ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13September 1996

Value-serializability and an Architecture for Managing Transactions in Multiversion Objectbase Systems

Ahmad R. Hadaegh and Ken Barker

Published in collaboration with the
British Computer Society

BCS

Springer

Value-serializability and an Architecture for Managing Transactions in Multiversion Objectbase Systems

Ahmad Reza Hadaegh
Department of Computer Science, University of Manitoba
Winnipeg, Canada

Ken Barker
Department of Computer Science, University of Manitoba
Winnipeg, Canada

Abstract

Multiversioning of objects in an objectbase system provides increased concurrency and enhanced reliability. The last decade has seen proposals for managing transactions in multiversion database systems. A new transaction model, a new correctness criterion, and an architecture that exploit multiple versions in objectbase systems are described in this paper. The architecture contains three main components that ensure correct concurrent serializable executions of transactions that satisfy our correctness criterion and provides a spring board for several open problems that must ultimately be addressed.

1 Introduction

Traditional multiversion database environment 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 environment where contention between read-only and update queries is problematic. This paper presents a model of versioned objects for an objectbase environment and concentrates on the role of versions in increasing concurrency control.

An objectbase consists of a set of objects which contain structure and behavior. The structure is the set of attributes *encapsulated* by the object. 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 in another object.

Multiple users may access a database 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. We distinguish two types of transactions: *user transactions* and *version transactions*. A user transaction is a sequence of method invocations on objects. Method executions are managed as version transactions. A version transaction is the execution of read/write operations on a version of an object and any nested method invocations.

Concurrent execution of a set of transactions must be controlled so that the final result of the execution is equal to the result of a serial execution of the transactions. An objectbase system is provided with a scheduler that orders the operations of the concurrent transactions based on a correctness criterion. Conflict-serializability and view-serializability are two correctness criteria often selected for transaction models. This paper introduces a new correctness criterion called *value*-serializability. Value-serializability seems to be more efficient to implement than conflict-serializability and is not NP-complete like view-serializability.

Correctness criteria are enforced by concurrency control algorithms that ensure serialization 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, this paper introduces the object versioning techniques used to build a framework for developing an 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 originated from the same user transactions.

The primary contribution of this paper are as follows:

1. It describes a model that lays out the fundamental concepts needed to manage transactions in a multiversion objectbase system.
2. It introduces a new correctness criterion which allow more scheduling and can be less costly to implement than some well-known traditional correctness criteria.
3. It presents an architecture and illustrates the steps required to implement a suitable optimistic concurrency control.
4. Finally, some solutions that will motivate some challenging open problems such as transactions reconciliation are discussed.

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. Section 4 introduces *value*-serializability and compares it with other correctness criteria. An architecture for our model and details of its components is presented in Section 5. Finally, Section 6 makes some concluding remarks and describes future work.

2 Related Work

Using multiple versions of data items for transaction synchronization was first proposed by Reed [12] and subsequently by Bayer, *et al.* [2]. Multiversioning permits enhanced concurrency, simplifies recoverability, and supports temporal data management. This section briefly reviews relevant multiversion and objectbase concurrency control literature.

2.1 Objectbase Concurrency Control

Our model is closely related to those of Hadzilacos and Hadzilacos [7], and Zapp and Barker [14]. Zapp and Barker define object serializability and a serialization graph technique to capture intra-object and inter-object transaction synchronization found in Hadzilacos and Hadzilacos' model. Intra-object synchronization serializes operations within an object. Inter-object synchronization ensures consistency of the independent synchronization decisions made at each object. We adapt and extend Zapp and Barker's model to the multiversion objectbase environment.

User transactions and *object transactions* lead immediately to the nested transaction model. Each transaction forms a tree whose root (the *top-level* transaction) is the user transaction and whose descendants are object transactions. To ensure correctness a history called *global object history* is defined that contains both the ordering relation of object transactions executed at an object and the ordering of the user transactions.

Zapp and Barker also present an architecture that describes the transaction facilities and a suitable concurrency control algorithm. The architecture is composed of two major components: an Execution Monitor and an Object Processor. The purpose of the Execution Monitor is to provide a user interface and to schedule the method invocations on behalf of user transactions. Methods are converted to object transactions and are executed by the Object Processor. In processing method executions, the Object Processor retrieves and updates object attributes by accessing the persistent object store. The Execution Monitor and the Object Processor ensure intra-object and inter-object serialization, respectively.

2.2 Multiversion Objectbase Concurrency Control

Nakajima [10] presents an optimistic multiversion concurrency control mechanism. Multiversioning techniques are applied to the concepts of backward and forward commutativity [13]. Basically, two operations executing on an object commute if they can be scheduled in any order. Nakajima argues that forward commutativity uses the latest committed version of the objects to determine a conflict relation while backward commutativity uses the current states

of the objects. 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: committed and uncommitted versions. The most recent committed version of an object o^i is called the last committed version of o^i ($LCV(o^i)$), and the most recent uncommitted version of o^i is called the current version of o^i ($CV(o^i)$). When transaction T_j invokes a method M^{ik} in object o^i a new uncommitted version of o^i ($NV(o^i)$) is created for T_j. If the return result from $NV(o^i)$ backward commutes with $CV(o^i)$ or forward commute with $LCV(o^i)$, $NV(o^i)$ becomes the new current version of o^i and replaces the old current version. Otherwise, $NV(o^i)$ is discarded and T_j invokes method M^i again.

Graham and Barker [4] proposed another optimistic concurrency control for objectbase systems. In this algorithm each transaction T_i obtains copies of the objects it requires and is executed independently. Thus, transactions do not interact with each other until commit time. At commit time, transactions that have read stale data must be reconciled before they commit.

Two types of reconciliation are introduced: *simple reconciliation* and *complex reconciliation*. Simple reconciliation merges the result of the execution of two versions o^{f1} and o^{f2} of object o^f accessed by two transactions T_1 and T_2, respectively and provides a serialization order between T_1 and T_2. Versions o^{f1} and o^{f2} can be merged if T_1 and T_2 do not access common data in a conflicting manner.

Complex reconciliation is attempted if simple reconciliation cannot be performed. Complex reconciliation of two transactions T_1 and T_2 may require the less costly transaction be reexecuted against the state created by another transaction. The cost of the reexecution of a transaction is estimated by static compile time analysis. Complex reconciliation of a transaction is mainly partial reexecution of the operations which have accessed stale data. Reconciling an unsuccessful transaction at commit time is often a less costly procedure than the complete roll-back and reexecution of the transactions.

3 The Computational Model

This section defines objects, transactions, and introduces versions *vis a vis* objects.

3.1 Object Model

Zapp and Barker's object model defines a set of objects that are uniquely identifiable containing structure (attributes) and behavior (methods) [14]. We adapt this definition for our model.

Definition 1 *(Object):* An object is an ordered triple, $o = (f, S, M)$, where:

1. f is a unique object identifier,
2. S is the object's structure, composed of attributes such that $\forall a_i, a_j \in S$, $a_i \neq a_j$, and
3. M is the object's behavior, composed of methods such that $\forall m_i, m_j \in M$, $m_i \neq m_j$. ■

Point (1) assigns unique identifiers to each object. Point (2) specifies the attributes of an object and point (3) specifies the methods of an object. This paper identifies object f by o^f. The method and the structure of o^f are unambiguously referenced by M^f and S^f, respectively.

An object is versionable in that several versions can be derived from one object. Versions of an object must have the same structure and methods as the object. Versions are either active, committed, or aborted. An active version of an object begins as a copy of the object which can then be manipulated independent of all other such versions. The structure of an active version may be modified extensively for some period. Eventually, the modified active version commits and becomes a committed version if its state is consistent with the current state of the object. Otherwise, the state of the active version is modified again and if it still can not be committed, it becomes an aborted version. Committed versions are merged with the object, creating a new state for the object. Aborted versions are disposed.

An active version is identified by a pair $< f, c >$ where the first element is the object identifier in which the versions is derived and the second a unique version identifier. Thus we adapt the notational shorthand where v^{fc} identifies active version c of object o^f. An arbitrary data item x in v^{fc} is unambiguously denoted x^{fc}.

3.2 Transaction Model

A nested transaction is described by a tree where the root is the top*-level transaction*, a sequence of intermediate transactions, and a set of leaf transactions. The top-level transaction and its descendants, constitute a *transaction family*. Transaction families appear atomic to other transaction families[1]. Formal definitions of flat and nested transactions appear in the literature [11, 14].

We now precisely describe how our transaction model maps to the traditional nested model [9]. User submit transactions that invoke a set of object methods. Transactions submitted by a user are atomic so the underlying system must ensure that the nesting of methods resulting from them are also atomic. Users only know about the set of methods they submit to the system. Subsequent method invocations performed on behalf of these initial methods are transparent to the users. Thus, nested transactions submitted by the users may be divided into two groups. The first group includes top-level transactions explicitly created by the users and the second contains transactions occurring as a consequence of the method invocations made by the top-level transactions. The transactions in the first group are *user transactions* and those in the second group *version transactions*.

A user transaction cannot directly modify an object's state in the objectbase. This is accomplished by the methods it invokes. Such methods are converted to version transactions. Version transactions are created by the system and each operate on active versions of specific objects.

3.2.1 User Transactions

We denote an operation p of user transaction i as τ_{ip}, the set of all operations of transaction i by OS_i, and the termination condition as $N_i \in \{\text{commit, abort}\}$. User transaction i is denoted UT_i. Operation τ_{ik} of user transaction UT_i is an invocation of a subtransaction denoted by T^f_{ij}. The subtransaction T^f_{ij} refers to the jth subtransaction of UT_i operating on v^{fi}. The set of operations for UT_i is $OS_i = \cup_k \tau_{ik}$, where the τ_{ik}'s are enumerated by finding the transitive closure of the method invocations made by UT_i.

Definition 2 *(User Transaction):* A user transaction UT_i is a partial order ($\sum_i$,$\prec_i$), where:

1. $\sum_i = OS_i \cup \{N_i\}$,
2. For any two τ_{ip},$\tau_{iq} \in OS_i$, if *depends*(τ_{ip},τ_{iq}) or *depends*(τ_{iq},τ_{ip}) then $\tau_{iq} \prec_i \tau_{ip}$ or $\tau_{ip} \prec_i \tau_{iq}$, respectively,
3. $\forall \tau_{ip} \in OS_i$, where $\tau_{ip} = T^f_{ij}$ then $N_{ij} = N_i$, and
4. $\forall \tau_{ip} \in OS_i$, $\tau_{ip} \prec_i N_i$. ■

Point (1) enumerates the operations in UT_i. Point (2) states that dependent operations of a user transaction must be ordered. It introduces a boolean function called $depends$ which accepts two operations at least one being a method invocation and returns true if there is no dependency in the internal semantics of the operations. The rationale for the $depends$ function is discussed below. Point (3) ensures that the termination conditions of all subtransactions invoked by the user transaction are the same as the termination condition of the user transaction. Point (4) restricts any operation of the user transaction from occurring after the user transaction terminates.

3.2.2 Version Transactions

Additional notation is required. A version transaction from step k of UT_i, executing on v^{fi} is denoted VT^f_{ik}. The version transaction VT^f_{ik} is created when operation τ_{ik} of UT_i invokes a method of o^f.

The direct and indirect descendant transactions of a version transaction VT^f_{ik}, are $VT^a_{ik1}, VT^b_{ik2}, ..., VT^k_{ikn}$. When some VT^e_{ikj} attempts to complete, it enters a pre-commit state where it is ready to commit subject to the commitment of its parent transaction. The operation pc denotes entry into the pre-commit state by a nested subtransaction so the operation set of VT^f_{ik} is $OS_{ik} = \{\cup_p \tau_{ikp}\}$, where $\tau_{ikp} \in \{read, write, pc, VT^e_{ikj}\}$. Two version transactions may execute on a common version of an object if they are from the same user transaction and access the same object.

[1] Assuming, as we do throughout this paper, that the transaction nesting is *closed* [9].

Definition 3 *(Version Transaction):* A version transaction VT_{ik}^{f} is a partial order $VT_{ik}^{f} = (\Omega_{ik}^{f}, \prec_{ik}^{f})$, where:

1. $\Omega_{ik}^{f} = OS_{ik} \cup \{N_{ik}\}$,
2. (a) for any two τ_{ikp},$\tau_{ikq} \in OS_{ik}$, if $\tau_{ikp} = w(x^{fi})$ and $\tau_{ikq} = w(x^{fi})/r(x^{fi})$, for any x^{fi}, $\tau_{ikp} \prec_{ik}^{f} \tau_{ikq}$ or $\tau_{ikq} \prec_{ik}^{f} \tau_{ikp}$,
   (b) for any two τ_{ikp},$\tau_{ikq} \in OS_{ik}$ if $\tau_{ikp} = VT_{ikj}^{e}$ and *depends*(τ_{ikp},τ_{ikq}) or *depends*(τ_{ikq},τ_{ikp}), $\tau_{ikq} \prec_{ik}^{f} \tau_{ikp}$ or $\tau_{ikp} \prec_{ik}^{f} \tau_{ikq}$, respectively,
3. if $\tau_{ikp} = pc$, then τ_{ikp} is unique and $\forall \tau_{ikq} \in OS_{ik}$, $p \neq q$ $\tau_{ikq} \prec_{ik}^{f} \tau_{ikp}$,
4. $\forall \tau_{ikp} \in OS_{ik}$, where $\tau_{ikp} = VT_{ikj}^{e}$ then $N_{ikj} = N_{ik}$, and
5. $\forall \tau_{ikp} \in OS_{ik}$, $\tau_{ikp} \prec_{ik}^{f} N_{ik}$. ■

Only those points different than Definition 2 are discussed. Point (2a) orders the conflicting local operations of the version transaction. Point (2b) orders the conflicting operations of two subtransactions of a version transaction which are invoked on the *same version*. Point (3) indicates that all operations of a version transaction must occur before its pre-commit operation.

The significance of the $depends$ function (point 2 in Definition 2 and point 2b in Definition 3) is that it provides information to allow intra-transaction concurrency. This implies that operations of a transaction which do not depend on each other can be freely executed concurrently.

Fully describing the implementation of the $depends$ function requires a deep examination of compiler construction and a thorough treatment of the runtime systems. Clearly this is beyond this paper's scope but a brief discussion of the fundamental compile-time techniques should be sufficient to demonstrate feasibility. A more complete description is available in Graham [6] and others [1, 5].

To implement the $depends$ function static information captured at compile time can be used to obtain some knowledge related to dependence between the object methods. The following defines the necessary data structures.

- **extent(msg)**: The extent of a method invocation (message step [7]) msg is the set of all object methods invoked directly or indirectly by msg.
- **RS(s)/WS(s)**: the set of all attributes referenced for read/write by statement s in a method. If the statement is a message step, the readset and the writeset are the input parameters and the output parameters of the message step, respectively.
- **RS(M)/WS(M)**: the set of all attributes referenced for read/write by method M.
- **Conflict-Set(msg,msg')**: is a set of pairs $< M_{if}, M_{jf} >$ where M_{if}, and M_{jf} are two methods of object o^{f} such that $M_{if} \in extent(msg)$, $M_{jf} \in extent(msg')$, and M_{if} and M_{jf} may access attributes in o^{f} in conflicting manner.

Unfortunately, the information provided by the above data structures are only captured conservatively. For example, $extent(m)$ determines all possible methods that may be invoked directly or indirectly if the message step msg is executed. During the execution of a method some sections of a method are pruned off by the conditional statements and only a subset of the code is executed. Since the $depends$ function is using this information, the result of the $depends$ function is also calculated conservatively.

Dependency between two operations can be of three forms: *direct dependency*, *indirect dependency*, and *hidden dependency*. Direct dependency occurs if the two operations directly conflict in the local object. Indirect dependency occurs if the operations commonly access conflicting methods in some other object. Hidden dependency happens if two operations conflict indirectly in the local object (typically the result of recursion). The following shows an example of each form and discusses the methods to detect the dependencies.

Direct dependency is the most trivial case. Figure 1A shows an example of direct dependency between two statements s_1 and s_2 in method m_{1f}. Clearly s_1 and s_2 access conflicting operations. This dependency can be detected by comparing the readsets and the writesets of s_1 and s_2.

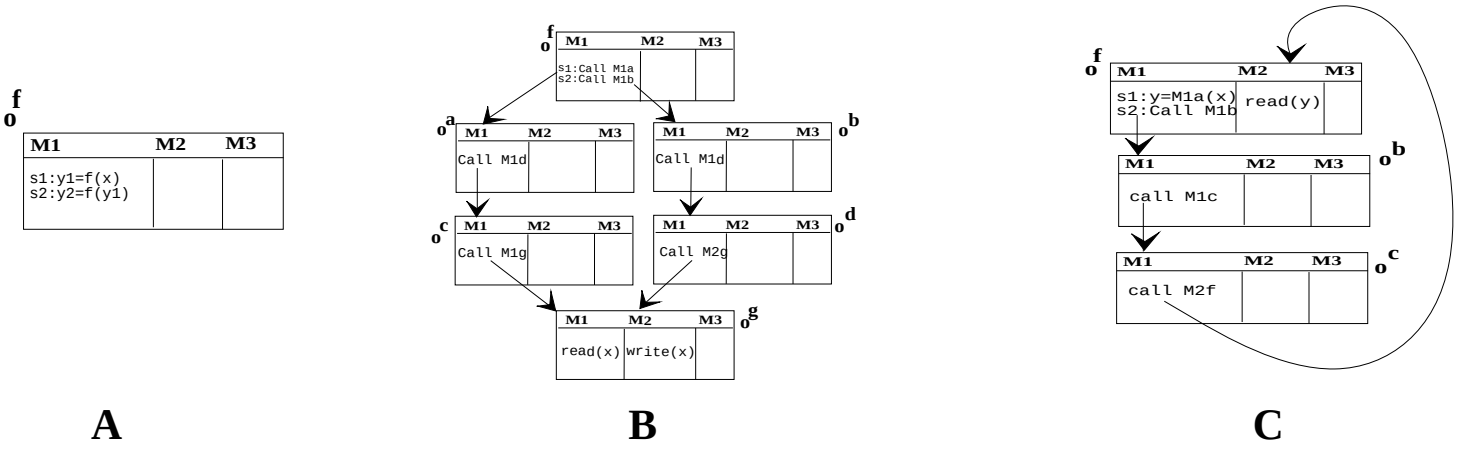

Figure 1: dependency of the statements in a method

Figure 1B shows an example of indirect dependency between two statements s_1 and s_2 in m_{1f}. s_1 and s_2 do not conflict locally but both indirectly invoke some conflicting methods in o^g. This dependency can be detected by comparing the $extent(s_1)$ with the $extent(s_2)$ and building the $Conflict\text{-}Set(s_1, s_2)$. If the conflict-set is empty no indirect dependency occurs; otherwise, potential indirect dependency exist.

Figure 1C illustrates an example of hidden dependency. Note that s_1 and s_2 neither directly nor indirectly conflict. But s_2 indirectly access some other methods in o^f which has some conflicting operation with s_1. This dependency can be detected by comparing the readset and the writeset of s_1 with the readset and the writeset of a method that is indirectly invoked by s_2 in o^f. Other forms of hidden dependencies are also possible. For example, in Figure 1C, s_2 may conflict with some methods that may be called by s_1 in o^f indirectly. Similarly, s_1 and s_2 may call methods m_{2f} and m_{3f}, respectively and conflicts may occur between s_1 and m_{3f}, or s_2 and m_{2f}, or m_{2f} and m_{3f}.

If the result of the $depends$ function is false, the two operations can be freely executed concurrently. Otherwise, if a local dependency exists, one operation is blocked until the other is completely executed. If the two operations are not locally dependent, but the result of the $depends$ function warns about the potential indirect dependency or hidden dependency, the two operations can be executed concurrently as long as their executions are serialized based on a defined correctness criterion.

4 Value-serializability

This section describes a new correctness specification called *value*-serializability that relaxes the restrictive properties of conflict-serializability but is not NP-complete like view-serializability. Before discussing the specification, several notational elements need to be provided and an extension to the traditional definition of a "history" must be stated. First, without loss of generality, a history H is always a committed projection of a schedule created by a scheduler in the system [3]. Further, a read/write operation by user transaction UT_i in history H is represented as $r_i(x, v)/w_i(x, v)$ where v the value read/written by UT_i. Now a history is defined as follows:

Definition 4 *(History):* A complete history over a set of user transactions $\{UT_1, UT_2, ...UT_n\}$ is a partial order with ordering relation $\prec_H$ where:

1. $\sum_H = \sum_i \bigcup \{\bigcup_k^f \Omega_{ik}^f\}$, (1< i <n)
2. $\prec_H \supseteq \prec_i \bigcup \{\bigcup_k^f \prec_{ik}^f\}$, and
3. for every two operations τ_{ip} and $\tau_{jq} \in \sum_H$, and two distinct values u and v, if $\tau_{ip} = w_i(x, u)$ and $\tau_{iq} = r_j(x, v)/w_j(x, v)$, either $\tau_{ip} \prec_H \tau_{jq}$ or $\tau_{jq} \prec_H \tau_{ip}$. ■

Point (1) enumerates the operations of all transaction families (see Definition 3). Point (2) defines the ordering relation for the operations of all transaction families. Point (3) indicates that two conflicting operations belonging to two transaction families must be ordered if they read or write distinct values.

4.1 Serializability

Conflict serializability states that a conflict occurs if two operations access the same data and at least one is a write operation. Our notion of conflict is called $value\text{-}conlict$ and says that conflicting operations occurs when different values are read/written. For example, two write operations that write the same value into a data item x can be executed in any order; or, if x has a value v, a read and a write operation on x can be processed in any order on x as long as the write operation overwrites the same value v on x. As another example consider the following:

$$A = \{w_p(x,5), ..., w_r(x,10), ..., w_q(x,5)\}$$

Suppose A is a projection of history H and operation $r_i(x,5)$ is an operation of UT_i that may or may not be in A. Note that if A does not contain any write operations of UT_i, it makes no difference if $r_i(x,5)$ reads from $w_p(x,5)$ (happens before $w_r(x,5)$) or reads after $w_q(x,5)$ (happens after $w_r(x,5)$). Set A is defined to be a range for operation $r_i(x,v)$. formally defined as follows:

Definition 5 *(Range):* Given three user transactions $UT_i, UT_p, UT_q \in H$, set A is a range for operation $r_i(x,v_i)$ if:

1. A is a projection of H,
2. $w_p(x,v_p)$ of UT_p and $w_q(x,v_q)$ of UT_q are the first and the last elements in A, respectively and $v_i = v_p = v_q$, and
3. A contains no write operation of UT_i on any data item. ■

Thus two write operations accessing the same data item x value-conflict if they write different values into x. A read and a write operation on x also value-conflict if the write operation does not occur in the range of the read operation. This gives rise to the concept of the equivalence between two histories.

Definition 6 *(Value-conflict Equivalent):* Two history H_1 and H_2 are value-conflict equivalent if H_1 and H_2 are defined over the same set of user transactions, and have the same operations, and the order of their value-conflicting operations is the same. ■

A history is serializable if it is equivalent to a $serial$ history [3]. A history is serial if for every two transactions UT_1 and UT_2 in the history, all operations of UT_1 occurs before all operations of UT_2 or vice-versa [3]. Thus a value-serializable history is:

Definition 7 *(Value-serializable):* A history is *value*-serializable if it is value-conflict equivalent to a serial history. ■

4.1.1 The Value-serializability Theorem

Suppose history H is defined over a set of user transactions $T = \{UT_1, UT_2, ...UT_n\}$. We determine whether H is value-serializable by constructing a graph called a *Value Serialization Graph* denoted $VSG(H)$. The $VSG(H) = (V,E)$ where a vertex $v \in V$ represents a transaction $T \in H$, and an edge in E from vertex v_i to vertex v_j indicates that at least one operation of UT_i proceeds and value-conflicts with an operation of UT_j in H.

Theorem 4.1 *(Value-serializability Theorem):* A history H is value-conflict serializable iff $VSG(H)$ is acyclic.
Proof (sketch):
(if): Suppose H is a history over $T = \{UT_1, UT_2, ...UT_n\}$ and $VSG(H)$ is acyclic. Without loss of generality, assume $UT_1, UT_2, ..., UT_n$ are committed in H. Thus $UT_1, UT_2, ..., UT_n$ represent the nodes of $VSG(H)$. Since $VSG(H)$ is acyclic it can be topologically sorted. Let $i_1, i_2, ..., i_n$ be a permutation of $1, 2, ..., n$ such that $UT_{i1}, UT_{i2}, ..., UT_{in}$ is a topological sort of $VSG(H)$. Let H_s be a serial history over $UT_{i1}, UT_{i2}, ..., UT_{in}$. We prove that H is value-conflict equivalent to H_s. Let τ_{ip} and τ_{jq} be operations of UT_i and UT_j, respectively such that τ_{ip} and τ_{jq} value-conflict and τ_{ip} precedes τ_{iq} in H ($\tau_{ip} \rightarrow \tau_{jq}$). By definition of $VSG(H)$, there is an edge from UT_i to UT_j in $VSG(H)$.

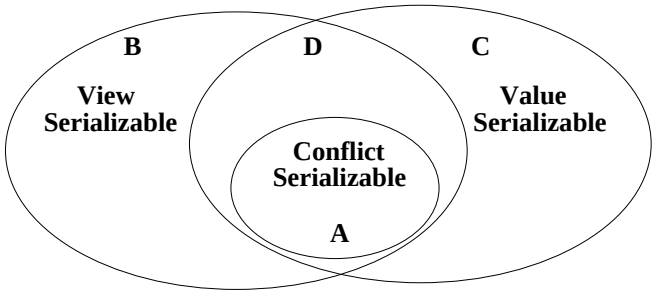

Figure 2: Relationship between value, view, and conflict-serializability

Therefore, in any topological sort of $VSG(H)$, UT_i must appear before UT_j. Consequently, in H_s all operations of UT_i appear before any operation of UT_j. Thus any two value-conflicting operations are ordered in H in the same way as in H_s. Thus H is value-conflict equivalent to H_s.
(only if): Suppose history H is value-conflict serializable. Let H_s be a serial history that is value-conflict equivalent to H. Consider an edge from UT_i to UT_j in $VSG(H)$. Thus there are two value-conflicting operation τ_{ip} and τ_{jq} of UT_i and UT_j, respectively, such that $\tau_{ip} \rightarrow \tau_{jq}$ in H. Because H is value-conflict equivalent to H_s, $\tau_{ip} \rightarrow \tau_{jq}$ in H_s. This indicates that because H_s is serial and τ_{ip} in UT_i proceeds τ_{jq} in UT_j, it follows that UT_i appears before UT_j in H_s. Now suppose there is a cycle in $VSG(H)$ and without loss of generality let that cycle be $UT_1 \rightarrow UT_2 \rightarrow ... \rightarrow UT_k \rightarrow UT_1$. This cycle implies that in H_s, UT_1 appears before UT_2 which appears ... before UT_k which appears before UT_1 and so on. Therefore, each transaction occurs before itself which is an absurdity. So no cycle can exist in $VSG(H)$. Thus, $VSG(H)$ must be an acyclic graph. ■

4.2 Relationship with other Correctness Criteria

The value serialization graph discussed above shows that the decision problem that determines if a history is value-serializable can be solved in polynomial time. This is because a cycle in the value serialization graph can be detected in polynomial time. Thus value-serializability is not an NP-complete problem. This section further compares value-serializability with view and conflict serializabilities in terms of scheduling and the cost of implementation.

Figure 2 depicts the relationships among these criteria and we argue that each subset is non-empty. Consider the following histories:

$H_1 = \{r_2(x,5), r_1(x,5), w_2(x,6), c_1, c_2\}$
$H_2 = \{w_1(x,5), w_2(x,6), w_2(y,7), w_1(y,8), c_2, w_3(x,9), w_3(y,10), c_3, w_1(z,11), c_1\}$
$H_3 = \{w_1(z,1), w_1(x,5), c_1, r_2(y,1), r_3(x,5), w_2(x,5), c_2, w_3(y,1), c_3\}$
$H_4 = \{r_1(y,5), r_3(w,1), r_2(y,5), w_1(y,5), w_1(x,1), w_2(x,1), w_2(z,1), w_3(x,1), c_1, c_2, c_3\}$

Histories H_1, H_2, H_3, and H_4 are elements of sets A, B, C, and D, respectively. Clearly H_1 is conflict-serializable. H_2 is view-serializable but it is not conflict-serializable because $w_1(x,5)$ proceeds and conflicts with $w_2(x,6)$ and $w_2(y,7)$ proceeds and conflicts with $w_1(y,8)$. H_2 is also not value-serializable because any two conflicting operations in H_2 do value-conflict too. H_3 is value-serializable because $r_3(x,5)$ and $w_2(x,5)$ do not value-conflict so $VSG(H3)$ does not contain a cycle. However, H_3 is neither view equivalent to T_1, T_2, T_3 nor T_1, T_3, T_2[2]. History H_4 is view-serializable to T_2, T_1, T_3. It is also value-serializable to T_1, T_2, T_3 because $r_2(y,5)$ and $w_1(y,5)$ do not value-conflict and $VSG(H_4)$ is acyclic.

In some environment, concurrency control algorithms that enforce value-serializability can be less costly and more efficient to implement than the ones which use conflict serializability. A common concurrency control algorithm that uses conflict serializability is two phase locking (2PL). Suppose *two phase value locking* (2PVL) is the corresponding concurrency control that enforces value serializability. The following compares $2PL$ verses $2PVL$.

Consider the execution sequence of $2PL$. If a lock is required, a request of $2PL$ is made to the system kernel in privileged mode that requires the suspension of the currently running process, a lock acquisition, and a control switch back to the first process. This is an extremely expensive process that involves approximately one hundred (100) machine cycles (if conflict does not occur) or more (if conflict occurs) [8].

[2]We do not need to check other combinations because T_1 terminates before T_2 and T_3.

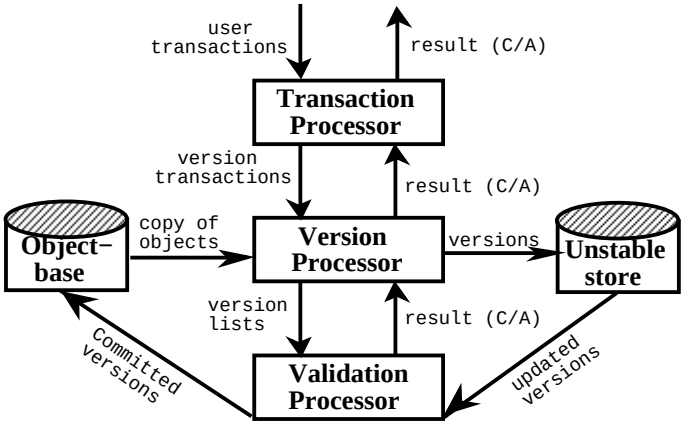

Figure 3: The components of the architecture

If the compilers can detect through static analysis, that a "value" is not in conflict, then the process above can be usurped for this particular access. The cost of $2PVL$ would be a comparison operation between the current value and one read, at the time the transaction initially began execution. This requires only three (3) machine cycles. If you include the cost of the initial reads and the storage of these initial values, it only costs a total of ten (10) cycles. This results in a magnitude savings at execution time.

Unfortunately, two conditions make the scenario problematic. First, if the transactions do actually value-conflict, the locking mechanism ($2PL$) must be added to the checking cost which leads to a ten percent increase in overhead. Secondly, the compiler must embed the comparison operations into the methods which requires a substantial rewrite of the compiler itself and will slow down the compilation process. The former concern is an issue of ongoing research while the latter is irrelevant since it is a pre-runtime issue.

Therefore, environment with low data contention or where the domain of values for the data items is small will benefit the most from $2PVL$. On the other hand, if transactions are constantly updating a small member of data items with a wider range of values (typically by hot-spots) $2PL$ will outperform $2PVL$.

5 Architecture

A versioned object store is comprised of two portions: a persistent *stable* objectbase and a non-persistent *unstable* working store. The objectbase contains persistent objects and the unstable working store keeps the active versions. An active version v^{fi} is created by copying an object o^f from the objectbase and assigning a unique version identifier i. Active versions may be promoted to committed versions where their contents are merged with the objects in the objectbase thereby creating new states for the objects. For our purposes it is sufficient to assume that committed versions are not maintained historically in the objectbase. It should be noted that relaxing this constraint may significantly increase concurrency and has an important impact on the specification of serialization in a multiversion system, but this is beyond the scope of this paper.

5.1 Architectural Model

Three major components form the basis of our architecture: the *Transaction Processor*, the *Version Processor*, and the *Validation Processor* (Figure 3). The Transaction Processor accepts user transactions and returns results to the user. It processes transactions for syntactic correctness, converts the method invocations to version transactions, and schedules version transactions (using the $depends$ function) for user transactions. The Version Processor receives the version transactions from the Transaction Processor and creates new versions of the objects required by the version transactions by copying objects from the objectbase. The active versions associated with the version transactions of a given user transaction are logically grouped into a *version list* after their completion and are submitted to the Validation Processor. The Validation Processor examines the version list and decides whether to abort or commit the user transaction.

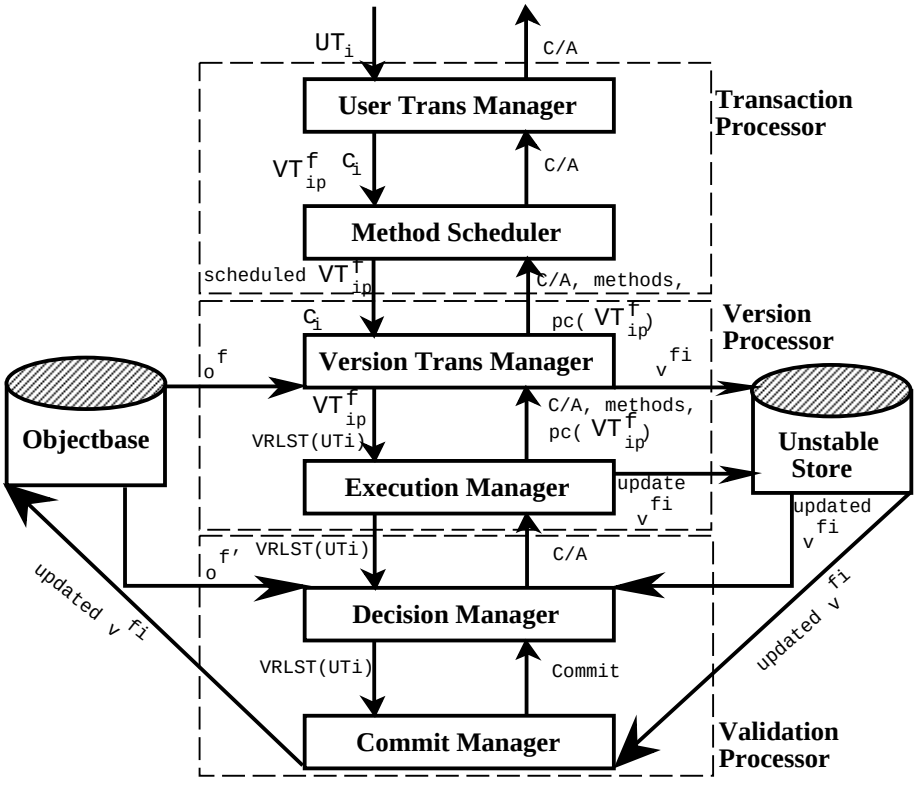

Figure 4: The architecture

Figure 4 shows the components of the architecture in detail. The Transaction Processor contains two components: the User Transaction Manager and the Method Scheduler. The User Transaction Manager coordinates the execution of user transactions by converting the method invocations to version transactions denoted by VT_{ip}^{f}'s and passes them to the Method Scheduler. The notation VT_{ip}^{f} refers to a version transaction of UT_i executing on o^f. The Method Scheduler permits concurrent execution of a user transaction's version transactions (enforcing intra-UT concurrency control) so that version transactions of a single user transaction invoked on the same object are ordered before they are sent to the Version Processor. Version transactions of multiple user transactions are executed concurrently.

The Version Processor contains two components: the Version Transaction Manager and the Execution Manager. The Version Transaction Manager receives the scheduled version transactions (VT_{ip}^{f}'s) from the Method Scheduler. An active version of o^f (v^{fi}) is requested from the objectbase and placed in the unstable store. Next the Version Transaction Manager passes VT_{ip}^{f} to the Execution Manager. The Execution Manager executes the operations of VT_{ip}^{f} updating v^{fi} in the unstable store.

The Version Transaction Manager also builds a version list for each active user transaction. The version list of UT_i ($VRLST(UT_i)$) is a set that logically records the objects referenced by UT_i. Every time a version of an object (v^{fi}) is created for UT_i, the Version Transaction Manager appends f (the object identifier) to $VRLST(UT_i)$. When all the version transactions of UT_i terminate, $VRLST(UT_i)$ is passed to the Execution Manager. The Execution Manager submits $VRLST(UT_i)$ to the Validation Processor.

The Validation Processor checks the validity of the updated versions referred to in the version list. It has two components: the Decision Manager and the Commit Manager. The Decision Manager compares each updated version (v^{fi}) referred in the version list with its related object ($o^{f'}$) in the objectbase. The purpose of the comparison is to determine if updated active versions would create inconsistency in the objectbase. An updated version v^{fi} is consistent with o^f if the attributes accessed in v^{fi} have not been accessed in o^f since v^{fi} was created. If the results made from all the updating versions are consistent with the current state of their corresponding objects in the objectbase, the version

list is passed to the Commit Manager. The Commit Manager promotes the updated versions to committed versions and merges the committed versions with their corresponding objects in the objectbase, thereby creating new states for the objects.

6 Conclusion

We have presented a formalism for multiversion objects and transactions on them. We have presented a serializability theory and an architecture which can be used as the basis for the development of optimistic concurrency control protocols. Detailed algorithms will appear later in subsequent work. Several open problems present themselves. First, enhancements to the basic optimistic algorithm reflected in our model are yet to be developed. Such an algorithm can exploit the maintenance of historical objects by providing increased opportunities to serialize committing transactions. The availability of historical data may help enhance reliability in addition to the obvious benefits of tracking data values over time. Reconciliation is still a largely unexplored research area. Successful research that detects "incorrect" data items but makes them consistent with the rest of the information in the objectbase would have significant impact on both multiversion and semantic database systems.

7 Acknowledgment

This research was partially supported by the Natural Science and Engineering Research Council (NSERC) of Canada under operating grant (OGP-0105566).

References

[1] A. Aho, R. Sethi and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. *ACM Transactions on Database Systems* 5(2):139–156,1980.

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

[4] P. Graham and K. Barker. Version Management in Multiversion Object Bases. *Proceedings of ICCA Conference*, 1994.

[5] P. Graham, M. Zapp, and K. Barker. *Concurrency Control in Object-Based Systems*. University of Manitoba. Technical Report: 92–07, June 1992.

[6] P.J. Graham. *Applications of Static Analysis to Concurrency Control and Recovery in Objectbase Systems.* Ph.D. thesis, University of Manitoba, 1994.

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

[8] C. Mohan". Commit-LSN: A Novel and Simple Method for Reducing Locking and Laching in Transaction Processing Systems. *In Proceedings of 16th VLDB Conference*, pages 1-14, 1990.

[9] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[10] T. Nakajima. Commutativity Based Concurrency Control for Multiversion Objects. *In Proceedings of the International Workshop on Distributed Object Management*, pages 101–119, 1992.

[11] M.T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[12] D. Reed. *Naming and Synchronization in a Decentralized Computer System*. MIT Laboratory for Computer Science, MIT/LCS/TR-205, 1978.

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

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