A New Conflict Relation for Concurrency Control and Recovery in Object-based Databases

SangKeun Lee
Department of Computer Science
Korea University
lsk@truth.korea.ac.kr

SoonYoung Jung
Department of Computer Science
Korea University
jsy@truth.korea.ac.kr

Chong-Sun Hwang
Department of Computer Science
Korea University
hwang@truth.korea.ac.kr

Abstract

This paper proposes preservation as a new conflict relation in an object-based database. By explicitly including reverse-operations which bridge the gap between concurrency control and recovery, preservation can be used independently of execution contexts to which different recovery algorithms and/or object models give rise, and further it forms a basis for formulating semantics-based recovery. This paper also makes a two-dimensional(i.e., execution contexts and operations' specifications) comparison between preservation and other conflict relations. In each execution context, our formal comparison reveals that preservation-based concurrency control achieves more concurrency than commutativity-based one.

1 Introduction

In contrast to the read/write database systems, object-based database systems can increase concurrency using semantic information of high-level operations on abstract data types. That's because the information able to be used to increase concurrency is represented in the specifications of objects. Maintaining semantic information, however, often leads to throughput degradation. Thus, the semantic information should be used, in a restricted way, in hot spot objects and the objects accessed by applications, e.g., long-lived transactions, that need the concurrency to provide necessary functionality.

Several protocols based on commutativity of high-level operations have been proposed[1,5,7,16] and weak relations, such as recoverability[3] and invalidation[14], have also been proposed in the literature. Unlike commutativity which considers both the state of an object and the return values of operations, weak relations consider only the return values of operations with respect to some particular execution order of operations. It has been shown that weak relations-based concurrency controls can achieve more concurrency than commutativity-based one from both the theoretical[3,13,14] and the practical[17] points of view. However, most conflict relations have been defined in different execution contexts to which different recovery algorithms and/or object models give rise, and thus conflict relations to be able to be used in database systems are necessarily dependent upon execution contexts. Moreover, though the concurrency controls based on some conflict relations, such as recoverability[3], need the support of special recovery systems different from the standard ones, there has been no explicit considerations of the recovery perspective within the conflict relations.

From these observations, this paper proposes a new conflict relation called *preservation*. By explicitly including the *reverse-operations* which bridge the gap between concurrency control and recovery, preservation can be used independently of execution contexts and further it forms a basis for formulating semantics-based recovery algorithms. From the performance perspective, by considering only the return values of operations with respect to some particular execution order of operations, preservation is weaker than commutativity.

The remainder of this paper is organized as follows. We describe database model in Section 2, and define(or modify) related conflict relations in Section 3. In Section 4, we propose a new conflict relation called preservation. In Section 5, we make a formal comparison between preservation and related conflict relations, then in Section 6, describe a preservation-based concurrency control. Recovery issues are discussed in Section 7, and conclusions are in Section 8.

2 Database Model

Transactions in database systems perform operations on objects. A transaction T is modeled by a tuple $(OP_T, <_T)$, where OP_T is a set of abstract operations and $<_T$ is a partial order on them. Concurrent execution of a set of transactions $T_1, \dots, T_n$ gives rise to a log E = $(OP_E, <_E)$. OP_E is $(\cup_i OP_{Ti})$ and $(\cup_i <_{Ti})$

$\subseteq <_E$. $<_E$ is a partial order on the operations in OP_E. If $O_i <_E O_j$, we say that O_j executed after O_i. The execution log is *serializable* if there exists a total order $<_S$, called a serialization order, on the set $\{ T_1, \ldots, T_n \}$ such that if an operation O_i in transaction T_i conflicts with O_j in T_j, and if $T_i <_S T_j$, then $O_i <_E O_j$. Execution of operations on different objects can be thought of as generating logs E_k for each object k such that log E is the union of all these logs. Each object has a type, which defines a possible set of states of the object, and a set of primitive operations that provide the only means to create and manipulate objects of that type. The specification of an operation indicates the set of possible states and the responses that will be produced by that operation when the operation is begun in a certain state. Formally, the specification is a function f such that

$f : S \rightarrow S \times V$, where $S = \{s_1, s_2, \ldots\}$ is a set of states, and $V = \{v_1, v_2, \ldots\}$ is a set of return values.

For a given state $s \in S$, we define two components for the specification of an operation: $return(o, s)$ which is the return value produced by an operation o, and $state(o, s)$ which is the state produced after execution of o. The definitions of $state(o, s)$ and $return(o, s)$ can be extended to a sequence of operations O. Thus, $state(O, s)$ is the state produced after execution of operations in O, and $return(O, s)$ is the union of the return values of operations in O. The specification of an object defines the set of possible sequences of operations for this object. A sequence of operations h is *legal* if it pertains to the specification of an object. Transactions access and manipulate the objects of database through operations. A transaction either commits on all objects or aborts on all objects.

The differences of recovery algorithms and/or object models employed in database systems give rise to different *execution contexts*. In order to eliminate the effects of an aborted operation, if the aborted operation's undo-operation(or compensating-operation) recorded in write-ahead log is performed, the database systems are called the UIP(Update In Place) model[10], whereas if an aborted operation recorded in the invoking transaction's private workspace(intentions list) is only removed from it, the database systems are called the DU(Deferred Update) model[10]. In the biversion object model[16], each object has two states, namely the current state and committed state. In this paper, it is assumed that execution contexts are classified into these three models.

3 Related Conflict Relations

We here define several related conflict relations on the given database model. In the DU model, the definition of forward commutativity[1] is as follows.

Definition 1 Consider two operations o_1 and o_2 executed concurrently by transaction T_1 and T_2 respectively on the same object. Operations o_1 and o_2 "Forward Commute(FC)", and denoted by (o_1 FC o_2), iff
a): for all state $s \in S$ such that $state(o_1, s)$, $return(o_1, s)$, $state(o_2, s)$, and $return(o_2, s)$ are defined,
b): $state(o_1, state(o_2, s)) = state(o_2, state(o_1, s))$, and $return(o_1, s) = return(o_1, state(o_2, s))$, and $return(o_2, s) = return(o_2, state(o_1, s))$. □

Intuitively, operations o_1 and o_2 forward commute, if both execution "$o_1; o_2$" and "$o_2; o_1$" are legal and would have the same effects on the object(i.e., the same state) and on the transactions(i.e., the same return values). In the DU model, the state of an object on which only the effects of committed transactions are reflected is visible to concurrent operations. This execution context is represented in a) of Definition 1.

In the UIP model, the definition of backward commutativity[1] is as follows.

Definition 2 Consider two operations o_1 and o_2 such that o_1's execution is immediately followed by the execution of o_2. Operations o_1 and o_2 "Backward Commute(BC)", and denoted by (o_1 BC o_2), iff
a): for all state $s \in S$ such that $state(o_1, s)$, $return(o_1, s)$, $state(o_2, state(o_1, s))$, and $return(o_2, state(o_1, s))$ are defined,
b): $state(o_1, state(o_2, s)) = state(o_2, state(o_1, s))$, and $return(o_1, s) = return(o_1, state(o_2, s))$, and $return(o_2, s) = return(o_2, state(o_1, s))$. □

Intuitively, operations o_1 and o_2 backward commute, if the execution in reverse order "$o_2; o_1$" are legal and would have the same effects on the object and on the transactions. In the UIP model, the state of an object on which the effects of both committed transactions and active ones are reflected is visible to concurrent operations. This execution context is represented in a) of Definition 2.

In the biversion object model, the definition of forward-backward commutativity[16] is as follows.

Definition 3 Consider two operations o_1 and o_2 such that o_1's execution is immediately followed by the execution of o_2. Operations o_1 and o_2 "Forward-Backward Commute(FBC)", and denoted by (o_1 FBC o_2), iff
a): for all state $s \in S$ such that $state(o_1, s)$, $return(o_1, s)$, $state(o_2, s)$, $state(o_2, state(o_1, s))$, and $return(o_2, s) = return(o_2, state(o_1, s))$ are defined,
b): $state(o_1, state(o_2, s)) = state(o_2, state(o_1, s))$, and $return(o_1, s) = return(o_1, state(o_2, s))$. □

In the biversion object model, the invoked operation can see both the current state and the committed state of an object. With this execution context, it is possible to determine whether $return(o_2, s) = return(o_2, state(o_1, s))$ condition is satisfied or not. Thus, the $return(o_2, s) = return(o_2, state(o_1, s))$ condition is in a) of Definition 3.

The definition of recoverability[3], one of weaker relations than commutativity, is as follows.

Definition 4 Consider two operations o_1 and o_2 such that o_1's execution is immediately followed by the execution of o_2. An operation o_2 is "Immediately Relatively Recoverable(RR_I)" with respect to operation o_1, and denoted by (o_2 RR_I o_1), iff for all state $s \in S$ such that $return(o_1, s)$ and $state(o_1, s)$ are defined, $return(o_2, s) = return(o_2, state(o_1, s))$. □

Intuitively, recoverability captures what happens when operations are removed from an execution log E. In contrast to recoverability, invalidation[14] defined in the DU model captures what happens when operations are inserted into an execution log E. We should note that, in the DU model, recoverability and invalidation are identical in the sense that these relations detect conflicts when one operation's return value is affected by another concurrent execution.

4 Preservation Relation

4.1 Motivation

A new conflict relation, called preservation, is motivated from two observations, one from concurrency control and another from recovery. First, most conflict relations can be used in only particular execution contexts, as shown in Section 3. Second, even though the concurrency controls based on some conflict relations are necessarily supported by special recovery systems, there has been no explicit considerations of the recovery perspective within the conflict relations. For instance, in the UIP and biversion object models, it would be impossible to use recoverability[3] without the support of special recovery systems different from the standard ones in which recovery is processed by simply performing undo-operations for aborted operations. That's because the recoverability-based concurrency control might cause incorrect state of an object or cascading aborts without the support of the context-sensitive recovery systems.

Let us consider Account object where deposit(x1) and post(x2) operations are defined. Deposit(x1) returns OK after increments the balance of account by x1, and post(x2) returns OK after interests x2the UIP model, the initial balance of account is 100, and deposit(100)'s execution by one active transaction is followed by post(5)'s execution by another active transaction, both of which return OKs. According to the definitions of recoverability and invalidation, deposit(100) and post(5) operations are relatively recoverable, and deposit(100) does not invalidate post(5). Thus, these two operations are allowed to execute concurrently and the balance of account will be 210. In the case of deposit(100)'s abort, however, the balance of account would be incorrect 110 under the standard recovery process where the undo-operation for deposit(100) is simply performed. Even if this incorrect state of Account object is detected, cascading abort of causing post(5)'s abort occurs. This phenomenon also occurs in the biversion object model where deposit(100)'s undo-operation present in intentions list executes on the current state of Account object. In order to circumvent this problem, the context-sensitive recovery, in which undo of the post(5) is followed by undo of the deposit(100) and is followed by redo of the post(5), is needed. Recoverability in itself, however, addresses no considerations of recovery and abstracts that kind of recovery from the concurrency control.

From these observations, this paper comes up with preservation as a new conflict relation. The novelty of preservation is the explicit inclusion of reverse-operations within the definition, which is intended to bridge the gap between concurrency control and recovery or execution contexts to which particular recovery systems give rise. We start by defining the reverse-operations which will be explicitly considered in determining conflicts between operations, then we define preservable operations which are allowed to execute concurrently. Lastly, we illustrate the use of preservation as the basis for conflict relations with Account object.

4.2 Reverse-Operations and Preservable Operations

In this paper, an object is extended to contain explicit reverse-operations, along with the type defining the possible state set of the object and the (regular) operations accessible to the object. A reverse-operation for an update operation on an object is defined as follows.

Definition 5 Consider one update operation, o, defined on some object. A special operation, denoted by $\bar{o}$, is a "reverse-operation" for o, iff for all state $s \in S$, $state(\bar{o}, state(o, s)) = s$. □

Definition 5 states that a reverse-operation $\bar{o}$ for an update operation o obliterates the effect of o on an object(in case of non-update operations the reverse-operations does not need to be provided). From this viewpoint, any reverse-operation must successfully complete. In addition, reverse-operations are used only for obliterating the effects of regular operations and cannot be used as regular operations themselves. Thus, in a sense, a reverse-operation is just another name for an undo-operation. The main difference is, however, that reverse-operations will be explicitly considered in determining conflicts between operations in order to consider the recovery perspective during concurrency control. That is the reason for making another name.

It should be noted that, from the definition, there may exist several reverse-operations for one update operation. For example, one form of a reverse-operation for push(x1) in Stack object may involve removing the pushed element, x1, from the Stack, and another may involve removing the top element in the Stack.

Since, in the DU model, update operations present in the intentions list have only to be removed to obliterate the effects of them on objects, the reverse-operations are practically used only in the UIP and biversion object models where the reverse-operations which are present in write-ahead log(UIP model) or intentions list(biversion object model) should be executed during undo process.

We then, with the reverse-operations, define an "immediately preservable" relation as follows.

Definition 6 Consider two operations o_1 and o_2 such that o_1's execution is follow ed by execution of o_2. An operation o_2 is "immediately preservable" with respect to o_1, denoted by $(o_2\ P_I\ o_1)$, iff for all $s \in S$, there exists $\overline{o_1}$ for an operation o_1 such that, $return(o_2, s) = return(o_2, state(o_1, s))$, and $state(o_2, s) = state(\overline{o_1}, state(o_2, state(o_1, s))$. □

The $state(o_2, s) = state(\overline{o_1}, state(o_2, state(o_1, s))$ condition, which checks if the state of an object is preserved correct after the execution of a reverse-operation for an aborted operation, is contained in Definition 6. Thus, if a reverse-operation for push(x1) in Stack object involves removing the pushed element, x1, then two push(x1) operations are immediately preservable, whereas two push(x1) operations are not if a reverse-operation for push(x1) involves removing the top element in the Stack.

With the immediately preservable relation, a commuting relation between two operations is defined as follows.

Definition 7 Two operations o_1 and o_2 "commute", and denoted by $(o_1\ C\ o_2)$, iff for all states $s \in S$, $state(o_2, state(o_1, s)) = state(o_1, state(o_2, s))$, $(o_2\ P_I\ o_1)$, and $(o_1\ P_I\ o_2)$. □

From Definition 7, we can make the following observations: First, commutativity is a symmetrical property, whereas preservation is not. Second, commutativity implies preservation.

So far, $(o_2\ P_I\ o_1)$ was used to denote the fact that o_2 was immediately preservable with respect to o_1 when o_2 was executed after o_1. We extend the concept to include the case where o_2 is preservable to o_1 in spite of intervening operations that have executed but not committed yet.

Definition 8 Consider a sequence of operations $O = \{o_1, ..., o_{n-1}\}$, which has executed but not committed yet, and an operation o_n such that for $\forall 1 \le i < n$, $o_i <_E o_{i+1}$. An operation o_n is "preservable" to o_1, denoted by $(o_n\ P\ o_1)$, iff for all $s \in S$ and for any sequence O_{sub} of O,
there exists a collection of each corresponding reverse-operation for each update operation in O_{sub}, $\overline{O_{sub}}$, such that $return(o_n, state(O - O_{sub}, s)) = return(o_n, state(O, s))$,and $state(o_n, state(O - O_{sub}, s)) = state(\overline{O_{sub}}, state(o_n, state(O, s)))$. □

Definition 8 states that the preservable relation between two operations, in case there are intervening operations which have executed but have not committed yet, is defined as the relation where the state of an object is preserved correct and o_n's return value is not affected after any subsequence O_{sub} of O is aborted.

Lemma 1 Consider a sequence of operations $O = \{o_1, ..., o_{n-1}\}$, which has executed but not committed yet, and an operation o_n such that for $\forall 1 \le i < n$, $o_i <_E o_{i+1}$. For $\forall 1 \le x < n$, if $(o_n\ P_I\ o_x)$ then $(o_n\ P\ o_1)$.

Proof: Let F denote the operations that execute between o_n and o_1. The proof is done by induction on $|F|$.

- Induction base : $|F| = 1$, i.e., F contains only one operation so $O = \{o_1, o_2\}$. Since $(o_3\ P_I\ o_2)$, the state of object will be preserved correct and the return value of o_3 will not be affected even after o_2's abortion. And since $(o_3\ P_I\ o_1)$, $(o_3\ P\ o_1)$.
- Induction hypothesis : $|F|$ = k-1, i.e., F contains k-1 operations so $O = \{o_1, o_2, ..., o_k\}$. For $\forall 1 \le x \le k$, we assume that if $(o_n\ P_I\ o_x)$ then $(o_n\ P\ o_1)$.
- Induction step : $|F|$ = k, i.e., F contains k operations so $O = \{o_1, o_2, ..., o_k, o_{k+1}\}$. Since $(o_n\ P_I\ o_{k+1})$ and $(o_n\ P_I\ o_k)$, we can get $(o_n\ P\ o_k)$ by using a reasoning similar to the base case. Further, due to the induction hypothesis, for $\forall 1 \le x \le k+1$, if $(o_n\ P_I\ o_x)$ then $(o_n\ P\ o_1)$. □

From the above Lemma 1, we know that an operation which is invoked on an object is allowed to execute concurrently only when it is immediately preservable to all the operations which have executed but not committed yet on that object.

4.3 Example

It is assumed here that each operation is atomically executed and return values of operations are taken into account in determining conflicts; this consideration increases concurrency between transactions[9,10,13]. With Account object, we show how execution contexts affect the preservation table. Account object provides deposit(x1), withdraw(x2), and post(x3) operations, and each reverse-operation for each operation. Each reverse-operation for deposit(x1), withdraw(x2), and post(x3) involves decrementing the balance by x1, incrementing by x2, and decrementing the balance to be $balance \times 100/(100+x3)$ respectively. The preservation for Account object is shown in Table 1.

OP$_2$ requested / OP$_1$ executed	deposit(x1) /OK	withdraw(x2) /OK	withdraw(x2) /Insufficient	post(x3) /OK
deposit(x1') /OK	P	P_D $P_{(B)}$	P_U P_B	P_D
withdraw(x2') /OK	P	P_U P_B	P_D $P_{(B)}$	P_D
withdraw(x2') /Insufficient	P	P	P	P
post(x3')/OK	P_D	P_D	P_U, P_B	P

Table 1. Preservation for Account

In Table 1 , the entries denoted by P, P_D, P_U, P_B, and $P_{(B)}$ indicate, respectively, preservation in every models, only in the DU, UIP, biversion object model, and may be preservable or may not depending on the state of an object in the. biversion object model. The notation (OP2, OP1) used is meant that an operation OP2 is invoked when OP1 has executed not committed yet. Consider the pair (withdraw(x2)/OK, deposit(x1')/OK). In the UIP model, this pair is not preservable because the return value OK of withdraw(x2) could be different if previously executed deposit(x1')/OK operation aborted, whereas this pair is preservable in the DU model. In contrast, in the biversion object model, this pair is preservable if the return value of withdraw(x2) is OK from both the current and the committed state of Account object.

5 A Formal Comparison to Related Conflict Relations

In this section, we make a two-dimensional comparison between preservation and other conflict relations: one dimension is execution contexts, and the other is operations' specifications.

The reason for taking execution contexts as one dimension is that it is very difficult to compare the conflict relations each of which is defined in different execution context. For example, the two executions of the same operations, one generated in the DU model where FC is defined and the other in the UIP model where BC is defined, are different. That is, the case occurs that the execution recognized as serializable in the UIP model is not necessarily serializable in the DU model[16]. Thus, we take different execution contexts as one dimension for the formal comparison.

Operations' specifications indicate both the state of an object and the return values of operations after the execution of concurrent operations. From both the theoretical[3,13,14] and the practical[17] points of view, weak relations, such as recoverability and invalidation which consider only the return values of operations, achieve more concurrency than commutativity relations which consider both the state of an object and the return values of operations. Thus, we take operations' specifications as the other dimension for the formal comparison.

With these two dimensions(i.e., execution contexts and operations' specifications), we make a formal comparison between preservation and others, as is shown in Table 2. Here, it is assumed that the standard recovery systems, in which recovery is processed by simply performing the undo-operations for aborted operations, are employed.

Execution Contexts / Operation's Specification	UIP model	DU model	Biversion object model
state of an object & return values of operations	BC	FC	FBC
return values of operations	*Preservation*	*Preservation* Recoverability Invalidation	*Preservation*

Table 2. Formal comparison between preservation and other conflict relations

According to the formal comparison in Table 2, preservation is identical to recoverability and invalidation in the DU model. However, preservation can be used in every execution contexts, while still being weaker than commutativity. The seemingly overhead of preservation that the reverse-operations should be provided can be alleviated if the fact is taken into account that undo-operations equivalent to the reverse-operations are inherent in the UIP or biversion object model.

6 Preservation-based Concurrency Control

In this section, we discuss the issues related to a concurrency control based on preservation semantics. Each object scheduler O_k maintains it's dependency graph DG_k, which is composed of commit dependency graph OG_k and conflict relation graph CG_k. These graphs are defined as follows.

Definition 9

- $OG_k = (\mathrm{N}, \mathrm{M})$ is "cOmmit dependency Graph" at object k, where N is the set of nodes corresponding to active transactions that have begun execution but not committed, and M is a set of edges. An edge e belonging to M is a directed edge from T_j to T_i if T_i has executed o_i and T_j has executed o_j such that, $o_i <_{Ek} o_j$, and $\neg(o_j\ C\ o_i)$ but $(o_j\ P_I\ o_i)$.
- CG_k = (N, M) is "Conflict relation Graph" at object k, where N is the set of nodes corresponding to active transactions that have begun execution but not committed, and M is a set of edges. An edge e belonging to M is a directed edge from T_j to T_i if T_i has executed o_i and T_j has executed o_j such that, $o_i <_{Ek} o_j$, and $\neg(o_j\ P_I\ o_i)$.
- $DG_k (= OG_k \cup CG_k)$ is "Dependency Graph" at object k. □

The dependency DG_k is defined at object k. We must ensure, however, that there is no cycle in the dependency graph in the whole database system. To this end, we define the dependency graph in the database system in the following Definition 10.

Definition 10 $DG(= \cup DG_k)$ is "Dependency Graph in the database system". □

It is assumed that cycles formed in the DG can be handled using several known techniques of deadlock detection and resolution[11,12]. Each object scheduler O_k controls concurrent execution of operations using a concurrency control algorithm shown in Figure 1. A concurrency control proposed here is a 2 Phase Locking(2PL) scheme using typed locks, in which concurrency control operates at each invocation of an operation(i.e., a pessimistic concurrency scheme). If the invoked operation is immediately preservable to or commutes with operations belonging to all other active transactions, it can be executed by forming the commit dependency among trans actions which are in preservable but not commuting relation. Otherwise, it generates a conflict and the conflict relation is formed between transactions, and then the transaction is blocked until the conflict disappears; that is, until the termination(committed or aborted) of the blocking transaction. The data structures, procedures, and functions employed in a concurrency control algorithm are the following:

- Data Structures

  E_k : the execution log containing the set of operations executed at object k

  OG_k : the commit dependency graph at object k

  CG_k : the conflict relation graph at object k

  $DG(= \cup DG_k)$: the dependency graph in the database system.

- Procedures and Functions

  $o \Leftarrow E_k$: taking out from E_k one operation, o, one at a time.

  $(T_i \rightarrow T_j) \Rightarrow CG_k$: adding the edge from T_i to T_j in the CG_k.

  $(T_i \rightarrow T_j) \Rightarrow OG_k$: adding the edge from T_i to T_j in the OG_k.

  Cycle(DG) : returning TRUE if there exists any cycle in the DG, otherwise returning FALSE.

  Execute(o) : executing the operation o at object k.

  Abort(Ti) : aborting the active transaction T_i.

  Block(Ti) : making the active transaction T_i blocked until all the others which conflict with T_i commit or abort.

```
Input    $o_i$ : the invoked operation by an active $T_i$
Algorithm
  conflict := false;
  while ($E_k \neq \emptyset$) and $\neg$ conflict loop
    $o_j \Leftarrow E_k$;
    if $\neg(o_i P_I o_j)$ then
      conflict := true; $(T_i \rightarrow T_j) \Rightarrow CG_k$;
      if Cycle(DG) then Abort($T_i$); endif;
    else if $\neg(o_i C o_j)$ then
      $(T_i \rightarrow T_j) \Rightarrow OG_k$;
      if Cycle(DG) then Abort($T_i$); endif;
    endif
  endloop
  if conflict then Block($T_i$);
              else Execute($o_i$);
  endif
```

Figure 1. Preservation-based concurrency control algorithm

We now describe two properties which a presented concurrency control has in the following Definition 11 and Lemma 2.

Definition 11 An operation o_i invoked by transaction T_i is "safe" in a log E, if for all uncommitted operations $o_j <_E o_i$ $(j \neq i)$, $(o_i\ P\ o_j)$.

To ensure that the intended semantics of the operations are guaranteed in spite of transaction aborts, we shall require that all operations in a log be safe.

Lemma 2 A log E is free from cascading aborts if it contains only safe operations.

Proof : From Definition 11, safe operations in a log E are preservable, and from Definition 8, the state of database is preserved correct and return values of preservable operations are not affected even after one or more operations are aborted. Therefore, a log E is free from cascading aborts. □

Cascading aborts can be avoided by allowing only preservable operations to execute concurrently. A presented concurrency control algorithm produces a serializable log by scheduling operations such that cycles are not formed in the DG. We show this fact in the following Theorem 1.

Theorem 1 Cycles are not formed in the dependency graph in database system DG if a log E contains only safe operations.

Proof : From Definition 11, safe operations in a log E are preservable, and each object scheduler aborts the transaction that invoked an operation participating in forming a cycle in DG. Therefore, cycles are not formed in DG if a log E contains only safe operations. □

7 Recovery Issues

In this section, we discuss the recovery issues combined with the preservation-based concurrency control. In case of a transaction abort during normal functioning in the DU model, the recovery process consists of simply discarding all the operations of the aborted transaction from the corresponding intentions list. In contrast, in the UIP and biversion object models, the recovery process combined with preservation-based concurrency control consists of cancelling all of its update operations by executing, in reverse order, the corresponding reverse-operations. In this way, the effects of all the operations executed by other transactions after aborted one are not lost since these operations are preservable to the aborted one. This process is also applied to the case with a system failure when the *restart* procedure has to perform *undo* action, which is to suppress the effects of uncommitted transactions already incorporated to the database.

In addition to the standard recovery algorithms, we insist that the notion of preservation form the basis for formulating semantics-based recovery algorithms satisfying the requirements. Thus, several approaches to the semantics-based recovery algorithms can be constructed. We here propose one approach combined with the preservation-based concurrency control by enriching the semantics of reverse-operations in a way that meet the preservation requirements. The major motivation of extending recovery systems is to allow more concurrency in the UIP and biversion object models. We start by defining the enriched semantics of reverse-operations based on the resulting actions that will appear in the log as follows.

Definition 12 Consider an active update operation, o_i, and all the following active operations, $o_j \ldots o_k$, on some object x. The "enriched semantics" of $\overline{o_i}$ is $\overline{o_k} \ldots \overline{o_j}\,\overline{o_i} o_j \ldots o_k$ if there exists any active operation $o_{i'}$ such that $o_i <_{Ex} o_{i'}$ and $\neg(o_{i'}\ C\ o_i)$, otherwise it is $\overline{o_i}$. □

Note that the enriched semantics of one reverse-operation is comprised of undo and redo of operations *context-sensitively* in a way that meets the requirements of preservation. For example, in case of deposit(100)'s abort in the example in Section 4.1, the context-sensitive recovery with the enriched semantcs of reverse-operations involves the undo of post(5), followed by undo of deposit(100), and followed by redo of post(5). Furthermore, if we combine *before-state* and reverse-operations in recovery systems, the recovery systems with the enriched semantics of reverse-operations can be made efficiently. The value of an object x just before an operation o_i is executed is referred to as the before-state of x with respect to the operation o_i. The before-state of x may be umcommitted or committed value, whereas *before-image*[2, 10] of x is committed one. An operation restoring the before-state of the accessed object with respect to o_i is denoted by before-state(o_i). Note that the prefix comprised of a sequence of reverse-operations in Definition 12, $\overline{o_k} \ldots \overline{o_j}\,\overline{o_i}$, is exactly corresponding to the operation before-state(o_i). Thus, by restoring the before-state instead of executing the sequence of reverse-operations for all the following operations, the recovery systems can be made efficiently. For this end, we define the expanded reverse-operations which contain the enriched semantics as follows.

Definition 13 Consider one active update operation, o_i, and all the following active operations, $o_j \ldots o_k$, on some object x. An "expanded reverse-operation" for o_i, denoted by $Expanded(\overline{o_i})$, is defined such that $Expanded(\overline{o_i})$ is *before-state*$(o_i) o_j \ldots o_k$ if any active $o_{i'}$ such that $o_i <_{Ex} o_{i'}$ and $\neg(o_{i'}\ C\ o_i)$, otherwise it is $\overline{o_i}$. □

Each time an update operation, o_i, is executed on an object, the pair comprised of before-state(o_i) operation and $\overline{o_i}$ is recorded. Assuming these pairs, we can develop an alternative recovery system as follows:

When o_i aborts, the $Expanded(\overline{o_i})$ is executed.

That is, if there exists the following active operations, $o_j \ldots o_k$, which update the same object and any operation among them which does not commute with the aborted one, then *before-state*$(o_i) o_j \ldots o_k$ is executed. Otherwise, the reverse-operation for aborted one is executed. Note that, in case of the execution of *before-state*$(o_i) o_j \ldots o_k$, each before-state of $o_j \ldots o_k$ should be re-constructed semantically right. This kind of recovery systems proposed here is much more complex than existing standdard ones, but it is the cost we pay for allowing more transactions to execute concurrently in the UIP and biversion object models.

8 Conclusions

The semantics of abstract data types have been exploited in the scope of commutativity and weak relations. We pointed out, however, that most conflict relations are not general enough to be used independently of execution contexts and some conflict relations have no explicit considerations of the recovery perspective even though concurrency controls based on these conflict relations are necessarily supported by context-sensitive recovery. In this paper, we proposed preservation as a new conflict relation which explicitly includes the reverse-operations within the definition. Since the reverse-operations bridge the gap between concurrency control and recovery, preservation can be well fitted into every execution contexts.

Thus, one contribution of this paper is the provision of a single notion of conflict relation independent of execution contexts. An additional value of preservation lies in the fact that it forms a basis for formulating semantics-based recovery algorithms satisfying the requirements mentioned. Since the reverse-operations have a great impact on the conflicts in context of preservation, more elaborate recovery algorithms, such as context-sensitive ones, will have the potential for weaker conflicts. With respect to the performance issues, preservation-based concurrency control achieves more concurrency than commutativity from the theoretical point of view.

Acknowledgements

Special thanks go to the reviewers who have provided helpful comments on drafts of this paper.

References

[1] Weihl, W. "Commutativity-Based Concurrency Control for Abstract Data Types," *IEEE Trans. Comput.* Vol.37, No.12, pages 1488–1505, 1988.

[2] Berstein, P. A., Hadzilacos, V.,. and Goodman, N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, Mass. 1987.

[3] Badrinath, B., and Ramamritham, K. "Semantics-Based Concurrency Control : Beyond Commutativity," *ACM Trans. Database Syst.* Vol.17, No.1, pages 163–199, 1992.

[4] Agrawal, D., Abbadi, A., and Singh, A. "Consistency and Orderability : Semantics-Based Correctness Criteria for Databases," *ACM Trans. Database Syst.* Vol.18, No.3, pages 460–486, 1993.

[5] Garcia-Molina, H. "Using semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Trans. Database Syst.* Vol.8, No.2 , pages 186–213, 1983.

[6] Weihl. W. and Liskov. B. H. "Implementation of Resilient. Atomic Data Types," *ACM Trans. Program. Lang. Syst.* Vol.7, No.1, pages 244–269, 1985.

[7] Weihl, W. "Specification and Implementation of Atomic Data Types," Ph.D. Thesis MIT/LCS/TR-314, MIT, 545 Technology Square, Cambridge, Mass., 1984.

[8] Skarra, A. H., and Zdonik, S. B. Concurrency Control and objected-oriented databases. In Object-Oriented Concepts, Databases, and Applications. Won Kim and F. H. Lochovsky, pages 395–421, ACM Press, New York, 1989.

[9] Schwartz, P. M., and Spector, A. Z. "Synchronizing Shared Abstract Data Types," *ACM Trans. Comput. Syst.* Vol.2, No.3, pages 223–250, 1984.

[10] Berstein, A. J., and Lewis, P. M. Concurrency in Programming and Database Systems. Jones and Bartlett, Boston London. 1993.

[11] Mukesh Singhal. Deadlock Detection in Distributed Systems. In Distributed Computing Systems. Thomas L., Casavant, and Mukesh Singhal, pages 52–71, IEEE Computer Society Press, California, 1994.

[12] Knapp, E. "Deadlock Detection in Distributed Databases," *ACM Comput. Surv.* Vol.19, No.4, pages 303–328, 1987.

[13] Anastassopoulos, P., and Jean Dollimore. A Unified Approach to Distributed Concurrency Control. In Distributed Computing Systems. Thomas L., Casavant, and Mukesh Singhal, pages 545–571, IEEE Computer Society Press, California, 1994.

[14] Herlihy, M. P. "Apologizing versus asking permission : Optimistic concurrency control for abstract data types," *ACM Trans. Database Syst.* Vol.15, No.1, pages 96–124, 1990.

[15] Herlihy, M. P., and Weihl, W. "Hybrid concurrency control for abstract data types," In *Proceedings of the 7th ACM Symposium on Principles of Database Syst.* pages 201–210, 1988.

[16] Guerni, M., Ferri, J., and Pons, J. -F. "Concurrency and Recovery for Typed Objects using a New Commutativity Relation," In Deductive and Object-Oriented Databases. Lecture Notes in Computer Science 1013. Ling, T. W. et al., pages 411–428, 1995.

[17] Mak, H. K. and Wong, M. H. "An Experimental Study of Semantics-Based Concurrency Control Protocols," Technical Report CS-TR-95-08. The Chinese University of Hong Kong, Shatic, N.T., 1995.