Recovery for Transaction Failures in Object-Based Databases

Man Hon Wong

Department of Computer Science

The Chinese University of Hong Kong

Shatin, N.T., Hong Kong

Fax: (852)-603-5024

Email: mhwong@cs.cuhk.hk

Phone: (852)-609-8406

Abstract

In this paper, a set of recoverability theory is derived for an object-based database. Instead of considering serializability and recoverability as two orthogonal concepts, we simply keep serializability as the only correctness criterion and require serializability to be maintained even when failures of transactions may occur. Based on this fundamental notion of correctness, the definition of recoverability is derived. Fortunately, the recoverability theory derived in this way is a generalization of the traditional recoverability theory in the read/write model. In addition, we find that the set of strict histories depends on the strength of the inverse operations being used to cancel the effects of aborted operations. At one extreme, when the strongest inverse operations are used, the set of strict histories is the same as the set of avoid cascading aborts histories. At the other extreme, when the weakest inverse operations are used, the set of strict histories is the same as the set of rigorous histories.

1 Introduction

A user interacts with a database by performing a series of operations, referred to as a transaction. Concurrency control and recovery are two of the main components for transaction management. Concurrency control enforces serializability [4, 2] which requires the effects of a history to be equivalent to a serial history. Recovery mandates recoverability [8, 2] which ensures the correctness of a history even when failures of transactions may occur. Intuitively, a history is recoverable if whenever an uncommitted transaction aborts, its effects can be wiped out without subsequently aborting any committed transaction. Traditionally, a history is considered correct if it is both serializable and recoverable. The serializability and recoverability theory in the read/write model, where data objects can only be manipulated by read and write operations, has been well developed and applied in most of the commercial database systems.

Recently, databases are increasingly being used to model advance applications, and the traditional read/write model is found to be too restrictive for such applications. In particular, databases or the hot spots of databases are modeled by collections of abstract data objects and semantics of objects are used to improve concurrency. Various concurrency control protocols which adopt different serializability criteria have been proposed [10, 13, 14, 6, 9, 3, 15, 16]. However, relatively not much work has been done on the recovery issues in such models. Actually, serializability and recoverability are two closely related concepts which cannot be considered separately. There are two issues that deserve more investigation. Firstly, we have to study whether the traditional recovery theory is valid on the semantically rich object-based model. Consider the following history on counter objects X and Y which can be accessed by increment, decrement, read and write operations. Assume that the initial values of the counters are zero.

Transaction T_1:	increases counter X by 10
Transaction T_2:	writes 10 to counter X
Transaction T_3:	reads counter X and get 10
Transaction T_3:	reads counter Y and get 0
Transaction T_2:	writes 5 to counter Y
Transaction T_2:	commits
Transaction T_3:	commits

If semantics serializability [14] is adopted, the above history is serializable in the order of $T_1T_3T_2$, since the final states of the counter objects and the response to each operation is the same as that of the serial history executed in the order of $T_1T_3T_2$. The above execution is also recoverable, if the traditional recoverability in the read/write model is adopted. It is because the only read-from relation in the above history is that T_3 reads X from T_2, and T_3 commits after T_2 has committed. Although the above history is both "serializable" and "recoverable", it is not acceptable since the history is not correct if T_1 aborts. If the operation from T_1 is removed, the remaining history is not semantically serializable, since it is not "equivalent" to any serial history. In a serial history, T_3 would either see 0 in both counter X and Y or 10 in counter X and 5 in counter Y. Although the above example is ad hoc, at least it shows that some serializability criteria are not compatible with the traditional recoverability criterion.

Secondly, we have to study whether the traditional recovery method is an efficient method for object-based databases. Consider the following history on a counter object:

Transaction T_1:	increases the counter by 5
Transaction T_2:	decreases the counter by 2
Transaction T_2:	commits
Transaction T_1:	commits

Suppose conflict serializability with conflict relations derived from the complement of commutativity relations is adopted. The operation from T_1 does not conflict with the operation from T_2, because the increment operation commutes with the decrement operation (i.e. swapping the two operations does not affect the state of the object and the responses to the operations) Hence, they can be executed concurrently when concurrency control is the only concern. However, if the traditional recovery criterion in the read/write model is used, the decrement operation from transaction T_2 is considered a read and write operation, since the decrement operation is required to read the state written by the increment operation from transaction T_1 before it can determine and update the state of the counter object. Therefore, transaction T_2 reads from transaction T_1, and hence the above history is not recoverable under the traditional read/write model because T_2 has committed before T_1 commits. Recently, some researchers [11, 12] have proposed to use the semantics of data objects to relax the requirements for recoverable histories. In particular, inverse operations are used to wiped out the effects of aborted transactions. In the above example, the effects of the increment operation from transaction T_1 can be undone by executing a decrement operation (inverse of the increment operation) without affecting transaction T_2. Therefore, the above history could be considered recoverable, if a more advance recovery mechanism based on abstract data objects is used.

From the above examples, we notice that the traditional recovery mechanism may not be an efficient recovery mechanism for object-based databases. In some cases, it may even be incompatible with the serializability criterion adopted in the system and lead to unacceptable results. In this paper, we attempt to derive a recoverability criterion for an object-based database. Our approach is quite different from the previous approach in the way that we try to derive recoverability from the fundamental notion of recovery. We first adopt a correctness criterion in a failure free environment. Then recoverability is derived from the fundamental notion that requires the correctness criterion to be satisfied even when failures of transactions may occur. Fortunately, the recoverability derived in this way is a generalization of the traditional recoverability in the read/write model. Similarly, the notions of avoid cascading aborts histories, strict histories and rigorous histories are also defined for the object-based database model.

In the implementation, inverse operations are used to cancel the effects of aborted operations. We also point out in this paper that the set of histories accepted by these correctness criteria depend on the synchronization constraints imposed by the inverse operations. The inverse operation for an operation is in general not unique. Moreover, different inverse operations for an operation place different synchronization constraints on the execution of the operation. Consider the following execution on a read/write object [1]:

Transaction T_1: write 10 to the object
Transaction T_2: write 20 to the object

If the inverse operation of a write operation is to restore the before image [2] of the write operation, the above history is not acceptable. Consider the case that when T_1 aborts, it is obviously incorrect to restore the before image of the write operation of T_1 blindly. However, if a more sophisticated inverse operation is used, the above history is acceptable. A more sophisticated inverse operation for a write operation W can be implemented as follows: the log is scanned backward to find the most recent unaborted write operation. If the operation is executed after W, no action has to be done. Otherwise, restore the value written by the write operation. By using such an inverse operation for the write operation, the above history becomes acceptable. This example illustrates that the set of acceptable histories depends on the set of inverse operations being used. This issue has also been observed in [1]. In this paper, we will further investigate how the strength of inverse operations affects the set of accepted histories. We say that the strength of an inverse operation is strong if the synchronization constraints it places on the execution is weak. We find out that the set of strict histories depends on the strength of the inverse operations being used. At one extreme, when the strongest inverse operations are used, the set of strict histories is the same as the set of avoid cascading aborts histories. At the other extreme, when the weakest inverse operations are used, the set of strict histories is the same as the set of rigorous histories.

[1] Note that a read/write object is a special case of an abstract data object.

2 Preliminaries

We first describe the database model used in our system, and then present the notion of conflict serializability for object-based databases.

2.1 The Database Model

The database is a collection of *abstract data objects*, which can only be accessed through the operations defined by the specifications of the objects. For example, a counter can be defined as an abstract data object which can only be accessed by increment, decrement, read and write operations. Users can interact with the database by executing transactions [4, 5]. A transaction can access an object or modify the state of the object only by invoking operations defined for that object. The corresponding result of the invocation is returned to the transaction through a response message. For example the response to an increment operation must be "ok" and the response to a read operation is the value in the counter. Each transaction must wait for the response to its last invocation before invoking the next operation. A transaction either commits on all objects or aborts on all objects.

An invocation "*inv*" and the corresponding response "*res*" on an object X form an *operation*, written as $X : [inv, res]$. Unlike the traditional read/write model, the operation type of an operation in an object-based database depends on both the invocation and the response. Operations with the same invocation but different responses may have different conflict relations with other operations. In our model, *update-in-place* recovery strategy [8, 14] is used for the execution of operations; i.e., the effects of an operation is incorporated to an object immediately after its execution. An *operation sequence* is a series of operations. An operation sequence on an object is *valid*, if it satisfies the specification of the object. We assume that all the objects are *total objects*. That means there must be a response "res" for any valid invocation "inv" executed on a valid state of an object X, and the operation $X : [inv, res]$ will bring the object to another valid state. Hence, an operation sequence is valid if all the responses of the operations in the sequence are correct with respect to the specification. For example, if the initial state of a counter object X is 0, then $X : [increment(5), ok] \cdot X : [read, 5]$ is a valid operation sequence, and $X : [increment(5), ok] \cdot X : [read, 3]$ is an invalid operation sequence, where "$\cdot$" denotes concatenation of operation sequences. In addition, we assume that for any object X, the specification of the object satisfies the *prefix closure property*, i.e, any prefix of a valid operation sequence is valid.

Two operation sequences h_1 and h_2 are *equieffective* [9] if, for any operation sequence g, $h_1 \cdot g$ is valid if and only if $h_2 \cdot g$ is valid. That means whenever an operation sequence is valid on h_1, the same operation sequence (with the same responses) is valid on h_2, and vice versa. In other words, two operation sequences are equieffective if they lead the object to the states that are indistinguishable from the responses of further operations executed on the object. Since, the state of an object is uniquely determined by the sequence of operations that has been executed on the object, we can simply define the state of an object as an operation sequence, where the initial state is represented by the empty operation sequence. Of course, we can use a more efficient method for the implementation of an object's state. A *history*, which indicates the order of operations and events from different transactions, is a sequence which contains commit events, abort events and operations. The *operation sequence* for a history H, denoted $opseq(H)$, can be obtained from H by removing the abort events, the commit events and all the operations of aborted transactions. An operation sequence of h restricted on an object X, denoted $h|X$, is defined as a subsequence of operations executed on X in h.

2.2 Conflict Serializability

As mentioned in Section 1, the notion of recoverability is closely related to the correctness criterion for concurrency control. Therefore, in this section, the correctness criterion adopted in our system under a failure free environment is defined first. Then the notion of recoverability will be derived in the next section. We begin by reviewing the traditional definition of conflict serializability [4, 2]. Two operations *conflict* if they both operate on the same data object and at least one of them is a write operation. Two operation sequences are *conflict equivalent* if they contain the same set of operations and order the conflicting operations in the same way. For convenience, we define two histories to be conflict equivalent if their operation sequences are conflict equivalent. A history is *conflict serializable* if it is conflict equivalent to a serial history. The same essence for conflict equivalence and conflict serializability can be adopted in the object based model. However, the notion of conflict in the object-based model has to be redefined. Since, we don't want to restrict our model by adopting a particular conflict relation, we try to extract the essential property of a conflict relation and then develop the correctness criterion based on this property. Therefore, the recoverability theory developed in this paper is generalized enough for any system that adopts a conflict relation which satisfies the property.

Let R be a binary relation defined over the operations of an object X. Let h be a valid operation sequence on object X. The R-graph for the operation sequence h, denoted $R\text{-}graph(h)$, is defined as follows:

Definition 1 *R-graph(h) is a directed graph, where the nodes are the operations in h. There is an edge from operation p to operation q if and only if $(p, q) \in R$ and p appears before q in h.*

Intuitively, we want to define the notion of conflict so that if two operation sequences h_1 and h_2 are conflict equivalent, then we have the following property:

1. h_1 is valid if and only if h_2 is valid.

2. h_1 and h_2 are equieffective.

Suppose h and g are two operation sequences which contain the same set of operations. It can be observed that g orders related operations in the same way as h if and only if g can be obtained from the topological sort of R-*graph*(h). Therefore, we have the following requirement for conflict relations:

Definition 2 *A symmetric relation R defined over the operations of an object X is a conflict relation of X, if for any valid operation sequence h on X, all the operation sequences obtained from the topological sort of R-graph(h) are valid and equieffective.*

Note that from the definition, we require the conflict relation to be a symmetric relation, so that the conflict equivalent relation is also symmetric. This property is used in the proofs of the only-if parts of some of the theorems, such as Theorem 2. The implication is that if an asymmetric conflict relation is used, a weaker notion of recoverability may be derived. However, the details will not be discussed in this paper.

Given a history H, we define the conflict graph of H, denoted $CG(H)$, as the union of R_X-*graph*$(opseq(H)|X)$, for all objects X involved in H, where R_X is the conflict relation adopted for object X. In other words, the set of nodes in $CG(H)$ contains the unaborted operations in H. There is an edge from operation p to operation q, if p and q are executed on the same object, p conflicts with q and p appears before q in H. The serialization graph of a history H, denoted $SG(H)$, can be obtained from $CG(H)$ by merging all the operations which belong to the same transaction into a node and then removing all the loops with only one edge. Actually, the definition of serialization graph is the same as that of the read/write model. We claim without showing the proof that a history H is serializable if and only if $opseq(H)$ is valid and $CG(H)$ is acyclic.

Before concluding this section, we would like to give a conflict relation that satisfies the above property. In fact, any symmetric relation R defined over the operations of an object X is a conflict relation, if R contains the complement of the right backward commutativity relation [7, 14] of X. The right backward commutativity relation for an object is defined below.

Definition 3 *An operation p right commutes backward with an operation q, if for any state s, whenever $s \cdot p \cdot q$ is valid, $s \cdot q \cdot p$ is valid and is equieffective to $s \cdot p \cdot q$.*

For example, let p and q be two successful withdraw operations on a bank account object such that $p \cdot q$ is valid on state s. It can be verified that operation sequence $q \cdot p$ is also valid on state s and equivalent to $p \cdot q$, i.e., both operation sequences lead the object to the same state. Therefore two successful withdraw operations do not conflict with each other. For another example, let p be a deposit operation and q be a successful withdraw operation. The operation q does not right commute backward with p because the condition that $p \cdot q$ is valid on a state does not in general imply $q \cdot p$ is also valid on the state. Therefore the withdraw operation conflict with the deposit operation.

3 Recoverable Histories

In the previous section, the correctness criterion under a failure free environment - conflict serializability is introduced. In this section, recoverability, which enforces the correctness of a history even when failures of transactions may occur, is defined. Intuitively, a history is recoverable, if the history is still correct after all the events and the operations of uncommitted transactions are removed. Therefore, the fundamental definition of correct histories which are both serializable and recoverable can be defined as follows:

Definition 4 *A history is both serializable and recoverable (SR-RC), if the history is serializable and it is still serializable after the operations and the events of all the uncommitted transactions are removed.*

Let's illustrate the idea of Definition 4 with the following examples. Consider the sample history I in Figure 1 on a counter object X with initial value of 10. The history is actually a valid serial history. Therefore, it is serializable. The history is still serializable after the uncommitted transaction T_1 is removed, since the remaining history is still a valid serial history. Therefore, the history is SR-RC. Consider the sample history II in Figure 1 on counter X with initially value of 10. After the operations

Sample History I		Sample History II	
T_1:	X:[increase(5),ok]	T_1:	X:[increase(5),ok]
T_2:	X:[decrease(3),ok]	T_2:	X:[decrease(3),ok]
T_3:	X:[increase(2),ok]	T_3:	X:[read(),12]
T_2:	commit	T_3:	commit
T_3:	commit		

Figure 1: Sample Histories

and the events of the uncommitted transactions T_1 and T_2 are removed, the history becomes non-serializable, since the remaining history is not equivalent to any valid serial history. In a valid history, the response of the read operation should be 10. Therefore the history is not SR-RC.

Now, we want to derive a simple method to test for SR-RC histories. Consider a serializable history H. Let H' be a history obtained from H by removing the operations of uncommitted transactions. $CG(opseq(H'))$ can be obtained from $CG(opseq(H))$ by removing the nodes corresponding to the removed operations and all the edges associated to the nodes. Hence, $CG(H')$ is a subgraph of $CG(H)$. Recall that the serialization graph of a history can be obtained from the conflict graph of the history by merging all the operations which belong to the same transaction into a node, and then removing all the loops with a single edge. Therefore, $SG(H')$ is a subgraph of $SG(H)$. Since H is conflict serializable, $SG(H)$ is acyclic. Hence $SG(H')$ is also acyclic. However, the condition that $SG(H')$ is acyclic is insufficient to conclude that H' is conflict serializable, since $opseq(H')$ may not be valid. H' will be serializable, In short, we show that if H is serializable and the responses of the operations in H' remains valid, then H is SR-RC. if $opseq(H')$ is valid, i.e. the responses of the remaining operations remain unchanged.

The converse of the above property can also be shown easily. Let H be SR-RC and let H' be the history obtained from H by removing all the operations and the events of uncommitted transactions. By the definition of SR-RC , H' is conflict serializable. Hence, $opseq(H')$ is conflict equivalent to a valid operation sequence g of a serial history G. From the property of conflict equivalent, $opseq(H')$ is valid. Hence, the responses for all the operations in H' remain unchanged. Therefore, we have the following theorem.

Theorem 1 *An history is SR-RC if and only if*

1. *The history is serializable.*

2. *After removing all the operations of uncommitted transactions, the responses of the remaining operations remain unchanged.*

Since the first requirement in Theorem 1 is actually the requirement for serializability. Therefore, we define recoverability as the second requirement of Theorem 1. From the property of conflict graphs, we can further refine the definition of recoverable histories. We claim that when an operation p is removed from a serializable history H the responses of all the operations will remain unchanged if and only if all the operations which have a path from p in CG(H) do not depend on p in $opseq(H)$. A detailed proof is given in Lemma 1 of the appendix. The definition of dependency is defined as follows:

Definition 5 *An operation q does not depend on p in an operation sequence h, if removing p from any operation sequence which is conflict equivalent to h does not invalidate the response of q.*

Therefore, we can refine the theorem for recoverability as follows:

Theorem 2 *A history is recoverable if and only if for any operation p of an uncommitted transaction and for any operation q of a committed transaction which has a path from p in the conflict graph, q does not depend on p in the history.*

We can determine whether a history is recoverable efficiently if we can compute the dependency relation efficiently. For convenience, we may use a static dependency relation (based on operation types) defined in Definition 6. However, the static dependency relation is stronger than the dependency relation defined in Definition 5. Therefore, when the static dependency relation is used, the only-if part of theorem 2 may not hold.

Definition 6 *An operation q does not depend on operation p, if for all operation sequences h_1 and h_2, whenever $h_1 \cdot p \cdot h_2 \cdot q$ and $h_1 \cdot h_2$ are valid, then $h_1 \cdot h_2 \cdot q$ is valid.*

Now, let's illustrate the idea of Theorem 2 with the sample histories in Figure 1. The conflict graphs of the sample histories I and II are shown in Figure 2. In the conflict graph of the sample history I, no operation has a path from the increment operation of the uncommitted transaction T_1. Hence the sample history I is recoverable. Consider the conflict graph of the sample history II. The read operation of the committed transaction T_3 has paths to the increment and the decrement operations

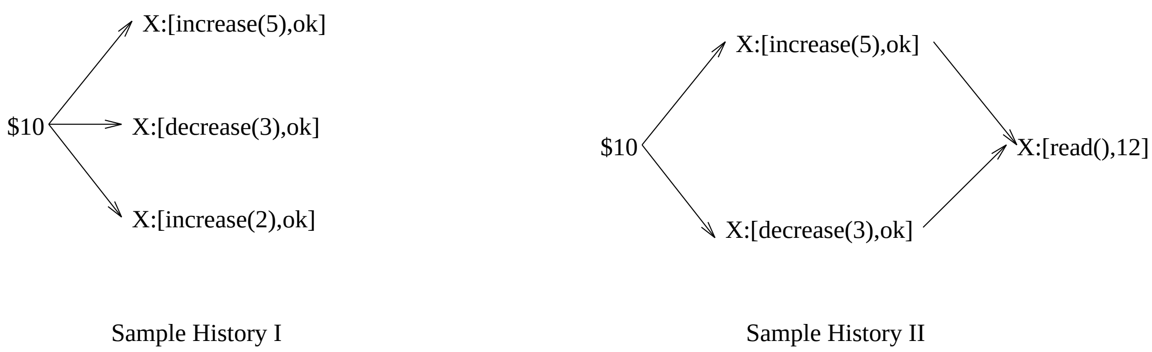

Figure 2: Conflict Graphs for the Sample Histories I and II

of the uncommitted transactions T_1 and T_2. Moreover, the read operation depends on the increment and the decrement operations. Thus, the sample history II is not recoverable.

Although recoverability is defined, the requirement of recoverability for a history can be interpreted in two ways. In the first way, the requirement in Theorem 2 is not required to be satisfied in every prefix of a history. Given a history, testing of recoverability can be done in the following way.

1. The history is scanned from the beginning.
2. The conflict graph is updated for each operation scanned.
3. Whenever, an abort event is scanned, the conflict graph will be examined to see whether there exist operations of committed transactions which have a path from the operations of the aborted transaction.
4. When the end of the history is reached, the conflict graph will also be examined to see whether there exist operations of committed transactions which have a path from the operations of the uncommitted transaction.

The history is considered non-recoverable, once the test in step 3 or 4 turns out to be true.

For example, consider the following history on a read/write object X:

T_1: X:[write(5),ok]
T_2: X:[read(),5]
T_2: commit
T_1: commit

The above history is considered recoverable testing procedure is used, since there is no abort event and no uncommitted transaction; therefore step 3 and 4 will never be executed. However, such an interpretation of recoverability is undesirable, since the requirement is not prefix-closed. Consider a prefix of the above history:

T_1: X:[write(5),ok]
T_2: X:[read(),5]
T_2: commit

The above history will become incorrect if T_1 aborts, since the response to the read operation may not valid. Therefore, the above non-prefix closed interpretation of recoverability is undesirable. In the other interpretation of recoverability, the requirement in Theorem 2 is required to be satisfied in every prefix of a history. This approach makes a pessimistic assumption that all the uncommitted transactions may abort. Practically, we test for recoverability in the following steps.

1. The history is scanned from the beginning.

2. The conflict graph is updated for each operation in the history.

3. Whenever a commit event is scanned, the conflict graph will be examined to see whether there exists an operation of the committed transaction which has a path from an operation of an uncommitted transaction.

The history is considered non-recoverable, once the test in step 3 turns out to be true.

Note that this prefix closed interpretation of recoverability is consistent with the traditional definition of recoverability in the read/write model. In the read/write model, a read operation conflicts and depends on the write operations from which it reads, and there is no other dependency in the read/write model. Therefore, from Theorem 2, a history in the read/write is recoverable whenever a transaction commits after the commitment of all the transactions from which it reads. Hence, the definition of recoverability derived for the object-based model is a generalization of that of the read/write model. In order to be consistent with the naming convention in the read/write model, we call the requirement for the second interpretation as recoverable (RC) and the requirement for the first interpretation as non-prefix closed recoverable (NPCRC). Although NPCRC is meaningless in the practical sense, we do mention it here for the sake of comparison with reducibility [12] in section 6. In the rest of the paper, SR-RC will refer to the class of histories which satisfy the SR-RC requirement in Definition 4 for every prefix of the histories. SR-NPCRC will refer to the non-prefix closed requirement of the SR-RC in Definition 4.

4 Avoid Cascading Aborts Histories

Enforcing recoverability guarantees that the effects of committed transactions are not affected by aborting any uncommitted transactions. However, it does not guarantee that uncommitted transactions are not affected. Hence, aborting a transaction may trigger further abortions of affected transactions, a phenomenon known as cascading abort [2]. In order to avoid cascading aborts, a stronger correctness criterion is required to enforce that a history is still correct after the operations and the events of any subset of uncommitted transactions are removed. Intuitively, the correctness criterion requires that removing the operations of any uncommitted transaction does not affect the operations of unaborted transactions. Hence we have the following definition for histories which are both serializable and avoid cascading aborts.

Definition 7 *A history is serializable and avoids cascading aborts (SR-ACA), if and only if the history is serializable and it remains serializable after the operations and the events of any subset of uncommitted transactions are removed.*

Note that the definition of ACA is similar to that of RC. RC requires that removing of aborted transactions does not affect committed transactions. While ACA requires that removing of aborted transactions does not affect unaborted transactions. By using the similar derivation for recoverable histories, we can obtain the following theorem for avoid cascading aborts histories.

Theorem 3 *A history is ACA if for any operation p of an uncommitted transaction and for any operation q of another transaction which has a path from p in the conflict graph, q does not depend on p in the history.*

We assume that the requirement of ACA is enforced for every prefix of a history. As mentioned above, in the read/write model, a read operation depends on the write operation from which it reads, and there is no other dependency in the read/write model. Hence, ACA is guaranteed, if a transaction may read only those values that are written by committed transactions. Therefore, this result is also a generalization of the ACA in the read/write model.

5 Strict Histories

As discussed in the last section, when RC or ACA is enforced together with serializability, it is guaranteed that the effects of a history is conflict equivalent to that of a serial history even when abortion of transactions may occur. However, we have not described any method to restore the state of an object after abortion of transactions. In this section, we introduce the idea of using inverse operations to undo the effects of aborted operations. As will become apparent later, the set of inverse operations being used the system is closely related to the set of strict history.

In the traditional read/write model, a history is strict if no operation is allowed to execute on an object X before all transactions that have previously written X are committed or aborted. The reason of enforcing strictness in the read/write model is that when strictness is enforced, the simple method of restoring the before image of a write operation can be used to undo the effects of the write operation [2]. The idea of strictness was extended for the object-based model in [11] in which a rather complicated state-based approach is used to define inverse operations and conflict relations. In this paper, a similar notion of strictness is also adopted.

Definition 8 *A history is strict if recovery for operations that abort in the history can be performed by simply executing their inverse operations.*

However, a much simpler definition of inverse operations is adopted in this paper. Inverse operations are defined formally as follows:

Definition 9 *An operation, p^{-1}, is an inverse operation of p on object X, if for all possible state s of X, whenever $s \cdot p$ is valid, then $s \cdot p \cdot p^{-1}$ is valid and equieffective to s.*

Readers can simply think that our system works in the following way. Whenever an operation p from T_p is executed, its inverse p^{-1} is also executed. p is executed with update-in-place strategy, therefore its effects is incorporated to the object immediately. But p^{-1} is executed after p with deferred update strategy. Therefore, the effect of p^{-1} is not incorporated to the object immediately. When T_p commits, p^{-1} will be aborted; otherwise when T_p aborts, the inverse operation will commit, hence the effects of p^{-1} will be incorporated to the object to cancel the effects of p. It has been shown in [14] that the complement of the right backward commutative relation is an optimal conflict relation when update-in-place recovery strategy is used, while the complement of the forward commutativity relation is an optimal conflict relation when deferred update recovery strategy [8, 14] is used. Therefore, we use right backward commutativity relation to derive the conflict relation between normal operations, but we use forward commutative relation to derive the "conflict relation" between an inverse operation and a normal operation. The definition of forward commutativity [14] is formally defined as follows:

Definition 10 *An operation p forward commutes with an operation q, if for any state s, whenever $s \cdot p$ and $s \cdot q$ are valid, then $s \cdot p \cdot q$ and $s \cdot q \cdot p$ are valid and equieffective to each other.*

The reason that we execute an inverse operation with deferred update strategy and require the inverse operation to forward commute with the active operations from other transactions is quite tricky and deserve more explanation. Readers may ask why we don't simply execute p^{-1} at the end of the history when p aborts and requires p^{-1} to right commute backward with other active operations. The reason is that from the definition of inverse operations (see Definition 9), we only require an inverse operation p^{-1} to be valid on the state after p is executed. The requirement that an inverse operation right commutes backward with all the other active operations cannot guarantee that the execution of the inverse operation p^{-1} is valid on the current state of the object when p aborts. For example, suppose operation p is valid on a state s, from the definition of inverse operations, we know that $s \cdot p \cdot p^{-1}$ is valid. However, if an operation q is executed after p, we cannot guarantee that $s \cdot p \cdot q \cdot p^{-1}$ is valid even when p^{-1} right commutes backward with q. However, if we enforce that the inverse operation p^{-1} forward commute with all the other active operations, p^{-1} must be valid on the current state of the object when p abort. For example, suppose operation p is valid on a state s, we know that $s \cdot p \cdot p^{-1}$ is valid. If an operation q is executed after p and we enforce that p^{-1} forward commute with q, it can be concluded that $s \cdot p \cdot q \cdot p^{-1}$ is valid and equivalent to $s \cdot p \cdot p^{-1} \cdot q$. A more detailed derivation for the theory of strict history is given below.

Consider an operation p in a history H. For brevity, assume that p is the only operation of the transaction which invokes p. It can be observed from the property of conflict graphs that any operation which has a path from p in $CG(H)$ must appear after p in any history which is conflict equivalent to H. All the other operations may appear before or after p in a history which is conflict equivalent to H. Let H' be a history which is conflict equivalent to H such that only those operations that have a path from p appear after p. Let $h_1 \cdot p \cdot h_2$ be $opseq(H')$. By definition of inverse operations, $h_1 \cdot p \cdot p^{-1}$ is valid and equieffective to h_1. It can be concluded that both $h_1 \cdot p \cdot p^{-1} \cdot h_2$ and $h_1 \cdot p \cdot h_2 \cdot p^{-1}$ are valid and equieffective to each other, if p^{-1} forward commutes with all the operations in h_2.

In other word, the effects of p in H' can be undone by executing p^{-1} at the end of H', if p^{-1} forward commutes with all the operations that appears after p in H'. Since, H and H' are conflict equivalent, the effects of p in H can also be be undone by p^{-1}, if all the operations which have a path from p in $CG(H)$ forward commute with p^{-1}. For the case that the transaction which invokes p has more than one operation, we can undo the effects of the operations by executing their inverse operations in the reverse order in which they appear in the transaction. Therefore, we have the following theorem for strictness.

Theorem 4 *A history H is strict, if for any operation p of an active transaction T_p in H, p^{-1} forward commutes with all the operations which do not belong to T_p and have a path from p in $CG(H)$.*

In the read/write model, a write operation conflicts with both read and write operations and its inverse operation (restoring before image) does not forward commute with both read and write operations. Therefore, from Theorem 4, a history is strict when no object may be read or overwritten until all transactions that previously wrote into it terminates. Therefore, the Theorem 4 is also consistent with the definition of strictness in the read/write model.

A protocol which accepts strict histories can be implemented in the following way. When an operation q is executed, if q conflicts directly or indirectly with an operation p of another active transaction T_p and p^{-1} does not forward commute with q, then the execution of q has to be delayed until T_p commits or aborts. When a transaction abort, the inverse operations will be executed in the reverse order of the normal operations of the aborted transaction.

It can be observed from Theorem 4 that the commutativity property of the inverse operations being used determines the set of strict histories. In the following sections, the two extreme cases of the strict histories are discussed.

5.1 The Strongest Inverse Operation: Universal Inverse Operation

A Log-based recovery action, known as the universal inverse operation, is proposed in this section to undo the effects of any operation with any semantics. We claim that the universal inverse operation provides the strongest commutativity relation (weakest conflict relation). Therefore, when the universal inverse operation is used, the set of strict histories is optimal.

The idea of the universal inverse operation is that when an operation is aborted, the initial state of the object is restored, and then all the unaborted operations on the object will be redone. Note that the universal inverse operation is independent of the semantics of the aborted operation. Therefore, it can be used to undo the effects of any operation with any semantics provided that the executions of the operations are deterministic. Of course such an implementation is not effective. But it is not difficult to come up with a more efficient implementation which involves redoing only those operations which execute after the aborted operation. However, we are not going to discuss the detailed implementation in this paper. Instead, we are interested in the commutativity property of the universal inverse operation.

Let u_p be the universal inverse operation which undoes the effects of operation p on object X. Assume that $h_1 \cdot p \cdot q$ is a valid operation sequence on object X. (Note that according to the definition of the universal inverse operations, $h_1 \cdot p \cdot u_p$ must be valid.) If u_p is executed at the end of the operation sequence, i.e. $h_1 \cdot p \cdot q \cdot u_p$, all the operations in h_1 and operation q are redone on the initial state of the object. Therefore, its effects is the same as $h_1 \cdot q$, if $h_1 \cdot q$ is valid. If the universal operation is executed right after p; i.e., $h_1 \cdot p \cdot u_p \cdot q$, then all the operations in h_1 is redone on the initial state first, and then operation q is executed. Therefore, its effects is also the same as $h_1 \cdot q$, if $h_1 \cdot q$ is valid. Hence, with the assumption that $h_1 \cdot p \cdot q$ and $h_1 \cdot p \cdot u_p$ are valid, $h_1 \cdot p \cdot q \cdot u_p$ and $h_1 \cdot p \cdot u_p \cdot q$ are valid and equieffective to each other, if removing of p does not affect the response of q. Therefore, u_p forward commutes with q, if removing of p does not affect the response of q. This property is formally stated below:

Theorem 5 *The universal inverse operation u_p which undoes the effects of an operation p in a history forward commutes with an operation q if q does not depend on p in the history.*

Therefore, when the universal inverse operation is used, a history H is strict, if for any operation p of transaction T_p, and for any operation q such that q has a path from p in $CG(H)$ and does not belong to T_p, then q does not depend on p. In other word, the set of strict histories is equal to the set of ACA histories when the universal inverse operation is used.

Moreover, there does not exist an inverse operation which has a stronger commutativity relation than that of the universal inverse operation. Suppose p^{-1} is an inverse operation of p which has a stronger forward commutativity relation than that of u_p; i.e. , there exists a valid operation sequence $h_1 \cdot p \cdot h_2$ such that p^{-1} forward commutes with all the operations in h_2 but some of the operations in h_2 depend on p. From the property of an inverse operation (see Definition 9), $h_1 \cdot p \cdot p^{-1}$ is valid and is equieffective to h_1. Since $h_1 \cdot p \cdot h_2$ is also valid, by the property of the forward commutativity relation, $h_1 \cdot p \cdot p^{-1} \cdot h_2$ is valid; i.e. $h_1 \cdot h_2$ is valid. It contradicts with the fact that some of the operations in h_2 depend on p. Hence the strength of the universal inverse operation is optimal. Therefore, if we can find an inverse operation which has the same commutative relation of the universe inverse operation, its strength is optimal.

Theorem 6 *An inverse operation p^{-1} of an operation p has the same strength as the universal inverse operation if for any normal operation q, p^{-1} does not commute with q only when q depends on p.*

For example, consider a counter object with increment, decrement and read operations. Decrement operations commute with all the operations except read operations, but read operations depend on decrement operations. Hence, a decrement operation has the same strength as the universal inverse operation for the increment operations, since it satisfies the condition in Theorem 6. However, if write operations are also possible operations on the counter object, then decrement operations do not have the same strength as the universal operation, since decrement operations do not forward commute with write operations and write operations do not depend on decrement operations.

When the inverse operations being used have the same strength of the universal inverse operation, a protocol which accepts RC histories can be implemented in the following way. When a transaction T requests commitment, all the operations of T will be examined. If there exists an operation q of T which conflicts directly or indirectly[2] with and depends on an operations p of another active transaction T_p, then the commitment of T has to be delayed until T_p has finished. When a transaction aborts, all the affected transactions will be identified recursively. A transaction is affected if one of its operations conflicts directly or indirectly with and depends on an operation of the aborted or affected transactions. All the operations of the affected transactions (include the original transaction which initiates the abortion) will be undone by executing their inverse operations in the reverse order in which they appear in the history.

5.2 The Weakest Inverse Operations

We say that the strength of an inverse operation is strongest if the commutativity relation with the normal operations is optimal. On the other hand, the strength of an inverse operation is weakest if the commutativity relation with the normal operations is minimal. Obviously, the commutativity relation is minimal when the inverse operation does not commute with any normal operation. We want to show that when the weakest inverse operations are used, the set of strict histories is equal to the set of rigorous histories (RG).

Definition 11 *A history is rigorous if all the operations from different active transactions do not conflict with each other.*

From Theorem 4, a history is strict, if for any operation p of active transaction T_p, p^{-1} commutes with all the operations which do not belong to T_p and have a path from p in the conflict graph of

[2] An operation a conflict indirectly with an operation b, if there exists an operation c such that a conflict directly with c and c conflict directly or indirectly with b.

the history. Therefore, if the inverse operations do not commute with any normal operation, the only possible scenario for a strict history is that for any normal operation p of active transaction T_p, there exists no operation which does not belong to T_p and has a path from p in the conflict graph. That means all the active operations do not conflict with each other if they belong to different transactions. Hence, by definition, the history is rigorous (RG).

If a history is rigorous, all the operations from different active transactions do not conflict with each other. Therefore, for any operation p of active transaction T_p, there does not exist an operation which does not belong to T_p and has a path from p in the conflict graph. From Theorem 4, the history is strict. In short, we show that when the weakest inverse operations are used, the set of strict histories is equal to the set of rigorous histories.

6 Comparison

In this section, we discuss the relationships among the newly introduced correctness criteria, the reducibility and the prefix-reducibility introduced in [12] and the traditional correctness criteria in the read/write model. In [12], a schedule (history) is expanded by adding undo actions (inverse operations) to the original schedule to cancel the effects of the uncommitted or aborted transactions. An expanded schedule is reducible (RED), if it can be transformed into a serial schedule by

1. Swapping adjacent operations which are commutative.
2. Combining and removing an operation with its inverse if they are adjacent with each other.
3. Removing any operation with null effect.

If a history is RED, obviously the history is still serializable after all the aborted operations are removed; i.e. SR-NPCRC. However, there are two reasons that SR-NPCRC histories may not be RED. First, the inverse operations used in [12] is much weaker than the universal inverse operation. For example, the inverse operation used in [12] for a write operation in the read/write object is to restore the before image of the write operation which is weaker than the universal inverse operation. Second, the way they expand a schedule impose some requirements on the commutativity relation between inverse operations.

For example, let p be a write operation executed by transaction T_1 on a counter object X and let q be an increase-read operation executed by transaction T_2 on the same counter object, where an increase-read operation increases the counter by one and then returns the value of the counter to the transaction. Since p conflicts (does not commute) with q and q depends on p. The abortion of T_1 will trigger the abortion of T_2, as shown in the following history:

T_1: X:[write(2),ok] (p)
T_2: X:[increase-read(),ok] (q)
T_1: abort
T_2: abort

In [12], the order of the inverse operations added to the original history follows the order of the abort events. Therefore, the history is expanded as follows:

T_1: X:[write(2),ok] (p)
T_2: X:[increase-read(),ok] (q)
T_1: p^{-1}
T_1: commit
T_2: q^{-1}
T_2: commit

Since, q depends on p, p^{-1} must not commute with q. Hence, the above history is RED only if p^{-1} commutes with q^{-1}. For example, if p^{-1} is a write operation which restores the before image of the write operation of T_1 and q^{-1} is a decrement operation, the above history is not RED. However, in our definition of SR-NPCRC, the commutativity requirement between inverse operations is not involved. In our implementation, when T_1 aborts, all the affected operation will be identified by using the conflict graph and the dependency relation. In the above history q of T_2 is affected and is required to be aborted. Then the inverse operations will be executed in the reverse order of their corresponding operations in the history; i.e., q^{-1} before p^{-1}. In short, RED $\subset$ SR-NPCRC.

Prefix-reducible (PRED) [12] enforces RED for every prefix in a history,. Similarly, SR-RC enforces SR-NPCRC for every prefix in a history. Since, RED $\subset$ SR-NPCRC because of the above two reasons, it is not difficult to show that PRED $\subset$ SR-RC. As pointed out in Section 3, the SR-RC defined in this paper is consistent with the conventional definition in the read/write model. It is claimed in [12] that RED is incomparable to SR-RC. The reason of existing histories which are RED but not SR-RC is obvious, since RED is a non-prefix closed property, while SR-RC is a prefix closed property. The reason of existing histories which are SR-RC but not RED is because the power of the inverse operations used in [12] is not optimal. The following is a similar example given in [12] to show that there exists a history in SR-RC but it is not RED:

T_1: X:[write(5),ok]

T_2: X:[write(6),ok]

T_1: abort

T_2: abort

The above history is SR-RC but not RED because the inverse of $X : [write(5), ok]$ does not commute with $X : [write(6), ok]$. Note that the above history is also ACA (SR-ACA),since the write operation of T_2 does not depend on the write operation of T_1. Therefore, there exists a history which is ACA (SR-ACA) but not RED.

Consider the following history:

T_1: X:[write(5),ok]

T_2: X:[read(),5]

The above history is PRED (and hence, RED), because for any prefix of the above history, the expanded schedule can be reduced to a null schedule. However, the history is not ACA (SR-ACA), since the read operation of T_2 depends on the write operation of T_1. Hence, the abortion of T_1 will trigger the abortion of T_2. From these examples, we can conclude that ACA (SR-ACA) is incomparable to PRED and RED.

From the definitions, it is easy to show that ACA is a strict subset of RC and RG is a strict subset of ACA. Therefore, $RG \subset ACA \subset RC$. In section 5, we show an interesting result that the set of strict histories depends on the set of inverse operations being used. At one extreme, when the universal inverse operation is used, ST is the same as ACA. At the other extreme, when the inverse operations do not commute with any normal operation, ST is the same as RG. In short, we summarize the relationships of various classes of histories in the Figure 3, where the boundary of SR-ST is somewhere in the shaded area.

7 Conclusion

The notions of recoverable histories, avoid cascading aborts histories, strict histories, rigorous histories are derived from their fundamental meanings on the semantically rich object-based model. When only read/write objects are used, all the correctness criteria proposed in this paper will reduce to

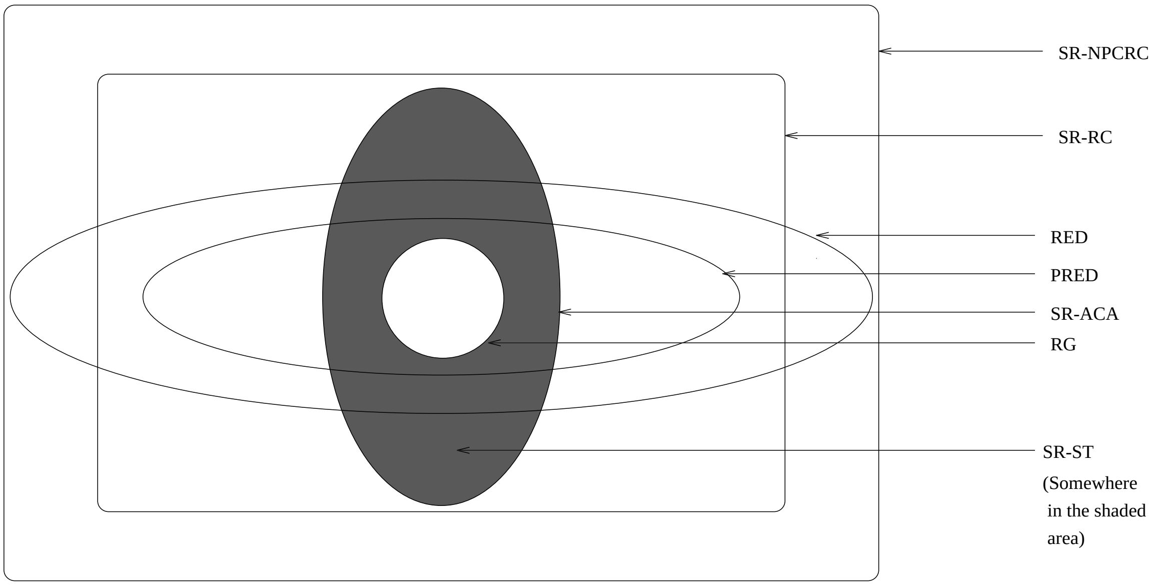

Figure 3: Hierarchy of Various Classes of Histories

their counterparts in the traditional read/write model. Therefore, the theory developed in the object-based model is a generalization of those in the read/write model. Practical protocols using inverse operations have been proposed to accept different classes of histories. And the required strength of the inverse operations for different classes of histories are discussed. In this paper, we define the strength of inverse operations based on their commutativity property with the normal operations. We show that the strength of an inverse operation is optimal when its commutativity relation with the normal operations is the same as the universal inverse operation. It is interesting to point out that the set of strict histories depends on the strength of the inverse operations being used. At one extreme, when the strongest inverse operations are used, the set of strict histories is the same as the set of avoid cascading aborts histories. At the other extreme, when the weakest inverse operations are used, the set of strict histories is the same as the set of rigorous histories.

The proposed correctness criteria are also compared with the reducibility and prefix-reducibility proposed in [12]. It has been shown that our non-prefix closed notion of recoverability with serializability is less restricted than reducibility (non-prefix closed) and our prefix closed notion of recoverability with serializability is also less restricted than prefix-reducibility (prefix closed). Moreover, practical methods have been proposed in this paper to test for both non-prefix closed and prefix closed recoverable histories.

Acknowledgements

The author would like to thank Prof. Divyakant Agrawal for his comments on the early draft of this paper.

References

[1] G. Alonso, D. Agrawal, and A. El Abbadi. Reducing Recovery Constraints on Locking based Protocols. In *Proceedings of the Thirteenth ACM Symposium on Principles of Database Systems*, pages 129–138, May 1994.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, Massachusetts, 1987.

[3] P. K. Chrysanthis, S. Raghuram, and K. Ramamritham. Extracting Concurrency from Objects: A Methodology. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 108–117, May 1991.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[5] J. N. Gray. Notes on database systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[6] T. Hadzilacos and V Hadzilacos. Transaction Synchronisation in Object Bases. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 193–200, 1988.

[7] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2–24, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[8] T. Härder and A. Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4), December 1983.

[9] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[10] H. F. Korth. Locking Primitives in a Database System. *Journal of the ACM*, 30(1):55–79, January 1983.

[11] R. Rastogi, H.F.Korth, and A. Silberschatz. Strict Histories in Object-Based Database Systems. In *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems*, pages 288–299, May 1993.

[12] H. Schek, G. Weikum, and H. Ye. Towards a Unified Theory of Concurrency Control and Recovery. In *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems*, pages 300–311, May 1993.

[13] P. M. Schwarz and A. Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[14] W. E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–283, April 1989.

[15] M. H. Wong and D. Agrawal. Context-Specific Synchronization for Atomic Data Types. In *Proceedings of the International Conference on Database Theory*, pages 201–215, October 1992.

[16] M. H. Wong and D. Agrawal. Context-based Synchronization: An Approach Beyond Semantics for Concurrency Control. In *Proceedings of the ACM SIGMOD-SIGACT Symposium on Principles of Databases*, pages 276–287, June 1993.

A Appendix

Lemma 1 *When an operation p is removed from a serializable history H, the responses of all the operations will remain unchanged if and only if all the operations which have a path from p in $CG(H)$ do not depend on p in $opseq(H)$.*

Proof: The proof for the only if part is trivial. Let G be the history obtained from H by removing operation p. If the responses of all operations in H remain unchanged after removing p, the responses of all the operations in G remain valid. From the property of conflict equivalent, the responses of the operations in any history which is conflict equivalent to G is valid. That means removing p in any history that is conflict equivalent to H does not change the responses of the remained operations . Hence, all operations which have a path from p do not depend on p in $opseq(H)$.

The proof of the if part is given as follows. Let H' be an operation sequence which is conflict equivalent to H such that all the operations appeared after p must have a path from p in $CG(H)$. Let G and G' be the histories obtained from H and H' by removing p, respectively. Since, all the operations which have a path from p in $CG(H)$ do not depend on p, all the responses of operations in H' remain unchanged after p is removed. Hence, $opseq(G')$ is valid. Obviously, G and G' are conflict equivalent. Therefore, $opseq(G)$ is valid. Thus, all the responses of operations in H remain unchanged after p is removed.