Communication Mechanism on Autonomous Objects

Yutaka Ishikawa
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba
Ibaraki, 305 JAPAN
yisikawa@etl.go.jp

Abstract

In the concurrent object-oriented programming methodology, a system is described by concurrent objects which communicate with each others by various communication facilities, i.e., synchronous/asynchronous(future) message passing. Those facilities help up to implement application programs based on the client/server model. It is, however, difficult to describe application programs such that concurrent objects may simultaneously initiate communication with each other. Such objects are called *autonomous* objects. In this paper, we propose the notion of the **visible** and **intensive** sets, and a communication mechanism using those sets which enables us to handle communication among autonomous objects safely and easily.

1 Introduction

In concurrent object-oriented programming languages [5, 6, 3, 4, 11, 10], an application program is described as a set of concurrent objects which communicate with each others by using various message passing facilities such as synchronous/asynchronous(or future) message passing forms. The concurrency and synchronization are programmed by using those communication facilities and objects. Indeed, concurrent object-oriented languages help us to describe an application program which can be modeled as the client/server or master/slave relation. A client or a master always initiates communication to a server or a slave. However, it is difficult to program autonomous objects which have equal rights. The autonomous objects may initiate communication to other objects at any time. This implies that two autonomous objects initiate communication to each other simultaneously. For example, an object `O` sends a message to another object `P` and at the same time `P` sends another message to `O`. If neither `O` nor `P` performs for a message due to waiting for the reply message, this causes deadlock. This kind of communication inherently causes synchronization problems.

To reduce the complexity of the simultaneous request/reply among concurrent objects, we usually create another object which handles messages from both autonomous objects. The party of communication may send a request to that object instead of the direct communication to the party. For example, Linda[2] supports a mechanism called "tuple space" to handle such communication. However, the direct communication among autonomous objects is required in some applications because we can not create any shared resources.

In order to overcome the limitations of the concurrent object-oriented languages, a new communication mechanism is proposed in this paper. In the following section, we discuss issues related to communication among autonomous objects by programming an example in two types of concurrent

OOPSLA'92, pp. 303-314

object-oriented languages. Then, we make it clear what capabilities are missing in those languages. In section 3, we propose the notion of the *visible set* and the *intensive set.* Then communication and inheritance facilities over those sets are introduced. Another facility called *transitional method* is proposed in section 4. The facility helps us to handle the object behavior according to its states. By using those facilities, the programmer easily develop autonomous objects. In section 5, some examples including the *inheritance anomaly* problem[7, 8] are shown to demonstrate the capabilities of the proposed facilities.

2 Issues

Most concurrent object-oriented programming languages provide for synchronous/asynchronous message passing forms to specify parallelism among objects. In the synchronous message passing, a sender object must wait for the reply of the receiver object. In the asynchronous message passing (or future), instead of waiting for the reply message immediately, a sender object obtains a special object called *future object* which will receive the reply message. The sender object may access the result via the future object.

Those communication facilities help us to describe an application program which is modeled as the client/server relation. Moreover, the inheritance mechanism allows us to reuse a program. It is, however, difficult to develop an application program which consists of autonomous objects each of which communicates with each others simultaneously. To make it clear what are issues related to concurrent objects, we show the following simple example in this paper:

> There are two autonomous mobile robots running. We do not assume that there are any shared resources to communicate with those robots. When a robot detects that it will soon collide with another robot, the robot initiates the negotiation action. This implies that two robots may initiate the negotiation action at the same time.
>
> For more simplicity, we assume that a robot is moving from the east to the west while another robot is moving from the north to the south as shown in Figure 1. The strategy of avoiding collision is just *stopping* or *going ahead.* Here we define a very simple negotiation protocol:
>
> 1. A robot sends a **negotiation** message with a random number to another robot,
> 2. If the permission to move is received then
>   (a) the moving action is proceeded,
>   (b) After passing the critical area, a **goahead** message is sent to the other robot.
> 3. Otherwise stop moving until receiving a **goahead** message.
> 4. If a robot receives a **negotiation** message from another robot, it determines whether or not the permission is given to the sender according to the number received and its state.

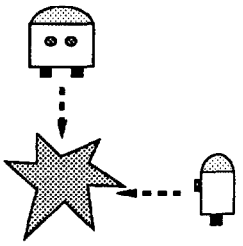

Figure 1: Negotiation between Mobile Robots

It should be noted that the example includes a mutual exclusion problem but the central issue is how communication among autonomous objects is safely and easily described in this paper. Because we cannot assume any shared resources, it is impossible to employ the "tuple space" of Linda[2] or a

	Communication among Autonomous Objects	Inheritance
Flat M.A.	*Complicated*	*Yes*
Nested M.A.	*Easier than Flat M.A.*	*No*

M.A. = Message Acceptance

Table 1: Comparison of Languages based on Flat and Nested Message Acceptance Mechanisms

similar mechanism. Thus, we have to program concurrent objects each of which communicates with each other at the same time.

Concurrent object-oriented languages are classified into two types in terms of the way of message acceptance, i.e, i) the *flat message acceptance* and ii) the *nested message acceptance*. By the flat message acceptance we mean that a method for a message is defined as the flat way but not defined within another method such as in Smalltalk-80. Such concurrent object-oriented languages include Concurrent Smalltalk[10] and Orient84/K[5]. In the nested message acceptance, a method for a message is defined within another method such as ABCL[11].

Throughout programming the example by two types of message acceptance in the following subsections, we will show the difficulty of programming the example by those. That is, no concurrent object-oriented languages provide both suitable mechanisms of communication among autonomous objects and inheritance as shown in Table 1.

2.1 Flat Message Acceptance

Languages based on the *flat message acceptance* such as Concurrent Smalltalk[10] and Orient84/K[5] support the single thread of control within an object since the multiple threads yield the complexity of programming, e.g., synchronization among threads. This implies that incoming messages of an object are serialized and each message is handled at a time. Some languages provide a mechanism to perform for an express message when such an express message arrives during the

```
1 [class Vehicle
2   (supers Object)
3   (vars     peer state myValue)
4 ]
5 [express method Vehicle negotiation: hisValue
6   (vars ...)
7   if state = #negotiating then
8     "both robots initiate the negotiation action
9      at the same time"
10    if hisValue > myValue then
11      state ← #giveaway;
12      reply #accept
13    else
14      reply #notaccept
15    endif
16  else
17    state ← #giveaway;
18    reply #accept
19  endif
20 ]
21 [method Vehicle run
22   (vars  lstate  ret)
23   myValue ← Random new;
24   state ← #moving;
25   "moving action is here"
26   [atomic
27     if state = #moving then
28       state ← #negotiating;
29       lstate ← #mustwait;
30       ret ← peer negotiation: myValue &
31     else
32       "state is #giveaway"
33       lstate ← #nowait;
34     end
35   ]
36   if lstate = #mustwait then
37     [waitfor ret]
38     if ret = #accept then
39       "moving action is here"
40       peer goahead &
41     else
42       "waiting for peer's action"
43     endif
44   else
45     "waiting for peer's action"
46   endif
47   "..."
48 ]
49 [method Vehicle goahead
50   "moving again"
51 ]
```

Figure 2: Flat Message Acceptance

execution of an object. Because the current execution is suspended and another message is handled in this case, we say that the execution is preemptive. A method for an express message is called an *express method* in this paper.

We can program the example in those languages. Instead of using a specific language, the program is written in a model language because of keeping the generality of languages of the *flat message ac-*

ceptance.

As shown in Figure 2, the `Vehicle` class is defined where methods `negotiation:`, `run`, and `goahead` are defined. The method `negotiation:` is defined as an express method in line 5.

A region in lines 26 to 35 is called an `atomic` region which is executed without preemption. That is, even when a `negotiation:` message arrives, the execution for the message is postponed during the execution of the `atomic` region.

Method `run` is the main routine of the `Vehicle` class. In line 25, the moving action must be programmed. When the object detects that it will collide with another vehicle object, the execution falls into line 26 to negotiate with the other object.

The `state` variable is keeping the object's state. The state has one of `#moving`, `#giveaway`, and `#negotiating`. State `#moving` denotes that the object is moving. State `#giveaway` denotes that the object permits another object to move while `#negotiating` is the state of the negotiation phase.

In the atomic region on lines 26 to 35, the object's action is determined according to the `state` variable. The atomic region is needed because the `state` variable may be changed by the `negotiation:` method which will be invoked at any time except for during the execution of an atomic region.

If the `state` variable is still `#moving` in line 27, then the `state` is changed to `#negotiating` and send the `negotiation:` message to another object asynchronously. That is, instead of waiting of the reply message immediately, a special object called *future object* is stored in the `ret` variable in line 30. It should be noted that we cannot wait for the reply message here. This is because the other object may also be sending the `negotiation:` message to the object. This is a deadlock case if we wait for the reply message in the atomic region. The `lstate` variable is introduced in order to receive the reply message outside the atomic region, i.e., in line 37.

The `waitfor` statement is a facility for the synchronization of the asynchronous message passing. The asynchronous message passing in line 30 is synchronized in line 37. In other words, the `waitfor` statement in line 37 is to wait for receiving the reply message from the `peer` object at the *future object* stored in the `ret` variable. Because the `waitfor` statement is executed outside of the atomic region, an express message `negotiation:` may be handled when the message arrives.

If the `Vehicle` object receives the permission from the `peer` object, the moving action is proceeded in line 39 and then the `goahead` message is sent to the other robot in order to make the `peer` object move.

In this way, to maintain the consistency of a state variable kept in an object and to avoid a deadlock, programming is very complicated in languages of the flat message acceptance.

2.1.1 Inheritance Anomaly

Let us take look at the `negotiation:` method. The `state` variable is checked in line 7 since the method will be invoked when the object's state is `#moving` or `#negotiating`. This implies that we have to rewrite the method if we add another state into the `state` variable, e.g., `#stopping`. Thus, it is difficult to inherit such a method with adding new states.

2.2 Nested Message Acceptance

We show a program example written in a language based on the *nested message acceptance.* Instead of using a specific language, we use a model language again.

Figure 3 shows the program of the example. The `select` statement such as Ada is introduced to receive messages from other objects. In the `select` statement in line 7, the object is waiting to receive the `run` message defined in line 8 or the `negotiation:` message defined in line 42. When the `run` message arrives at the object, the `run` method defined in line 8 is executed.

The `negotiation:` method in line 42 is defined with the `express` keyword. This means that the `negotiation:` method may be executed when message `negotiation:` arrives while the `run`

```
 1 [class Vehicle
 2   (supers Object)
 3   (vars     peer state myValue)
 4 ]
 5 [method Vehicle mainroutine
 6  (vars    ret)
 7  [select
 8     [method run
 9       myValue ← Random new;
10       state ← #moving;
11       "moving action is here"
12       [atomic
13        if state = #moving then
14          peer negotiation: myValue % self &
15          [select-loop
16             [method result: ret
17               if ret = #accept then
18                 "moving action is here"
19                 peer goahead &
20               else
21                 state ← #giveaway;
22               endif
23               exit-loop]
24          or [method negotiation: hisValue % dst
25               "both robots initiate the negotiation
26                action at the same time"
27               if hisValue > myValue then
28                 state ← #giveaway;
29                 dst result: #accept &
30               else
31                 dst result: #notaccept &
32               endif ] ]
33        endif
34       ]
35       if state = #giveaway then
36         "waiting for peer's action"
37         [method goahead
38            "moving action is here"
39         ]
40       endif
41       "...." ]
42   or [express method negotiation: tmp % dst
43       state ← #giveaway.
44       dst result: #accept &]
45  ]
46 ]
```

Figure 3: Nested Message Acceptance

method is executed. In the method the sender object is kept by the `dst` variable, so that the reply message is sent to the sender object explicitly. This semantics is derived from ABCL/1[1].

The semantics of the asynchronous message expression in line 14 is based on ABCL/1 [2]. The `negotiation:` message is sent to the `peer` object asynchronously with explicitly asserting the reply destination, i.e., the sender itself in this case. Then the `select-loop` statement in line 15 waits for two events: i) receiving the reply message from the `peer` object in line 16, and ii) receiving the `negotiation:` message in line 24.

As shown in Figure 3, we do not need state `#negotiation` and the `lstate` variable. This is because the `negotiation:` message acceptance is multiply defined in this program. The first message reception in line 42 is active at state `#moving` while the second message reception in line 24 is enable during the negotiation phase.

In the *nested message acceptance* mechanism, an extra state and variables are not needed. However, we can not reuse such a program because we can not overwrite the behavior of a *nested message acceptance.*

2.3 Summaries and Requirements

Generally speaking, a problem we have considered arises in the following scenario: when an object `O` performs some actions as shown below and at the same time an object `P` sends a message M_2 to `O` and blocks for the reply.

1. object `O` checks a variable V_o which reflects the object's state,
2. `O` sends a message M_1 to object `P` according to variable V_o, and then
3. changing the value of V_o according to the reply from `P`.

If actions 1 through 3 are executed without preemption, i.e., not performing for M_2, then `O` and `P` may cause a deadlock. If those actions are executed with preemption, then the mutual exclusion problem occurs. That is, variable V_o might be changed by another method which is invoked at the acceptance of message M_2 during the execution.

To avoid the problem, we have shown two programming examples. Actions 1 and 2 are performed with non-preemptive and then action 3 is realized by the synchronization of the asynchronous

[1] In ABCL/1, the message acceptance form is written as "`(=> negotiation: tmp @ dst ...)`". Because the "@" mark is reserved for another meaning in this paper, "%" is used.

[2] The expression is equivalent to "`peer <= [negotiation: myValue] @ self`" in ABCL/1.

message passing (or the future communication), so that the execution is preemptive. Thus, the object O can receive the M_2 message and performs for the message while the object waits for the reply of M_1. It is, however, rather complicated programming. In order to program such communication safely and easily, the following mechanisms must be supported by a programming language.

- State and Method Behavior
  The behavior of a method depends on the object's state. However, if we introduce an explicit state variable in order to program a method, it is difficult to inherit the method as described in section 2.1.1. A mechanism to control the behavior without accessing an extra state variable in a method is required.

- Communication and State
  During communication with another object, the object's state may be changed by performing for an incoming message. A communication mechanism is demanded to handle such communication safely.

- Inheritance
  The inheritance mechanism is needed to reuse a program. Since we can not redefine a method which has been defined as a nested method in a parent object, a language based on the flat message acceptance is chosen.

3 Dynamic Method Scoping

To control the behavior of a method without any extra state variables, we propose the notion of *visible set* in this section. Then, *intensive set* is introduced in order to handle express messages during communication with other objects safely. Finally, an inheritance mechanism with *visible set* is introduced in this section.

3.1 Method Set

First, a *method set* is introduced in order to define several actions of a method in several states.

```
1 [class ExampleObject
2   (supers nil)
3   (vars   ...)
4 ]
5 [mset #A1 method ExampleObject M1
6   /* method body : bm11 */
7 ]
8 [mset #A1 method ExampleObject M2
9   /* method body : bm21 */
10 ]
11 [mset #A2 method ExampleObject M1
12   /* method body : bm12 */
13 ]
14 [method ExampleObject M0
15   /* method body : bm00 */
16   anObject msg @ #A1;
17 ]
```

Figure 4: `ExampleObject`

Let a method be represented by a pair of (n, b) where n denotes a method name and b denotes the method's body. The pair is called *method name-body pair*. A member of a method set is a method name-body pair or another method set. A method set is denoted by a symbol name. For example, method sets $\{(\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21})\}$ denoted by `#A1` and $\{(\texttt{M1}, b_{m12})\}$ denoted by `#A2` are represented as follows:

$$[\texttt{\#A1},\ \{(\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21})\}]$$
$$[\texttt{\#A2},\ \{(\texttt{M1}, b_{m12})\}]$$

A program of the above example is shown in Figure 4. To define a method included in a method set, the `mset` keyword is added before the `method` keyword.

The method `M1` is multiply defined in this example. The method name `M1` is defined with the method body b_{m11} in the `#A1` method set while the same name is defined with another method body b_{m12} in the `#A2` method set.

3.2 Visible Set

In a *visible set* whose member is a method set or a method name-body pair, all method name-body pairs in the *visible set* are visible in the runtime routine. An auxiliary set called *invisible set* which is another method set is introduced in order to define operations over the visible set of an object. Let a *method scope* be a pair of the visible and invisible

$$
\begin{aligned}
V &= \{\ (\texttt{M0}, b_{m00}),\\
&\quad\ [\texttt{\#A1}, \{(\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21})\}],\\
&\quad\ [\texttt{\#A2}, \{(\texttt{M1}, b_{m12})\}]\ \}\\
IV &= \{\}
\end{aligned}
$$

Figure 5: Initial Visible and Invisible Sets in `ExampleObject`

sets. An object has method scope $M_s = (V, IV)$ where I and IV are a visible and an invisible sets, respectively. For example, the **ExampleObject** defined in Figure 4 has the initial visible and invisible sets shown in Figure 5.

The basic operations over the method scope are **include** and **exclude**. To describe the semantics of the **include** and **exclude** operations, we assume that the following scenario is performed in the initial visible and invisible sets shown in Figure 5.

1 **include** #A1;
2 **include** #A2;
3 **exclude** #A2;

After the execution in line 1, the visible and invisible sets are changed below. The object may perform for messages M0, M1, and M2 at this point.

$$
\begin{aligned}
V &= \{\ (\texttt{M0}, b_{m00}), (\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21}),\\
&\quad\ [\texttt{\#A2},\ \{(\texttt{M1}, b_{m12})\}]\ \}\\
IV &= \{[\texttt{\#A1},\ \{[\texttt{\#A1},\ \{(\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21})\}]\}]\}
\end{aligned}
$$

In this way, the **include** operation taking a symbol name is to add all members of a method set denoted by the symbol name into V. The method set denoted by the symbol name is excluded from V. The set is included in a new method set denoted by the same symbol name and the new set is added in IV.

After the execution in line 2, the visible and invisible sets are changed as follows:

$$
\begin{aligned}
V &= \{\ (\texttt{M0}, b_{m00}), (\texttt{M1}, b_{m12}), (\texttt{M2}, b_{m21})\ \}\\
IV &= \{[\texttt{\#A2},\ \{(\texttt{M1}, b_{m11}), [\texttt{\#A2},\ \{(\texttt{M1}, b_{m12})\}]\}],\\
&\quad\ [\texttt{\#A1},\ \{[\texttt{\#A1},\ \{(\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21})\}]\}]\}
\end{aligned}
$$

That is, method $(\texttt{M1}, b_{m11})$ is hidden and method $(\texttt{M1}, b_{m12})$ is visible. In this way, if the same name of a method name-body pair is already included in V, the method name-body pair is replaced. The old method name-body pairs and the method set denoted by the parameter of the **include** operation are included in a new method set. The new set is added in IV.

Issuing the **exclude** operation in line 3, the sets are changed as follows:

$$
\begin{aligned}
V &= \{\ (\texttt{M0}, b_{m00}), (\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21}),\\
&\quad\ [\texttt{\#A2},\ \{(\texttt{M1}, b_{m12})\}]\ \}\\
IV &= \{[\texttt{\#A1},\ \{[\texttt{\#A1},\ \{(\texttt{M1}, b_{m11}), (\texttt{M2}, b_{m21})\}]\}]\}
\end{aligned}
$$

The **exclude** operation taking a symbol name performs the following tasks: i) finding a method set denoted by the symbol name in IV and excluding the set from IV (let the set be ms), ii) for all method name-body pairs in ms, replacing a method name-body pair in V with another method name-body pair in ms such that those are the same method name, and iii) method sets which are members of ms are added in V. The semantics of **include** and **exclude** operations is formally described in the appendix.

The initial visible set is modified by the **include** statement in the object definition. For example, if the **ExampleObject** definition is modified as below, the initial visible and invisible sets are the same as the result of issuing the **include** operation taking **#A1**.

1 [**class** ExampleObject
2 (**supers** nil)
3 (**vars** ...)
4 (**include** #A1)
5]

The *visible set* and operations over the set enable us to write a program in which the behavior(or body) of a method is changed without introducing any explicit variables accessed in the method. In other words, by those facilities it is easy to program the same functionality written in a language of the nested message acceptance.

It should be emphasized that we can define other operators over the *visible set*, e.g., an operation which replaces all members of the visible set with all members of a method set.

```
 1 [class ExampleDerived
 2   (supers    ExampleObject sset #P1)
 3   (vars      ...)
 4 ]
 5 [method ExampleDerived M1
 6    /* method body : bm31 */
 7   scope  #P1;
 8   ...
 9   scope;
10 ]
```

Figure 6: `ExampleDerived`

3.3 Intensive Set

It is required that an object should perform for a message on the way to communicate with another object. To handle such a case, the *intensive set* is proposed. Let *intensive set* I be a set whose member is a method set or a method name-body pair. Methods represented by method name-body pairs in I are immediately executed when messages for those methods arrive at the object. In other words, all the *express methods* described in section 2 are managed in the *intensive set.*

The communication facility over the *intensive set* is called *intensive communication.* The following statement is an example of the intensive communication, which is extracted from line 16 in Figure 4.

```
anObject msg @ #A1;
```

In this statement, all members of the method set denoted by `#A1` are added in *intensive set* I during the communication with `anObject`. That is,

$$I = \{(\texttt{M1}, b_{m11}),\ (\texttt{M2}, b_{m21})\}$$

During the communication, the object may accept and execute methods included in the *intensive set*, i.e., methods M1 and M2. After the communication is done, the *intensive set* is changed to empty.

An example below is the short form of the intensive communication. In this program, the object sends the `msg` message to `anObject` and at the same time it can receive messages `do:` and `e:`. It is easy to expand this form to the original form. The method set name might be created by a compiler.

$M_s = (\ \{(\texttt{M1}, b_{m31})\},\ \{\}\ )$
$I = \{\}$
$[\texttt{\#P1},\ (\ \{(\texttt{M0}, b_{m00}),$
$[\texttt{\#A1},\ (\texttt{M1}, b_{m11}),\ (\texttt{M2}, b_{m21})],$
$[\texttt{\#A2},\ (\texttt{M1}, b_{m12})]\},$
$\{\}\ )]$

Figure 7: Ms, I, and method scope `#P1` in `ExampleObject`

```
1    anObject msg @ [
2      [method do: arg ... ]
3      [method e: ..]
4    ]
```

The intensive communication supports the execution of *express methods* during the communication. We will describe the notion of the *transitional method* in section 4 to provide for the execution of *express methods* during the method execution.

3.4 Inheritance

When we define an object `O` inheriting from another object `P`, we sometimes need a mechanism such that methods defined in `P` are inherited and those methods are encapsulated. The visibility of those methods should be controlled at the runtime. To realize the capability, we provide for a mechanism to create a *method scope* whose visible set is a set of all method sets defined in a superclass. The `scope` statement is introduced in order to replace the visible and invisible sets of M_s with ones of such a *method scope.*

An example shown in Figure 6 defines the `ExampleDerived` object inheriting from `ExampleObject` defined in Figure 4. The `sset` keyword followed by symbol name `#P1` is added in the declaration of superclass `ExampleObject` in line 2 so that all methods defined in the superclass are included in a method scope denoted by `#P1`. The initial $M_s = (V, IV)$, I, and method scope `#P1` are shown in Figure 7. At the object creation, method $M1$ is only visible.

To show how to change M_s of the

```
 1 [class Vehicle
 2   (supers   Object)
 3   (vars     peer myValue)
 4   (include  #normal)
 5 ]
 6 [mset #negotiating method Vehicle negotiation: hisValue
 7   "both robots initiate negotiation at the same time"
 8   if hisValue > myValue then
 9     reply #accept
10   else
11     reply #notaccept
12   endif
13 ]
14 [mset #normal method Vehicle negotiation: hisValue
15   reply #accept
16   transit run-gave;
17 ]
18 [mset #normal transitional method Vehicle run
19   (vars    a)
20   "moving action is here"
21   myValue ← Random new;
22   a ← peer negotiation: myValue @ #negotiating;
23   if a == #accept then
24     "moving action is here"
25     peer goahead &
26   else
27     "waiting for peer's action"
28   endif
29 ]
30 [mset #normal local method Vehicle run-gave
31   "Action is moving"
32   "waiting for peer's action"
33 ]
34 [mset #normal method Vehicle goahead
35   "moving"
36 ]
```

Figure 8: Programming in the Proposed Facilities

`ExampleDerived` object, suppose that the object performs for `M1`. After the **scope** statement taking `#P1` is issued in line 7, M_s is changed as follows:

$$M_s = (\{\ (\texttt{M0}, b_{m00}),\ [\texttt{\#A1},\ (\texttt{M1}, b_{m11}),\ (\texttt{M2}, b_{m21})],\ [\texttt{\#A2},\ (\texttt{M1}, b_{m12})]\},\ \{\})$$

That is, method $M0$ is only visible at this point. In line 9, the **scope** statement taking no arguments is issued. This means that M_s is changed to the original one:

$$M_s = (\{\ (\texttt{M1}, b_{m31})\ \}, \{\})$$

4 Transitional Method

In communication among autonomous concurrent objects, an object needs to receive and perform for a message while a method of the object is executed. In this case, the state of the object may be changed and the object's behavior is changed according to those changes. We propose the notion of the *transitional method* to describe such a program. An example is shown below:

```
 1 [transitional method Example normal-run
 2   [protect ...]
 3   ...
 4   [protect ...]
 5   ...
 6 ]
 7 [method Example abnormal-run
 8   ...
 9 ]
10 [method Example negotiation:
11   ...
12   transit abnormal-run
13 ]
```

In this example, the `normal-run` method is defined with the keyword `transitional` in line 1. Such a method is called *transitional method*. In a *transitional method*, the execution of the method

```
 1 [class Lock
 2    (include   #free)
 3 ]
 4 [mset #free method Lock lock
 5    exclude #free
 6    include #lock
 7 ]
 8 [mset #lock method Lock unlock
 9    exclude #lock
10    include #free
11 ]
12
13 [class Buffer
14    (vars   ...)
15    (include   #notfull)
16 ]
17 [mset #notfull method Buffer put
18    ...
19 ]
20 [mset #full method Buffer get
21    ..
22 ]
```

Figure 9: `Lock` and `Buffer`

may be suspended and the control is changed to another method except for the execution of the `protect` statement. In other words, during the execution of a *transitional method*, all members of the *visible set* are included in the *intensive set*.

For example, when the `negotiation:` message arrives during the execution of method `normal-run`, the execution is changed to the `negotiation:` method. When the `transit` statement in line 12 is issued, then the execution is changed to the `abnormal-run` method instead of resuming the execution of method `normal-run`. If the execution of method `negotiation:` is terminated without issuing the `transit` statement, the execution of method `normal-run` is resumed.

5 Examples

In this section, two programming examples are shown to demonstrate the capabilities of our proposed communication facilities.

5.1 Negotiation Protocol

The Autonomous Mobile Robots example described in section 2 is programmed in Figure 8. In this program, two method sets are defined, i.e., `#negotiating` and `#normal`. As you can see, no

```
 1 [class LockBuffer
 2    (supers   Lock   sset #visible #normal
 3              Buffer  sset #visible)
 4    (scope #visible)
 5 ]
 6 [after method LockBuffer lock
 7    scope #normal
 8 ]
 9 [after method LockBuffer unlock
10    scope #visible
11 ]
```

Figure 10: `LockBuffer`

explicit state variables are programmed to control the behavior of a method, and programming by using the facilities proposed is very simple. Moreover, it is easy to modify the behavior of a method. For example, if the `#stopping` state is added in the object, the action of the `negotiation:` method in the `#stopping` state is programmed as follows:

```
1 [mset #stopping method Vehicle negotiation: hisVal
2    ...
3 ]
```

5.2 Inheritance Anomaly

If a language supports the *after method* mechanism like in CLOS, the visible set helps the programmer to overcome some of the *inheritance anomaly* problems in concurrent objects [1, 7, 8]. In the inheritance anomaly problems in concurrent objects, inheritance can not be employed due to the synchronization constraints. For example, if we want to create an object `O` inheriting from another object `P`, it is sometimes impossible because the constraints of the methods execution in `P` can not be changed in `O`. Here we program one of inheritance anomaly problems[7, 8].

> We have already the `Lock` and `Buffer` objects shown in Figure 9. Now we want to program the `LockBuffer` object inheriting from both `Lock` and `Buffer` objects. The behavior of `LockBuffer` is as follows: `put` and `get` messages are accepted till the `lock` message arrives at `LockBuffer`. After the execution of method `lock`, the execution of methods `put` and `get` are post-

poned until the `unlock` method is executed.

Figure 10 shows the programming example. Two method scopes `#visible` and `#normal` are defined. All methods in `Lock` and `Buffer` are included in the `#visible` method scope while all methods in `Lock` are included in the `#normal` method scope. The `lock` and `unlock` methods are defined as *after method*. Those methods control the M_s method scope of the object shown in lines 7 and 10. Though two methods are defined in `LockBuffer` but those are not totally redefined.

6 Concluding Remarks

Our major motivation in this paper was to develop a significant communication mechanism for programming autonomous objects. At a glance, it seems that the *visible set* is almost the same as the *enable set* [9]. The major difference is that several actions of a method are defined in different method sets. By defining method sets, we can program several method's actions depending on an object's state across the inheritance hierarchy. Moreover, communication among autonomous objects is safely and easily programmed by using the *visible* and *intensive sets*.

Examples shown in this paper give us the demonstration of the capabilities of the functionalities proposed in this paper. We believe that the *visible* and *intensive sets* and its related communication and inheritance facilities are not only for a specific language but also adapted into other concurrent object-oriented programming languages. We are currently implementing those facilities on a reflective language called AL-1[3, 4] to realize them.

Acknowledgements

The author would like to thank Professor Mario Tokoro for comments and suggestions on the draft of this paper.

References

[1] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X. Rousset de Pina. A Synchronization Mechanism for Typed Objects in a Distributed System. *Sigplan Notice*, Vol. 24, No. 4, pp. 105–107, April 1989.

[2] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 7, pp. 80–112, January 1985.

[3] Yutaka Ishikawa. Reflection Facilities and Realistic Programming. *Sigplan Notice*, Vol. 26, No. 8, pp. 101–110, August 1991.

[4] Yutaka Ishikawa and Hideaki Okamura. A New Reflective Architecture: AL-1 Approach. In *The Second OOPSLA 91 Workshop on Reflection and Metalevel Architecures*, 1991.

[5] Yutaka Ishikawa and Mario Tokoro. A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation. In *Proceedings of OOPSLA-86*, pp. 232–241, September 1986.

[6] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. In *Proceedings of OOPSLA-90*, pp. 289–298, October 1990.

[7] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Inheritance in Concurrent Object-Oriented Language. In *7th Conference Proceedings Japan Society for Software Science and Technology*, pp. 65–68, 1990.

[8] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. A. Agha, editor, *To be published in a forthcomming book on Concurrent Object-Oriented Computing*. 1992.

[9] C. Tomlinson and V. Singh. Inheirtance and synchronization with Enabled-Sets. In *Proceedings of OOPSLA'89*, pp. 103–112, October 1989.

[10] Yasuhiko Yokote and Mario Tokoro. Experience and Evolution of ConcurrentSmalltalk. In *Proceedings of OOPSLA-87*, pp. 406–415, October 1987.

[11] Akinori Yonezawa, editor. *ABCL An Object-Oriented Concurrent System.* MIT Press, 1990.

A Formal Description of Dynamic Method Scoping

Let a method be represented by a pair of (n, b) where n denotes a method name and b denotes its body. The pair is called *method name-body pair.* Let a *method set* be a set of method name-body pairs and other method sets. A *method set* is denoted by a symbol name. For example,

$$[\texttt{\#msym}_1,\ \{(mm_1,\ bb_1),\ (mm_2,\ bb_2),\ [\texttt{\#msym}_2,\ \{(mm_1,\ bb_1)\}]\}]$$

is a method set whose name is "$\texttt{\#msym}_1$" and members are method name-body pairs "$(mm_1,\ bb_1)$", "$(mm_2,\ bb_2)$", and method set "$[\texttt{\#msym}_2,\ \{(mm_1,\ bb_1)\}]$".

Let *visible set* V be a set of method sets and method name-body pairs. Let *invisible set* IV be a method set. Method scope M_s is a pair of the visible and invisible sets, i.e., $M_s = (V, IV)$. Function $vf : M_s \rightarrow V$ is defined to extract the visible set from method scope M_s. Function $ivf : M_s \rightarrow IV$ is defined to extract the invisible set from method scope M_s.

The basic operations over the method scope are **visible** and **invisible**. Let N be a variable over the method names and let B and C be a variable over the method bodies. Let S and T be a variable over the symbol names and let X and Y be variables over the method sets.

visible S
After the execution, new M_s becomes:
$M_s = (nV, nIV)$
where

$$\begin{aligned}
&{}^{\forall}X\ [S,X] \in \mathbf{vf}(M_s),\\
&nV = \{(N,B) \mid (N,B) \in \mathbf{vf}(M_s),\\
&\qquad \text{not}({}^{\exists}C\ (N,C) \in X)\}\\
&\qquad \cup \{[T,Y] \mid [T,Y] \in \mathbf{vf}(M_s), T \neq S\}\\
&\qquad \cup \{(N,B) \mid (N,B) \in X\}\\
&\qquad \cup \{[T,Y] \mid [T,Y] \in X\}\\
&nIV = \{IV,\ [S,\ \{(N,B) \mid (N,B) \in \mathbf{vf}(M_s),\\
&\qquad\qquad {}^{\exists}C\ (N,C) \in X\}\\
&\qquad\qquad \cup \{[T,Y] \mid [T,Y] \in \mathbf{vf}(M_s),\\
&\qquad\qquad\qquad T = S\}]\}
\end{aligned}$$

Figure 11: The semantics of **visible**

invisible S
After the execution, new M_s becomes:
$M_s = (nV, nIV)$
where

$$\begin{aligned}
&{}^{\forall}X\ [S,X] \in \mathbf{ivf}(M_s),\\
&nV = \{(N,B) \mid (N,B) \in X\}\\
&\qquad \cup \{[T,Y] \mid [T,Y] \in X\}\\
&\qquad \cup \{(N,B) \mid (N,B) \in \mathbf{vf}(M_s), (N,B) \notin X\}\\
&\qquad \cup \{[T,Y] \mid [T,Y] \in \mathbf{vf}(M_s), [T,Y] \notin X\}\\
&nIV = \{[T,Y] \mid [T,Y] \in \mathbf{ivf}(M_s),\ T \neq S\}
\end{aligned}$$

Figure 12: The semantics of **invisible**

Then, the semantics of the "**visible** S" and "**invisible** S" operations is given in Figure 11 and Figure 12.