IOM/T: An Interaction Description Language for Multi-Agent Systems

Takuo DOI
University of Tokyo
7-3-1, Hongo, Bunkyo-ku
Tokyo, Japan
tdoi@nii.ac.jp

Yasuyuki TAHARA
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo, Japan
tahara@nii.ac.jp

Shinichi HONIDEN
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo, Japan
honiden@nii.ac.jp

ABSTRACT

A multi-agent system is a useful approach for the complex systems. One of the important concepts of multi-agent systems is cooperativeness, or interactions. However, existing languages for implementing interactions lack expressiveness. This causes gaps between design and implementation. This paper analyzes language functionalities for implementing interactions. Furthermore, a new interaction description language IOM/T is proposed based on the findings. Interaction would become easy to implement based on design using IOM/T.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Multi-Agent System, Interaction, AUML

1. INTRODUCTION

In recent years, the speed of informational circulation is increasing due to the expansion of networks. Software has to deal with larger and more complex data and has to manage changes of an environment. This suggest the need for software that possesses human features such as autonomy and cooperativeness, since it is difficult for a human to check everything. A multi-agent system is one solution to this approach. One of the most important concepts of multi-agent systems is interactions. However, existing languages for implementing interactions lack expressiveness. This causes gaps to occur between design and implementation.

AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.

We consider the design and implementation in the development of multi-agent systems giving attention to interaction.

In the design phase, we will use AUML sequence diagrams [12] to describe interaction since it is a standard representation for interaction developed by FIPA [11]. Furthermore, AUML sequence diagrams are based on UML which is mainstream in current software development and many developers are familiar with it. The design of interactions includes the following:

- Message sequence
- Constraints on messages

In the implementation phase, there are some existing languages such as AgenTalk[15], COOL[1], COSY[3] and Q[14]. We can use these to describe interactions. We can also use Java if we use Java-based agent platforms such as FIPA-OS[9] and JADE[4]. The implementation of interactions includes the following:

- Precise message format and the method for handling it
- Method for realizing constraints on messages

It may be possible to add information regarding implementation as a note to a design created with AUML sequence diagrams. However, it is difficult to represent all the required information that should be included in implementation in design since the description capability of design is limitted. We have to design message sequences and implement interactions based on the result of design in order to develop multi-agent systems. Existing languages lack the ability to express message sequences. Implementation in these languages lose the information that is clear in design phase. Thus, the gap between design and implementation becomes larger.

We proposed an interaction description language named IOM/T (Interaction Oriented Model by Textual representation)[7]. This language has correspondences with AUML sequence diagrams and enable us to implement interactions based on AUML sequence diagrams easily. However, the language lacks the ability to represent all the interaction operator. We modify this language adding some notations. In this paper, we show the new version of IOM/T and how IOM/T corresponds to AUML sequence diagrams.

Below, this paper is structured as follows. In section 2, we consider the problem of implementing interaction with existing languages. Then in section 3, we provide a specification of IOM/T and in section 4, we verify the correspondence between IOM/T and the AUML sequence diagrams using π-calculus. In section 5, we evaluate IOM/T by comparing it with related work and some conclusion are presented in section 6.

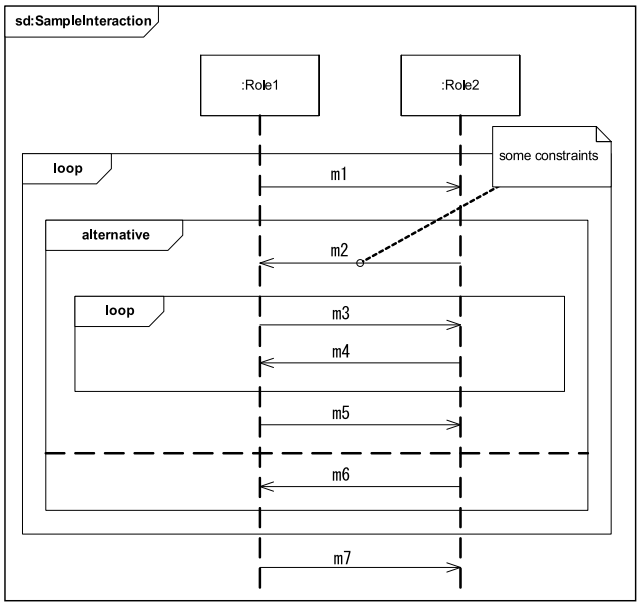


Figure 1: An example AUML sequence diagram

```
 1  public class Role1Behaviour
 2    extends jade.core.behaviours.Behaviour {
 3    int state_ = 0;
 4    boolean isTerminate_ = true;
 5    public void action() {
 6      switch (state_) {
 7      case 0:
 8        if (/* is first loop continue? */) {
 9          state_ = 1;
10        } else {
11          state_ = 7;
12        }
13        break;
14      case 1:
15        /* send message m1 */
16        state_ = 2;
17        break;
18      case 2:
19        /* recv message m2 or m6 */
20        if (/* is first case? */) {
21          state_ = 3;
22        } else {
23          state_ = 6;
24        }
25        break;
26      case 3:
27        if (/* is second loop continue? */) {
28          state_ = 4;
29        } else {
30          state_ = 5;
31        }
32        break;
33      case 4:
34        /* send message m3 */
35
36        // recv message m4
37        ACLMessage msg = myAgent.receive();
38        Object content = msg.getContent();
39        /* handle the content */
40
41        state_ = 3;
42        break;
43      case 5:
44        /* send message m5 */
45        break;
46      case 6:
47        state_ = 0;
48        break;
49      case 7:
50        /* send message m7 */
51        isTerminate_ = true;
52        break;
53      }
54    }
55    public boolean done() {
56      return isTerminate_;
57    }
58  }
```

Figure 2: An implementation of Role1 in JADE

2. PROBLEMS WHEN IMPLEMENTING INTERACTIONS

In this section, we consider the problems that arise with existing languages when implementing interactions. In section 2.1, we show an implementation with JADE, which is currently one of the most available agent platforms. In section 2.2, we explain the problems in implementing interactions with existing languages referring to the example code written with JADE.

2.1 Implementation for JADE

Interactions in JADE consist of actions that include message transmission and reception. Therefore, we implement agent actions of each agent that would participate in the interaction. We need keep track of state transition of agent actions for interactions. We have to use variables which represent the current state. Thus, a state transition is represented as a change in their value, and an agent action is represented as conditional branches based on their values. An agent action is implemented as a subclass of the $Behaviour$ class for JADE. The *action()* method is invoked until the *done()* method returns $true$ for the instance of the $Behaviour$ class. For example, the interaction represented in Fig.1 is implemented as Fig.2 and Fig.3 .

2.2 Problem with Existing Languages

We can implement interactions as shown in the previous subsections. However, the implementations have following problems:

- Decreased maintainability due to dispersal of codes.
- Decreased readability due to lack of intuitive state transitions.

A message exchange represented as an arrow in the AUML sequence diagram is implemented in the actions of the sender agent and the receiver agent. The message transmissions and the message reception are implemented in the definitions of different classes. However, they actually have a strong relationship and we will have to modify both of them if some changes to the message exchanges are to be applied. Thus, maintainability of interactions decreases. We explicitly set variable values to new values that represent the next state. However if the transitions are complex, it will not be easy to understand all of the state transitions and to determine the part where we should add information concerning the implementation. Suppose we have to modify the message creation on line 37 in Fig.3 . We may modify the corresponding message reception and the way to deal with the message, which is specified on line 38 in Fig.2 . However, it is not clear whether the modification is necessary since this part is described in a different class. Furthermore, it is also not easy to determine the part which we have to modify in Fig.2 . Furthermore, when we want to add the information which is represented as a note in the AUML sequence diagram, we will have to add it to the part somewhere around the line 19 in Fig.2 and around the line 23 in Fig.3 . However, the actual location where we have to add the required feature is not obvious.

The problems stated above are not specific to JADE. These problems hold true with AgenTalk, COOL,COSY and Q and other Java based language. Comparison with these languages are described in section 5.

```
1  public class Role2Behaviour
2    extends jade.core.behaviours.Behaviour {
3    int state_ = 0;
4    boolean isTerminate_ = true;
5    public void action() {
6      switch (state_) {
7      case 0:
8        /* recv message m1 or m7 */
9        if (/* is first loop continue? */) {
10         state_ = 1;
11       } else {
12         state_ = 7;
13       }
14       break;
15     case 1:
16       if (/* is first case? */) {
17         state_ = 2;
18       } else {
19         state_ = 6;
20       }
21       break;
22     case 2:
23       /* send message m2 */
24       state_ = 3;
25       break;
26     case 3:
27       /* recv message m3 of m5 */
28       if (/* is second loop continue? */) {
29         state_ = 4;
30       } else {
31         state_ = 5;
32       }
33       break;
34     case 4:
35       // send message m4
36       ACLMessage msg = new ACLMessage();
37       /* create the content of message */
38       msg.setContent(...);
39       myAgent.send(msg);
40
41       state_ = 3;
42       break;
43     case 5:
44       state_ = 0;
45       break;
46     case 6:
47       /* send message m6 */
48       state_ = 0;
49       break;
50     case 7:
51       isTerminate_ = true;
52       break;
53     }
54   }
55   public boolean done() {
56     return isTerminate_;
57   }
58 }
```

Figure 3: An implementation of Role2 in JADE

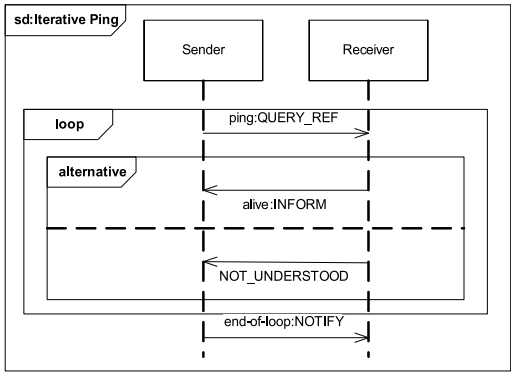


Figure 4: Iterative Ping Protocols described as an AUML sequence diagrams.

3. IOM/T:INTERACTION ORIENTED MODEL BY TEXTURAL REPRESENTATION

In this section, we propose an interaction description language called IOM/T which bridges the gap between design and implementation. In section 3.1, we show the design philosophy and in section 3.2 we provide the language specification.

3.1 Design Philosophy

The design philosophy of IOM/T is follows:

- Implementations should correspond with AUML sequence diagrams.
- Interactions should be described in a single structure.
- IOM/T should represent explicit control structures for state transition.
- The syntax of IOM/T should be similar to that of Java.

The aim of IOM/T is to bridge the gap between design and implementation. It is important that the implementation in IOM/T corresponds with the design in AUML sequence diagrams. IOM/T should not disperse an interaction into multiple agents since AUML sequence diagrams can represent it in a single diagram. This functionality would enable us to implement and modify interactions easily. IOM/T should have explicit control structures, which enable us to understand the message sequences intuitively as does the AUML sequence diagrams. This increases readability of implementations and helps us to add information concerning the implementation based on designs. While Java lacks expressiveness, most agent developers use this language. The syntax of IOM/T should be similar to that of Java in order to decrease the cost of learning IOM/T.

3.2 Specification of IOM/T

In this subsection, we present the specification of IOM/T using a simple example interaction, Iterative Ping Protocol, which is represented in Fig.4 . In this interaction, a $Sender$ sends a ping message to a $Receiver$. The $Receiver$ receives the message and decides whether it should reply with an "alive" message. Then it replies an "alive" message or a $NOT\_UNDERSTOOD$ message. And henceforth, it will continue with this communication repeatedly. The interaction described in IOM/T is shown in Fig.5. The syntax rule of IOM/T is provided in appendix A.

3.2.1 Definition of Interaction

In IOM/T, an interaction is represented as an *interaction* structure and has a unique identifier. In Fig.5 we defined an interaction whose identifier is $PingProtocol$. An *interaction* structure consists of the definition of roles which participate in this interaction and the definition of message sequences among the roles.

3.2.2 Definition of Role

We define roles using *role* structures. The *role* structures have a unique identifier within the interaction and consist of the definitions of role functionality, the definitions of information variable and the definitions of sub-interaction. We append an * to the *role* in order to represent that there are multiple agents which play this role. The definition of role functionality is specified using Java's method definition notation. The definition of information variable is specified using Java's field definition notation. When we have to deal with a large interaction, we divide it into multiple sub-interactions and represent the entire interaction using them. The definition of sub-interaction specifies what sub-interaction the role participates in and what role it plays in the specified sub-interaction, using the keyword $using$.

```
 1  interaction PingProtocol {
 2    role Sender {
 3      AID getTarget();
 4      boolean isContinue();
 5      void knowAsDead();
 6    }
 7    role Receiver {
 8      boolean doesReply();
 9      ACLMessage res;
10    }
11    protocol {
12      while (Sender.isContinue()) {
13        play Sender {
14          ACLMessage ping = new ACLMessage();
15          ping.setReceiver(getTarget());
16          ping.setContent("(ping)");
17          ping.setPerformative("QUERY_REF");
18          sendAsync(ping); //# m1
19        }
20        play Receiver {
21          ACLMessage ping = recvBlock(); //# m1
22          res = ping.createResponse();
23        }
24
25        if (Receiver.doesReply()) {
26          play Receiver {
27            res.setContent("(alive)");
28            res.setPerformative("INFORM");
29            sendAsync(res); //# m2
30          }
31          play Sender() {
32            ACLMessage msg = recvBlock(); //# m2
33          }
34        } else {
35          play Receiver {
36            res.setContent(ping.getContent());
37            res.setPerformative("NOT_UNDERSTOOD");
38            sendAsync(res); //m3
39          }
40          play Sender {
41            ACLMessage msg = recvBlock(); //# m3
42            knowAsDead();
43          }
44        }
45      }
46      play Sender {
47        ACLMessage msg = new ACLMessage();
48        msg.setReceiver(getTarget());
49        msg.setContent("(end-of-loop)");
50        msg.setPerformative("INFORM");
51        sendAsync(msg);
52      }
53    }
54  }
```

Figure 5: Iterative Ping Protocol in IOM/T

In Fig.5 , we define two roles. One is $Sender$ on lines 2-6 and the other is $Receiver$ on lines 7-10. In $Sender$, we define three functionalities, $getTarget()$ which determines the target of message, $isContinue()$ which determines whether to continue this interaction and $knowAsDead()$ which is called when the target does not reply correctly. In $Receiver$, we define a functionality, $doesReply()$ which determines whether it should reply correctly, and an information variable res which holds the reply message. Here, we do not define sub-interactions since this interaction is simple.

3.2.3 Definition of Message Sequences

We define message sequences using the *protocol* structure. The *protocol* structure consists of role actions and control structures. Table 1 shows the list of control structures.

In Fig.5 , the $protocol$ structure on lines 11-54 define the message sequence. Role actions on lines 13-19 and 20-23 represent role actions that are to be executed sequentially and the $if\text{-}else$ structure on lines 25-44 represents a conditional branch. Then the $while$ structure on lines 12-45 specifies that these sequences are to be repeated.

3.2.4 Role Action

sequential	the sequence of role actions represents that the role actions are executed according to the order of the sequence.
loop	the *while* structure represents that the contents of this structure are to be executed repeatedly until the specified role functionality determines to continue. The role action which notifies the end of loop to the participants must follow this structure.
conditional branch	the *if-else* structure represents conditional branch. Each block represents the execution and one of the block is executed on the basis of the specified condition.
conditional execution	the *if* structure represents conditional execution. This block is executed if the specified condition is fulfilled.
parallel	the *parallel* structure represents parallel execution. Each block represents the execution and these actions are executed parallel.
weak sequence	the *weak* structure represents weak sequence execution.
strong sequence	the *strong* structure represents strong sequence execution.
negative	the *negative* structure represents negative execution.
critical	the *critical* structure represents critical region.
ignore	the *ignore* structure represents the parts which should be ignored.
consider	the *consider* structure represents the parts which should be considered.
assert	the *assert* structure represents the parts which should be asserted.

Table 1: List of control structures

We define role actions using *play* structures which has one of the role identifier. We basically describe the contents of role actions using Java. We can use some extension. First, we can use role functionalities and information variables. Second, we can use some functionalities for handling FIPA ACL[10]. FIPA ACL is a specification for messages among agents and consists of several elements including $Sender$, $Receiver$, $Performative$ and $Content$. We can use the *ACLMessage* class which represents ACL Messages and has the methods for accessing these elements. We describe the message transmission using functions *sendSync()* and *sendAsync()*, and message reception using functions *recvBlock()* and *recvNonblock()*. Then, we can control the interactions, the beginning of sub-interactions and the termination of current interaction. The function *beginInteraction()* represents the beginning of sub-interaction and the function *terminateInteraction()* represents the termination of a current interaction.

In Fig.5 , the Sender determines the target by using functionality $getTarget()$ and sends a $(ping)$ message on lines 13-19. The Receiver receives the message on lines 20-23. The Receiver then decides which message it send to Sender on the basis of the condition on line 25.

3.2.5 Execution of Interaction

Interactions can be implemented as described in the preceding sections. However, it is also necessary to provide implementations of role functionalities in a multi-agent system. We describe the mapping of functionality to agent implementations using the *playing* structures which describe what methods the agent use as role functionalities. We can use *beginInteraction()* in order to begin an interaction. Fig.6 shows an example of an agent who plays the Sender.

```
 1  public class MyAgent extends Agent {
 2    playing PingProtocol.Sender {
 3      getTarget = getPingAgent;
 4      isContinue = isPingContine;
 5      knowAsDead = knowAsDead;
 6    }
 7    AID getPingAgent() { ... }
 8    boolean isPingContinue() { ... }
 9    void knowAsDead() { ... }
10  }
```

Figure 6: A specification of agent functionalities

4. AUML AND IOM/T

In this section, we show the equivalence of AUML sequence diagrams and IOM/T. We use π-calculus to verify the equivalence. π-calculus is a formal model for concurrent process and the processes are described using the following syntax rules. $P, P_i(i = 1, 2)$ represent processes, α represents actions and a,x represents name. $\overline{x}$ <> represents message transmission through x, $x()$ represents message reception through x. $\alpha.P$ represents action execution, $P_1|P_2$ represents parallel, $P_1 + P_2$ represents selection, $new\ a\ P$ represents the binding of the name x in P.

$$\begin{aligned} P &::= 0 \mid \alpha.P \mid P_1|P_2 \mid P_1 + P_2 \mid new\ a\ P \\ \alpha &::= \tau \mid \overline{x} <> \mid x() \end{aligned}$$

4.1 Formalization of AUML sequence diagrams

In this subsection, we show the formal model of AUML sequence diagrams using π-calculus. In this paper we consider only $alternative$, $option$, $loop$ and $parallel$ Combined Fragment since we deal with message sequences. For each lifeline, we formalize the process $R_i(i = 1, \cdots, n)$ as $[[R_i]]E$ by applying the rule described in Table 2 and Table 3 recursively. The interaction is formalized as follows with the message identifier $m_i(i = 1, \cdots, p)$ and end of loop identifier $end_i(i = 1, \cdots q)$:

$$new\ m_1 \cdots m_p\ end_1 \ldots end_q([[R_1]] | \cdots | [[R_n]])$$

4.2 Formalization of IOM/T

In this subsection, we shows the formal model of IOM/T using π-calculus. We deal only with message exchanges since our aim is to verify the equivalence to AUML sequence diagrams. Furthermore we do not deal with the condition of $while$ structure and $if - else$ structure for the sake of simplicity. The correspondence between transmissions and receptions are based on the identifier specified as comment since the meaning of the receiver field of $ACLMessage$ class is not included in IOM/T. For each role, we formalize the process $R_i(i = 1, \cdots, n)$ as $[[R_i]]E$ by applying the rule described in Table 2 and Table 3 recursively. The interaction is formalized as follows with the message identifier $m_i(i = 1, \cdots, p)$ and end of loop identifier $end_i(i = 1, \cdots q)$:

Structure of Lfeline	Structure of Code	Formalization : A(x)

N	play R {
	ACLMessage msg;
	sendAsync(msg); //#m
	}
	N
		A(x) = new next(
		$\overline{m<>}$.$\overline{next<>}$ | next().[[N]](x))
		m is a unique identifier of the arrow

N	play R {
	ACLMessage msg =
	recvBlock(); //#m
	}
	N
		A(x) = new next(
		m ().$\overline{next<>}$ | next().[[N]](x))
		m is a unique identifier of the arrow

alternative P1 ... Pn N	if (…) {
	P1
	}
	else if (…) {
	…
	} else {
	Pn
	}
	N
		A(x) = new done(
		[[P1]](done) + ··· + [[Pn]](done)
		| done().[[N]](x))

option P N	if (…) {
	P
	}
	N
		A(x) = new done(
		([[P]](done) + $\overline{done<>}$)
		| done().[[N]](x))

loop P loop termination N	while (S.functionality()) {
	P
	}
	play S {
	/* loop termination */
	}
	N
		A(x) = new done,next(
		[[P]](done)
		| (done().A(x)
		+ (done().$\overline{end<>}$.$\overline{next<>}$
		| next().[[N]](x))))
		end is a unique identifier of the loop

loop P loop termination N	while (S.functionality()) {
	P
	}
	N
		A(x) = new done, next(
		[[P]](done)
		| (done().A(x)
		+ (done().$\overline{end<>}$.$\overline{next<>}$
		| next().[[N]](x))))
		end is a unique identifier of the loop

parallel P1 ... Pn N	parallel {
	…
	flow { P1 }
	…
	flow { Pn}
	}
	N
		A(x) = new done(
		[[P1]](done) | ··· | [[Pn]](done)
		| done().···.done().[[N]](x))

sd R P	interaction id{
	role R { … }
	…
	protocol{
	P
	}
	}
		R = new done([[P]](done) | done())

Table 2: A formal model for AUML sequence diagrams and IOM/T part 1.

$$new\ m_1 \cdots m_p\ end_1 \ldots end_q([[R_1]] | \cdots | [[R_n]])$$

4.3 Equivalence between AUML sequence diagrams and IOM/T

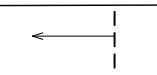
We have formalized both the AUML sequence diagrams and IOM/T. Details of proof are not shown due to space limitations. The $loop$ and $alternative$ CombinedFragments correspond to $while$ structures and $if - else$ structure. We can prove the equivalence by structural induction. For example, if $P1 \cdots Pn, N$ in the alternative Combined Fragment are equivalent to $P1 \cdots Pn, N$ in the $if - else$ structure, these representation are formalized to same formula. Therefore, they are equivalent.

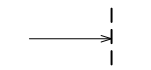
We can formalize both representation of Iterative Ping Protocol as showed in Fig.7 .

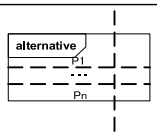
5. DISCUSSION

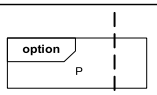
In this section, we evaluate IOM/T by comparing to related work. The interaction shown in Fig.1 can be described as shown in Fig.9 .

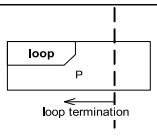
Structure of Lifeline	Structure of Code	Formalization : A(x)

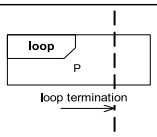
	play R {
	ACLMessage msg;
	sendAsync(msg); //#m
	}
		A(x) = $\overline{m}\langle\rangle.\overline{x}\langle\rangle$
		m is a unique identifier of the arrow

	play R {
	ACLMessage msg =
	recvBlock(); //#m
	}
		A(x) = m().$\overline{x}\langle\rangle$
		m is a unique identifier of the arrow

	if (…) {
	P1
	}
	else if (…) {
	…
	} else {
	Pn
	}
		A(x) = [[P1]](x) + ‥‥ + [[Pn]](x)

	if (…) {
	P
	}
		A(x) = ([[P]](x) + $\overline{x}\langle\rangle$)

	while (S.functionality()) {
	P
	}
	play S {
	/* loop termination */
	}
		A(x) = new done,next(
		[[P]](done)
		| (done().A(x)
		+ (done().$\overline{end}\langle\rangle$.$\overline{next}\langle\rangle$
		| next().$\overline{x}\langle\rangle$)))
		end is a unique identifier of the loop

	while (S.functionality()) {
	P
	}
		A(x) = new done, next(
		[[P]](done)
		| (done().A(x)
		+ (done().end().$\overline{next}\langle\rangle$
		| next().$\overline{x}\langle\rangle$)))
		end is a unique identifier of the loop

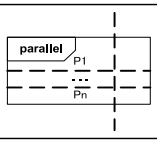
	parallel {
	…
	flow { P1 }
	…
	flow { Pn}
	}
		A(x) = new done(
		[[P1]](done) | ‥‥ | [[Pn]](done)
		| done().‥‥.done().$\overline{x}\langle\rangle$)

Table 3: A formal model for AUML sequence diagrams and IOM/T part 2.

Interactions represented in IOM/T are described in a single code and correspond with the design in AUML sequence diagrams. Thus, the correspondence between message transmissions and message receptions is clear. For instance, the message transmission on line 13 corresponds with the message reception on line 16. This correspondence is clearer than that which is described with existing languages. This increases maintainability of interactions. When we have to modify the code regarding message exchanges, we can easily find the message correspondence and parts that the modification would affect. For example, suppose some changes occurs on line 35 where the content of the message is created. We have to modify the part where Role1 receives the message and handles it. In IOM/T we can easily find that the part is on line 42. The same applies when we add information regarding the implementation. Suppose we have to implement the information represented as a note in Fig.1 which specifies timing constraints. Role2 should create messages on the basis of the time it takes for creating them. Thus, we change the method to create message on the basis of the remaining time. We will implement this on lines 20-21 as follows:

```
if (/* tight constraints */) {
  msg.setContent(roughContent);
} else {
  msg.setContent(preciseContent);
}
```

Role1 should judge whether the constraints are fulfilled. Role1 should output error logs when the constraints are not fulfilled. Thus, we will implement the following on line 24:

$$
\begin{aligned}
Sender &= new\ done(L_1|done()) \\
L1 &= new\ next(\overline{m1}\langle\rangle.\overline{next}\langle\rangle \\
&\quad |\ next().(m2().\overline{done}\langle\rangle\ +\ m3().\overline{done}\langle\rangle)) \\
&\quad |\ (done().L1\ +\ done().\overline{end}\langle\rangle.\overline{done}\langle\rangle) \\
Receiver &= new\ done(L2|done()) \\
L2 &= new\ next(m1().\overline{next}\langle\rangle \\
&\quad |\ next().(\overline{m2}\langle\rangle.\overline{done}\langle\rangle\ +\ \overline{m3}\langle\rangle.\overline{done}\langle\rangle)) \\
&\quad |\ (done().L2 + done().end().\overline{done}\langle\rangle) \\
I &= new\ m1\ m2\ m3\ end(Sender|Receiver)
\end{aligned}
$$

Figure 7: A formal model of Iterative Ping Protocol.

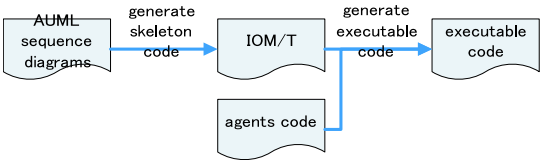


Figure 8: An overview of implementation using IOM/T.

```
if (/* are constraints fulfilled? */) {
  /* normal execution */
} else {
  /* output error log */
}
```

We should be able concentrate on what we have to implement for these constraints. However, if we use existing languages, we will be confused as to where we are supposed to implement the constraints. In the case of IOM/T we can easily determine the part where we are supposed to implement the constraints since we are able to recognize that we have to implement matters regarding the first message after the conditional branch in Fig.1 and the implementation for the message is clear in Fig.9 . Furthermore, the difference between IOM/T and existing languages become clearer if we implement more complex interactions, especially interactions that include parallel execution. The number of states increases exponentially in existing languages due to the number of combination of the states in a parallel process. As a result, maintainability and readability decreases. For IOM/T this will not be an issue since we just implement each parallel process separately.

In this section, we mainly perform comparison with the implementation for JADE. However, it is the same for the other languages such as AgenTalk, COOL, COSY and Q. Although these languages have interesting characteristic of their own, we have to divide the interaction into multiple agents in order to implement it in these languages. Q has representation for state transitions but we have to use explicit transitions and state transitions cannot be understood intuitively.

The gap between design and implementation is a common problem of multi-agent systems. One solution is to enhance the design descriptions [8, 5]. Another is to develop implementation description [2]. Our work belongs to the latter solution, we believe this is a first interaction language that is designed to bridge the gap between the AUML sequence diagrams and implementation.

Fig.8 shows an overview of the process starting from the AUML sequence diagrams to the implementation of a multi-agent system.

First, we generate skeleton codes of IOM/T from AUML sequence diagrams. Some researches have been proposed regarding this process. M.P.Huget considers the process of generating Java code from AUML sequence diagrams[13]. M.Dinkloh has developed a tool which generates skeleton code for JADE from AUML sequence diagrams[6]. IPEditor[16] also generates skeleton code for Bee-gent[17] from a graphical design. We do not show the way to generate skeleton code in IOM/T due to space limitation. However, it is not difficult since IOM/T has structures that correspond to AUML sequence diagrams. Next, we implement interaction by adding information to the skeleton codes. We can concentrate on what we should implement since IOM/T improve maintainability and readability of implementation of interaction. By the way, we will not limit the target agent-platform for multi-agent systems. The IOM/T compiler will convert the implementation in IOM/T and the implementation of agents for the platform into implementations of multi-agent systems for that platform. The implementation of agents includes the methods which realize the role functionalities. The implementation of agents is beyond the scope of this paper. The implementations in IOM/T include information which the design descriptions lack. Thus, the IOM/T compiler can generate executable implementation of multi-agent system for the target agent-platform. This allows us to reuse the implementation of interactions. We have developed the IOM/T compiler for JADE.

6. CONCLUSION

In this paper, we have proposed a new interaction description language called IOM/T. Correspondence with the design in AUML is not considered in existing languages. IOM/T corresponds with AUML sequence diagrams and allows us to describe without dispersing codes of interactions. IOM/T is capable of describing intuitive state transitions. Thus, maintainability and readability of implementations of interactions increase if we use IOM/T. Furthermore, we have provided the formal model for AUML sequence diagrams and IOM/T and proved the equivalence of both representations.

In the future, we will create a more precise formal model of IOM/T and develop compilers for various agent-platforms. We also plan to verify the effectiveness of IOM/T through demonstration experiments.

```
 1  interaction Sample {
 2    role Role1 {
 3      boolean isFirstLoopContinue();
 4      boolean isSecondLoopContinue();
 5    }
 6    role Role2 {
 7      boolean isFirstCase();
 8    }
 9    protocol {
10      while (Role1.isFirstLoopContinue()) {
11        play Role1 {
12          ACLMessage msg;
13          sendAsync(msg); //# m1
14        }
15        play Role2 {
16          ACLMessag msg = recvBlock(); //# m1
17        }
18        if (Role2.isFirstCase()) {
19          play Role2 {
20            ACLMessage msg;
21            sendAsync(msg); //# m2
22          }
23          play Role1 {
24            ACLMessage msg = recvBlock(); //# m2
25          }
26          while (Role1.isSecondLoopContinue()) {
27            play Role1 {
28              ACLMessage msg;
29              sendAsync(msg); //# m3
30            }
31            play Role2 {
32              ACLMessage msg1 = recvBlock(); //# m3
33              ACLMessage msg2
34                   = msg1.createResponse();
35              /* create the content of message */
36              msg.setContent(...);
37              sendAsync(msg2); //# m4
38            }
39            play Role1 {
40              ACLMessage msg = recvBlock(); //# m4
41              Object content = msg.getContent();
42              /* handle the content */
43            }
44          }
45          play Role1 {
46            ACLMessage terminateMsg;
47            sendAsync(terminateMsg);
48          }
49        } else {
50          play Role2 {
51            ACLMessage msg;
52            sendAsync(msg); //# m5
53          }
54          play Role1 {
55            ACLMessage msg = recvBlock(); //# m5
56          }
57        }
58      }
59      play Role1 {
60        ACLMessage terminateMsg;
61        sendAsync(terminateMsg);
62      }
63    }
64  }
```

Figure 9: An implementation in IOM/T

7. ADDITIONAL AUTHORS

Additional authors: Nobukazu YOSHIOKA (National Institute of Informatics, email: nobukazu@nii.ac.jp)

8. REFERENCES

[1] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.

[2] L. Braubach, A. Pokahr, and D. Moldt. Goal representation for BDI agent systems. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop*, 2004.

[3] B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic, configurable, cooperation protocols for multi-agent systems. *In LNAI, From Reaction to Cognition, MAAMAW 93*, pages 157–171, 1993.

[4] CSELT. JADE. http://sharon.cselt.it/projects/jade/.

[5] V. T. da Silva, R. Choren, and C. J. de Lucena. A UML based approach for modeling and implementing multi-agent systems. In *The Third International Joint Conference on Autonomous Agents & Multi Agent Systems AAMAS2004*, pages 914–921, 2004.

[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] T. DOI, N. YOSHIOKA, Y. TAHARA, and S. HONIDEN. Bridging the gap between AUML and implementation using IOM/T. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop*, 2004.

[8] L. Ehrler and S. Cranefield. Executing Agent UML diagrams. In *The Third International Joint Conference on Autonomous Agents & Multi Agent Systems AAMAS2004*,

pages 906–913, 2004.

[9] emorphia. FIPA-OS. http://www.emorphia.com/research/about.htm.

[10] FIPA. FIPA ACL message structure specification. http://www.fipa.org/specs/fipa00061/.

[11] FIPA. The foundation for intelligent physical agents. http://www.fipa.org.

[12] FIPA. Interaction diagrams. http://www.auml.org/auml/documents/ID-03-07-02.pdf.

[13] M.-P. Huget. Generating code for Agent UML sequence diagrams. Technical report, University of Liverpool Department of Computer Science, 2002.

[14] T. Ishida. Q: A scenario description language for interactive agents. *IEEE Computer*, 2002.

[15] K. Kuwabara, T. Ishida, and N. Osato. Agentalk: Describing multiagent coordination protocols with inheritance. In *Proc. 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '95)*, 1995.

[16] 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.

[17] TOSHIBA Corporation. Bee-gent. http://www2.toshiba.co.jp/beegent/index.htm.

APPENDIX

A. SYNTAX RULE

$$
\begin{array}{rcl}
InteractionDef & ::= & \text{interaction } Id\ InteractionBody \\
InteractionBody & ::= & RoleDefs\ ProtocolDef \\
RoleDefs & ::= & RoleDef \\
 & | & RoleDefs\ RoleDef \\
RoleDef & ::= & \text{role } *_{opt}\ Id\ RoleBody \\
RoleBody & ::= & \{\ RoleBodyDefs_{opt}\ \} \\
RoleBodyDefs & ::= & RoleBodyDef \\
 & | & RoleBodyDefs\ RoleBodyDef \\
RoleBodyDef & ::= & SubProtocolDef \\
 & | & FuncDef \\
 & | & InformationDef \\
SubProtocolDef & ::= & \text{using } InteractionName.RoleName; \\
FuncDef & ::= & ReturnType\ FuncId(Arguments_{opt}); \\
InformationDef & ::= & InformationType\ InfomationId; \\
ProtocolDef & ::= & \text{protocol } ProtocolBody \\
ProtocolBody & ::= & \{\ ProtocolBlocks\ \} \\
ProtocolBlocks & ::= & ProtocolBlock \\
 & | & ProtocolBlocks\ ProtocolBlock \\
ProtocolBlock & ::= & \text{play } RoleId\ ActionBlock \\
 & | & LoopBlock \\
 & | & AlternativeBlock \\
 & | & OptionBlock \\
 & | & ParallelBlock \\
 & | & WeakBlock \\
 & | & StrongBlock \\
 & | & NegativeBlock \\
 & | & CriticalBlock \\
 & | & IgnoreBlock \\
 & | & ConsiderBlock \\
 & | & AssertBlock \\
LoopBlock & ::= & \text{while(}\ WhilePredicate\ \text{)\{}\ ProtocolBlocks\ \} \\
WhilePredicate & ::= & RoleId.FuncId(Args_{opt}) \\
AlternativeBlock & ::= & IfBlock\ ElseIfBlocks_{opt}\ ElseBlock
\end{array}
$$

$$
\begin{array}{rcl}
IfBlock & ::= & \text{if(}\ IfPredicate\ \text{)\{}\ ProtocolBlocks\ \} \\
ElseIfBlocks & ::= & ElseIfBlock \\
 & | & ElseIfBlock\ ElseIfBlocks \\
ElseIfBlock & ::= & \text{else if(}\ IfPredicate\ \text{)\{}\ ProtocolBlocks\ \} \\
ElseBlock & ::= & \text{else \{}\ ProtocolBlocks\ \} \\
OptionBlock & ::= & \text{if(}\ IfPredicate\ \text{)\{}\ ProtocolBlocks\ \} \\
IfPredicate & ::= & RoleId.FuncId(Args_{opt}) \\
ParallelBlock & ::= & \text{parallel \{}\ JudgeFuncs\ ParallelSequences\ \} \\
JudgeFuncs & ::= & JudgeFunc \\
 & | & JudgeFunc\ JudgeFuncs \\
JudgeFuc & ::= & RoleId{:}FuncId \\
ParallelSequences & ::= & ParallelSequence \\
 & | & ParallelSequence\ ParallelSequences \\
ParallelSequence & ::= & \text{flow \{}\ ProtocolBlock\ \} \\
WeakBlock & ::= & \text{weak \{}\ ProtocolBlocks\ \} \\
StrongBlock & ::= & \text{strong \{}\ ProtocolBlocks\ \} \\
NegativeBlock & ::= & \text{negative \{}\ ProtocolBlocks\ \} \\
CriticalBlock & ::= & \text{critical \{}\ ProtocolBlocks\ \} \\
IgnoreBlock & ::= & \text{ignore \{}\ ProtocolBlocks\ \} \\
ConsiderBlock & ::= & \text{consider \{}\ ProtocolBlocks\ \} \\
AssertBlock & ::= & \text{assert \{}\ ProtocolBlocks\ \}
\end{array}
$$

$$
\begin{array}{rcl}
PlayingDef & ::= & \text{playing} \\
 & & ProtocolId.PlayerId\ PlayingDefBody \\
PlayingDefBody & ::= & \{FuncMappings_{opt}\} \\
FuncMappings & ::= & FuncMapping \\
 & | & FuncMappings\ FuncMapping \\
FuncMapping & ::= & NormalFuncMapping \\
 & | & ProtocollFuncMapping \\
NormalFuncMapping & ::= & FuncId\ =\ MethodId \\
ProtocolFuncMapping & ::= & ProtocolSpecifiers.FuncId\ =\ MethodId \\
ProtocolSpecifiers & ::= & ProtocolSpecifier \\
 & | & ProtocolSpecifiers\ ProtocolSpecifier \\
ProtocolSpecifier & ::= & ProtocolId.PlayerId
\end{array}
$$