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 $o$ is called the last committed version of $o$ (denoted $LCV(o)$), and the most recent uncommitted version of $o$ is called the current version of $o$ (denoted $CV(o)$). When transaction $T_i$ invokes method $M_{f_k}$ on object $o$, a new uncommitted version of $o$ (denoted $NV(o)$) is created for $T_i$. If the return result from $NV(o)$ backward commutes with $CV(o)$ or forward commutes with $LCV(o)$, $NV(o)$ becomes the new current version of $o$ and replaces the old current version. Otherwise, $NV(o)$ 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, \( \sigma^1 \) and \( \sigma^2 \) of object \( \sigma' \) accessed by two transactions \( T_i \) and \( T_j \), respectively and provides a serialization order between \( T_i \) and \( T_j \). If versions \( \sigma^1 \) and \( \sigma^2 \) cannot be directly merged (due to the use of stale data), complex reconciliation is attempted. Complex reconciliation of two transactions, \( T_i \) and \( T_j \) 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 \( \sigma' \). 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 \( \sigma' \) (denoted \( v^i \)) begins as a copy of a persistent object and is manipulated independently of all other versions. Methods of object \( \sigma' \) are denoted as \( m^i_1, m^i_2, m^i_3, \ldots \), 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 \( m^i_1, m^i_2, \) and \( m^i_3 \), the version transactions generated from these three methods are denoted as \( VT^i_{1j}, VT^i_{2i}, \) and \( VT^i_{3i} \) respectively. In general, each version transaction \( VT^i_{jk} \) corresponds to the \( k^{th} \) version transaction of \( UT_i \) that should be executed against the version of object \( \sigma' \) called \( v^h \).

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 \( o' \), we denote \( v^\beta \) and \( o^\beta \) to represent an active version, and a committed version respectively.

Creating an active version, \( v^\beta \), 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 \( v^\beta \) requires recording \( v^\beta \) as \( o^\beta \) 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 \( \text{depends} \) 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](image)

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 \( (v^\beta) \), referred by its object family \( id \) in the version list with the most recent committed versions of object family \( f \) \( (o^\delta) \)'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 \( v^\beta \) is consistent with the state of other committed versions of \( o' \) if the attributes accessed in \( v^\beta \) have not been accessed since \( v^\beta \) was created. If this is the case for every active
version $v^i$ 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.

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 $o^0$ denoted by $v^0$ and $v^4$, respectively. $UT_1$ executes $m^1_f$ against $v^0$ and $UT_4$ executes $m^4_f$ against $v^4$. $UT_1$ commits and its associated committed version $o_1^1$ is recorded in the objectbase (Fig. 3b). Then $UT_2$ starts, obtains a copy of $o^1$ ($v^2$), executes $m^2_f$ against $v^2$, and commits. $v^2$ is promoted to $o_2^2$ and is recorded in the objectbase (Fig. 3c). Next $UT_3$ starts, obtains a copy of $o^2$ ($v^3$), executes $m^3_f$ against $v^3$ and commits. $v^3$ is promoted to $o_3^3$ and is recorded in the objectbase (Fig. 3D). Now $UT_4$ terminates; however, $v^4$ cannot be committed as the last committed version in the chain because the value of $c$ read in $v^4$ is stale. Although $v^4$ 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 $v^4$ 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 $v^i_j$ to $o^i_j$ and locating $o^i_j$ 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 $v^i$ originates from $o^i$ 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 $v^i_j$. $ValidPos[v^i]$ refers to the position in the version-chain of object family $f$ where $v^i$ can be inserted.

Fig. 4. Finding a position in the chain
When $UT_i$ terminates, the Validation Processor checks if $v^\beta$ can be inserted in the proper position in the chain. This process starts from the top of the chain where $o^{\alpha_0}$ is located and proceeds down until either a position is found to insert $v^\beta$ or no position can be found. $v^\beta$ can be inserted between any two committed versions $o^\beta$ and $o^{\beta+1}$ 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 $v^\beta$) can read the same information from $o^{\alpha_0}$ as it originally read from $o^{\alpha_k}$. 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 $v^\beta$. For example, suppose $x$ is read by $UT_{p+1}$ ($x \in \text{readset}(o^{\alpha_{p+1}})$ and it was modified last by $UT_p$ in $o^\beta$ ($x \in \text{writeset}(o^\beta)$). If $v^\beta$ does not contain the same value for $x$ as $o^\beta$ does, $UT_{p+1}$ cannot be serialized after $UT_i$ in object family $f$.

\begin{align*}
\text{x in readset}(o^\beta) & \quad \text{x in write}\text{set}(o^\beta) & \quad \text{x in write}\text{set}(v^\beta) & \quad \text{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 \\
\end{align*}

\begin{align*}
\text{x in write}\text{set}(o^{fm}) & \quad \text{x in write}\text{set}(v^\beta) & \quad \text{x in write}\text{set}(v^\beta) & \quad \text{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 \\
\end{align*}

**A**

**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 $v^\beta$ below $o^k$ ($o^{\alpha_{k+1}} \leq o^k \leq o^{\alpha_n}$). Similarly, Fig. 5B checks if it is possible to place $v^\beta$ above $o^{fm}$ ($o^\beta \leq o^{fm} \leq o^\beta$). The truth tables in Figure 5 show that $v^\beta$ can be inserted in the chain between $o^\beta$ and $o^{\beta+1}$ if:

\[ \forall (o^{\alpha_{k+1}} \leq o^k \leq o^{\alpha_n}) \]

\[ \begin{align*}
& \text{if } x \in \text{readset}(o^k) \text{ and } x \in \text{writeset}(o^\beta) \text{ and } x \notin \bigcup_{s=p+1}^{k-1} \text{writeset}(o^s) \text{ then } \\
& UT_i \text{ reads the same value for } x \text{ as } UT_i \text{ writes into } x.
\end{align*} \]

\[ \forall (o^\beta \leq o^{fm} \leq o^\beta) \]

\[ \begin{align*}
& \text{if } x \in \text{writeset}(o^{fm}) \text{ and } x \in \text{writeset}(o^\beta) \text{ and } x \notin \bigcup_{s=m+1}^p \text{writeset}(o^s) \text{ then } \\
& UT_i \text{ reads the same value for } x \text{ as } UT_i \text{ writes into } x.
\end{align*} \]

Lines 1 through 14 in Fig. 6 attempt to find a proper position in the version-chain where $v^\beta$ 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 $v^\beta$ can be inserted above a committed version $o^\beta$, it must be checked to be sure that every data item $x^\beta$ read in $v^\beta$ is still unchanged in $o^\beta$.

Simple comparison of each data item $x^\beta$ read in $v^\beta$ with its corresponding $x^\beta$ in $o^\beta$ is not possible because during the manipulation of $v^\beta$, $x^\beta$ 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 $v^\beta$ are still unchanged in $o^\beta$, $x^\beta$ in $o^\beta$ is compared with the value of $x^\beta$ in the BeforeImage of $v^\beta$ (line 4).
Algorithm for Decision Manager
begin

!!!!!!!!!! CHECKING INTRA-OBJECT SERIALIZABILITY
for every $v^\beta \in VRLST(UT)$ do
  pos ← 0
  $o^\beta ← o^\beta$ !!! STARTING FROM TOP OF THE CHAIN
  if $\exists x^\beta \in \text{readset}(v^\beta)$ such that BeforeImage($x^\beta$) $\neq x^\beta$ then
    pos ← pos +1
    $o^\beta ← \text{next committed version (going down the chain)}$
  go to line 4
if a $o^\beta$ is found, then
  for every $o^\delta$ that occurs above $o^\beta$ in the chain
    if $\exists x^\beta \in \text{readset}(o^\beta)$ such that $x^\beta \in \text{writeset}(v^\beta)$ and $x^\beta \neq x^\beta$ then
      $\forall o^\delta (o^\delta \geq o^\beta)$, if $x^\beta \not\in \text{writeset}(o^\delta)$ then
        Abort UT$_i$
    or else
      Abort UT$_i$
      ValidPos[v] ← pos
    go to line 4
end

!!!!!!!!!! CHECKING INTER-OBJECT SERIALIZABILITY
for every $v^\beta \in VRLST(UT)$ do
  for every $o^\delta$ that occurs above $v^\beta$ in the chain do
    if $\text{readset}(v^\beta) \cap \text{writeset}(o^\delta) \neq \{\}$ OR $\text{readset}(o^\delta) \cap \text{writeset}(v^\beta) \neq \{\}$ then
      add an edge from UT$_i$ to UT$_j$ in GSG
    for every $o^m$ that occurs below $v^\beta$ in the chain do
      if $\text{readset}(v^\beta) \cap \text{writeset}(o^m) \neq \{\}$ OR $\text{readset}(o^m) \cap \text{writeset}(v^\beta) \neq \{\}$ then
        add an edge from UT$_i$ to UT$_j$ in GSG
      if there is a cycle in GSG then
        Abort UT$_i$
        Send VRLST(UT) and ValidPos[\{\} to the Commit Manager
    end
  end
end

Fig. 6. The Decision manager

If a proper $o^\beta$ is found, the Decision Manager has to check whether $v^\beta$ can be placed below the
committed versions $o^{\beta+1}$, $o^{\beta+2}$, ..., $o^\beta$ in the version chain (i.e: serializing UT$_{p+1}$, UT$_{p+2}$, ..., UT$_p$ after UT$_i$
in object family $f$). This process starts from $o^{\beta+1}$ and proceeds up to $o^\beta$. For every $o^\delta$ between $o^{\beta+1}$ and $o^\beta$ in
the version-chain, if $x$ is a variable that is read in $o^\delta$ and it has been last modified by one of $o^{\beta+1}$, $o^{\beta+2}$, ...
$o^\beta$, the value of $x$ read in $o^\beta$ becomes invalid if UT$_i$ writes a different value into $x$ in $v^\beta$ (lines 8-12). If a
valid position for the insertion of $v^\beta$ 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 $o^\beta$ and $o^\gamma$ 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 $o^\beta$
occurrns before $o^\beta$ (UT$_2$ $\rightarrow$ UT$_3$) in object family $f$ (Fig. 7a) and $o^\beta$ occurs before $o^\gamma$ (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.
Fig. 7. Example of a possible inter-object serialization Problem

The Global Serialization Graph for a set of user transactions \( T = \{ UT_1, UT_2, \ldots, UT_n \} \) 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 \( f \), \( of^i \), associated with \( UT_i \) which occurs above the committed version \( of^j \) 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 \( of^i \) and \( of^j \), 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 \( of^i \) to \( of^j \). An edge in the global serialization graph shows the serialization order of two user transactions.

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

\[
\text{writeset}(of^k) \cap \text{readset}(v_{fi}) \neq \{\} \text{ OR } \text{writeset}(v_{fi}) \cap \text{readset}(of^k) \neq \{\}
\]

Similarly, let \( of^m \) be a committed version located between \( of^0 \) and \( of^p \). An edge from \( UT_i \) to \( UT_m \) is added to the \( GSG(T) \) if the following condition holds:

\[
\text{writeset}(v_{fi}) \cap \text{readset}(of^m) \neq \{\} \text{ OR } \text{writeset}(of^m) \cap \text{readset}(v_{fi}) \neq \{\}
\]

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 \( \text{ValidPos} \) are passed to the Commit Manager (see Fig. 6 lines 15-24).

**Commit manager:** Before an active version \( v^j \) is added between \( v^0 \) and \( v^{p+1} \), in the version-chain, the Commit Manager must ensure that insertion of \( v^j \) in the chain leaves other committed versions in a consistent state. Note that during the execution of \( UT_i \) against \( v^j \), 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 \( v^j \).

Thus when \( v^j \) is inserted above \( v^0 \), the Commit Manager must check if the value of every attribute \( x^j \) in \( v^j \) that is not accessed by \( UT_i \) is the same as its corresponding attribute \( x^0 \) in \( v^0 \). If this is not the case, the value of \( x^0 \) is propagated to \( x^j \) overwriting the old value of \( x^0 \) (Fig. 8 lines (1-6)).

An example is shown in Fig. 9. Suppose \( UT_i \) has obtained a copy of \( v^0 \) and during its lifetime other committed versions \( v^1, v^2 \), and \( v^3 \) have been created (9A). Suppose the Decision Manager has found that
v\textsuperscript{o} can be inserted between \(\sigma^{f}\) and \(\sigma^{g}\) (9B). Although attribute \(c^{f}\) in \(v^{o}\) has not been accessed by \(UT_{i}\), the value of \(c^{o}\) has to overwrite the value of \(c^{f}\) in order to serialize \(UT_{i}\) after \(UT_{j}\) in the object family \(f\).

Algorithm for commit manager

begin
for every \(v^{o} \in VRLST(UT_{i})\) do
level <- ValidPos[\(v^{o}\)]
\(\sigma^{o} \leftarrow\) the committed version in object family \(f\) that is supposed to occur just below \(v^{o}\)
for every data \(x^{o}\) in \(v^{o}\) and corresponding \(x^{p}\) in \(\sigma^{o}\) do
if \(x^{o} \neq x^{p}\) and \(x^{o} \notin (\text{readset}(v^{o}) \cup \text{writeset}(v^{o}))\) then
\(x^{o} \leftarrow x^{p}\)
for each \(x^{o} \in \text{writeset}(\sigma^{o})\) do
for each \(o^{o}\) that occurs above \(o^{o}\) do
if \(x^{o} \notin \text{writeset}(o^{o})\) then
\(x^{ o} \leftarrow x^{p}\) ! PROPAGATING THE VALUES
else
break;
end
promote \(v^{o}\) to \(o^{o}\) and store it in \(\text{ValidPos}[v^{o}]\)
Commit \(UT_{i}\)
end

Fig. 8. The commit manager

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

Finally, \(v^{o}\) is promoted to the committed version \(o^{o}\) 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. $o'$ contains the attributes $A' = \{a, b, c, d, e\}$ and the methods $M = \{m_f^1, m_f^2, m_f^3\}$. $UT_1$ starts, obtains a copy of $o^0$ called $v^1$, and executes $m_f^1$ against $v^1$. Then $UT_2$ and $UT_3$ start and each obtains a copy of $o^0$. $UT_2$ executes $m_f^2$ against its own copy $v^2$ and $UT_3$ executes $m_f^3$ against $v^3$ (Figure 10B). Suppose $UT_1$ and $UT_2$ commit, thereby creating new committed versions $o^2$ and $o^2$, 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 $o^2$). Similarly, the value of $a=10$ read by $UT_3$ is stale since $UT_1$ has modified $a$ ($a=40$ in $o^2$). Thus $v^3$ can neither be placed above $o^2$ nor between $o^2$ and $o^2$ in the version-chain of object family $f$. Further, placing $v^3$ between $o^2$ and $o^2$ serializes $UT_3$ after $UT_3$. This cannot be done either because $UT_3$ reads attribute $c$ which is being modified by $UT_3$ in $v^3$. Since $v^3$ 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 $v^3$, the new values should be read for $a$ and $b$ from $o^2$ (the last committed version in object family $f$) and the code related to $a$ and $b$ in $m_f^3$ 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](image)

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

The following data structures are associated with method $m_f^j$.

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

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

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

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

Figure 11 shows a three address code for $3m_f^j$, and the arrays $TACM_f^j$, $FinalWriteM_f^j$, and $ReadsFromM_f^j$. $TACM_f^j$ is created by the compiler. The values in arrays $ReadsFromM_f^j$ and $FinalWriteM_f^j$ can be calculated by scanning through $TACM_f^j$. The following discusses how to find the values of bits in each element of $bitstringM_f^j$.

---

**Fig. 11.** The three address code for $m_f^j$ and associated data structure

---

**Fig. 12.** Finding the related statements
Suppose user transaction $UT_j$ has obtained a copy of the last committed version of $o'$, $v^\theta$, and has executed one of the methods $m_k^i$ ($1 \leq k \leq 5$) shown in Fig. 12 against $v^\theta$. Suppose $UT_j$ cannot be committed and reconciliation is required. If $s_i:k=p*q$ is a statement in $m_k^i$ 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 $s_i.. s_{i-1}$ nor in $s_{i+1}.. s_n$

**Case 2**: $p$ is unmodified in $s_i.. s_{i-1}$ but is modified in $s_{i+1}.. s_n$

**Case 3**: $p$ is modified in $s_i.. s_{i-1}$ but is unmodified in $s_{i+1}.. s_n$

**Case 4**: $p$ is both modified in $s_i.. s_{i-1}$ 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 $s_f$ ($s_f \leq s_i \leq s_{i-1}$) 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 $v^\theta$ 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 $v^\theta$. This is because $p$ is modified both before and after the execution of $s_f$. Let $s_f$ be the last statement that modifies $p$ prior to $s_i$, and $s_f$ be the statement that issues the final write on $p$ during the execution. In order to re-execute $s_f$, $s_f$ 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 $v^\theta$. Thus $s_f$ 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 $s_i$ overwrites the value of $k$ that is written earlier by $s_i$ ($s_i > s_f$), the statement that executes the final write on $k$ must also be re-executed.

**The algorithm**: Suppose $x \in RS(m_k^i)$ and set $X$, contain the statements in $3m_k^i$ which read the value of $x$ from the objectbase. If it is found that $x$ is stale, every statement $s_j \in X$ plus other statements in $3m_k^i$ which directly or indirectly relate to each $s_j$ 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 $x \in RS(m_k^i)$.

**Procedure** $BitString(RS(m_k^i))$

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

Fig. 13. Procedure BitString for simple methods

First, for each variable $x \in RS(m_k^i)$, 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).
Fig. 14. Find related code procedure for simple methods

\begin{verbatim}
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_j) then (3)
      FindRelatedCode(s_j, ReadsFromMkf[], FinalWriteMkf[], bitstringMkf[x]) (4)
  set bit s_i in bitstringMkf[x] to 1 (5)
  k ← ws(si) (6)
  for every si in which s_i = ReadsFromMkf[k, s_j] do (7)
    if s_i is not marked then (8)
      FindRelatedCode(s_j, ReadsFromMkf[], FinalWriteMkf[], bitstringMkf[x]) (9)
  s_i ← FinalWriteMkf[k] (10)
  if s_i is not marked then (11)
    FindRelatedCode(s_i, ReadsFromMkf[], FinalWriteMkf[], bitstringMkf[x]) (12)
end
\end{verbatim}

Fig. 15. The reconciliation procedure for method \( m^f \)

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 \( \text{ReadsFromMkf} \) and \( \text{FinalWriteMkf} \) arrays at compile time for a method \( m^f \). 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].
Reconciling instead of aborting unsuccessful transaction…

\[ s_1: k = p \times 10 \]
\[ s_2: \text{if} \ (a > b) \ \text{then} \]
\[ \{ s_3: c = x + 1 \]
\[ s_4: p = 5 \} \]
\[ \text{else} \]
\[ \{ s_5: c = x + 2 \]
\[ s_6: p = 10 \} \]
\[ s_7: h = c \times 10 \]
\[ s_8: \text{while} \ (a < b) \ \text{do} \]
\[ s_9: t = t + 5 \]
\[ s_{10}: a = k \times 2 \]
\[ s_{11}: k = k + 10 \]
\[ s_{12}: \text{endwhile} \]

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 \( m_i \) is a directed graph \( CFG(m_i) = (V,E) \) where each vertex \( v_x \in V \) represents a basic block \( BB_x \) and an edge from \( v_x \) to \( v_y \) indicates that the control may pass from \( BB_x \) to \( BB_y \) in \( m_i \). 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 \( m_i \) is a sequence of basic blocks \(<BB_{ko}, BB_{k+1},..., BB_{k(n-1)}>\) where \( BB_{ko} \) is the entry node in the control flow graph for the method \( m_i \), \( BB_{k(n-1)} \) 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 \( BB_{k(c+1)} \).

![Control flow graph of a program segment](image_url)

**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.

<table>
<thead>
<tr>
<th>expr1</th>
<th>expr2</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>(&lt;BB_1, BB_2, BB_5&gt;)</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>(&lt;BB_1, BB_2, BB_5&gt;)</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>(&lt;BB_1, BB_3, BB_4, BB_5&gt;)</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>(&lt;BB_1, BB_3, BB_5&gt;)</td>
</tr>
</tbody>
</table>

In this example, there are only three distinct paths because when \( expr1 \) is true, the result of \( expr2 \) is irrelevant. Consider method \( mk_f \) again. For each control flow path \( CFP_{ij} \) of \( mk_f \), we define \( 3CFP_{ij} \) to be the corresponding three address code for \( CFP_{ij} \). \( TACCFP_{kj}, \text{ReadsFromCFP}_{kj}, \text{and FinalWriteCFP}_{kj} \) are similarly associated with \( CFP_{ij} \). The following additional data structure is associated with \( mk_f \):

- \( \text{CondBlkMkf[i]} \): is a one dimensional array of records. Each record contains two fields: \( \text{begin} \) and \( \text{end} \). The index of each element corresponds to a statement \( s_i \) in \( 3mk_f \). If \( \text{CondBlkMkf}[s_i].\text{begin} = s_b \) and \( \text{CondBlkMkf}[s_i].\text{end} = s_e \), \( s_i \) 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 \( \text{begin} \) and the \( \text{end} \) of the outermost block. If \( s_b \) and \( s_e \) are zero, \( s_i \) is not in a conditional block.

- \( \text{condqueueMkf} \): is a queue of statements in \( 3mk_f \). Every time a statement \( s_i \) is marked for re-execution, it is pushed into \( \text{condqueueMkf} \) if it is within a conditional block. When \( s_i \) is popped from the \( \text{condqueueMkf} \), the beginning 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.

The \( \text{BitString} \) procedure requires a major modification. In Fig. 13, statements related to each \( x \in RS(mk_f) \) are searched along with all possible control flow paths in \( mk_f \). In addition, if any statement \( s_i \) that is directly or indirectly related to \( x \) is within a conditional block, \( \text{condqueueMkf} \) is not empty.
Reconciling instead of aborting unsuccessfull transaction…

conditional block, all statements in that conditional block are considered to be related to \( x \). The \textit{FindRelatedCode} procedure has been slightly modified to find statements such as \( s_i \) and put them in \textit{condqueueMkf} (Fig. 19 lines 5-8). The \textit{BitString} procedure pops statements such as \( s_i \) from the \textit{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).

\begin{verbatim}
Procedure FindRelatedCode(s_i, ReadsFromCFPkj[], FinalWriteCFPkj[], var bitstringMkf[x])
begin
  for every \( y \in rs(s_i) \) do
    \( s_j \leftarrow \text{ReadsFromCFPkj}[y, s_i] \)  (1)
    if (\( s_j \) is not marked) and (FinalWriteCFPkj[y] \( > s_i \)) then
      FindRelatedCode(s_j, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[x])  (2)
  if \( s_i \) is not marked then
    set bit \( s_i \) in bitstringMkf[x] to 1  (3)
    if CondBlkMkf[s_i].start \( != 0 \) then
      push \( s_i \) into condqueueMkf  (4)
  Let \( k \) be such that \( k \in ws(s_i) \)  (5)
  for every \( s_j \) in which \( s_i = \text{ReadsFromCFPkj}[k, s_j] \) do
    if \( s_j \) is not marked then
      FindRelatedCode(s_j, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[x])  (6)
  \( st \leftarrow \text{FinalWriteCFPkj}[k] \)  (7)
  if \( st \) is not marked then
    FindRelatedCode(st, ReadsFromCFPkj[], FinalWriteCFPkj[], bitstringMkf[x])  (8)
end

Fig. 19. Procedure find related code for complex methods
\end{verbatim}

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


