Iranian Journal of Science & Technology, Transaction B, Engineering, Vol. 30, No. B2
Printed in The Islamic Republic of Iran, 2006
© Shiraz University

RECONCILING INSTEAD OF ABORTING UNSUCCESSFUL TRANSACTIONS IN MULTIVERSION OBJECTBASES[*]

A. R. HADAEGH[**] AND K. BARKER

Dept. of Computer Science, University of Calgary, California State University, San Marcos,
Email: ahadaegh@csusm.edu

Abstract– This paper describes an architectural model to facilitate multiversion objects that are explicitly designed to enhance concurrency. The reader should be aware that version management has been used in the object literature in several ways, most commonly dealing with design issues. Our goal here is related to concurrency control and reliability, so care must be taken to ensure the reader is not misled by this overloading of terminology found in the literature. Within the context of concurrency the key aspects addressed by this paper are: 1) An architectural model is developed to support multiversioning that provides the well-known ACID transaction properties; 2) An optimistic concurrency control algorithm that functions on this architecture is described and demonstrated to be correct with respect to a correctness criterion; 3) The algorithm is enhanced to examine the history of past versions with the goal of inserting a committing transaction at a time earlier in the sequence when it would have been valid if other, later transactions had not been completed before this one attempted to commit; and 4) Based on static analysis information, other algorithms are developed to optimize the compiler in order to generate reconciliation procedures automatically from the initial transaction specification.

Keywords– object-oriented, optimistic concurrency control, serializability, nested transactions, reconciliation

1. INTRODUCTION

Traditional multiversion database environments have used data versioning for historical purposes as well as issues related to transaction management. Data versioning reduces the overhead involved in recovery, and impacts concurrency, especially in an environment where contention between read-only and update queries is problematic. This paper presents a model of versioned objects for an objectbase environment where the objectbase consists of a set of objects which contain structure and behavior. An object's structure is the set of attributes encapsulated by it. An object's behavior is defined by procedures called methods. A method's operations can read or write an attribute or invoke another method, possibly on another object.

Multiple users may access the objectbase at the same time and their access must be controlled to avoid concurrency anomalies such as lost updates and inconsistent reads. Transactions are used to facilitate this control. Traditionally transactions are defined as a sequence of read and write operations on passive data. In an object-oriented system a transaction consists of a sequence of method invocations which perform operations on object attributes on the transaction's behalf.

Concurrent execution of a set of transactions must be controlled so that the final result of the execution is equivalent to the result of some serial execution of the transactions (i.e. "serializable" [1], [2]). An objectbase system is provided with a scheduler that orders the operations of the concurrent

[*]Received by the editors March 2, 2005; final revised form October 8, 2005.
[**]Corresponding author

transactions based on a correctness criterion. Correctness criteria are enforced by concurrency control algorithms that ensure the serializability of concurrently executing transactions. Concurrency control algorithms are divided into two broad categories: pessimistic and optimistic. Pessimistic protocols block the transactions by deferring the execution of some conflicting operations. Optimistic algorithms do not block the transactions, but validate their correctness at commit time. Focusing on the centralized objectbase environment (not Distributed System [3]), this paper introduces an object versioning technique and develops a related optimistic concurrency control algorithm. Our model introduces two types of concurrency control: inter-UT and intra-UT concurrency. Inter-UT concurrency refers to the concurrent execution of multiple user transactions. Intra-UT concurrency refers to concurrent execution of multiple subtransactions originating from the same user transaction.

One problem with optimistic concurrency algorithms [4-6] is that transactions are aborted and possibly restarted if they cannot be serialized based on the defined correctness criterion. The goal of this research is to reconcile unsuccessful transactions instead of aborting them. Two methods of reconciliation have been addressed in this paper: simple reconciliation and complex reconciliation. Simple reconciliation involves changing the commit order of some of the transactions without making the state of the objectbase inconsistent. Complex reconciliation involves partial re-execution if the data it used was incorrect.
The paper begins by describing related work on multiversion concurrency control and transaction models for objects in Section 2. Section 3 describes our model and defines its key concepts. Architecture for our model and details of its components is presented in Section 4. Section 5 discusses reconciliation and explains the conditions under which transactions can be reconciled. The required static analysis information which demonstrates the steps required to generate reconciliation procedures at compile time is illustrated in this section. Finally, Section 6 makes some concluding remarks and suggests directions for future work.

2. RELATED WORKS

Using multiple versions of data items for transaction synchronization was first proposed by Reed [7]. Multiversioning allows for enhanced concurrency, simplifies recoverability, and supports temporal data management. This section reviews relevant multiversion and objectbase concurrency control literature. Nakajima [4] presents an optimistic multiversion concurrency control mechanism. Multiversioning techniques are applied to the concepts of backward and forward commutativity introduced by Weihl [8]. According to Weihl, two operations executing on an object commute if they can be scheduled in any order without affecting the result of computation. Nakajima argues that forward commutativity requires the latest committed versions of the objects to determine a conflict relation, while backward commutativity requires the current states of the objects. Using multiversioning, forward and backward commutativity relations are combined into a new relation called the *general commutativity relation*. A general commutativity relation exists between two operations if they either backward commute or forward commute.

In Nakajima's model, each object consists of a collection of versions. The versions are classified into two groups: the committed and the uncommitted versions. The most recent committed version of an object o^f is called the last committed version of o^f (denoted $LCV(o^f)$), and the most recent uncommitted version of o^f is called the current version of o^f (denoted $CV(o^f)$). When transaction T_i invokes method M^f_k on object o^f, a new uncommitted version of o^f (denoted $NV(o^f)$) is created for T_i. If the return result from $NV(o^f)$ backward commutes with $CV(o^f)$ or forward commutes with $LCV(o^f)$, $NV(o^f)$ becomes the new current version of o^f and replaces the old current version. Otherwise, $NV(o^f)$ is discarded and T_i invokes method M^f_k again.

Graham and Barker [5] proposed another optimistic concurrency control scheme for objectbase systems. In this algorithm each transaction T_i obtains copies of the objects it requires and is executed independently of other transactions. Thus, transactions do not interact with each other until commit time. At commit time, transactions that have read stale data must be reconciled before they commit.

Two types of reconciliation are introduced: *simple reconciliation* and *complex reconciliation*. Simple reconciliation merges the result of the execution of two versions, o^{f1} and o^{f2} of object o^f accessed by two transactions T_1 and T_2, respectively and provides a serialization order between T_1 and T_2. If versions o^{f1} and o^{f2} cannot be directly merged (due to the use of stale data), complex reconciliation is attempted. Complex reconciliation of two transactions, T_1 and T_2 may require the less costly transaction be re-executed (partially or entirely) against the state created by another transaction. The cost of the re-execution of a transaction is estimated by static compile time analysis [9]. Graham and Barker's model is limited to flat transactions and therefore does not consider the advanced transaction models. Further, their work does not explain how reconciliation procedures are generated at compile time.

3. THE COMPUTATIONAL MODEL

The objectbase system introduced in this paper consists of a set of objects. Each object has a unique identifier with a set of attributes and methods. Attributes form the structure of the object, and methods modify the state of the attributes. An object with identifier *f* is denoted o^f. Objects are versionable in that several versions can be derived from a given object. The versions of the objects are either committed or active. An active version *i* of an object o^f (denoted v^{fi}) begins as a copy of a persistent object and is manipulated independently of all other versions. Methods of object o^f are denoted as $m^1{}_f$, $m^2{}_f$, $m^3{}_f$, …, etc.

Transactions are submitted to the objectbase and are executed concurrently. We have employed the nest transaction model introduced by Moss [10] and have divided the transactions into two categories: User Transactions and Version Transactions. Each User Transaction contains a set of method invocations of objects. Each method associated with a user transaction is converted to a Version Transaction that is executed on a version of an object. For example, if User Transaction *i* (UT_i), consists of three methods $m^a{}_1$, $m^b{}_2$, and $m^c{}_3$, the version transactions generated from these three methods are denoted as $VT^a{}_{i1}$, $VT^b{}_{i2}$, and $VT^c{}_{i3}$ respectively. In general, each version transaction $VT^f{}_{ik}$ corresponds to the k^th^ version transaction of UT_i that should be executed against the version of object o^f called v^{fi}.

Unlike user transactions, version transactions manipulate local attributes using read and write operations, execute pre-commit operations on the object versions, and may also invoke other methods. Version Transactions that are generated from the methods of the same object are all executed against one object version. However version transactions that are generated from methods of different objects execute on separate versions.

Concurrent execution of a set of truncations forms a schedule. In order to ensure that the objectbase always moves from one consistent state to another, the schedule must be serializable. This means, the end result of the concurrent execution of a set of transactions must be equivalent to a serial execution of those transactions. A serializable schedule is correct if it is done based on some correctness criteria. The most common correctness criterion is Conflict-serializability [1]. We have introduced a new correctness criterion called *value-serializability* that relaxes the limitation of conflict-serializability. For a complete definition of user transaction, version transaction, user and version transaction schedules and all related issues to value-serializability the reader can refer to [6].

4. THE ARCHITECTURE

A versioned object store is comprised of two portions: a non-persistent *unstable* working store, and a persistent *objectbase*. An unstable store contains active versions. An active version is derived from a committed version, undergoes some changes, and may eventually become a new committed version. Committed versions are maintained in the objectbase. If the number of committed versions exceeds a system defined limit, some are archived. An object, with its committed versions, constructs an *object family*. To distinguish different version types of o^f, we denote v^{fi} and o^{fi} to represent an active version, and a committed version respectively.

Creating an active version, v^{fi}, from a committed version of the object family *f*, requires copying a committed version and giving it a unique version identifier *i*. Promoting an active version v^{fi} requires recording v^{fi} as o^{fi} in the objectbase.

The architectural model: Three major components form the basis of our architecture: the *Transaction Processor*, the *Version Processor*, and the *Validation Processor* (Fig. 1). The Transaction Processor accepts user transactions and returns results to the user. It processes transactions for syntactic correctness and performs coordination functions for inter-object method executions by converting the method invocations to version transactions and scheduling version transactions for user transactions. The version transactions are scheduled (using the *depends* function) so that version transactions of a single user transaction invoked on the same active version are ordered before they are processed. Version transactions of multiple user transactions are executed concurrently.

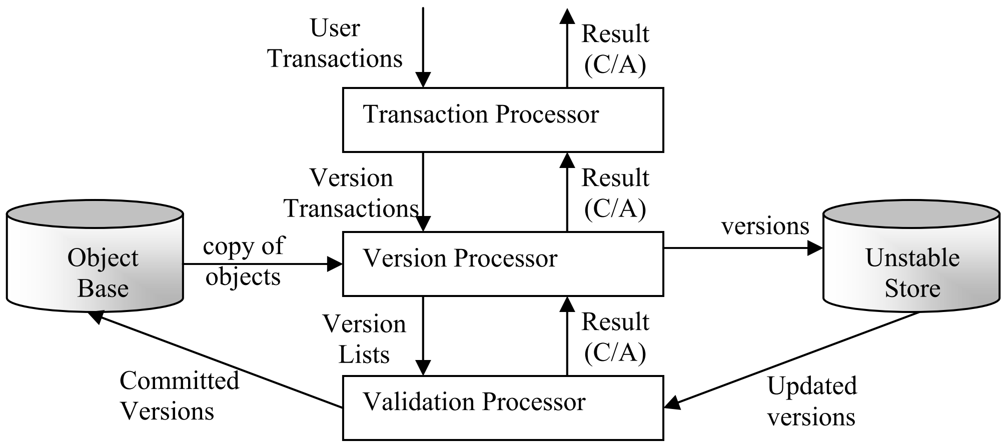


Fig. 1. The main components of the architecture

The Version Processor receives the scheduled version transactions from the Transaction Processor and creates new active versions of the objects required by the version transactions by copying from the committed versions of the objects from the objectbase. The active versions associated with the version transactions of a given user transaction (UT_i) are logically grouped into a *version list*}(*VRLST(UT*$_i$*))* after their completion and are submitted to the Validation Processor.

The Validation Processor examines the version lists and decides whether to abort or commit the user transactions. It has two components: the Decision Manager and the Commit Manager (Fig. 2).

The Decision Manager compares each updated active version (v^{fi}), referred by its object family *id* in the version list with the most recent committed versions of object family *f* (o^{fi}'s) in the objectbase. The purpose of the comparison is to determine if updated active versions would create inconsistency in the objectbase. An updated version v^{fi} is consistent with the state of other committed versions of o^f if the attributes accessed in v^{fi} have not been accessed since v^{fi} was created. If this is the case for every active version v^{fi} of UT_i, $VRLST(UT_i)$ is sent to the Commit Manager, the Commit Manager promotes the updated versions to committed versions and records them beside other committed versions in the objectbase.

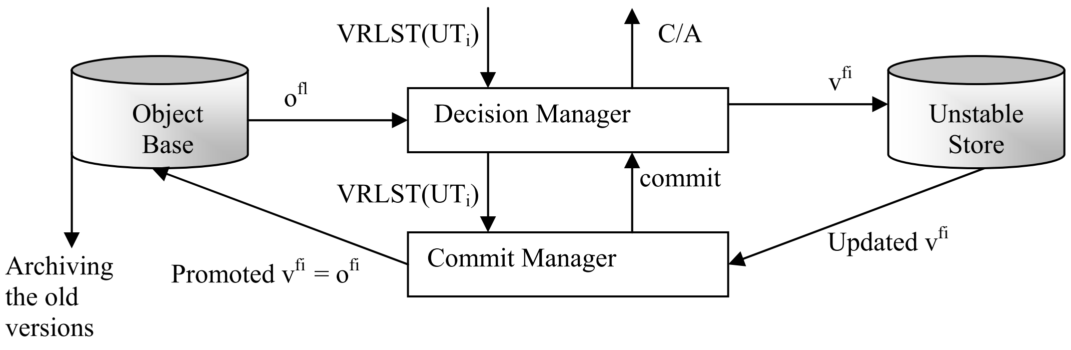


Fig. 2. The validation processor

If the states of some active versions of UT_i are not consistent with their corresponding committed versions in the objectbase, UT_i has read stale data and should be aborted. In complex systems where transactions are usually long and the overhead involved with the execution of each transaction is not trivial, aborting and restarting an unsuccessful transaction UT_i may significantly degrade the system performance. Our approach modifies the Decision Manager to reconcile UT_i before it decides to abort it. The Decision Manager checks if it is possible to change the commit order of UT_i and some recently committed user transactions. This is called *simple reconciliation*. Changing the commit order of the user transactions is possible if both intra-object and inter-object serializability are ensured and the objectbase consistency is still maintained after simple reconciliation is complete. If simple reconciliation is not possible, *complex reconciliation* is performed. In complex reconciliation of UT_i, the stale data are reread from the objectbase and some operations of UT_i, only the ones related to these stale data, are re-executed against the updated active versions of UT_i. When the results made from all the updated versions of UT_i become consistent with the current state of their corresponding objects in the objectbase, $VRLST(UT_i)$ is passed to the Commit Manager. If both simple reconciliation and complex reconciliation fail, UT_i is aborted and its active versions are disposed.

5. RECONCILIATION

This section details algorithms related to simple reconciliation and complex reconciliation.

a) Simple reconciliation

Figure 3 shows an example of the case when the execution of a user transaction is not successful and reconciliation is required.

Figure 3a shows the original state of the object family *f*. UT_1 and UT_4 start first and each receives a copy of o^{f0} denoted by v^{f1} and v^{f4}, respectively. UT_1 executes m^f_1 against v^{f1} and UT_4 executes m^f_4 against v^{f4}. UT_1 commits and its associated committed version o^{f1} is recorded in the objectbase (Fig. 3b). Then UT_2 starts, obtains a copy of o^{f1} (v^{f2}), executes m^f_2 against v^{f2}, and commits. v^{f2} is promoted to o^{f2} and is recorded in the objectbase (Fig. 3c). Next UT_3 starts, obtains a copy of o^{f2} (v^{f3}), executes m^f_3 against v^{f3} and commits. v^{f3} is promoted to o^{f3} and is recorded in the objectbase (Fig. 3D). Now UT_4 terminates; however, v^{f4} can not be committed as the last committed version in the chain because the value of *c* read in v^{f4} is stale. Although v^{f4} cannot be placed at the top of the version chain in the object family *f*, it might be possible to insert it in the lower levels without making the state of other committed versions inconsistent. Simple reconciliation is the process that checks and inserts v^{f4} in an appropriate position in the version-chain. This is done by the Validation Processor.

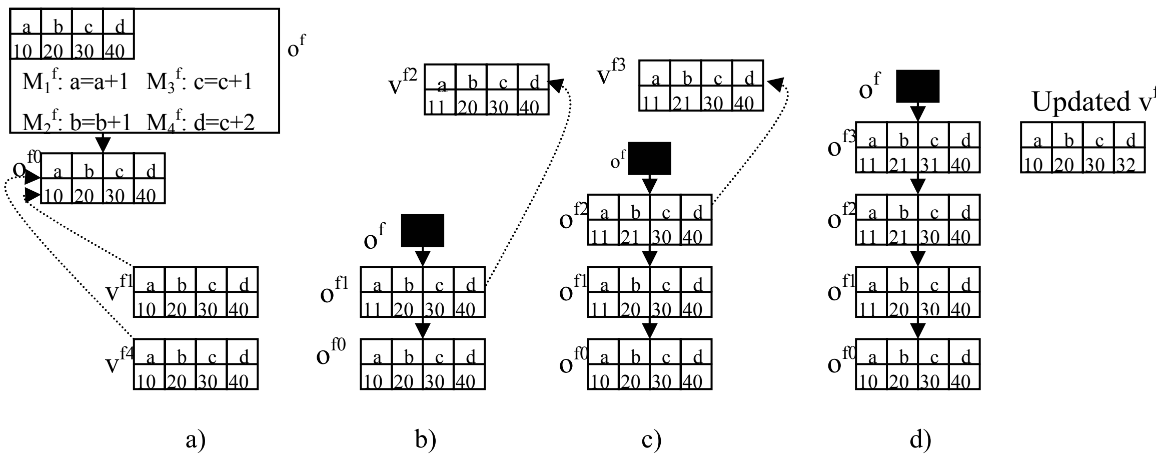


Fig. 3. An example when simple reconciliation is required

b) Decision manager

If it is found that a user transaction UT_i cannot be committed normally, the Decision Manager processes $VRLST(UT_i)$ and finds if simple reconciliation of UT_i is possible. Simple reconciliation is performed if two conditions are satisfied. First, promoting each active version v^{fi} to o^{fi} and locating o^{fi} in the object family, *f* must ensure intra-object serializability at the object family *f*. Second, the entire transaction system must be inter-object serializable.

Intra-object serializability-Consider Fig. 4. Suppose active version v^{fi} originates from o^{fj} when UT_i references object family *f*. The following data structures are required:

[BeforeImage(v)]: is a snapshot of a version *v* before *v* is modified. This snapshot is required to validate *v* at commit time.

[readset(v)]: is a set of data items read in version *v* by some transactions.

[writeset(v)]: is a set of data items written in version *v* by some transactions.

[ValidPos[]]:is a one dimensional array of integers. The index of each element corresponds to an active version v^{fi}. *ValidPos[*v^{fi}*]* refers to the position in the version-chain of object family *f* where v^{fi} can be inserted.

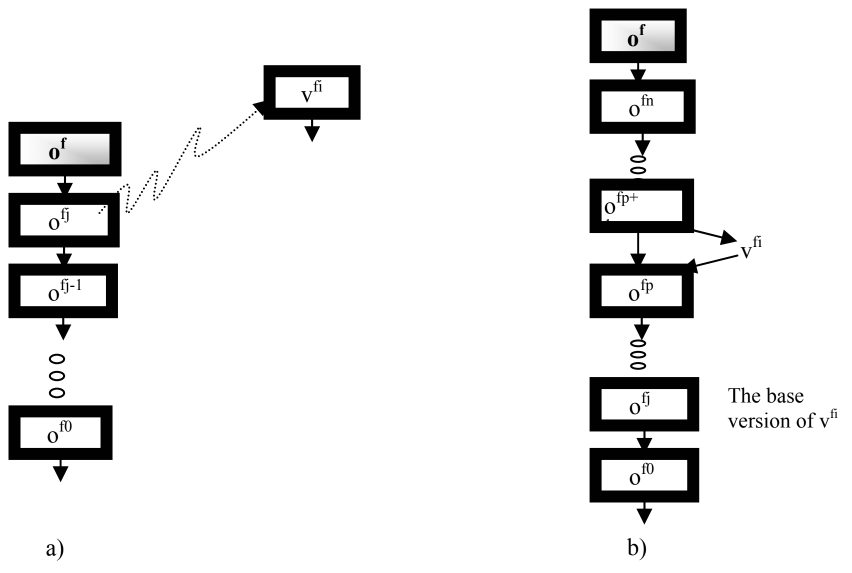


Fig. 4. Finding a position in the chain

When UT_i terminates, the Validation Processor checks if v^{fi} can be inserted in the proper position in the chain. This process starts from the top of the chain where o^{fn} is located and proceeds down until either a position is found to insert v^{fi} or no position can be found. v^{fi} can be inserted between any two committed versions o^{fp} and o^{fp+1} if:

1. committed transactions UT_0, UT_1, ..., UT_p can be serialized before UT_i in object family *f,* and
2. committed transactions UT_{p+1}, UT_{p+2},..., UT_n can be serialized after UT_i in object family *f.*

The first condition holds if UT_i (the user transaction associated with v^{fi}) can read the same information from o^{fp} as it originally read from o^{fj}. The second condition can be satisfied if for every data item x in the object family *f,* in which $UT_{p+1,}$ UT_{p+2},... UT_n reads x from one of UT_0, UT_1,... UT_p, the value of x should not have been modified by UT_i in v^{fi}. For example, suppose x is read by UT_{p+1} ($x \in$ *readset*(o^{fp+1}) and it was modified last by UT_p in o^{fp} ($x \in$ *writeset*(o^{fp}). If v^{fi} does not contain the same value for x as o^{fp} does, UT_{p+1} cannot be serialized after UT_i in object family *f.*

x in readxset(o^{fk})	x in writeset(v^{fi})	x in U writeset(v^{fs})	Allowed
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1/0
1	1	1	1

A

x in writeset(o^{fm})	x in writeset(v^{fi})	x in U writeset(v^{fs})	Allowed
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1/0
1	1	1	1

B

Fig. 5. Possible cases when reconciliation may or may not succeed

Figure 5A shows all possible cases that arise when the scheduler attempts to insert v^{fi} below o^{fk} ($o^{fp+1} \le o^{fk} \le o^{fn}$). Similarly, Fig. 5B checks if it is possible to place v^{fi} above o^{fm} ($o^{f0} \le o^{fm} \le o^{fp}$). The truth tables in Figure 5 show that v^{fi} can be inserted in the chain between o^{fp} and o^{fp+1} if:

$\forall\ o^{fk}\ (o^{fp+1} \le o^{fk} \le o^{fn})$

if $x \in$ *readset*(o^{fk}) **and** $x \in$ *writeset*(v^{fi}) and $x \notin \cup_{s=p+1}^{k-1}$ *writeset*(o^{fs}) **then**
UT_k reads the same value for x as UT_i writes into x.

and

$\forall\ o^{fm}\ (o^{f0} \le o^{fm} \le o^{fp})$

if $x \in$ *writeset*(o^{fm}) **and** $x \in$ *readset*(v^{fi}) and $x \notin \cup_{s=m+1}^{p}$ *writeset*(o^{fs}) **then**
UT_i reads the same value for x as UT_m writes into x.

Lines 1 through 14 in Fig. 6 attempt to find a proper position in the version-chain where v^{fi} can be inserted. The process starts from the top of the version-chain (lines 2-3) and proceeds down the version-chain (lines 5-7). To determine if v^{fi} can be inserted above a committed version o^{fp}, it must be checked to be sure that every data item x^{fi} read in v^{fi} is still unchanged in o^{fp}.

Simple comparison of each data item x^{fi} read in v^{fi} with its corresponding x^{fp} in o^{fp} is not possible because during the manipulation of v^{fi}, x^{fi} may have been modified. Thus, when a version is created, a snapshot of its original state (*BeforeImage* of the version) is preserved before the version is modified. The *BeforeImage* contains the original values of the attributes in the version. Therefore, to ensure that all of the data items read from v^{fi} are still unchanged in o^{fp}, x^{fp} in o^{fp} is compared with the value of x^{fi} in the BeforeImage of v^{fi} (line 4).

```
Algorithm for Decision Manager
begin
!!!!!!!!! CHECKING INTRA-OBJECT SERIALIZABILITY
for every $v^{fi} \in VRLST(UT_i)$ do                                                   (1
    pos ← 0                                                                              (2
    $o^{fp} \leftarrow o^{fn}$ !!! STARTING FROM TOP OF THE CHAIN                        (3
    if $\exists\ x^{fi} \in readset(v^{fi})$ such that $BeforeImage(x^{fi}) \neq x^{fp}$ then    (4
      pos ← pos +1                                                                       (5
      $o^{fp} \leftarrow$ next committed version (going down the chain)                  (6
      go to line 4                                                                       (7
    if a $o^{fp}$ is found, then                                                         (8
      for every $o^{fk}$ that occurs above $o^{fp}$ in the chain                         (9
      if $\exists\ x^{fk} \in readset(o^{fk})$ such that $x^{fi} \in writeset(v^{fi})$ and $x^{fk} \neq x^{fi}$ then   (10
        $\forall\ o^{fs}\ (o^{fp+1} \leq o^{fs} \leq o^{fk})$, if $x^{fs} \notin writeset(o^{fs})$ then            (11
            Abort $UT_i$                                                                 (12
    or else
    Abort $UT_i$                                                                         (13
  $ValidPos[v^{fi}] \leftarrow pos$                                                      (14
!!!!!!!!! CHECKING INTER-OBJECT SERIALIZABILITY
for every $v^{fi} \in VRLST(UT_i)$ do                                                    (15
  for every $o^{fk}$ that occurs above $v^{fi}$ in the chain do                          (16
    if $readset(v^{fi})$ value-∩ $writeset(o^{fk}) \neq \{\}$ OR $readset(o^{fk})$ value-∩ $writeset(v^{fi}) \neq \{\}$ then   (17
        add an edge from $UT_k$ to $UT_i$ in $GSG$                                       (18
  for every $o^{fm}$ that occurs below $v^{fi}$ in the chain do                          (19
    if $readset(v^{fi})$ value-∩ $writeset(o^{fm}) \neq \{\}$ OR $readset(o^{fm})$ value-∩ $writeset(v^{fi}) \neq \{\}$ then   (20
      add an edge from $UT_k$ to $UT_i$ in $GSG$                                         (21
  if there is a cycle in $GSG$ then                                                      (22
    Abort $UT_i$                                                                         (23
Send $VRLST(UT_i)$ and $ValidPos[]$ to the Commit Manager                               (24
end
```

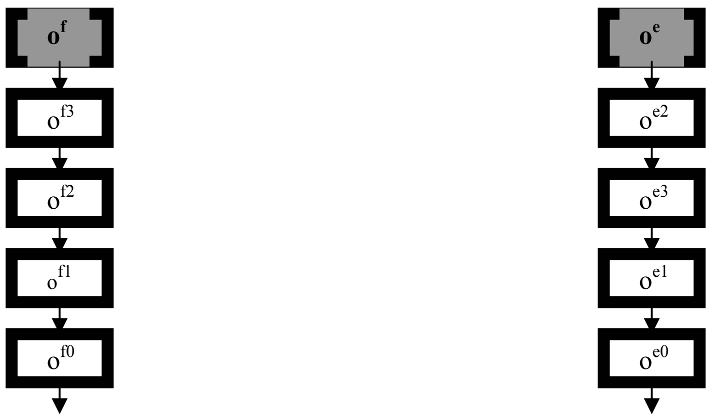
Fig. 6. The Decision manager

If a proper o^{fp} is found, the Decision Manager has to check whether v^{fi} can be placed below the committed versions o^{fp+1}, o^{fp+2},... o^{fn} in the version chain (i.e: serializing UT_{p+1} , UT_{p+2}, ..., UT_n after UT_i in object family *f*). This process starts from o^{fp+1} and proceeds up to o^{fn}. For every o^{fk} between o^{fp+1} and o^{fn} in the version-chain, if *x* is a variable that is read in o^{fk} and it has been last modified by one of o^{f0}, o^{f1}, ... o^{fp}, the value of *x* read in o^{fk} becomes invalid if UT_i writes a different value into *x* in v^{fi} (lines 8-12). If a valid position for the insertion of v^{fi} is found, this position is recorded in a data structure called *ValidPos* (line 14). Otherwise, if valid positions cannot be found for some of the active versions of UT_i, UT_i is aborted (lines 12,13).

Inter-object serializability: The above process only ensures intra-UT serializability at each object family referenced by UT_i. The Decision Manager must also check if UT_i is inter-object serializable with respect to other committed transactions in the objectbase. The following example describes a situation when inter-UT serializability may not be ensured.

Suppose three user transactions UT_1, UT_2, and UT_3 each accessing both object family *f* and object family *e*. UT_1 starts first, commits, and creates committed versions o^{f1} and o^{e1} in the objectbase. Then UT_2 and UT_3 start, make copies (active versions) of the committed versions created by UT_1 in object families *f* and *e*, and execute concurrently.

Figure 7 shows the state of the object families *f* and *e* after UT_2 and UT_3 commit. Clearly, since o^{f2} occurs before o^{f3} ($UT_2 \rightarrow UT_3$) in object family *f* (Fig. 7a) and o^{f3} occurs before o^{f2} ($UT_3 \rightarrow UT_2$) in object family *e* (Fig. 7b), inter-object serializability may not be ensured. The *Global Serialization Graph* provided by Zapp and Barker [11] can be used to control inter-object serializability.



a) Object family f b) Object family e

Fig. 7. Example of a possible inter-object serialization Problem

The Global Serialization Graph for a set of user transactions T = $\{UT_1, UT_2,..., UT_n\}$ denoted *GSG*(T) is an acyclic graph (*V,E*), where *V* is a set of vertices and *E* is a set of edges. Each vertex $v_i \in V$ represents a user transaction UT_i and an edge from v_i to v_j indicates that there exist a committed version of an object family *f*, o^{fi}, associated with UT_i which occurs above the committed version o^{fj} associated with UT_j in the version-chain of object family *f* and UT_i and UT_j have accessed some value-conflicting operations in o^{fi} and o^{fj}, respectively. Thus, if there is an edge going from v_i to v_j in *GSG*(T) then there must be a pointer going directly or indirectly from o^{fi} to o^{fj}. An edge in the global serialization graph shows the serialization order of two user transactions.

We now define the *value-*∩ (*value-intersection*) of two sets *A* and *B* to be a set $C=\{x_1, x_2,..., x_n\}$ if for $1 \le i \le n$, $x_i \in A$, $x_i \in B$, and the value of x_i in *A* is not equal to the value of x_i in *B*. Consider Fig. 4 again. Let o^{fk} be a committed version in the version-chain located between o^{fp+1} and o^{fn}. An edge from UT_k to UT_i is added to the global serialization graph if the following condition holds.

$$\textit{writeset}(o^{fk})\ \textit{value-}\cap\ \textit{readset}(v^{fi}) \neq \{\}\ \textbf{OR}$$
$$\textit{writeset}(v^{fi})\ \textit{value-}\cap\ \textit{readset}(o^{fk}) \neq \{\}$$

Similarly, let o^{fm} be a committed version located between o^{f0} and o^{fp}. An edge from UT_i to UT_m is added to the *GSG*(T) if the following condition holds:

$$\textit{writeset}(v^{fi})\ \textit{value-}\cap\ \textit{readset}(o^{fm}) \neq \{\}\ \textbf{OR}$$
$$\textit{writeset}(o^{fm})\ \textit{value-}\cap\ \textit{readset}(v^{fi}) \neq \{\}$$

When the edges are added to the *GSG*(T), the graph is checked for a cycle. As long as no cycle is detected, inter-object serializability is ensured. When both intra-object and inter-object serializabilities are ensured *VRLST*(UT_i) and *ValidPos* are passed to the Commit Manager (see Fig. 6 lines 15-24).

Commit manager: Before an active version v^{fi} is added between o^{fp} and o^{fp+1}, in the version-chain, the Commit Manager must ensure that insertion of v^{fi} in the chain leaves other committed versions in a consistent state. Note that during the execution of UT_i against v^{fi}, some other user transactions may operate on the data items in their own versions in which UT_i does not access the corresponding data in v^{fi}.

Thus when v^{fi} is inserted above o^{fp}, the Commit Manager must check if the value of every attribute x^{fi} in v^{fi} that is not accessed by UT_i is the same as its corresponding attribute x^{fp} in o^{fp}. If this is not the case, the value of x^{fp} is propagated to x^{fi} overwriting the old value of x^{fi} (Fig. 8 lines (1-6)).

An example is shown in Fig. 9. Suppose UT_i has obtained a copy of o^{f0} and during its lifetime other committed versions o^{f1}, o^{f2}, and o^{f3} have been created (9A). Suppose the Decision Manager has found that

v^{fi} can be inserted between o^{f1} and o^{f2} (9B). Although attribute c^{fi} in v^{fi} has not been accessed by UT_i, the value of c^{f1} has to overwrite the value of c^{fi} in order to serialize UT_i after UT_1 in the object family *f*.

```
Algorithm for commit manager
begin
for every $v^{fi} \in VRLST(UT_i)$ do                                                         (1
  level ←ValidPos[$v^{fi}$]                                                                   (2
  $o^{fp}$ ← the committed version in object family $f$ that is supposed to occur just below $v^{fi}$     (3
  for every data $x^{fi}$ in $v^{fi}$ and corresponding $x^{fp}$ in $o^{fp}$ do                          (4
    if $x^{fi} \neq x^{fp}$ and $x^{fi} \notin (readset(v^{fi}) \cup writeset(v^{fi}))$ then             (5
      $x^{fi} \leftarrow x^{fp}$                                                              (6
  for each $x^{fi} \in writeset(o^{fi})$ do
    for each $o^{fk}$ that occurs above $o^{fi}$ do                                          (7
      if $x^{fk} \notin writeset(o^{fk})$ then                                               (8
        $x^{fk} \leftarrow x^{fi}$ ! PROPAGATING THE VALUES                                   (9
      else
        break;                                                                                (10
  promote $v^{fi}$ to $o^{fi}$ and store it in $ValidPos[v^{fi}]$
Commit $UT_i$                                                                                 (11
end
```

Fig. 8. The commit manager

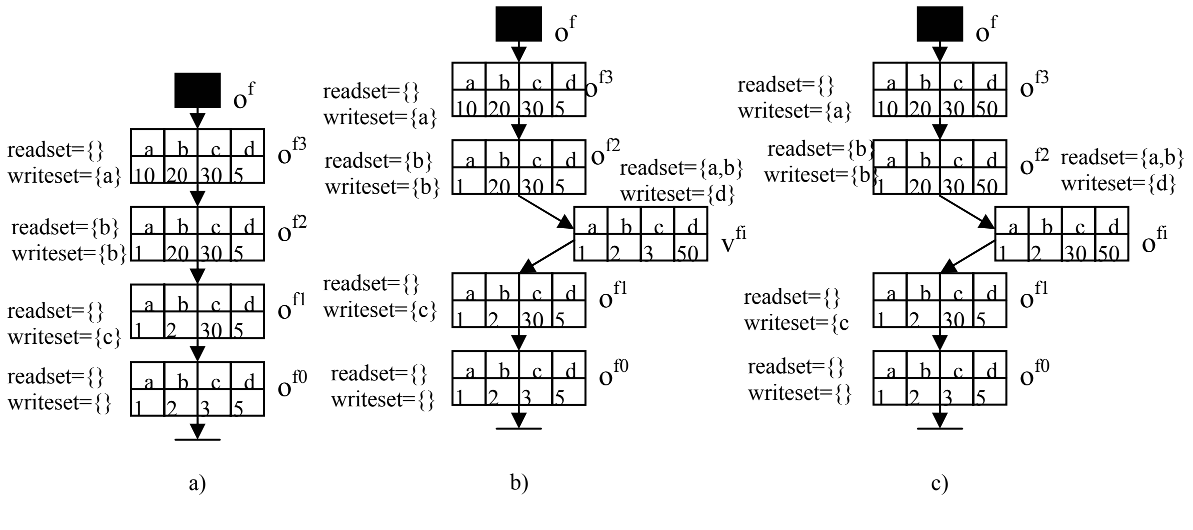


Fig. 9. Propagation of the values to higher level committed versions

Further, consider Fig. 9B. If a user transaction requests a copy of the most recent information in the object family *f*, a copy of o^{f3} is obtained. However, o^{f3} in Fig. 9B does not include all of the recent information after v^{fi} joins the object family *f*. The reason is that the value of *d* in o^{f3} is 5, which is not the most recent value of *d* in the object family *f*. The most recent value of *d* is 50, seen only in v^{fi}. This problem can be solved by propagating the values of the variables in the writeset of v^{fi} to their corresponding variables of the other committed versions which occur above it (Fig. 8 lines 7-10).

Finally, v^{fi} is promoted to the committed version o^{fi} and inserted in the proper position in the version chain. When all active version transactions of user transaction UT_i are promoted to committed versions and recorded in the objectbase, the Commit Manager stores the promoted active versions referenced by UT_i in the objectbase and commits UT_i (Fig. 8 line 11). Figure 9C shows the correct states of all committed versions in version family *f*.

Complex reconciliation: Figure 10 shows an example of the case when simple reconciliation of a user transaction is not successful and complex reconciliation is required.

Figure 10A shows the state of the object family *f* before UT_1, UT_2 and UT_3 start executing. o^f contains the attributes $A^f=\{a,b,c,d,e\}$ and the methods $M^f=\{m_1^f,\ m_2^f,\ m_3^f\}$. UT_1 starts, obtains a copy of o^{f0} called v^{f1}, and executes m_1^f against v^{f1}. Then UT_2 and UT_3 start and each obtains a copy of o^{f0}. UT_2 executes m_2^f against its own copy v^{f2} and UT_3 executes m_3^f against v^{f3} (Figure 10B). Suppose UT_1 and UT_2 commit, thereby creating new committed versions o^{f1} and o^{f2}, respectively. Now UT_3 terminates. It is not possible to serialize UT_3 after UT_1 and UT_2 because the value of $b=20$ read by UT_3 is stale since UT_2 has modified *b* ($b=50$ in o^{f2}). Similarly, the value of $a=10$ read by UT_3 is stale since UT_1 has modified *a* ($a=40$ in o^{f1}). Thus v^{f3} can neither be placed above o^{f2} nor between o^{f1} and o^{f2} in the version-chain of object family *f*. Further, placing v^{f3} between o^{f0} and o^{f1} serializes UT_1 after UT_3. This cannot be done either because UT_1 reads attribute *c* which is being modified by UT_3 in v^{f3}. Since v^{f3} can not be located anywhere in the version-chain, it is considered an invalid version. Note that the stale data read by UT_3 are *a* and *b*. To validate v^{f3}, the new values should be read for *a* and *b* from o^{f2} (the last committed version in object family *f*) and the code related to *a* and *b* in m_3^f should be re-executed. An algorithm is now developed to show how to generate a reconciliation procedure for a method which accepts a set of stale data and re-executes only the code related to the stale data.

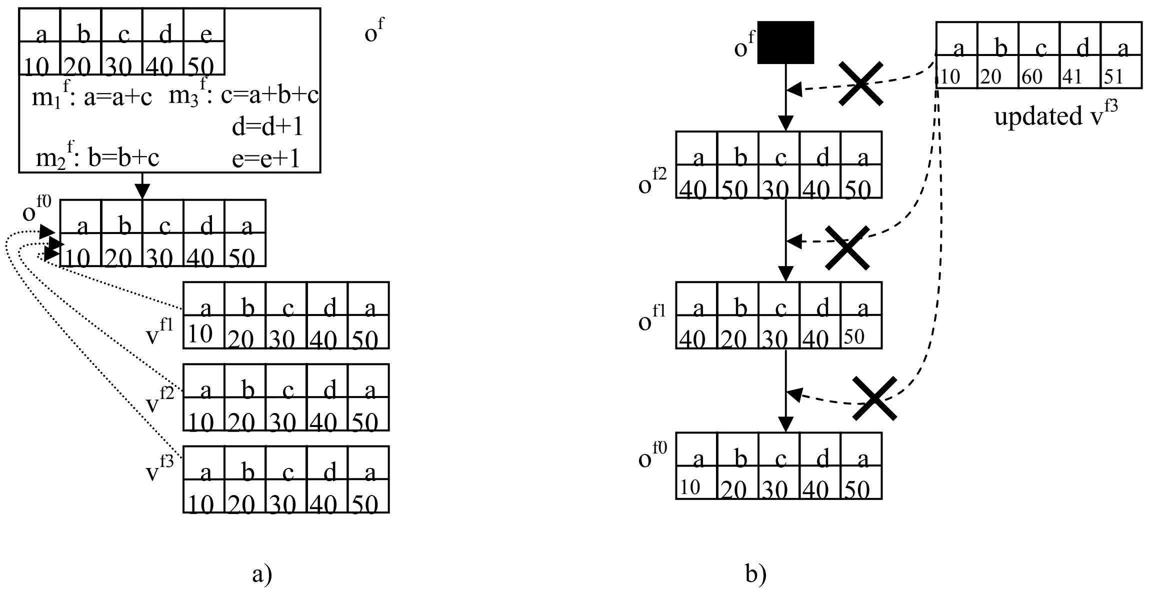


Fig. 10: Case when simple reconciliation fails and complex reconciliation is required

If a statement s_i in a method should be re-executed, related statements to s_i should also be found and re-executed. The data dependency [12], [13] relation between the statements in a method is determined by considering the *Three Address Code* for the method. The three address code is an encoded program in which all of the complex statements have been decomposed to their simplest form and cannot be decomposed further [14]. Section 5.2.1 explains complex reconciliation for *simple methods* where routines include simple read and write statements. Then the approach is extended in Section 5.2.2 for complex methods which may contain loops and other conditional statements.

Simple methods: Some notation is required. Suppose m_k^f is a method of object o^f. $WS(m_k^f)$ and $RS(m_k^f)$ denote the writeset and readset of attributes written and read, respectively, by m_k^f. These are generated conservatively at compile time [15]. $3m_k^f$ denotes the corresponding three address code for m_k^f and s_i refers to statement *i* in $3m_k^f$. $rs(s_i)$ and $ws(s_i)$ denote the readset and the writeset of s_i, respectively. If s_i and s_j are two statements in $3m_k^f$, $s_i<s_j$ indicates that s_i precedes s_j in $3m_k^f$. Similarly, $s_i>s_j$ means s_i follows s_j in $3m_k^f$.

The following data structures are associated with method m_k^f.

TACMkf[]: is a one dimensional array of records. *TACMkf* (Three Address Code for m_k^f) represents $3m_k^f$. Each record contains four fields: *op, arg1, arg2*, and *result* where *op* is the operation and other fields contain the operands.

FinalWriteMkf[]: is a one dimensional array of integers. The index of each element corresponds to a variable $x \in RS(m_k^f)$. If $FinalWriteMkf[x] = s_i$, s_i is the last statement of $3m_k^f$ that modifies *x*. i.e: s_i makes the final write operation on *x*.

ReadsFromMkf[ ]: is a two dimensional array of integers. Each row corresponds to a variable $x \in RS(m_k^f)$ and each column corresponds to a statement of $3m_k^f$. If $ReadsFromMkf[x,s_i]=s_j$, s_i reads *x* from s_j (i.e: s_j is the last statement that modifies *x* prior to the execution of s_i).

bitstringMkf[]: is a one dimensional array of bits. A bitstring is associated with each variable $x \in RS(m_k^f)$ *(bitstringMkf[x])*. The length of each bitstring is equal to the size of the array *TACMkf*. The value of each bit is either *1* or *0*. A *1* in the i^{th} position of the *bitstringMkf[x]* indicates that the s_i should be re-executed if *x* is stale.

Figure 11 shows a three address code for $3m_k^f$, and the arrays *TACMkf*, *FinalWriteMkf*, and *ReadsFromMkf*. *TACMkf* is created by the compiler. The values in arrays *ReadsFromMkf* and *FinalWriteMkf* can be calculated by scanning through *TACMkf*. The following discusses how to find the values of bits in each element of *bitstringMkf*.

s1:k = k+c
s2:h=h−2
s3:x=x*k;
s4:c=x*h
s5:m=m*70
s6:j=j/m
s7:k=x*3
s8:m=m+1
s9:m=m/h

a) 3 Mfk

	<result arg1 op arg2>			
s1:	k	k	+	c
s2:	h	h	−	2
s3:	x	x	*	k
s4:	c	x	*	h
s5:	m	m	*	70
s6:	j	j	/	m
s7:	k	x	*	3
s8:	m	m	+	1
s9:	m	m	/	h

b) TACMkf [ ]

	s1	**s2**	**s3**	**s4**	**s5**	**s6**	**s7**	**s8**	**s9**
k			s1						
c									
h				s2					s2
x				s3			s3		
m						s5		s5	s8
j									

c) ReadsfromMkf[ ]

k	s7
c	s4
h	s2
x	s3
m	s9
j	s6

d) FinalWriteMkf[ ]

Fig. 11. The three address code for m_k^f and associated data structure

```
$m_1^f$:          $m_2^f$:          $m_3^f$:          $m_4^f$:          $m_5^f$:
------------------------------------------------------------------------------------------------
|$s_1$:.....      |$s_1$:.....      |$s_1$:.....      |$s_1$:.....      |$s_1$:.....
|.....            |.....            |.....            |.....            |.....
|.....            |.....            |$s_f$: $p=m*n$   |$s_f$:$p=m*n$    |.....
|.....            |.....            |.....            |.....            |.....
|$s_{i-1}$:...    |$s_{i-1}$:...    |$s_{i-1}$:...    |$s_{i-1}$:...    |$s_{i-1}$:...
|$s_i$:$k=p*q$    |$s_i$:$k=p*q$    |$s_i$: $k=p*q$   |$s_i$: $k=p*q$   |$s_i$: $k=p*q$
|$s_{i+1}$:...    |$s_{i+1}$:...    |$s_{i+1}$:...    |$s_{i+1}$:...    |$s_{i+1}$:...
|.....            |.....            |.....            |.....            |.....
|.....            |$s_t$:$p=u*v$    |.....            |$s_t$:$p=u*v$    |$s_t$:$k=u*v$
|.....            |.....            |.....            |.....            |.....
|$s_n$:.....      |$s_n$:.....      |$s_n$:.....      |$s_n$:.....      |$s_n$:.....
------------------------------------------------------------------------------------------------
```

Fig. 12. Finding the related statements

Suppose user transaction UT_j has obtained a copy of the last committed version of o^f, v^{fj}, and has executed one of the methods m_k^f $(1 \le k \le 5)$ shown in Fig. 12 against v^{fj}. Suppose UT_j cannot be committed and reconciliation is required. If $s_i{:}k=p*q$ is a statement in m_k^f that is either directly or indirectly related to the stale data read by UT_j, s_i should be re-executed. Consider operand $p\in rs(s_i)$. There are four possible cases when the re-execution of s_i may cause the re-execution of other statements:

Case 1: p is neither modified in $s_1 .. s_{i-1}$ nor in $s_{i+1}.. s_n$
Case 2: p is unmodified in $s_1.. s_{i-1}$, but is modified in $s_{i+1}.. s_n$
Case 3: p is modified in $s_1 .. s_{i-1}$, but is unmodified in $s_{i+1}.. s_n$
Case 4: p is both modified in $s_1.. s_{i-1}$ and in $s_{i+1}.. s_n$

In Cases 1 and 2, since p is not modified prior to s_i, a new value for p is reread from the objectbase and re-execution of s_i may only cause re-execution of the statements which relate directly or indirectly to the other two operands k and q. In Case 3, let s_f $(s_1\le s_f \le s_{i-1})$ be the last statement that modifies p before s_i. Since p is not modified in $s_{i+1}.. s_n$, s_f is the statement that makes the final write on p during the execution. This value of p that is created by s_f is available in the updated version v^{fj} and is reread when s_i is re-executed. In Case 4, the correct value of p is neither available in the objectbase nor in v^{fj}. This is because p is modified both before and after the execution of s_i. Let s_f be the last statement that modifies p prior to s_i, and s_t be the statement that issues the final write on p during the execution. In order to re-execute s_i, s_f must also be re-executed to calculate the value of p that should be read by s_i. However, re-execution of s_f overwrites the value of p that is in the v^{fj}. Thus s_t must also be re-executed so that the correct final value of p can be re-recorded in the objectbase when re-execution terminates (Note: operation $q\in rs(s_i)$ is handled analogously).

Now consider the following case for operand $k\in ws(s_i)$. In Cases (1-5) shown in Fig. 12, when s_i is re-executed, every other statement in $s_{i+1} .. s_n$ that reads the value of k produced by s_i must be re-executed. Further, in Case 5, since s_t overwrites the value of k that is written earlier by s_i $(s_t > s_i)$, the statement that executes the final write on k must also be re-executed.

The algorithm: Suppose $x\in RS(m_k^f)$ and set X, contain the statements in $3m_k^f$ which read the value of x from the objectbase. If it is found that x is stale, every statement $s_i \in$ X plus other statements in $3m_k^f$ which directly or indirectly relate to each s_i should be re-executed. Procedures *BitString* and *FindRelatedCode* shown in Fig. 13 and Fig. 14, respectively, find the statements which directly or indirectly relate to each variable $x \in RS(m_k^f)$.

```
Procedure Bitstring($RS(m_k^f)$)
begin
for every $x \in$ RS($m_k^f$) do                                                    (1)
  X← set of statements in 3 $m_k^f$ that read $x$ from the objectbase                (2)
  for every $s_i \in$ X do                                                          (3)
    FindRelatedCode($s_i$, ReadsFromMkf[], FinalWriteMkf[], bitstringMkf[$x$])       (4)
end
```

Fig. 13. Procedure BitString forsimple methods

First, for each variable $x\in RS(m_k^f)$, procedure *BitString* finds the statements which are related *directly* to x (Figure 13 lines 1-2), and then calls procedure *FindRelatedCode* to search for the statements which are *indirectly* related to x (lines 3-4). The *FindRelatedCode* is a recursive procedure that accepts a statement s_i and marks statements which are related directly or indirectly to s_i (refer to Fig. 14). It contains three parts. First, statements in which s_i reads from them are recursively found and marked (lines 1-4). Then s_i is marked (line 5). Finally, given that $k \in ws(s_i)$, statements which read k from s_i are recursively selected and marked for re-execution (lines 6-9). In addition, the last statement that modifies k (*FinalWriteMkf*[k]) should also be marked for re-execution (lines 10-12).

```
Procedure FindRelatedCode($s_i$, ReadsFromMkf[], FinalWriteMkf[], var bitstringMkf[x])
begin
for every y ∈ rs($s_i$) do                                                        (1)
    $s_j$ ← ReadsFromMkf[y, $s_i$]                                                (2)
    if ($s_j$ is not marked) and (FinalWriteMkf[y] > $s_i$) then                  (3)
        FindRelatedCode($s_j$, ReadsFromMkf[], FinalWriteMkf[], bitstringMkf[x])  (4)
set bit $s_i$ in bitstringMkf[x] to 1                                             (5)
Let k be such that k ∈ ws($s_i$) then                                             (6)
for every $s_j$ in which $s_i$ = ReadsFromMkf[k, $s_j$] do                        (7)
    if $s_j$ is not marked then                                                   (8)
        FindRelatedCode($s_j$, ReadsFromMkf[], FinalWriteMkf[], bitstringMkf[x])  (9)
$s_t$ ← FinalWriteMkf[k]                                                          (10)
if $s_t$ is not marked then                                                       (11)
    FindRelatedCode($s_t$ , ReadsFromMkf[ ], FinalWriteMkf[ ], bitstringMkf[x])   (12)
end
```

Fig. 14. Find related code procedure for simple methods

Procedure *ReconcilM-k-f* (see Fig. 15) accepts a set of stale data and re-executes the code related to these stale data. The bitstrings of all stale data are merged into one bitstring called *string* to find out what code should be re-executed (lines 1-3). The value of each bit in *string* is either *0* or *1*. A *1* in the i^{th} position of *string* indicates that s_i in $3m_k^f$ should be re-executed (lines 4-6).

```
Procedure ReconcilM-k-f(StaleData)
begin
string ← empty                                                     (1)
for every x ∈ StaleData do                                         (2)
    string ← string OR bitstringMkf[x]  !!!!!!! bit union of two string   (3)
for i = 1 to sizeof(TACMkf) do                                     (4)
    if $i^{th}$ bit of string = 1 then                             (5)
        re-execute $s_i$  !!!ACTUAL RE-EXECUTION TAKES PLACE IN HERE     (6)
end
```

Fig. 15. The reconciliation procedure for method m_k^f

Complex methods: In methods which include conditional statements, finding the values of the bitstrings is complicated by two problems. First, it may not be possible to determine all of the values of *ReadsFromMkf* and *FinalWriteMkf* arrays at compile time for a method m_k^f. For example in Figure 16A, the last write operation on *p* is either s_4 or s_6 depending on whether $a>b$ or not. Also it is not possible to know if s_7 reads *c* from s_3 or s_5 until the code is executed. To solve this problem, statements related to a variable (such as *c* in the example) can be selected conservatively for re-execution in both if and else blocks.

The second problem is that, although some statements are not affected directly or indirectly by the stale data, they may still have to be re-executed. For example, in Fig. 16B, when *k* is stale, only s_1, s_3, s_4, and s_5, should be re-executed. However, the final value of *t* in s_2 depends on the number of times the loop is executed. If the number of iterations in the loop changes during the re-execution, it changes the value of *t* calculated during the execution. Our approach is to consider all of the statements in a conditional block for re-execution, whenever a particular statement in that conditional block must be re-executed. This is excessively conservative and can be improved, but we leave this goal as a subject for future research.

We now turn our attention to improving these algorithms as more execution sequences can be reconciled, thereby increasing concurrency. This enhancement is performed by modifying some of the algorithms previously presented to support complex methods. We begin by introducing the concept of *control flow information* [16-18].

```
$s_1$: $k=p*10$
$s_2$: if ($a>b$) then
      {$s_3$: $c=x+1$
       $s_4$:$p=5$}
   else
      {$s_5$: $c=x+2$
       $s_6$: $p=10$}
$s_7$: $h=c*10$
```

a)

```
$s_1$:while ($a<b$) do
      $s_2$:$t=t+5$
      $s_3$: $a=k*2$
      $s_4$: $k=k+10$
$s_5$:endwhile
```

b)

Fig. 16. Finding related blocks using loop

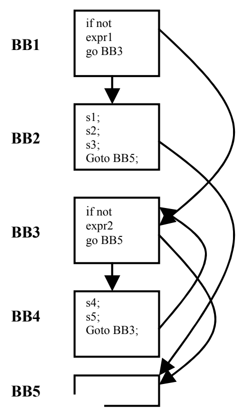
Control flow information: A method contains a collection of executable sections called *basic blocks*. A basic block is a sequence of consecutive steps entered at the beginning and exited at the end without halt or branching except on the last operation. An algorithm to derive the basic blocks of a program routine is presented by Aho, *et al* [16]. In brief, the algorithm accepts the *Three Address Code* of a method and determines the basic block *leaders*. A leader is the first statement of each basic block and is defined as the first statement in the program, or any statement that is the target of a conditional or unconditional goto, or any statement that immediately follows a goto or conditional goto statement. For each leader, its basic block consists of the leader and all statements up to, but not including the next leader or the end of the program.

A basic block may contain a branch statement which links the basic block to another basic block. Relationships between basic blocks are captured in a *control flow graph* [16]. The control flow graph of a method m_k^f is a directed graph $CFG(m_k^f) = (V,E)$ where each vertex $v_x \in V$ represents a basic block x (BB_x) and an edge from v_x to v_y indicates that the control may pass from BB_x to BB_y in m_k^f. A control flow graph does not provide any information regarding how many times a basic block may be visited (reflecting the loop structure). It only shows if and when a basic block may be executed. Figure 17 provides an example of a program, and its corresponding control flow graph.

In a particular execution of a method, only a subset of the basic blocks is visited. This is because, based on the current state of the object, the control (conditional) statements may prevent the execution of some basic blocks. For example, in Fig. 17B, if *expr1* is true during the execution, BB_2 is visited; otherwise, control is passed to BB_3. A sequence of basic blocks visited during an execution of a method forms a *control flow path* [16]. A Control Flow Path (CFP_x) through method m_k^f is a sequence of basic blocks $<BB_{ko}, BB_{k1},..., BB_{k(n-1)}>$ where BB_{ko} is the entry node in the control flow graph for the method m_k^f, $BB_{k(n-1)}$ is an exit node of the graph, and there exists an edge from BB_{kc} to $BB_{k(c+1)}$ ($0 \leq c < n-1$) to indicate that the control flows directly from BB_{kc} to $BB_{k(c+1)}$.

```
if expr1 then
    s1;
    s2;
    s3;
else
    while expr2 do
            s4;
            s5;
    endwhile;
endif;
```

a) The program routine



b) The control flow graph

Fig. 17. Control flow graph of a program segment

If a method contains *n* control statements, there are at most 2^n control flow paths. Prior to the execution of a method, it may not be possible to determine which path will be executed. However, it is possible to enumerate the control flow paths of a method at compile time. The following table shows all possible control flow paths through the program in Fig. 17A.

expr1	expr2	path
true	true	$<BB_1, BB_2, BB_5>$
true	false	$<BB_1, BB_2, BB_5>$
false	true	$<BB_1, BB_3, BB_4, BB_5>$
false	false	$<BB_1, BB_3, BB_5>$

In this example, there are only three distinct paths because when *expr1* is true, the result of *expr2* is irrelevant. Consider method m_k^f again. For each control flow path CFP_{kj} of m_k^f, we define $3CFP_{kj}$ to be the corresponding three address code for CFP_{kj}. *TACCFPkj*, *ReadsFromCFPkj*, and *FinalWriteCFPkj* are similarly associated with CFP_{kj}. The following additional data structure is associated with m_k^f:

CondBlkMkf[]: is a one dimensional array of records. Each record contains two fields: *begin* and *end*. The index of each element corresponds to a statement s_i in $3m_k^f$. If *CondBlkMkf*[s_i].begin = s_b and *CondBlkMkf*[s_i].end= s_e, s_i is within a conditional block where s_b and s_e are the start and the end of that conditional block. If s_i is within a nested conditional block, s_b and s_e represent the *begin* and the *end* of the outermost block. If s_b and s_e are zero, s_i is not in a conditional block.

condqueueMkf: is a queue of statements in $3m_k^f$. Every time a statement s_i is marked for re-execution, it is pushed into *condqueueMkf* if it is within a conditional block. When s_i is popped from the *condqueueMkf*, the begining and the end of its associated conditional block, s_b and s_e, are searched and all of the statements within s_b and s_e are selected for re-execution.

```
Procedure Bitstring($RS(m_k^f)$)
begin
for every $x \in RS(m_k^f)$ do                                                          (1)
  X ← set of statements in $3m_k^f$ that read $x$ from the objectbase                    (2)
  for every $s_i \in X$ do                                                                (3)
    for each $CFP_{kj}$ do                                                                (4)
      if $s_i$ is in the $CFP_{kj}$ then                                                  (5)
        FindRelatedCode($s_i$, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[$x$])    (6)

  !!!! FINDING THE STATEMENTS IN THE CONDITIONAL BLOCK                                    (7)
  while condqueueMkf is not empty do                                                      (8)
    $s_k$ ← pop an element form condqueueMkf                                              (9)
    $s_b$← CondBlkMkf[$s_k$].begin                                                        (10)
    $s_e$ ← CondBlkMkf[$s_k$].end                                                         (11)
    for $s_i$= $s_b$ to $s_e$ do                                                          (12)
      for each $CFP_{kj}$ do                                                              (13)
        if $s_i$ is in the $CFP_{kj}$ then                                                (14)
          FindRelatedCode($s_i$, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[$x$])  (15)
end
```

Fig. 18. Procedure bit string for complex methods

The *BitString* procedure requires a major modification. In Fig. 13, statements related to each $x \in RS(m_k^f)$ are searched within m_k^f only. This is because m_k^f consists of only a single basic block. This is extended in Fig. 18(lines 1-6) where related statements to each $x \in RS(m_k^f)$ are searched for along all possible control flow paths in m_k^f. In addition, if any statement s_i that is directly or indirectly related to x is within a conditional block, all statements in that conditional block are considered to be related to x. The *FindRelatedCode* procedure has been slightly modified to find statements such as s_i and put them in *condqueueMkf* (Fig. 19 lines 5-8). The *BitString* procedure pops statements such as s_i from the *condqueueMkf*, searches for the beginning and the end of the conditional block associated with s_i, and processes all statements within that conditional block for re-execution (Fig. 18 lines 8-15).

```
Procedure FindRelatedCode($s_i$, ReadsFromCFPkj[], FinalWriteCFPkj[], var bitstringMkf[x])
begin
for every $y \in rs(s_i)$ do                                                        (1)
  $s_j \leftarrow$ ReadsFromCFPkj[y, $s_i$]                                           (2)
  if ($s_j$ is not marked) and (FinalWriteCFPkj[y] > $s_i$) then                      (3)
    FindRelatedCode($s_j$, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[x])      (4)
if $s_i$ is not marked then                                                          (5)
  set bit $s_i$ in bitstringMkf[x] to 1                                              (6)
  if (CondBlkMkf[$s_i$].start != 0) then !!! $s_i$ IS IN THE CONDITIONAL BLOCK        (7)
    push $s_i$ into condqueueMkf                                                     (8)
Let $k$ be such that $k \in ws(s_i)$                                                 (9)
for every $s_j$ in which $s_i$ = ReadsFromCFPkj[k, $s_j$] do                          (10)
  if $s_j$ is not marked then                                                        (11)
    FindRelatedCode($s_j$, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[x])      (12)
$s_t \leftarrow$ FinalWriteCFPkj[k]                                                   (13)
if $s_t$ is not marked then                                                          (14)
  FindRelatedCode($s_t$, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[x])        (15)
end
```

Fig. 19. Procedure find related code for complex methods

6. CONCLUSION

We have presented a formalism for describing multiversion objects and transactions on them. We have also presented architecture which can be used as the basis for the development of optimistic concurrency control protocols. In this paper, we showed that preserving previous states of an object can enhance concurrency. We also illustrated how to generate a reconciliation algorithm for the object models. Reconciliation procedures accept a set of "incorrect" data items, but make them consistent with the rest of the information in the objectbase. We discussed what a compiler requires so it can generate the reconciliation algorithms based on static analysis information. Future work may relax some constraints and improve the reconciliation algorithms. This paper does not address the issues related to the reliability of the system. Perez [19], [20] and Wang [21] suggested developed additional recovery components that could be added to the architecture, but do not consider reconciliation into account in detail. A proper recovery manager algorithm should be developed to ensure the system reliability at least in case of transaction and system failures. Further, this work may be improved to extend this research to a distributed objectbase environment

REFERENCES

1. Bernstein, P., Goodman, N. & Hadzilacos, V. (1987). *Concurrency control and recovery in database systems*. Addison Wesley, Reading, Massachusetts.
2. Gray & Reuter. (1994). *Transaction processing concepts*. Addison-Wesley Publishing Company.
3. Bhalla S.(2003). Asynchronous operations in distributed concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 721–733.
4. Nakajima. T. (1992). Commutativity based concurrency control for multiversion objects. *Proceedings of the International Workshop on Distributed Object Management*, 101–119.

5. Graham, P. C. J. & Barker, K. E. (1994). Effective optimistic concurrency control in multiversion object bases. *Proceedings of the International Symposium on Object Oriented Methodologies and systems (ISOOMS),* Palermo, Italy, in Springer-Verlag LNCS-858, 313-328.
6. Hadaegh, A. & Barker, K. (1996). Value-serializability and architecture for managing transactions in multiversion objectbase systems. *Proceeding of Third International Workshop on Advances in Databases and Information Systems*, *1*, Moscow, 126–133.
7. Reed. D. (1978). Naming and synchronization in a decentralized computer system. *Technical report MIT/LCS/TR-205, MIT Laboratory for Computer Science.*
8. Weihl, W. E. (1988). Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers, 37*(12), 1488–1505.
9. Graham, P. J. (1994). Applications of static analysis to concurrency control and recovery in objectbase systems. Ph.D. thesis, University of Manitoba.
10. Moss, J. E. B. (1985). *Nested transactions–An approach to reliable distributed computing*. The MIT Press.
11. Zapp, M. E. & Barker, K. (1993). On concurrency control in object bases. *Mid-Continent Information Systems Conference* (MISC'93), 91–97, Fargo, USA.
12. Wolfe, M. (1989). *Optimizing super-compilers for supercomputers*. MIT Press.
13. Tripathy, S. & Panda, B. (2001). Post-intrusion recovery using data dependency approach. *Proceedings of the 2001 IEEE*, *Workshop on Information Assurance and Security*, 156–160, New York.
14. Allen, R. & Kennedy, K. (2001). Optimizing *compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers, second edition.
15. Graham, P. & Barker, K. (1994). Improved scheduling in object bases using statically derived information. *The International Journal of Microcomputer Applications*.
16. Aho, A., Sethi, R. & Ullman, J. (1989). *Compilers principles, techniques, and tools*. Addison-Wesley Publishing Company.
17. Li, T., John, L. K., Sivasubramaniam, A., Vijaykrishnan, N. & Rubio, J. (2002). Understanding and improving operating system effects in control flow prediction. *ACM Sigplan Notices, 37*(10), 68-80.
18. Kaspersky, K. (2003). *Code optimization: effective memory usage*. A-List Publishing.
19. Perez, B. & Hadaegh, A. (2003). Recovery in multiversion objectbase systems. *The 2003 International Conference on Emerging Technologies (ICET'03)*, Minneapolis, USA.
20. Perez, B. & Hadaegh, A. (2004). Building a reliable multiversion object-based system. *Hawaii International Conference in Computer Science (HICSS'04)*, Honolulu, Hawaii, USA.
21. Wang, J. & Hadaegh, A. (2005). Recovering the reconciled objects. *Proceedings of International Advanced Database Conference (IADC 2005) San Diego,* California, 49-58.