Effective Optimistic Concurrency Control in Multiversion Object Bases[*]

Peter Graham and Ken Barker
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada R3T 2N2
{pgraham,barker}@cs.umanitoba.ca

June 28, 1994

Abstract

The use of versioned data has proven its value in many areas of Computer Science including concurrency control. In this paper we examine the use of versioned objects in object bases for the purpose of enhancing concurrency. We provide a framework for discussing multi-version objects which includes fundamental definitions, the abstraction of objects as automata and a model of object method executions as transactions. A practical optimistic concurrency control protocol for multiversion objects is then presented within the developed framework. This protocol avoids the high roll back costs associated with optimistic protocols in two ways. First, a less restrictive definition of conflict, compared to other definitions, is used to determine when concurrent executions are invalid. Fewer conflicts means fewer roll backs are necessary. Second, a *reconciliation* process is described which permits cost-effective recovery from invalid concurrent executions rather than roll back.

1 Introduction

Object base systems have been of particular interest to the research community recently with many interesting results being produced. One of the major motivations for developing object base systems is their suitability in supporting the advanced applications expected over the next decades. Anticipated future systems include cooperative work environments [8] and computer assisted design

[*]This research was partially supported by the Natural Science and Engineering Research Council (NSERC) of Canada under Operating Grants (OGP-0105566 and OGP-0121527).

systems [2], both of which support multiple users working toward some common goal, often on overlapping components. We argue that such applications require versioned objects to effectively provide the capabilities expected of them.

There are many reasons to support multi-version objects in an object base:

- Versioned objects permit increased concurrency in the execution of object method invocations.
- Design applications often require that multiple versions of objects be kept for history, software management, and recovery purposes.
- Systems having long-lived transactions may use multi-version objects to permit concurrent access to data by those transactions.

Little work has directly addressed the issues associated with supporting versioned objects in object base systems. The lack of interest is due to the absence of a carefully constructed model describing when versions are created, when versions may be deleted, and when *concurrent* object access is valid. It is also because of the drawbacks attributed to optimistic concurrency control protocols which are characteristic of multiversion concurrency control. This paper addresses both of these issues.

First, a simple model is proposed that describes the elements of an object base system that are of interest. These include *objects*, *transactions*, and *versions*. Then we formally define what constitutes a correct execution sequence in a way analogous to the definition of *serializability*.

Second, we address the fundamental deficiency of existing optimistic concurrency control schemes, namely, *roll back* cost. Optimistic schemes offer much reduced concurrency control overhead when conflicts are rare but require costly roll backs when conflicts occur. We reduce this cost in two ways; we provide a conflict definition which is less restrictive than that normally applied, and we *reconcile* rather than abort. We describe the basics of a reconciliation process which combines the effects of *conflicting* concurrent transactions on an object to produce a single, consistent object version. This is done at lower cost than aborting and re-executing one of the conflicting transactions.

This paper is organized as follows. Related work is described in Section 2. Section 3 presents our model. Section 4 discusses conflicts as they occur between transactions in an object base and defines a non-restrictive, fine-grained, conflict criterion. The reconciliation process is detailed in Section 5 while Section 6 presents a concurrency control algorithm that uses reconciliation in a multiversion object base. Finally, Section 7 makes some concluding remarks and proposes directions for future research.

2 Related Work

The seminal work on multiversion concurrency control in conventional databases is that of Bernstein and Goodman [3] which defines the model where *writes* to data items result in the creation of new versions of those items. This permits late *read*s to overlap with subsequent operations thereby enhancing concurrency. The basic work they describe has been extended and refined by numerous researchers including, Wu, *et al* [22] and Morzy [17].

Kelter [13] has addressed the problem of providing concurrency control for versioned objects in CAD databases and Kaifer has presented a "framework" for cooperative work based on versions [11]. Recently, Kaifer and Schöning suggest a mechanism for mapping existing version models to complex object models [12]. This work provides a concrete application of versioning in object systems.

Some existing object bases do support object versions, most notably ORION [15] and IRIS [21]. Object versions in these systems are, however, maintained explicitly by the user [14]. We exploit object versions exclusively for concurrency control purposes and thus the existence of multiple versions is transparent to the user.

To avoid rollback, multiple versions of an object which have been concurrently accessed must be reconciled to produce a single consistent version. Such reconciliation requires the capture of semantic information about the operations performed on object versions. Much work in the literature incorporating semantics into concurrency control is based on *commutativity* as described by Weihl [19]. A serious drawback to the use of commutativity is that the commutativity relations must be generated *manually*. This places an undue burden on the programmer. Other approaches refine concurrency control by using forms of semantic information about objects which are derived *automatically*. These include the work of Malta and Martinez [16], Graham, *et al* [7], and Hakimzadeh and Perrizo [10].

Reconciliation has been suggested for use in distributed systems to permit concurrent access to *all* file replicas despite network partitioning (e.g. the LOCUS distributed operating system [18]. Rather than limiting access to a single replica while the network is partitioned, reconciliation is used to "combine" the effects of any conflicting updates that occurr while the network is partitioned. Permitting optimistic access to replicas has also been suggested for use in distributed databases by Davidson [4]. Two fundamental differences between Davidson's work and ours are that she requires commitment delays (until recovery is performed) and requires complete transaction re-execution rather than reconciliation.

3 The Model

In our model, an object base consists of a set of uniquely identified objects each containing structural and behavioural components. The structural component is a set of uniquely identified data items referred to as *attributes* whose values define the object's state. The behavioural component is a set of procedures, usually called *methods*, that are the only means of accessing the structural components and thereby modifying the object's state. This paper denotes the j^{th} attribute of object O_i as a_{ij}. Similarly, an object's method(s) are identified using the notation m_{ij}.

Object base users execute transactions on objects by invoking methods that manipulate their attributes. A transaction is a partial order of operations on objects in the object base. We denote an operation (i.e. *method invocation*) of a transaction T^i on object O_j by m^i_{jk} and the set of all operations of transaction T^i by OS^i. All transactions terminate by either committing or aborting. $N^i \in \{\text{commit}, \text{abort}\}$ is the termination condition for T^i.

Definition 1 A *transaction* T^i is a partial order $(\Sigma^i, \prec^i)$ where:

1. $\Sigma^i = OS^i \cup \{N^i\}$.
2. For any two conflicting operations $m^i_{jm}, m^i_{jn} \in OS^i$, then either $m^i_{jm} \prec^i m^i_{jn}$ or $m^i_{jn} \prec^i m^i_{jm}$.
3. $\forall m^i_{jk} \in OS^i, m^i_{jk} \prec^i N^i$. ■

At any time, there exists a single committed version of each object in the object base called the *last committed version* (LCV). When a user accesses an object by issuing a transaction, an *active version* (AV) (which is a copy of the current LCV) is *derived*. Thus, there is an active version for each transaction accessing a given object. The LCV from which an AV is derived is called the *base version* of the object for the corresponding transaction. The version which is the *last* committed version changes as new versions are successfully committed. Committed versions are deleted once all active versions derived from them have either committed or aborted. All active versions which conflict with committed versions must be *reconciled* or aborted so that only "correct" effects are reflected in the object base.

Definition 2 An object version may be *committed* iff

1. it is an object in its initial state, or
2. the object has been moved from one consistent state to another consistent state by a committed transaction. ■

3.1 Execution Model

Having defined the basic elements of a multiversion object base, it is now necessary to understand how these exist and interact. Multiversion object bases have multiple objects, several concurrently executing transactions, and possibly multiple versions of each object. This is viewed as a transaction system whose execution sequences may be captured by a *history*.

Intuitively, a history is a partial order of the executions of transaction operations where the ordering relation must include those pairs of operations that conflict.

Definition 3 A *history* H, of a set of transactions $\mathcal{T} = \{T^1, T^2, \ldots, T^n\}$, is a partial order $(\Sigma_T, \prec_T)$ where:

1. $\Sigma_T = \bigcup_{i=1}^{n} \Sigma^i$.
2. $\prec_T \supseteq \bigcup_{i=1}^{n} \prec^i$.
3. For any two conflicting operations $m_{kr}^i, m_{ks}^j \in \Sigma_T$, either $m_{kr}^i \prec_T m_{ks}^j$, or $m_{kr}^j \prec_T m_{ks}^i$. ■

This definition enables us to reason about histories as transaction operation execution sequences using the ordering relation $\prec_T$. For example, given that the initial state of an object O_k, prior to the execution of the transactions in T is S_k, then the three method invocations occurring in the order; $m_{kr}^i \prec m_{ks}^j \prec m_{kt}^l$ will move O_k into some state $S_{k'}$. This can be considered a sequence of state transition functions such that $S_{k'} = m_{kt}^l(m_{ks}^j(m_{kr}^i(S_k)))$.

Transaction models usually base correctness on a *serial* ordering where a single transaction executed alone on a consistent object base will produce a consistent object base. Multiversion object bases can use the same correctness criterion tailored to suit their unique characteristics. Since each transaction creates its own active object, it can freely modify the object without concern for other users. When a transaction commits it can replace the existing committed version of the object with the "new" active version and leave the object base in a consistent state.

Definition 4 A transaction T^i which accesses an object O_j will create an active object O_j^i, modify it to create $O_j^{i'}$ and then replace O_j with $O_j^{i'}$ when it commits. A *serial* history (H_s) is one which follows this sequence indivisibly for all transactions at all objects. ■

In a serial execution (Definition 4) only one active version of any object will ever exist at a time (due to indivisibility). Thus, there can be no interleaving of operations from different transactions at an object.

Definition 5 A history H is serializable iff H is equivalent to some serial history H_S. ■

Serializable histories and their corresponding transactions are, by definition, correct.

4 Conflicts

Using optimistic concurrency control *any* pair of concurrent method invocations on an object will result in a conflict requiring one method execution (and the corresponding transaction) to be aborted. One way to limit aborts, is through the use of a precise notion of conflict. The basis of our definition of conflict is traditional read/write and write/write conflicts but at a finer level of granularity than the entire object (as would be used, for instance in systems which apply object-level locking such as Orion [6, 15], O2 [5] and IRIS [21]). The attribute read/write behaviour of methods (i.e. the semantic information) needed to determine conflicts can be determined *automatically* when the methods are compiled.

Conflict *may* be occur when two method invocations occur concurrently on a single object O_i. In this case, O_i is initially in some state S_i and two method executions (say $f^s = m^s_{ix}$ and $g^t = m^t_{iy}$) perform state transforming functions on their local copies of the object. Since f^s and g^t are scheduled concurrently, we assume that they are unrelated[1] and their execution order is irrelevant as long as it is serializable. Thus to be correct, the new object state after the concurrent execution of f^s and g^t must be either $f(g(S_i))$ or $g(f(S_i))$. There is, however, no guarantee that an uncontrolled concurrent execution of f and g will produce a final object state equivalent to either. Transaction T^s which executes f will produce a version of O_i in state $f(S_i)$ while transaction T^t (executing g) will produce a version having state $g(S_i)$. These versions may not be "compatible" (i.e. $g(S_i) \neq f(S_i) \neq f(g(S_i)) \neq g(f(S_i))$). To know when concurrent executions will be incompatible we must know when the method executions conflict. We extend the common notion of conflict to object versions as follows:

Definition 6 Two *versions* of an object are said to *conflict* if they are produced by conflicting method executions on active versions of an object derived from the same base version. ■

When two concurrent method invocations are made against some object O_i, two active versions (O_i' and O_i'') are created and these may be conflicting versions. Four possible "Reconciliation Cases" are possible depending on the kind of conflict which occurs (if any), each of which must be resolved to ensure object consistency.

Reconciliation Case 1 – If f^s and g^t are both read-only transactions with respect to the attributes of O_i, then no conflict occurred and no processing is required.

Reconciliation Case 2 – If f^s is read-only at O_i but g^t is not then the new object state must be set to $g^t(S_i)$. Similarly if g^t is read-only but f^s is not, the new object state must be set to $f^s(S_i)$. Additionally transaction T^s must be serialized before T^t in the first case and T^t must be serialized before T^s in the second.

Reconciliation Case 3 – If f^s and g^t conflict and f^s is a costly (as judged, perhaps, by a statically derived estimate of average number of instructions executed) operation then set the new object state to be $f^s(S_i)$ and re-execute g^t. Similarly if g^t is the costly operation but f^s is not then set the new object state to be $g^t(S_i)$ and re-execute f^s.

Reconciliation Case 4 – If f^s and g^t conflict and both f^s and g^t are costly then apply a reconciliation procedure to combine the effects of f^s and

[1]It is assumed that serial dependences in method executions are respected so concurrent method executions on an object arising from the same transaction do not occur.

Condition 1: $WS(f) \bigcap WS(g) = \phi$
Condition 2: $WS(f) \bigcap RS(g) = \phi$
Condition 3: $RS(f) \bigcap WS(g) = \phi$

Figure 1: Intersection Conditions

g^t to produce a new version having the consistent state $s' = f^s(g^t(S_i))$ or $g^t(f^s(S_i))$ whichever is cheaper to compute. The selection of reconciliation function determines the serialization order of transactions T^s and T^t.

To determine conflict in Reconciliation Cases 3 and 4 we define, for each method, m_{ik} in object O_i a read set ($RS(m_{ik})$) and a write set ($WS(m_{ik})$) of the object attributes. The contents of these sets are the *elemental* attributes of the object. Thus, if an object contains an array, each element in the array is considered to be a *separate* attribute which may appear in one, or both, of the read and write sets of each method in the object. Simple intersections between the read and write sets of method pairs is the basis for our determination of conflicts.

Consider two methods f and g on O_i and the three intersection conditions given in Figure 1. If all these intersection conditions hold then the updates made to the attributes of an object by method executions f^s and g^t are distinct and independent so the resulting object versions do not conflict. This means that both versions may be committed and the transactions may be serialized in either order. Unless both transactions are read-only "simple reconciliation" (discussed later) is required to ensure a final, consistent object state.

Intersection condition 1 tests to ensure that the updates are to distinct object attributes. The subsequent conditions test to ensure that no transaction reads a stale attribute value. If only the first condition and *one* of the last two holds then the Read-Write conflict may lead to an incorrect execution depending on the serialization order chosen. If the appropriate serialization order (having the writer follow the reader) is met, then no conflict occurs despite the intersection of the read and write sets. For example, if conditions 1 and 2 hold but 3 does not, then there is a conflict between transactions T^s and T^t at object O_i but provided that the transactions are serialized in the order $T^s \rightarrow T^t$ the execution is still correct. This is because an attribute read by f^s which is written by g^t is *not* stale. Since T^t serializes after T^s, f^s is expected to read the *old* value of the attribute.

5 Reconciliation

When intersection conditions 1,2,3 (or 1,2 or 1,3) hold and the required serialization order is followed (for 1,2 or 1,3), it is possible to combine the effects of the two transactions using "simple reconciliation".

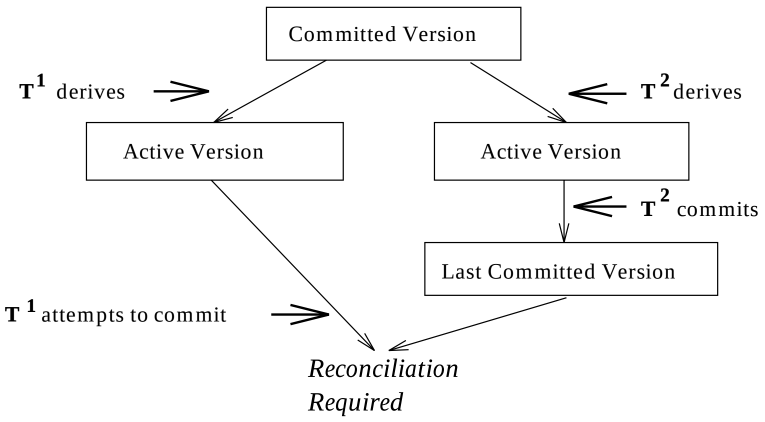

Figure 2: Reconciliation

Definition 7 *Simple reconciliation* is the process of reconciling f^s and g^t by setting the new object state on an attribute by attribute basis as follows: If S_i is the original object state and S_i' is the new, consistent object state, then for each attribute a_{ix}:

$$S_i'(a_{ix}) = \begin{cases} f^s(S_i)(a_{ix}) & \text{if } a_i \in WS(f^s) \\ g^t(S_i)(a_{ix}) & \text{if } a_i \in WS(g^t) \\ S_i(a_{ix}) & \text{otherwise} \end{cases}$$

where the notation $f^s(S_i)$ indicates the state of object O_i produced by executing method f from transaction T^s on the object state S_i. ■

Complex reconciliation is required whenever two costly method executions produce *conflicting* versions based on violation of the intersection conditions of Figure 1. Figure 2 illustrates this situation (assuming T^1's accesses conflict with those of T^2). Although several method invocations may concurrently occur on the same object, in the interest of brevity, we deal only with the two invocation case. We argue that this approach illustrates the basic concept of reconciliation without exposing the unnecessary complexity of the more general problem[2].

We assume that nested objects are atomic. Thus an object attribute which is itself an object, is treated as a *single* entity. Any method invocation (i.e. operation) made on such a "sub-object" is conservatively assumed to conflict with any other such method invocation. This is not necessary but supporting non-atomic sub-objects is beyond the scope of this paper.

Reconciliation is most beneficial when the number of attributes involved in the conflict is small compared to the total number of attributes referenced. In this case only a small percentage of the accessed attributes have been incorrectly updated and rollback would result in a great deal of the method's work being unnecessarily lost. By reconciling rather than rolling back, this work is not

[2]Extension to general forms of sharing is the subject of a future paper.

lost[3] and a savings in execution overhead is achieved. Ideally, the reconciliation process will totally preserve the non-conflicting part of a method execution's work while re-executing only the conflicting part. In no case will reconciliation be more expensive than roll back and re-execution.

The results of the method execution which serializes (and commits) first are correct without modification but those of the other method execution must be changed so that they reflect the results which would have been produced had the initial state of the object used been that produced by the the first method's execution. Which object attributes must be recalculated to achieve this effect can be easily determined and we know that those attributes must be recalculated according to the execution semantics of the *second* method. Thus, if the operations m^s_{ij} and m^t_{ik} are to be serialized in the order $T^s \rightarrow T^t$ and the intersection condition $WS(m^s_{ij}) \cap RS(m^t_{ik}) = \phi$ is violated then any computation performed in m^t_{ik} which relies on an attribute value in $WS(m^s_{ij})$ must be re-executed using the state $m^s_{ij}(S_i)$ as input. In effect, full reconciliation may thus be viewed as a *partial* re-execution of one of the methods involved.

The leading question is: "How to automatically generate the reconciliation procedures for each possible pair of object methods in each possible serialization order?" The reconciliation procedures suggested in this paper do not attempt to exploit *high level* semantics as in Weihl's work on commutativity. It is the need to understand high level program semantics that makes the automatic generation of commutativity tables and compensating transactions difficult. Each reconciliation procedure is a subset of one of the methods being reconciled and the determination of the subset is accomplished by exploiting well-understood data dependence [1] techniques.

Definition 8 Given two conflicting operations m^s_{ij} and m^t_{ik} to be serialized in the order $T^s \rightarrow T^t$, the *conflict set* for reconciliation specifies which attributes of O_i must be re-read from the state $m^s_{ij}(S_i)$ by the partial re-execution of m^t_{ik} and is defined as: $\mathcal{CS}(m^s_{ij}, m^t_{ik}) = WS(m^s_{ij}) \cap RS(m^t_{ik})$ ■

Definition 9 Given two computation steps (i.e. operations acting on one or more attributes) s_i and s_j in a method, a *true dependence* (denoted $s_i \delta s_j$) occurs if s_j reads an attribute which was previously written (as judged by the execution semantics of the method specification language) by s_i. ■

Definition 10 Given two computation steps s_i and s_j in a method, an *anti dependence* (denoted $s_i \overline{\delta} s_j$) occurs if s_j writes an attribute which was previously read (as judged by the execution semantics ...) by s_i. ■

Definition 11 Given two computation steps s_i and s_j in a method, an *output dependence* (denoted $s_i \delta^o s_j$) occurs if s_j writes an attribute which was previously written (as judged by the execution semantics ...) by s_i. ■

The attribute inducing the dependence is commonly specified as a subscript to the dependence symbol (e.g. $s_1 \delta_{a_{ix}} s_2$).

[3]Furthermore, the work of other method executions from the same transaction is not lost.

These definitions of dependence are easily extended to incorporate call statements to methods in sub-objects by considering such statements to read any attributes used as input parameters and to write any used as output parameters. This simple extension is sufficient because objects provide encapsulation. Dependence relations may also be extended to involve more than two computation steps by applying a form of transitive closure. For example, given the dependence relations; $s_1 \delta_{a_{ix}} s_2 \overline{\delta}_{a_{ix}} s_3$ there is also a transitive output dependence $s_1 \delta^{o}_{a_{ix}} s_3$ since s_1 writes a_{ix} and so does s_3 in a subsequent statement. Such transitive dependencies are indicated by suffixing a superscript asterisk to the dependence symbol (e.g. $s_1 \delta^{o*}_{a_{ix}} s_3$).

Existing techniques used in optimizing and parallelizing compilers can be used to partially order all the computation steps in a method according to their true data dependencies.

Definition 12 The dependencies between the computation steps in method m_{ij} ($OPS(m_{ij})$) induce a partial order $(OPS(m_{ij}), \prec_\delta)$ which must be followed in any correct execution of that method or a reconciliation procedure which is a subset of it. ∎

Definition 13 A computation step $s_p \in m^t_{ik}$ reads a *critical input attribute* a_{ix} if it accesses it for reading, if $a_{ix} \in \mathcal{CS}(m^s_{ij}, m^t_{ik})$, and if there is no output dependence $s_q \delta^{o*}_{a_{ix}} s_p$ for any $s_q \in m^t_{ik}$. ∎

Any computation step which directly reads a critical input attribute or which depends on a computation step that directly reads a critical input attribute, *depends* on that critical input attribute. All computation steps that depend on a critical input attribute must be re-executed to reconcile the method execution of which they are a part.

Computation steps may be simple assignments or complex, multi-instruction code blocks containing entire control structures. A simple assignment step accesses fewer attributes so it is less likely to depend on a critical input attribute and thereby require re-execution to effect reconciliation. Because it is impossible to know run-time values during compilation (when the reconciliation procedures are generated) and due to the inherent limitations on the computation of dependence information, it is not always possible to completely reduce each complex computation step to a collection of simple assignment operations. Thus, compile-time generation of reconciliation procedures is "conservative". Current dependence analysis techniques do provide sufficient information to make reconciliation practical in the domain of object-based systems.

Definition 14 The *dependence graph* of method m_{ij} is a graph $DG(m_{ij}) = (V, E)$ where V, the vertices in the graph, are the computation steps in m_{ij} and E, the edges connecting vertices in the graph, are determined by $\prec_\delta$ (the orderings determined by the *true* dependencies). ∎

The dependence graph of a method is a DAG[4] which effectively abstracts the computation performed by the method. Each reconciliation procedure may be similarly abstracted as a sub-graph of the dependence graph of the method.

For each pair of methods m_{ij} and m_{ik} in an object O_i two reconciliation procedures $reconcile(m_{ij}, m_{ik})$ and $reconcile(m_{ik}, m_{ij})$ must be generated, one for each possible serialization order.

Definition 15 A *reconciliation procedure* $reconcile(m_{ij}, m_{ik})$ is abstracted by the sub-graph of $DG(m_{ik})$ consisting of those vertices which are dependent on the critical input attributes in $\mathcal{CS}(m_{ij}, m_{ik})$ and the edges which connect those vertices. ∎

The automatic construction of an actual reconciliation procedure given its corresponding dependence sub-graph substitutes the appropriate computation steps from m_{ik} for each vertex in the sub-graph and emits the resulting code in an order given by a topological sort of the computation steps consistent with the partial order determined by the edges of the sub-graph (i.e. consistent with $\prec_\delta$).

6 Multi-Version Concurrency Control with Reconciliation

We have discussed both conflicts and the reconciliation process as if they were applicable to exactly two *active* transactions attempting to commit *simultaneously*. It is unacceptable to postpone the commitment of one transaction while waiting for another to complete. Thus, in the following algorithm we apply the basic techniques in a way which allows a single transaction to commit at a time.

The algorithm in Figure 3 describes the scheduling of multi-version object transactions with reconciliation. It accepts a transaction (T^i) as input and produces a response indicating success (commit) or failure (abort). When T^i is submitted the set of objects accessed $\mathcal{BS}^i$ is determined (line (2)). The algorithm assumes this is accomplished *á priori*. An active version is made of each object accessed and the operations on each object are mapped to references to that version (line (5)). The transaction then executes operations without restriction or further intervention from the scheduler until it either aborts or is prepared to commit. If T^i issues an abort operation, procedure **version_abort** (line (1)) is called from (line (7)) to discard all the active object versions created and the algorithm terminates and returns a failure indication.

Commit processing begins at line (12) where it is determined if another transaction has committed a new version of the object since the committing transaction began – this is the test for potential conflict. The various reconciliation cases are dealt with by (lines (13 – 21)). Note that read only access

[4]This assumes that loops are treated as single computation steps. This simplifying assumption does not affect the correctness of the computation only the granularity of the operations and hence the likelihood that they will need to be recomputed during reconciliation.

```
procedure Version Transaction Scheduler ($T^i$) : returns (result);
input:   $T^i$ : transaction to be executed;
output: result : boolean;

    procedure version_abort ($T^j, \mathcal{BS}^j$);                                    (1)
    input:   $T^j$ : transaction to abort;
             $\mathcal{BS}^j$ : Base Set of $T^j$;
    begin
        forall $O_k \in \mathcal{BS}^j$ do discard $O_k^{j'}$
    end /* version_abort */;

begin /* Transaction Scheduling */
    $\mathcal{BS}^i \leftarrow$ set of objects read/written by $T^i$;                          (2)
    forall $O_k \in \mathcal{BS}^i$                                                     (3)
        derive an active object $O_k^{i'}$ from the LCV of $O_k$;                        (4)
    execute $T^i$: mapping each operation on $O_l$ to an operation on $O_l^{i'}$          (5)
    if ($T^i$ aborts) then begin                                                      (6)
        version_abort ($T^i, \mathcal{BS}^i$);                                            (7)
        exit (FALSE);                                                                 (8)
    end;
    if ($T^i$ is prepared–to–commit) then                                             (9)
        forall $O_k \in \mathcal{BS}^i$ do begin                                         (10)
            lock ($O_k$);                                                             (11)
            if ($O_k$'s LCV is not the base version of $O_k^{i'}$) then begin          (12)
                /* A new LCV exists so ... */
                if (case 2 and $O_k^{i'} = LCV$) then                                 (13)
                    /* do reconciliation – we read stale data */
                    $complex\_reconcile(LCV(O_k), O_k^{i'})$ ;                         (14)
                else if (case 3) then                                                 (15)
                    /* reconciliation is cheap so do it */
                    $complex\_reconcile(LCV(O_k), O_k^{i'})$ ;                         (16)
                else if (case 4) then                                                 (17)
                    if (conditions in Figure 1 are met for the
                        order $T^{LCV} \rightarrow T^i$ at $O_k$) then                       (18)
                        $simple\_reconcile(LCV(O_k), O_k^{i'})$ ;                      (19)
                    else
                        $complex\_reconcile(LCV(O_k), O_k^{i'})$ ;                     (20)
                else ; /* this version is OK */                                      (21)
            end;
        begin_atomic;                                                                (22)
            forall $O_k \in \mathcal{BS}^i$ do begin                                     (23)
                commit $O_k^{i'}$ as $O_k$ if $O_k^{i'}$ was updated;                     (24)
                unlock ($O_k$);                                                       (25)
            end
        end_atomic;                                                                  (26)
        exit (TRUE) ;                                                                (27)
        end;
end /*Version Transaction Scheduler */
```

Figure 3: Transaction Scheduling with Reconciliation

occurring in Reconciliation Cases 1 and 2, are handled implicitly (lines (21) and (24)) because no new version is written if the active version is unchanged. Line (13) tests if T^i was read only at O_k and then determines if stale data was read (due to a previous committed transaction). If so, complex reconciliation (see Definition 15) is performed (line (14)). Similarly complex reconciliation is performed (line (16)) for Reconciliation Case 3. If T^i is the costly operation we must still reconcile it with the inexpensive one which created the current LCV since we have no ability to change the serialization order so that $T^i \rightarrow T^{LCV}$. If T^i is the inexpensive operation rather than re-executing it we also reconcile. This is reasonable since we know that reconciliation will never be more expensive than re-execution (the worst case is a reconciliation procedure which contains all the statements of the corresponding method). Reconciliation Case 4 is handled by performing either simple or complex reconciliation (lines (18 – 20)). Simple reconciliation (see Definition 7) is performed whenever the conditions in Figure 1 that correspond to the serialization order $T^{LCV} \rightarrow T^i$ are met. Complex reconciliation is performed otherwise.

The correctness of this algorithm is predicated upon the execution being equivalent to some serial execution. If no concurrent execution of methods occurs at any object, then the multi-version execution is equivalent to a non-multi-version serial execution and is therefore correct. This is easily seen because the commitment order is serial at every object. Since commitment involves generating a new LCV, which will be used by subsequent transactions, and because of the transactions' atomicity and durability properties, we are guaranteed that a subsequent method execution will see only correct, committed object state information when it derives its active version.

When two concurrent method executions occur at an object, one will finish and commit first. According to the algorithm presented, this determines the serialization order to be the commit order[5]. The committing transaction produces a new LCV. In any serial execution it is this version of the object which must be used as the base version for method invocations arising from any transaction serializing after the one which produced the new LCV. Since concurrency has occurred, the second method execution may have seen stale data from the previous LCV (its base version). For concurrent execution to be correct, the second method execution must execute as if it had read the new LCV and used it as its base version.

If the concurrent method executions do not conflict given the serialization order determined by their transaction's commitment order then the second method execution may commit freely without reconciliation and since it did not conflict with the first, it can safely generate a new LCV. If both method executions were read-only then no new LCV need be written by either committed transaction. If only the second method execution writes attributes, then it simply commits creating a single new LCV which is correct and consistent. If the first method execution writes attributes then, since we assume no conflict between the method executions, it must have written data that cannot affect the execution of the

[5]This need not be the case and is the subject of future research.

second method execution. This is the simple reconciliation case. We cannot allow the second transaction to commit and write its version back as the new LCV since this would not correctly reflect the updates made to object attributes by the first method execution (i.e. the lost update problem). We correct this by creating a new LCV which consists of the correctly updated attributes produced by *both* method executions.

Finally, if the method executions do conflict then complex reconciliation must be applied. To produce a correct execution, the result of complex reconciliation must be a version of the object in a state equivalent to that which would have resulted from the serial execution of the two method executions in the order of their transactions' serialization. This is exactly the effect provided by executing a reconciliation procedure constructed in the fashion described in the paper. Since any operation within the second method which depends on an attribute written by the first is re-executed as a part of the reconciliation procedure and since the execution order of statements within the reconciliation procedure adheres to the serial semantics (i.e. dependencies) of the original method, correctness is assured.

Correctness can also be illustrated in terms of transaction histories. Non-conflicting concurrent executions have histories which are serializable. Conflicting concurrent executions have non-serializable histories (since no concurrency control was enforced *a priori*). Reconciliation effectively re-orders the conflicting operations in a non-serializable history (re-executing operations as necessary) to make the history serializable.

7 Conclusions and Future Work

We have presented a formalism for describing multi-version object base systems and the transactions on them. We have also presented an algorithm (Figure 3) which offers both low-overhead concurrency control and negligible roll back costs. Further, the main cost associated with our approach (creating reconciliation procedures) is incurred at method compilation time and thus results in no runtime overhead.

Much work remains to be done. The stated limitations of the work presented in this paper can be improved upon. This includes the limitation of reconciling only *two* concurrent method executions on an object and the treating of sub-objects as atomic data. Our assumption that objects are not shared by multiple parents is too restrictive because sharing is a fundamental principle of object bases. Unfortunately, while this problem is simple to manage theoretically, a solution to it is quite difficult to implement efficiently.

As each transaction commits at a given object, the algorithm presented only attempts to commit the corresponding active version of the object *after* all other committed versions. This is restrictive since it means that the serialization order must follow the commit order. This too is unnecessary. Under certain conditions, it is possible to serialize a committing version *behind* other versions which have already committed. This may permit more transactions to actually commit without having to invoke a reconciliation procedure and is also the basis for allowing commitment of versions from different base versions.

The conditions under which "out-of-order" serialization may be done are straightforward, but the change in serialization order affects the serialization properties of the algorithm. No local atomicity [20] property is met so inter-object serializability must be *explicitly* ensured. The separation of intra- and inter-object serializability was suggested by Hadzilacos and Hadzilacos [9] and a model was proposed by Zapp and Barker [24, 23]. Efficiently ensuring inter-object serializability is another difficult problem.

Finally, it is difficult to judge the real costs inherent in maintaining the required multiversion object environment without having a prototype system. Similarly, speculation on the savings achieved by reconciling rather than re-executing is meaningless without a set of real applications and a system to test them with. Clearly, an important step in future research will be the construction of a testbed system to verify quantitatively the ideas proposed in this paper.

References

[1] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[2] N.S. Barghouti and G.E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[3] P.A. Bernstein and N. Goodman. Multiversion Concurrency Control – Theory and Algorithms. *ACM Transactions on Database Systems*, 8(4):465 – 483, 1983.

[4] S.B. Davidson. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Transactions on Database Systems*, 9(3):456 – 481, 1984.

[5] O. Deux *et al*. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91 – 108, 1990.

[6] J.F. Garza and W. Kim. Transaction Management in an Object-Oriented Database System. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 37 – 45. ACM, 1988.

[7] P.C.J. Graham, M.E. Zapp, and K. Barker. Applying Method Data Dependence to Transactions in Object Bases. Technical Report TR 92-7, University of Manitoba, Dept. of Computer Science, 1992.

[8] S. Greenberg, editor. *Computer-supported Cooperative Work and Groupware*. Academic Press, 1991.

[9] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2 – 24, 1991.

[10] H. Hakimzadeh and W. Perrizo. Instance Variable Access Locking for Object-Oriented Databases. *International Journal for Micro and Mini Computer Applications*, 1993. *in print*.

[11] W. Kaifer. A Framework for Version-based Cooperation Control. In *Proceedings of the 2nd International Symposium on Database Systems for Advanced Applications*, 1991.

[12] W. Kaifer and H. Schöning. Mapping a Version Model to a Complex-Object Data Model. Technical report, University Kaiserslautern, 1993.

[13] U. Kelter. Concurrency Control for Design Objects with Versions in CAD Databases. *Information Systems*, 12(2):137 – 143, 1987.

[14] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[15] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109 – 124, 1990.

[16] C. Malta and J. Martinez. Automating Fine Concurrency Control in Object-Oriented Databases. In *Proceedings of the International Conference on Data Engineering*, pages 253 – 260, 1993.

[17] T. Morzy. The Correctness of Concurrency Control for Multiversion Database Systems with Limited Number of Versions. *IEEE*, pages 595 – 604, 1993.

[18] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating Systems. In *Proceedings of the Ninth ACM Symposium on the Operating Systems Principles*, pages 49 – 70, Bretton Woods, New Hampshire, 1983.

[19] W.E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488 – 1505, 1988.

[20] W.E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions of Programming Languages and Systems*, 11(2):249 – 282, 1989.

[21] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implemntation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63 – 75, 1990.

[22] K-L. Wu, P.S. Yu, and M-S. Chen. Dynamic Finite Versioning: An Effective Versioning Approach to Concurrent Transaction and Query Processing. *IEEE*, pages 577 – 586, 1993.

[23] M.E. Zapp and K. Barker. Modular Concurrency Control Algorithms for Object Bases. In *International Symposium on Applied Computing: Research and Applications in Software Engineering, Databases, and Distributed Systems*, pages 28–36, Monterrey, Mexico, October 1993.

[24] M.E. Zapp and K. Barker. The Serializability of Transactions in Object Bases. In *Proceedings of the International Conference on Computers and Information*, pages 428–432, Sudbury, Canada, May 1993.