JOURNAL OF KOREA MULTIMEDIA SOCIETY, VOL. 6, NO. 4, JULY 2003

# Database Construction for Design of the Components Software by Using an Incremental Update Propagation

Am-Suk Oh and Oh-Hyun Kwon

## ABSTRACT

Engineering design applications require the support of long transactions in cooperative environments. The problem of the existing copy/update/merge approaches is that the partial effects of a committed transaction may be not part of the merged version. This paper introduces a new cooperative transaction model, which allows updates to be progressively notified or propagated into other transactions accessing the same object. To support incremental update propagation and notification, we use the term **dynamic dependency** to define the intertransaction dependency relationships among all the objects checked out from the public database. Consistency in multiple copies of the same object is achieved by a **two-phase delta-merge protocol**. Our model provides a synchronization of cooperative updates performed in several workspaces without using locking mechanisms.

**Key words:** Transaction Model, Update Propagation, Components Software

## 1. INTRODUCTION

Engineering design applications such as CAD, software design, GIS, etc., require the support of interactive transactions of long duration in cooperative environments. With long transactions, a group of users works cooperatively for a long time on the same object. What is the most important in an interactive long transaction system is the exposure of uncommitted data and the efficiency and satisfaction of the user.

Long transactions must allow a group of users to work in separate locations without waiting for each other, since it is inefficient to force transactions to wait while other transactions make access to the same object. Conventional concurrency control mechanisms thus are not suitable for cooperative applications, because of their too restricted synchronization of concurrent accesses.

Up to now, most solutions for long transactions have adopted the copy/update/merge approaches. In those existing version-merging methods, versions are independently derived from the same object in separate workspaces. After long transactions are terminated, the versions are merged together. These approaches, however, do not consider the problem of relaxed atomicity. Atomicity of a transaction requires that a merged version must contain either all the effects of a transaction or none of the effects of the transaction. Therefore, the problem of the existing copy/update/merge approaches is that the partial effects of the committed transaction may be not part of the merged version. A randomly selected partial effect of the committed transaction for making a final merged version may not be acceptable to concurrent engineers.

This paper introduces a new cooperative transaction model which supports the cooperative work of a group of users on the same object. In our model, each user has his own workspace that is

*"This research was supported by Busan Techno-Park e-biz Transformation Project for Korea Footwear Industry"*

*Am-Suk Oh is with the Dept. of Multimedia Engineering, Tongmyoung Univ. of Information Techonology, 535, Yongdang-dong, Nam-gu Busan, 608-711, Korea. E-mail : asoh@tmic.tit.ac.kr*

*Oh-Hyun Kwon is with the Dept. of Computer Engineering, Tongmyoung Univ. of Information Techonology, 535, Yongdang-dong, Nam-gu Busan, 608-711, Korea. E-mail : ohkwon@tmic.tit.ac.kr*

***not*** *isolated from other workspaces*. This is a major difference from the traditional transaction model. Our model allows transactions to progressively propagate their updates in other transactions accessing the same object. This mechanism provides cooperative updates of the shared object, in which each user may update his own local data in isolation, but also may exchange incomplete, yet stable, updates with other workspaces prior to transaction commit.

What is the most important for cooperative work is to maintain the consistency of overlapping data from multiple workspaces. We assume that the cooperative work environment consists of a public database and one or more workspaces. The consistency control in multiple copies of the same object is not performed in the public database, but in the workspaces. Our model does not use the version control method to support cooperative updating on the same object.

Our goals for cooperative work consist of the followings:

(1) unification of concurrent work that may be acceptable to the various concurrent engineers,

(2) updates in isolation at a workspace,

(3) incremental propagation in and/or notification of updates in other transactions accessing the same object, before each transaction is terminated.

We approach this problem by defining the concept of dynamic relationship, which represents the temporary dependencies among the copies being updated in several workspaces. We use the term **dynamic dependency** to categorize update propagation and notification. In case that the same object in the public database is copied by two or more workspaces, dynamic dependencies represent which copies are dependent upon which other copies. We don't use the version-merging approach to resolve update conflict because long merging times lead to longer response time. Instead, we propose a **two-phase delta-merge protocol** in which each user at a workspace independently makes modifications, with coordination among transactions being achieved by the incremental merging mechanism. This protocol handles the problem of update conflict by direct coordination between two or more workspaces.

## 2. A WORKSPACE TRANSACTION MODEL

In this section, we propose a new cooperative transaction model that supports incremental update propagation for synchronization of concurrent work. A *workspace transaction* is any unit of work that is performed independently at any workspace. A workspace transaction is an interactive transaction consisting of a set of subtasks. Once a workspace transaction copies an object to in its own workspace from the public database, other transactions may also check it out. We assume that there is no locking for workspace transactions, even if they have shared the same object with another transactions.

The cooperative workspace transaction implies that a transaction should not be forced to wait until other transactions accessing the same object are completely terminated. Cooperative workspace transactions support concurrent updates. In other words, each cooperative transaction updates its own data in isolation. However, we don't use versions for resolving update conflict. Instead, resolution of update conflict is achieved by incremental propagation among workspace transactions. To support incremental update propagation, the workspace transaction management must be enabled to propagate the update of the contents of a workspace directly in other transactions

A workspace transaction involves a set of transaction operations. Each workspace is not shared with any other transactions. A workspace transaction can read an object from the public database, update it at its own workspace, and check in the updated object to the public database.

Fig. 1 is an example of workspace transactions.

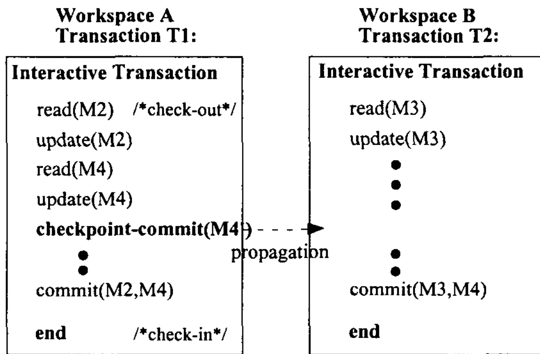



Fig. 1. A Workspace Transaction Example

A workspace transaction T1 updates two programs, M2 and M4, at workspace A. The other transaction T2 edits M3 and M4 at workspace B. Here we assume that M2 is **decomposed** into M4 and M5, where M4 is also used by M3. Let 'Ti/X' be the transaction Ti working at the workspace X. For example, T1/A is a transaction T1 at the workspace A. T1/A checkouts M2 and M4 from the public database by a **check-out** operation. When T1/A updates M4, the update may be propagated into the workspace B by processing the operation **checkpoint-commit**(M4').

A **checkout** operation copies an object from the public database to a workspace. The *read(M)* operation, shown in Fig. 1, corresponds to the checkout operation. The converse of the checkout operation is the **checkin** operation. The operation *commit* in Fig. 1 corresponds to the checkin operation. It copies back the updated object to the public database.

Since a workspace transaction requires cooperation with concurrent engineers, it is desirable for other transactions to be forced to read its partial result, which is uncommitted data, before the transaction is completely finished. The operation **checkpoint-commit** is used to save the intermediate changes of a transaction to its own workspace memory, while the transaction continues to perform its work. The idea of this paper is to use the operation *checkpoint-commit* for supporting incremental update propagation and notification. On receiving the *checkpoint-commit* operation, the transaction management system should propagate the intermediate changes checkpoint-committed from the transaction in other transactions. We assume that the operation checkpoint-commit can be issued by an user in an interactive transaction.

In the conventional versioning mechanism, a merge-conflict may occur because the versions derived from the same object are merged together, after transactions are finished. The problem of the batch merging method is that the partial effects of a transaction chosen for making the merged version may not be acceptable to the cooperative users. Therefore, we have to be able to make a single object version that is acceptable to all users.

Fig. 2 shows that the conventional version-merge method and checkpoint-commit method proposed in this paper. In Fig. 2 (a), Ma'(final version of T1) and Mb'(final version of T2) should be merged and that is serious overhead. On the other hand, in Fig. 2 (b), the final version of T1 and T2 is same. Thus, there is no more overhead to merge version.

## 3. TWO-PHASE DELTA-MERGE PROTOCOL

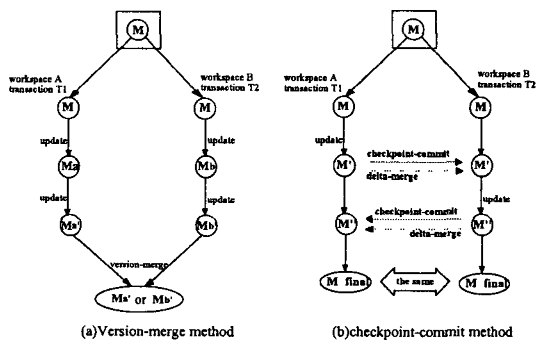



Fig. 2. Version-Merge and Checkpoint-Commit

To support the incremental merging of different updates in the same master copy, we propose a two-phase delta-merge protocol, which is similar to the two-phase commit protocol. The two-phase

delta-merge protocol is used whenever a given transaction shares the same object copied from the public database with several transactions.

### 3.1 Delta Propagation

When the checkout operation is performed, an object in the public database is copied to a specific workspace. The data set of the workspace consists of both an initial master copy and an initial empty delta. The **master copy** is the entire copy of the object checked out from the public database. In addition, we keep the **delta** (which is the difference of the modified object from the master copy). Thus, if an object participating in a duplicated relationship is updated, the delta is only propagated in other workspaces when the update is checkpoint-committed from a given transaction. We use dynamic dependencies for propagating delta in other relevant objects.

### 3.2 Resolution of Update Conflict

If an interactive transaction issues the operation checkpoint-commit to broadcast its partial updates, then the checkpoint-committed updates must be propagated in all the other transactions accessing the same object. Here is the way it works: We define a **producer** as a transaction which checkpoint-commits its partial updates, and **participants** as other transactions which access the same object. Since any transaction can become a producer, a producer is floating, i. e., not fixed. That checkpoint-commit is handled by a system component called **coordinator**.

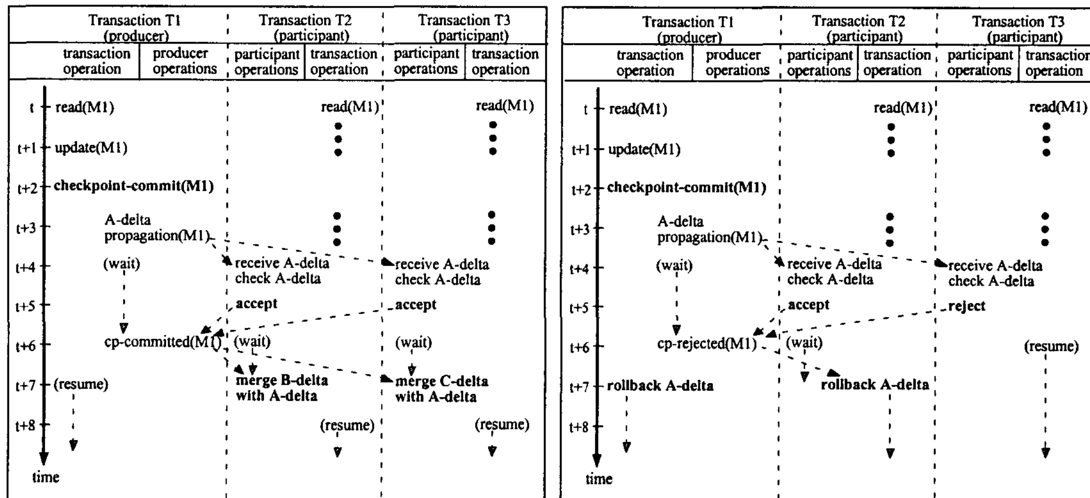
On receiving the *checkpoint-commit* request, the coordinator performs the following two-phase process:

1. It forces all participants to write the delta log propagated from the producer into their own physical log. If the forced writing is successful, each participant replies *"accept"* or *"reject"*.
2. When the coordinator has received replies from all participants, the coordinator informs each participant of its decisions. If each participant receives the "OK" message from the producer, the delta propagated from the producer is **merged automatically or manually** with the participant's delta.

This protocol notifies or propagates the checkpoint-committed data of an interactive transaction into other workspaces by using dynamic dependencies. In the case of **logical** relationships, it is sufficient to send the change notification to other workspaces. However, if the dynamic dependency is **duplicated**, the update should be propagated, and be acknowledged by all relevant workspaces. For simplicity, we will only consider the problem of update propagation.

Fig. 3 shows the two-phase delta-merge protocol for incremental merging of different updates of the same object. The operation *checkpoint-commit* issued by an interactive transaction initiates the two-phase delta-merge protocol. There are two kinds of internal operations for implementing the *checkpoint-commit*: coordinator operations, and participant operations. The coordinator operations involve *X-delta-propagation*, *cp-committed (checkpoint-committed)* and *cp-rejected(checkpoint-rejected)*, etc.

- ***X-delta-propagation*** is the first phase operation that broadcasts **delta** into other transactions accessing the same master copy.
- ***cp-committed*** is the second phase operation. The coordinator receives replies from all participants and informs each participant of it's decision, "cp-committed". If each participant received the message "cp-committed", the delta of individual participant is merged with the delta propagated from the producer.
- ***cp-rejected*** is also second phase operation. The operation cp-rejected informs each participant of "rollback" because one or more of all the participants would reject the delta propagation. If the producer receives the message *'cp-rejected'*




(a) when the delta propagation is accepted (b) when the delta propagation is rejected

Fig. 3 Coordination Between a Producer and a Participant

from the coordinator, the producer must update its uncommitted data again.

The participant operations involve *receive-delta*, *check-delta*, *merge X-delta with Y-delta*, *accept/reject*, etc.,

• ***receive-delta*** is an operation that makes a copy of the delta received from the producer at the receiver's workspace.

• ***check-delta*** is an operation that asks each user to accept or reject the propagated delta.

• ***merge X-delta with Y-delta*** is an operation that merges the receiver's delta with the delta propagated from the producer.

In this paper, it is managed by communication between the public process and the client process or among client processes that the two-phase delta-merge protocol for incremental merging of different updates of the same object. Thus, actions which each process should take are as follows:

■ **public process:**

(1) Receives request of a list of related work-space from the producer client process

(2) Notifies a list of related workspace based on the dynamic dependencies to producer client process

■ **producer client process:**

(1) Requests a list of related workspace from the public process.

(2) Receives a list of related workspace from the public process and propagates delta to participants based on it.

(3) Waits "accept" or "reject".

(4) If a producer(T1) has obtained "accept" from all participants(T2, T3), then T1 transfers message "pre-committed" to T2, T3.

(5) If a producer(T1) has obtained "reject" from one or more participants(T2, T3), then T1 transfers message "pre-rejected" to T2, T3 and rollback its own delta.

■ **participant client process:**

(1) Receives and checks a delta from a producer (T1) and notifies "accept" or "reject" to T1.

(2) If a participant(T2 or T3) will accept the delta, then T2 or T3 transfers message "accept" to T1 and waits message "pre-committed" from T1.

At this time, If T2 and T3 have obtained "pre-committed" from T1, then T2 and T3 merge their own delta with T1's delta. Otherwise(if obtain "pre-rejected"), they rollback T1's delta that has been propagated.

(3) If a participant(T2 or T3) will reject the delta, then T2 or T3 transfers message "reject" to T1 and resumes.

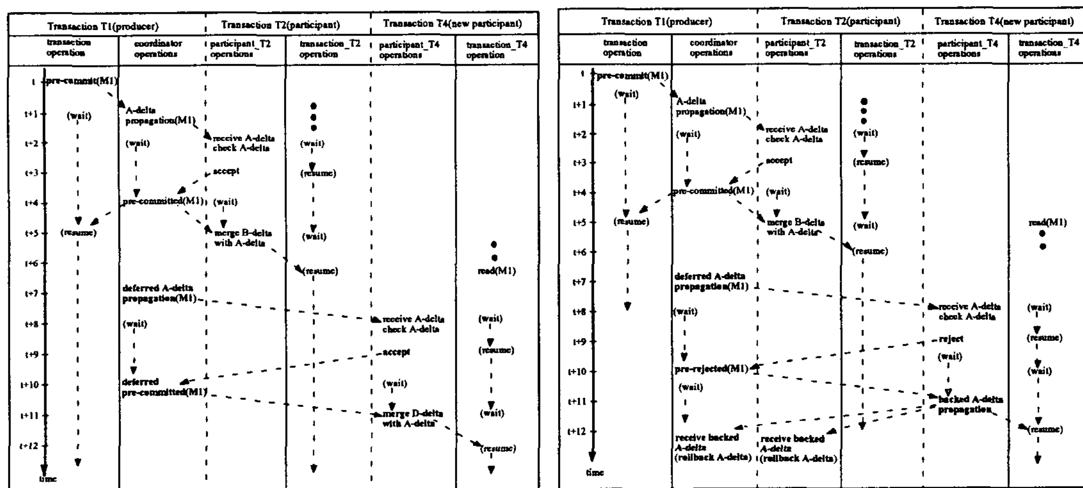
We now show how the above protocol ensures the synchronization of concurrent updates. Here we address only the deferred check-out operation and the concurrent pre-committed operation.

1. After an update propagation is completely processed, a new participant can be checked out. After T1 updates M1 and the update is propagated into T2, a new participant T4 checks out of M1 at time t+6, as shown in Fig. 4(a). At the time t+6, M1 is already updated by T1 and also the update is propagated into M1 of T2. However, M1 of T4 is still not updated. Consequently, as soon as the newly updated object M1 is checked out from the public database, the update of T1 must be also propagated in T4. This requires a deferred propagation in a new transaction. To process the deferred propagation, the coordinator should issue the operation *deferred delta propagation,* which will be sent to all new participants. If the new participant accepts the deferred propagation, the *deferred pre-commit* will be handled similarly to the *pre-committed.* If the new participant rejects the deferred delta-propagation, the coordinator issues the *pre-rejected* operation to the new participant. The new participant, on receiving the *pre-rejected*, becomes a secondary producer, for handling the back propagation of the update of M1 in T1 and T2 to rollback the A-delta. The process of *'backed delta propagation'* is similar to *'delta-propagation'*. The secondary producer differs from the primary producer in that the primary producer updates directly its own object, but the secondary producer requests to rollback only updates of other transactions.

2. When T1 and T5 issue the operation *pre-commit* simultaneously, the delta propagation can be accepted and/or rejected dependent upon the decisions of two participants. When the coordinator of T1 sends A-delta to T5 at the same time as the coordinator of T5 sends E-delta to T1, the delta propagation from T1 and T5 can be simultaneously processed without causing any problems, as shown in fig. 5.

## 4. PROCESS MODELING

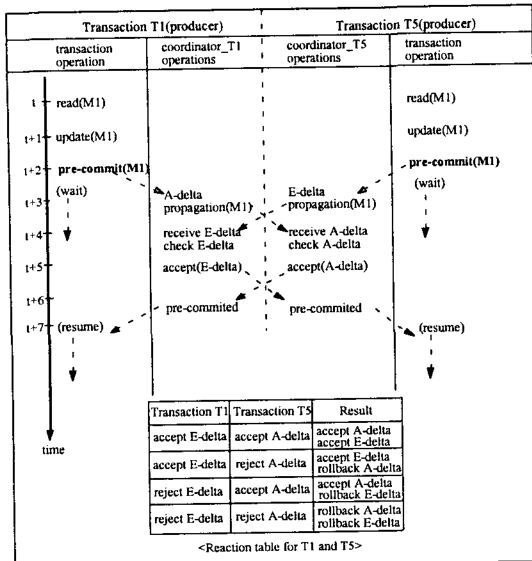
In this section we describe a process, which is




(a) When the deferred delta propagation is accepted by a new participant

(b) When the deferred delta propagation is rejected by a new participant

Fig. 4. The Process of Deferred Propagation




| Transaction T1 | Transaction T5 | Result |
|---|---|---|
| accept E-delta | accept A-delta | accept A-delta<br>accept E-delta |
| accept E-delta | reject A-delta | accept E-delta<br>rollback A-delta |
| reject E-delta | accept A-delta | accept A-delta<br>rollback E-delta |
| reject E-delta | reject A-delta | rollback A-delta<br>rollback E-delta |

<Reaction table for T1 and T5>

Fig. 5. The Process of Simultaneous Propagation between two producers

the design model for the two-phase delta-merge protocol. A public process is carried out in the server and a client process is carried out in a client. The public process manages the global state of a module which is checked-out from the public database and generates dynamic dependencies and stores them into a dependency dictionary. The client process manages the local state of a module that a workspace transaction checks-out. The public process and the client process manage the state of a module by defining actions which should be taken when a transaction operation is issued. The transition of a module from one state to another is controlled by communication between the public process and the client process or among client processes.

### 4.1 Public Process Modeling

Table 1 defines actions that the public process should take when a transaction operation issues.

### 4.2 Client Process Modeling

Table 2 defines actions that the client process should take when a transaction operation issues.

### 4.3 Algorithms

Algorithm1, algorithm 2 and algorithm 3 represent communication among processes for two-phase delta-merge protocol. A message transfer/receipt among processes is a sort of rendezvous program. When a transaction operation is issued, a communication among processes is done by the issue of a internal operation predefined in a process. A message transfer is reprersented by

Table 1. Actions of Public Process for State Transition

*action_1(check-out (wait Ù checked-out)) : generate dynamic dependencies and send a module*
*action_2(check-out (checked-out Ù checked-out)) : generate dynamic dependencies and send a module*
*action_3(checkpoint-commit(accepted) (checked-out Ù pre-committed)) : inform dependencies to producer*
*action_4(checkpoint-commit(rejected) (checked-out Ù checked-out)) : inform dependencies to producer*
*action_5(check-in (checked-out Ù wait)) : delete dependencies*
*action_6(check-out (pre-committed Ù deferred)) : generate dynamic dependencies and send a module*
*: send deferred participant ID to producer*
*action_7(checkpoint-commit(accepted) (pre-committed Ù pre-committed)) : inform dependencies to producer*
*action_8(checkpoint-commit(rejected) (pre-committed Ù pre-committed)) : inform dependencies to producer*
*action_9(check-in (pre-committed Ù wait)) : delete dependencies*
*: update the module of the public DB*
*action_10(check-out (deferred Ù deferred)) : generate dynamic dependencies and send a module*
*action_11(checkpoint-commit(accepted) (deferred Ù pre-committed)) : no actions*
*action_12(checkpoint-commit(rejected) (deferred Ù checked-out)) : no actions*

Table 2. Actions of Client Process for State Transition

action_1(check-out (initial Ù checking-out)) : receive a module

action_2(update (checking-out Ù checking-out)) : no actions

action_3(checkpoint-commit(accepted) (checking-out Ù pre-committing))
<producer client process>
: request a list of related clients from the public process
: propagate delta to participants
: wait return signal
: send pre-committed to participants
<participant client process>
: receive and check propagated delta
: send accept signal to producer
: wait pre-committed from producer
: merge delta with propagated delta

action_4(checkpoint-commit(rejected) (checking-out Ù checking-out))
<producer client process>
: request a list of related clients from the public process
: propagate delta to participants
: wait return signal
: send pre-rejected to participants
: rollback delta
<participant client process>
: receive and check propagated delta
: send reject signal to producer

action_5(update (pre-committing Ù pre-committing)) : no actions

action_6(checkpoint-commit(accepted) (pre-committing Ù pre-committing))
<producer client process>
: request a list of related clients from the public process
: propagate delta to participants
: wait return signal
: send pre-committed to participants
<participant client process>
: receive and check propagated delta
: send accept signal to producer
: wait pre-committed from producer
: merge delta with propagated delta

action_7(checkpoint-commit(rejected) (pre-committing Ù pre-committing))
<producer client process>
: request a list of related clients from the public process
: propagate delta to participants
: wait return signal
: send pre-rejected to participants
: rollback delta
<participant client process>
: receive and check propagated delta
: send reject signal to producer

action_8(check-in (pre-committing Ù initial))
<producer client process>
: request a list of related clients from the public process
: notify replace-master-copy to participants
: notify reset-delta to participants
<participant client process>
: replace master copy by master copy checked-in
: reset delta

"***send*** send-message ***to*** process" and message receipt is represented by "***receive*** receive-message ***from*** process".

```
1  Algorithm PUBLIC_PROCESS
2  begin
3  char current-state, next-state, transaction-op;
4  loop(true) {
5   wait signal
6   get client-ID, module and transaction operation;
7   fetch the module from public DB;
8   check the current-state of module;
9     while (the current-state is check-outed) {
10        if (transaction-op is 'check-out'){
11           generates dynamic dependency;
12           send module to participant client-process;
13           set next-state to check-outed; }
14        if (transaction-op is 'checkpoint-commit') {
15           receive request of related clients list from
             producer client-process;
16           send related clients list to producer client-
             process based on dynamic dependency;
17                if (delta is accepted)
18                     set next-state to pre-committed;
19                else
20                     set next-state to check-outed;
21        }
22        if (transaction-op is 'check-in') {
23         deletes dynamic dependency
24         set next-state to wait; }
25    }end while;
26  }end loop;
27  end
```

Algorithm 1. Public Process

```
1  Algorithm PRODUCER_CLIENT_PROCESS
2  begin
3  char current-state, next-state, transaction-op;
4  loop(true) {
5  wait signal;
6  get updated module and transaction operation;
7  check the current-state of module;
8    while (the current-state is checking-out) {
9        if (transaction-op is 'checkpoint-commit') {
10          send request of related clients list to public process;
11          receive related clients list from public process;
12          send delta-propagation to all participants;
13          wait return signal from all participants;
14          if (all return signals are "accept") {
15               send pre-committed message to all participants;
16               set level to 1;
17               set next-state to pre-committing; }
18          else {
19               send pre-rejected message to participants
                 accepting delta;
20               rollback delta;
21               set next-state to checking-out; }
22    }end if;
23   }end while;
24  }end loop;
25  end
```

Algorithm 2. Producer client process

```
1  Algorithm PARTICIPANT_CLIENT_PROCESS
2  begin
3  char current-state, next-state, transaction-op;
4  loop(true) {
5  wait signal;
6  receive delta-propagation from producer client-process;
7  check delta;
8  if (participant accepts delta) {
9       send accept signal to producer client-process;
10      wait pre-committed message from producer client
        -process;
11      if (receive "pre-committed" from producer
        client-process) {
12      merge own delta with delta propagated from producer;
13      set level to 1;
14      set next-state to pre-committing; }
15      else {
16          rollback delta;
17          set next-state to checking-out; }
18       }
19       else {
20       send reject signal to producer client-process;
21       set next-state to checking-out;
22       resume; }
23  }end loop;
24  end
```

Algorithm 3. Participant client process

## 5. CONCLUSIONS

In this paper, we proposed a new two-phase delta-merge protocol as an approach to incremental merging, for cooperative work. While the traditional version merging model provides well for batch merge in the public database, our model provides for incremental update propagation in

multiple workspaces, through dynamic dependency. Our model shows how the concurrent updates of the same object separately performed in multiple workspaces can be merged together before transactions are completely finished. In summary, we show how the problem of batch merging can be solved by means of the two-phase delta-merge protocol based on dynamic dependency. So far, we have focused on defining a framework of a cooperative transaction model, as an alternative approach to the existing version-merging scheme. Our further research focuses on formalizing the two-phase delta merge protocol to prove our cooperative transaction model. In this paper, we have not dealt with implementation issues, including internal operations handled by the coordinator, local transaction managers for handling workspace transactions, and recovery control mong workspaces.

## 6. REFERENCES

[1] Attie, P. C., et al, "Specifying and enforcing intertask dependencies", in Proc. Intl Conf. on Very Large DataBases, 1993, pp. 134-145.

[2] Bandinelli, S., et el., "Software Process Model Evolution in the SPADE Environment", IEEE Trans. Software Eng., vol. 19, 1993, pp. 1128-1144.

[3] Bandinelli, S., et el., "Modeling and Improving an Industrial Software Process", IEEE Trans. Software Eng., vol. 21, no. 5, May, 1995, pp. 440-454.

[4] Bhattacharya. S, et el, "Coordinating Backup/Recovery and Data Consistency Between Database and File Systems ", Proceedings of ACM SIGMOD International Conference on Management of Data, 2002, pp. 500-512.

[5] Curtis, B., et el., "Process Modeling", Comm. ACM, vol. 35, no. 9, 1992, pp. 75-90.

[6] Geogakopolous, D., et el, "Specification and management of extended transactions in a programmable transaction environment", In Proc. of the 10th IEEE Int. Conference on Data Engineering, 1994, pp. 462-473.

[7] Gillmann. M, et el, "Workflow Management with Service Quality Guarantees", Proceedings of ACM SIGMOD International Conference on Management of Data, 2002, pp. 228-240.

[8] Heinlein, C., "Workflow and Process Synchronization with Interaction Expression and Graphs, In Proc. of the 17th IEEE Int. Conference on Data Engineering, 2001, pp. 243-253.

[9] Hoppe, H. U., Zhao, J., "C-TORI: An Interface for Cooperative Database Retrieval", 5th In Conf., DEXA '94, 1994, pp. 103-113.

[10] Kim, W., Modern Database systems: Cooperative Transactions for Multipleuser Environments, Addison-Wesley Publishing Company, 1995.

[11] Mays E., et al., "A Persistent Store for Large Shared Knowledge Bases", IEEE Transactions On Knowledge And Data Engineering, Vol.3, NO.1, March 1991, pp. 33-41.

[12] Mittal, N.,"Database Managed External File Update", In Proc. of the 17th IEEE Int. Conference on Data Engineering, 2001, pp. 557-567.

[13] Perry, D. E., and Kaiser, G. E.,"Models of Software Development Environments", IEEE Transactions on Software Engineering, Vol.17, NO.3, March 1991, pp.283-295.

[14] Rusinkiewicz, M., et al., "Towards a Model for Multidatabase Transactions", International Journal of Intelligent and Cooperative Information Systems, Vol. 1, No. 3, 1992.

### Am-Suk Oh

Am-Suk Oh is an associate professor at Department of Multimedia Engineering, TongMyong University of Information Technology. He received his B.S. and M.S. degrees in computer science from Busan National University and Chung-ang University in 1984 and 1986, respectively. He received Ph.D degree in 1997 in computer engineering of Busan National University. His current research interests are Web Database, Multimedia Database and XML Database etc.
E-mail: asoh@tmic.tit.ac.kr

### Oh-Hyun Kwon

Oh-Hyun Kwon is an associate professor at Department of Computer Engineering, TongMyong University of Information Technology. He received his B.S. and M.S. degrees from Naval Academy and Naval Postgraduate School Monterey CA. in 1975 and 1980, respectively. He received Ph.D degree in 1989 in computer engineering of Chung-ang University. His current research interests are Component-Based Software Engineering, Object-Oriented Analysis Design and Implementation, System Software etc.
E-mail: ohkwon@tmic.tit.ac.kr

*For information of this article, please send e-mail to: asoh@tmic.tit.ac.kr(Am-Suk Oh)*