Shared Data Management Mechanism for Distributed Software Development Based on a Reflective Object-Oriented Model

Masakazu Hori[1], Yoichi Shinoda and Koichiro Ochimizu[2]

[1] INTEC Systems Laboratory Inc.
ISL Bldg., 3-23, Shimoshin-Machi, Toyama 930, Japan
[2] School of Information Science,
Japan Advanced Institute of Science and Technology
Asahidai 15, Tatsunokuchi, Ishikawa 923-12, Japan

Abstract. In this paper, we propose a new object-oriented mechanism to manage shared data in distributed software development with following features.

1. *Workspace Manager Object* and *Artifact Object* manage the range of responsibility for a software engineer's task and control data sharing.
2. *Autonomous Mediator Object* supports negotiation relevant to the modification of shared data and coordination for the negotiation.
3. Each object has a meta-object. By the mechanism, it is possible to dynamically select available actions based on a variety of situations.

In the software development environment having these features, a software engineer can work having the only knowledge about the range of responsibility for his task and the relationships with other engineers who share data with him. In addition, the environment provides a mechanism to change policies flexibly in a cooperative work style for data sharing and modification of shared data.

1 Introduction

Shared data plays an important role at cooperative works in distributed software development. A software engineer uses shared data to form a common understanding with other engineers for the completion of his task. The way how to support data sharing and modification of the shared data is one of the key factors that affect the progress of engineers' task. Therefore we should analyse an actual work style and reflect it to a software development environment.

In a distributed software development, we should guarantee each software engineer[3] to work independently, and coordinate their works if necessary. If an engineer must understand all the details of the development, the overhead of the extra work would become extremely large. Therefore, we need a model and an environment to support cooperative work such that an engineer can work having the only knowledge about the range of responsibility for his work and

[3] We call software engineer as engineer in short from here.

the relationships with other engineers who share data with him. Based on this idea, we propose a new architecture to manage shared data.

In the present style of distributed development, an engineer spends considerable time for managing shared data as well as completing his work. For example, assume a designer asks a programmer to develop a program with a given program specification. Then, the specification becomes a shared data between them. The specification is likely to receive continuous modifications, for example, to correct a bug in the module interface. These modifications must be approved by all the members under the influence. This work style increases the overhead, particularly in a distributed environment.

We take into account the following three points to manage shared data.

1. Shared data changes as time passes.
   When an engineer asks another engineer for a development task, some data, for example, specification, is shared. The contents and the range of shared data change subsequently according to the progress of the development. We need a mechanism that detects such modifications and provides support for the engineer to correctly perform the development task regardless of the modifications.
2. Modification requests on shared data are processed by a human engineer.
   If an engineer requests to modify shared data, related engineers should decide how to deal with the modification through negotiation. We therefore need a mechanism to support such negotiation, and then reflect the results to the shared data. Unlike a strict transaction system, it is seldom that multiple engineers request modifications of a same data simultaneously. In this sense, we need a mechanism to control the modification of the shared data that is biased toward the better human-computer interaction, rather than the control based on a serializability theory [1].
3. We need to select a proper method for the modification of shared data according to the work status of relevant engineers at the time of modification.
   A method to modify shared data changes according to the degree of the influence to other engineers. For example, we may need only to notify how to modify a specification if the modification is to take place at the beginning of a development, because the modification yields little or no influence to programmers' task. However, at the end of the development, a designer and programmers must need to discuss whether they should modify it, and in some case, they might put it off. We therefore need a mechanism to select a proper method from multiple candidates to deal with a modification request.

To summarize, the three issues in the management of shared data in a distributed software development environment are: (1) the contents and the range of shared data change dynamically and frequently, (2) we should consider a human factor when we modify shared data, and (3) the work style of an engineer depends on various situations during development.

In this paper, we propose a set of solutions for these issues.

1. We introduce a concept of a *distributed workspace* managed by the corresponding *workspace manager object*. Here, by object we mean a software module which has an internal state, and performs an operation on the state by the acceptance of a message in a form of method invocation. The distributed workspace corresponds to the range of responsibility for an engineer's work. We define *shared workspace* as the space overlapped by several distributed workspaces.
2. An *autonomous mediator object* supports negotiation related to the modification of shared data and coordination for the negotiation.
3. Each object has a meta-object. By the mechanism, it is possible to dynamically invoke required actions when changes in work style are introduced. These actions include modification of shared data, modification in range of sharing, and selection of engineer's activity based on his work status.

We use *computational reflection* [2] to realize dynamic change and selection of a method invocation. A system with a reflective architecture is capable of computing the configuration and the process of computation of itself. Many works have been reported on mechanisms to change the configuration of basic system dynamically, for example, in operating systems [3, 4, 5, 6, 7]. In this paper, we show that the computational reflection is also useful for a mechanism to change policies flexibly in cooperative work style (Figure 1).

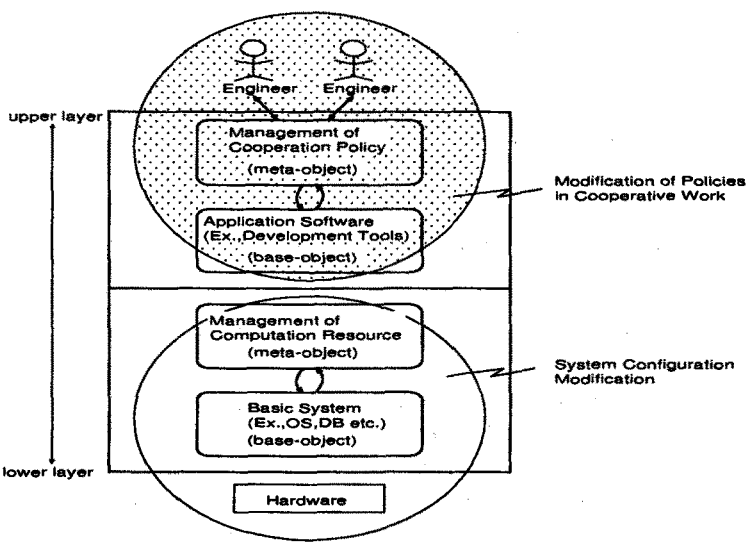


Fig. 1. Computational Reflection Applied to Modification of Policies in Cooperative Work

This paper is organized as follows. Section 2 gives the outline of the proposed object-oriented model for a distributed software development environment that concentrates on control of data sharing. We define distributed workspace as an engineer's responsibility for development, and then define a shared data as a component in a product of distributed workspaces. Based on these definitions, we introduce workspace manager, artifact object and autonomous mediator object. We will also describe roles of each object and interrelations between them. In Section 3, we analyze requirements in functionality for the workspace manager object and the artifact object which are the components of the distributed workspace. In Section 4, we consider the type of negotiation observed when an engineer requests to modify a shared data. We will elaborate the functionalities of autonomous mediator that supports negotiation based on the negotiation types considered above. Section 5 gives practical example of "*Cooperative Write*" in detail. Section 6 exhibits the outline of prototype implementation that we build to verify the feasibility of the model. Section 7 gives concluding remarks.

2 A Distributed Software Development Model for Data Sharing

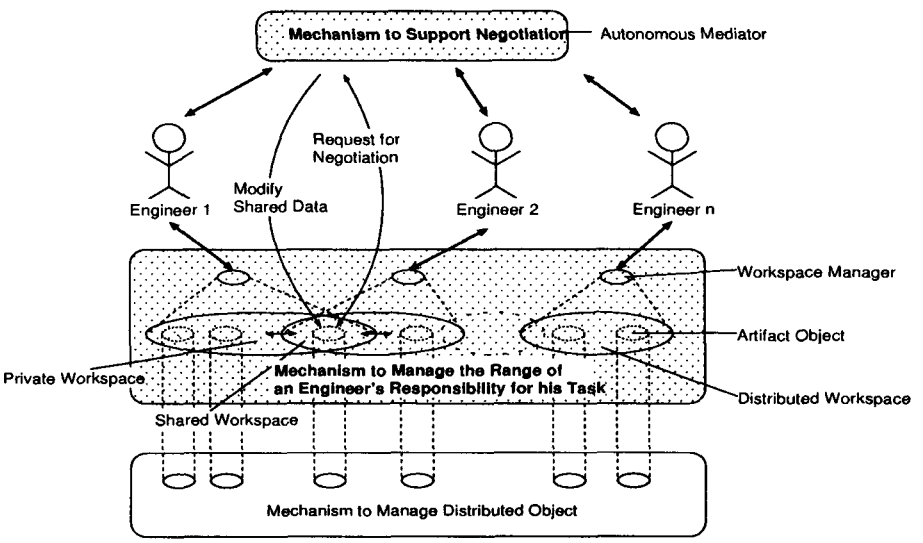


Fig. 2. The Outline of a Distributed Software Development Environment for Data Sharing

Figure 2 summarizes our view of how data should be shared, and how mechanisms to support the sharing should be placed. A mechanism for distributed object management [3, 9, 10], which guarantees distributed transparency [8], is

located in the lowest layer. The two mechanisms that we are going to propose in this paper are located above this layer. The first mechanism's role is to manage the range of responsibility for engineer's task, and another mechanism's role is to support negotiation concerning a modification of shared data. These two mechanisms are briefly discussed in the following two subsections.

2.1 Management of the Range of Responsibility for Engineer's Task

Let us consider that an engineer's task is to develop a requested intermediate artifact. We define a range of engineer's responsibility by a set of intermediate artifacts relevant to the engineer's current task. We call this range as *distributed workspace*. An *artifact object* manages a corresponding artifact that is a component of the distributed workspace. Each engineer has his own *workspace manager*, which manages his distributed workspace and controls access to artifacts in the workspace.

We define a *shared workspace* as an intersection of multiple distributed workspaces and a *private workspace* as the difference between a distributed workspace and shared workspaces.

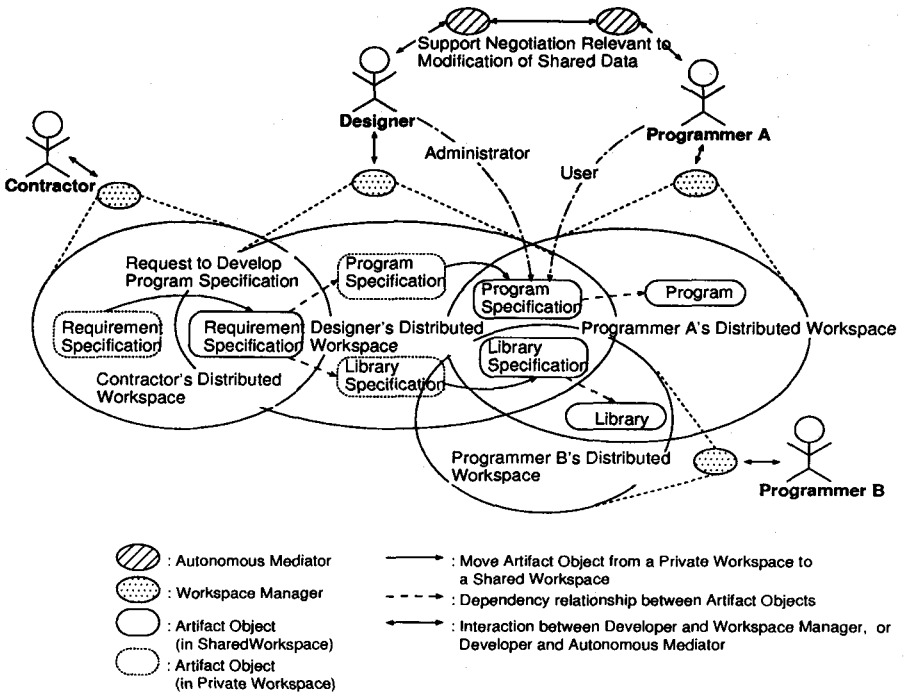
Since an artifact object in a shared workspace is accessed from multiple engineers, the object performs user-wise access control on each method invocation. An engineer accesses the artifact object through his workspace manager to cope with possible sharing situations. The workspace manager then selects a proper method for an actual access to the artifact object based on current activity status of the engineer and through negotiation of other workspace managers which shares the artifact object with.

2.2 Negotiation Support for Shared Data Modification

Shared data is modified after all the engineers, whose workspace contains a shared workspace[4] in which the shared data resides, agree with a modification request. We need not only to make a negotiation by collecting their opinions about the request, but we also have to perform a processing based on the result of the negotiation. Autonomous mediators are generated by meta of each relevant workspace manager when shared data is requested to modify, and they establish a necessary environment to support negotiation among relevant engineers. One engineer may act as a coordinator who detects a problem and coordinates relevant members if necessary. In such case, rest of the members act as negotiators to exchange their opinions.

The relationship between the mechanism described in this section and the mechanism described in Section 2.1 is as follows: (1) When an engineer requests a modification to shared data, a request to begin a negotiation is generated and sent to the mechanism that support negotiation. (2) After the negotiation completes, a method of a processing the shared data is notified to the shared artifact object to carry out the actual modification.

[4] A shared workspace and distributed workspaces which configure it change as time passes.



	Base-object	Meta-object
Autonomous Mediator	1. Ask a Question to a Developer 2. Invoke a Method in Other Object	1. Control a Method Invocation to the Base-object by a Half Protocol (State Transition Diagram)
Workspace Manager	1. Manage Artifact Objects in Workspace(Move between a Private Workspace and a Shared Workspace, and Change the Access Right at the Same Time) 2. Accept the Access to an Artifact Object from an Engineer	1. Ask the Meta-object of an Administrator's Workspace Manager for the Right to Modify an Artifact Object in a Shared Workspace 2. Generate an Autonomous Mediator to Support Negotiation
Artifact Object	1. Manage an Intermediate Artifact	1. Control a Method Invocation to the Base-object

Fig. 3. The Roles of Each Object

2.3 Dynamic Change of Work Styles by Computational Reflection

In this section, we show the role of each meta-object concerning the following two issues.

1. Dynamic change of access rights for shared data.
2. The way to deal with a request from an engineer who is not allowed to modify shared data.

Let us now explain about an assumption for the application of computational reflection to these issues.

To reflect the change of the range of engineer's responsibility for his work to an environment, we need to change access rights dynamically. In addition, we allow only one engineer to modify shared data simultaneously. This condition is considered not too restrictive, since simultaneous requests of modification of a single data by multiple engineers is a rare case in the distributed development.

We summarize the assumption about a typical work style to modify shared data in this paper.

> We assume that there exists an administrator for each shared data who has an exclusive modification access to the data. Other engineers do not have the right to modify the shared data. If they want to modify the data, they notify the administrator, and then they begin negotiation for permission of modification.

Our understanding is that it is more important to change access rights timely for an engineer who needs to modify shared data. That is, access control mechanism we discuss here has a policy that guarantee the soundness of the work performed by the engineer as much as possible. For example, if we receive a modification request from an engineer who can not be granted for immediate access, the engineer may be notified of the time when he will be granted, or the access may be granted through negotiation.

Based on these work styles, we explain the way how shared data is generated and modified according to Figure 3. The upper part of Figure 3 shows the outline of interrelations between objects. The table at the lower part summarizes the major roles for each base-object and meta-object.

- **Generation of a shared data**
  Shared data is generated when a development is requested. For example, when "designer" requests "programmer A" to develop a program, "designer" requests the base of his workspace manager to move a program specification from his private workspace to the workspace shared with "programmer A". The base of the workspace manager also notifies access right for "programmer A" to the meta of the artifact object.
- **Modification of a shared data**
  Meta of workspace manager snoops method invocations at base of the workspac manager. If a base's first attempt to modify a shared artifact object is refused by the object, meta of the workspace manager will contact the meta of the workspace manager which is owned by the administrator of the artifact object for the grant to execute the modification. For example, when programmer A requests the base of his workspace manager to modify program specification which is shared with designer, the meta of the workspace manager receives a message that the request is refused. Therefore the meta of programmer A's workspace manager asks the meta of designer's workspace manager for his right for the modification of the program specification. The

```
1. [object meta-workspace-manager
2.  (state [queue := [queue-gen <== :new]] ;;  a message queue
3.         [scriptSet := scripts]          ;;  a set of scripts
4.         [mode := ':dormant]              ;;  an execution mode
5.         ... )
6.  (script
7.   (=> [:message M R S] ;;  an arrival of a message & receiving it
8.      [queue <= [:enq [Message Reply Sender]]]
9.      (when (eq mode ':dormant)
10.       [mode := ':active]
11.       [Me <= :begin]))
12.  (=> :begin ;;  acceptance of a message & script execution
13.     ;;  dequeue one triple [M R S] from a queue
14.     (match [queue <== :deq]
15.      (is [M R S]
16.         (match (find-script M R S scriptSet)
17.           (is [Bindings ScriptBody]
18.               ;;  Start a script evaluation
19.              [evaluator
20.                <= [:do ScriptBody [env-gen <==
21.                 [:new Bindings state]] Me Group-manager evaluator]
22.                @ [cont ' [Me <= :end]]]])
23.           (otherwise
24.            (warn "~S cannot handle the message ~S" [den Me] M))))))
25.  (=> :end ;;  after a script execution
26.     (if (not [queue <== :empty?])
27.       [Me <= :begin]
28.       [mode := ':dormant]))
29.  (=> [:reject owner time M R S]
30.     (match M
31.       (is [:set-artifact a-obj a-data]
32.          [[meta owner] <= [:want-to-modify a-obj a-data]]))
33.     ...)
34.   ... )]
```

Fig. 4. Example Definition for Meta of Workspace Manager in ABCL/R2

meta of designer's workspace manager deals with the situation by selecting one of the prepared procedures. A result may vary. The modification might be allowed at the specific time, or the meta-object might prompt the designer for human intervention. If the designer judges that negotiation is necessary, the meta of his workspace manager generates an autonomous mediator which implements a protocol for the negotiation concerning the modification of the program specification.

3 Distributed Workspace

3.1 Workspace Manager

Workspace manager manages distributed workspace composed of a set of artifact objects, and it accepts requests from its owner to invoke a method in artifact object.

Several reports are published concerning a mechanism to deal with multiple objects, for example, a mechanism to invoke methods in multiple objects by one

invocation [11, 12] and a mechanism to change the behavior of objects dynamically [4]. Techniques to manage a single object belonging to several different groups by group-wise methods are also reported [3, 13]. Based on these studies, the workspace manager is designed to be capable of dealing with multiple objects for implementation of distributed workspace. That is, shared artifact object is managed by multiple workspace managers with the different policy depending on each engineer's work style.

An engineer's workspace manager keeps an attribute which indicates "*role*" of the engineer in addition to information to manage artifact objects in the workspace. It determines the range of responsibility for the engineer's work. The "*role*" can be expressed as follows:

$$role = ((artifact_1, work_1), \cdots, (artifact_n, work_n)) \quad (1)$$

where *role* means the role of an engineer, *artifact* is the name of an artifact and *work* is an operation performed on the artifact by him. Examples of *work* are "*edit*", "*modify*", "*test*", etc. We use an attribute "*role*" for access control of shared data. When we ask an engineer for a development, we submit a pair "*(artifact,work)*" to the base of his workspace manager. At the same time, the workspace manager notifies the kind of work and name of the engineer to meta of the artifact object for an establishment of access control.

We show an example for a typical role made by meta of workspace manager. Described below is a 4-step behavior of meta of workspace manager when an engineer does not have right to modify a shared data.

1. *Meta of workspace manager asks meta of an administrator's workspace manager for permission to modify.*
   When an engineer requests to modify an artifact object in a shared workspace even though he does not have the right to modify it, the meta of the artifact object notifies the meta of his workspace manager of the refusal. Then, the meta of the workspace manager requests the meta of an administrator's workspace manager to acquire the right to modify.
   Figure 4 is an example implementation of meta of workspace manager written in a concurrent reflective object-oriented language ABCL/R2 [5, 6]. The intended behavior of the meta of workspace manager is to detect a request to modify, and then asks the meta of an administrator's workspace manager for permission to modify.
   (a) Meta of workspace manager dequeues one triple [*M R S*] from the message queue(line 14). *M* is a message sent to base of workspace manager, *R* is the reply destination for *M*, and *S* is the sender object identity of *M*.
   (b) Meta of workspace manager creates a new environment, and then evaluates the body of the script[5] under the new environment by "evaluator" object (line 16 - 21).

[5] In ABCL/R2, implementation of method is called script.

(c) When the script is executed normally, a message *:end* is sent to meta of the workspace manager (line 22). However, if the method invocation is refused by an artifact object, meta of the workspace manager receives a message [*:reject ...*] from meta of the artifact object (line 29). Then, meta of workspace manager sends a message [*:want-to-modify ...*] to meta of the administrator's workspace manager to acquire the permission to modify.

2. *Meta of an administrator's workspace manager judges whether it should notify the administrator of the arrival of inquiry.*
   The administrator sets a necessary condition for this decision to meta of his workspace manager in advance. Some of the possible actions are: (1) If the request is for a specific artifact object that does not accept the negotiation, the request is immediately turned down, (2) a negotiation request will be accepted at the specified date, or (3) the negotiation will begin immediately.
3. *Meta of the workspace manager presents the request to the administrator.*
   It presents the request by the output format which the administrator chooses. For example, it displays the difference between the original data and the requested one (for example, output format from *diff* command in UNIX). Nature of the modification may also be presented at this point.
4. *Meta of the workspace manager follows an instruction of how to deal with the request given by the administrator.*
   The administrator instructs meta of workspace manager how to deal with the request based on the information presented at the previous step. Examples of instructions are *"begin negotiation"*, *"allow modification"*, *"reserve modification "*, and etc.

3.2 Artifact Object

An artifact object corresponds to each artifact produced during software development life cycle. In this paper, we treat a coarse grained object for artifact object such as specification sheet, source program, and etc. Artifact object has methods to handle artifact data, for example, generation, modification, and etc. We define artifact object as an abstract class which represents artifact. Each artifact object is defined as a subclass of the artifact object class. We treat the data of artifact as an attribute of the artifact object.

In general, we should consider the management of shared data from the views of concurrency control [1], access control [14], version control [15, 16], and configuration management [17]. Since we restrict one person at a time to modify shared data, we do not have to consider concurrency control.

Meta of artifact object controls a method invocation to the base-object. If an invocation is refused, it provides the following data to the client: (1) the name of an administrator for this object, and(2) the date if an administrator accepts a negotiation request in the future.

Figure 5 illustrates an example of access control by meta of artifact object. Meta of artifact object sends a message to "acl" object to verify if a method

```
1. [object meta-artifact-object
2.  (state ...
3.         [acl := nil]  ;;  an object to manage access rights control list
4.         [owner  := nil]  ;;  an owner of this object.
5.         [time   := nil]) ;;  time to accept negotiation
6.  (script
7.   (=> [:message M R S]
8.       ... )
9.   (=> :begin
10.      (temporary right)
11.      (match [queue <== :deq]
12.        (is [M R S]
13.            ;;  get the access right for a message M from acl
14.            [right := [acl <== [:check-right (second M) (first M)]]]
15.            (cond ((right)
16.              (match (find-script M R S scriptSet)
17.                (is [Bindings ScriptBody]
18.                    ...)
19.                (otherwise
20.                 (warn "~S cannot handle the message ~S~%" [den Me] M)
21.                  [Me <= :end])))
22.              ((t [[meta S] <= [:reject owner time M R S]]
23.                  [Me <= :end])))))
24.        (otherwise
25.         [Me <= :end])))
26.  ...)]
```

Fig. 5. Example Definition for Meta of Artifact Object in ABCL/R2

invocation is allowed for a client before it is executed (line 14). If it receives the reply of acceptance, it evaluates the message and executes the script as normal(line 17 and 18). Otherwise it does not execute it and notifies the time to accept a negotiation request to the meta of client's workspace manager. The merit of access control by meta-object is that we can localize the control in a meta-object rather than scattering it around the methods in base-objects.

4 Negotiation Support for Modification of Shared Data

4.1 Requirements for a Mechanism to Support Negotiation

We specifically address the following two requirements concerning the negotiation[6] support for modification of shared data.

1. An ability to support negotiation with a half protocol and change it dynamically.
   According to our experiences that we took part in several software development projects, we believe that we can formalize the process of the negotiation to some extent, and, thus, we should provide a mechanisms to support it. For example, we can specify the order of messages exchanged between the

[6] In this section, negotiation corresponds to one relevant to the modification of shared data.

relevant engineers, the way to acquire necessary information, the way to process the acquired data, and so on[7]. A half protocol refers to such process, and specifies the flow of an engineer's behavior depending his role during negotiation. It is a "*half*" protocol because a protocol generally specifies the flow of messages between two sites. An actual protocol is formed when two half protocols actually exchange messages during a negotiation.
In Section 4.2, we formalize the aspects of negotiation as the type of negotiation. We also present the way of dynamically replacing the half protocol in Section 4.4.

2. An ability to support coordination according to the progress state of negotiation.
Since several engineers in distributed sites negotiate, we need a coordinator, who detects a problem and coordinates engineers for it. That is, a coordinator needs to detect who meets trouble and help him to promote the progress. In Section 4.4, we will show an example that supports coordination by a progress state of negotiation and a negotiation history, which are specified by a series of state transitions.

4.2 Types of Negotiation

In this section, we investigate the type of negotiation, which corresponds to a unit in a half protocol.

We assume that a negotiation consists of the three steps: (1) gathering the necessary information for decision, (2) making a decision, and (3) post-processing that depends on the decision. We denotes the type of such negotiation by eight tuple.

$$\langle S, G, I, R, O, gather, decide, action \rangle \tag{2}$$

S is a set of negotiation states. When all the necessary data in a state is gathered, negotiation moves to a new state. G denotes a set of possible selections for decision. For example, when we are to vote for "*admit*" or "*reject*", these two items become the elements of G. I is a set of information gathered to make a decision. We can either ask an engineer or invoke a method of the object in order to acquire the information. R is a set of decisions. For example, when we select our opinion from G, the selected elements from G become elements of R. O is a set of messages which are sent as a result, for example, "*notify result*", "*begin other type of negotiation*", "*execute a tool*", and etc. A function "*gather*" gathers the necessary data to specify the subject of decision. It is invoked after the data for S and G are determined.

$$gather : S \times G \rightarrow I \tag{3}$$

The decision R is determined by function "*decide*" with inputs I and G.

$$decide : I \times G \rightarrow R \tag{4}$$

[7] We do not consider formalization of the decision process itself in this paper.

O is generated by function "*action*" with input R. After the O is generated, the negotiation moves to a new state.

$$action : R \times S \rightarrow O \times S \tag{5}$$

During the evaluation of these three functions, an engineer may intervene with the process. For example, an engineer may select his choice from presented candidates to make a decision.

4.3 Autonomous Mediator

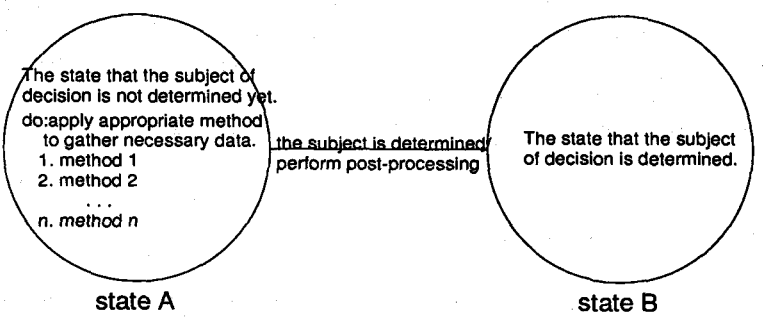


Fig. 6. The Notation of a Half Protocol Managed by Mediator

We refine autonomous mediator[8] as a model for implementation of the negotiation types that contains half protocols based on the type of negotiation discussed in the previous section.

By "*autonomy*", we mean that the mediator is capable of automatically selecting new method of execution from a set of prepared methods if a previous method fails. By this autonomy, mediator deals with an abnormal situation, for example, when it could not get the reply from an engineer or an environment by a deadline.

Mediator autonomously performs a processing by a state transition based on the rule in Figure 6. It moves from state A to state B via three steps: (1) applying a selected method i. If it fails, another method is selected and applied, (2) making a decision based on the gathered data, (3) performing a post-processing according to the decision. These three steps correspond to evaluating the three functions "*gather*","*decide*", and "*action*" in order.

Let us now examine the role of base and meta of the mediator. Base of the mediator implements all the necessary methods to act autonomously. It interacts with outside by an inquiry of an engineer and the method invocation of other object. Meta of the mediator controls autonomous processing by the base-object

[8] We call mediator in short from here.

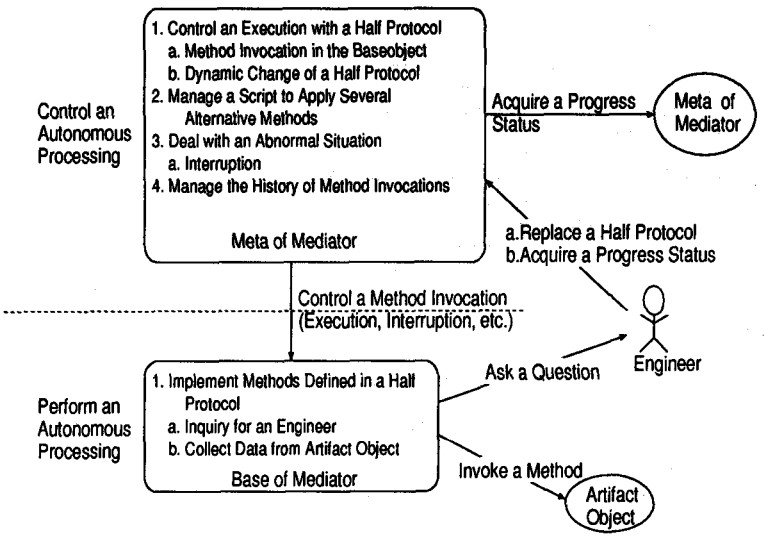


Fig. 7. Reflection Mechanism for Mediator

from the following four points. (1) It manages a half protocol implemented as a state transition diagram. It controls an execution in the base-object. It can change a half protocol dynamically by exchanging the state transition diagram. (2) It manages a script to apply several alternative methods prepared for an error. (3) When it is interrupted, it performs various processing depending upon the situation. (4) It manages a history of method invocation to supply the necessary data for the support of coordination.

4.4 The Internal Mechanisms of Autonomous Mediator

A Mechanism for Dynamic Replacement of Half Protocol The reason to replace a half protocol dynamically is that we probably come across the situation different from which we anticipate at the beginning of negotiation. One of the possible cause for this is that we insufficiently specify each parameter for the type of negotiation defined by tuple (2) in Section 4.2. We suppose the following situation as for the insufficiency of each parameter. For example, S may not include a necessary state, or items in G may not be sufficient, or O might include an undefined post-processing.

The Mediator contains a mechanism for dynamically changing half protocol by one of two ways. Each case is described below using the code fragment for meta of the mediator in Figure 8.

```
1. [object meta-mediator
2.   (state ...
3.    [abuf := nil] ;;  associated list for mem-no
4.    [c-state := ':get-decision]   ;;  current state
5.    [c-func  := ':gather]         ;;  current function
6.    ;; state transition diagram
7.    [state-trans
8.     := '((:get-decision ((:gather ((:set-reply mem-no)))
9.                          (:decide :get-decision)
10.                         (:action ((yes  :change)
11.                                   (no   :nochange)))
12.                         (:next-state ((yes  :end)
13.                                       (no   :end)))))))]
14.  (script
15.   (=> :end
16.      (cond ((equal c-func ':gather)
17.             ( ... ))  ;;  process for ':gather function
18.            ((equal c-func ':decide)
19.             ( ... ))  ;;  process for ':decide function
20.            ((equal c-func ':action)
21.             ( ... ))) ;;  process for ':action function
22.      (if (not [queue <== :empty?])
23.          [Me <= :begin]
24.          [mode := ':dormant]))
25.   (=> [:exchange-state-trans new-st]
26.      ...
27.      [state-trans := new-st])  ;;  set a new state transition diagram
28.   (=> [:add-user new-user]
29.      [users := (append users (list new-user))]
30.      (rplacd (assoc 'mem-no abuf) (1+ (cdr (assoc 'mem-no abuf)))))
31.   (=> [:start-new-mediator mediator-type]
32.      [[mediator-gen <= [:new mediator-type Me]]]) ;;  generate a new mediator
33.   ...)]
```

Fig. 8. Dynamic Replacement Mechanism of a Half Protocol for Meta of Mediator

1. *Replacement of a half protocol*

This method also accompanies of data used in the half protocol. A variable "*state-trans*" in line 7 - 13 denotes a state transition diagram which defines a half protocol. A variable "*:get-decision*" in line 8 represents the name of a state, and the method name for the functions "*gather*","*decide*", and "*action*" are specified.

Mediator exchanges a half protocol when the method [*:exchange-state-trans* ...] in line 25 is invoked. After it verifies that no other method is executing, it sets new state transition diagram as the half protocol (line 27)[9]. As an example of accompanied data modification, we use the case that the participants of the negotiation increases. This case is notified to the mediator by the message [*:add-user* ...] in line 28, which increments the number to wait for the end of *:set-reply* method in "*gather*" function by one.

[9] Note that, in our research, we aim to realize a mechanism that meta of mediator reuses the history of negotiation and changes the structure of a state transition diagram dynamically, when it changes a half protocol. However, at the current implementation, meta of mediator exchanges a half protocol and retries negotiation from the beginning.

```
1. [object meta-mediator
2.  (state
3.   [c-state := ':get-decision] ;;  current state
4.   [c-func  := ':gather]       ;;  current function
5.   [monitor-obj := [monitor-gen <== :new]])
6.  (script
7.   (=> :begin
8.      (match [queue <== :deq]
9.        (is [M R S]
10.           [monitor-obj <= [:monitor Me :begin [M R S]]]
11.           (match (find-script M R S scriptSet)
12.             (is [Bindings ScriptBody]
13.                [evaluator
14.                 <= [:do ScriptBody [env-gen <== [:new Bindings state]]
15.                    Me Group-manager evaluator]
16.                 @ [cont ` [monitor-obj <= [:monitor Me :end [M R S]]]
17.                      [Me <= :end]]]))))))
18.  (=> :current-status?
19.     !(list c-state c-func))
20.  (=> :access-history? @ R
21.     [monitor-obj <= :access-history? @ R])
22.  (=> :abort
23.     ...
24.     [c-state := ':end])
25.  )]
```

Fig. 9. Mechanism to Support Negotiation Coordination

2. *Delegation to other types of mediators*
   It is often the case that we need an assistance from other types of mediators. As the subject of negotiation is analyzed in detail, we often come across the situation which we do not anticipate before negotiation. For example, we may have an obscure point about the modification. We deal with this situation by the method [:*start-new-mediator* ...] (line 31) to delegate to other type of mediator to solve it.

A Mechanism to Support Coordination for Negotiation In this section, we discuss a mechanism to support an engineer, acting as a coordinator, to coordinate a negotiation according to Figure 9. The coordinator needs to gather the necessary information to know the progress status of negotiation. The coordinator assists an engineer if he is in trouble. For this purpose, meta of mediator keeps track of a progress status in a variable "*c-state*" and a history of method invocation in "*monitor-obj*" object. The history is taken before and after the execution of the main method (line 10 and 16). We can retrieve the history by invoking the method ":*access-history?*" (line 20). By examining the history, we can obtain the knowledge about the progress of negotiation. We can also detect whether negotiation is moving towards affirmative or negative side in a total at the time. In addition, we perceive the possibility of an engineer in a trouble if we detect unnecessary repetition and stagnant at the same state. In this case, a coordinator inquires the status for the engineer and knows his situation in more detail in order to give some advice. However, if a coordinator judges that they

cannot continue the negotiation, he requests all meta of mediators to interrupt the negotiation, or initiate replacement of a half protocol.

5 Practical Example: Cooperative Write

5.1 The Situation of Cooperative Write

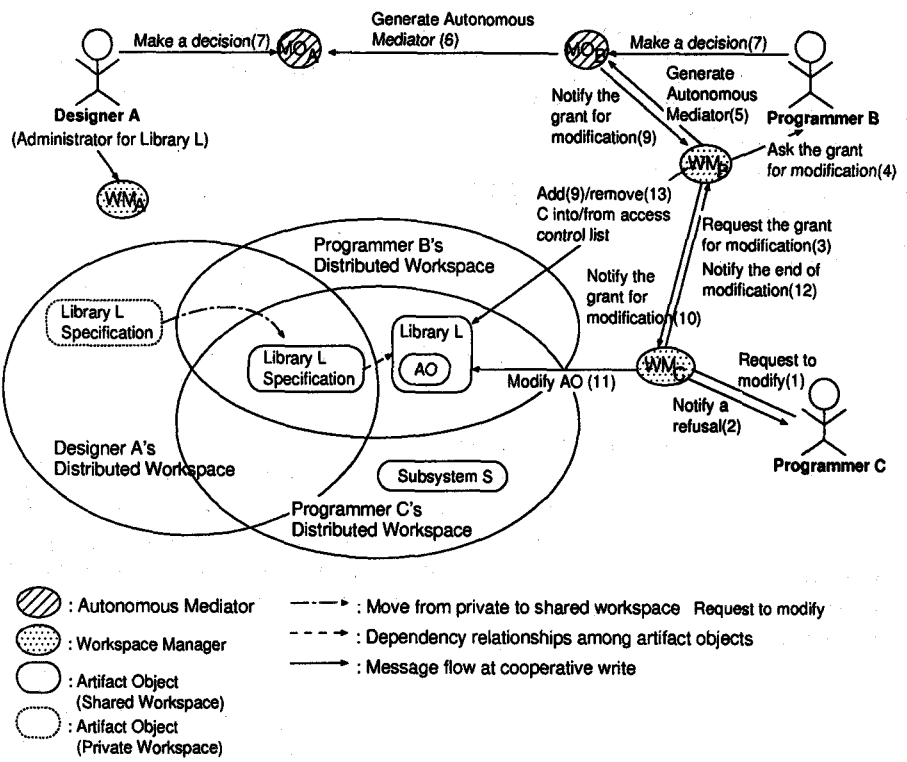


Fig. 10. Cooperative Write

We assume a project whose members develop a software in distributed places using C++ language. Consider the situation that a designer A and two programmers B and C in this project work cooperatively(Figure 10). A designer A designed the specification of library L and asked B to develop a program according to the specification. A programmer B develops library L. A programmer C develops sub-system S and uses L for his development. That is, B has the right to modify L, but C does not. In this situation, let us assume that C notices a problem in an interface of a class "*libclass*" in L. The programmer C will begin negotiating with A and B to modify the interface.

5.2 Scenario of Cooperative Write

A typical scenario of the Cooperative Write is as follows. The number in parentheses corresponds to the number in a message flow in Figure 10.

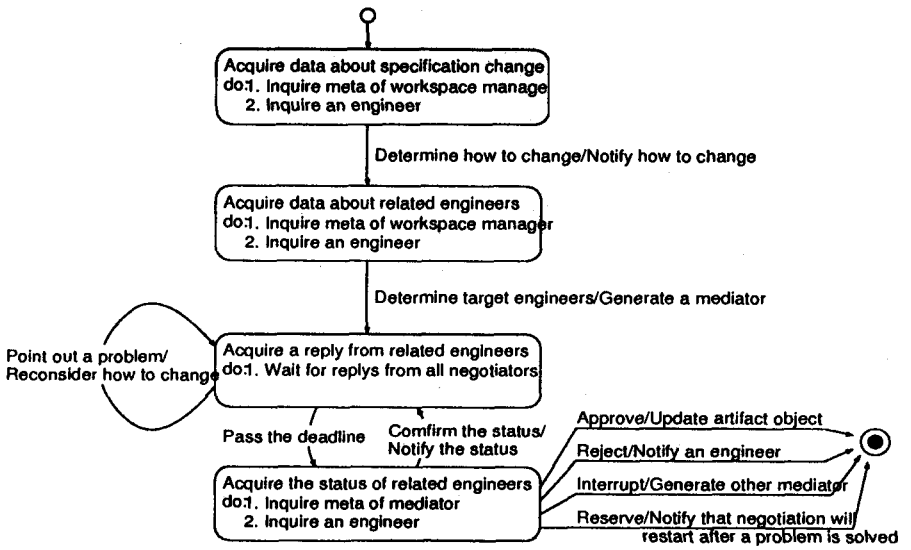


Fig. 11. A State Transition Diagram for the Mediator to Support a Coordinator

1. C reads an artifact object AO which defines the interface of "*libclass*", and then sends the modification request with a new interface to C's workspace manager WM_C (1).
2. Meta of C's workspace manager MWM_C notifies C that the modification request is rejected because C has no right to modify AO (2). MWM_C requests B, who is an administrator of AO, to allow to modify AO (3).
3. MWM_B reserves the modification request and asks B if he permits the modification (4).
4. If B permits the modification, MWM_B starts MO_B to negotiate with A because A designs the specification of AO (5). B takes a role of coordinator and MO_B assists B during negotiation. Figure 11 shows the state transition diagram in MWM_B.
5. MO_B generates MO_A to ask A's opinion (6). Figure 12 shows a state transition diagram on MWM_A. The goal for the execution by MO_B is to select a decision from 1) approve, 2) reject, 3) interrupt, or 4) reserve.
6. A and B negotiate each other through MO_A and MO_B, and finally select a decision from the four candidates described in the step 5 above (7).
7. A result of selection is notified to MWM_B (8). If the is "approve", MWM_B temporary adds C into the access control list for modification in MAO until he completes the modification (9).

8. MWM_B notifies MWM_C that C can modify AO (10), and then C modifies AO[10] (11).
9. MWM_C notifies MWM_B that C has finished the modification (12). C is removed from the access control list in MAO (13).

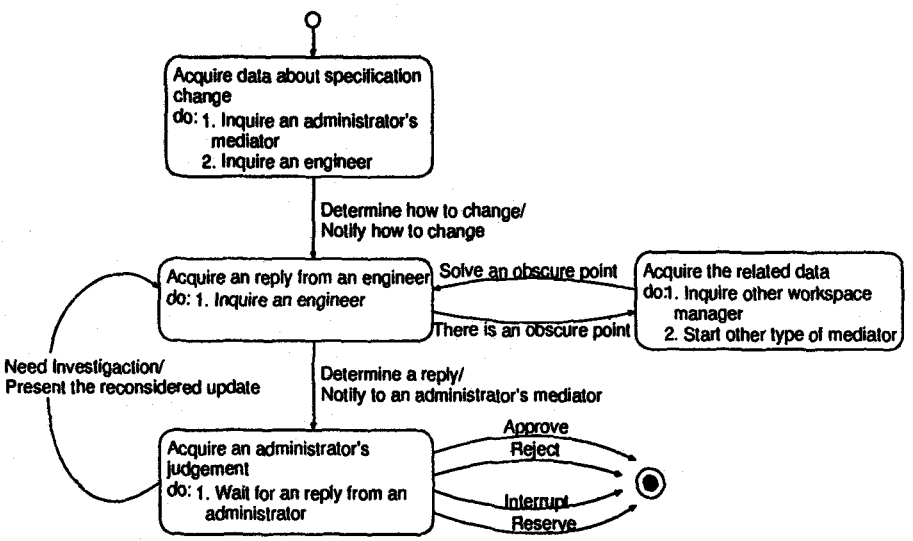


Fig. 12. A State Transition Diagram for the Mediator to Support a Negotiator

6 Prototype System Implementation

We implemented a prototype system and investigated the feasibility of the model. In this section, we report the result.

We can easily implement base/meta of object in ABCL/R2 because it is an object-oriented concurrent reflective language. ABCL/R2 is a public domain software and it runs on Sun SparcStation and SONY NEWS in our environment. Since ABCL/R2 does not provide a mechanism for (1) access to distributed object (2) object migration between different workstations, we implemented these functionalities on ABCL/R2.

7 Conclusion

We have proposed an object-oriented architecture for data sharing and modifications of the shared data in a distributed software development environment.

[10] Note that in this scenario, we dropped the following details: (1) the case that the result of negotiation between A and B is not "approve" (2) the case that C is allowed and delegated the right for modification.

Since shared data generated during distributed software development are considerable source of cooperative works, a mechanism to support the data sharing must consider human-computer interaction. We modeled a distributed workspace as a range of responsibility for an engineer's work. We have proposed a mechanism to present and modify shared artifact object jointly by the workspace manager and the object. We have also proposed a mechanism to support negotiation required for modification of shared data. We have analyzed the types of negotiation and introduced the autonomous mediator for an implementation model of the type.

We have used computational reflection to modify policies in cooperative work style. This approach has two advantages: (1) We can customize an environment dynamically according to the change of the work style, because we can select a proper method from ones which are prepared in advance, and (2) We can select a method to deal with an abnormal situation if the first method did not finish normally.

We should notice that the computational reflection is indistinguishable in the design of workspace manager and mediator object. Workspace manager changes access method to shared data and mediator collects data by multiple substitute methods.

Currently we have a plan of an experiment to apply our result to a real distributed environment. We are going to investigate the way to deal with more complex situations [19]. The model and system which we propose in this paper is considered to be a sub-system of *CSCSD* (*Computer Supported Cooperative Software Development*) "*JIZAI*" [20]. *JIZAI* integrates the support of software development and communication among engineers.

8 Acknowledgments

We thank to Hidehiko Masuhara for his kind advice for the implementation issues by ABCL/R2.

References

1. Bernstein,P.A., Hadzilacos,V., and Goodman,N.: Concurrency Control and Recovery in Database Systems. (1987), Addison-Wesley Publishing Company.
2. Maes,P.: Concepts and experiments in computational reflection. Proc. of OOPSLA'87, (1987), pp. 147-155.
3. Yokote,Y.: The Apertos Reflective Operating System: The Concept and Its Implementation. Proc. of OOPSLA '92, (1992), pp. 414-434.
4. Watanabe,T. and Yonezawa,A.: An actor-based metalevel architecture for group-wide reflection. Proc. of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), (1990). (Also in LNCS No. 489, 1991).
5. Masuhara,H., Matsuoka,S., Watanabe,T. and Yonezawa,A.: Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. Proc. of OOPSLA '92, (1992) pp. 127-144.

6. Matsuoka,S., Watanabe,T., and Yonezawa,A.: Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. Proc. of ECOOP '91, LNCS No. 512, (1991), pp. 231-250.
7. Okamura,H., Ishikawa,Y., and Tokoro,M.: Metalevel Decomposition in AL-1/D. LNCS No. 742, (1993), pp. 110-127.
8. Rodden,T., Mariani,J.A. and Blair,G.: Supporting Cooperative Applications. Computer Supported Cooperative Work, No. 1, (1992), pp. 41-67.
9. Mullender,S.J., Rossum,G.van, Tanenbaum,A.S., Renesse,R.van, and Staaveren,H.van: Amoeba A Distributed Operating System for the 1990s. Computer, May (1990), pp. 44-53.
10. OMG: The Common Object Request Broker: Architecture and Specification. OMG Document Number 1991.12.1, Revision 1.1, OMG, Draft 0, Dec. (1991).
11. Pardyak,P.: Group Comminication in an Object-Based Environment. Proc. of International Workshop on Object-Orientation in Operating System, (1992), pp. 106-116.
12. Black,A.P. and Immel,M.P.: Encapsulating Plurality. Proc. of ECOOP '93. LNCS No. 707, (1993), pp. 57-79.
13. Belkhatir,N., Estublier,J. and Melo,W.L.: Software Process Model and Work Space Control in the Adele System. Proc. of 2nd ICSP, (1993), pp. 2-11.
14. Hailpern,B., and Ossher,H.: Extending Objects to Support Multiple Interfaces and Access Control. IEEE Trans. on Soft. Eng., Vol. 16, No. 11, (1990), pp. 1247-1257.
15. Adams,E.W., Honda,M., and Miller,T.C.: Object Management in a CASE Environment. Proc. of 11th ICSE, (1989), pp. 154-163.
16. Katz,R.H.: Toward a Unified Framework for Version Modeling in Engineering Databases. ACM Computing Surveys, Vol. 22, No. 4 (1990), pp. 375-408.
17. Gallo,T., Serrano,G. and Tisatu,F.: ObNet: An Object Oriented Approach for Supporting Large, Long-Lived, Highly Configurable Systems. Proc. of the 11th ICSE, (1989), pp. 138-144.
18. Yonezawa,A., editor. ABCL: An Object-Oriented Concurrent System. Computer Systems Series. (1990), The MIT Press.
19. Araki,K., Okamura,K., Saeki,M., Miura,N., Ochimizu,K., Sinoda,Y., and Kaiya,H.: Supporting Cooperative Works Through Computer Network. Summer Workshop in Tateyama, IEICE, Vol.95, No.1, (1995-07), pp.105-112 (in Japanese).
20. Ochimizu,K., Kadowaki,C., Fujieda,K., and Hori,M.: Design of A Software Development Environment JIZAI for Distributed Software Development. Technical Report of IEICE, SS94-18, (1994) (in Japanese).