrams. By applying this approach, we will be able to bridge the gap between AUML and implementation. Moreover, the representation in IOM/T allows easy understanding of interaction protocol flows and improves maintainability.

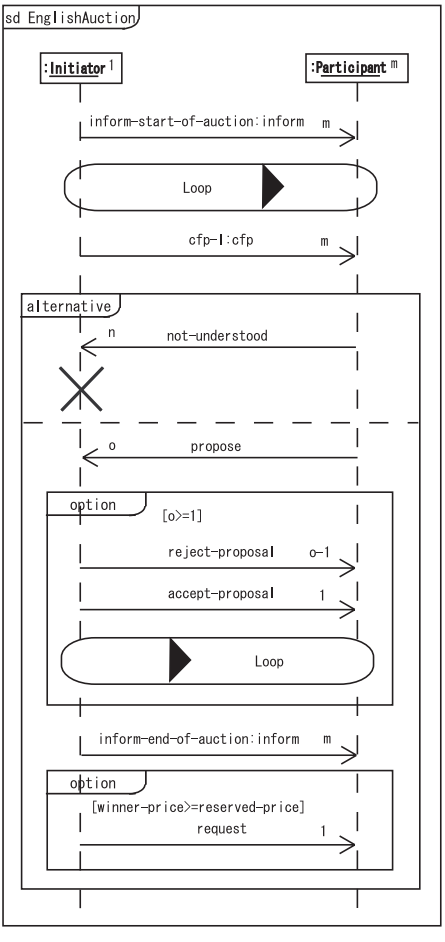
This work is currently in progress. We are developing an Integrated Development Environment for this approach, and verifying the validity of this approach through development examples. We will also investigate the inclination of IOM/T thruogh the development examples. In addition we are considering the needs for a formal semantics of IOM/T. Finally we plan to extend IOM/T to make IOM/T equivalent to AUML sequence diagrams.

References

1. Amal El Fallah-Seghrouchni Alexandru Suna. A mobile agents platform; architecture, mobility and security elements. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop*, 2004.
2. M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.
3. Lars Braubach, Alexander Pokahr, Winfried Lamersdorf, and Daniel Moldt. Goal representation for bdi agent systems. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop*, 2004.
4. B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic, configurable, cooperation protocols for multi-agent systems. *Lecture Notes in Computer Science*, 957:157–, 1995.
5. CSELT. JADE. `http://sharon.cselt.it/projects/jade/`.
6. M. Dinkloh and J. Nimis. A tool for integrated design and implementation of conversations in multi-agent systems. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2003 Workshop*, 2003.
7. FIPA. Agent UML. `http://www.auml.org/`.
8. FIPA. FIPA Modeling: Interaction Diagrams. `http://www.auml.org/auml/documents/ID-03-07-02.pdf`.
9. Marc-Philippe Huget. Generating code for Agent UML sequence diagrams. Technical report, University of Liverpool Department of Computer Science, 2002.
10. Marc-Philippe Huget. Model checking Agent UML protocol diagrams. Technical report, the department of computer science, University of Liverpool., 2002.
11. Toru Ishida. Q: A scenario description language for interactive agents. *IEEE Computer*, 2002.
12. K. Kuwabara, T. Ishida, and N. Osato. Agentalk: Describing multiagent coordination protocols with inheritance. 1995.
13. Nicholas R. Jennings Shamimabi Paurobally. Developing agent interaction protocols using graphical and logical methodologies. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2003 Workshop*, 2003.
14. Y. Tahara, A. Ohsuga, and S. Honiden. Mobile agent security with the IPEditor development tool and the Mobile UNITY language. In *Proc. of Agents 2001*, pages 656–662. ACM Press, 2001.
15. B. Vitteau and M.-P. Huget. Modularity in interaction protocols. In *Agent communication languages and conversation policies AAMAS 2003 Workshop*, 2003.
16. Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

A English Auction

A.1 English Auction Protocol Represented in AUML Sequence Diagram



A.2 EnglishAuction Protocol Described in IOM/T

```
protocol EnglishAuction {

  player Initiator {
    List getParticipants();
    boolean isEndAuction();
    String getCurrentPrice();
    ACLMessage selectBids(List bids);
    AID getWinner();
    List participants;
    Date cfpTime;
  }
```

```
player * Participant {
  void knowBeginAuction();
  boolean isBid(String item, String current);
  boolean
    isUnderstandable(String item, String current);
  String getBidPrice(String item, String current);
  void knowPrice(String price);
  void knowAccept();
  void knowReject();
  void knowRequest(String req);
  boolean isBid;
  boolean isAccept;
}

interaction {
  player Initiator {
    participants = getParticipants();
    ACLMessage msg = new ACLMessage();
    msg.setPerformative("INFORM");
    msg.setContent("(inform-start-of-auction)");
    msg.setReceiver(participants);
    sendAsync(msg);
  }

  player Participant {
    ACLMessage msg = recvBlock();
    knowBeginAuction();
  }

  while (!Initiator.isEndAuction()) {

    player Initiator {
      ACLMessage msg = new ACLMessage();
      msg.setPerformative("CFP");
      msg.setContent(getItem() + ":" + getCurrentPrice());
      msg.setReceiver(participants);
      sendAsync(msg);
      cfpTime = Calender.getInstance().getTime();
    }

    player Participant {
      ACLMessage cfp = recvBlock();
      String[] str = msg.getContent().split(":", 2);
      String item = str[0];
      String current = str[1];
      isAccept = false;
      if (isUnderstandable(item, current)) {
        ACLMessage res = cfp.createResponse();
        res.setPerformative("NOT_UNDERSTOOD");
        res.setContent(cfp.getContent());
        sendAsync(res);
        terminateProtocol();
      }
      isBid = isBid(item, current);
      if (isBid) {
        String price = getBidPrice(item, current);
        ACLMessage res = cfp.createResponse();
        res.setPerformative("PROPOSE");
        res.setContent(price);
        sendAsync(res);
      }
    }

    player Initiator {
      List bids = new LinkedList();
      Calender cal = Calender.getInstance();
      while (cal.getTime().getTime() - cfpTime.getTime() < 10 * 1000) {
```

```
        ACLMessage bid = recvNonBlock();
        if (bid.getPerformative().equals("NOT_UNDERSTOOD")) {
          participants.remove(bid.getSender());
        } else if (bid.getPerfromative().equals("PROPOSE")) {
          bids.add(bid);
        }
      }
      ACLMessage accept = selectBids(bids);
      if (accept != null) {
        Iterator itr = bids.iterator();
        while (itr.hasNext()) {
          ACLMessage bid = (ACLMessage)itr.next();
          ACLMessage msg = bid.createResponse();
          msg.setContent(bid.getContent());
          if (accept == bid) {
            msg.setPerformative("ACCEPT_PROPOSAL");
            sendAsync(msg);
          } else {
            msg.setPerformative("REJECT_PROPOSAL");
            sendAsync(msg);
          }
        }
      }
    }

    player Participant {
      if (isBid) {
        ACLMessage msg = recvBlock();
        if (msg.getPerformative().equals("ACCEPT_PROPOSAL")) {
          knowAccept();
          isAccept = true;
        } else {
          knowReject();
        }
      }
    }

  }

  player Initiator {
    ACLMessage msg = new ACLMessage();
    msg.setPerformative("INFORM");
    msg.setReceiver(participants);
    msg.setContent(getCurrentPrice());
    sendAsync(msg);
    ACLMessage msg = new ACLMessage();
    msg.setPerformative("REQUEST");
    msg.setReceiver(getWinner);
    msg.setContent(getCurrentPrice());
    sendAsync(msg);
  }

  player Participants {
    ACLMessage msg = recvBlock();
    String price = msg.getContent();
    knowPrice(price);
    if (isAccept) {
      ACLMessage req = recvBlock();
      knowRequest(req.getContent);
    }
  }

 }
}
```