Distrib. Syst. Engng 2 (1995) 192–201. Printed in the UK

A multiversion mechanism for intra-object concurrency

Toshio Hirotsu†, Hiroko Fujii‡ and Mario Tokoro§

Department of Computer Science, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223, Japan

Received 28 November 1994, in final form 5 July 1995

Abstract. In this paper, we propose an extended concurrent object model for distributed multiuser systems called the multiversion concurrent object (MCO). The original concurrent object model is simple because it excludes internal concurrency, but this may cause long blocking on its execution during message sending and external device accesses. Thus, these blockings may decrease system performance. We introduce a multiversion mechanism to the original concurrent object model. Each MCO makes a copy of its own state, which is called a version, for executing a method that is not interleaved with other similar copies during execution. The results from concurrent execution are merged upon termination of each method. The MCO simplifies programming, since synchronization statements are not required inside its methods. Using this model for shared objects, we can construct a system in which each user can work freely without suffering from other user's unexpected actions. We present a prototype implementation of MCO and some performance evaluations.

1. Introduction

Object-oriented techniques have recently begun to be widely applied in many areas of computer science, including both applications and operating systems. High modularity of objects attained by the encapsulation and the message passing, which is a uniform communication style between objects, is the main reason for its popularity. In multiuser systems, each user accesses an object asynchronously and concurrently. Programmers have to prevent the interleaves of the multiple simultaneous access in the conventional (non-concurrent) object-oriented model. Although some expert programmers can write programs which allow high concurrency inside an object, this is too complex for the average programmer.

The (single-version) *concurrent object* model is one approach to this problem [1, 23]. In the concurrent object model, each object has a message queue and a single virtual processor; requests are handled one by one. Programmers do not have to maintain consistency inside an object with this approach, and it is simpler for constructing large complex systems. In this model, the execution of a method on an object is suspended by sending a message to other objects and accessing certain devices. This kind of suspension may become longer in distributed and multiuser systems. In applications that include user interaction, such suspension is caused by waiting for user inputs and takes a long time. We cannot predict the duration of this suspension for distributed systems because of its incomplete information of the whole system. The long suspension delays all succeeding requests in the message queue, thereby decreasing the performance of all objects.

In this paper, we propose a new object model called the *multiversion concurrent object* (MCO) which extends the conventional single-version concurrent object (SCO) to solve the problems of the long discontinuation of execution. In MCO, each object looks like a conventional concurrent object and the methods in an object are executed concurrently in safety using multiple versions of the object state. In this paper, we describe the basic concept of this MCO model and show some result of the prototype implementation.

2. Background

An object is a module that encapsulates its own local data and the operations accessing the data, and interacts with other objects by passing messages. Recently, objects have been used as components for various applications and operating systems [10]. In a multiuser distributed system, multiple users access the objects asynchronously and concurrently; this may cause inconsistencies among multiple objects. Consistency of objects can be classified into the following two categories.

- Intra-object consistency: guaranteeing consistent accesses to the internal data of an object against the multiple concurrent accesses.

† E-mail address: hirotsu@mt.cs.keio.ac.jp
‡ E-mail address: kanko@mt.cs.keio.ac.jp
§ E-mail address: mario@mt.cs.keio.ac.jp. Also affiliated with Sony Computer Science Laboratory Inc., 3-14-13 Higashi Gotanda, Shinagawa-ku, Tokyo 141, Japan. E-mail: mario@csl.sony.co.jp

0967-1846/95/040192+10$19.50

- Inter-object consistency: guaranteeing consistent orders of the conflicting executions among multiple objects.

Linearizability [7, 8] is a correctness criterion for the concurrent accesses of multiple requests to a data object. A request is represented as a pair of an invocation event and a response event. The access history H of an object is a sequence of events, and a serial history is defined, where

(i) The first event of H is the invocation event, and
(ii) Each invocation event of H except the final one is followed by a response event.

When the response of the request m_0 ($res(m_0)$) precedes the invocation of the request m_1 ($inv(m_1)$), the history H is considered to have the partial order $m_0 <_H m_1$. When all events in a history H' are equivalent to those of the correct serial history H_s and $<_H \subseteq <_{H_s}$ holds, the history H' is defined as linearizable with H_s. This is a correctness criterion for interleaved execution of multiple methods. Sequential consistency [12] is a weaker correctness criterion than linearizability for interleaved accesses to shared data by multiple processors. Only the equivalence of the events is considered for the correctness; the order of the method execution is not considered. In these earlier publications, some correctness criteria are given for concurrent execution of methods, whereas the consistency inside an object is preserved by certain primitives such as locks, semaphores and monitors. The preservation of consistency is the programmer's responsibility.

Another approach to the problems of intra-object consistency is preventing interleaving by restricting the method's invocations. A concurrent object is a self-contained object that has a message queue and a virtual processor [1, 23]. It invokes the requests one by one from the message queue that serializes the multiple concurrent requests; concurrency inside an object is not allowed. This makes it easier to write programs. However, the throughput of a system may be lower since invocations are delayed in the message queue. This (single-version) concurrent object model is used in concurrent object-oriented languages such as ConcurrentSmalltalk [20] and ABCL [22], and is also used in recent operating systems such as Apertos [21]. However, application of the concurrent object model to distributed applications will cause serious performance problems.

To preserve inter-object consistency, the notion of transactions has been introduced to objects. Local Atomicity [19], Cooperative Atomicity [13] and TMO [9] have been proposed to maintain this kind of consistency. Weihl discussed commutativity relations that are appropriate for particular recovery algorithms and the concurrency control algorithms using those commutativity relations [18]. These commutativity relations hold for all states of the object. In the Multiversion Atomic Objects [13, 15], the commutativity relations on each state of the object are proposed. The *Possible Executions* scheme is also proposed for multiversion atomic objects, which makes all possible serializable orders including the aborts of transactions [14].

In these schemes for preserving inter-object consistency, each object is assumed to handle requests one by one. This limitation helps to avoid the complexity of dealing with intra-object concurrency in programming their algorithm; however, it causes a delay for invocation of the requests. In the commutativity relations used in those atomicity properties, the return value of the execution is taken into account in order to relax the commutativity relations. The delays of the method invocation will be harmful for deciding the commutativity.

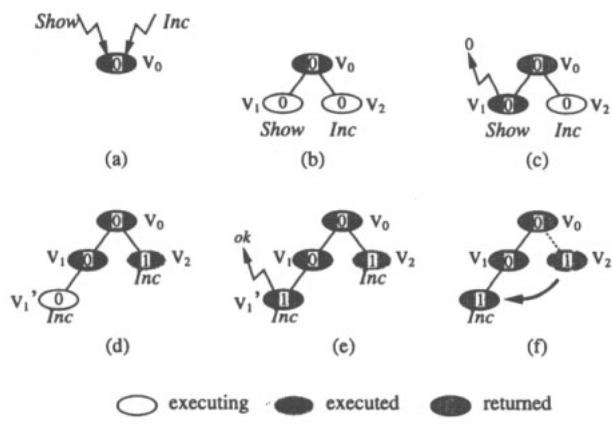

Figure 1. Basic concepts of MCO.

To recapitulate these previous works, each object satisfies a sequential specification. For models that allow concurrency inside an object, such as Linearizability, the programmer should maintain the consistency inside an object against multiple concurrent requests using some exclusion primitives like locks. However, for a model that is excluding concurrency inside an object, the consistency in it is automatically preserved but a long delay for method invocation may occur.

3. Multiversion concurrent object

We propose a new object model called multiversion concurrent object (MCO), which safely allows concurrent execution in an object. This new object model offers a kind of optimistic scheduling and reduces the delay of method invocation in the single-version concurrent object (SCO). In this model, each object consists of several copies of its own state, which are called versions, and each request is executed on the version for each request separately. In this section, we describe the concept and mechanism of the MCO.

3.1. Basic concept

The purposes of MCO are:

- **to reduce the influence** of a method execution on other executions caused by blocking during the execution, and
- to preserve consistency inside an object automatically without programming mutual exclusion.

We will first describe the basic action of an MCO. We will use the simple counter as an example.

When a message arrives in the MCO, the object selects a suitable version to execute that request and creates a new version as a copy of it. The request is executed on the new version. When another request arrives on executing the previous requests, another new version is created for the request and the request is executed separately on it. For example, if two messages of *Inc* and *Show* arrive simultaneously, the versions are created for executing each request as shown in figure 1(b). This means that multiple concurrent accesses can be executed simultaneously without causing interference among them on an MCO. These multiple separated versions are useful for preventing interference among multiple concurrent executions. However, a method execution may not reflect the result of the other method executions. To reflect the result of a method execution to the other succeeding executions, the MCO should check whether the executions infringe on its sequential specification before it returns the reply back to the sender of the request. The sequential specification means that the object executes a method used on the result of all the preceding method executions. For the sake of checking the specification, each object maintains a linear tree of the versions called a *linearized version history*. It is equivalent to the serially executed history of replied requests. Before the reply of each request, the version on which that request is executed is merged with the linearized version history by re-executing the request on the tail of the linearized version history. This mechanism is called *merging*. In the example of a simple counter object, when a *Show* method has finished prior to the concurrent *Inc* request, the version that the *Show* method is executed on is added to the linearized version history (figure 1(c)). After that, when the concurrent *Inc* method is executed, the MCO creates a new version v_1' as the copy of the tail version v_1 of the linearized version history, and re-executes the *Inc* request on the newly created version (figure 1(d)–(e)). With the MCO making a serially executed history in this way, it may seem that there is no advantage over SCO, but there is. If the re-execution produces no difference on the internal state of the object or on the reply of the execution, the reply can be sent back to the sender without re-execution or before re-execution. In our example, the *Inc* method does not have to re-execute, but the version on which the *Inc* method has been executed must be connected at the tail of the linearized version history (figure 1(f)).

3.2. Object

In this section, we show the basic components of the MCO. An MCO consists of a set of versions, a version manager and the specification information of the object (Figure 2). A version is a copy of the state of the MCO, and has a parent version that is the origin for copying. This means that a set of versions forms a tree structure. This version tree represents the execution history of an MCO. The version manager receives a request for the MCO, selects a version suitable for executing the request, makes a copy of it to execute that request, and then assigns the request to the copy. The manager also verifies the state of the version whether the result of the finished execution satisfies

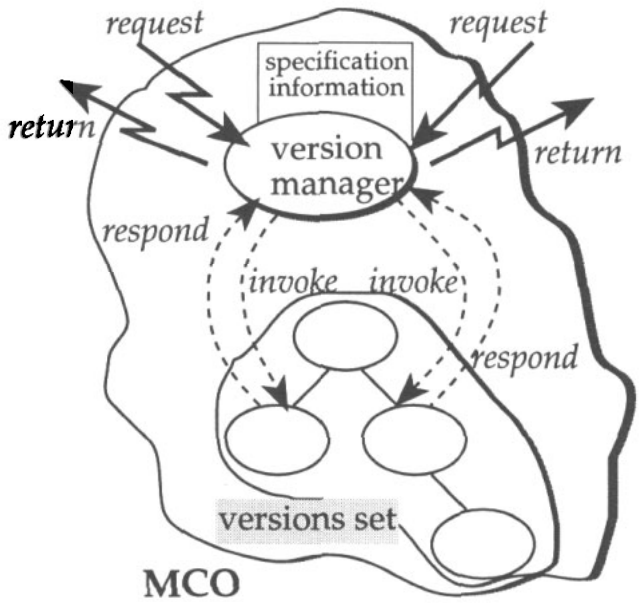

Figure 2. Structures of MCO.

the object's sequential specification. This specification information is used for deciding the requirement of the re-execution and the possibility of returning the reply ahead of the re-execution. We show the definition of the MCO below.

Definition 1. MCO (Multiversion Concurrent Object). *Let VT be a version tree, let VM be a version manager, and let SP be the specification information of the object. An MCO (Multiversion Concurrent Object) O is defined as*

$$O = (VT,\ VM,\ SP).$$

□

In this definition, a version tree *VT* can be represented as a set of versions and a set of parent–child relationships.

Definition 2. Version tree. *Let V be a set of versions, let E be a set of parent–child relations, and let v_0 be an initial version. A version tree is defined as*

$$T = (V,\ E,\ v_0),$$

where

$$V = \{v_i \mid 0 \leq i \leq n,\ v_i \text{ is a version}\},$$
$$E = \{(v_i, v_j) \mid v_i \text{ is the parent version of } v_j\}.$$

The parent version represents the previous image of a method execution. The method is executed on the new version copied from the parent version. □

The version manager verifies whether the result of a concurrent execution on the state of the object is the same as the serially executed results. If it is the same, the manager returns the reply to the sender. If not, the manager creates a new version to execute serially, and the method is re-executed on the version.

We will define some terminology for describing the algorithm of the MCO. First, we define the *linearized*

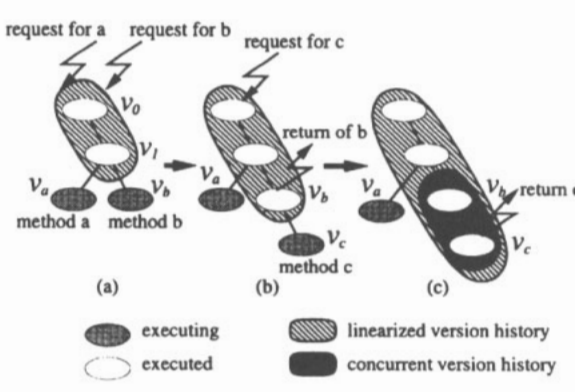

Figure 3. Serialized versions and concurrent version.

version history that represents the history of the linear executions.

Definition 3. Linearized version history. *A linearized version history can be defined as a linear tree of versions on an MCO, which is a part of the whole version tree. This tree contains all executions in which the methods have already returned a result to the sender of the request. A linearized version history LVT is represented as*

$$LVT = (V_s, E_s, v_0),$$

where

$$\begin{aligned} V_s &= \{vs_i \mid 0 \le i \le m,\ v_{si} \in V\} \\ E_s &= \{(vs_i,\ vs_{i+1}) \mid 0 \le i < m, \\ &\qquad vs_i \text{ is the parent version of } vs_{i+1}\} \end{aligned}$$

V_s is a set of versions on which finished and replied methods are executed. E_s is a set of edges that make up a linear tree. □

Next, we define the *concurrent version history* as the version tree on which the requests are executed concurrently with a request. The concurrent execution is also defined.

Definition 4. Concurrent version history. *A concurrent version history is a subtree of the linearized version history. When a method m_A has finished its execution on version v_A, the concurrent version history of the version v_A is a* ***subtree of linearized version history from the parent of v_A to its tail. When this subtree is not empty, all the executions in the subtree are considered as concurrent executions with*** *the execution of method m_A.* □

We depict a linearized version history and concurrent version history in figure 3. In this figure, three requests of method execution, *a*, *b*, and *c* arrive. The request of *b* arrives before the completion of the execution of *a*, and is executed on the version v_b that has the same parent as version v_a executing method *a* (figure 3(a)). The linearized version history is the subtree between the initial version v_0 and the version v_1 at this moment. When method *b* has finished and the result is sent back to the sender, the linearized version history grows until version v_b (figure 3 (b)). Finally, when method *c* has been executed and its result is sent back, the linearized version history and concurrent version history of v_a grow to v_c as depicted in figure 3 (c).

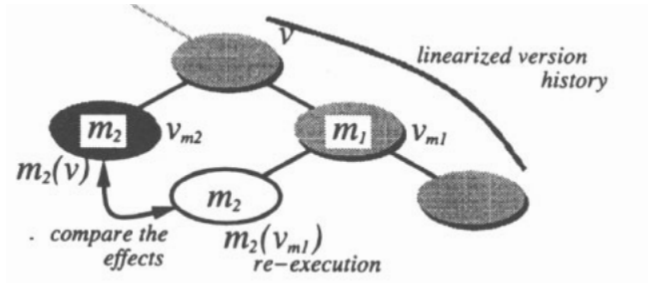

Figure 4. Mergeability relation.

3.3. Mergeability

Mergeability is a criterion to decide the difference between a result from execution of one method compared to the result of execution from another method (figure 4). This relation implies whether the re-execution causes different results at the internal state of the object and the reply of the execution. This is included in the specification information on the object. We now introduce some notation to describe mergeability. The notation of $m(v)$ denotes the execution of method m on the version copied from version v. The notation of $m(v) \rightarrow v_m$ denotes that the result version of the execution $m(v)$ is v_m. *Int_Effect*$(m(v))$ denotes the effects of the execution of method m on the internal state of the object, such as for modifying the variables. *Ext_Effect*$(m(v))$ denotes the external effects of the execution of method m, such as sending some messages and returning the result.

We first define *strong mergeability*. Consider the situation where two methods are executed on different versions with the same parent. If both the internal and external effects of re-execution on the result of the other are the same as those for the initial execution, these two methods are considered as strongly mergeable. Strong mergeability means that the re-execution is not required when merging up the versions.

Definition 5. Strong mergeability. *If two methods m_1 and m_2 are executed concurrently on each copied version of a parent version v, suppose that m_1 has finished its execution first and $m_1(v) \rightarrow v_{m_1}$. If the following relations hold, the method m_2 is defined as strongly mergeable with m_1.*

$$\begin{aligned} &Int\_Effect(m_2(v_{m_1})) \equiv Int\_Effect(m_2(v)) \\ &Ext\_Effect(m_2(v_{m_1})) \equiv Ext\_Effect(m_2(v)) \end{aligned}$$

□

By this definition, when the effects of re-execution for the method m_2 are the same as those for initial execution, the method m_2 is considered as strongly mergeable with the concurrently executed method m_1. This means that the object can merge the version v_{m_2} with the linearized version history without re-execution, where v_{m_2} is the result of the method m_2 on the copy of version v (this can be denoted as $m_2(v) \rightarrow v_{m_2}$).

Weak mergeability requires re-execution to maintain integrity of the internal state but the external effects of

re-execution to be the same as those for initial execution. Weak mergeability means that re-execution of the method causes different effects only on the internal state of the object. In this case, the object can send back the result of the method execution before re-execution.

Definition 6. Weak mergeability. *When two methods m_1 and m_2 are executed concurrently, suppose that the execution of m_1 has finished, and denote v_{m_1} by $m_1(v) \rightarrow v_{m_1}$.*

If the following relation holds, the method m_2 is weakly mergeable with method m_1:

$$Ext\_Effect(m_2(v_{m_1})) \equiv Ext\_Effect(m_2(v))$$

□

In the mergeability relations discussed above, the target (merging) version is connected to the tail of the concurrent version history. We are also able to consider the reverse order of this relation, in which the target version is inserted before the concurrent version history. This case is defined as *reverse-order mergeability*. For example, this mergeability is useful for when the execution of a method that does not change the internal state finishes later than the concurrent executions.

Definition 7. Reverse-order mergeability. *When two methods m_1 and m_2 are executed concurrently on a copied version of v, suppose that execution of m_1 has finished first and $m_1(v) \rightarrow v_{m_1}$. After that, when execution of m_2 has finished and $m_2(v) \rightarrow v_{m_2}$ holds, reverse order mergeability can be defined as follows:*

If the following relations hold, the execution of m_2 is strongly reverse-order mergeable with the execution of m_1.

$$Int\_Effect(m_1(v_{m_2})) \equiv Int\_Effect(m_1(v))$$
$$Ext\_Effect(m_1(v_{m_2})) \equiv Ext\_Effect(m_1(v))$$

If the following relations hold, the execution of m_2 is weakly reverse-order mergeable with the execution of m_1.

$$Ext\_Effect(m_1(v_{m_2})) \equiv Ext\_Effect(m_1(v))$$

□

3.4. General classification of the methods

Different mergeability can be defined for each different object and it depends on the characteristics of each application. We offer the general mergeability relations between the method groups that are generally classified by the action of each method. The criteria of this classification are based on two points. One is whether the object propagates the internal state to the others, and the other is whether the object modifies the internal state. The method that does not modify the internal state but propagates to the other objects during its execution is called type *E (export only)*. The method that modifies the internal state during the execution but does not propagate to others is called type *M (modify only)*. The method called type *ME (modify/export)* is also defined. It modifies the internal state of the object and propagates it to the others. The method that does not modify and does not propagate the internal state of the object is called type *N (none)*. Table 1 lists the methods. This information about the internal state can be derived by analysing each method. We define the general mergeability relations between these method groups. The reverse-order weak mergeability is not used, because it would cause cascading re-execution after the insertion point. Table 2 shows how the mergeability relations are defined. Each combination shows whether the method type at the top is mergeable or non-mergeable with the method type at the left. For example, the black circle in the leftmost column in the second row shows that the methods classified as type *E* are strongly mergeable in reverse order with the methods classified as type *M*. This relation can be substantiated by the fact that the execution of a type *E* method can merge with the linearized version history only if the version that executes the type *E* method is inserted before the execution of the methods classified as type *M*.

These classifications of the method type can be decided both at design time (statically) and at execution time (dynamically). In order to decide mergeability at the design time, an analysis of the method source code, which is a kind of data-flow analysis, will be required at the compilation time to classify each method. The language support for defining mergeability is preferable to enhance the relation suitable to each application. In order to decide the mergeability at the execution time, each object has to reserve the message log with the versions as the trace of the execution. A comparison of the state of the versions with the parent version shows whether the method modifies the internal state, and checking the message log reveals whether the method exports the state externally.

Table 1. Method classification.

	modify	not modify
export	ME	E
not export	M	N

Table 2. General mergeability relations.

	E	M	ME	N
E	◎	◎	◎	◎
M	●	○	×	◎
ME	●	○	×	◎
N	◎	◎	◎	◎

◎— Strongly mergeable
○— Weakly mergeable
●— Reverse strongly mergeable
×— Non-mergeable

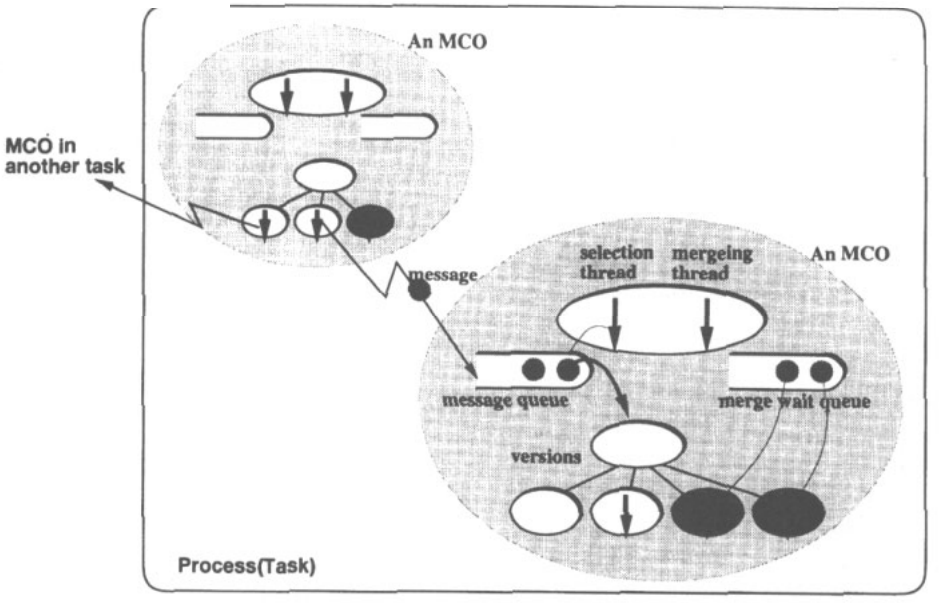

Figure 5. Prototype system of the MCO.

3.5. Basic actions of the multiversion concurrent object

In this section, we precisely describe the algorithm of the MCO, including how each mergeability relation is used. When a method satisfies weak mergeability and the object has returned the result before re-execution, such re-execution cannot be aborted. The version manager delays the next version selection and the validation until the end of such re-execution to prevent aborting that execution. In the algorithm shown below, we assume that the effects of message sending on other objects can be rolled back. We denote the checks on whether the re-execution after the return of the result with weak mergeability is still running by *WeakRedo()*. If such re-execution is running, *WeakRedo()* returns *true*, otherwise it returns false.

[The algorithm of the MCO]

(i) **[Version Selection]** When a request arrives, the version manager proceeds with the following steps.

(a) If *WeakRedo()* == *true*,
version selection is delayed until the end of re-execution, and the request is kept in its message queue.

(b) If *WeakRedo()* == *false*,
a new version is created as a copy of the tail of the linearized version history, and the request is assigned to the created version. The newly created version becomes a child of the tail of the linearized version history. The invocation of the message is recorded in the message log.

(ii) **[Method Execution]** The method corresponding to the request is executed on the assigned version. The execution of the method changes the internal state of the version and sends the requests to the other objects. Sending the message and receiving the reply during the method execution is recorded in the message log.

(iii) **[End of Execution]** When the execution of the method has finished, it notifies the version manager. This notification corresponds to a request for merging to the version manager, and the version manager keeps the request in its request queue.

(iv) **[Version Merging]** If *WeakRedo()* == *true*, the version merging is delayed, otherwise the version manager processes the version merging for each finished execution in its request queue one by one. The version merging process is shown below. The returning of each method is recorded in the message log.

(a) If the concurrent version history is empty,
the manager connects the merging version to the tail of the linearized version history, and returns the result to the sender.

(b) If the method executed on the merging version is strongly mergeable to all methods executed in the concurrent version history,
the manager connects the merging version to the tail of the linearized version history, and **returns the result to the sender.**

(c) If the method executed on the merging version is strongly or weakly mergeable to each method executed in the concurrent version history,
the version manager first returns the result to the sender. After that, it re-executes the method that has been executed on the merging version to the tail of the linearized version history. If some messages are sent to other objects during the initial execution, the messages for erasing the effects of those messages are sent before re-execution.

(d) If the method executed on the merging version is strongly reverse-order mergeable with the method executed on top of the concurrent version history,
the manager inserts the merging version before the concurrent version history, and

returns the result to the sender.

(e) If there is a method in the concurrent version history that is not mergeable with the method executed on the merging version,

the manager creates a new version as a copy of the tail of the concurrent version history, and re-executes on it the method that has been executed on the merging version. If some messages are sent to other objects during the initial execution, the messages for erasing the effects of those messages are sent before the re-execution.

An MCO externally returns the same result as serial executions of this algorithm. The MCO basically executes each request on the tail of the linearized version history, and return the result after it becomes the tail of the linearized version history. It is clear that the result of these executions is the same as that for serial execution. Mergeability means that the effect of the method execution causes the same external effects during re-execution. The result is returned before re-execution only when the execution is mergeable to the finished concurrent execution. In this case, re-execution causes the same external effects as the preceding execution, and the result of execution is the same as that for serial execution.

4. Prototype implementation

We have implemented a prototype system of MCO on SUN SPARC workstations running SunOS 4.1 with the Sun Lightweight Process (Sun LWP) library (figure 5). In this section, we show some preliminary evaluations of MCO. The goal of this evaluation was to understand the characteristics of the MCO with respect to the number of requests to execute the methods, and evaluate the blocking time by waiting for the reply from the external object during a method execution. These characteristics are important to know in finding suitable applications for MCO. We created a simple sample object that is a kind of counter, and fed many different types of request sequences into it. Each version of this sample object uses about 20 bytes for its own data area, and about 80 bytes for management information. The object has three methods, which are *'refer,' 'add,'* and *'mod_refer.'* The *'refer'* method does not modify the object's internal state but propagates it to the others. It is categorized as type *E*. The *'add'* method modifies the internal state but does not propagate it. It is categorized as type *M*. The *'mod_refer'* method modifies the internal state and propagates it to the others. It is categorized as type *ME*. Execution of the *'mod_refer'* method can cause duration of blocking. This blocking delays simulated waiting times for the replies from the other objects and waiting time for external device accesses. Here, we supposed that this blocking duration is the same for re-execution. We also implemented the SCOs to compare with the MCOs, and we injected several random sequences of requests to both MCO and SCO. The reason we selected SCO is its similarity to MCO in programmability. There are other object models that allow concurrency inside an object, but they require that all mutual exclusions be written down.

Table 3. Basic costs on MCO.

Object creation	Version creation (including selection)	Version creation (without selection)
5177.0 μs	270.8 μs	197.2 μs

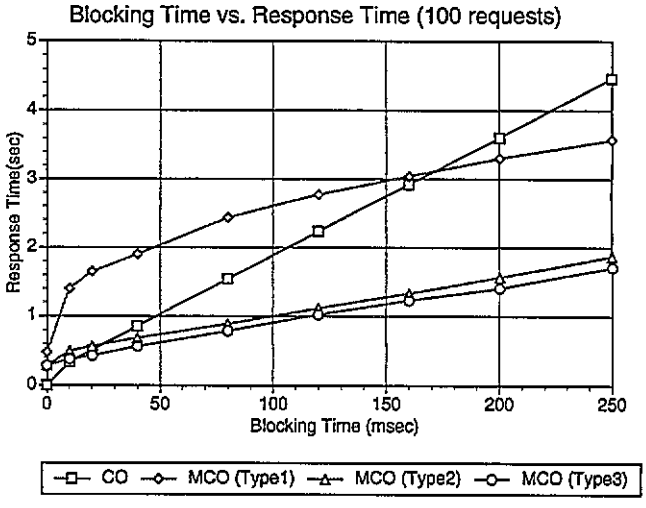

Figure 6. Relation between blocking time and average response time.

We implemented the following three types of policies on version selection and merging in the MCO to explain the effects of re-execution.

Type 1. The MCO executes methods concurrently without restriction. Multiple re-executions with non-mergeable methods are allowed in this policy. This algorithm was explained in the previous section.†

Type 2. The MCO prevents multiple re-executions with non-mergeable methods at the same time. This restriction avoids the problem of the concurrent re-executions causing another re-execution because of the conflicts between them.

Type 3. The MCO prevents multiple re-executions, and multiple ME-type method executions at the same time on the version selection.

First, we evaluated the costs of an object creation and a version creation. We show the average of 5000 measurements in table 3. The cost of object creation includes the time required for creating an initial version, allocation and initialization of memory, assignment of an object id, and creation of an execution thread for the version manager. We measured two kinds of costs for version creation. One includes the time of searching for the parent of the new version, memory allocation, initialization, and thread creation. The other includes all costs excepting searching.

Next, we measured the effect of blocking delays during a method execution. We fed in several sequences of a

† When the re-execution of method with weak mergeability is still running, the version selection and merging are suppressed in the algorithm. This is not considered the restriction as discussed here, rather it is a basic requirement for the algorithm.

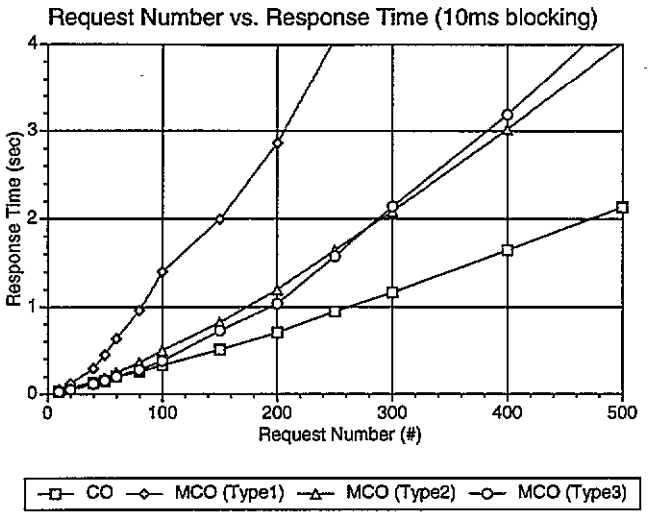

Figure 7. Number of requests and average response time (10 ms blocking, burst requests).

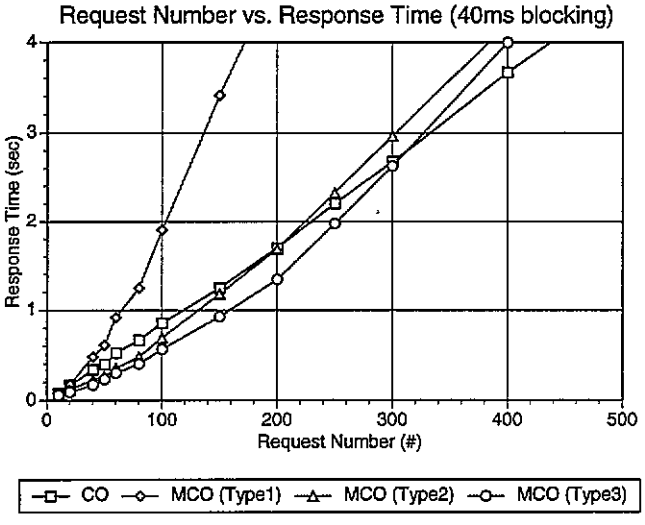

Figure 8. Number of requests and average response time (40 ms blocking, burst requests).

hundred requests in which three kinds of request were arriving at the rate of 1:1:1 on average. The relation between blocking time and the average response time is depicted in figure 6. MCO with type 1 policy shows the same performance as SCO at 180 ms of blocking time. All of the useless blocking time is consumed by the overhead of re-execution at this point. The results of type 2 and type 3 policies indicate the benefits of restricting the re-execution of the methods and the inopportune execution. From these results, we used a blocking time of 10 ms where SCOs show better performance, and a blocking time of 40 ms where MCOs show better performance.

After that, we measured the relation between the number of requests and the average response time. The relation for a blocking time of 10 ms is depicted in figure 7 and for a blocking time of 40 ms in figure 8. For comparison, the relation for a blocking time of 200 ms is also shown in figure 9. Considering the mechanism of MCO, there must be a point where MCOs consume all the wasteful blocking caused by SCOs waiting for replies. This is shown as the intersection of the graphs of MCO

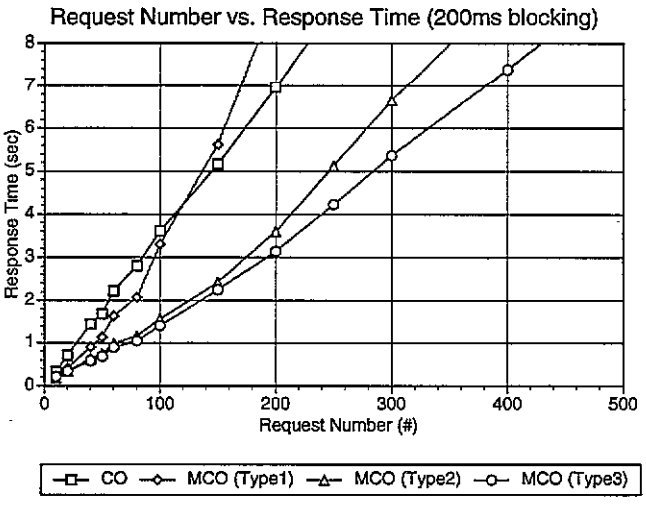

Figure 9. Number of requests and average response time (200 ms blocking, burst requests).

and SCO. Where the blocking time is short (figure 7, the MCO shows a greater cost for a lower number of requests. When an MCO has longer blocking time during its method execution (figure 8), the MCO with a type 2 or type 3 policy shows better performance than SCO in some condition for these experiments. When MCO has much longer blocking time (figure 9), MCO with a type 1 policy results in better performance in some conditions, and type 2 and type 3 policies show better performance in every condition.

5. Discussion

In this section, we discuss the characteristics of MCO based on the results of the evaluations in the previous section, and compare our approach with previous work.

5.1. Characteristics and performance of MCO

As shown in figure 6, MCO with a type 1 policy gives better performance when the blocking delays during the *'mod_refer'* method become longer than 180 ms. This blocking duration cannot be predicted because it depends on the circumstance and application. The type 1 policy, which allows unlimited re-execution, is acceptable when the method requires a long time for synchronization. An example is waiting for some inputs from users in multiuser applications, such as CSCW. While the type 1 policy requires larger costs for its execution, it still has an advantage in that it makes many possible trees of the executions. These trees can be used for deciding mergeability based on the results of the executions, and for recovery. The type 2 and type 3 policies that restrict concurrency are also considered useful. MCO does not sacrifice its performance even if the blocking duration is longer. The duration of blocking in a method execution is considered as the sum of delay on message passing and the execution time on the receiver object. We are estimating the minimum blocking duration to be more than 20 ms from the response time measured in our prototype systems. This implies that MCO is especially useful in distributed systems where the delays of message passing are long.

Figures 7 to 9 show that MCO has some limits on its performance for a particular number of requests. This means that the wasteful blocking delays during a method execution are consumed by the re-executions at this point. These limits are decided from the length of blocking duration in a method, and MCO cannot improve performance when that blocking is too short. Nevertheless, the performance of MCO in the worst cases can be bounded to that of SCO, if it controls the version selection mechanism to reduce the wasteful invocation and re-execution by the help of the information that can be obtained from MCO itself. In the MCO model, the degree of concurrency is easy to control, because the policy of that control is separated from each method execution into the 'version selection' policy.

To sum up these discussions, the general characteristics of MCO are that it is stable for a long blocking time during a method execution and increase of the requests. For future applications, the delays due to message passing and waiting for the services at other objects may become long and unpredictable where each service may be accomplished by communication with the other services by message passing. In applications which support users' jobs in distributed and multiuser systems, such as CSCW, a long blocking time in a method may occur because of waiting for users' responses. The stability of MCO will be useful in these situations. We can make programs without considering the delays in each method and the congestion of the requests. Each MCO seems to be a single-thread object on its program. This makes it easier to construct modular systems. We consider that MCO is especially useful in constructing applications for distributed and multiuser systems.

There still remain two problems with MCO. One is to cancel effects of messages which have been sent during a method execution at the re-execution time. The other is the high cost of version creation and re-execution. MCO invokes methods optimistically, so re-execution will be inevitable to maintain the integrity of the object by rollbacks and redo. When re-executing a method, the effects of the previous execution should be cancelled before re-execution. We assumed that each message sending can be rolled back and that the wiping out of a sent request requires a cost similar to its original sending. In the MCO model, the version tree works as the history of the state, and the object is able to keep all traces of the execution on an object by reserving the log of messages. The rollback in the first half of the assumption can be accomplished using these traces of the object executions. It will be useful to combine with the transaction mechanisms [3, 5] to simplify these rollbacks. One reason for the back half of this assumption is that erasing the effects are low cost only by cutting the versions. The other reason is that some re-sending can be avoided if both the sent request and re-sending request are the same by checking the message log.

The cost of version creation is high for large objects. The technique of copy-on-write will be useful for reducing these costs. These schemes help to suppress the costs for version creation and re-execution. The costs of re-execution will be reduced by using the execution results to decide the mergeability. We show the static classification of methods in this paper, but these relations can be decided from the result of each execution. For example, when a user selects the 'Cancel' operation on a pop-up menu, the method opening the menu can be categorized to E-type method or N-type method. In this case, the methods that are classified to type ME may be considered as type M or N, and higher mergeability can be achieved.

We are interested in the integration of MCO with a reflection mechanism, such as that developed in AL-1/D [16, 17] and Apertos, to implement its adaptation to the environment. One extreme policy could be that it executes methods in serial like SCO, and the other end could be that it aggressively utilizes concurrency. MCO can work better in this environment by introducing mechanisms for changing its version selection policies dynamically.

5.2. Comparison with previous work

In Linearizability [7, 8], an object can invoke multiple methods concurrently. Linearizability provides the serial behaviour to an object, but programmers must pay attention to mutual exclusion among multiple concurrent method executions inside an object. The correctness of each object depends on its implementation. In MCO, the effects of a method execution are encapsulated within a version. This mechanism frees programmers from needing to know about the internal state during a method execution; programmers do not have to write statements for excluding harmful interleaving among multiple methods. An object based on Linearizability can be implemented on MIMD multiprocessor architecture using special primitives to access a shared memory [6]. The object is copied to local memory for executing a method, and linearizability is checked on writing it back to the shared memory using special hardware instructions. In this work, the detection of linearizability is based on 'read/write' primitives, and the conflict between 'read' and 'write' causes an execution failure. MCO generates a serial history of execution using the specification of the object to determine the integrity of the object, and achieves concurrency at a higher level.

Our approach is a kind of optimistic scheduling. In previous works on optimistic control such as [2, 3, 11, 4], consistency among transactions is checked based on read/write sets of data. Harmful interleaving is checked by the validation based on the read/write conflicts. One of the problems with these approaches is that nested message sending from objects is not considered. Our approach of MCO allows sending of nested messages. Another difference between the previous approaches and ours is that previous optimistic concurrency controls are based on read/write conflicts. Our MCO uses mergeability that can be determined by using the semantic information of the effect on the executions.

In some research on atomic transactions, such as Local Atomicity [19], Cooperative Atomicity [13] and TMO [9], concurrency inside an object has been excluded and the serial specification to simplify its algorithms is discussed. MCO provides a serial history to caller objects, but handles the multiple requests concurrently inside it, so these techniques can be used for atomic transactions. MCO

can benefit from those techniques because some rollbacks may be caused and transactions can be used for deciding the limits of rollbacks.

In the Multiversion Atomic Object, which is proposed by Nakajima [13, 14, 15], multiple versions are used to increase the commutativity that is decided from the result of its executions. In MCO, multiple versions are used to execute multiple requests concurrently and safely by encapsulating its execution into each version. The created versions for concurrent executions are also useful to increase commutativity relations the same as the versions used in the Multiversion Atomic Object.

In the discussions on commutativity by Weihl [18] and the Multiversion Atomic Object by Nakajima [13], they focus on the consistent order among multiple transactions that make up the bulk of processing, and they used commutativity among operations on each object to relax the ordering constraints. In MCO, we focus on the consistency among concurrent multiple executions on an object, and we introduce mergeability to decide the constraints among multiple executions of methods. Mergeability is different from commutativity because of this difference of purpose. Commutativity decides whether two methods can execute in a given order. Mergeability decides whether an execution can be serialized before or after another concurrent execution, and it uses weaker constraints than commutativity.

6. Conclusion

In this paper, we proposed an extended concurrent object model called multiversion concurrent object (MCO). It provides serial specification externally, but executes multiple requests within it concurrently. We have implemented the prototype system of this model, and revealed its characteristics through the preliminary evaluations. The result is that MCO is stable against long duration blocking of a method execution, which is caused by waiting for responses from users, accessing external devices, and sending messages to remote sites. This means that MCO is useful in applications for multiple users in distributed systems, such as CSCW. The advantages of the concurrent object model are preserved in MCO. From another point of view, this model separates the policy and mechanism of the consistency inside an object. The MCO preserves the consistency automatically, using the versions and merging. The constraints of the synchronization and the degree of the concurrency can be controlled by the policies of version selection and merging. In the future, we plan to study the integration of MCO with some atomic transaction schemes. It will not require any modification to our model. In the previous work on atomic transactions, each object is assumed to provide a serial history that our MCO provides. To improve performance, the techniques of reflection and copy-on-write can be applied to MCO.

References

[1] Agha G, Wegner P, and Yonezawa A (eds) 1993 *Research Directions in Concurrent Object-Oriented Programming* (Cambridge, MA: MIT Press)

[2] Agrawal D. Bernstein A J, Gupta P and Sengupta S 1987 Distributed optimistic concurrency control with reduced rollback *Distributed Computing* **2** 45–59

[3] Bernstein P, Hadzilacos V and Goodman N 1987 *Concurrency Control and Recovery in Database Systems* (New York: Addison Wesley)

[4] Goyal P, Narayanan T S and Sadri F 1993 Concurrency control for object base *Information Systems* **18** 167–80

[5] Gray J and Reter A 1993 *Transaction processing : concepts and techniques* (San Mateo, CA: Morgan Kaufmann)

[6] Herlihy M P 1993 A methodology for implementing highly concurrent data objects *ACM Trans. Programming Language System* **15** 745–70

[7] Herlihy M P and Wing J M 1987 Axioms for concurrent objects *Proc. 14th Ann. ACM Symp. on Principles of Programming Languages* pp 13–26

[8] Herlihy M P and Wing J M 1990 Linearizability : a correctness condition for concurrent objects *ACM Trans. Programming Language System* **12** 463–92

[9] Hirotsu T and Tokoro M 1992 Object-oriented transaction support for distributed peresistent objects *Proc. 2nd Int. Workshop on Object Orientation in Operating Systems* pp 13–25

[10] Won K and Lochovsky F H (eds) 1989 *Object-Oriented Concepts, Databases and Applications* (New York: Addison Wesley)

[11] Kung H T and Robinson J T 1981 On optimistic method for concurrency control *ACM Trans. Database Systems* **6** 213–26

[12] Lamport Leslie 1979 How to make a multiprocessor computer that correctly execute multiprocess programs *IEEE Trans. Computer* C **28** 690

[13] Nakajima T 1990 Atomicity in multiversion atomic object *PhD thesis* Keio University

[14] Nakajima T 1993 *Possible Execution in Multiversion Atomic Objects (in Japanese)* vol 10, pp 19–34

[15] Nakajima T 1994 Commutativity based concurrency control for multiversion objects *Distributed Object Management* ed M Tamer Özsu, U Dayal, P Valduriez pp 231–24 (San Mateo, CA: Morgan Kaufmann)

[16] Okamura H, Ishikawa Y and Tokoro M 1992 AL-1/D: A distributed programming system with multi-model reflection framework *Int. Workshop on New Models for Software Architecture'92 Reflection and Meta-level Architecture*

[17] Okamura H, Ishikawa Y and Tokoro M 1993 Metalevel decomposition in AL-1/D ed S Nishio and A Yonezawa *Int. Symp. on Object Technologies for Advanced Software*

[18] Weihl W E 1988 Commutativity-based concurrency control for abstract data types *IEEE Trans. Computer* **37** 1488–505

[19] Weihl W E 1989 Local atomicity properties: modular concurrency control for abstract data types *ACM Trans. Programming Language System* **11** 249–83

[20] Yokote Y 1988 A study on object-oriented concurrent programming languages *PhD thesis* Keio University

[21] Yokote Y 1992 The apertos reflective operating system: the concept and its implementation *OOPSLA '92 Proc.* pp 414–434

[22] Yonezawa A 1988 *ABCL: An Object-Oriented Concurrent System* (Cambridge, MA: MIT Press) (in press)

[23] Yonezawa A and Tokoro M (eds) 1987 *Object Oriented Concurrent Programming* (Cambridge, MA: MIT Press)