Multiversion Concurrency Control in Objectbased Systems

by

Ahmad Reza Hadaegh

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Doctor of philosophy
in
Computer Science

Winnipeg, Manitoba, Canada, 1997

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

MULTIVERSION CONCURRENCY CONTROL IN OBJECTBASED SYSTEMS

BY

AHMAD REZA HADAEGH

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Current approaches to enhancing concurrency in database systems have focused on developing new transaction models that typically demand changes to either the atomicity, consistency, or isolation properties of the transactions themselves. Indeed much of this work has been insightful but most of these attempts suffer from being either computationally infeasible or requiring the transaction designer be sufficiently knowledgeable to determine, *á priori*, how the application semantics must interface with the transaction model (see [Wei88] or [Wei91] for examples). Our approach exploits an object-oriented world and is different than others because the transaction designer is not expected to have intimate application knowledge; this load is placed on the transaction system which must determine, by static analysis, how to maintain database consistency.

We adopt an extremely aggressive approach whereby each transaction is given its own copy (version) of all of the objects it needs to execute so it can proceed without interruption from other processes running on the same system. This dissertation describes an overall 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, but 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. Within the context of concurrency the key aspects addressed by this thesis are: 1) A new correctness criterion is described that emits more histories than conflict serializability and is computationally tractable; 2) An architectural model is developed to support multiversioning that provides the well-known ACID transaction properties; 3) An optimistic concurrency control algorithm that functions on this architecture is described and demonstrated to be correct with respect to the new correctness criterion; 4) 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 completed before this one attempted to commit; and 5) Based on static analysis information, algorithms are developed to modify the compiler to generate reconciliation procedures automatically from the initial transaction specification.

Acknowledgements

Finally the painful days of studying for my PhD are over. I feel great and I am ready to concentrate on my next destination. Before I close this chapter of my life I owe acknowledgements to some people.

First, I would like to thank my advisor Dr. Ken Barker, whose determination, motivation, ingenuity, and management helped me getting through my PhD. I will never forget his encouragement and support.

I would also like to thank the members of my PhD committee, Dr. Randal Peters, Dr. Mark Evans, Dr. Robert McLeod, and my external examiner Dr. Vadim Doubrovski for their insight and their comments. I am especially grateful to Dr. Peters for the time he spent in discussion with me and giving interesting comments on my research.

Thanks are also due to the members of the Advanced Database Research Laboratory, especially Dr. Peter Graham, who read drafts of articles related to my thesis and motivated my thoughts at early stage.

I would like to express my sincere gratitude to my wife, my parents and my wife's parents for their encouragement and love that gave me strength to do this dissertation.

Finally I would like to dedicate this masterpiece to my wife Sahar and my son Miad who have waited patiently in this cold land for the day when the snow melts and my vehicle can progress down the road of Phd. I also would like to dedicate this thesis to my parents Reza Hadaegh and Parvin Farahnak through whose prayers and tears Almighty Allah (my beneficent and merciful God) turned our dream into a reality.

Contents

List of Figures

Chapter 1

Introduction

Multiversioning for the purpose of enhancing concurrency and reliability is not a completely unexplored topic (see [BGH87, Nak92, Mor93, WYC93]), but the approach has been less than successful primarily because of the application domains where it has been proposed. For example, using a complicated multiversioning scheme to manage a "short-lived" business transaction like an account balance update is not justifiable because of the amount of overhead and relatively simple data structures involved. These application domains demand that very little time be spent in runtime overhead to support multiversioning and there is little to be gained from spending time before hand preparing for a partial rollback when complete re-execution could occur much more rapidly. However, current transactions and the systems they execute on are becoming increasingly complex and their execution times have increased substantially. "Long-lived" transactions such as those found in design systems, multidatabase systems, or cooperative information systems will benefit greatly from the ability to rollback to a consistent state and then proceed forward without the need to completely re-execute. This dissertation assumes that the environment of interest is of the more complicated variety. We now digress briefly to provide a framework for the rest of the dissertation. The balance of the introduction reviews several issues related to the myriad themes raised in this dissertation.

Traditional multiversion databases environments have used data versioning for historical purposes as well as for issues related to transaction management. Data versioning reduces the overhead involved in recovery and impacts concurrency especially in environments where contention between read-only and update transactions is problematic. This dissertation addresses the problem of concurrency control in a multiversion objectbase environment. An objectbase provides a persistent repository for data stored as objects. Objects contain structure and behavior. The structure is the set of attributes encapsulated by the objects. 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. The fundamental difference between traditional data models and those supported by object-oriented systems is the encapsulation property of the objects.

Users' requests submitted to a system are called *user transactions*. A user transaction includes a set of messages sent to the objects to execute the routines in the objects. Objectbase systems are provided with transaction management facilities to control the effects of the user transactions on the objectbase.

One aspect of transaction management is *concurrency control*. A concurrency control algorithm allows transactions to interleave and shares the resources among the transactions; thereby, utilizing the resources and enhancing the system performance. A concurrency control algorithm must guarantee that each transaction appears to execute atomically and in isolation from other concurrent transactions. It must also ensure that each transaction leaves the database in a consistent valid state.

Traditional transaction models consist of 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 as well as any nested method invocations. Since the nested invocations

are themselves managed as transactions, our model exploits *nested transactions* [Mos85].

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" [BGH87, GR94]). 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. View serializability allows more schedules than conflict serializability but has been shown to be an NP-complete problem [Pap86]. Some correctness criteria developed for advanced database systems relax the restrictive properties of conflict serializability. Examples include *quasi serializability* [DE89], serializability for polytransactions [SRK91], and *reverse serializability* [KM94]. This dissertation introduces a new correctness criterion called *value*-serializability. Value serializability is more efficient to implement than some traditional correctness criteria and permits a greater range of schedules than those used by computationally efficient algorithms like two phase locking.

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. Pessimistic protocols tend to perform better when transactions usually access some common data; whereas, optimistic protocols are more desirable when contention between the transactions is relatively low. Focusing on the centralized objectbase systems, this dissertation introduces the object versioning technique and develops related optimistic concurrency control algorithm. The concurrency control is performed based on static information captured at pre-run time. Our model introduces two types of concurrency control: *inter-UT* and *intra-UT* concurrency. Inter-UT concurrency is done optimistically and refers to the concurrent execution of multiple user transactions. Intra-UT concurrency is performed pessimistically and refers to concurrent execution of multiple subtransactions originated from the same user transaction.

Another aspect of transaction management is *recovery*. To make an objectbase reliable, transaction management should ensure that a recovery mechanism is provided. An objectbase may fail in several ways, including: transaction failure, system failure, media failure, etc. Since failures may leave the objectbase in an invalid state, recovery mechanism are needed to ensure that no intermediate results of failed transactions remain in the objectbase and that all "lost" effects of successful transactions are propagated to the persistent objectbase. Although recovery is closely related to concurrency control, this dissertation does not address it in detail.

Data versioning in database systems impacts concurrency and recovery. In a multiversion system, transactions create new data versions as they access data. This reduces contention between transactions accessing the "same" data and thereby increases concurrency. Further, since transactions only affect their own versions, recovery from a failure should be easier than in a single-version environment.

Multiversioning has been also applied to the advanced database environment [CK86, Nak92, GB94b, HB96]. Objects may have several versions. A version is usually created from the object or another existing version of the object, goes through a series of changes, and eventually may become a fixed stable version. If the number of versions exceeds a certain limit, some versions are purged or archived in secondary storage.

1.1 Motivation

In classical databases, data are centralized and have simple structure. Since transactions are short-lived, locking techniques usually do not defer the execution of related transactions for a long period. Further aborting and restarting unsuccessful transactions may not be problematic in optimistic concurrency control protocols because of the relatively small cost of the re-execution. Therefore, conventional concurrency control is suitable and successful.

In advanced database environments, users interact with complex and possibly distributed data. Transactions are relatively long and may be executed in parallel [NRZ92] so

mechanisms are required to provide inter-transaction synchronization. Locking a page or an object may unnecessarily delay the execution of some operations which in turn severely degrades the system performance. Optimistic protocols are even less desirable for such environment because by aborting a transaction, not only the resource and the time allocated for the transaction is wasted, successful termination of the transaction will not be guaranteed if the transaction is restarted. Thus a transaction may be unduly delayed before it can be executed successfully.

This dissertation develops an optimistic concurrency control to overcome the above limitation. When a transactions cannot be serialized at commit time, it is reconciled instead of being aborted. Reconciliation can be *simple* which may involve changing the commit order of the transactions or *complex* in which partial re-execution of the transactions is required to make the result of the transactions consistent with respect to the current state of the objects in the objectbase.

Object versioning techniques [Nak92, GB94b], nested transaction models [Mos85], and static analysis information [Gra94] have significant roles in the reconciliation algorithm. Since nested transactions are modularized, reconciliation is performed only against the subtransactions which have referenced stale data. Therefore, nesting reduces the problem space. Multiversioning provides a mechanism to maintain the previous state of the objects in the objectbase. If a transaction cannot be committed relative to current state of the objects, it might be committable relative to an older state. Pre-run time static analysis information used to increase concurrency may also be used to reconcile the unsuccessful transactions at commit time. Reconciliation may involve partial re-execution if the data it used was incorrect.

This dissertation makes the following contributions:

1. Provides a taxonomy for reasoning about transactions in a multiversion objectbase.
2. Defines a computational model for multiversion objectbases and identifies the difficulties encountered in providing concurrency control.
3. Proposes a two level abstraction for the system: a *user level* and a *system level.*

- An optimistic concurrency algorithm is provided to serialize user requests at the user level.
- A pessimistic protocol serializes execution of system transactions at the system level.

4. Defines a correctness criterion for serializability.
5. Proposes a concurrency control algorithm and describes its implementation.
6. Proposes reconciliation algorithms to ensure that transactions commit if no failure occurs.

1.2 Taxonomy

The taxonomy shown in Figure 1.1 characterizes possible database environments in terms of (1) historical, (2) complexity, and (3) multiversioning support. The data complexity dimension refers to the structural complexity of the data in a system. The historical dimension refers to systems where some of the previous data values are preserved as the state of data changes overtime, and the multiversioning dimension indicates whether the data and/or the schema are versioned in a system.

The complexity dimension divides systems into three categories: simple data, abstract data types, and object-oriented. The point at the origin refers to conventional systems such as relational schemes where data is collected in sets of tables each containing a collection of formatted homogeneous data. The data is composed of primitive types such as integers or strings and are accessed "directly" by the users.

Abstract data types (ADT) are supported by some programming languages. An abstract data type contains a set of data and a set of operations on the data. It refers to a way of packaging structure and their operations into a useful collection. Operations are the only means of accessing and manipulating the data. This is also called *encapsulation*. ADT's have simple interfaces. They are implemented in arbitrary ways representation such as arrays and pointers, but their representation is transparent to the users so they can be used by understanding their abstract interface's properties [Sta94].

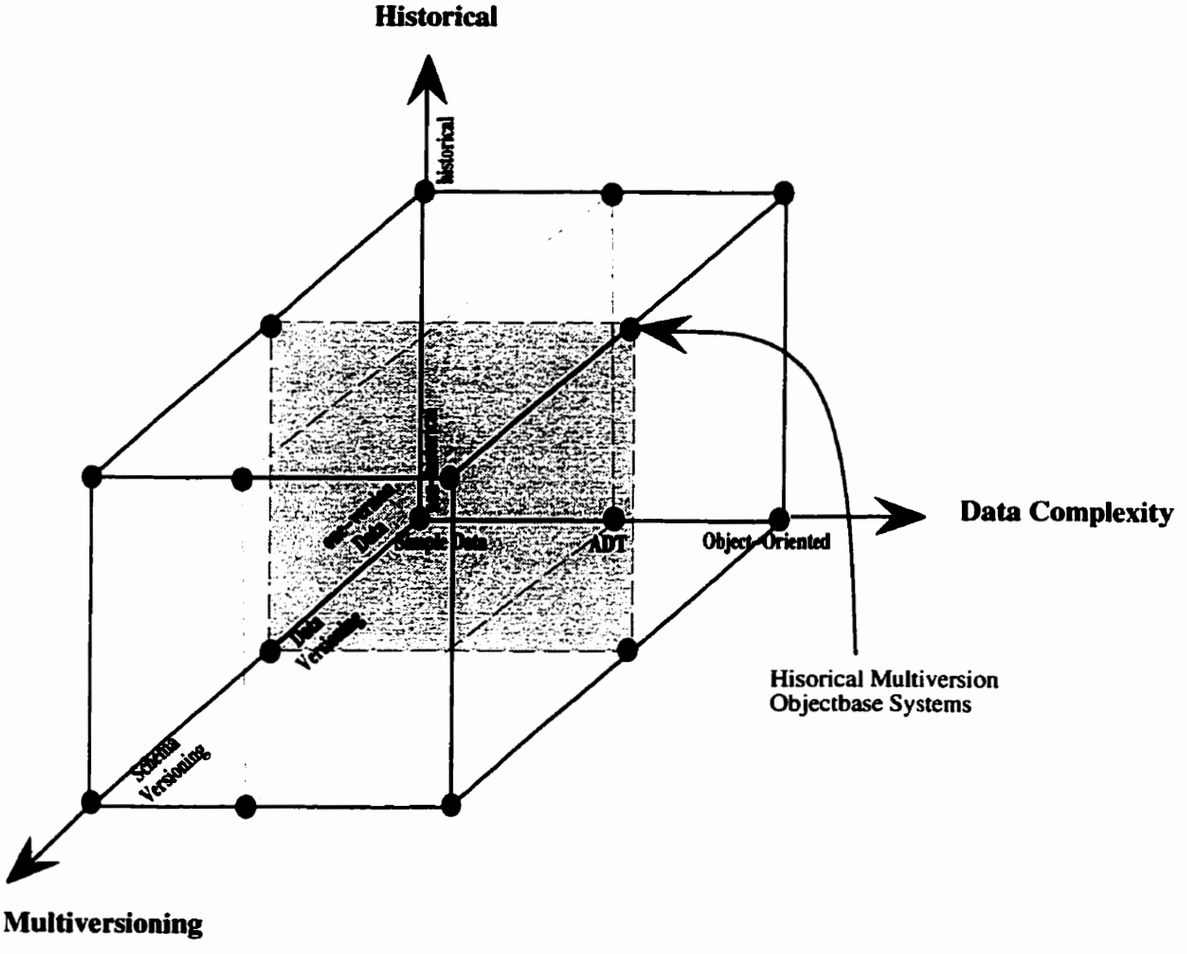


Figure 1.1: Historical Multiversion Objectbase Design Taxonomy

The next level is objectbase systems. In objectbase systems, entities are modeled as objects. Like an abstract data type, an object encapsulates a set of data and their operations. The data defines the structure of an object and operations controlling the object's behavior. It is generally assumed that the behavioral aspect moves objects from one consistent state to another. In addition, objectbase systems provide *inheritance* and *aggregation*. Inheritance allows some objects to obtain some or all of the characteristics of other objects. Aggregated objects, sometimes called *complex objects*, have hierarical structure. Complex objects include other objects which in turn may include other objects, so they can be viewed as being nested. Complex objects may either physically contain other objects in their structure or they may reference other objects by storing their object *ids*.

The second dimension is version management. This dimension includes three types of systems: single-version data, multi-version data, and schema versioning. In a single-version data environment, versioning does not occur. Therefore, every time a data item is modified, the new information replaces the old. However, in a multi-version data environment versions of data items can be preserved. New versions are created whenever data items are modified. When data items are updated, the database may enter a temporarily inconsistent state that is transparent to the users. Versions of data items may eventually become immutable and they can only be changed by deriving new versions [BGH87, MPL92, Nak92]. The last point in this dimension refers to systems which support schema versioning. Kim and Chau [KC88] discuss the schema versioning in an object-oriented prototype called Orion [KGBW90]. They explain the implication of schema versioning in a multi-user environment. Support for schema versioning is not the subject of this dissertation.

The historical dimension is divided into two parts: non-historical and historical systems. Non-historical systems do not keep the previous values of data as data changes overtime. The only records of past data that is reachable is through backup copies of the data and transaction logs. Historical systems keep several snapshots of data. Snapshots of the data are immutable. They indicate the evolution of data as data is modified. Snapshots are usually produced after some modification to data [RH90]. Storage management becomes a

problem as the number of snapshots grows in the system. Therefore, some of the snapshots are archived in a separate secondary storage and only brought to the system on demand.

In this dissertation, we address historical multiversion objectbase systems in which versioning is limited to objects. Investigation of other points on the taxonomy is left as future research.

1.3 Outline

The outline of this dissertation is as follows. The related material in Chapter 2 discusses concurrency control in nested transactions, object-oriented data models, and multiversion data models that will motivate this dissertation. Chapter 3 defines a computational model and its key concepts. It also lays out the relevant concepts of data dependency and compile time static analysis used for enhancement of transaction management. Chapter 4 illustrates an architecture for our model, describes its components, and details its implementation. Simple reconciliation is described in Chapter 5. The conditions under which transactions may be reconciled are presented too. Chapter 6 describes complex reconciliation and provides the steps required to generate the reconciliation procedures. Finally, Chapter 7 makes some concluding remarks and suggests directions for future work.

Chapter 2

Related Work

In this chapter, the fundamental terminology used in conventional databases is reviewed. We investigate previous work directly related to this research. The structure of this chapter is as follows: Section 2.1 provides a summary of classical transactions and transaction management including concepts of concurrency control and recovery. Section 2.2 introduces nested transactions while Section 2.3 presents the characteristics of object oriented models. Transaction management and concurrency control in object based systems are also studied in Section 2.3. Section 2.4 describes techniques applicable to multiversion database systems. Finally, Section 2.5 makes some summary comments.

2.1 Classical Data Models

Before providing a detailed formalism, an intuitive overview is provided to guide the subsequent more detailed discussion. Users interact with the database by sending their requests in the form of a *transaction*. Transactions are sets of read and write operations. The operations are executed against the data in the database. Executing transactions in progress are called *active transactions*. An active transaction either aborts or commits. If a transaction commits, its execution has been successful and the result of the execution persist in the

database. A transaction aborts as the result of software or hardware problems leaving the database unchanged by the transaction's execution.

A traditional transaction is considered correct if it supports the properties of atomicity, consistency, isolation, and durability. These are known as *A.C.I.D* properties [Gra81]. Atomicity ensures that a transaction either completes successfully or it has no effect on the database. Consistency requires that a successfully committed transactions must move the database from one consistent state to another. Isolation guarantees that a transaction does not read the intermediate results of other transactions. Finally, durability guarantees updates of a committed transaction remain in the database in the face of system or transaction failure.

The remainder of this section defines *flat transactions*, *history*, and *serializability* in classical databases. In addition, we review the concurrency control protocols widely used in these environments.

2.1.1 Flat Transaction

Flat transactions are the "application programs" used to access the data in traditional database systems. Before defining flat transaction, we need some nomenclature. We denote operation p of transaction i as τ_{ip}, a set of all operations of transaction i as OS_i, and the termination operation as $N_i \in$ {commit, abort}. Operation $\tau_{ip} \in \{r(x), w(x)\}$, where $r(x)$ and $w(x)$ denote read and write operations, respectively; and x refers to an arbitrary data item in the database.

Definition 1 *(Flat Transaction):* A flat transaction T_i is a partial order $T_i=(\sum_i, \prec_i)$ where[1]:

1. $\sum_i = OS_i \bigcup \{N_i\}$,
2. for every $\tau_{ip}, \tau_{iq} \in OS_i$, operating on some data item x in the database, if either one is a $w(x)$, $\tau_{ip} \prec_i \tau_{iq}$ or $\tau_{iq} \prec_i \tau_{ip}$, and

[1] A standard notation for a binary relation $\prec_i$ is a set of pairs (p, q) such that $p \prec_i q$

3. $\forall \tau_{ip} \in OS_i$, $\tau_{ip} \prec_i N_i$. ■

Point 1 enumerates the operations in T_i. Point 2 states that conflicting operations occurring within a transaction are ordered by $\prec_i$. Point 3 prevents any operations of a transaction from occurring after the transaction terminates.

2.1.2 Histories and Serializability

A history (schedule) records the execution order of a set of transactions. It contains the order of the operations of the transactions executed against the data in the database. The execution order of two operations in a history is significant if they conflict. Two operations conflict if they are executed on the same data item and one of them is a write operation. Given $\mathcal{T} = \{T_1, T_2, ..., T_n\}$, a history for $\mathcal{T}$ is defined as follows:

Definition 2 *(History):* A history H is a partial order $H = (\sum, \prec_H)$, where:

1. $\sum = \bigcup_j \sum_j$ where $\sum_j$ is the domain of transaction $T_j \in \mathcal{T}$,
2. $\prec_H \supseteq \bigcup_j \prec_j$ where $\prec_j$ is the ordering relation for transaction T_j at the $DBMS$, and
3. for any two conflicting operations $p, q \in H$, either $p \prec_H q$ or $q \prec_H p$. ■

Point 1 guarantees that the operations in H are precisely the operations submitted by T_1, T_2, ..., T_n. Point 2 refers to all operation orderings specified within the transactions. Point 3 ensures that the conflicting operations of all transactions are ordered. The definition of history and transaction enables the discussion of serializability and reliability in a database management system.

Correctness Criteria

A common correctness criterion used for classical transactions is *Conflict serializability.* Conflict serializability guarantees that conflicting operations in a set of transactions are executed so that transaction executions produce the same result as if they are executed serially. Serial execution of a set of transactions in a history is defined as follows:

Definition 3 *(Serial):* A history H is serial (SR), iff ($\exists p \in T_i, \exists q \in T_j$, where $p \prec q$) $\Longrightarrow$ ($\forall r \in T_i, \forall s \in T_j, r \prec s$). ■

This definition indicates that for every two transactions T_i and T_j, either all operations of T_i appear before all operations of T_j or vice versa.

Executing transactions serially unnecessarily orders the non-conflicting operations as well. However, transactions can interleave freely so only conflicting operations are ordered. Before we define this concept of correctness, we must define when two histories are equivalent.

Definition 4 *(Conflict Equivalent):* Two histories are conflict equivalent if they are defined over the same set of transactions and identical operations of nonaborted transactions are ordered in the same way. ■

Definition 5 *(Conflict Serializable):* A history H is conflict serializable iff it is conflict equivalent to a serial history. ■

Serializability of a history is checked by constructing a serialization graph. For history H over transactions $\mathcal{T} = \{T_1, T_2, ..., T_n\}$, we define a serialization graph for H by:

Definition 6 *(Serialization Graph):* The serializable graph (SG) for H, denoted $SG(H)$, is a directed graph $SG(H) = (\Gamma, \lambda)$. The nodes ($\Gamma$) are the transactions in $\mathcal{T}$ that are committed in H and the edges (λ) are all $T_i \rightarrow T_j$ ($i \neq j$) such that some operations of T_i precede and conflict with some operations of T_j in H. ■

Theorem 2.1.1 A history H is serializable iff $SG(H)$ is acyclic.
Proof: See Bernstein *et al.* [BGH87] page 33. ■

Another possible correctness criterion is *view serializability.* While a conflict serializable history must be conflict equivalent to a serial history, a view serializable history must be view equivalent to a serial history. Before we define view equivalent, we need to explicitly define "reads from" and "final write" relations.

Definition 7 *(Reads-From):* A transaction T_i reads x from T_j in a history H, if:

1. $w_j(x) \prec_H r_i(x)$,
2. $r_i(x) \prec_H a_j$,
3. $\forall w_k(x)$, if $w_j(x) \prec_H w_k(x) \prec_H r_i(x)$, then $a_k \prec_H r_i(x)$. ■

Point 1 states that the write operation on x in T_j must precede the read operations in T_i. Point 2 ensures that T_i reads x from T_j, only if T_j is either still active or already committed. Point 3 guarantees that no other non-aborted transactions have updated x between the time that T_j updated x and T_i read x.

Definition 8 *(Final Write):* The final write of data item x in a history H is the operation $w_i(x) \in H$, such that $a_i \notin H$ and for any $w_j(x) \in H$ $(j \neq i)$ either $w_j(x) \prec_H w_i(x)$ or $a_j \in H$. ■

Definition 9 *(View Equivalent):* Two histories H_1 and H_2 are view equivalent if:

1. they are defined over the same set of transactions and have the same operations,
2. for any nonaborted T_i and T_j in H_1 and H_2 and for any x, if T_i reads x from T_j in H_1 then T_i reads x from T_j in H_2, and
3. for each x, if $w_i(x)$ is the final write of x in H_1 then it is also the final write of x in H_2. ■

Note that in the above definition, equivalence of two histories is not based on ordering of the conflicting operations but histories are checked for the same reads-from and final write relations. This leads to the definition of view serializability.

Definition 10 *(View Serializable):* A history is view serializable if it is view equivalent to some serial history. ■

Most systems use conflict serializability. Recently researchers have attempted to create new correctness criteria to overcome the limitation of conflict serializability but that do not suffer from being NP-complete, like view serializability [DE89, SRK91, KM94]. These initiatives are discussed in the next chapter.

2.1.3 Reliability

Reliability refers to the resiliency of a system to various types of *failure* and the system's ability to recover from the failures. Database failures can occur from software or hardware faults. The three most common types of failures are *transaction failure*, *system failure*, and *media failure*.

If a transaction fails to complete its task, it will abort. Abortion occurs for a number of reasons. For example, if a user inputs improper values for the data, the transaction cannot proceed with its execution. It is also possible to abort a transaction involved in a deadlock cycle to allow other transactions to obtain resources and continue their executions.

A system failure occurs if one or more components of the system fail. For example, a power failure, main memory fault, or a software fault in the operating system. System failures typically results in the loss of information in main memory.

Finally, media failure refers to the failure of a secondary storage device. For example, a disk head may crash or the controller fails to operate properly. In some systems, the disk information is copied to another secondary storage periodically to help protect against media failures [OS91].

Recovery

Recovery refers to the mechanism used by a database system to bring the database into a consistent state after a failure occurs. Therefore, when a failure occurs, the recovery system must ensure that the effects of all committed transactions remain and no effect of aborted or active transactions persist in the database.

The most common recovery mechanisms use *logging*. A log is stored in non-volatile memory where the effects of transactions on the data are recorded. When a failure occurs, the log is used to *undo* the effects of active and aborted transactions and *redo* committed transactions. Different forms of logging have been used in traditional databases [BGH87] and are being developed for the newer systems [RM89, Mos87, Wie94].

A non-serial history may not always be recoverable. For example, it is possible that transaction T_2 reads data written by transaction T_1 before T_1 commits. If T_2 commits and later T_1 aborts, abortion of T_1 triggers abortion of T_2. Unfortunately, T_2 cannot be aborted because its commit operation may have made its effects user visible [BGH87]. Therefore, to create a recoverable history some operations need to be ordered. Using the "reads from" relation (Definition 2.1.7), a recoverable history is defined.

Definition 11 *(Recoverable):* A history H is recoverable (RC) if when T_i reads from T_j $(i \neq j)$ in H and $c_i \in H$, $c_j \prec_H c_i$. ■

This assures that a transaction commits after all transactions (other than itself) from which it read have already committed.

It is also possible to set more restriction on the order of the operations to avoid cascading aborts.

Definition 12 *(Avoid Cascading Abort):* A history H avoids cascading abort (ACA) if whenever T_i reads x from T_j $(i \neq j)$, $c_j \prec_H r_i(x)$. ■

This ensures that transactions read only values written by committed transactions or itself.

Definition 13 *(Strict):* A history H is strict (ST) if whenever $w_j(x) \prec_H o_i(x), (i \neq j)$, either $a_j \prec_H o_i(x)$ or $c_j \prec_H o_i(x)$ where $(o_i(x) \in (r_i(x), w_i(x)))$. ■

Strictness means that no data item may be read or overwritten until the transactions that previously wrote it terminates. The following theorem is a useful characterization of these recovery types.

Theorem 2.1.2 $ST \subset ACA \subset RC$

Proof: See Bernstein *et al.* [BGH87] page 35. ■

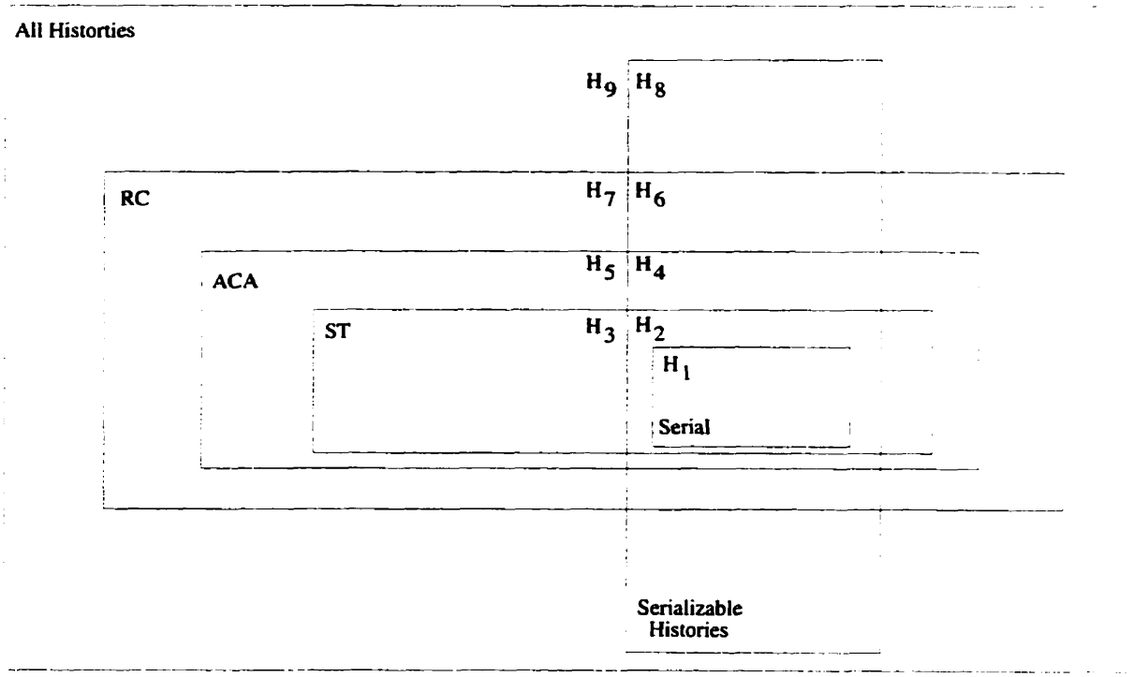


Figure 2.1: Relationship between histories

The relationship between histories is shown in Figure 2.1. A history can be a combination of one of more of the above types. For example, as shown in Figure 2.1, H_1 is the set of all serial histories. H_2, H_4, H_6, and H_8 refer to serializable histories which are strict, avoid cascading abort, recoverable and non-recoverable, respectively. Similarly, H_3, H_5, H_7, H_9 are collection of non-serializable histories which are strict, avoid cascading abort, recoverable and non-recoverable, respectively. It can be shown that each set represented by $H_1 ... H_9$ is non-empty.

2.1.4 Concurrency Control

"Classical" concurrency control methods are divided into two broad categories: pessimistic and optimistic. A pessimistic algorithm orders transactions by delaying the processing of conflicting operations. Operation ordering can occur statically (before execution) or

dynamically (during execution). Optimistic algorithms do not delay the operations as they arrive but aborts them if serializability is violated. Obviously, they may be re-executed.

Pessimistic Approach

Most systems [EGLT76, BG81, CFLN82, ZB93a] use a pessimistic algorithm called *two phase locking* to support concurrency control. Transactions obtain read and write locks so that multiple transactions may read a common data item but only a single transaction can hold a write lock. A transaction can acquire locks until it releases one, that is the end of phase one, from which point it is prohibited from acquiring any other locks. Most systems implement a version of two-phase locking called *strict two phase locking* where all locks are held to the end of the transaction to avoid cascading aborts.

An alternative approach is timestamping that was initially proposed by Reed [Ree78]. A unique ordered timestamp is associated with each transaction before the transaction is executed. Every time an operation of transaction T_i accesses data item x, x obtains the timestamp of T_i. Conflicting access to x can only be in increasing order of timestamp. Therefore, if T_i has already read x, another transaction T_j can write x if timestamp of T_j is greater than the timestamp of T_i. Furthermore, if T_i has written x, T_j can access x if its timestamp is larger than the timestamp of T_i. Disallowing an operation of a transaction, leads to abortion of that transaction.

Optimistic Approach

Other systems use an optimistic concurrency algorithm. Early work by Kung and Robinson [KR81] proposed the *serial validation algorithm.* This algorithm records read and write operations of each transaction T_i in two separate sets called the *readset*(T_i) and *writeset*(T_i), respectively. Transactions are executed with no restriction until commit time; however, they must be validated before the final commit or abort decision is made. Let T_j be a transaction committed during the life time of T_i. T_i is committed if for all T_j, the intersection of readset(T_i) with writeset(T_j) is empty. This indicates that T_i is allowed to commit if and only if no other transaction has updated any data item read during the time T_i is active. Otherwise, T_i is aborted and re-executed.

Another optimistic concurrency approach constructs a serialization graph for the history of the transactions after it completes [Bad79]. Every time a transaction commits, a serialization graph is constructed. If the graph contains a cycle, the transaction is aborted and its operations are rolled back. Otherwise, the effects of the transactions are recorded in the database.

Mixed Method

Different concurrency control methods can be combined to enhance concurrency in some cases. For example, Carey [Car87] combined Kung and Robinson optimistic serial validation algorithm [KR81] with timestamp ordering [Ree78] to produce a *timestamp based serial validation algorithm*. Carey argues that with large numbers of transactions committed during the life time of T_i, Kung and Robinson's algorithm is too costly. Carey assigns each transaction T_i a start up timestamp $S\text{-}TS(T_i)$ and a commit timestamp $C\text{-}TS(T_i)$. Further, each data item x is timestamped $TS(x)$ with the value of the commit timestamp of the most recent write of x. T_i can commit iff for every x in the readset of T_i, $S\text{-}TS(T_i) > TS(x)$. The main advantage of this algorithm over the simple optimistic method presented by Kung and Robinson is that stale data read by a transaction may be detected before the transaction is entirely executed. This provides lower execution cost and better resource utilization.

In brief, some concurrency control algorithms perform better in one environment than another. Optimistic algorithms are appropriate for the environment where the likelihood of the conflicting transactions is low. However, pessimistic protocols are suitable where the rate of conflict is relatively high. Comparison of the performance of pessimistic versus optimistic protocols is done and discussed in the literature [Car83a, ACL87, CM86]. We will review them later in this chapter.

Classical transaction models are most appropriate for business and administrative environments where transactions are simple, short-lived, and non-hierarical. Applications of new environments such as Computer Aided Design systems (CAD) or some cooperative environments demand more advanced transaction models. In the next sections, we study some advanced transaction and data models which motivate our research.

2.2 Nested Transactions

Nested transactions are an extension of flat transactions that include subtransactions in addition to primitive read and write operations. The idea of nesting transactions was extensively investigated after the work of Reed [Ree78] and Moss [Mos85]. This section reviews nested transactions and discusses their advantages over classical models.

A nested transaction is a tree where nodes are subtransactions and edges are calls to subtransactions. The root node is called a *top-level transaction.* The leaves are the primitive reads and writes on data. Except for the leaves, other nodes are called *parents* and can have unlimited children. The top-level transaction does not have a parent. Transactions which call other transactions either directly or indirectly are called ancestors. Transactions which are called either directly or indirectly by other transactions are called the descendants. Since a transactions may call itself recursively, we speak of proper descendant and proper ancestors to exclude the transaction itself. A transaction hierarchy includes the transaction descendants. The top-level transaction and its proper descendants form a transaction family.

One advantage of nested transactions over classical transactions is that nested transactions provide a means of controlling concurrency within transactions. In a classical model, a long running flat transaction may have to be broken into several shorter transactions to produce better and faster performance [PKH88]. Breaking a transaction in this way creates overhead. In a nested transaction model, a long running nested transaction dynamically creates a set of subtransactions and distribute the task among them. Subtransactions work independently and can be executed simultaneously.

Another advantage is that nested transactions provide better control over transaction failure. The effects of a failure is limited to a portion of a transaction. In classical transactions when there is a failure, the entire transaction is aborted. Therefore, the system time and resources used for the transaction is wasted. This problem is more significant when the transaction is long and it is close to its commit point. In nested transactions when a subtransaction fails, we may only need to recover that subtransaction. Since subtransactions

act independently, failure of one component does not necessarily effect other components.

Nested transactions fall into two broad categories: *closed nested transactions* and *open nested transactions.* In closed nested transactions [Mos85], the partial result of subtransactions are not visible to other transaction families. If a child commits, all of its scheduling information is passed to its parent. Conversely, if it aborts, resources such as locks are released back to the system. A parent can commit if and only if all of its children have committed. Abortion of a parent causes abortion of its children.

In open nested transactions [WS91], once a subtransaction commits, its result are recorded in the database and become visible to other transaction families. Since other transaction families may read and update the partial results of other transaction families, problems such as cascading aborts and loss-of-update may occur [BOH+91]. Compared to the closed nested model, open nesting allow more concurrency because subtransactions do not need to wait for the result of other subtransactions when they commit.

An alternative model

Fekete *et al.* [FLMW90] introduced two ways of viewing nested transactions: *procedural abstraction* and *data abstraction.* The material presented above refer to procedural abstraction. Data abstraction, known as *multilevel transactions*, is another variant of nested transactions as defined by Weikum [Wei91] and Berri *et al.* [BBG89]. A multilevel transaction model has a layered system structure. Each layer (level) provides a well-defined interface of objects and operations. The implementation of one level is done based on the objects and operations of the level below it. The bottom level includes primitive operations which cannot be further decomposed. In contrast to procedural abstraction transaction models, in a multilevel transaction tree, the depth is fixed and all leaf nodes are at the same level.

2.2.1 Concurrency Control

The proposed concurrency control techniques for nested transactions known as *upward lock inheritance* was provided by Moss [Mos85]. We review the basic rules and discuss extensions suggested by others.

Operations acquire three types of lock: a read, a write, or a none lock. Based on restrictiveness, the lock modes are ordered as *none* < *read* < *write*. Transactions may either *hold* or *retain* the locks. If a transaction holds a lock, it can access the locked object. If a transaction retains a lock, it cannot access the object but it blocks other transactions from setting conflicting locks anywhere in their transaction's hierarchy; however, the descendants of the retainer potentially can use the lock. When a transaction becomes a retainer of a lock, it remains as the retainer of the lock until the transaction terminates. The basic locking rules for nested transaction T_i presented by Moss [Mos85] are:

- Transaction T_i can hold a write lock on x, if no other transaction holds a conflicting lock on x. Only ancestors of T_i may retain a conflicting lock on x.
- Transaction T_i can hold a read lock on x, if no other transaction holds a write lock on x. Only ancestors of T_i may retain a conflicting lock on x.
- When Transaction T_i commits, it passes all of its hold/retain locks to its parent. The parent retains the locks it receives from its children.
- When Transaction T_i aborts, all of its locks are discarded. This does not effect any of the proper ancestors of T_i holding or retaining the same locks.

When a parent inherits a lock on x from its child, it allows other transactions in its hierarchy to view and update the current state. The parent, however, prevents transactions in other hierarchies from updating x as long as it keeps the lock.

In the nested transaction model presented by Moss [Mos85], non-leaf nodes do not contain primitive operations. More recent work has relaxed or eliminated this restriction. Recent nested transaction models [HH91, Zap93] now allow subtransactions to include primitive operations and calls to other subtransactions. Harder and Rothermel [HR87] argue

that upward inheritance is not always applicable in this recent work. For example, a parent transaction may read x and issues a child to update x. Upward inheritance fails in this case because the parent must wait for its child to terminate successfully before it can commit and the child must wait for its parent to release its lock. To solve this problem, Harder and Rothermel [HR87] suggest another locking algorithm called *downward inheritance.*

In downward inheritance, transaction T_i holding a lock, can pass the lock to any of its descendants in the same hierarchy. After passing the lock, T_i retains the lock in the same mode. This rule can also create a problem in some cases. For example, a transaction may read a few objects and create some subtransactions to execute other operations on these objects. Children obtain the locks from the parent and may execute simultaneously. Concurrent executions of the subtransactions increase concurrency; however, since the parent and its children share the same information, one may overwrite the information read by another.

Harder and Rothermel [HR93] subsequently suggest a solution. They add two more rules to the basic locking rules of Moss [Mos85]. Using the same locking modes as described above ($none < read < write$), a transaction T_i holding a lock in mode M can

- downgrade its lock to mode M' ($M' < M$), if it retains the lock in mode M and holds the lock in mode M'.
- upgrades its lock to mode M' ($M < M'$) if no other transaction holds the lock in a conflicting mode with M' and all of the transactions that retain the lock in conflicting mode with M' are ancestors of T_i.

This algorithm is less restrictive than the simple downward inheritance and it enhances overall concurrency for nested transactions because it allows more desirable decomposition of transactions into a set of cooperating subtransactions.

2.3 Object-Oriented Data Models

Conventional data models are simple but suffer from several limitations [BM91, HPC93, ABD+89]. They are not suitable in environments where data is complex, schemas changes frequently, and transactions take much more than a few seconds to execute. Object-oriented data models may overcome these limitations. They satisfy the requirements of conventional data models such as data-sharing, consistency, integrity, and concurrency control while introducing many useful features. This section explains the concept of object data models and discusses some of the features of object data models which are not efficiently supported in conventional data models. Transaction management and concurrency control in object based systems are also addressed in this section.

2.3.1 The Concept

Object data models model *all* entities as objects. Each object is uniquely identified. An object contains a set of attributes which form its *structure*. Attributes are either simple: such as integers and strings; or complex values, which may be objects. The values of the attributes determine the state of the objects. Objects also include a set of procedures called *methods*. A method of an object contains a set of operations on that object. The execution of a method is the only mean to change the state of the objects.

Some features which separate object data models and conventional data models are *encapsulation, inheritance, and aggregation.* These characteristics create a flexible environment in terms of object manipulation, and impact on concurrency control and recovery.

Encapsulation

Encapsulation ensures that only local object methods access an object's attributes. This is similar to the concept of *abstract data types.*

Inheritance

Objects sharing the same structure and behavior are grouped into a *class.* A class can

be a *specification/generalization* (subset/superset) of one or more other classes. Class B is a subclass of class A if B inherits the properties of A. A class with its direct and indirect subclasses form a class hierarchy. This hierarchy is actually a lattice if multiple inheritance is supported. Different forms of inheritance appear in the literature [ABD+89]. Detailed discussion is beyond the scope of this thesis.

Aggregation

Aggregate objects known as *complex objects* contain other objects which in turn may include other objects. Therefore, complex objects have a tree structure too. A complex object is treated as a unit of retrieval and integrity enforcement.

A type of complex object called a *composite object* is implemented in Orion [KGBW90]. A composite object relationship is defined between objects to form a *part-of* relationship. Part-of relationship indicates that an object is a part of another object. Two important features of composite objects are: first, if the root object is deleted, all component objects (except the ones shared with other objects) are also deleted; second, a constraint such as a lock on the root object propagates to all other components.

Other Characteristics

- Objects may maintain several versions of an object. Versions of an object play an important role in transaction management and schema evolution.
- Objects must be persistent. Persistency ensures that objects survive the execution of a process and they can be reused in another process.
- Object data model may support schema change. Possible modifications include changes to the definition of a class and changes to the inheritance hierarchy.

2.3.2 Prototypes

Since the foundation of Simula in 1958 and Smalltalk [Gol84] in 1980, many object oriented prototypes have been developed [WLH90, KGBW90, BB89, RH90]. These prototypes targeted many advanced application models such as Computer Aided Design (CAD/CAM),

artificial intelligence, and office information systems. The prototypes are constructed by either (1) building on the top of traditional systems or (2) building a "new" facility for storing persistent objects. In the first category, a new layer of transaction management is added above underlying system to provide object-oriented facilities (eg. Postgres [WLH90] and Exodus [Car89]). Conversely, systems in the second category (eg. Orion [KGBW90] and O_2 [BDK92]) require new transaction management tailored to objects. Since this research belongs in the second category, we limit the review to these systems.

2.3.3 Transaction models

Issues of transaction management such as serializability, concurrency control, recovery, and reliability are the key factors motivating this dissertation. Some recent material related to transaction mangement in objectbases include [RKS93, Zap93, Wie94, Gra94, Özs94]. The following paragraphs review work in this area. Note that since this section reviews other researchers work the terminology used here may be different than that described earlier. If this occurs, we will define the term as the other researchers have and use it with their meaning in this section. After this section, we will return to our terminology as defined earlier.

Rakow *et al.* [RGN90] proposed an object model using open nested transactions. The objectbase contains a set of homogeneous objects and a set of transactions on these objects called *object-oriented transactions*. Operations accessing objects are called *actions*. Actions are grouped into sets. The set for action A contains all of the actions called either directly or indirectly by A. An object-oriented transaction forms a tree where the nodes represent the actions and edges their invocation. The leaves are called *primitive actions* which are simple operations on objects.

In Rakow *et al.*'s algorithm, serialization is defined for transaction accessing individual object and for the entire transaction system. In the former case, an object schedule is defined at each object. An object schedule should be conflict equivalent to a serial schedule at that

object to ensure correctness [RGN90]. This is done by checking two types of dependency at each object: *transaction dependency* and *action dependency.* Two transactions on an object have a dependency if some actions in their hierarchies conflict. Two actions have a dependency if an ordering is enforced between some of their primitive operations, or they are transactions on other objects in which they are transaction dependent. Two object schedules are equivalent if they have the same transaction dependency relation. Furthermore, to ensure serializability for the entire transaction system, another dependency relation called *added dependency* is introduced. An added dependency is given to two actions A and B, if A and B are actions of two different objects which include some actions in object C, and C creates a transaction dependency between A and B.

Hadzilacos and Hadzilacos [HH91] introduced another object model using closed nested transactions. In this model, an object consists of a collection of attributes and a set of procedures called *methods.* Methods are the sequences of *local* and *message steps.* A local step of an object results from the execution of primitive read and write operations on the object. Message steps invoke methods possibly in other objects. Transactions submitted by the user only include message steps. Objects execute the messages by invoking the appropriate methods.

Hadzilacos and Hadzilacos define two types of synchronization: *Intra-object* and *inter-object* synchronization. Intra-object synchronization serializes operation within an object. Inter-object synchronization ensures consistency of the independent synchronization decision made at each object. This implies that if T_1 and T_2 are two transactions accessing object o^f and T_1 is ordered before T_2, T_1 must be ordered before T_2 in all of the objects accessed by T_1 and T_2. Intra-object and inter-object serializability together must guarantee the serializability of overall computations in all objects.

A history H consists of a set of method executions on objects where local and message steps of the methods are partially ordered according to intra and inter object synchronization rules. Methods cannot be invoked recursively in H. If method M_1 is ordered before method M_2 in H, all the methods called by M_1 are ordered before the methods called by

M_2.

Serialization of a given history is checked by constructing a *direct access graph* where nodes are the message steps and edges are calls to the message steps. An edge is added between two nodes N_1 and N_2 if either

- descendants of N_1 and N_2 conflict or
- if L is the least common ancestor of N_1 and N_2 and ancestors of N_1 and N_2 conflict within L's hierarchy.

If the graph is acyclic, H is serializable.

2.3.4 Concurrency Control

A concurrency control algorithm based on Hadzilacos and Hadzilacos [HH91] model is presented by Agrawal and El Abbadi [AA92]. This algorithm is the extension of locking protocol presented in [Mos85] for closed nested transaction models. Agrawal and El Abbadi define two types of relations between operations: *shared relation* and *ordered relation*. If two operations conflict, each possess an ordered relation with respect to the other. When an operation τ acquires a lock on an object, the lock has an ordered relation with respect to all operations with which an ordered relation and a shared relation with respect to locks of all operations with which τ has a shared relation.

The algorithm is as follows. A lock is associated with each operation τ. Shared and ordered lock relations with other locks is set based on the above description. A transaction maintains all its locks until it either commits or aborts. A transaction must wait for the termination of all its children before it commits. Locks held by a transaction are discarded if a transaction aborts. A transaction holding a lock passes the lock to its parent when it terminates successfully. Finally, T_i must wait for T_j to commit, if some lock held by T_i have ordered relation with some lock held by T_j and a parent of T_i is a proper ancestor of T_j. This is called the *Ordered Commitment Rule.*

Another algorithm proposed by Resende and El Abbadi [RA92] is based on the serialization graph. They also use Hadzilacos and Hadzilacos' model. Their algorithm constructs a set of graphs; one for each method execution M. It is called Stored Children's Serialization Graph of M ($SCSG(M)$). $SCSG(M)$ is defined as a set of nodes $\Gamma(M)$ which represent the terminated children of M, and a set of edges $\lambda(M)$ which refer to relative execution ordering of children of M. Initially, $SCSG(M)$ is empty. A child c of M is added to the graph when c terminates. If another child c' of M already exists in the graph, an edge $c' \rightarrow c$ is added if either of the following two conditions hold:

- Given that L is the least common ancestors of c and c', and some ancestors of c in L are ordered before some ancestors of c' in L.
- Given that c' is a sibling of c, then there is local step in c' or in a descendant of c' which precedes and conflicts with another local step in c or in a descendant of c.

After c and its related edges are added to the graph, the graph is tested for cycles. If no cycle exists, the process continues; otherwise, the top level transaction associated with M is aborted. This algorithm is optimistic in the sense that an operation does not wait for others to finish execution. A transaction is aborted and may be restarted if the result of its execution does not meet the serializability criterion.

Zapp and Barker [ZB93c, ZB93a, ZB93b] define object serializability and a serialization graph technique to capture serializability in Hadzilacos and Hadzilacos' model. An architecture describing interactions between the components of the transaction facilities and a concurrency control algorithm are presented.

Zapp and Barker [ZB93c] define two types of transactions: *user* and *object* transactions, by adapting concepts from the nested transaction model. Each transaction forms a tree whose root (the *top-level* transaction) is the user-transaction and whose descendants are called *object transactions*. Two types of histories are required. One for object transactions and another for user transactions. An object history defines the ordering relation of object transactions that have executed at an object. A user history defines an ordering relation of user transactions which contain the orderings of the user transaction operations. To ensure

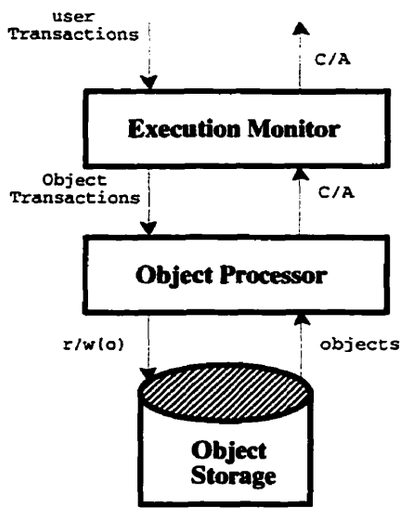


Figure 2.2: Zapp and Barker's architecture

serializability and thereby assess correctness, the user history and the object history are combined into a global history called the *global object* history.

Zapp and Barker's architecture is composed of two major components: an Execution Monitor and an Object Processor (see Figure 2.2). The purpose of the Execution Monitor is to provide an interface to users and to schedule the method invocations on behalf of user transactions. To schedule a method, the Execution Monitor submits the method to the Object Processor as an object transaction. The Object Processor schedules and executes each individual method's operations. In processing method executions, the Object Processor retrieves and updates object attributes by accessing the persistent object store. The result from the execution of a method is returned to the Execution Monitor. When a transaction commits or aborts, the Object Processor guarantees the ACID properties.

The expanded architecture of Figure 2.2 is shown in Figure 2.3. The Execution Monitor contains two components: the Transaction Manger and the Method Scheduler. The Transaction Manager receives user transactions and submits methods to the Method Scheduler. The Method Scheduler is responsible for scheduling user transaction operations and passes

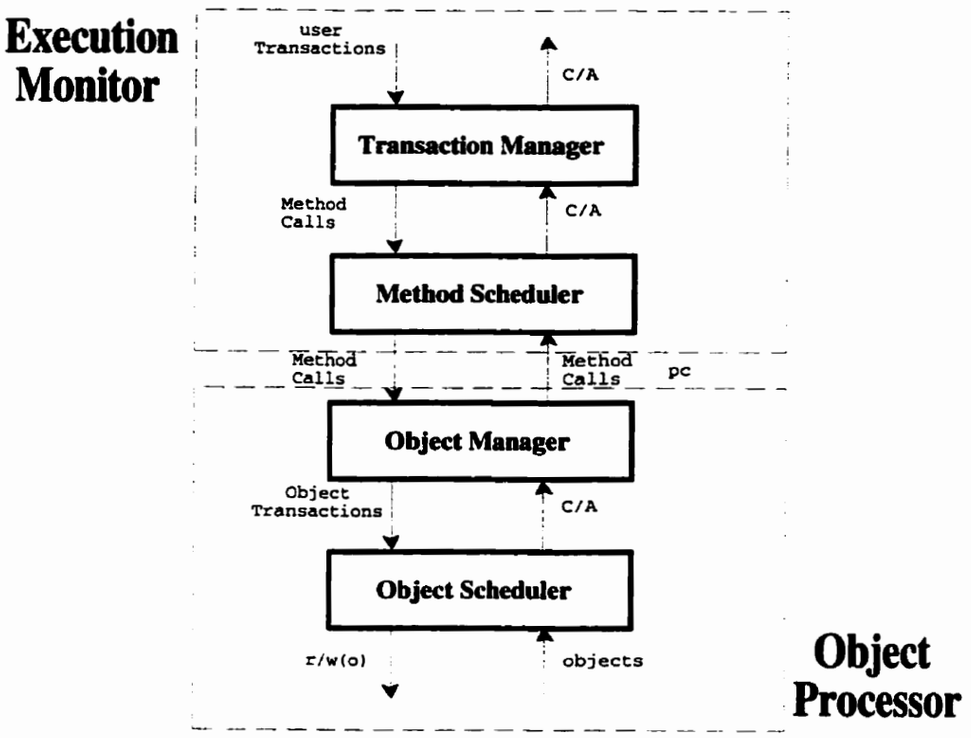


Figure 2.3: Zapp and Barker's expanded architecture

the scheduled methods to the Object Processor for execution.

The Object Processor also contains two components: the Object Manager and the Object Scheduler. When the Object Manager receives a method, it converts the method to an object transaction and sends it to the Object Scheduler. The purpose of the Object Manager is to facilitate the communication between the Method Scheduler and the Object Scheduler. The object transaction received by the Object Scheduler is executed against objects in the object base.

The operations of an object transaction are read, write, and method invocations. The read and write operations are executed against an object in the objectbase. The method invocations must be sent to the Method Scheduler so that they can be scheduled with other methods of the user transaction level. When an object transaction terminates (successfully or unsuccessfully), the Object Scheduler sends the result of the execution to the Object Manager and the Object Manager in turn passes the message to the Method Scheduler. An object transaction pre-commits as the result of a successful termination. A pre-committed object transaction must wait until it receives a commit message from a coordinator. If an object transaction aborts, it must release all the resources it is holding. The execution of two object transactions T_1 and T_2 on an object is controlled as in the strict two phase locking protocol. If T_2 requests a lock on an object which conflicts with the lock already set by T_1, T_2 is blocked. When T_1 pre-commits, T_2 can be processed if T_1 and T_2 are from the same user transaction. Otherwise, T_2 must wait until T_1 commits completely and releases its locks.

Another pessimistic concurrency control algorithm is presented by Graham and Barker [GB95, GB94a]. Concurrency is performed based on the static analysis information captured at pre-run time. Information includes *control flow information*, *method invocation information*, and *attribute reference information*. Control flow information mainly details what sections of the methods might be executed. Method invocation information determine the calling sequence between objects, and attribute reference information in a method include the read/write relationships. The detail of the deriving these information are presented

in [Gra94].

Graham and Barker discuss intra-object and inter-object concurrency as follows. Suppose a user transaction UT_i directly invokes a set of message steps m_{i1}, m_{i2}, ..., m_{in}. For every pair of message steps $< m_{ik}, m_{ij} >$, m_{ik} and m_{ij} are executed concurrently if they directly or indirectly access a disjoint set of objects. Otherwise, for every object o^f which may commonly be accessed (possibly indirectly) by m_{ik} and m_{ij} in a conflicting manner (conservative assumption), a message is sent to o^f to enforce serialization order between m_{ik} and m_{ij}.

Each object o^f contain a local scheduling graph (LSG). Vertices of $LSG(o^f)$ represent the methods (object transactions) that are either active at o^f or may eventually execute at o^f. When an ordering message $m_{ik} \rightarrow m_{ij}$ (m_{ik} happens before m_{ij}) is sent to o^f, an edge is added to the $LSG(o^f)$ to order the execution of m_{ik} after m_{ij} at o^f. When a method is invoked in o^f, its execution is blocked if $LSG(o^f)$ indicates that some other methods should be executed first. When the execution of an object transaction terminates, its corresponding node and all related edges are removed from $LSG(o^f)$; thereby, some object transactions may be unblocked and executed.

The *Global Serialization Graph* of Zapp and Barker [ZB93b] is adapted to control inter-object serialization. When a new user transaction arrives, the set of objects it may reference is compared with the set of objects that might be referenced by the currently active transactions. Suppose set A and set B represent the set of objects referenced by the new transaction UT_i and a currently active transaction UT_j, respectively. If UT_i and UT_j both access $o^f \in A \cap B$ in a conflicting manner, a message is sent to o^f to set a serialization order between UT_i and UT_j. Graham and Barker have developed a function which returns the serialization order of the new user transaction with respect to the currently active transactions.

The above algorithm prevents deadlock problems because serialization is done statically prior to the execution of the transactions. However, some sub-transactions may still be aborted and rolled back. For example, if the new user transaction UT_i is scheduled after

all other currently active user transactions, no roll back is necessary. Otherwise, if UT_i is decided to be serialized before a currently active transaction UT_j, subtransactions of UT_j operated at some object o^f, may be rolled back if UT_i references o^f.

2.4 Multiversion Data Models

Most database systems keep one version of data. In multiversion database systems more than one version of data can be created and stored. Versions associated with data item x show the evolution of x as x is updated by some transactions.

Using multiple versions of data items as a transaction synchronization technique can enhance concurrency and support recovery. Transactions interested in the old versions of data can be executed concurrently with other transactions which compete to access the latest committed versions of data. In addition, multiversion systems do not usually require logging because before and after images can be searched via the versions. Multiversion systems have increased overhead such as purging unnecessary versions and controlling their number.

This section reviews multiversion serialization theory including notions of serializability in a multiversion environment. Concurrency control and how to enhance concurrency through versioning is also described.

2.4.1 Histories and Serializability

A multiversion history is different from a single-version history in at least two ways. First a "write" operation on a data item x may produce a new version of x, keeping both versions. Second, if more than one version of x exists, a "read" operation may not always be restricted to reading the most recent version of x. This indicates that the definition of *conflict* may also change in a multiversion system. For example, two write operation each producing a different version do not necessarily conflict. Similarly, a read operation

reading the old versions does not conflict with a write operation unless other restrictions are involved in the model. Bernstein *et al.* [BGH87] extend one-version serializability to multiversion serializability as follows:

- A write operation of T_i on x produces a new version x_i.
- A read operation of T_i reads x_i. If x_i has not been produced, the last committed version of x is read.
- Before a transaction commits, every transaction which produced versions it read must have committed already.

Based on the above conditions, a serial multiversion history called *one-copy serial* is defined as follows:

Definition 14 *(One-Copy Serial):* A serial multiversion history H is one-copy serial if for every T_i and T_j in H:

1. $(\exists p \in T_i, \exists q \in T_j$, such that $p \prec q) \Longrightarrow (\forall r \in T_i, \forall s \in T_j, r \prec s)$ and,
2. for all i, j, and x, if T_i reads x from T_j, then either $T_i = T_j$, or T_j is the last transaction preceding T_i that wrote into any version of x. ■

A multiversion serializable history is defined.

Definition 15 *(One-Copy serializable):* A multiversion history is one-copy serializable if it is conflict equivalent to some one-copy serial multiversion history. ■

This serializability definition does not set any limit on the number of versions that can be created for each data item. Practical system considerations such as system storage capacity requires that limits be placed on the number of versions that can be simultaneously managed. Problems related to space limitation caused by maintaining multiple versions have been discussed in the literature [BHR80, HP86, PK84, CG85, Mor93] but it is beyond the scope of this thesis.

2.4.2 Concurrency Control

Several concurrency control algorithms have been proposed for centralized and distributed multiversion databases [SR81, BG81, CFLN82, Lau83]. Multiversion timestamp ordering was introduced by Reed [Ree78]. The multiversion two phase locking protocol was proposed by Chan *et al.* [CFLN82]. Examples of initial optimistic concurrency control schemes in multiversion environment are presented in [Ree78, SLR76].

Bernstein and Goodman [BG83] modify traditional timestamp ordering and two phase locking for the multiversion environment. Before discussing their work, some nomenclature is required. We use the notation $r_i(x_j)$ when T_i reads a version of x written by transaction j, and $w_i(x_i)$ denotes a write operation on data item x by transaction i; thereby creating a new version x_i. Further, acquisition of a read or a write lock on x indicates that no other transaction can obtain a conflicting lock on any version of x.

Multiversion timestamp ordering ($MVTO$) assigns each transaction T_i and its operations a unique timestamp, $ts(T_i)$. A multiversion timestamp scheduler, processes operations in a first-come, first-served order. When the scheduler executes a read operation of transaction T_i, the version of x with the largest timestamp less than or equal to $ts(T_i)$ is read. A write operation of T_i is processed in one of two possible ways. A write operation is rejected if the scheduler has already processed a $r_j(x_k)$ such that $ts(T_k) < ts(T_i) < ts(T_j)$. Otherwise, the scheduler executes $w_i(x_i)$. Note that the last committed version of x may not necessarily be the x with the largest timestamp. For example, it is possible that $ts(T_i) < ts(T_j)$ and the scheduler processes a $w_j(x_j)$ before a $w_i(x_i)$. Finally, the commit of T_i, c_i, is delayed until all other transactions, T_j, that wrote versions read by T_i have committed. This delay ensures recoverability.

A *multiversion two phase locking* ($MV2PL$) scheduler uses three lock types; *read, write,* and *certify.* Two locks conflict if either one is a certify lock or write locks. Read and write locks are acquired during transaction execution as in two phase locking [EGLT76]. In processing a read operation on a version of x, the *ith* version of x is read if the scheduler

has processed a $w_i(x_i)$ already. Otherwise, the last committed version of x is read. Read locks are granted only if no other transaction holds a conflicting certify lock (i.e.: $r_i(x_j)$ is executed if no certify lock is holding x_j). When the scheduler receives a write operation on a version of x, it delays the execution if another transaction has a write or a certify lock on any version of x. Otherwise, it sets a lock on x and executes $w_i(x_i)$. A new version of x, x_i, results from this execution. When a transaction wants to commit, its write locks are changed to certify locks. The effect of certify locks is to delay transaction T_i's commitment until all of the active transaction readers of some versions of the data items written by T_i terminates. Thus the scheduler can only convert a write lock on x_i into a certify lock if there are no read locks on any certified versions of x. A version of x, x_i, is certified when T_i, the transaction that wrote x_i, commits. T_i is certified when all the versions it read have been certified.

Read-only Versus Update Transactions

One of the main advantages of multiversion concurrency control ($MVCC$) protocols over one-version concurrency algorithms is the reduction of data contention between read only transactions and update transactions. Read only transactions may see old versions of data but they can be executed concurrently with other update transactions. Some recent work have been done by Mohan *et al.* [MPL92] and Wu *et al.* [WYC93]. Wu *et al.* use dynamic finite versioning to enhance the performance of systems where short update transactions and long read only queries execute concurrently. Queries read from a small, fixed number of dynamically derived transaction consistent logical snapshot of the database. Snapshots may include old versions of some data.

Logical snapshot production may require stopping transactions submission and wait until all active transactions commit. An alternative solution is to mark the versions produced by active transactions and remove the marks when transactions commit. In this case, the snapshot of the database will include the unmarked data. Wu *et al.* argue that the former solution is not feasible because it is inefficient to stop and restart the process. They also claim that in the latter solution, searching for the marked versions incur a high cost because

versions can be scattered around in the database.

Wu *et al.* suggests the following algorithm. When transaction T_i becomes active, its *id* is entered in a list kept for the active transactions. Every time T_i produces a new version of data item x (x_i) it maintains a time-invariant footprint of T_i. When T_i commits, its *id* is removed from the list. A snapshot of the database is created by including all the versions in which their time-invariant footprint are not from the transactions in the list.

2.4.3 Multiversioning in Objectbases

The notion of version control is provided in object-oriented prototypes such as Orion [KGBW90], Iris [WLH90], and Avance [BB89]. The following summerizes some common features of versions in these protocols:

- Several versions can be derived from an object and other versions can be derived from these versions. An object with all its direct or indirect versions derived from it form a version hierarchy.
- Each version has its own unique identifier. Therefore, it can be accessed and modified directly.
- Versions are divided into two groups: stable and unstable versions. Stable versions are considered consolidated and are not usually updatable. Unstable versions are not yet consolidated and can undergo modification. Versions derived from a stable version are unstable versions. Unstable versions can be promoted to stable versions on user request or automatically by the system.

2.4.4 Concurrency control in Multiversion Objectbases

Nakajima [Nak92] presents an optimistic multiversion concurrency control mechanism. Multiversioning techniques are applied to the concepts of backward and forward commutativity introduced by Weihl [Wei88]. 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 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^f is called the last committed version of o^f (denoted $LCV(o^f)$), and the most recent uncommitted version of o^f is called the current version of o^f (denoted $CV(o^f)$). When transaction T_j invokes method m_k^f in object o^f a new uncommitted version of o^f (denoted $NV(o^f)$) is created for T_j. If the return result from $NV(o^f)$ backward commutes with $CV(o^f)$ or forward commutes with $LCV(o^f)$, $NV(o^f)$ becomes the new current version of o^f and replaces the old current version. Otherwise, $NV(o^f)$ is discarded and T_j invokes method m_k^f again.

Another multiversion transaction model for objectbase systems was proposed by Graham and Barker [GB94b]. An object is characterized based on its initial state, a set of valid states, its operations, and a set of transaction functions which move the object from one valid state to another. A transaction is simply a set of read and write operations. Objects are versioned and there exist only a single committed version of each object in the object base called the last committed version (denoted LCV). A set of transactions $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ which require access to object o^f, each obtains a copy of the last committed version of o^f, o^{fi}. The version o^{fi} is called the *base version* for the transactions in $\mathcal{T}$. The created copies are called the *active versions.* When every transaction in $\mathcal{T}$ terminates (commits or aborts), o^{fi} is deleted if a new committed version of o^f, o^{fj}, is created by another transaction.

Each transaction acquires its own set of active versions of objects. A transaction modifies its versions by executing the methods and does not interact with other transactions. If a transaction aborts, its versions are discarded. If a transaction completes successfully, before it commits, all of its active versions which conflict with their corresponding LCV in the object base must be reconciled. This allows correct effects to be reflected in the objectbase. When a transaction commits, its active versions become the new last committed versions.

Graham and Barker proposed an optimistic concurrency control algorithm. The main feature of this algorithm is that of reconciling the unsuccessful transactions instead of aborting them. The reconciliation procedure is as follows. Assuming that two transactions T_1 and T_2 execute against two separate copies of object o^f, o^{f1} and o^{f2}. The following four conditions are possible:

1. T_1 and T_2 are read only transactions
2. one of T_1 or T_2 is read only transaction and the other one is not
3. T_1 and T_2 are update transactions and one has less costly operations relative to another, or
4. T_1 and T_2 are update transactions and they have equally costly operations.

In case 1, the state of o^f remains unchanged because transactions do not conflict. In case 2, the new state of o^f is set to the update transaction. In case 3, the less costly transaction should be re-executed against the new state created by the other transaction. In the last case, either transaction can be executed before another. Thus, the execution order of T_1 and T_2 can be in either order depending on which is cheaper to compute. Therefore, the serialization order is determined by reconciliation.

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, or at least one transaction does not read the same data written by another transaction.

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 re-executed against the state created by another transaction. The cost of the re-execution of a transaction is estimated by static compile time analysis [Gra94]. Complex reconciliation of a transaction is mainly partial re-execution of operations which have read stale data in

that transaction. Reconciling an unsuccessful transaction at commit time is often a less costly procedure than the complete roll-back and re-execution of the transactions.

2.5 Performance Comparison

Carey and Mohan [CM86] investigated the performance of three algorithms. Reed's multiversion timestamp [Ree78], the version pool algorithm which is a type of multiversion two phase locking used by Computer Cooperation of America (CCA) in their *LDM* database system [CFLN82], and multiversion serial validation algorithm of [Car83b] which is based on the algorithm of Kung and Robinson [KR81]. The algorithms were compared both with each other and with their corresponding single versions (basic timestamp ordering, two phase locking, and serial validation algorithm). The performances overheads were analyzed based on throughput, average response time, number of disk accessed per head, work wasted due to restarts, and space required for old versions.

Experiments were performed with various mixes of read only and update transactions. It was shown that as the number of read only transactions increases relative to update transactions, the multiversion algorithms perform better than the single version counterparts. As the number of update transactions in the mix increased, the performance of basic timestamp ordering and serial validation algorithms significantly fell because some long-term read-only transactions starved and multiversion and regular two phase locking algorithms outperformed other algorithms.

In general, it was shown that multiversion protocols provided improvements in performance by allowing large read-only transactions to access previous versions of data items. Storage overhead to maintain old versions required satisfying read requests for ongoing transactions is not large, but the overall size of the version pool becomes significant when the read only transactions accessed more than 10 percent of the database or the number of update transactions becomes very large.

2.6 Summary

In this chapter, we presented an overview of the classical transaction model to create a framework for this dissertation. Then nested transactions and the issue of concurrency control in both closed and open nested environment were reviewed. Next, we explained some of the characteristics of object-oriented data models which reflect object manipulation, and concurrency control. Finally, the role of version management in database systems and the advantages and disadvantages of creating multiple versions of data were discussed. We also briefly mentioned some common features of object versioning in object-oriented prototypes.

The material presented by Graham and Barker [GB94b, GB94a, GB95] and by Zapp and Barker [ZB93c, ZB93a, ZB93b] are the closest to our research. We have adapted the model of Zapp and Barker for a closed nested transaction environments. Zapp and Barker's algorithm is a pessimistic approach. Graham and Barker [GB94b] provide a mechanism to maintain versions of objects in the objectbase; however, their work does not discuss reconciliation in detail for nested transaction models. We extend this work to a model and architecture for multiversion objectbases which supports nested transactions and provide a suitable reconciliation algorithm.

Chapter 3

The Computational Model

This chapter begins by formally defining class, methods, objects, and transactions. We then extend this formalism to reflect our computational model which incorporates versioning. Next a new correctness criterion is introduced to ensure serialization of concurrently executing transactions. Finally, concepts and definitions related to data dependency and information which can be captured by static analysis are presented.

3.1 Fundamental Concepts and Definitions

This section provides the definition of some fundamental concepts used in an object-oriented environment.

A *class* is a collection of homogeneous objects; that is, objects of the same structure and behavior belong to one class. Informally classes define the types of the objects.

A class $C = (),\mathcal{CA},\mathcal{CM})$ where i is the unique class identifier, CA is the set of attributes that the class defines such that forall $ca_j, ca_k \in CA$, $ca_j \neq ca_k$, and CM is the set of methods that the class defines such that forall $cm_j, cm_k \in CM$, $cm_j \neq cm_k$. In this dissertation, class i is denoted as C^i. The set of attributes and the set of methods of C^i are unambiguously referenced by CA^i and CM^i, respectively.

$CA^i = \{ca_1^i, ca_2^i, ..., ca_m^i\}$ contains the attributes defined by C^i. An attribute $ca_j^i \in CA^i$ is of type t where j is the unique attribute identifier in C^i and t is either a class (composite/complex objects [KGBW90, BDK92]) or a *primitive type.* A primitive type is a builtin type whose semantics is well-defined and understood by the compiler. Typically primitive types are those implemented by the underlying ontology. For example, a system set of primitive types might be $\{int, float, char, string\}$.

$CM^i = \{cm_1^i, cm_2^i, ..., cm_n^i\}$ is a set of methods in C^i. Each method $cm_j^i \in CM^i$ contains:

- $I_j^i = (IP_1, IP_2, ..., IP_r)$,
- $O_j^i = (OP_1, OP_2, ..., OP_s)$, and
- $S_j^i = (S_{j1}^i, S_{j2}^i, ..., S_{jt}^i)$. ■

where I_j^i and O_j^i are the set of input and output parameters, respectively and S_j^i specifies the executable statments of a class method.

A class groups a set of homogeneous objects that are created at object instantiation time. Zapp and Barker's [ZB93c] object model defines a set of uniquely identifiable objects containing structure (attributes) and behavior (methods). We adapt this definition for our model.

Definition 16 *(Object):* An object is an ordered triple, $o = (f, A, M)$, where:

1. f is a unique object identifier,
2. A is the object's structure, defined by attributes such that $\forall a_i, a_j \in A$, $a_i \neq a_j$, and
3. M is the object's behavior, defined by 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 (3) specifies the methods of an object. This dissertation identifies object

f by o^f. The set of methods and the attributes of o^f are unambiguously referenced by $M^f = \{m_1^f, m_2^f, ..., m_n^f\}$ and $A^f = \{a_1^f, a_2^f, ..., a_n^f\}$, respectively. Note that we have used cm_j^i and m_j^i to denote class method and object method, respectively. In the subsequent sections, method and object method are used interchangeably if appropriate.

The values of an object's attributes determine the state of the object. The state of an object can only be modified by transactions. A transaction is a sequence of operations executing on the objects. As mentioned earlier, transactions are either flat or nested. Flat transactions were defined in Section 2.1.1. A nested transaction may be 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.

The nomenclature used for flat transactions is extended to nested transactions. The direct and indirect descendant transactions of a nested transaction, NT_i, are NT_{i1},NT_{i2},...,NT_{in}. When some NT_{ik} 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. Thus, the operation set of NT_i is $OS_i = \cup_k \{\tau_{ik}\}$, where $\tau_{ik} \in \{read, write, pc, NT_{ik}\}$.

One feature of nested transactions is that it is possible to execute subtransactions concurrently. Two subtransactions can be executed concurrently if there is no dependency relation [GB95] in their internal semantics; therefore, freedom from conflict can be verified. Zapp and Barker [ZB93c] define a boolean function, *depends*, which takes two operations *within* a transaction, at least one being a subtransaction invocation and returns true if there is a dependency relation that requires the transactions be ordered.

Definition 17 *(Nested Transaction):* A nested transaction NT_i is a partial order $NT_i = (\Omega_i, \prec_i)$ where:

1. $\Omega_i = OS_i \cup \{N_i\}$,
2. (a) for any two τ_{ip},$\tau_{iq} \in OS_i$, if $\tau_{ip} = w(x)$ and $\tau_{iq} \in (r(x), w(x))$ for some x, $\tau_{ip} \prec_i \tau_{iq}$ or $\tau_{iq} \prec_i \tau_{ip}$,

(b) for any two τ_{ip},$\tau_{iq} \in OS_i$ if $\tau_{ip} = NT_{ip}$ and *depends*(τ_{ip},τ_{iq}) or *depends*(τ_{iq},τ_{ip}), $\tau_{iq} \prec_i \tau_{ip}$ or $\tau_{ip} \prec_i \tau_{iq}$, respectively,

3. if $\tau_{ip} = pc$, τ_{ip} is unique and $\forall \tau_{iq} \in OS_i$, $p \neq q$, $\tau_{iq} \prec_i \tau_{ip}$,
4. $\forall \tau_{ip} \in OS_i$, where $\tau_{ip} = NT_{ip}$ then $N_{ip} = N_i$, and
5. $\forall \tau_{ip} \in OS_i$, $\tau_{ip} \prec_i N_i$. ■

Point (2a) orders the conflicting local operations of the nested transaction. Point (2b) allows the concurrent execution of subtransactions but it orders their conflicting operations. The significance of the *depends* function in this point is that it provides information to allow intra-transaction concurrency. The function *depends*(τ_{ip}, τ_{iq}) returns true if internal semantics of operation τ_{ip} depends on operation τ_{iq}. Detailed implementation appears later in this chapter. Point (3) indicates that all operations of a nested transaction must occur before its pre-commit operation. Point (4) ensures that the termination conditions of all subtransactions invoked by the nested transaction are the same as the termination condition of the nested transaction itself. Point (5) places all the operations of a nested transaction before its termination operation.

3.2 Versionable Objects

We now extend the model of Zapp and Barker [ZB93c] to support versionable objects. Objects are versionable in that several versions may be derived from an object. A version of an object is formally defined as follows:

Definition 18 *(Version)*: A version of an object $v = (f, c, A, M)$, where:

1. f is the unique object identifier of v,
2. c is the unique version identifier of v,
3. A is the object's structure, defined by identifiable attributes such that $\forall a_i, a_j, \in A$, $a_i \neq a_j$, and

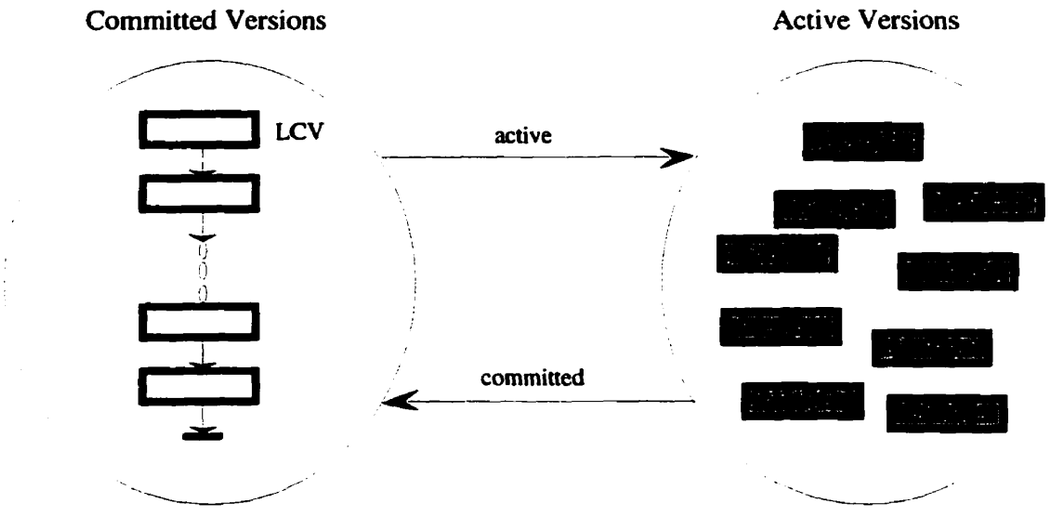


Figure 3.1: An abstract view of active and committed versions of an object

4. M is the object's behavior, defined by identifiable methods such that $\forall m_i, m_j, \in M$, $m_i \neq m_j$. ■

Point (1) identifies the object from which v has been derived. Point (2) distinguishes the versions of an object from each other. Points (3) and (4) are unchanged from Definition 16. All versions of an object must have the same object identifier and methods but we farther annotate the identifier to indicate the version identification.

Versions of an object are either committed or active. Committed versions are sequenced according to some correctness criteria so that the most recent correct version is stored at the head of the sequence and is called the *last committed version* (LCV). An active version of an object begins as a copy of the last committed version which can then be manipulated independent of all other such versions. When an active version attempts to commit, the correctness specification is used to determine if and where in the committed version sequence the active version can be inserted.

Figure 3.1 shows the active versions and the committed versions of an object o^f. When an active version of o^f is created, its state may be modified extensively for some period.

Eventually, the modified active version may commit and become a new committed version if its state is consistent with the states of other committed versions of o^f. Otherwise, the active version may be modified again and if it still cannot be committed, it is disposed. Committed versions are linked into a *version-chain*. A new committed version is added in an appropriate position in the version-chain that is identified by the correctness criterion. Once the new committed version is inserted in the version-chain, changes may need to be propagated to the last committed version and all intervening committed versions. Since the size of the version-chain is limited; periodically older committed versions are removed from the version-chain and are archived. The version-chain of an object effectively captures the evolution of the object (historical information) through time.

We adopt the notational shorthand where v^{fc} identifies active version c of object o^f. An arbitrary data item in v^{fc} is unambiguously denoted x^{fc}. Notations to represent committed versions will be introduced appropriately later in this chapter.

3.3 Transaction Model

Users submit transactions that invoke a set of object methods. Transactions submitted by a user are atomic so the underlying system must ensure that the nested method invocation produce atomic results too. Users submit methods that may subsequently invoke others. 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 group 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 are *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 eventually converted to version transactions. Version transactions are created by the system and operate on active versions of the relevant objects.

3.3.1 User Transactions

The nomenclature used for nested transactions is also used for the definition of user and version transactions. User transaction i is denoted UT_i. Operation τ_{ik} of user transaction UT_i is an invocation of a subtransaction denoted by T_{ik}^f. The subtransaction T_{ik}^f refers to subtransaction k of UT_i operating on active version 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 19 *(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 $\tau_{ip}, \tau_{iq} \in OS_i$, if *depends*$(\tau_{ip}, \tau_{iq})$ 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 $\tau_{ip} = T_{ip}^f$ then $N_{ip} = N_i$, and
4. $\forall \tau_{ip} \in OS_i$, $\tau_{ip} \prec_i N_i$. ■

The points in this definition are directly reflective of those found earlier in Definitions 17 and 18.

3.3.2 Version Transactions

Additional notation is required. The k^{th} step of a version transaction of UT_i, executing on v^{fi} is denoted VT_{ik}^f. The version transaction VT_{ik}^f is created when operation τ_{ik} of UT_i invokes a method of o^f. This nomenclature may be more easily understood by noting that superscripts represent object identifiers while subscripts identify transactions and their operations.

Recall from Section 3.1 that $NT_{ik1}, NT_{ik2}, ...NT_{ikn}$ represent the subtransactions of nested transaction NT_{ik}. This is extended for version transactions so a descendant of VT_{ik}^f

is denoted VT^e_{ikp} that represents subtransaction p of VT^f_{ik} executes on v^{ei}. The set of operations for VT^f_{ik} is $OS_{ik} = \cup_p\{\tau_{ikp}\}$, where $\tau_{ikp} \in \{read, write, pc, VT^e_{ikp}\}$. Two version transactions may execute on a common active version of an object (i.e.: method executions from the same user transaction share a single active version of a particular object).

Definition 20 *(Version Transaction):* A version transaction VT^f_{ik} is a partial order $VT^f_{ik} = (\Omega^f_{ik}, \prec^f_{ik})$, where:

1. $\Omega^f_{ik} = OS_{ik} \cup \{N_{ik}\}$,
2. (a) for any two τ_{ikp},$\tau_{ikq} \in OS_{ik}$, if $\tau_{ikp} = w(x^{fi})$ and $\tau_{ikq} \in (w(x^{fi}), r(x^{fi}))$, for any x^{fi}, $\tau_{ikp} \prec^f_{ik} \tau_{ikq}$ or $\tau_{ikq} \prec^f_{ik} \tau_{ikp}$,
   (b) for any two τ_{ikp},$\tau_{ikq} \in OS_{ik}$ if $\tau_{ikp} = VT^e_{ikp}$ and $depends(\tau_{ikp},\tau_{ikq})$ or $depends(\tau_{ikq},\tau_{ikp})$, $\tau_{ikq} \prec^f_{ik} \tau_{ikp}$ or $\tau_{ikp} \prec^f_{ik} \tau_{ikq}$, respectively,
3. if $\tau_{ikp} = pc$, then τ_{ikp} is unique and $\forall\tau_{ikq} \in OS_{ik}$, $p \neq q$ $\tau_{ikq} \prec^f_{ik} \tau_{ikp}$,
4. $\forall\tau_{ikp} \in OS_{ik}$, where $\tau_{ikp} = VT^e_{ikp}$ then $N_{ikp} = N_{ik}$, and
5. $\forall\tau_{ikp} \in OS_{ik}$, $\tau_{ikp} \prec^f_{ik} N_{ik}$. ■

The significance of the Points (1) (3–5) are similar to that for nested transactions. The only significant difference between nested transactions and version transactions is that the latter can only access the data items of its active version of an object. Point (2b) orders the conflicting operations of two subtransactions of a version transaction which are invoked on the *same active version*.

3.4 Serializability

Traditional databases use conflict or view serializability correctness criteria [BGH87]. This section begins by introducing a new correctness criterion called *value-serializability*. Value serializability relaxes the restrictive properties of conflict serializability. Value serializability is then extended to 1*value-serializability* which is the correctness criterion developed for a multiversion database environment.

3.4.1 Value Serializability

Before discussing the specification, several notational elements need to be provided and an extension to the traditional definition of a "history" [BGH87] 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 [BGH87]. Further, a read operation by user transaction UT_i in history H is represented as $r_i(x, v)$ and a write operation as $w_i(x, v)$ where v is the value read or written by UT_i. A history is now defined as follows:

Definition 21 *(History):* A complete history over a set of user transactions $\mathcal{T}=\{UT_1, UT_2, ...UT_n\}$ is a partial order $(\sum_H, \prec_H)$ where:

1. $\sum_H = \bigcup_i \{\bigcup_k^f \Omega_{ik}^f\}$, $(1< i <n)$
2. $\prec_H \supseteq \bigcup_i \{\bigcup_k^f \prec_{ik}^f\}$, 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 $\tau_{jq} \in (r_j(x, v), w_j(x, v))$, 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 20). 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 overlapping values.

Conflict serializability states that a conflict occurs if two operations access the same data and at least one is a write operation. Our definition of conflict is called *value-conlict*. Before formally defining this concept we discuss the differences between traditional and value-conflicts. Value-conflict relaxes the confict definition in two ways. First, conflicting operations which read/write the same values may not necessarily value-conflict. For example, two write operations that write the same value into a data item x can be executed in any order. Further, any read and write operations which utilize the same data value will not value-conflict if the read proceeds the write operation in the history. For example, consider the following:

$$H_p = \{w_1(x,5), w_2(x,10), r_3(x,10), w_4(x,10)\}$$

In H_p, $r_3(x,10)$ and $w_4(x,10)$ conflict, but they do not value-conflict because their ordering does not effect the value of x. However, a read and a write operation value-conflict if the read operation reads-from (see definition 7) the write operation. For example, T_3 reads from T_2 in H_p. Reordering the execution of $w_2(x,10)$ and $r_3(x,10)$ requires T_3 reads from T_1 which implies it would incorrectly use the value of 5.

Second, some conflicting operations that read/write distinct values into a data item x may not value-conflict. For example, consider the following:

$$H = \{w_1(x,1), r_2(x,1), w_3(x,3), w_4(x,4), w_5(x,1), w_2(x,2), w_6(x,6), c_1, c_2, c_3, c_4, c_5, c_6\}$$

Consider,

$$A = \{w_1(x,1), r_2(x,1), w_3(x,3), w_4(x,4), w_5(x,1)\}$$

a projection of H. Since A does not contain any write operation of UT_2, it makes no difference if $r_2(x,1)$ reads from $w_1(x,1)$ or from $w_5(x,1)$. This implies that $r_2(x,1)$ may be executed in any order with respect to $w_3(x,3)$ or $w_4(x,4)$ as long as it reads either from $w_1(x,1)$ or from $w_5(x,1)$. Thus, under these conditions, $r_2(x,1)$ does not value-conflict with $w_3(x,3)$ or with $w_4(x,4)$.

Projection A is called a *range* for $r_2(x,1)$ and is defined as follows:

Definition 22 *(Range):* Given three transactions $UT_i, UT_p, UT_q \in H$, projection A in H is a range for operation $r_i(x,v_i)$ if:

1. $w_p(x,v_p)$ 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
2. A contains no write operation by UT_i on any data item. ■

The rational for Point (3) is explained as follows. Consider the following history:

$$H_1 = \{r_2(x,3), w_1(x,1), r_2(x,1), w_3(x,3), w_2(y,2), w_4(x,4), w_5(x,1), c_1, c_2, c_3, c_4, c_5\}$$

and projection

$$A_1 = \{w_1(x,1), r_2(x,1), w_3(x,3), w_2(y,2), w_4(x,4), w_5(x,1)\}$$

of H_1. For operation $r_2(x,1)$ in H_1, A_1 satisfies the first condition of Definition 22 but not the second condition. Suppose the value of y for $w_2(y,2)$ in H_1 is calculated based on the value of x read by UT_2 (ex: $y = x+1$). Note that in H_1, $w_2(y,2)$ uses the value of x read by $r_2(x,1)$. However, this will not be the case if $r_2(x,1)$ reads from $w_5(x,1)$ instead. Consider

$$H_1' = \{r_2(x,3), w_1(x,1), w_3(x,3), w_2(y,4), w_4(x,4), w_5(x,1), r_2(x,1), c_1, c_2, c_3, c_4, c_5\}$$

Since $r_2(x,1)$ reads from $w_5(x,1)$, the value of y in $w_2(y,4)$ is calculated based on the value of x read by $r_2(x,3)$. Clearly, H_1 and H_1' do not have the same set of operations so they can not be equivalent. This implies that in H_1, the execution order of $r_2(x,1)$ with respect to $w_3(x,3)$ and $w_4(x,4)$ is important because A_1 contains a write operation of UT_2. Therefore, $r_2(x,1)$ value-conflicts with $w_3(x,3)$ and $w_4(x,4)$ and A_1 is not considered as a range for $r_2(x,1)$.

The notion of value-conflict can now be formally defined as follows:

Definition 23 *(Value-conflict):* Two operations τ_{ip} and τ_{jq} in a history H value-conflict if:

- τ_{ip}=$w_i(x,v)$, τ_{jq}=$w_j(x,u)$, and $u \neq v$,
- τ_{ip}=$w_i(x,v)$, τ_{jq}=$r_j(x,u)$, $\tau_{ip} \rightarrow \tau_{jq}$[1], and τ_{jq} reads from τ_{ip} in H, or
- τ_{ip}=$w_i(x,v)$, τ_{jq}=$r_j(x,u)$, $u \neq v$ and τ_{ip} is not in any range of τ_{jq} in H. ■

This gives rise to the concept of the equivalence between two histories.

Definition 24 *(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. ■

[1] $p \rightarrow q$ means that operation p proceeds the operation q in the history

A history is serializable if it is equivalent to some *serial* history [BGH87]. Recall that in a serial history (Definition 3) operations of distinct transactions do not interleave and transactions are executed in a total order.

Definition 25 *(Value Serializable):* A history is *value*-serializable if it is value-conflict equivalent to some serial history. ■

The Value Serializability Theorem

Suppose history H is defined over a set of user transactions $\mathcal{T} = \{UT_1, UT_2, ...UT_n\}$. 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 $v_i \in V$ represents a transaction $UT_i \in \mathcal{T}$, 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 3.4.1 *(Value Serializability Theorem):* A history H is value-conflict serializable iff $VSG(H)$ is acyclic.

Proof (sketch):

(if): Suppose H is a history over $\mathcal{T} = \{UT_1, UT_2, ...UT_n\}$ 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 $VSG(H)$. 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. ■

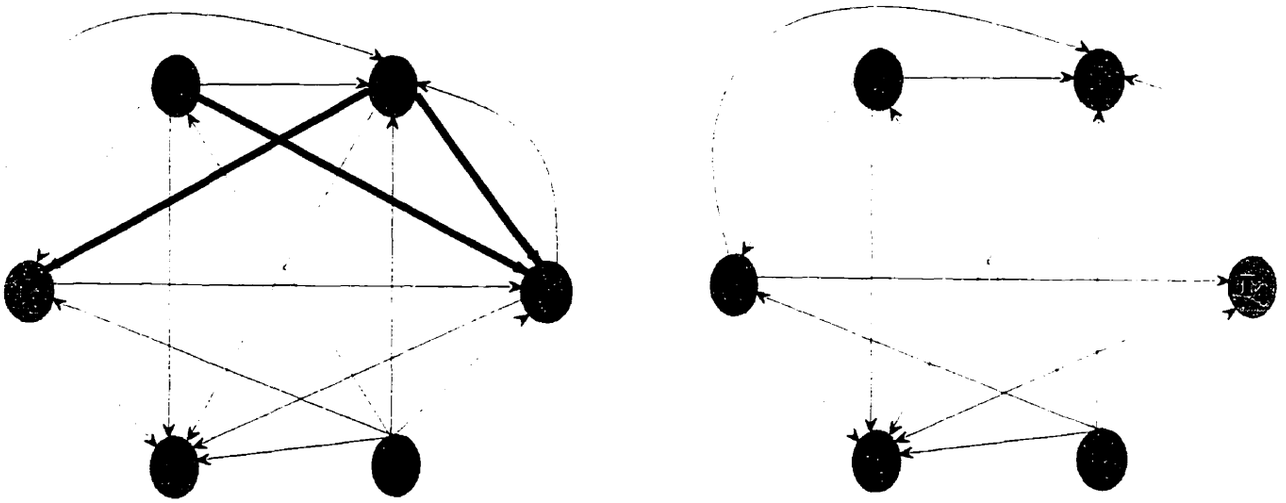
An Example

Consider the following history:

$$H = \{w_6(x,5), w_1(x,1), r_2(x,1), w_3(x,3), w_4(x,1), w_2(x,10), r_5(x,10), w_5(x,9), c_6, c_1, c_3, c_4, c_2, c_5\}$$

Figure 3.2 shows serialization graph and the value-serialization graph for H. The edges in the serialization graph refer to the conflicting operations in H and the ones in the value serialization graph correspond to the value-conflicting operations. Note that the three bold edges in Figure 3.2A are not included in Figure 3.2B. The edge $T_2 \rightarrow T_4$ reflects the operations $r_2(x,1)$ and $w_4(x,1)$ in H because $r_2(x,1)$ conflicts with $w_4(x,1)$ in H. However, $r_2(x,1)$ and $w_4(x,1)$ do not value-conflict because they read and write the same value and $r_2(x,1)$ happens before $w_1(x,1)$ in H in which changing their order does not reflect the value read by $r_2(x,1)$. Thus the edge $T_2 \rightarrow T_4$ is not added to the value serialization graph. Similarly, it is not necessary to add the edge $T_1 \rightarrow T_4$ to the value-serialization graph because the operations $w_1(x,1)$ and $w_4(x,1)$ write the same value into value x and do not value-conflict. The edge $T_2 \rightarrow T_3$ which is associated with the operations $r_2(x,1)$ and $w_3(x,3)$ is also not added to the value-serialization graph because $w_3(x,3)$ occurs in the following range of $r_2(x,1)$.

$$H = \{w_1(x,1), r_2(x,1), w_3(x,3), w_4(x,1)\}$$



A: Serialization Graph B: Value–serialization Graph

Figure 3.2: serialization graphs for conflict and value-conflict serializabilities

Figure 3.2A contains a cycle because it considers all of the conflicting operations; whereas, Figure 3.2B is acyclic because edges only correspond to the value-conflicting operations. A topological sort of vertices (transactions) in Figure 3.2B produces the following serial history H_s which is value-conflict equivalent to H.

$$H_s = \{w_6(x,5), c_6, w_1(x,1), c_1, w_3(x,3), c_3, w_4(x,1), c_4, r_2(x,1), w_2(x,10), c_2, r_5(x,10), w_5(x,9), c_5\}$$

Relationship with other Correctness Criteria

Since the edges of the value serialization graph for history H are determined based on the value conflicting operations in history H, there is only one way to construct the value serialization graph for H. Selection of every two operations in a history takes $O(n^2)$ to decide if they do value-conflict. It also takes $O(n^2)$ to check if a projection of a history is a range for an operation. Furthermore, once a serialization graph for a history is constructed, a cycle in the value serialization graph can be detected in polynomial time. Thus the decision problem that determines if a history is value serializable can be solved in polynomial time.

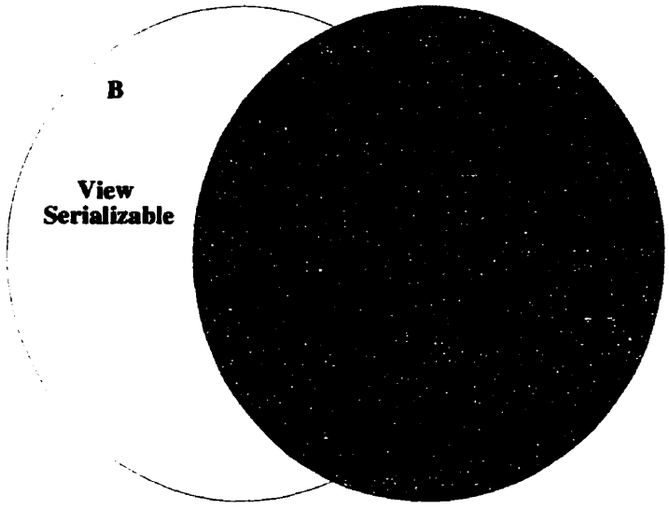


Figure 3.3: Relationship between value, view, and conflict serializabilities

The following compares value serializability with view and conflict serializabilities in terms of scheduling and the cost of implementation.

Figure 3.3 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 $VSG(H_3)$ 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].

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

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 environments, 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 is made to the system kernel in privileged mode which 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) [Moh90].

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 minimally 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, environments 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 $2PVL$.

3.4.2 1Value Serializability

This section extends value serializability and develops a suitable correctness criterion called *1value serializability* for a multiversion environment. Value serializability and 1value serializability are analogous to conflict serializability and 1-copy serializability developed by Bernstein *et al.* [BGH87].

Since traditional databases keep a single version of each data item in the database, this dissertation refers to a traditional history as single version history (SV history). Similarly a serial history (Definition 3) is called a single version serial history (SV serial history). The specifications and notational elements introduced for SV history in the previous section are adopted for the definition of multiversion history. The only extension is that a read and a write operation on a data item x is denoted by $r_i(x_j, u)$ and $w_i(x_i, v)$, respectively. A specific version of x accessed for a read or a write operation are annotated with the identifier of the user transaction which has produced that version of x.

A multiversion history is formally defined as follows:

Definition 26 *(Multiversion History):* A complete multiversion history (MV history) over a set of user transactions $UT_1, UT_2, ...UT_n$ is a partial order $H = (\sum_H, \prec_H)$ where:

1. $\sum_H = \bigcup_i \{\bigcup_k^f \Omega_{ik}^f\}$ $(1 < i < n)$,
2. $\prec_H \supseteq \bigcup_i \{\bigcup_k^f \prec_{ik}^f\}$,
3. for every two operations τ_{ip} and $\tau_{jq} \in \sum_H$, if $\tau_{ip} = w_i(x_i, u)$ and $\tau_{jq} = r_j(x_i, v)$ $\tau_{ip} \prec_H \tau_{jq}$ and $u = v$, and
4. if $\tau_{jp} = r_j(x_i, u) \in \sum_H$ and $c_j \in \sum_H$, then $c_i \prec_H c_j$. ■

Point (1) and (2) reflect the ones in Definition 21. Point (3) indicates that a transaction may not read a version until it has been produced. Point (4) ensures that before a transaction

commits, all transactions that produced versions it read must have already committed. This guarantees recoverability.

A multiversion serial history is defined as:

Definition 27 *(MV Serial):* A MV history H is MV serial, iff:

1. $(\exists p \in UT_i, \exists q \in UT_j$, where $p \prec q) \Longrightarrow (\forall r \in UT_i, \forall s \in UT_j, r \prec s)$, and
2. a read operation of UT_i on data item x can read any previously created version of x. ■

Note that in SV serial histories (Definition 3) only one version of a data item x, the last committed version, is available for transactions to access. However, this restriction is relaxed in MV serial histories.

Definition 28 *(Correspond):* A serial MV history H_1 corresponds to a SV serial history H_2 if:

1. H_1 and H_2 occur over the same set of transactions and there is one to one mapping between operations in H_1 and H_2, and
2. if $c_i \prec c_j$ in H_1, $c_i \prec c_j$ in H_2. ■

Point (1) indicates that for every read/write operation on a version of a data item x in a MV serial history there is a corresponding read/write operation on data item x in its corresponding SV serial history. Point (2) ensures that the commit order of the transactions in MV serial history is the same as the commit order of the transactions in its corresponding SV serial history.

Some MV serial histories may not behave as their corresponding SV serial histories. For example consider the following two histories:

$$H_1 = \{r_1(x_0, 3), w_1(x_1, 5), c_1, w_2(x_2, 10), c_2, r_3(x_1, 5), w_3(x_3, 20), c_3\}$$
$$H_2 = \{r_1(x, 3), w_1(x, 5), c_1, w_2(x, 10), c_2, r_3(x, 10), w_3(x, 40), c_3\}$$

H_1 is a MV serial history and H_2 is its corresponding SV serial history. Both H_1 and H_2 contain the same set of transactions UT_1, UT_2, UT_3, and execute the transactions in the same order of UT_1, UT_2, and UT_3. H_1 and H_2 do not behave similarly because in H_1, UT_3 reads the version of x produced by UT_1 and in H_2, UT_3 reads the version of x produced by UT_2.

Bernstein *et al.* [BGH87] introduce a subset of MV serial histories called 1-*serial* histories that behave the same as their corresponding SV serial histories.

Definition 29 *(1-Serial):* A MV history H is 1-serial, if it is *MV* serial and for all i, j, if UT_i reads x_j (the version of x produced by UT_j), then either $i{=}j$ or UT_j is the last transaction preceding UT_i that writes into any version of x. ■

Since our definition of MV history takes the values read or written by each operation into account, the above subset of MV serial histories, 1-serial histories, can be extended to include a wider range of MV serial histories.

Definition 30 *(1value Serial):* A MV history H is 1value serial, if it is *MV* serial and if UT_i reads x_k written by operation $w_k(x_k, u)$ of UT_k, then either:

- $i{=}k$,
- UT_k is the last transaction proceeding UT_i that writes into a version of x, or
- there exists a UT_j which is the last transaction preceding UT_i that writes into a version of x and $u{=}v$. ■

For example, consider the following MV history:

$$H_2 = \{r_1(x_0, 3), w_1(x_1, 4), c_1, r_2(x_1, 4), w_2(x_2, 3), c_2, r_3(x_0, 3), w_3(x_3, 6), c_3\}$$

UT_3 reads x_0 from UT_0 rather than x_2 from UT_2. Thus H_2 is not 1-serial. However, H_2 is 1value serial because whether UT_3 reads x_0 from UT_0 or x_2 from UT_2, it receives the same value and subsequent operations of UT_3 that depend on this read in H_2 are not effected.

Proposition 3.4.1 Two MV histories are equivalent if they have the same set of operations.
Proof: (see Bernstein *et al.* [BGH87] page 148). ■

Now a 1value serializable history is defined as follows:

Definition 31 *(1value Serializable):* A MV history is *1value serializable* if it is equivalent to a 1value serial history. ■

The 1value Serializability Theory

The following definition is required to discuss 1value serialization theory.

Definition 32 *(Version Order):* Given a MV history H and a data item x, a version order, $\ll$, for x in H is a total order of versions of x in H. A version order for H is the union of the version orders for all data items. ■

For example, the version order for

$$H_3 = \{w_1(x_1, 2), w_1(y_1, 3), c_1, w_2(x_2, 3), c_2, w_3(y_3, 3), c_3, w_4(x_4, 4), c_4, w_5(y_5, 5), c_5\}$$

is $x_1 \ll x_2 \ll x_4$, and $y_1 \ll y_3 \ll y_5$.

Given a MV history H and a version order $\ll$, the multiversion value serialization graph for H and $\ll$, $MVVSG(H, \ll)$ is (V, E) where a vertex in V represents a transaction $UT_i \in H$, and an edge in E from vertex UT_i to UT_j is either a *reads-from edge* or a *version order edge.* A reads-from edge is added to $MVVSG(H, \ll)$ from UT_i to UT_j if UT_j reads a version of a data item x created by UT_i. A version order edge is added as follows. For every two value conflicting operations $r_k(x_j, v)$ and $w_i(x_i, u)$ in H, (i.e. $u \neq v$ and $w_i(x_i, u)$ is not in the range of $r_k(x_j, v)$) if $x_i \ll x_j$ then include an edge from UT_i to UT_j ($UT_i \rightarrow UT_j$); otherwise, include $UT_k \rightarrow UT_i$. Note that in multiversion environments two write operations do not conflict because each write creates a different version of the data item.

The following shows the significance of version order edges in $MVVSG(H, \ll)$. Let $RFVSG(H)$ (stands for reads-from value serialization graph) be a subgraph of $MVVSG(H, \ll$

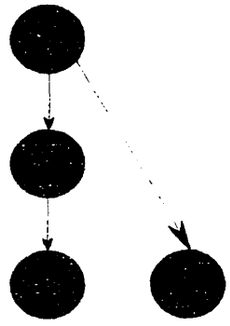


Figure 3.4: Reads-from edges of a serialization graph for a MV history

) which only includes the reads-from edges. Given that $RFVSG(H)$ is acyclic, a MV serial history H_s obtained from $RFVSG(H)$ by topologically sorting may not necessarily be 1value serial. For example, consider the following history:

$$H_4 = \{w_0(x_0,1), r_3(x_0,1), r_1(x_0,1), w_1(x_1,2), r_2(x_1,2), w_2(x_2,4), w_3(x_3,5), c_1, c_2, c_3\}$$

$RFVSG(H_4)$ is shown in Figure 3.4. A serial MV history H_{s4} obtained from topological sorting is:

$$H_{s4} = \{w_0(x_0,1), c_0, r_1(x_0,1), w_1(x_1,2), c_1, r_2(x_1,2), w_2(x_2,4), c_2, r_3(x_0,1), w_3(x_3,5), c_3\}$$

H_{s4} is not 1value serial because UT_3 does not read a proper version of x.

The purpose of version order edges is to prevent the above problem. For every two operations $r_k(x_j,u)$ and $w_i(x_i,v)$ in H, the version order edges can force $w_i(x_i,v)$ to either precede $w_j(x_j,w)$ or to follow $r_k(x_j,u)$ in H_s if necessary (when $w_i(x_i,v)$ does not value-conflict with $r_k(x_j,u)$ the order is not important). Reads-from edges together with version-order edges must find a 1value serial history for H, as long as $MVVSG(H,\ll)$ is acyclic. This leads us to the following theorem.

Theorem 3.4.2 *(1value serializability Theorem):* A MV history H is 1value serializable iff there exists a version order $\ll$ such that $MVVSG(H,\ll)$ is acyclic.

Proof:

(if): Let H_s be a MV serial history $UT_{i1}, UT_{i2}, ..., UT_{in}$, where $UT_{i1}, UT_{i2}, ..., UT_{in}$ is a

topological sort of $MVVSG(H, \ll)$. Since H_s has the same operations as H, by Proposition 3.4.1 H_s is equivalent to H. Now we need to show that H_s is 1value serial. Consider any reads-from relationship in H_s, say UT_k reads the version of x, x_j, from UT_j ($r_k(x_j, u)$), $k \neq j$. Let $w_i(x_i, v)$ ($i \neq j$ and $j \neq k$) be any other write operation on x in H. If $w_i(x_i, v)$ value-conflicts with $r_k(x_j, u)$, and $x_i \ll x_j$, $MVVSG(H, \ll)$ includes the version order edge $UT_i \rightarrow UT_j$ which forces UT_j to follow UT_i in H_s; otherwise, if $x_j \ll x_i$, $MVVSG(H, \ll)$ includes the version order edge $UT_k \rightarrow UT_i$, which forces UT_k to precede UT_i in H_s. Therefore, no transaction that writes a version of x falls in between UT_j and UT_k in H_s. Thus H_s is 1value serial.

(only if): Since H is 1value serializable, there exists a 1value serial history H_s that is equivalent to H. For a given $\ll$, let $VOVSG(H, \ll)$ (stands for version order value serialization graph) be a subgraph of $MVVSG(H, \ll)$ containing only version order edges. Version order edges depend only on the operations in H and $\ll$; they do not depend on the order of operations in H. Thus, since H and H_s have the same operations, $VOVSG(H, \ll) = VOVSG(H_s, \ll)$ for all version orders $\ll$.

Let $RFVSG(H_s)$ be a subgraph of $MVVSG(H_s, \ll)$ containing only the reads-from edges. All edges in $RFVSG(H_s)$ go in one direction (for convenience we call it "left-to-right"); that is if there is an edge $UT_i \rightarrow UT_j$ in $RFVSG(H_s)$, then UT_i precedes UT_j in H_s. Define $\ll$ as follows: $x_i \ll x_j$ only if UT_i precedes UT_j in H_s. All edges in $VOVSG(H_s, \ll)$ are also left-to-right. Therefore, all edges in $MVVSG(H_s, \ll) = RFVSG(H_s) \cup VOVSG(H_s, \ll)$ are also left-to-right. This implies $MVVSG(H_s, \ll)$ is acyclic. Since H and H_s are equivalent (Proposition 3.4.1), $MVVSG(H, \ll) = MVVSG(H_s, \ll)$. Since $MVVSG(H_s, \ll)$ is acyclic, so is $MVVSG(H, \ll)$. ■

The following provides an example to show that 1value serializability controls serialization order of the transactions both at each object (intra-object serializability) and in the entire system (inter-object serializability).

Suppose UT_1 and UT_2 are two user transactions concurrently accessing attributes x and y in o^p and H_p is the MV history over UT_1 and UT_2 in o^p. If UT_1 reads the version of x

produced by UT_2 and UT_2 reads the version of y produced by UT_1, $MVVSG(H_p, \ll)$ will contain a cycle and H_p is not 1value serializable. Now suppose UT_1 and UT_2 do not contain any read operation but each write a version of x and y. As long as no other transaction UT_k reads a version of x and/or a version of y produced by either UT_1 or UT_2, UT_1 and UT_2 can be serialized in any order. Otherwise, the version order edges can enforce a serialization order between UT_1 and UT_2. For example, consider the following MV history for object o^p:

$$H_p = \{w_1(x_1, 2), w_2(x_2, 3), r_k(x_2, 3), w_2(y_2, 4), w_1(y_1, 5), r_k(y_1, 5), c_1, c_2, c_k\}$$

The two reads-from edges in $MVVSG(H_p, \ll)$, $UT_1 \rightarrow UT_k$ and $UT_2 \rightarrow UT_k$, do not create a cycle. When the version order edges, $UT_1 \rightarrow UT_2$ and $UT_2 \rightarrow UT_1$, are added, $MVVSG(H_p, \ll)$ will contain a cycle to indicate that H_p is not 1value serializable. The edge $UT_1 \rightarrow UT_2$ is added because $w_1(x_1, 2)$ value-conflicts with $r_k(x_2, 3)$ and $x_1 \ll x_2$. Similarly, the edge $UT_2 \rightarrow UT_1$ is added because $w_2(y_2, 4)$ value-conflicts with $r_k(y_1, 5)$ and $y_2 \ll y_1$.

Similar argument to the above can be given for inter-object serializability. Suppose x is an attribute in o^p and y is an attributes in o^q, and UT_1 and UT_2 access o^p and o^q concurrently. Suppose, UT_1 commits before UT_2 at o^p and UT_2 commits before UT_1 at o^q. Now if a new transaction UT_k access o^p and o^q, it may read x_2 produced by UT_2 in o^p and y_1 produced by UT_1 in o^q. The MV histories of the transactions at o^p and o^q are as follows:

$$H_p = \{w_1(x_1, 1), c_1, w_2(x_2, 2), c_2, r_k(x_2, 2), c_k\}$$
$$H_q = \{w_2(y_2, 2), c_2, w_1(y_1, 1), c_1, r_k(y_1, 1), c_k\}$$

The $MVVSG(H_p, \ll)$ contains two edges. One is a reads-from edge $UT_2 \rightarrow UT_k$ and the other is the version order edge $UT_1 \rightarrow UT_2$. $UT_2 \rightarrow UT_k$ was added because UT_k reads from UT_2 and $UT_1 \rightarrow UT_2$ was added because $x_1 \ll x_2$ and $w_1(x_1, 1)$ value-conflicts with $r_k(x_2, 2)$. The $MVVSG(H_q, \ll)$ also contains two edges. One is a reads-from edge $UT_1 \rightarrow UT_k$ and the other is the version order edge $UT_2 \rightarrow UT_1$. $UT_1 \rightarrow UT_k$ was added

because UT_k reads from UT_1 and $UT_2 \rightarrow UT_1$ was added because $y_2 \ll y_1$ and $w_2(y_2, 2)$ value-conflicts with $r_k(y_1, 1)$. Although neither $MVVSG(H_p, \ll)$ nor $MVVSG(H_q, \ll)$ contain cycle, the union of both graphs has a cycle and the system is not inter-object 1value serializable. Thus by combining the multiversion value serialization graphs of all object histories, inter-object serializability is controlled.

3.5 Data Dependency

This section presents definitions related to data dependency and concurrency control. Required static analysis information used to enhance concurrency is also addressed. Some concepts may also be used for reconciliation.

Executable statements of a method are divided into two categories: *local steps* and *message steps* [HH91]. The *local steps*, $LS(m_j^i)$, of a method are those which operate on object attributes and the local variables of the method. The *message steps*, $MS(m_j^i)$, of a method correspond to method invocations. The set of all steps in a method is denoted $STEPS(m_j^i) = MS(m_j^i) \cup LS(m_j^i)$.

Method steps can be related by a partial order to allow intra-step serializability. Imposing execution order on some steps such as within the ones that access separate attribute sets might be unnecessary. However, other method steps must be executed in a particular order because the execution of one depends on the execution of the others. Data dependency defines access by defining *accessors* and *mutator* steps. An accessor step is a local step which reads an attribute value or a message step which uses an attribute value as an input parameter. A mutator step is a local step which assigns a value to an attribute or a message step where the attribute is one of its output parameters [Gra94].

Basic forms of data dependencies that exist between accessor and/or mutator steps of a method are *true dependence*, *anti dependence*, and *output dependence*. True dependence between a mutator step S_1 and an accessor step S_2 in a method ($S_1 \delta S_2$) occurs if S_1 and S_2 access the same attribute and S_1 precedes S_2. Anti dependence between an accessor

step S_1 and a mutator step S_2 in a method ($S_1\bar{\delta}S_2$) occurs if S_1 and S_2 operate on the same attribute and S_1 precedes S_2. Output dependence between two mutator steps S_1 and S_2 in a method ($S_1\delta^\circ S_2$) occurs if S_1 and S_2 modify the same attribute and S_1 precedes S_2 [Wol89].

The following two definitions generalizes the above dependence relations.

Definition 33 (*Arbitrary direct dependence*:) Arbitrary direct dependence between two steps S_1 and S_2 in a method ($S_1\delta^? S_2$) occurs if there is a true, anti, or output dependence between the steps. ■

Definition 34 (*Arbitrary indirect dependence*:) Arbitrary indirect dependence is the transitive closure of the arbitrary direct dependence relation. Two steps S_1 and S_2 in a method are arbitrary indirect dependent ($S_1\delta^* S_2$) if there is a chain of arbitrary direct dependencies between them (i.e.: $S_1\delta^* S_2 \equiv S_1\delta^? S_{i1}\delta^? S_{i2}\delta^?, ..., \delta^? S_{in}\delta^? S_2$ for some ($n > 0$)). ■

Now the partial order relation for the steps of a method is formally defined as:

Definition 35 Steps of a method m_j^i are related by a partial order $STEPS(m_j^i, \prec)$ where for every two steps $S_p, S_q \in m_j^i$, $S_p \prec S_q$ if $S_p\delta^? S_q$, or $S_p\delta^* S_q$ in $STEPS(m_j^i, \prec_s)$. ■

3.5.1 Definitions Related to Concurrency Control

Steps which are not related by the partial order $\prec$ (Definition 35) can be executed concurrently. To perform concurrency control, it is necessary to capture and compare the data items read/written in the steps. The *readset* of a step includes the input parameters, the local variables read from the users, and attributes retrieved from the local object. The *writeset* of a step contains the output parameters and the local object attributes which are modified when the step is executed. The readset and the writeset of a step S_{jk}^i in method m_j^i are denoted by $RS(S_{jk}^i)$ and $WS(S_{jk}^i)$, respectively. The information read/written by a method can be captured effectively by taking the union of the readsets and the writesets of

the steps in the method. Construction of readsets and writesets of a method are presented later in this dissertation.

To perform concurrency control, it is also required information be obtained related to inter-object communication. Objects communicate by passing information to each other through the message steps. A message step in a method *directly invokes* the object methods directly specified in the message step. Further, a message step *indirectly invokes* the object methods that are either directly or indirectly invoked by the method it directly invokes. To capture this information, the following definitions are needed.

Definition 36 *(extent)*: The extent of a message step S_{jk}^{i} ($extent(S_{jk}^{i})$) consists of all object methods that may be directly or indirectly invoked by its execution [Gra94]. ■

Definition 37 *(reachableset)*: The reachableset of a message step S_{jk}^{i} ($reachableset(S_{jk}^{i})$) consists of those objects containing one or more methods in the extent of the message step [Gra94]. ■

The above concepts can be also expressed for the entire method. The extent and the reachableset of a method are constructed as follows:

$$extent(m_j^i) = \bigcup_{S_{jk}^i \in MS(m_j^i)} extent(S_{jk}^i)$$

$$reachableset(m_j^i) = \bigcup_{S_{jk}^i \in MS(m_j^i)} reachableset(S_{jk}^i)$$

Concurrent nested transaction execution can occur in several forms. Three levels of potential concurrency are identified: *coarse-grained concurrency*, *medium-grained concurrency*, and *fine-grained concurrency* [Gra94].

1. Coarse-grained concurrency arises due to the availability of multiple concurrent nested transactions issued by different users (user transactions [HH91]).

2. Medium-grained concurrency results from the concurrent execution of subtransactions invoked by a user transaction (concurrent message steps).
3. Fine-grained concurrency occurs between local steps of a method.

Concurrent execution of subtransactions within a single user transaction is *intra-transaction* concurrency and corresponds to medium-grained concurrency. Concurrency between nested transactions issued by different user transactions is *inter-transaction* concurrency and corresponds to coarse-grained concurrency. This dissertation does not address fine-grained concurrency.

3.5.2 Static Information

Recall that our intention is to use static information to enhance concurrency in a multi-version objectbase system. The following sections provides the required static information and their representation. The detailed discussion that demonstrates the derivation of static information is presented in [Gra94] and will not be repeated in this dissertation.

The Required Static Information

The required static information can be divided into three general categories: *control flow information*, *method invocation information*, and *attribute reference information*. The control flow information details what sections of the code may be executed. It also determines the order of executable sections and the necessary conditions to execute each section. Control flow information is required for serialization within a method. Concurrent execution of some sections in a method is correct if the execution order of these sections is based on the partial ordering relation defined over the steps of the method (Definition 35). The control flow information reflects this partial ordering.

Method invocation information illustrates communication among the objects by detailing the calling sequence between the objects. Method invocation information is needed

to capture method's extent and reachable sets. Techniques such as *call graphs* [Ryd79, CCHK90] can be developed to represent communication between the objects.

Attribute reference information details the order in which the data is referenced and their read/write relationship. This information is needed to capture serializability between and within a method. Attribute reference information detects possible conflict between two concurrently executing methods of an object. Further, comparison of the attributes referenced by the steps of a method determine data dependency relation between the steps. The data dependence relation shows what steps of a method are related by the partial order. Steps which do not depend on each other locally can be executed concurrently.

Representation

This section presents graph techniques developed by Graham [Gra94] to represent control flow, method invocation, and attribute reference information. This section discusses how these information concepts relate to concurrency control and reconciliation.

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. The detailed algorithm to derive the basic blocks of a program routine is presented by Aho *et al.* [ASU86]. In brief, the algorithm accepts an encoded form of a program routine called *three address code* [3] and determines the basic block *leaders*. A leader is the first statement of each basic block and is determined by:

1. the first statement in the program,
2. any statement that is the target of a conditional or unconditional goto,
3. any statement that immediately follows a goto or conditional goto statement.

[3]The three address code is an encoded of the program in which all of the complex statements have been decomposed to their simplest form and cannot be decomposed further.

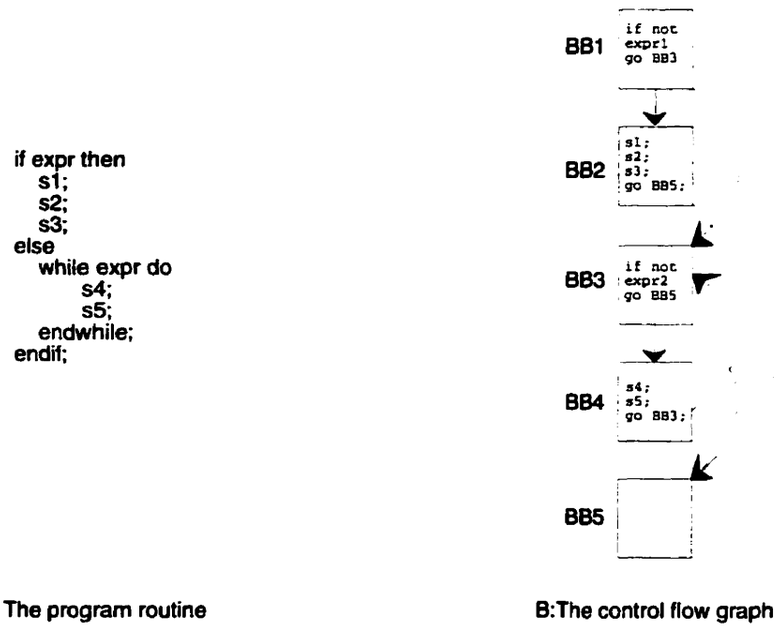


Figure 3.5: Control flow graph of a program segment

For each leader, its basic block consists of the leader and all the statements up to but not including the next leader or the end of the program.

A basic block may contain one or more branch statements which link the basic block to other basic blocks. The relation between basic blocks are captured in a *control flow graph* [ASU86].

Definition 38 *(Control Flow Graph)*: The control flow graph of a method m_j^i is a directed graph $CFG(m_j^i) = (V, E)$ where each vertex $v_x \in V$ represents a basic block x (BB_x) and an edge from v_x to v_y indicates that the control directly passes from BB_x to BB_y in m_j^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 is executed. Figure 3.5 provides an example of program, and its corresponding control flow

graph. Note that BB_1 is linked to BB_2 and BB_3, and BB_2 is linked to BB_5 and so on. Every basic block except the ones which represent the leaf nodes in the control flow graph is linked to one or more other basic blocks.

In a particular execution of a method, only a subset of the basic blocks are visited. This is because, based on the current state of the object, the control (conditional) statements prevent the execution of some basic blocks. For example in Figure 3.5B, if *expre*1 evaluates to true during the execution, BB_2 is visited; otherwise, the control is passed to BB_3. A sequence of basic blocks visited during an execution of a method forms a *control flow path* [ASU86].

Definition 39 *(Control Flow Path)*: Control Flow Path CFP_x through method m_j^i is a sequence of basic blocks $< BB_{k_0}, BB_{k_1}, ..., BB_{k_{n-1}} >$ where BB_{k_0} is the entry node in the control flow graph for the method m_j^i, $BB_{k_{n-1}}$ is an exit node of the graph, and there exists an edge from BB_{k_l} to $BB_{k_{l+1}}$ $(0 < l < n\text{-}1)$ to indicate that the control flows directly from BB_{k_l} to $BB_{k_{l+1}}$. ■

In general, if a method contains n control statements, there are at most 2^n control flow paths. Prior to the execution of a method, it might 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. Note that as the number of n increases, the number of control flow paths grows exponentially. But n is bounded by the size of the method and methods typically have small sizes in our environment. Further, n refers to the number of control flow statements in a method which is always smaller than the size of the method (i.e: not all of the statements in a method are control flow statements). Thus 2^n is manageable so enumerating the control flow paths at compile time is not problematic.

The following table shows all possible control flow paths through the program in Figure 3.5A.

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

In this example, there are only three distinct paths because when $expr1$ is true the result of $expr2$ is irrelevant.

Method Invocation Information

Conventionally, call graphs were used to illustrate the calling relations between the procedures of a program. Informally, a call graph is a directed graph where the vertices represent the routine calls in a program and the edges show how these routines invoke each other. The equivalent of call graph is *class call graph* [Gra94] for an objectbase environment. Class call graph shows if an object (an instance) of a class should communicate with some objects of its own class and/or objects of other classes; but, it does not specify with which objects. To construct the extent and the reachable set of the objects, it is necessary to identify the set of objects in the objectbase which an object o^f communicates. *Object call graphs* [Gra94] captures this information.

Definition 40 *(Object Call Graph):* The object call graph of an object method m_j^i is a directed graph $OCG(m_j^i) = (V, E)$ where:

1. $v_{root} \in V$ is the root of the graph which represents m_j^i,
2. $v_x \in V - \{v_{root}\}$ represents an object method that is either directly or indirectly invoked by m_j^i, and
3. Given that v_x and v_y represent two object methods m_p^x and m_q^y, respectively, an edge $e \in E$ from v_x to v_y indicates that m_p^x directly invokes m_q^y. ■

Construction of an object call graph for a method m_j^i is subject to two conditions. First, the object associated with m_j^i must have been created. Second, objects referenced directly

or indirectly by m_j^i must exist. In contrast to call graphs and class call graphs, object call graphs can be constructed at object instantiation time rather than at compile time. An object may not be instantiated unless it is referenced. However, once an object is instantiated, object call graphs of its methods can be used in subsequent access to that object.

Attribute Reference and Dependence Information

Attribute reference information together with control flow graphs are used to construct the dependence graph which reflects the dependence relation. As mentioned earlier, a method is encoded to a sequence of non-decomposable three address codes [ASU86] which minimizes the unnecessary dependency between the steps in a method. The dependence relation between any two steps in a method is determined by comparing their readsets and writesets. This gives rise to the following definition.

Definition 41 (*Dependence Graph:*) The dependence graph of a method m_j^i is a directed graph (possibly disconnected [BM76]) $DG(m_j^i) = (V, E)$ where each vertex $v_k \in V$ represents a step S_{jk}^i in m_j^i and a direct edge $e \in E$ from v_s to v_t indicates that $S_{js}^i \delta^? S_{jt}^i$. ■

A dependence graph chains the steps in a method according to the dependence relation which reflects the partial order relation between the steps (Definition 35). A dependence graph can be either a *connected* graph or a *disconnected* graph. A connected graph has only one root whereas a disconnected graph has two or more disconnected roots. Figure 3.6B and 3.6D illustrate an example of a connected graph and disconnected dependency graph of the methods of Figure 3.6A and 3.6C, respectively.

Suppose $STEPS(m_j^i)$ contains the steps in a method that occur according to the partial order defined between the steps. The writeset of a method m_j^i is constructed by taking the union of the writesets of the steps in m_j^i. The readset of m_j^i only contains the data that are either retrieved from the objectbase or the local variables entered by the users. The readset of m_j^i ($RS(m_j^i)$), and the writeset of m_j^i ($WS(m_j^i)$) are constructed as follows:

Let $RS(m_j^i) = \{\}$; $WS(m_j^i) = \{\}$;

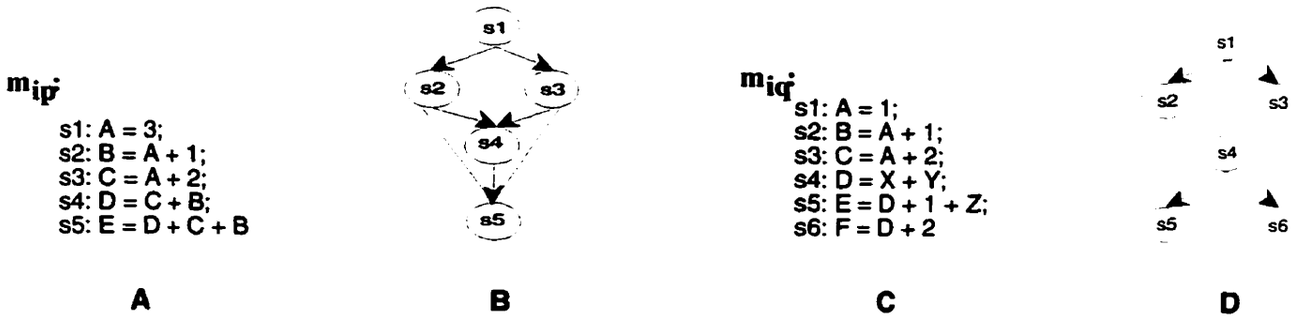


Figure 3.6: A connected and a disconnected dependence graph

For each $S^i_{jk} \in STEPS(m^i_j)$

$RS(m^i_j) = RS(m^i_j) \bigcup (RS(S^i_{jk}) - (WS(m^i_j) \bigcap RS(S^i_{jk})))$

$WS(m^i_j) = WS(m^i_j) \bigcup WS(S^i_{jk})$

end

Suppose, y is in the readset of step $S^i_{jk} \in STEPS(m^i_j)$, then y is also in the readset of m^i_j if it is not written by some previous steps that occur before S^i_{jk} in m^i_j. Therefore, in construction of the readset of m^i_j shown above, $(WS(m^i_j) \bigcap RS(S^i_{jk}))$ detects the data in S^i_{jk} which have been previously modified in m^i_j and $(RS(S^i_{jk}) - (WS(m^i_j) \bigcap RS(S^i_{jk})))$ collects the data which have not been accessed by previous steps in m^i_j. For example, in Figure 3.6A $RS(m^i_p) = \{\}$ because every data in $STEPS(m^i_p)$ has been already written in some previous steps in m^i_p. However, in Figure 3.6C, $RS(m^i_q) = \{X, Y, Z\}$ because X, Y, and Z are the only data which are read from the objectbase and not written in some previous steps.

The definition and the concepts discussed in this section are important factors for enhancing transaction management in a multiversion objectbase environment. Recall that the goal is to develop an optimistic concurrency control algorithm and apply the information captured from static analysis to increase concurrency control and produce reconciliation function for unsuccessful transactions at compile time. Information such as conservative construction of readset and writeset of methods determine if potential conflict exists be-

tween two transactions accessing the same object. Extent, reachableset, and attribute reference information form the basis of the implementation of the *depends* function (Definitions 19 and 20) which exploits intra-UT and inter-UT serializability. This is explained further in the next section.

Other information such as control flow path and dependence graph can impact the reconciliation algorithm. Reconciliation may involve re-execution of some operations. If re-execution is necessary, the dependency graph can be used to reread the stale data in order to re-execute the operations which are affected by the stale data. Further, if partial re-execution of a method affects other methods referenced or being referenced by the method, extent and reachableset can determine what other methods should be reconciled.

3.5.3 The *depends* Function

The detailed implementation of the *depends* function can now be discussed. Recall that the *depends* function accepts two operations within a transaction; at least one being a method invocation and returns true if there is a dependency in the internal semantics of the operations. The significance of the *depends* function 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, leaving the rest serialized.

Fully describing the implementation of the *depends* function requires a deep examination of compiler construction and a thorough treatment of the runtime systems. This is beyond this dissertation'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 [Gra94] and others [ASU86, GZB92].

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).

Algorithm 3.5.1 *(depends Function- the interface section)*

Algorithm *depends*($arg1, arg2$)
begin
if arg1=read/write OP_1 = arg1 **and** OP_2=arg2 (1)
elseif arg2=read/write OP_1 = arg2 **and** OP_2=arg1 (2)
else arg1=OP_1 **and** OP_2=arg2 (3)

!!!! *SECTION A* (Checking direct dependency between a local step and a message step)
if (OP_1=read) **and** ($OP_1 \in WS(OP_2)$) **OR** (4)
(OP_1=write) **and** ($OP_1 \in (WS(OP_2) \cup RS(OP_2))$) **then** (5)
return true; (6)
!!!! *SECTION B* (Checking hidden dependency between a local step and a message step)
elseif (OP_1 is a read/write on object o^f) **and** $o^f \in$ *reachableset*(OP_2) **then** (7)
for every m_i^f of $o^f \in$ *extent*(OP_2) **do** (8)
if (OP_1=read) **and** $OP_1 \in WS(m_i^f)$ **OR** (9)
(OP_1=write) **and** $OP_1 \in (RS(m_i^f) \cup WS(m_i^f))$ **then** (10)
return true (11)
!!!! *SECTION C* (Checking direct dependency between two message steps)
elseif
$RS(OP_1) \cap WS(OP_2) \neq \{\}$ **OR** (12)
$RS(OP_2) \cap WS(OP_1) \neq \{\}$ **OR** (13)
$WS(OP_1) \cap WS(OP_2) \neq \{\}$ (14)
return true; (15)
!!!! *SECTION D* (Checking indirect dependency between two message steps)
else if *Conflict-Set*(OP_1, OP_2) $\neq \{\}$ **then** (16)
return true (17)
!!!! *SECTION E* (Checking hidden dependency between two message steps)
elseif (OP_1 is a message step of a method in object o^f) **and** $o^f \in$ *reachableset*(OP_2) **then** (18)
for every m_i^f of $o^f \in$ *extent*(OP_2) **do** (19)
if ($RS(OP_1) \cap WS(m_i^f)$) $\neq \{\}$ **OR** (20)
($WS(OP_1) \cap (RS(m_i^f) \cup WS(m_i^f))$) $\neq \{\}$ **then** (21)
return true (22)
elseif (OP_2 is a message step of a method in object o^f) **and** $o^f \in$ *reachableset*(OP_1) **then** (23)
for every m_i^f of $o^f \in$ *extent*(OP_1) **do** (24)
if ($RS(OP_2) \cap WS(m_i^f)$) $\neq \{\}$ **OR** (25)
($WS(OP_2) \cap (RS(m_i^f) \cup WS(m_i^f))$) $\neq \{\}$ **then** (26)
return true (27)
!!!! *NO DEPENDENCY EXIST*
else
return false (28)
end

Figure 3.7: The *depends* function

The algorithm to implement the *depends* function is shown in Figure 3.7. The algorithm consists of several sections. The first two sections consider the case when one of the arguments passed to the *depends* function is a simple read/write operation and the other parameter is a method invocation. The next sections refer to the dependency between two method invocations. Suppose OP_1 represents the read/write operation and OP_2 represents the method invocation operation in sections A and B. Section A determines if the operations referenced by OP_2 (input and output parameters) locally depend on OP_1 (direct dependency). This dependency exist if OP_1 and OP_2 operate on the same data in a conflicting manner. Section B shows a form of hidden dependency between a simple read/write operation and a method invocation. Suppose OP_2 calls m_j^t (a method in object o^t) which in turn calls m_k^f (a method in object o^f). If OP_1 is an operation on object o^f, some attributes referenced by m_k^f may conflict with OP_1 and as the result OP_1 and OP_2 can be dependent.

Similarly, section C checks if two message steps invoked from the same object method locally depend on each other (direct dependency). This is done by comparing the data referenced by the two message steps. Section D checks if two message steps may indirectly call some conflicting methods in a common object (indirect dependency). The function $Conflict\text{-}Set(OP_1, OP_2)$ shown in line 16 detects such a dependency and can be defined as:

Definition 42 ($Conflict\text{-}Set$): The Conflict-set of two message steps $msg1$ and $msg2$ ($Conflict\text{-}Set(msg1, msg2)$) is a set of pairs $< m_i^f, m_j^f >$ where m_i^f, and m_j^f are two methods of object o^f such that $m_i^f \in extent(msg1)$, $m_j^f \in extent(msg2)$, and m_i^f and m_j^f may access attributes in o^f in conflicting manner. ■

If the conflict-set is empty, no indirect dependency occurs; otherwise, potential indirect dependency exists.

The function *Conflict-set* may not always detect some hidden dependencies. For example, if OP_1 and OP_2 are two message steps of method m_i^f in object o^f and o^f is an element of $reachableset(OP_2)$, then there exists at least a method $m_j^f \in extent(OP_2)$ in which OP_1 and the method m_j^f may access some conflicting attributes in o^f. This is another

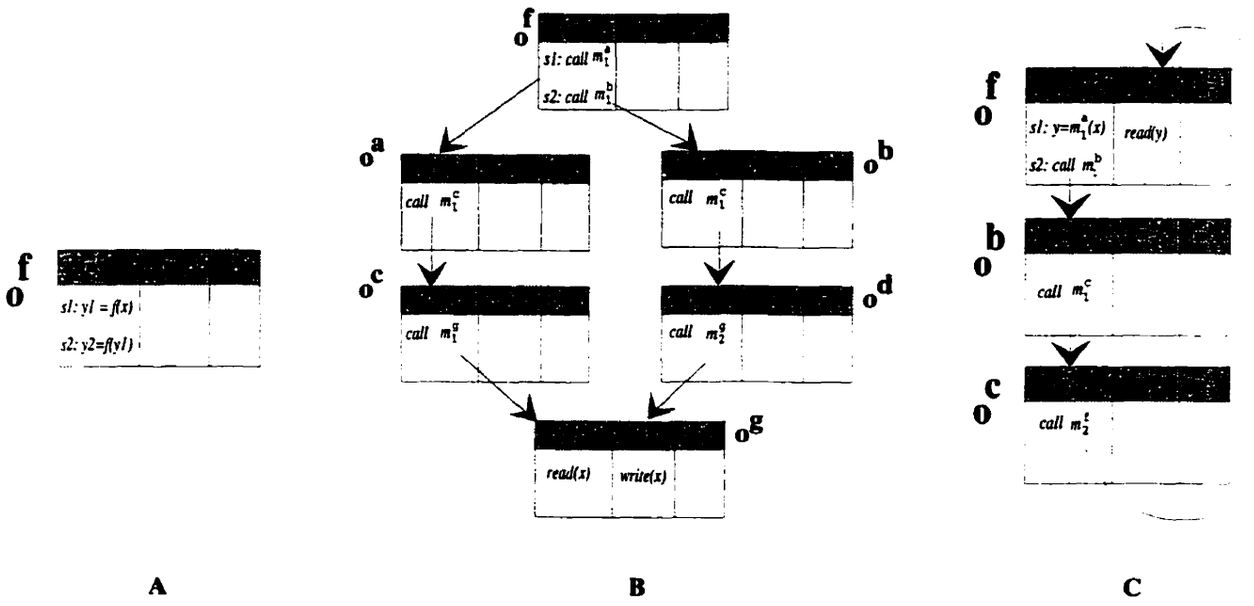


Figure 3.8: dependency of the statements in a method

form of hidden dependency which may exist between two message steps invoked from the same method. The last section in the algorithm (section E) determines if such a hidden dependencies exist. If none of the above conditions are satisfied, the *depends* function returns false indicating that no dependency exists between the two operations. Examples of direct, indirect, and hidden dependencies are illustrated in Figure 3.8A, 3.8B, and 3.8C, respectively.

Figure 3.8 illustrates several cases. Direct dependency is the easiest case. Figure 3.8A shows an example of direct dependency between two statements s_1 and s_2 in method m_1^f. Clearly s_1 and s_2 access conflicting operations. Comparing the readsets and the writesets of s_1 and s_2 determines that the execution of s_1 depends on the execution of s_2.

Figure 3.8B illustrates indirect dependency between two statements s_1 and s_2 in m_1^f. s_1 and s_2 do not conflict locally but both indirectly invoke some conflicting methods in o^g. The $Conflict\text{-}Set(m_1^a, m_1^b) = \{< m_1^g, m_2^g >\}$ because m_1^g and m_2^g belong to the extents of m_1^g and m_2^g, respectively, and access some conflicting operations in o^g.

Figure 3.8C illustrates an example of hidden dependency. Note that statements s_1 and s_2 neither directly nor indirectly conflict. But s_2 indirectly accesses 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 the method that is indirectly invoked by s_2 in o^f. Similarly, s_2 may conflict with some methods that may be called by s_1 in o^f indirectly.

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.

3.6 Summary of Assumptions

In the description of the architecture and the computational model presented in this chapter, we have made some assumptions. The assumptions are as follows.

1. Although the ideas in this research may also be applicable to some other database systems, discussion is limited to the objectbase environments only.

2. The objects referred to in this research only embody the key requirements of objects without including unnecessary extensions. This limitation simplifies the problem by the elimination of unnecessary special cases and makes the work more generally applicable because it does not need to cater to particular features of some objectbase systems.

3. We assume that during the execution of a set of transactions, the schema is static. Schema evolution may change the structure and the behavior of the objects which in turn may impact transaction management. Therefore, we assume that if a change in

the schema must occur, the system stops accepting the users requests and waits for the active transactions to terminate. Once the schema is changed, recomplied, and tested, the system accepts the user requests again.

4. This research adopts the convention of including method invocation using a procedure call syntax. Therefore, a method invocation (including one within a user transaction) can accept several parameter and where appropriate optionally returns explicit results. (eg: Routine $MethodName$(arg1,arg2,...)). This is merely the most familiar notation available.

5. Creating a version of an instance of a class (object) does not add a new instance to that class. In other words, versions are just the copies of objects that are used for the purpose of transaction management. Versions are non-persistent entities that may be promoted to stable objects.

6. The model assumes that a number of subtransactions are executed on versions of objects on behalf of the user transactions. It is assumed that the user transaction model identifies the version transactions as it is done in nested transaction models [Mos85].

7. Transaction nesting is closed in our model. When a subtransaction of a user transaction updates the data in the versions of objects, the changes are revealed to other subtransactions from the same transaction family after the subtransaction pre-commits. The changes made in some versions by the subtransactions of a transaction family are revealed to other transaction families when that transaction family commits.

8. Like flat transactions, it is assumed that $ACID$ properties are enforced on user transactions in our model. The execution of each user transaction is atomic; its results leaves the objectbase in a consistent state; its execution is isolated from the execution of other user transactions; and the changes it makes in the objectbase are persistent after it commits. Note that our concurrency control algorithm which will be presented in the next chapter ensures consistency and isolation of transactions. But, we assume that transactions are durable and the system is reliable.

9. Our model mainly utilizes encapsulation in the object model as it relates to concurrency. Other interesting object properties such as inheritance, aggregation, and polymorphism may impact transaction management but exploitation of these issues is beyond the scope of this thesis. Therefore, we treat each object individually and do not worry about how each object is instantiated. We assume that objects are created by some mechanism and our task is to *manage* these objects efficiently. In other words, whether an object has inherited properties of other objects (a part-of hierarchy) or from other classes (an inheritance hierarchy) does not impact on the correctness or performance of the algorithms presented in this thesis.

Chapter 4

The Architectural Model

This chapter introduces an architecture and an optimistic multiversion concurrency control algorithm. The algorithm describes the components of the architecture in detail. Examples illustrating complex parts of the algorithm are provided incrementally. We argue about the correctness of the algorithm based on intra-UT and inter-UT serializabilities.

4.1 The Architecture

A versioned object store is comprised of two parts: a non-persistent *unstable* working store, and a persistent *objectbase*. The unstable store contains active versions. Committed versions are maintained in the objectbase. An object, with its committed versions, construct an *object family*.

1. An active version is
   - mutable,
   - derived from a committed version, and
   - can be promoted to a committed version.
2. A committed version is
   - persistent,

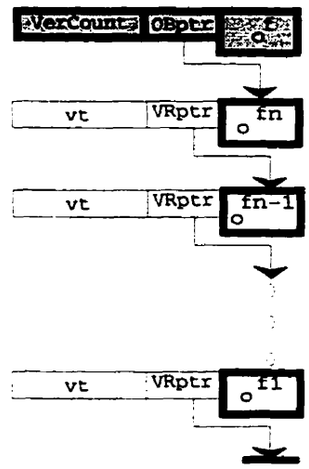


Figure 4.1: Logical structure of committed versions in an object family

- created from the promotion of an active version.

Beside the characteristics captured by Definition 16, an object needs system attributes that describe its versions. The system attributes of an object are:

- *OBptr*: points to the last committed version of an object.
- *VerCount*: keeps track of the number of committed versions of an object.

A version of an object also requires the following system attributes:

- *VRptr*: points to the next current committed version of the object; otherwise, it points to nil.
- *vt*: records the valid time [SA86] of the version. The valid time is when the transaction associated with the version commits and the version may be revealed.

To distinguish different version types of o^f, we denote v^{fi} and o^{fi} to represent an active version and a committed version of o^f, respectively. A particular data item x in version v^{fi} is unambiguously denoted as x^{fi}. Figure 4.1 shows a logical structure of committed versions of an object o^f in the objectbase. Committed versions $o^{f1}, o^{f2}, ..., o^{fn}$ are associated with

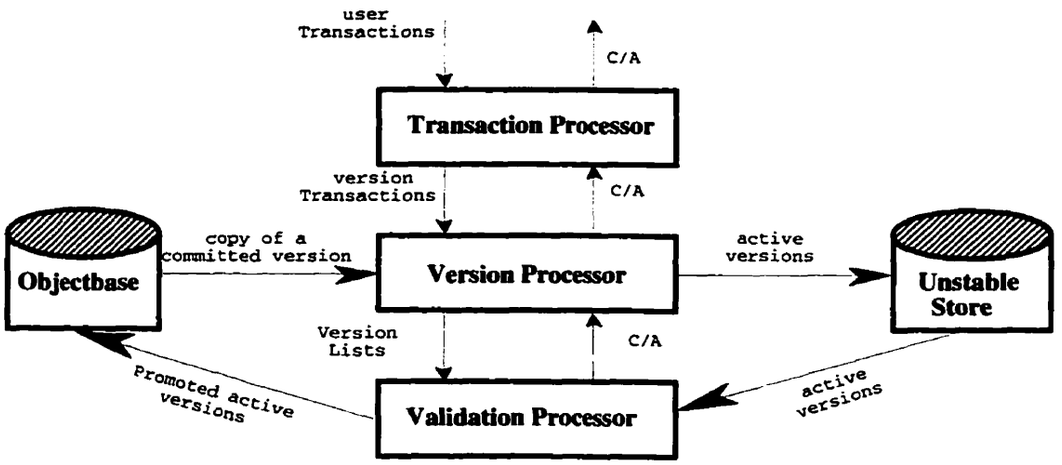


Figure 4.2: The Main Components of the Architecture

user transactions $UT_1, UT_2, ..., UT_n$, respectively and their positions in the version-chain of o^f reflect the serialization order of such transactions.

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

4.1.1 The Architectural Model

Three major components form the basis of our architecture: the *Transaction Processor*, the *Version Processor*, and the *Validation Processor* (Figure 4.2). 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 (using the *depends* function) for each user transaction. The Version Processor receives the 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 ver-

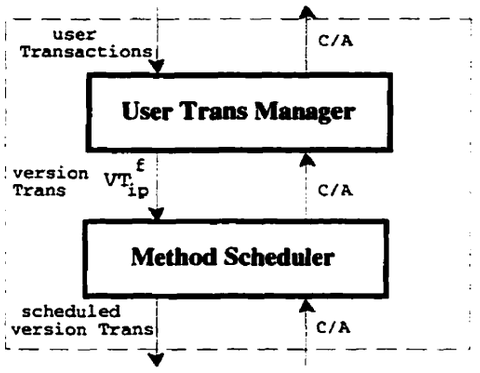


Figure 4.3: The Transaction Processor

sion 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. If it is possible to commit the user transaction, the Validation Processor promotes the active versions associated with it to committed versions.

Figure 4.3 shows the Transaction Processor in greater 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 version transaction p of UT_i executing on an active version of the object family 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 active version are ordered before they are sent to the Version Processor. Version transactions of multiple user transactions are executed concurrently.

Figure 4.4 shows the two components of the Version Processor: 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 from

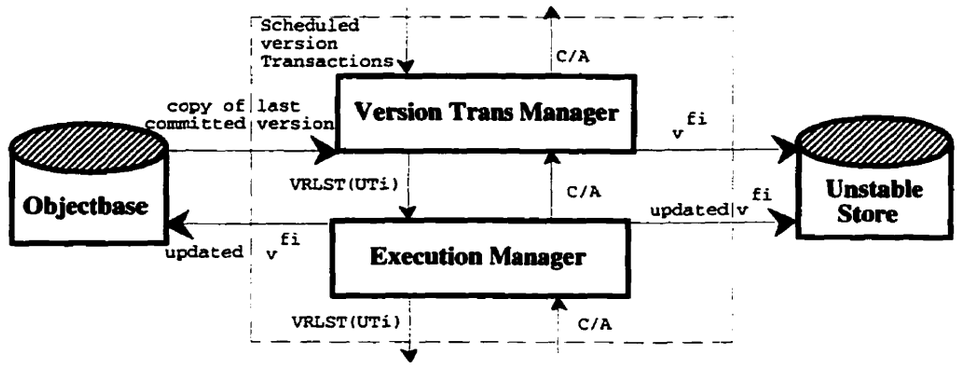


Figure 4.4: The Version Processor

the object family f in the objectbase (v^{fi}) is requested and is placed in the unstable store[1]. Unless it is specified, the active version v^{fi} originates from the last committed version in the object family f. 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 unstable store.

The Version Transaction Manager also builds a version list for each active user transaction. The version list of UT_i ($VRLST(UT_i)$) records the active versions referenced by UT_i. Every time an active version v^{fi} is created for UT_i, the Version Transaction Manager appends f (the object family identifier of v^{fi}) to $VRLST(UT_i)$. When every version transaction of UT_i completes, $VRLST(UT_i)$ 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 active versions (Figure 4.5). It has two components: the Decision Manager and the Commit Manager. The Decision Manager checks for the validity of each active version by comparing each updated active version (v^{fi}) referenced by its object family *id* in the version list with the last committed version of object family f in the objectbase.

[1] In this thesis, since each user transaction UT_i obtains at most one active version from an object family, the active version *id* can be the same as the user transaction *id*.

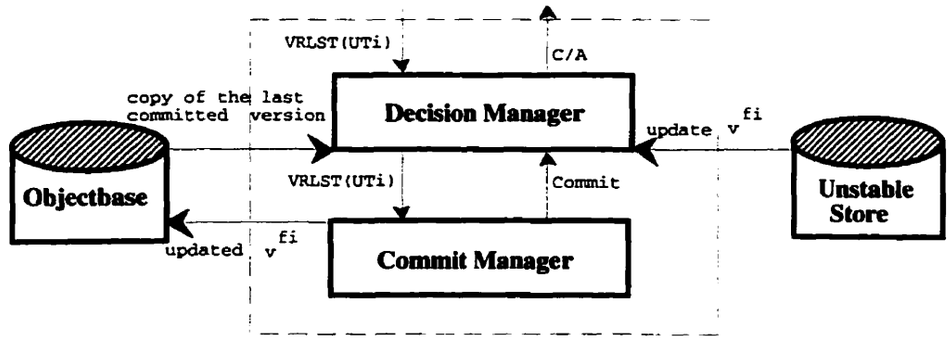


Figure 4.5: The Validation Processor

For example, assume Figure 4.6A shows the state of the objectbase just before the creation of v^{fi}. o^{fj} is the most current committed version in object family f when v^{fi} is created. Since v^{fi} originates from o^{fj}, o^{fj} is referred to as the *base version* of v^{fi}. It is possible that during the execution of UT_i other user transactions, UT_{j+1}, ..., UT_n commit and produce new committed versions $o^{fj+1}, o^{fj+2}, ..., o^{fn}$ in the object family f (see Figure 4.6B).

When UT_i terminates, the Decision Manager compares each updated active version v^{fi} referenced by its object family *id* in $VRLST(UT_i)$ with the most recent committed version of object family f, (o^{fn}). o^{fn} is the committed version located at the top of the version-chain of object family f. The purpose of the comparison is to determine if updated active versions would create inconsistency in the objectbase. The state of an updated active version v^{fi} is consistent with the states of committed versions in object family f if the values of the attributes read by UT_i in v^{fi} have not been modified in the objectbase during the life time of UT_i. If the states of all active versions of UT_i are consistent with states of their corresponding committed versions in the objectbase, $VRLST(UT_i)$ is sent to the Commit Manager; otherwise some active versions of UT_i are *invalid* and UT_i should be reconciled.

Reconciliation is the process of correcting the invalid active versions of UT_i with respect to the current state of the objectbase. First, the Decision Manager determines if it is possible to change the commit order of UT_i with respect to recently committed user transactions.

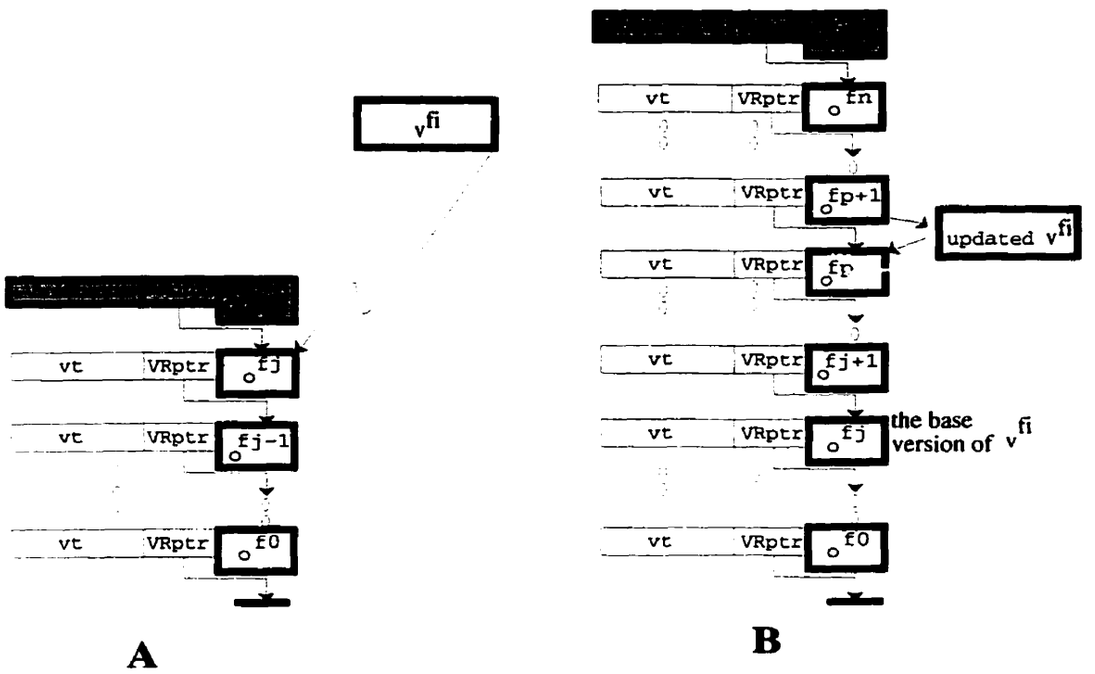


Figure 4.6: Insertion of an active version in the chain

Then, the Decision Manager attempts to find a position (based on the correctness criterion, 1value serializability) for each active version v^{fi} referenced by UT_i in the version-chain of the object family f. This process is called ***simple reconciliation.*** Consider Figure 4.6 again. An active version v^{fi} can be added between any two committed versions o^{fp} and o^{fp+1}, $j \leq p \leq n$, in the version-chain if:

1. the values of the data read by UT_i in v^{fi} when v^{fi} was originally created are still unchanged in o^{fp}, and
2. the data read in $o^{fp+1}, o^{fp+2}, ..., o^{fn}$ by $UT_{p+1}, UT_{p+2}, ..., UT_n$, respectively do not *value-intersect* (*value*-$\cap$) with the data written by UT_i in v^{fi}. The *value-intersection* of two sets A and B is a set $C = \{x_1, x_2, ..., x_n\}$ 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.

The first point indicates that although UT_i, the user transaction associated with v^{fi}, has read its required data items from o^{fj} to manipulate v^{fi}, v^{fi} can also be serialized after o^{fp} if UT_i could have read the required data items from o^{fp} instead. The second point indicates that if some user transactions $UT_{p+1}, UT_{p+2}, ..., UT_n$ read a data item x in which x was last modified by one of $UT_p, UT_{p-1}, ..., UT_0$, they can still read x from v^{fi} as long as UT_i has not modified the value of x in v^{fi}.

If v^{fi} can be added between any two committed versions of o^f, v^{fi} is considered a valid updated active version. Validity of all updated active versions in the $VRLST(UT_i)$ only ensures intra-object serializability. A mechanism must be developed to check inter-object serialization of the user transactions. This implies that for any two user transactions UT_i and UT_j, that have referenced some committed versions o^{fi} and o^{fj}, respectively, if o^{fi} is serialized before o^{fj}, then effectively for every object family k that is commonly accessed by both UT_i and UT_j, v^{ki} must be serialized before v^{kj}. Techniques such as *Global Serialization Graph* [ZB93c] can be used to determine inter-object serializability (see Section 2.3.4). The vertices in the global serialization graph represent the user transactions. An edge in the graph from UT_i to UT_j implies that some committed version o^{fi} of UT_i has been serialized after some committed version o^{fj} of UT_j in the object family f. A cycle in the graph can detect if a set of concurrently running user transactions is inter-object 1value serializable.

After checking both intra-object and inter-object serializabilities, the Decision Manager sends $VRLST(UT_i)$ to the Commit Manager.

If both intra-object and inter-object serializabilities are guaranteed, the Commit Manager promotes the updated active versions to committed versions and records them in the objectbase. Otherwise, version transactions of UT_i which have accessed the stale data may partially be re-executed against some of the active versions of UT_i. This process is called *complex reconciliation*. Complex reconciliation affects both the invalid active versions and other active versions which relate directly or indirectly to the invalid versions. In the complex reconciliation of a user transaction, the Decision Manager uses the information from the dependency graph and the control flow path graph to re-execute the basic blocks affected by the stale data. When complex reconciliation is complete, and the results made from all the updating versions of UT_i become consistent with the current state of their committed versions in their corresponding object families, $VRLST(UT_i)$ is passed to the Commit Manager. The Commit Manager promotes the updated versions to the committed versions, records the committed versions in the objectbase, and commits the user transaction.

Since the number of committed versions in the object families grows overtime, periodically some of committed versions are archived. Issues related to archiving the committed versions and their storage management are beyond the scope of this thesis. The complete architecture is shown in Figure 4.7.

In the rest of this chapter, we explain the function of each component of the architecture and describe the basic optimistic algorithm. Simple and complex reconciliations are studied in detail in the subsequent chapters.

4.2 The Implementation

The following is the list of the routines and data structures required by the Transaction Processor.

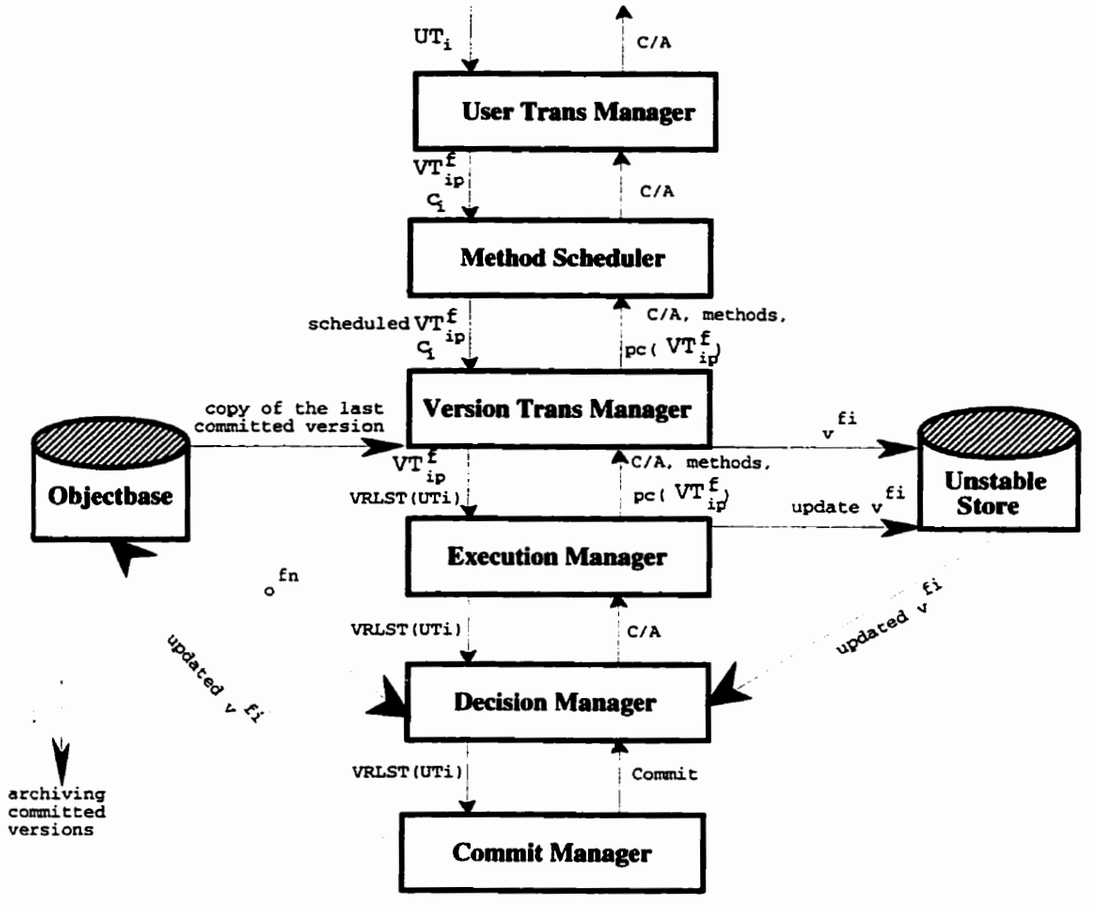


Figure 4.7: The Architecture

Send(C,MSG): Sends message MSG to component C. Each component contains a message queue. The message MSG is located at the back of the message queue of component C.

Output(Result): Submits the final result (*commit* or *abort*) of the execution of a user transaction to the user.

CreateVT(M): Converts the method invocation M to a version transaction. A method invocation is an operation of a user transaction or a version transaction.

SetOrder(f,VT,VT'): Sets an ordering between two version transactions VT and VT' which may commonly access some active versions of object family f in a conflicting manner. This ordering ensures that the execution of VT is serialized after VT'.

ReleaseOrder(f,VT,VT'): Removes the order between the two version transactions VT and VT' which have been previously set by the $SetOrder()$ function.

counter(UT): Is a counter kept for each user transaction UT. The counter is incremented when a direct/indirect method (version transaction) of UT is invoked and is decremented when a version transaction of UT terminates.

ConflictObject(VT,VT'): Is a function that returns a set of objects which may possibly be referenced in a conflicting manner by some subtransactions of VT and VT'. This function can be implemented based on the information captured from the $depends(VT, VT')$, $reachablesets(VT)$, $reachableset(VT')$, and $Conflict\text{-}set(VT, VT')$.

A user transaction, UT_i is submitted to the Transaction Processor (see Figure 4.8). The User Transaction Manager decomposes UT_i into a set of method invocations (lines 1-2), converts them to version transactions (line 4), and passes version transactions to the Method Scheduler on behalf of UT_i (line 5). This process continues until every operation of UT_i including the termination condition, c_i, is sent to the Method Scheduler (line 6). A *counter* is associated with each user transaction UT_i ($counter(UT_i)$) to control the number of pre-committed version transactions of UT_i. $counter(UT_i)$ is incremented when a method (version transaction) of UT_i is invoked (line 3) and it is decremented when a version transaction of UT_i pre-commits.

The Method Scheduler ensures the proper ordering defined by the *depends* function (Section 3.7) is enforced before submitting the version transactions to the Version Processor (see Figure 4.9). Multiple version transactions executing on a version of an object

Algorithm 4.2.1 *(User Transaction Manager - the interface section)*

Algorithm User.Trans.Mgr
input: A user transactions operation set ($\sum_i$) and its partial order ($\prec_i$) from the user, and the result of the termination of a user transaction from the Method Scheduler.
output: The operations of a user transaction to the Method Scheduler, or the result of the execution of a user transaction to the user.

```
begin
   **** INFORMATION RECEIVED FROM THE USER ****
   case input of
      UT_i:
         for every operation τ_ip ∈ Σ_i do                          (1)
            if τ_ip = m_j^f then                                    (2)
               counter(UT_i) ← counter(UT_i) + 1                    (3)
               VT ← CreateVT(m_j^f)                                 (4)
               Send(Method.Scheduler,VT)                            (5)
            elseif τ_ip = c_i
               Send(Method.Scheduler,c_i)                           (6)
   **** INFORMATION RECEIVED FROM THE METHOD SCHEDULER ****
      C/A(UT_i):
         Output(C/A(UT_i))                                          (7)
   end{case}
end
```

Figure 4.8: The User Transaction Manager

Algorithm 4.2.2 *(Method Scheduler - Ordering the related methods)*

Algorithm Method.Scheduler
input: A version transactions of a user transaction, the commit operation of a user transaction from the User Transaction Manager, and a version transaction of a user transaction, termination condition (pre-commit) of a version transaction, or the result of the execution of a user transaction from Version Transaction Manager.
output: The scheduled version transactions or the termination condition of a user transaction to the Version Transaction Manager, and the result of the execution of a user transaction to the User Transaction Manager.
var: received(c_i): It is a flag to indicate that the Method Scheduler has received c_i.

begin
case input of
**** INFORMATION RECEIVED FROM THE USER TRANSACTION MANAGER ****
VT^f_{ip}:
for all active VT^e_{iq} such that *depends*(VT^f_{ip}, VT^e_{iq}) **do** (1)
A $\leftarrow$ *ConflictObjects*(VT^f_{ip}, VT^e_{iq}) (2)
$\forall o^m \in A$ **do** (3)
SetOrder($o^m, VT^f_{ip}, VT^e_{iq}$) (4)
Send(Ver.Trans.Mgr,VT^f_{ip}) (5)
else
c_i:
received(c_i) $\leftarrow$ true (6)
**** INFORMATION RECEIVED FROM VERSION TRANSACTION MANAGER ****
C/A:
Send(User.Trans.Mgr,$C/A(UT_i)$) (7)
VT^e_{iq}:
for all VT^f_{ik} such that *depends*(VT^e_{iq},VT^f_{ik}) **do** (8)
$A \leftarrow$ *ConflictObjects*(VT^e_{iq}, VT^f_{ik}) (9)
$\forall o^m \in A$ **do** (10)
SetOrder($o^m, VT^e_{iq}, VT^e_{ik}$) (11)
Send(Ver.Trans.Mgr,VT^e_{iq}) (12)
$pc(VT^f_{ip})$:
counter(UT_i) $\leftarrow$ counter(UT_i) $-$ 1 (13)
while $\exists$ VT^e_{ik} such that *depends*(VT^e_{ik}, VT^f_{ip}) **do** (14)
A $\leftarrow$ *ConflictObjects*(VT^e_{ik}, VT^f_{ip}) (15)
$\forall o^m \in A$ **do** (16)
ReleaseOrder($o^m, VT^e_{ik}, VT^f_{ip}$) (17)
if (counter(UT_i) = 0) **and** (received(c_i)) **then** (18)
Send(Ver.Trans.Mgr,c_i) (19)
end{case}
end

Figure 4.9: The Method Scheduler

originating from the same user transaction may need to be ordered. For example, suppose the execution of VT_{ip}^{f} depends on the execution of VT_{iq}^{e} (line 1) and set A represent the set of all common objects that may be referenced by some descendants of VT_{ip}^{f} and VT_{iq}^{e} (line 2). The Method Scheduler sets a serialization order between VT_{ip}^{f} and VT_{iq}^{f} for the active versions of the objects in A to block the execution of VT_{ip}^{f}'s descendants at these active versions until VT_{iq}^{e} pre-commits (lines 3-4). Then it sends VT_{ip}^{f} to the Version Transaction Manager (line 5). When the Method Scheduler receives c_i (line 6), it holds c_i until all of the direct and indirect version transactions of UT_i pre-commit.

The following routines are also defined for the description of the Version Processor.

StoreUnstable(v): Stores active version v in the unstable store.

Copy(f): Returns a duplicate of the most recent committed version in object family f.

BeforeImage(v): Is a snapshot of a version v, before v is modified. This snapshot is required during the validation of the version at commit time.

readset(v): Is a set that collects the attributes in version v retrieved by the transactions.

writeset(v): Is a set that collects the attributes in version v updated by the transactions.

Delete(v): Deletes active version v from the unstable store.

Ordering(VT,VT',f): Returns true if the Method Scheduler has ordered the execution of version transactions VT after VT' in some active versions of object family f.

The Version Transaction Manager maintains a version list for each user transaction. A version list records the object *id* of the active versions that are referenced by a user transaction. Thus, each element of a version list refers to an object family identifier f in which an active version of o^f has been derived for a user transaction. The Version Transaction Manager receives scheduled version transactions (VT_{ip}^{f}'s) from the Method Scheduler (see Figure 4.10). For each VT_{ip}^{f} received, the Version Transaction Manager checks the version list of UT_i ($VRLST(UT_i)$) to ensure no active version associated with object family f for UT_i already exists in the unstable store (line 2). If this is the case, the Version Transaction Manager adds the object family identifier f to the $VRLST(UT_i)$

Algorithm 4.2.3 *(Version Transaction Manager - Creating active versions)*

Algorithm Ver.Trans.Mgr

input: Scheduled version transactions of a user transaction, or the commit operation of a user transaction from the Method Scheduler, and the termination condition(pre-commit) of a version transaction or the result of the execution of user transaction from the Execution Manager.

output: A version transaction of a user transaction, or the version list of a user transaction to the Execution Manager, and a version transaction of a user transaction, the termination condition of a version transaction, or the result of the execution of a user transaction to the Method Scheduler.

begin
  case input of
  **** INFORMATION RECEIVED FROM THE METHOD SCHEDULER ****
    c_i:
      Send(Exe.Mgr,$VRLST(UT_i)$) (1)
    VT^f_{ip}:
      if f **not** $\in VRLST(UT_i)$ **then** (2)
        $VRLST(UT_i) \leftarrow VRLST(UT_i) \cup \{f\}$ (3)
        $v^{fi} \leftarrow$ copy(o^{fn}) ! o^{fn} is the last committed version of o^f (4)
        BeforeImage(v^{fi}) = v^{fi} (5)
        StoreUnstable(v^{fi}) (6)
      Send(Exe.Mgr,VT^f_{ip}) (7)
  **** INFORMATION RECEIVED FROM THE EXECUTION MANAGER ****
    VT^e_{iq}:
      Send(Method.Scheduler,VT^e_{iq}) (8)
    $pc(VT^f_{ip})$:
      Send(Method.Scheduler,$pc(VT^f_{ip})$) (9)
    $C/A(UT_i)$:
      for every $v^{fi} \in VRLST(UT_i)$ **do** (10)
        Delete(v^{fi}) (11)
      Send(Method.Scheduler,$C/A(UT_i)$) (12)
  end{case}
end

Figure 4.10: The Version Transaction Manager

(line 3). Then it obtains a copy, v^{fi}, of a committed version of o^f (by default the most recent committed version in object family f) from the objectbase (line 4) and stores v^{fi} in the unstable store (line 6). Upon the creation of v^{fi}, a snapshot of the state of v^{fi} called $BeforeImage(v^{fi})$ is taken (line 5). $BeforeImage(v^{fi})$ is required for validating v^{fi} at commit time. The significance of *BeforeImage* will be discussed when we explain Validation Processor later in this section. Next, the Version Transaction Manager sends VT^f_{ip} to the Execution Manager (line 7).

When the Execution Manager (see Figure 4.11) receives VT^f_{ip}, it may not execute VT^f_{ip} immediately. Before the Execution Manager executes VT^f_{ip}, it needs to check if another conflicting version transaction VT^f_{iq} has already been ordered before VT^f_{ip} in the object family f. If this is the case, the Execution Manager blocks the execution of VT^f_{ip} until it is notified that VT^f_{iq} has pre-committed (line 2).

Eventually, VT^f_{ip} starts executing on v^{fi}. Recall from Section 3.3 that the operations of a version transaction are read, write, method invocation, and pre-commit. Also recall that x^{fi} refers to a data item in v^{fi}. When the operation is a "read" on attribute x^{fi}, x^{fi} is added to the readset of v^{fi}, if it has not been accessed before (lines 3-4). Then x^{fi} is read (line 5). When the operation is a "write" on x^{fi}, the Execution Manager checks if x^{fi} has already been updated. If this is not the case, it adds x^{fi} to the writeset of v^{fi} (lines 6-7). Then it updates x^{fi} (line 8). If the operation is τ_{iq} and τ_{iq} is the invocation of method j in object o^e (m^e_j), the Execution Manager increments $counter(UT_i)$ (line 9), changes the method invocation to version transaction VT^e_{iq} (line 10), and sends VT^e_{iq} to the Version Transaction Manager so it can be scheduled with other version transactions of UT_i (line 11).

Eventually, VT^f_{ip} terminates and processes the pre-commit operation. The pre-commit operation indicates the end of the execution of VT^f_{ip}. The Execution Manager notifies the Version Transaction Manager about the completion of VT^f_{ip} (line 12).

The following additional routines are required to describe the Validation Processor's algorithms.

Algorithm 4.2.4 *(Execution Manager - executing a version transaction)*

Algorithm Exe.Mgr

input: A version transaction of a user transaction, or the version list of the versions accessed by a user transaction from the Version Transaction Manager, and the result of the execution of a user transaction from the Decision Manager.

output: The version list of the versions accessed by a user transaction to the Decision Manager. and the result of the termination of a version transaction of a user transaction, the subtransaction of a version transaction, or the result of the execution of a user transaction to the Version Transaction Manager.

begin
  case input of
  **** INFORMATION RECEIVED FROM THE VERSION TRANSACTION MANAGER ****
    $VRLST(UT_i)$:
      Send(Decision.Mgr,$VRLST(UT_i)$) (1)
    VT_{ip}^f:
      begin ! begins the execution of VT_{ip}^f
        while $\exists VT_{iq}^f$ such that Ordering(VT_{ip}^f, VT_{iq}^f,f) in o^f *wait* (2)
        repeat
          case *operation* of
            read:
              if $x^{fi} \notin$ (readset(v^{fi}) $\cup$ writeset(v^{fi})) **then** (3)
                readset(v^{fi}) $\leftarrow$ readset(v^{fi}) $\cup$ $\{x^{fi}\}$ (4)
              read x^{fi} (5)
            write:
              if $x^{fi} \notin$ writeset(v^{fi}) **then** (6)
                writeset(v^{fi}) $\leftarrow$ writeset(v^{fi}) $\cup$ $\{x^{fi}\}$ (7)
              update x^{fi} (8)
            τ_{iq} = call m_j^e:
              counter(UT_i) $\leftarrow$ counter(UT_i) + 1 (9)
              $VT_{iq}^e \leftarrow$ CreateVT(m_j^e) (10)
              Send(Ver.Trans.Mgr,VT_{iq}^e) (11)
            pc:
              Send(Ver.Trans.Mgr,*pc*(VT_{ip}^f)) (12)
          end{case for the operations}
        Until *operation* = *pc*
      end ! ends the execution of VT_{ip}^f
  **** INFORMATION RECEIVED FROM THE DECISION MANAGER ****
    $C/A(UT_i)$:
      Send(Ver.Trans.Mgr,$C/A(UT_i)$) (13)
  end{case for input}
end

Figure 4.11: The Execution Manager

AddEdge(G,UT,UT'): Adds an edge in a directed graph G from the vertex that represents user transaction UT to the vertex that represents user transaction UT'.

Promote(v): Promotes active version v to a committed version.

RemoveVertex(G,UT): Removes the vertex which represents user transaction UT with all of its associated edges from graph G.

StoreObjectbase(v): Stores a committed version v at the top of the version-chain associated with v in the objectbase.

As shown in the architecture (Figure 4.7), eventually the User Transaction Manager submits c_i to the Method Scheduler. When the Method Scheduler receives c_i, it keeps c_i until all of the version transactions of UT_i terminate (counter(UT_i) is set back to zero) (Figure 4.9 line 18). Then It sends c_i to the Version Transaction Manager (Figure 4.9 line 19). When the Version Transaction Manager receives c_i, it sends $VRLST(UT_i)$ to the Execution Manager (Figure 4.10 line 1) and the Execution Manager passes $VRLST(UT_i)$ to the Decision Manager (Figure 4.11 line 1).

When a version list is received by the Validation Processor, it may not be validated immediately if other *related* version lists are being validated. Two version lists $VRLST(UT_1)$ and $VRLST(UT_2)$ are *related* if there exists an object family f in which UT_1 and UT_2 have commonly referenced f and the data items accessed in v^{f1} and v^{f2} by UT_1 and UT_2 respectively, value-conflict.

To order the validation of the *related* version lists, the Decision Manager constructs a *Validation Graph* (see Figure 4.12). A validation graph for a set of version lists $VRLST(UT_1)$, $VRLST(UT_2)$, ..., $VRLST(UT_n)$ is a graph (V, E), where V is a set of vertices and E is a set of edges. A vertex $v_i \in V$ represents a $VRLST(UT_i)$ and an edge v_i to v_j indicates that $VRLST(UT_i)$ and $VRLST(UT_j)$ are *related*. The validation graph orders the validation of *related* version lists. When a user transaction terminates, $VRLST(UT_i)$ with all of its associated edges are removed from the validation graph (lines 1-3). Thus, eventually, the Decision Manager processes $VRLST(UT_i)$ and must decide whether UT_i should be committed or aborted (line 4).

Algorithm 4.2.5 *(Decision Manager - Validating the versions accessed by a user transaction)*

Algorithm Decision.Mgr

input: The version list of the versions referenced by a user transaction from the Execution Manager, and successful termination (Commit) of a user transaction from the Commit Manager.

output: A validated version list of a user transaction to the Commit Manager, and the result of the execution of a user transaction to the Execution Manager.

begin
case input of
**** INFORMATION RECEIVED FROM THE EXECUTION MANAGER ****
$VRLST(UT_i)$:
for every $VRLST(UT_j) \in$ validation graph (1)
if $VRLST(UT_i)$ and $VRLST(UT_j)$ are *related* **then** (2)
AddEdge(validation.graph,$VRLST(UT_i)$,$VRLST(UT_j)$) (3)
if $VRLST(UT_i)$ is not related to any other version lists **then** (4)
for every $v^{fi} \in VRLST(UT_i)$ **do** (5)
$o^{fn} \leftarrow$ the last committed version in object family f (6)
if $\exists x^{fi} \in readset(v^{fi})$ such that $BeforeImage(x^{fi}) \neq x^{fp}$ **then** (7)
Send(Commit.Mgr,$Abort(UT_i)$) (8)
RemoveVertex(validation.graph, $VRLST(UT_i)$) (9)
Send(Commit.Mgr,$VRLST(UT_i)$) (10)
****INFORMATION RECEIVED FROM THE COMMIT MANAGER ****
$commit(UT_i)$:
RemoveVertex(validation.graph, $VRLST(UT_i)$) (11)
Send(Exe.Mgr,$commit(UT_i)$) (12)
end

Figure 4.12: The Decision Manager

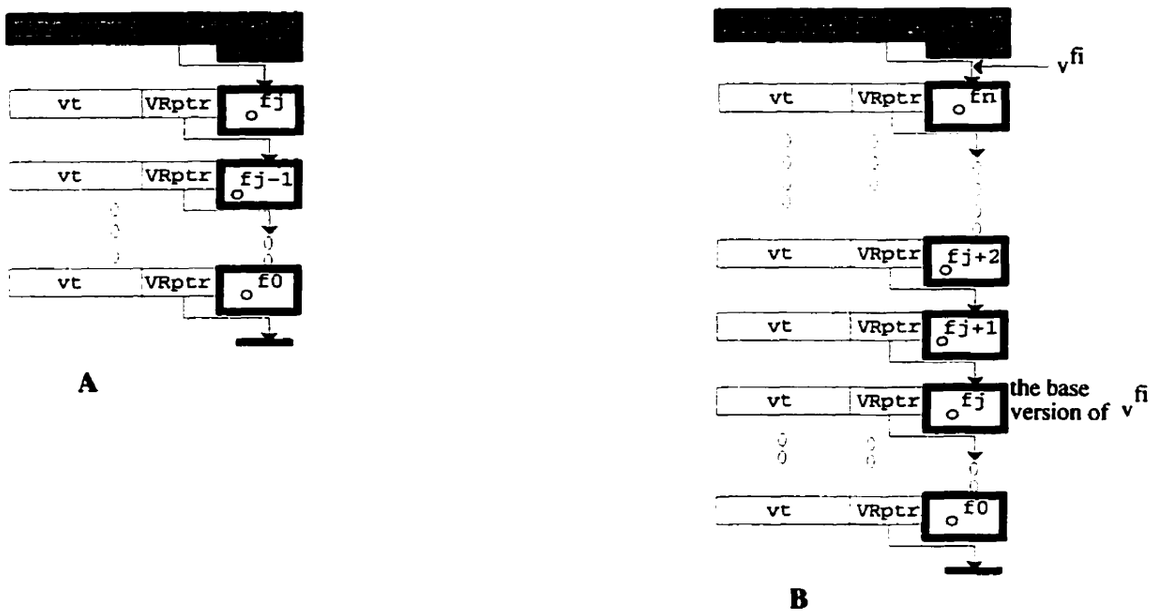


Figure 4.13: Revision may be required before promotion af active version

Consider Figure 4.13. Suppose active version v^{fi} originates from o^{fj} when UT_i references object family f. Recall that o^{fj} is the base version of v^{fi}. When UT_i terminates, the Decision Manager must check if it is possible to promote v^{fi} to a committed version and add it to top of the version-chain above o^{fn}. Note that during the life time of UT_i, other user transactions $UT_{j+1}, UT_{j+2}, ..., UT_n$ have committed and created committed versions $o^{fj+1}, o^{fj+2}, ..., o^{fn}$, respectively. v^{fi} can be inserted above o^{fn} in the version-chain, if it is possible to serialize UT_i after $UT_{j+1}, UT_{j+2}, ..., UT_n$ in the object family f. This condition holds if UT_i (the user transaction associated with UT_i) can read the same information from o^{fn} as it originally read from o^{fj}.

Thus, to determine if v^{fi} can be inserted above the committed version o^{fn}, the Decision Manager checks if every data item x^{fi} read in v^{fi} is still unchanged in o^{fn} (see Figure 4.12). Simple comparison of each data item x^{fi} read in v^{fi} with its corresponding x^{fp} in o^{fp} is not possible because during the manipulation of v^{fi}, x^{fi} may have been modified. Recall that when an active 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. Thus, to ensure that all data items read from v^{fi} are still unchanged in o^{fp}, x^{fp} in o^{fp} is compared with the value of x^{fi} in the BeforeImage of v^{fi} (line 5-7).

UT_i can be committed if the above condition holds for every active version referenced by UT_i. If this is not the case, UT_i is aborted (line 8) and $VRLST(UT_i)$ is removed from the validation graph so that other *related* version lists can be processed (line 9). Otherwise, $VRLST(UT_i)$ is sent to the Commit Manager (line 10).

Although some data items of the updated versions referred to in $VRLST(UT_i)$ are never accessed, they still have to be checked and updated if necessary (see Figure 4.14). Thus, for each v^{fi} referenced by UT_i (line 1), the Commit Manager compares data in v^{fi} and o^{fn}, the last committed version, against each other. If data item x^{fi} is not accessed in v^{fi} by any version transaction and its value is different from the value of x^{fn} in o^{fn} (line 2-4), the Commit Manager moves the value of x^{fn} to x^{fi} overwriting the old value (line 5). We call the process of updating attribute values in this way "revision", because some data items of the updated versions must be revised before they can be stored in the objectbase.

An example is shown in Figure 4.15. Suppose the state of the object family f just before UT_i starts is shown in Figure 4.15A. When UT_i becomes active, it obtains a copy of o^{f0}, v^{fi}, and executes m_3^f against v^{fi}. Figure 4.15B shows the state of object family f when UT_i terminates. Note that during the life time of UT_i other user transactions, UT_1 and UT_2, have committed and produced the committed versions o^{f1} and o^{f2}, respectively. UT_i have accessed attributes a and b whereas UT_1 and UT_2 have modified attributes c and d, respectively (assuming that UT_1 has executed m_1^f and UT_2 has executed m_2^f against their own versions). Although operations in UT_i does not conflict with the ones in UT_1 and UT_2, promoting v^{fi} to o^{fi} and inserting o^{fi} at the top of the version-chain causes a "loss of update" to the values of c and d (Figure 4.15B). Thus v^{fi} is revised first before it is promoted in which the values of c and d in o^{f2}, the last committed version in object family f, are moved to the attributes c and d in v^{fi}, overwriting the old values (Figure 4.15C).

Algorithm 4.2.6 *(Commit Manager - Revising and storing the versions accessed by a user transaction in the objectbase)*

Algorithm Commit.Mgr
input: $VRLST(UT_i)$: a version list of of a user transaction from the Decision Manager.
output: the successful termination of a user transaction to the Decision Manager.

begin
****INFORMATION RECEIVED FROM THE DECISION MANAGER ****
for every $(v^{fi}) \in VRLST(UT_i)$ **do** (1)
for every data x^{fi} in v^{fi} and corresponding x^{fn} in o^{fn} **do** (2)
$o^{fn} \leftarrow$ the last committed version in object family f. (3)
$\forall x^{fi}$ **if** $x^{fi} \neq x^{fn}$ **and** x^{fi} **not** $\in$ (readset(v^{fi}) $\cup$ writeset(v^{fi})) **then** (4)
$x^{fi} \leftarrow x^{fn}$ (5)
$o^{fi} \leftarrow$ Promote(v^{fi})) (6)
Storeobjectbase(o^{fi}) (7)
$vt.o^{fi} \leftarrow gettime()$ (8)
$o^f.VerCount \leftarrow o^f.VeRcount + 1$ (9)
Send(Decision.Mgr, $Commit(UT_i)$) (10)
end;

Figure 4.14: The Commit Manager

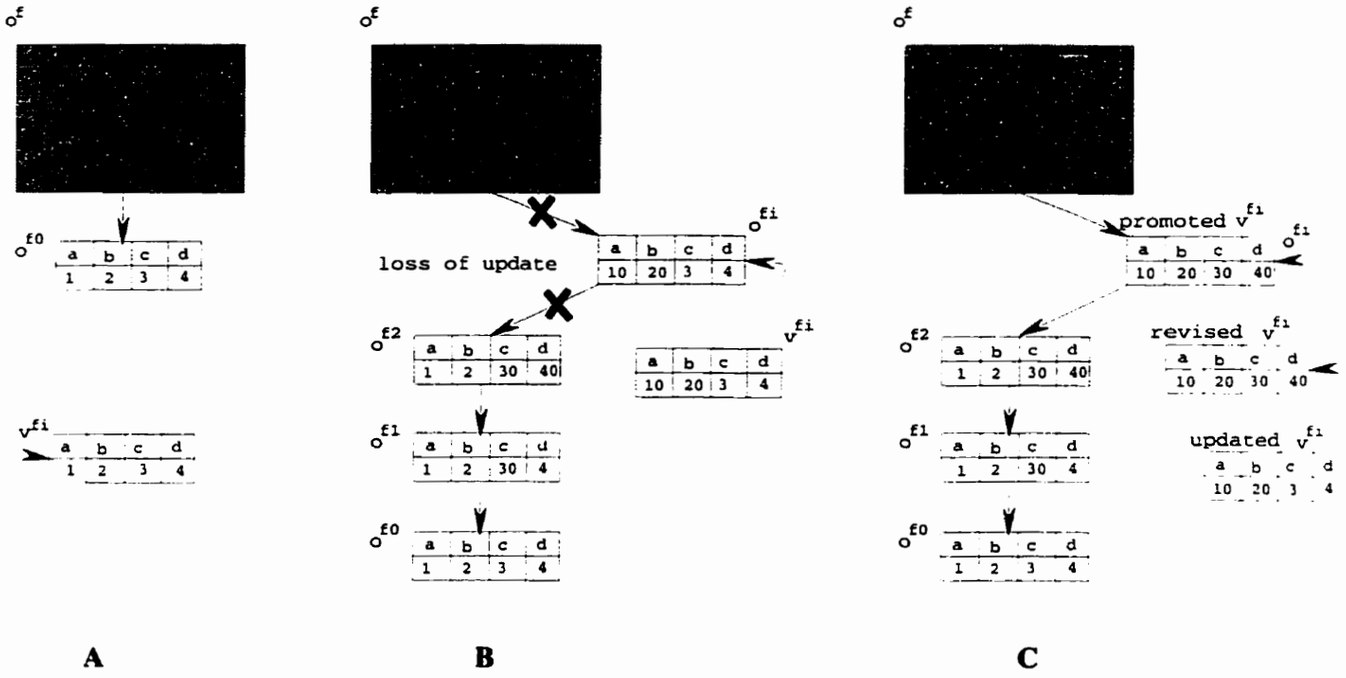


Figure 4.15: Revision of an updated active version

Thus, for each updated v^{fi} (see Figure 4.14), the Commit Manager revises v^{fi} (lines 2-5), promotes v^{fi} to a committed version o^{fi} (line 6), and places o^{fi} at the top of version-chain in object family f (line 7). o^{fi} is linked to other committed versions and becomes the last committed version in object family f. Upon the insertion of o^{fi} into the version-chain, the valid time of o^{fi} is recorded and the system attribute $VerCount$ is incremented (lines 8-9). Next, the Commit Manager sends a commit message to the Decision Manager indicating that the effect of UT_i is now committed in the objectbase (line 10). Once the Decision Manager receives the message, it removes $VRLST(UT_i)$ from the validation graph and passes the message to the Execution Manager (see Figure 4.12 lines 11,12). The final result of the UT_i will be eventually passed to the user.

Other Communications

Three different types of information flow from the Execution Manager to the Version Transaction Manager and from the Version Transaction Manager to the Method Scheduler. First, when the Execution Manager receives Commit/Abort result (C/A) of UT_i from the Decision Manager, it passes C/A to the Version Transaction Manager (Figure 4.11 line 13). The Version Transaction Manager retrieves the updated active versions associated with UT_i and removes them from the unstable store (Figure 4.10 lines 10-11). Then it passes C/A to the Method Scheduler (Figure 4.10 line 12).

Second, recall that the Execution Manager may submit some version transactions to the Version Transaction Manager. The Execution Manager can only execute the primitive operations (read, write, pc) against the active versions in the unstable store. Method invocations must be converted to version transactions first. Then the version transactions must be scheduled before they can be processed by the Execution Manager. Therefore, if the Execution Manager encounters operation τ_{iq} that is the method invocation m_j^e while processing VT_{ip}^f, it converts m_j^e to version transaction VT_{iq}^e (Figure 4.11 line 10) and sends VT_{iq}^e to the Version Transaction Manager (Figure 4.11 line 11). Since the version transactions must be scheduled with the other active version transactions of UT_i, the Version Transaction Manager passes the version transactions received from the Execution Manager

to the Method Scheduler (Figure 4.10 line 8).

Third when the Execution Manager completes the execution of a version transaction VT_{ip}^{f}, it sends the pre-commit message of VT_{ip}^{f} to the Version Transaction Manager (Figure 4.11 line 12). The Version Transaction Manager notifies the Method Scheduler about the completion of VT_{ip}^{f}, (Figure 4.10 line 9). When the Method Scheduler receives the pre-commit message of VT_{ip}^{f}, it removes the order between VT_{ip}^{f} and any version transaction that depends on VT_{ip}^{f} (Figure 4.9 lines 14-17). Since the pre-commit message refers to a termination of a version transaction of UT_i, the Method Scheduler decrements counter(UT_i) (Figure 4.9 line 13) and checks the counter to determine if the counter is set back to zero (Figure 4.9 line 18). A zero value for the counter indicates that all the methods of UT_i have been processed and completed. If this is the case, the Method Scheduler sends c_i to the Version Transaction Manager (Figure 4.9 line 19).

Finally, the User Transaction Manager receives the result (C/A of UT_i) and outputs it to the user (Figure 4.8 line 7).

4.3 Correctness of the Algorithm

This section explains serializability of the transactions at version and the user levels.

4.3.1 Version-Level Concurrency Control

Recall that two version transactions associated with two different user transactions never conflict. However, it is possible that the execution of a version transaction depends on another if both are associated with the same user transaction. Therefore, some mechanism must be provided to order these version transactions.

When the Method Scheduler receives a version transactions VT_{ip}^{f}, it calls the *depends* function to determine if VT_{ip}^{f} depends on some other active version transactions of UT_i. For example, if VT_{ip}^{f} depends on VT_{iq}^{e}, the Method Scheduler sets an ordering between VT_{ip}^{f}

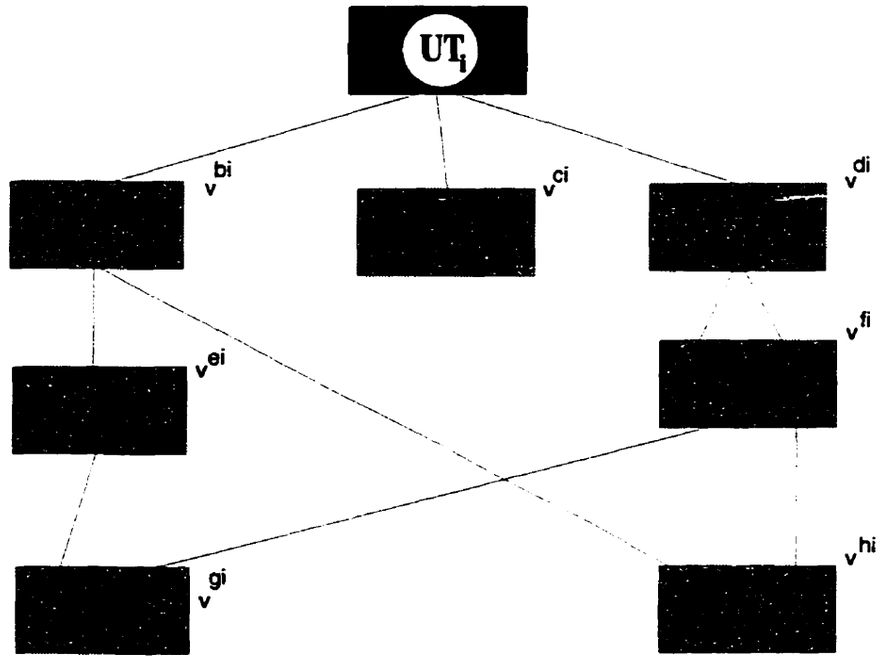


Figure 4.16: Intra-UT Concurrency Control

and VT^e_{iq} so that the execution of VT^f_{ip} is serialized after VT^e_{iq} in every active version of UT_i which may possibly be accessed by some descendants of VT^f_{ip} and VT^e_{iq}.

An example of intra-UT serializability is shown in Figure 4.16. The user transaction UT_i invokes three methods that are eventually converted to version transactions VT^b_{i2}, VT^c_{i3}, and VT^d_{i4}. VT^b_{i2} becomes active first and starts executing. Then the Method Scheduler processes VT^c_{i3}, and VT^d_{i4}. VT^c_{i3} can be executed with VT^b_{i2} since each references a set of separate objects. However, the set of objects in which VT^b_{i2} and VT^d_{i4} may commonly reference is $A = \{o^h, o^g\}$. Recall that, for each user transaction, only one active version of each object can exist in the unstable store, and all version transactions of a user transaction execute on that active version. In this example, descendants of VT^b_{i2} and VT^d_{i4} may commonly access the attributes of active versions in $A' = \{v^{hi}, v^{gi}\}$ in a conflicting manner.

To control intra-UT serializability, the Method Scheduler must ensure that in every

active version in A', the execution of the descendants of VT^b_{i2} happens before the descendants of VT^d_{i4} if necessary. Thus when VT^h_{i10} and VT^h_{i11} access conflicting operations in v^{hi}, VT^h_{i10} is ordered before VT^h_{i11} in v^{hi}. Similarly if VT^g_{i8} and VT^g_{i9} access conflicting operations, VT^g_{i8} is ordered before VT^g_{i9} in v^{gi}. Thus when the Version Transaction Manager sends VT^h_{i11} and VT^g_{i9} to the Execution Manager, they have to wait until VT^b_{i2} pre-commits. When VT^b_{i2} pre-commits, the Method Scheduler removes the order between VT^b_{i2} and VT^d_{i4} so that the descendants of VT^d_{i4}, VT^g_{i9} and VT^h_{i11}, can start executing.

4.3.2 User-Level Concurrency Control

User transactions are serialized by the Decision Manager using the validation graph that orders the *related* version lists. When a version list is sent to the Validation Processor, two cases are possible. If the Decision Manager and the Commit Manager are idle, $VRLST(UT_i)$ is added to the validation graph and then is processed by the Decision Manager. Otherwise, $VRLST(UT_i)$ is compared with other version lists in the graph to check if other *related* version lists should be validated before $VRLST(UT_i)$. When a user transaction UT_i commits or aborts, $VRLST(UT_i)$ and its incoming edges are removed so other version lists *related* to $VRLST(UT_i)$ are validated as long as they are not *related* to other version lists.

Although *related* version lists are ordered by the validation graph, non-*related* version lists may contain versions associated with the same transaction families. For example, if $VRLST(UT_1) = \{v^{f1}\}$ and $VRLST(UT_2) = \{v^{f2}\}$ and the data accessed in v^{f1} and v^{f2} are not in value-conflict, $VRLST(UT_1)$ and $VRLST(UT_2)$ are not *related* and may be validated concurrently. However, non-*related* versions associated with the same object family should be promoted and recorded in the objectbase one at a time. Thus, before a version v^{fi} is promoted to o^{fi} and recorded in the objectbase, the entire object family f is locked for validation; therefore, creating a critical section which prevents other versions from modifying the version-chain. This procedure serializes the validity of the version lists which in turn ensures inter-object serializability.

Chapter 5

Simple Reconciliation

This chapter introduces simple reconciliation. Simple reconciliation uses previously committed versions to commit a terminated user transaction. This implies, when a user transaction UT_i cannot be committed after some recently committed transaction UT_j, simple reconciliation is an attempt to commit UT_i before UT_j as long as the state of objectbase remains consistent. This chapter begins by giving an example of the case when normal committing procedure of a user transaction fails. Then it shows how to modify the components of the Validation Processor to do simple reconciliation of unsuccessful transactions. Examples are provided incrementally to verify complex parts of the algorithm. Finally, as an extension to the basic algorithm, it will be described how historical information can be retrieved from the objectbase.

Figure 5.1 shows an example of the case when execution of a user transaction is not successful and reconciliation is required. Figure 5.1A shows the original state of the object family f. UT_1 and UT_4 start first and each receives a copy of o^{f0} denoted by v^{f1} and v^{f4}, respectively. UT_1 executes m_1^f against v^{f1} and UT_4 executes m_4^f against v^{f4}. UT_1 commits and its associated committed version o^{f1} is recorded in the objectbase (Figure 5.1B). Then UT_2 starts, obtains a copy of o^{f1} (v^{f2}), executes m_2^f against v^{f2}, and commits. v^{f2} is promoted to o^{f2} and is recorded in the objectbase (Figure 5.1C). Next UT_3 starts, obtains

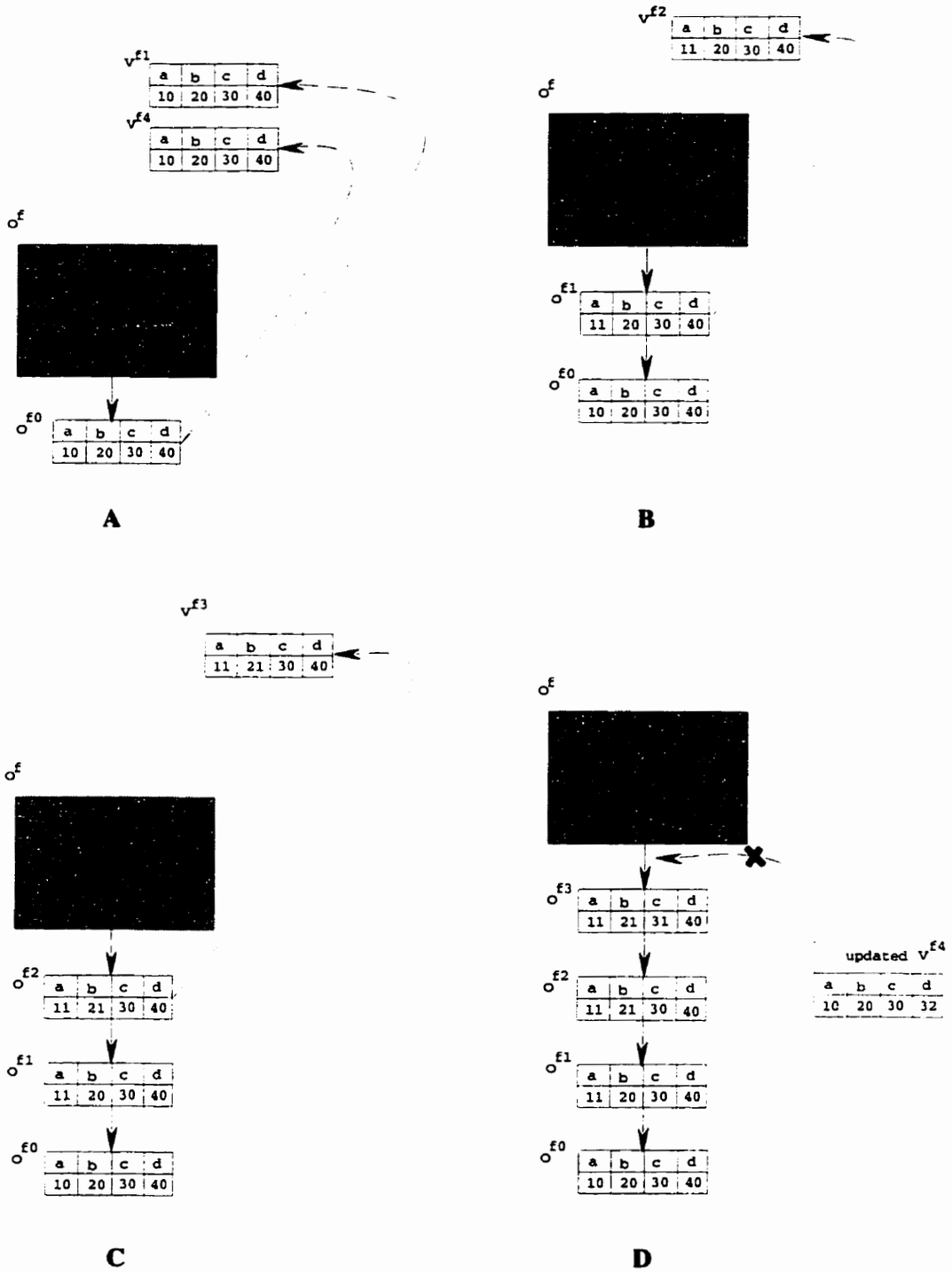


Figure 5.1: Reconciliation is required

a copy of o^{f2} (v^{f3}), executes m_3^f against v^{f3} and commits. v^{f3} is promoted to o^{f3} and is recorded in the objectbase (Figure 5.1D). Now UT_4 terminates; however, v^{f4} cannot be committed as the last committed version in the chain because the value of c read in v^{f4} is stale. Although v^{f4} cannot be at the top of version chain in the object family f, it might be possible to insert it in the lower levels without making the state of other committed versions inconsistent. Simple reconciliation is the process that checks and inserts v^{f4} in an appropriate position in the version-chain. This is done by the components of the Validation Processor, the Decision Manager and the Commit Manager.

5.1 Decision Manager

In the simple reconciliation of a user transaction (UT_i), for each active version v^{fi} in $VRLST(UT_i)$, the Decision Manager must find a valid position in the version-chain of object family f where v^{fi} may be inserted. This is subject to two conditions. First, insertion of v^{fi} in the version-chain of the the object family f must ensure intra-object serializability at the object family f. Second, the entire transaction system must remain inter-object serializable.

5.1.1 Intra-object Serializability

Consider Figure 5.2. Suppose active version v^{fi} originates from o^{fj} when UT_i references object family f. The following additional data structure is required:

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

The process of finding $ValidPos[v^{fi}]$ starts from the top of the chain where o^{fn} is located and proceeds down until either a position is found to insert v^{fi} or no position can be found. v^{fi} can be inserted between any two committed versions o^{fp} and o^{p+1} if:

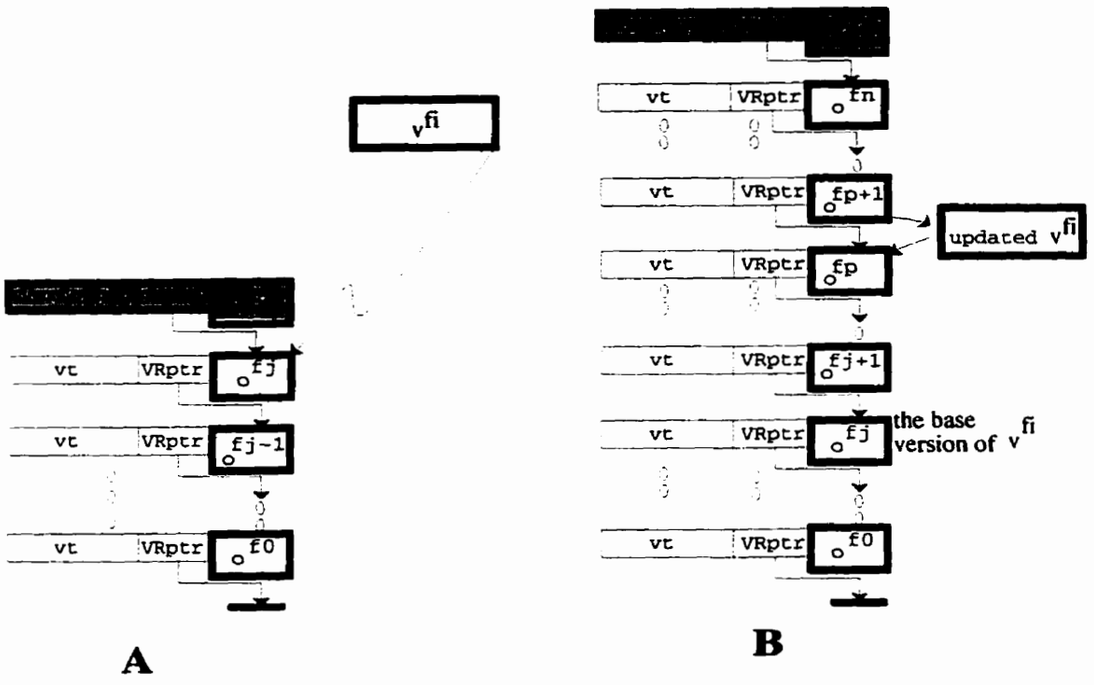


Figure 5.2: Finding a position in the chain

x in readset(o^{fk})	x in writeset(v^{fi})	x in $\cup$ writeset(o^{fs})	Allowed
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1/0
1	1	1	1

A

x in writeset(o^{fm})	x in readset(v^{fi})	x in $\cup$ writeset(o^{fs})	Allowed
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1/0
1	1	1	1

B

Figure 5.3: Possible cases when reconciliation may or may not succeed

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

The first condition holds if UT_i (the user transaction associated with v^{fi}) can read the same information from o^{fp} as it originally read from o^{fj} (the base version of v^{fi}). 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 have not been modified by UT_i in v^{fi}. For example, suppose x is read by UT_{p+1} ($x \in readset(o^{fp+1})$) and it was modified last by UT_p in o^{fp} ($x \in writeset(o^{fp})$). If v^{fi} does not contain the same value for x as o^{fp} does, v^{fi} can be placed between o^{fp} and o^{fp+1} because UT_{p+1} cannot be serialized after UT_i in object family f.

Figure 5.3A shows all possible cases that arise when the scheduler attempts to insert v^{fi} below o^{fk} ($o^{p+1} \leq o^{fk} \leq o^{fn}$). Similarly Figure 5.3B checks if it is possible to place v^{fi} above o^{fm} ($o^{f0} \leq o^{fm} \leq o^{fp}$). The truth tables in Figure 5.3 show that v^{fi} can be inserted in the chain between o^{fp} and o^{fp+1} if:

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

and

$\forall o^{fm}\ (o^{f0} \leq o^{fm} \leq o^{fp})$
if $x \in writeset(o^{fm})$ **and** $x \in readset(v^{fi})$ and $x \notin \bigcup_{s=m+1}^{p} writeset(o^{fs})$ **then**
UT_i reads the same value for x as UT_m writes into x.

Lines 1 through 14 in Figure 5.4 attempts to find a proper position in the version-chain where v^{fi} can be inserted. The process starts from the top of the version-chain (lines 2-3) and proceeds down the version-chain (lines 5-7). To determine if v^{fi} can be inserted above a committed version o^{fp}, every data item x^{fi} read in v^{fi} must be unchanged in o^{fp}. Recall the basic algorithm in the previous chapter (Section 4.2), to ensure that all of the data items read from v^{fi} are still unchanged in o^{fp}, x^{fp} in o^{fp} is compared with the value of x^{fi} in the BeforeImage of v^{fi} (line 4).

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

5.1.2 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.

Extension of Decision Manager to do Simple Reconciliation
begin
!!!!!!!!! CHECKING INTRA-OBJECT SERIALIZABILITY
for every $v^{fi} \in VRLST(UT_i)$ **do** (1)
pos $\leftarrow$ 0 (2)
$o^{fp} \leftarrow o^{fn}$!!! STARTING FROM TOP OF THE CHAIN (3)
if $\exists x^{fi} \in readset(v^{fi})$ such that $BeforeImage(x^{fi}) \neq x^{fp}$ **then** (4)
pos $\leftarrow$ pos +1 (5)
$o^{fp} \leftarrow$ next committed version (going down the chain) (6)
go to line 4 (7)
if a o^{fp} is found **then** (8)
for every o^{fk} that occurs above o^{fp} in the chain (9)
if $\exists x^{fk} \in readset(o^{fk})$ such that $x^{fi} \in writeset(v^{fi})$ **and** $x^{fk} \neq x^{fi}$ (10)
$\forall o^{fs}$ $(o^{fp+1} \leq o^{fs} \leq o^{fk})$, **if** $x^{fs} \notin writeset(o^{fs})$ **then** (11)
Abort UT_i (12)
else
Abort UT_i (13)
$ValidPos[v^{fi}] \leftarrow$ pos (14)

!!!!!!!!! CHECKING INTER-OBJECT SERIALIZABILITY
for every $v^{fi} \in VRLST(UT_i)$ **do** (15)
for every o^{fk} that occurs above v^{fi} in the chain **do** (16)
if $readset(v^{fi})$ *value-*$\cap$ $writeset(o^{fk}) \neq \{\}$ **OR**
$readset(o^{fk})$ *value-*$\cap$ $writeset(v^{fi}) \neq \{\}$ **then** (17)
add an edge from UT_k to UT_i in GSG (18)
for every o^{fm} that occurs below v^{fi} in the chain **do** (19)
if $readset(v^{fi})$ *value-*$\cap$ $writeset(o^{fm}) \neq \{\}$ **OR**
$readset(o^{fm})$ *value-*$\cap$ $writeset(v^{fi}) \neq \{\}$ **then** (20)
add an edge from UT_k to UT_i in GSG (21)
if there is a cycle in GSG **then** (22)
Abort UT_i (23)
Send $VRLST(UT_i)$ and $ValidPos[]$ to the Commit Manager (24)
end

Figure 5.4: The Decision Manager doing simple reconciliation

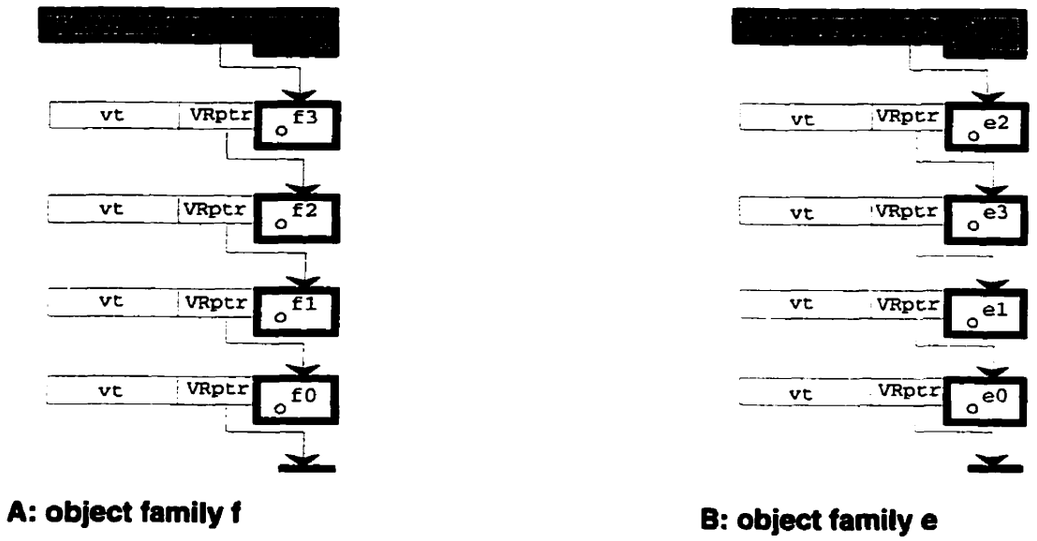


Figure 5.5: Example of a possible inter-object serialization Problem

Suppose three user transactions UT_1, UT_2 and UT_3 each accessing both object family f and object family e. UT_1 starts first, commits, and creates committed versions o^{f1} and o^{e1} in the objectbase. Then UT_2 and UT_3 start, make copies (active versions) of the committed versions created by UT_1 in object families f and e, and execute concurrently. Figure 5.5 shows the state of the object families f and e after UT_2 and UT_3 commit. Clearly, since o^{f2} occurs before o^{f3} ($UT_2 \rightarrow UT_3$) in object family f (Figure 5.5A) and o^{f3} occurs before o^{f2} ($UT_3 \rightarrow UT_2$) in object family e (Figure 5.5B) inter-object serializability may not be ensured. The *Global Serialization Graph* provided by Zapp and Barker [ZB93c] can be used to control inter-object serializability.

A Global Serialization Graph for a set of user transactions $T = \{UT_1, UT_2, ..., UT_n\}$ denoted ($GSG(T)$) is a graph (V, E), where V is a set of vertices and E is a set of edges. Each vertex $v_i \in V$ represents a user transaction UT_i and an edge from v_i to v_j indicates that there exist a committed version of an object family f, o^{fi}, associated with UT_i which occurs above the committed version o^{fj} associated with UT_j in the version-chain of object family f and UT_i and UT_j have accessed some value-conflicting operations in o^{fi} and o^{fj},

respectively. Thus, if there is an edge going from v_i to v_j in $GSG(\mathcal{T})$ then there must be a pointer (system attribute *VRptr* – see Section 4.1) going directly or indirectly from o^{fi} to o^{fj}. Thus, an edge in the global serialization graph shows the serialization order of two user transactions.

Recall that the *value-*$\cap$ (*value-intersection*) of two sets A and B is a set $C = \{x_1, x_2, ..., x_n\}$ 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 Figure 5.2 again. Let o^{fk} be a committed version in the version-chain located between o^{fp+1} and o^{fn}. An edge from UT_k to UT_i is added to the global serialization graph if the following condition holds.

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

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

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

When the edges are added to the $GSG(\mathcal{T})$, the graph is checked for a cycle. As long as no cycle is detected, inter-object serializability is ensured. When both intra-object and inter-object serializabilities are ensured $VRLST(UT_i)$ and *ValidPos* are passed to the Commit Manager (see Figure 5.4 lines 15-24).

5.2 Commit Manager

Based on the information in the *ValidPos*, the Commit Manager promotes the active versions to committed versions and locates them in the objectbase. Before an active version v^{fi} is added between o^{fp} and o^{fp+1} (see Figure 5.2), in the version-chain, the Commit

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

Figure 5.6: The Commit Manager doing simple reconciliation

Manager must ensure that insertion of v^{fi} in the version-chain leaves other committed versions of object family f in a consistent state. Note that during the execution of UT_i against v^{fi}, some other user transactions may operate on the data items in their own versions in which UT_i does not access the corresponding data in v^{fi}. Thus when v^{fi} is inserted above o^{fp}, the Commit Manager must check if the value of every attribute x^{fi} in v^{fi} that is not accessed by UT_i is the same as its corresponding attribute x^{fp} in o^{fp}. If this is not the case, the value of x^{fp} is propagated to x^{fi} overwriting the old value of x^{fi} (Figure 5.6 lines (1-6)).

An example is shown in Figure 5.7. Suppose UT_i has obtained a copy of o^{f0} and during its life time other committed versions o^{f1}, o^{f2}, and o^{f3} have been created (5.7A). Suppose the Decision Manager has found that v^{fi} can be inserted between o^{f1} and o^{f2} (5.7B). Although attribute c^{fi} in v^{fi} has not been accessed by UT_i, the value of c^{f1} has to overwrite the value of c^{fi} in order to serialize UT_i after UT_1 in the object family f.

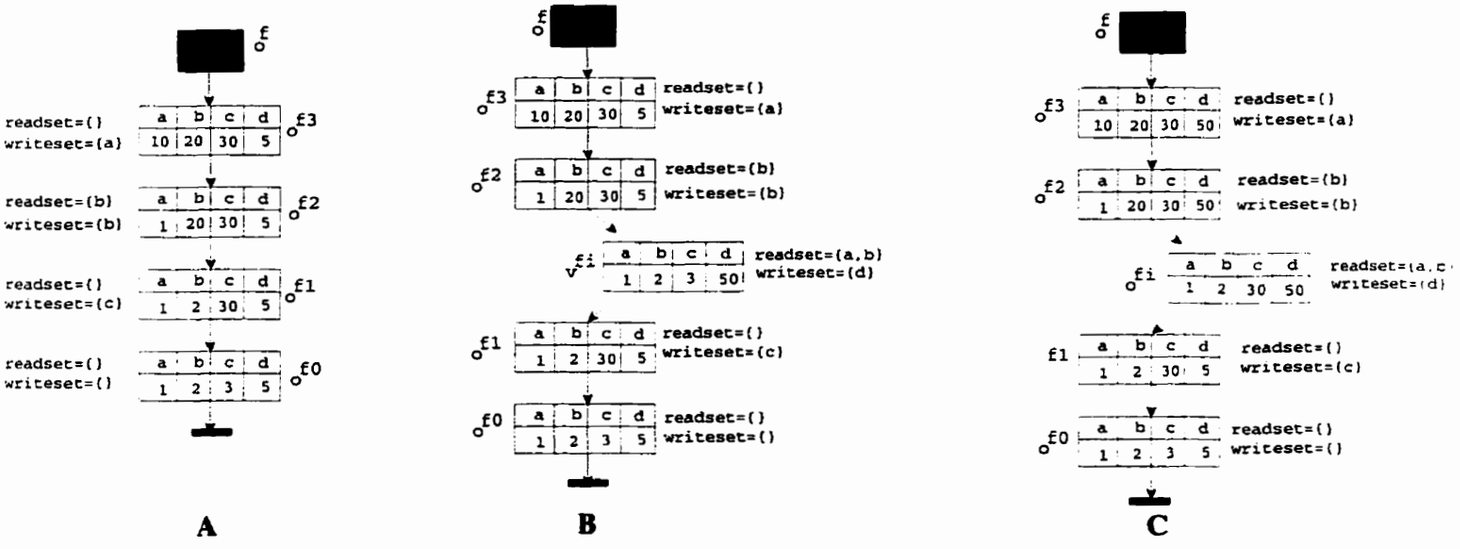


Figure 5.7: Propagation of the values to higher level committed versions

Further consider Figure 5.7B. If a user transaction requests a copy of the most recent information in the object family f, a copy of o^{f3} is obtained. However, o^{f3} in Figure 5.7B does not include all of the recent information after v^{fi} joins the object family f. The reason is that the value of d in o^{f3} is 5 which is not the most recent value of d in the object family f. The most recent value of d is 50 which is seen only in v^{fi}. This problem can be solved by propagating the values of the variables in the writeset of v^{fi} to the corresponding variables of the other committed versions which occur above it (Figure 5.6 lines 7-10).

Finally, v^{fi} is promoted to the committed version o^{fi} and inserted in the proper position in the version chain. When all active version transactions of user transaction UT_i are promoted to committed versions and recorded in the objectbase, the Commit Manager stores the promoted active versions referenced by UT_i in the objectbase and commits UT_i (Figure 5.6 line 11). Figure 5.7C shows correct states of all committed versions in version family f.

5.3 Retrieving Historical Information

One advantage of executing transactions in multiversion databases compared to traditional databases is that, it may be possible to execute read-only transactions concurrently with update transactions. Further, with the maintenance of historical data, transactions may request the information on past data values. This section presents the extensions to the algorithm to show how read-only transactions may retrieve historical data in the objectbase.

Three modifications are required. First a read-only user transaction is required to specify the time t in which the information should be retrieved. Second, since read-only transactions, do not modify the state of the objects, the Version Transaction Manager is not required to create a set of active versions for the associated version transactions. This reduces the overhead involved with creating the versions. Third, since no active version is created, the Execution Manager, is not require to send the result of the execution to the Validation Processor but the commit/abort result is sent directly to the user rather than to the Validation Processor.

Recall that the Method Scheduler keeps a *counter* for a user transaction to keep track of the number of active version transactions associated with that user transaction. Since there is no dependency between version transactions of a read-only user transaction UT_i, the Method Scheduler passes the version transactions of UT_i to the Version Transaction Manager once it receives them from the User Transaction Manager. Since version transactions of UT_i do not update any data, the Version Transaction Manager simply passes them to the Execution Manager without creating any active versions.

The Execution Manager directly executes the version transactions of a read-only user transaction against the committed versions in the objectbase. For each read operation on data item item x in version transaction VT_{ip}^{f}, if x is a data item of a committed version in object family f, the value of x at time t specified by the user is searched as follows.

1. Sort the committed versions in the version-chain of the object family in ascending order of their validation time.

2. Find the first committed version o^{f_q} in the sorted list in which x is in writeset of o^{f_q} and the validation time of o^{f_q} is less than or equal the requested time t.

Note that once committed versions are sorted based on their validation time, it is possible to make a binary search to find the first committed version in which its validation time is less than or equal to t and from there on, linear search is required up the sorted list to find the first committed version which has x in its writeset. If the value of x at time t is not found among the existing committed versions in the objectbase, the archived committed versions can be similarly searched.

Figure 5.8B illustrates the sorted list of the version-chain shown in Figure 5.8A. The committed versions $o^{f1}, ..., o^{f10}$ have been created from 5:00 o'clock until 6:31. Suppose a user transaction requests the value of b at time 6:15. Using a binary search, it is possible to find that o^{f6} has been the last committed version at 6:15. Starting from o^{f6} and going up the sorted list it is possible to find that UT_3 (the user transaction associated with o^{f3}) has been the last user transaction which has modified x before 6:15.

When version transaction VT_{ip}^{f} of a read-only user transaction terminates, the Execution Manager sends the pre-commit message of VT_{ip}^{f} to the Version Transaction Manager and Version Transaction Manager passes it to the Method Scheduler. Once the Method Scheduler receives the pre-commit message, it decrements the $counter(UT_i)$. If all of the version transactions of the read-only user transaction UT_i are terminated and the commit condition c_i for UT_i has been received by the Method Scheduler, the Method Scheduler sends a $Commit(UT_i)$ message up to the User Transaction Manager. Since temporal transactions do not need to go through the Validation Processor, they may not be aborted unless a transaction failure or a system failure occurs.

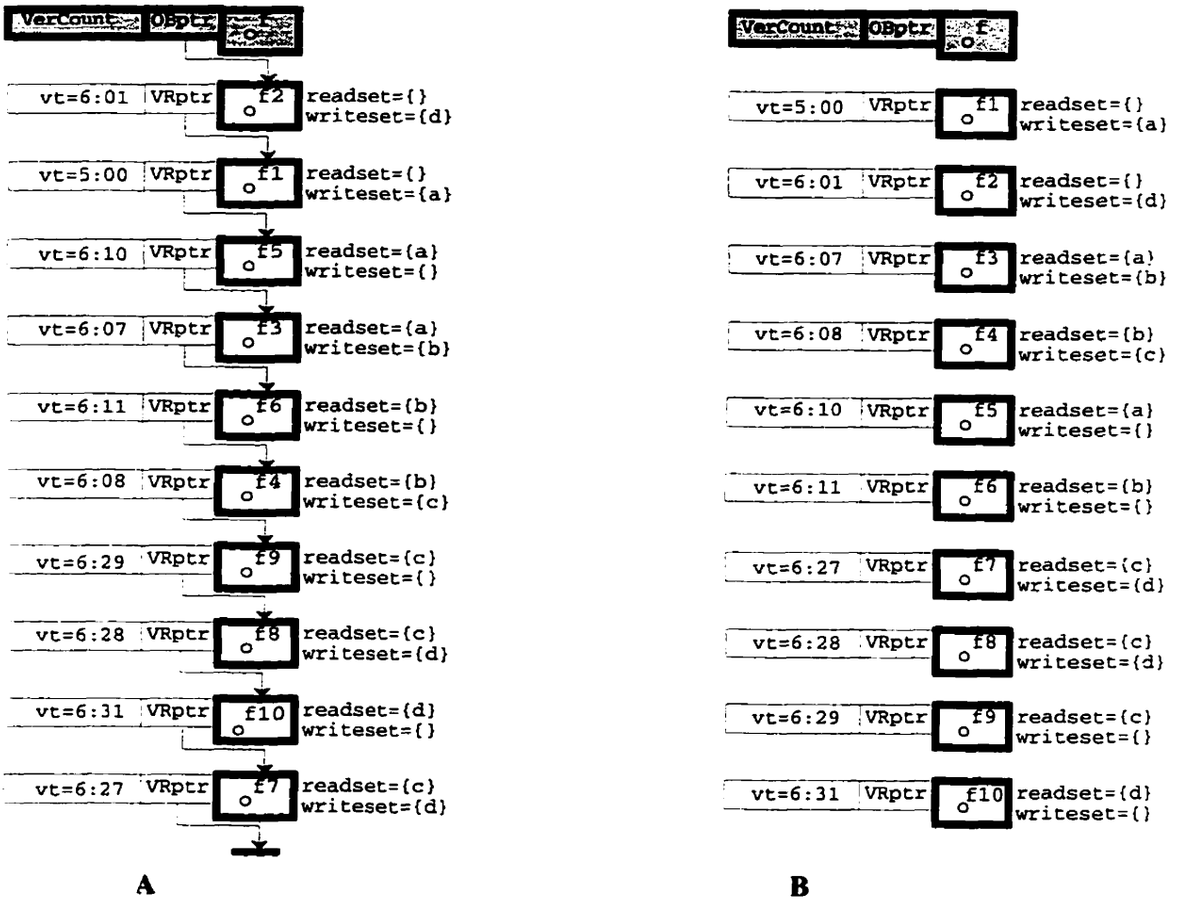


Figure 5.8: Retrieving a data item at a particular time

Chapter 6

Complex Reconciliation

Complex reconciliation will re-execute (partially or entirely) some version transactions of an unsuccessful user transaction. Re-execution is done on the active versions containing the stale data accessed by the version transactions. Complex reconciliation is only attempted when simple reconciliation fails. This chapter begins by giving an example of such a case. Algorithms are developed to detect the stale data and build appropriate reconciliation routines which re-execute the code related to the stale data.

Figure 6.1 shows an example of a case when simple reconciliation of a user transaction UT_i fails and complex reconciliation is required. Figure 6.1A shows the original state of the object family f. UT_1 and UT_4 start first and each receives a copy of o^{f0} denoted by v^{f1} and v^{f4}, respectively. UT_1 executes m_1^f against v^{f1} and UT_4 executes m_4^f against v^{f4}. UT_1 commits and its associated committed version o^{f1} is recorded in the objectbase (Figure 6.1B). Then UT_2 starts, obtains a copy of o^{f1} (v^{f2}), executes m_2^f against v^{f2}, and commits. v^{f2} is promoted to o^{f2} and is recorded in the objectbase (Figure 6.1C). Next UT_3 starts, obtains a copy of o^{f2} (v^{f3}), executes m_3^f against v^{f3} and commits. v^{f3} is promoted to o^{f3} and is recorded in the objectbase (Figure 6.1D). Now UT_4 terminates; however, v^{f4} cannot be committed anywhere in the chain since:

- inserting v^{fi} at the top of the chain is not possible because UT_i reads the stale attribute

Figure 6.1: Situation when Reconciliation is required

c,

- inserting v^{fi} between o^{f3} and o^{f2} is not possible because UT_i reads the stale attribute b,
- inserting v^{fi} between o^{f2} and o^{f1} is not possible because UT_i reads the stale attribute a, and
- inserting v^{fi} between o^{f1} and o^{f0} is not possible because UT_i modifies attribute c which has been accessed by UT_3 in o^{f3}.

Since v^{f4} cannot be inserted anywhere in the version-chain, UT_4 could not commit unless it is re-executed. One solution is to obtain a new copy of the last committed version from the object family f and re-execute m_4^f entirely. This is not feasible because although UT_4 has accessed stale attributes a, b, and c, it has also accessed e, f, g, and h which are not stale. Complete re-execution of m_4^f against v^{fi} re-reads both the stale and the non-stale data wasting time and the resources in the system. Another solution is to do complex reconciliation of UT_i in which the stale data accessed by UT_4 (a, b, c) are properly detected and are passed to a reconciliation procedure called *reconciliM-4-f* associated with method m_4^f. This reconciliation procedure accepts a set of stale data and re-executes only the code related directly or indirectly to the stale data.

6.1 Detecting Stale Data

Recall that the Decision Manager identifies an active version v^{fi} as *invalid* if v^{fi} contains some stale data. The first step of complex reconciliation is to find the stale data accessed by a version transaction executed against an invalid version. This section develops the related algorithms.

Suppose v^{fi} is an active version of object family f referenced by UT_i and $VT_{i1}^f, VT_{i2}^f, ..., VT_{in}^f$ are the version transactions which have executed against v^{fi}. If it is found that v^{fi} is invalid, some of these version transactions executed against v^{fi} may need to be partially re-executed.

A simple case is when only one version transaction VT_{i1}^f has executed against v^{fi}. Suppose $o^{f1}, o^{f2}, ..., o^{fm}$ are the committed versions of o^f which were created during the life time of UT_i.

Recall the definition of *value-intersection* (*value*-$\cap$) from Section 4.1.1. Let $WST^f = \bigcup_{j=1}^{m} writeset(o^{fj})$ be the set of data items which have been changed in the object family f during the life time of UT_i. If $RS(VT_{i1}^f)$ and $WS(VT_{i1}^f)$ are the readset and the writeset of VT_{ip}^f on v^{fi} during the execution, the stale data read by VT_{i1}^f denoted by $staledata(VT_{i1}^f)$ is:

$$staledata(VT_{i1}^f) = RS(VT_{i1}^f)\ value\text{-}\cap\ WST^f$$

Once the stale data are found, they are passed to some appropriate reconciliation procedures and VT_{i1}^f is partially re-executed against v^{fi} which are explained later in this chapter. After partial re-execution of VT_{i1}^f, v^{fi} becomes a valid version and can be promoted to a committed version.

Now suppose the active version v^{fi} has been accessed by two version transactions VT_{i1}^f and VT_{i2}^f. If VT_{i1}^f and VT_{i2}^f do not access value-conflicting operations, the stale date for each of them is found as explained above. If VT_{i1}^f and VT_{i2}^f value-conflict and VT_{i1}^f is processed first, the serialization order will be $VT_{i1}^f \rightarrow VT_{i2}^f$ or vice-versa. Otherwise, the serialization order will be $VT_{i2}^f \rightarrow VT_{i1}^f$. The following considers the case when serialization order is $VT_{i1}^f \rightarrow VT_{i2}^f$.

First, VT_{i1}^f is reconciled. The stale data for VT_{i1}^f are found as:

$$staledata(VT_{i1}^f) = RS(VT_{i1}^f)\ value\text{-}\cap\ WST^f$$

If $staledata(VT_{i1}^f) \notin \phi$, VT_{i1}^f is partially re-executed. Next, VT_{i2}^f is reconciled and the stale data accessed by VT_{i2}^f are found as:

$$staledata(VT_{i1}^f) = RS(VT_{i1}^f)\ value\text{-}\cap\ (WST^f \cup WS(VT_{i1}^f))$$

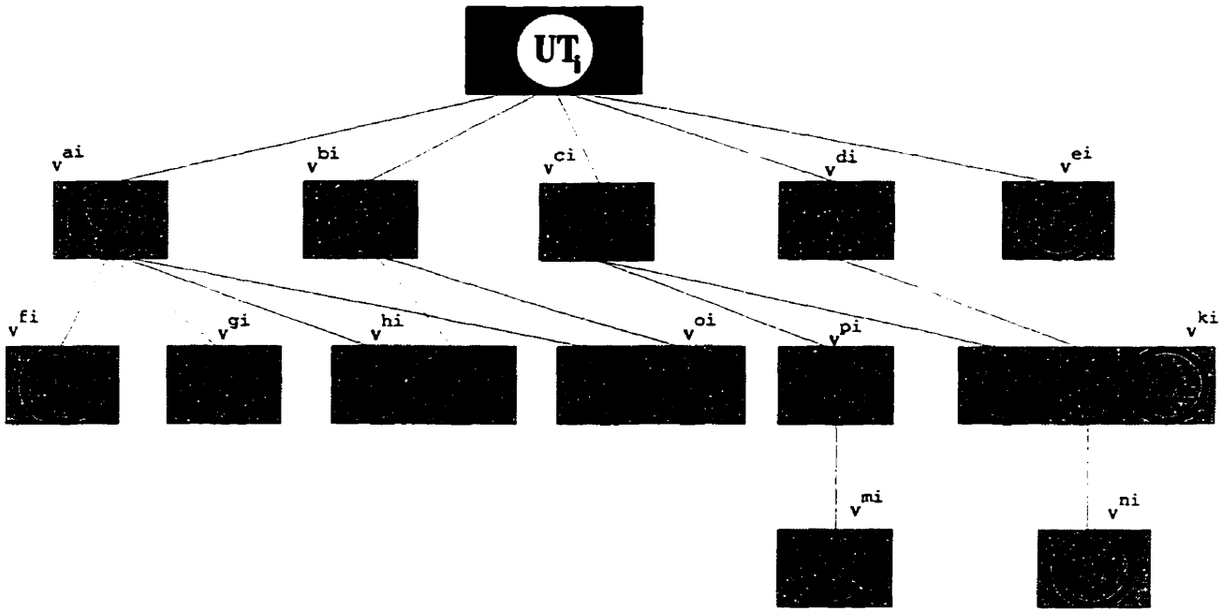


Figure 6.2: An invalid version may effect other valid versions

Note that $WST^f \cup WS(VT^f_{i1})$ determines the changes which have been made by recently committed transactions (WST^f) plus the ones made after the reconciliation of VT^f_{i1} ($WS(VT^f_{i1})$).

In general, to find the stale data for the *kth* version transaction (VT^f_{ik}) where k-1 other value-conflicting version transactions have already been reconciled and partially re-executed on v^{fi} is calculated as:

$$staledata(VT^f_{i1}) = RS(VT^f_{i1})\ value\text{-}\cap$$
$$(WST^f \cup WS(VT^f_{i1}) \cup WS(VT^f_{i2}) \cup ... \cup WS(VT^f_{ik-1}))$$

What to Reconcile?

A version transaction VT^f_{ip} which has accessed some stale data may pass its results to other versions transactions. For example, consider Figure 6.2 where the circles are version transactions, edges are the invocations, and rectangles represent the active versions. Suppose the Decision Manager has found v^{pi} invalid. Thus VT^p_{i12} has accessed some stale data and its results may not be correct. Note that VT^p_{i12} may pass its results to VT^m_{i16} in

```
Procedure Infected-Versions(UT_i)
begin
    for every VT^f_ip invoked directly by UT_i do                                   (1)
        if (v^fi is valid) and (VT^f_ip indirectly access an invalid version) then  (2)
            infected ← infected ∪{v^fi}                                             (3)
    for every two version transactions VT^f_ip and VT^e_iq directly invoked by UT_i do (4)
        if depends(VT^f_ip, VT^e_iq) or depends(VT^e_iq, VT^f_ip) then              (5)
            if v^fi ∈ (infected ∪ invalid) then                                     (6)
                infected ← infected ∪{v^ei}                                         (7)
            else if v^ei ∈ (infected ∪ invalid) then                                (8)
                infected ← infected ∪{v^fi}                                         (9)
    for every version transactions VT^f_ip directly invoked by UT_i do              (10)
        if v^fi ∈ (invalid ∪ infected) then                                         (11)
            Let m^f_j be the method associated with VT^f_ip                         (12)
            for every m^e_k ∈ extent(m^f_j) do                                      (13)
                infected ← infected ∪{v^ei}                                         (14)
end
```

Figure 6.3: Determining the Infected Active Versions Referenced by a User Transaction

v^{mi} and return other results to VT^c_{i3}. Then VT^c_{i3} may use this information, produce other results, and pass them to VT^k_{i13}. Thus beside the version transactions which have executed against some invalid versions, the results of every other version transaction which has been infected by the results of such versions has to be checked during the reconciliation process. If a version transaction VT^f_{ip} executes on an invalid version v^{fi} and passes some incorrect results to another version transaction VT^e_{iq} executing on v^{ei}, v^{ei} becomes an *infected version*. Complex reconciliation is done for version transactions which have accessed either an infected or an invalid version.

The algorithm shown in Figure 6.3 finds the infected versions accessed by the version transactions of UT_i. Note that information such as the *depends* function and *extent* are used to determine how an active version may be infected by an invalid version or another infected version.

6.2 Generating Reconciliation Routines

Partial re-execution is the second step of complex reconciliation. This section develops reconciliation algorithms which only re-execute code related to stale data.

If a statement (s_i) in a method should be re-executed, statements related to s_i should also be found and re-executed. The data dependency relation [Wol89] between the statements in a method is determined by considering the *T*hree *A*ddress *C*ode for the method. Three address code is an encoded form of a program in which all of the complex statements have been decomposed to their simplest form and cannot be decomposed further [ASU86]. Section 6.2.1 explains reconciliation for *simple methods* where routines include simple read and write statements. Then the approach is extended in Section 6.2.2 for *complex methods* which may include loops and other conditional statements.

6.2.1 Simple Methods

Suppose m_k^f is a method of object o^f. $3m_k^f$ denotes the corresponding three address code for m_k^f, and s_i refers to statement i in $3m_k^f$. $RS(s_i)$ and $WS(s_i)$ denote the readset and the writeset of s_i, respectively. If s_i and s_j are two statements in $3m_k^f$, then $s_i < s_j$ indicates that s_i precedes s_j in $3m_k^f$. Similarly, $s_i > s_j$ means s_i follows s_j in $3m_k^f$.

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

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

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

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

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

Figure 6.4 shows Three address code for m_k^f, and the arrays $TACMkf$, $FinalWriteMkf$, and $ReadsFromMkf$. $TACMkf$ is created by the compiler. The values in arrays $ReadsFromMkf$ and $FinalWriteMkf$ can be calculated by scanning through $TACMkf$. The following discusses how to find the values of each element of $bitstringMkf$.

Suppose user transaction UT_i has terminated and it is found that v^{fi}, an active version of UT_i that is accessed by a set of version transactions $VT_{i1}^f, VT_{i2}^f, ..., VT_{in}^f$, is invalid. The following explains how to do partial re-execution of VT_{ij}^f $(1 < j < n)$.

Suppose one of the methods m_k^f $(1 \leq k \leq 5)$ shown in Figure 6.5 is associated with VT_{ij}^f. If s_i:r=p*q is a statement in m_k^f that is either directly or indirectly related to the stale data read by VT_{ij}^f, s_i should be re-executed. Consider an operand $p \in RS(s_i)$. There are four possible cases when the re-execution of s_i may cause the re-execution of other statements:
Case 1: p is neither modified in $s_1..s_{i-1}$ nor in $s_{i+1}..s_n$ (ex: m_1^f is Figure 6.5).
Case 2: p is unmodified in $s_1..s_{i-1}$ but is modified in $s_{i+1}..s_n$ (ex: m_2^f is Figure 6.5).
Case 3: p is modified in $s_1..s_{i-1}$ but is unmodified in $s_{i+1}..s_n$ (ex: m_3^f is Figure 6.5).
Case 4: p is modified both in $s_1..s_{i-1}$ and in $s_{i+1}..s_n$ (ex: m_4^f is Figure 6.5).

In Cases 1 and 2, since p is not modified prior to s_i, a new value for p is re-read 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 r and q. In Case 3, let s_f $(s_1 \leq s_f \leq s_{i-1})$ 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. The value of p created by s_f is available in v^{fi} and is re-read when s_i is re-executed. In Case 4, the correct value of p may neither be available in the objectbase nor in v^{fi}. This is because p is modified both before and after the execution of s_i. Let s_f be the last statement that modifies p prior to s_i, and s_t be the statement that issues the final write on p during the execution. In order to re-execute s_i, s_f must also be re-executed to calculate the value of p that should be read

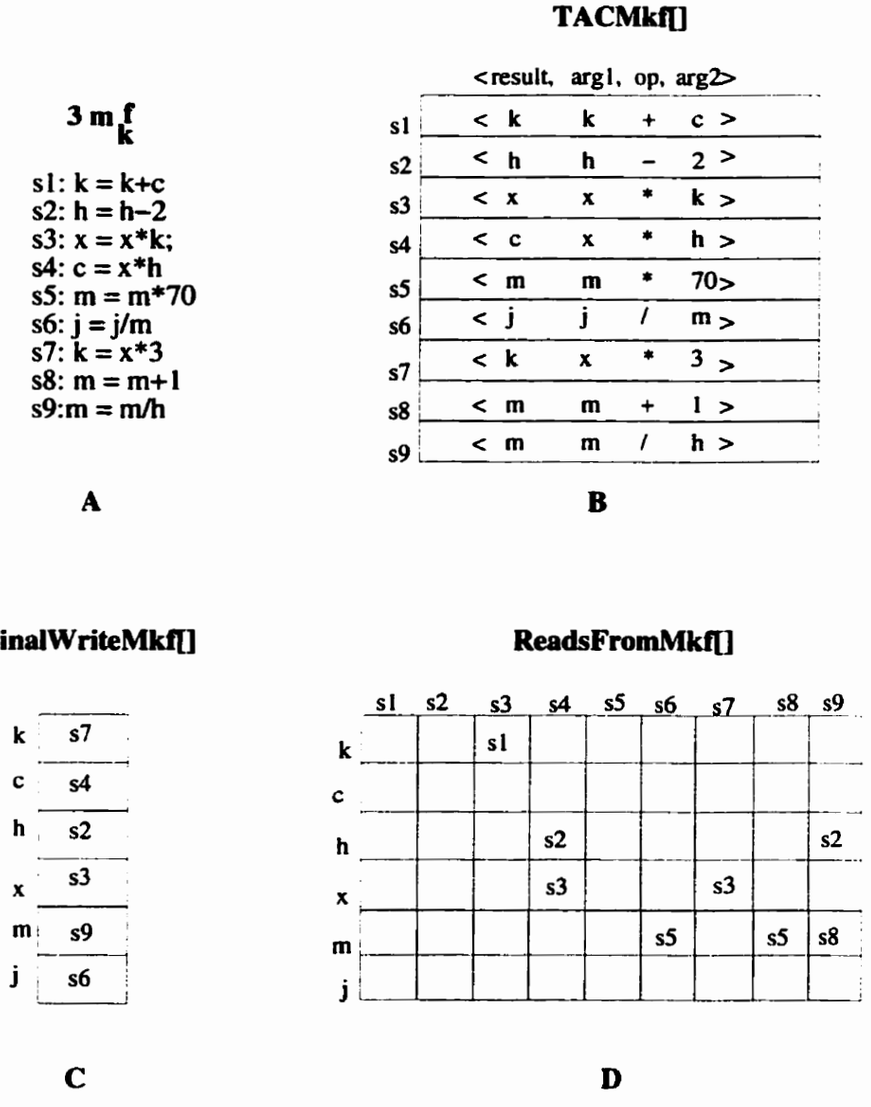


Figure 6.4: The *T*hree *A*ddress *C*ode for m_k^f and Associated Data Structure

m_1^f :	m_2^f :	m_3^f :	m_4^f :	m_5^f :
s_1 :	s_1 :	s_1 :	s_1 :	s_1 : .....
.....	.....	.....	.....	.....
.....	.....	$s_f : p = m * n$	$s_f : p = m * n$	.....
.....	.....	.....	.....	.....
s_{i-1} : ...	s_{i-1} : ...	s_{i-1} : ...	s_{i-1} : ...	s_{i-1} : ...
$s_i : r = p * q$	$s_i : r = p * q$	$s_i : r = p * q$	$s_i : r = p * q$	$s_i : r = p * q$
s_{i+1} : ...	s_{i+1} : ...	s_{i+1} : ...	s_{i+1} : ...	s_{i+1} : ...
.....	.....	.....	.....	.....
.....	$s_t : p = u * v$	.....	$s_t : p = u * v$	$s_t : r = u * v$
.....	.....	.....	.....	.....
s_n :	s_n :	s_n :	s_n :	s_n : .....
Case 1	Case 2	Case 3	Case 4	Case 5

Figure 6.5: The Related Statements

by s_i. However, re-execution of s_f overwrites the value of p that is in the v^{fi}. Thus s_t must also be re-executed so that the correct final value of p can be recorded in v^{fi} when re-execution terminates[1].

Now consider the following two cases for the result operand $r \in WS(s_i)$. In all of the Cases (1-5) shown in Figure 6.5, when s_i is re-executed, every other statement in $s_{i+1}..s_n$ that reads the value of r produced by s_i must be re-executed. Further, in Case 5, since s_t overwrites the value of r that is written earlier by s_i ($s_t > s_i$), the statement that executes the final write on r must also be re-executed.

The Algorithm

Suppose $x \in RS(m_k^f)$ and set $\mathcal{X}$ contains the statements in $3m_k^f$ which read the value of x from the objectbase. If it is found that x is stale, every statement $s_i \in \mathcal{X}$ in addition to those statements in $3m_k^f$ that directly or indirectly relate to each s_i should be re-executed. Pro-

[1]Operation $q \in RS(s_i)$ is handled analogously.

Procedure Bitstring($RS(m_k^f)$)
begin
for every $x \in RS(m_k^f)$ **do** (1)
    $\mathcal{X} \leftarrow$ set of statements in $3m_k^f$ that read x (2)
    for every $s_i \in \mathcal{X}$ **do** (3)
        FindRelatedCode(s_i, $ReadsFromMkf[]$, $FinalWriteMkf[]$, $bitstringMkf[x]$) (4)
end

Figure 6.6: Procedure BitString for Simple Methods

cedures *BitString* and *FindRelatedCode* shown in Figure 6.6 and Figure 6.7 respectively, find the statements which directly or indirectly relate to each variable $x \in RS(m_k^f)$.

First, for each variable $x \in RS(m_k^f)$, procedure *BitString* finds the statements which are related *directly* to x (Figure 6.6 lines 1-2), and then calls procedure *FindRelatedCode* to search for the statements which are *indirectly* related to x (lines 3-4). *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 Figure 6.7). It contains three parts. First statements which s_i reads from are recursively found and marked (lines 1-4). Then s_i is marked (line 5). Finally, given that $r \in WS(s_i)$, statements which read r from s_i are recursively selected and marked for re-execution (lines 6-9). In addition, the last statement that modifies r ($FinalWriteMkf[r]$) is also marked for re-execution (lines 10-12).

Procedure *Reconcil-M-k-f* (see Figure 6.8) accepts a set of stale data and re-executes the code related to the stale data. The bitstrings of all stale data are merged into one bitstring called *string* to find out what code should be re-executed (lines 1-3). A 1 in the i^{th} position of *string* indicates that s_i in $3m_k^f$ should be re-executed (lines 4-6).

6.2.2 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

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

Figure 6.7: FindrelatedCode Procedure for Simple Methods

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

Figure 6.8: The Reconciliation Procedure for Method m_k^f

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

A

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

B

Figure 6.9:

of the *ReadsFromMkf* and *FinalWriteMkf* arrays at compile time for a method m_k^f. For example in Figure 6.9A, the last write operation on p is either s_4 or s_6 depending on whether $a > b$. 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 *conservatively* selected 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 Figure 6.9B, 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 on but we leave this goal as a subject for future research.

Recall from Chapter 3 that a sequence of basic blocks visited during an execution of a method forms a control flow path. Consider a method of object o^f (m_k^f) which contains conditional statements. For each control flow path CFP_{kj} of m_k^f, we define $3CFP_{kj}$ to be the corresponding three address code for CFP_{kj}. $TACCFPkj$, $ReadsFromCFPkj$, and $FinalWriteCFPkj$ are similarly associated with CFP_{kj}. The following additional data

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

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

Figure 6.10: Procedure BitString for Complex Methods

structures are associated with m_k^f:

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

condqueueMkf: is a queue of statements in $3m_k^f$. Every time a statement s_i is marked for re-execution, it is pushed into $condqueueMkf$ if it is within a conditional block. When s_i is popped from the $condqueueMkf$, the begin 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 *BitString* procedure requires a major modification. In Figure 6.6, statements related to each $x \in RS(m_k^f)$ are searched within m_k^f only. This is because m_k^f consists of only a single basic block. This is extended in Figure 6.10 (lines 1-6) where related

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

Figure 6.11: Procedure FindRelatedCode for Complex Methods

statements to each $x \in RS(m_k^f)$ are searched for along all possible control flow paths in m_k^f. In addition, if any statement s_i that is directly or indirectly related to x, is within a conditional block, all statements in that conditional block are considered to be related to x. The *FindRelatedCode* procedure has been slightly modified to find related statements such as s_i and put them in $condqueueMkf$ (Figure 6.11 lines 5-8). The *BitString* procedure pops such statements from $condqueueMkf$, 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 (Figure 6.10 lines 8-15).

Chapter 7

Conclusions and Future Work

The purpose of this research is enhance concurrency in a multiversion environment. Multiversioning in an objectbase systems can also increase concurrency and enhance reliability. The taxonomy presented in Chapter 1 determines the environments where this research is applicable. The three dimensions of the taxonomy are historical, multiversioning, and data complexity. The historical dimension distinguishes the system where no historical record of the data is preserved versus the system where several snapshots of the data are produced as the data changes overtime. The multiversioning dimension shows the difference between the systems where several transient and working versions of data and/or schema can be created against the systems where only a single copy of each data and/or schema exist. The data complexity dimension compares the complexity of data between the object-oriented and conventional data models in terms of the structure and behavior of data.

Chapter 2 reviewed fundamental terminologies such as transaction, history, serializability, recovery, and reliability. The research environment is an objectbase. Object data models are characterized by their support for encapsulation, aggregation, and inheritance. The specific issue of concurrency control in objectbase systems is the key factor in motivating this research.

Chapter 3 presented the model. Two types of transactions were defined: user trans-

actions and version transactions. User transactions cannot directly access the objectbase. Method invocations are converted to version transactions by the system. A version transaction is analogous to the nested transaction model. Versions of objects are created as transactions become active in the system. Versions are maintained in secondary storages. All of the data manipulations of transactions are done against the versions. Eventually, versions may be disposed and purged from the system; or, they become persistent.

The architecture presented in Chapter 4. It contained three components: the Transaction Processor, the Version Processor, and the Validation Processor. The Transaction Processor accepts user transactions from the user, produces a set of versions transactions and passes the version transactions to the Version Processor. The Version Processor creates versions of the objects required by the version transactions, and executes the versions transactions against these versions. When all of the version transactions of a user transaction terminate, the Version Processor notifies the Validation Processor. The Validation Processor checks for the validity of the user transaction. A user transaction is valid if it has not read any stale data during its life time. If this is the case, the user transactions is committed; otherwise, the user transaction is reconciled.

Reconciliation was covered in Chapters 5 and 6. Two types of reconciliation were introduced: simple reconciliation and complex reconciliation. Simple reconciliation is done based on the information captured during the run time and it is an attempt to achieve a serializable schedule by changing the commit order of the currently committed transaction. This change of the commit order is subject to the condition that both intra-object and inter-object serializabilities are guaranteed.

Complex reconciliation is done based on static analysis information. The idea is to find the operations of an unsuccessful user transaction which have been influenced by the stale data and re-execute them. Based on the data dependency analysis, reconciliation procedures are generated for the object methods; thereby, enhancing concurrency in multiversion object systems. Reconciliation procedures accept a set of "incorrect" data items and ensure their consistency. We showed how a compiler can generate the reconciliation procedures based

on static analysis information. In brief, this dissertation made the following contributions:

1. Provided a taxonomy for reasoning about transactions in a multiversion objectbase.
2. Defined a computational model for multiversion objectbases.
3. Defined a new correctness criterion called value-serializability and argued about its feasibility.
4. Detailed the implementation of the *depends* function which has a major role in intra-UT serialization.
5. Proposed a concurrency control algorithm and described its implementation.
6. Proposed two types of reconciliation algorithms, simple reconciliation and complex reconciliation, to ensure that transactions commit in the absence of failure.
7. Argued about the correctness and suitability of the algorithms to the problem.

7.1 Algorithm Enhancements

Both the basic algorithm presented in Chapter 4 and the reconciliation algorithms are subject to improvements. For example, throughout this thesis, stale data are assumed to be detected by the Validation Processor after the transaction terminates. The problem is that a user transaction which reads the stale data at the early stage of its execution continues executing against the incorrect information to the end. Therefore, system time and resources used for the transaction are wasted. The algorithm can be modified to be a mixture of both pessimistic and optimistic approaches. Then it may be possible to detect the stale data before the transaction terminates completely to improve the overall performance.

As another example, recall that a active version of user transaction UT_i can be promoted if it is ensured that UT_i can be committed. Further, UT_i can be committed if the Decision Manager ensures that all of the active versions referenced by UT_i are valid. Thus, when an active version v^{fi} is validated, it must wait for the validation of other versions before it is promoted. During this period, many new transactions may obtain copies of the last committed version from object family f which do not include the changes from v^{fi}.

A	B
s_1 :**while** $(b > 0)$ **do**	s_1 :**while** $(c > 0)$ **do**
{	{
$s_2 : c = c - 1$	$s_2 : c = c - 1$
$s_3 : d = d + 1$	$s_3 : d = d + 1$
$s_4 : b = b - 1$	$s_4 : b = b - 1$
$s_5 : t = t + 1$	$s_5 : t = t + 1$
}	}

Figure 7.1: Re-execution of statements in a loop

One solution is that once a new v^{fj} for transaction UT_j is created, its content is merged with the corresponding active versions of objects for f which have been validated but they are waiting to be promoted. The problem with this approach is that if a failure occurs cascade aborts may be required. However, if we assume that failures do not occur frequently, the advantages of this approach may overweight its disadvantages.

Reconciliation algorithms may also be improved. In simple reconciliation, we have shown that a new committed version is inserted somewhere in the version-chain of its corresponding object family. Insertion does not change the order of the other committed versions in the chain with respect to each other. Recall that simple reconciliation cannot be performed if an active version can be inserted anywhere in the version-chain of its corresponding object family. However, it may be possible to insert the active version in the chain, if some of the committed versions in the chain are re-ordered; thereby, enhancing concurrency.

Complex reconciliation assumed that if a code in a conditional block should be re-executed, every code in the block should also be re-executed. This is unnecessarily restrictive. Consider Figure 7.1. Suppose variable b is stale and every code related directly and indirectly to b must be re-executed. In Figure 7.1A, the number of iterations of the *while* loop depends on the value of b. Therefore, if the number of loop iterations changes during the re-execution, it may effect the result of the operation of other statements which may not

even be related to b. On the other hand, Figure 7.1B shows that the number of iterations in the loop is determined by the value of c. Note that c is not related to the stale data b. Then as long as c is not stale, the number of iterations during the re-execution remains as it had during execution. If re-execution does not change the number of iterations in the loop, it is not necessary to re-execute statements s_2, s_3, and s_5, when b is detected to be the only stale data. The complex reconciliation algorithm should be improved to reduce the unnecessary overhead involved in the re-execution of such a conditional blocks.

7.2 Future Work

Implementation

Although we provided the pseudo code for most of the routines in this thesis, the actual implementation remains a topic of the future research. Before the system presented in this thesis is implemented optimizing compiler where information from static analysis and issues related to data dependency can be obtained, should be built.

Recovery

Beside concurrency control, another important aspect of transaction management is reliability and recovery. Usually, when we attempt to enhance the concurrency control algorithm, reliability is neglected because it is considered orthogonal. However, recovery algorithm should be provided to guarantee reliable transaction executions.

In traditional database system, the scope of the recovery is the entire database. When a failure occurs, new transactions cannot be submitted to the system unless the database is back to a consistent state. This system permits both transaction nesting and version tools be used to limit the scope of the recovery. Therefore, in case a failure occurs, new transactions can execute while recovering other user and/or version transactions.

Since we have two levels of abstractions, user and system level, the computational model may be improved to provide definitions for recoverable, avoid cascading aborts, and strict

user transactions as well as definitions for recoverable, avoid cascading aborts, and strict version transactions.

One obvious recovery solution is that it is possible to purge the active versions in the unstable store and restart the transactions. This might not always be a feasible solution when the transactions are already terminated and are being validated by the Validation Processor. Therefore, some form of recovery scheme may be required. For example:

- logs to coordinate the execution of user transactions,
- logs to coordinate the execution of version transactions of a particular user transaction, and
- logs to coordinate the execution of version transactions accessing a specific active version.

The recovery algorithms provided for advanced transaction models may provide solutions to the reliability and recovery issues [Mos87, RM89, Wie94].

Other Areas

Some work in this thesis can be extended and combined with other areas of research. For example, as it is done in temporal databases, we have added validation time [SA86] to the committed versions. Thus queries can be issued on past data to provide the user with historical information.

Further, in the taxonomy presented in Chapter 1, it is possible to develop a system where multiple versions of schema can be supported. Throughout this thesis the schema remains unchanged during the run time. This problem may be solved by having more than one version of schema. The active transactions continue their execution against the old schema, and the new transactions can be executed against the new schema. The transaction management should be improved to coordinate the serialization order between the transactions which are running under different versions of the schema.

Finally, our computational model and the architecture can be improved to be used for distributed environments. In some distributed systems, data is replicated in different sites.

Eventually, because of a problem such as a node or a link failure, the data at some sites may become stale. Thus some type of reconciliation may be required to make the states of these sites consistent with the states of other sites in the database.

Bibliography

[AA92] D. Agrawal and A. El Abbadi. A Non-Restrictive Concurrency Control for Object-Oriented Databases. In *Proceedings of 3rd International Conference on Extending Database Technology, Vienna, Austria*, pages 469–482, 1992.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-oriented Database System Manifesto. In *Proceeding of First International Conference Deductive and Object-Oriented Databases*, pages 40–57. Elsevier Science Publisher, B.V. Amesterdam, 1989.

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[Bad79] D. Badal. Correctness of Concurrency Control and Implementations in Distributed Databases. In *IEEE Proceeding of COMPSAC Conference*, pages 588–593. IEEE. November 1979.

[BB89] F. Bancilhon and P. Buneman. Version Control in an Object–Oriented Architecture. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 451–458. Addison–wesley, Reading, Mass., 1989.

[BBG89] C. Beeri, P. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of Association for Computing Machinary*, 36(2):230–269, 1989.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented Database Systems, the story of O_2.* Morgan Kaufmann Publishers, San Mateo, California, 1992.

[BG81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[BG83] P.A. Bernstein and N. Goodman. Multiversion Concurrency Control–Theory and Algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[BGH87] P. Bernstein, N. Goodman, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[BHR80] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. *ACM Transactions on Database Systems*, 5(2):139–156, 1980.

[BM76] J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications.* American Elsevier Publishing CO., INC., 1976.

[BM91] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4):33–47, April 1991.

[BOH+91] A. Buchmann, T. Ozsu, M. Hornick, D. Geogakopoulos, and F. Manola. A Transaction Model for active Distributed Object Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 5, pages 123–158. Morgan Kaufmann Publishers, 1991.

[Car83a] M. Carey. *Modeling and Evaluation of Database Concurrency Control Algorithms.* Ph.D. dissertation, Computer Science Div.(EECS), Univ. of California, Berkeley, 1983.

[Car83b] M. Carey. Multiple Versions and Performance of Optimistic Concurrency Control. Technical Report 517, University of Wisconcin-Madison, October 1983.

[Car87] M. Carey. Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions. *IEEE Transactions on Software Engineering*, 13(6):746–751, June 1987.

[Car89] M. Carey. Storage Management for Objects in Exodus. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–369. Addison-wesley, Reading, Mass., 1989.

[CCHK90] C. Callahan, A. Carle, M. Hall, and K. Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[CFLN82] A. Chan, S. Fox, W. Lin, and A. Nori. The Implementation of an Integrated Concurrency Control And Recovery Scheme. In *Proceeding of the ACM SIGMOD International Conference on the Management of Data*, pages 184–191, 1982.

[CG85] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, February 1985.

[CK86] H. Chou and W. Kim. A Unifying Framework for Version Control in a CAD. In *Proceeding of the 12th International Conference on VLDB*, pages 336–344. Microelectronics and Computer technology Corporation, 1986.

[CM86] M. Carey and W. Muhanna. The Performance of Multiversion Concurrency Control Algorithms. *ACM transactions on Computer Systems*, 4(4):338–378, November 1986.

[DE89] W. Du and A. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of Very Large Data Bases (VLDB)*, pages 347–355, 1989.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624 - 632, November 1976.

[FLMW90] A. Fekete, N. Lynch, M. Merrit, and W. Weihl. Commutativity-based Locking for Nested Transactions. *Journal of Computer and System Sciences*, 41(1):65–156, 1990.

[GB94a] P. Graham and K. Barker. Enhancing Intra-transaction Concurrency in Object Bases. *Journal of computing and information*, 1(1):795–811, May 1994.

[GB94b] P. Graham and K. Barker. Effective Optimistic Concurrency Control in Multiversion Object Bases. In *Proc. International Symposium on Object Oriented Methodologies and Systems (ISOOMS)*, volume 858, pages 313–328. In Springer-Verlag Lecture Notes in Computer Science, September 1994.

[GB95] P. Graham and K. Barker. Improved Scheduling in Object Bases Using Statically Derived Information. *The International Journal of Microcomputer Applications (IJMA)*, 14(3):114–122, 1995.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[GR94] Gray and Reuter. *Transaction Processing Concepts.* Addison-Wesley Publishing Company, 1994.

[Gra81] J. Gray. The Transaction Concept: Virtues and Limitation. In *Proceedings of the 7th International Conference on VLDB*, pages 144–154, 1981.

[Gra94] P.J. Graham. *Applications of Static Analysis to Concurrency Control ans Recovery in Objectbase Systems.* Ph.D. thesis, University of Manitoba, 1994.

[GZB92] P. Graham, M. Zapp, and K. Barker. Concurrency Control in Object-Based Systems. Technical Report technical report: 92–07, University of Manitoba, June 1992.

[HB96] A. Hadaegh and K. Barker. Value-serializability and an Architecture for Managing Transactions in Multiversion Objectbase Systems. In *Proceeding of third international workshop on Advances in Databases and Information Systems, Moscow*, pages 126 -133. September 1996.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2–24, 1991.

[HP86] T. Hadzilacos and C. Papadimitriou. Algorithmic Aspects of Multiversion Concurrency Control. *Journal of Computer and System Sciences*, 3(3):297–310, 1986.

[HPC93] A.R Hurson, S.H. Pakzad, and J. Cheng. Object-Oriented Database Management Systems: Evolution and Performance Issues. *IEEE Computer Society Press. [First appeared in IEEE Computers*, 26(2):48–60, 1993.

[HR87] T. Harder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *IBM Research Report RJ 5803 (58533), Almaden Research Center*, 1987.

[HR93] T. Harder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39–74, 1993.

[KC88] W. Kim and H. Chou. Versions of Schema for Object–oriented Databases. In *Proceedings of the 14th International Conference on VLDB*, pages 148–159, 1988.

[KGBW90] 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.

[KM94] H. Kwon and S. Moon. Reverse Serializability as a Correctness Criterion For Optimistic Concurrency Control. *Microprocessing and Microprogramming*, 40(10–12):759–762, December 1994.

[KR81] H.T. Kung and J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213 - 226, June 1981.

[Lau83] G. Lausen. Formal Aspects of Optimistic Concurrency Control In a Multiple Version Database System. *Information Systems*, 8(4):291–301, February 1983.

[Moh90] 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.

[Mor93] T. Morzy. The Correctness of Concurrency Control for Multiversion Database Systems with Limited Number of Versions. In *Proc. of 9th Int. Conf. on Data Engineering*, pages 595–605, Vienna, 1993. IEEE Computer Society Press.

[Mos85] J.E.B. Moss. *Nested Transactions - An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[Mos87] J.E.B Moss. Log–based Recovery for Nested Transactions. In *Proceeding of the 13th International Conference on VLDB*, pages 427–432, 1987.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *Proceeding of ACM SIGMOD Int. Conf. on Management of Data*, pages 124–133, IBM Almaden Research Center, San Jose, CA 95120, USA, 1992.

[Nak92] T. Nakajima. Commutativity Based Concurrency Control for Multiversion Objects. In *Proceedings of the International Workshop on Distributed Object Management*, pages 101–119, 1992.

[NRZ92] M.H. Nodine, S. Ramaswamy, and S.B. Zdonik. A Cooprative Transaction Model for design Databases. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 3. Morgan Kaufmann Publishers, 1992.

[OS91] P. Butterworth A. Otis and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10):64–77, October 1991.

[Özs94] M.T. Özsu. Transaction Models and Transaction Management in Object-Oriented Database Management Systems. In A. Dogac, M.T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, volume 130, pages 147–184. NATO ASI Series: Springer-Verlag, 1994.

[Pap86] C.H. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[PK84] C. Papadimitriou and C. Kanellakis. On Concurrency Control by Multiple Versions. *ACM Transactions on Database Systems*, 9(1):89–99, 1984.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for Open Ended Activities. In *Proceedings of the 14th International Conference on VLDB*, pages 26–37, August 1988.

[RA92] R.F. Resende and A. El Abbadi. A Graph Testing Concurrency Control for Object Bases. *NSF IRI-917904*, 1992.

[Ree78] D. Reed. Naming and Synchronization in a Decentralized Computer System. Technical Report MIT/LCS/TR-205, MIT Laboratory for Computer Science, 1978.

[RGN90] T.C. Rakow, J. Gu, and E. Neuhold. Serializability in Object-Oriented Database Systems. In *CH2840-7/0000/0112@1990 IEEE*, pages 112–120, Dolivostrabe 15, D–6100 Darmstadt, West Germany, 1990. Integrated Publication and Information Systems Institute (IPSI).

[RH90] M. Stonebraker L. Rowe and M. Hirohama. The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–141, March 1990.

[RKS93] R. Rastogi, H. Korth, and A. Silberschatz. Strict Histories in Object–Oriented Database Systems. *ACM PODS*, pages 288–299, 1993.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-ahead Logging for Nested Transactions. In *Proceedings of the 15th International Conference on VLDB*, pages 337–346, 1989.

[Ryd79] B. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 5(3):216–225, 1979.

[SA86] R. Snodgras and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, 1986.

[SLR76] R. Stearns, P. Lewis, and D. Rosenkrants. Concurrency Control for Database Systems. In *IEEE Conference of Foundation of Computer Science*, pages 19–32, November 1976.

[SR81] R.E. Stearns and D.J. Rosenkrantz. Distributed Database Concurrency Control Using Before-Values. In *Proceeding of the ACM-SIGMOD Conference on Management of Data*, pages 74–83, June 1981.

[SRK91] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using Polytransactions to Manage Interdependent Data. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 14, pages 555–581. Morgan Kaufmann Publishers, 1991.

[Sta94] T.A. Standish. *Data Structure Algorithm, and Software Principles.* Addison-Wesley Publishing Company, 1994.

[Wei88] W.E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488 - 1505, 1988.

[Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[Wie94] C. Wieler. *Reliable and Recoverable Transactions in Object Bases.* Master's thesis, University of Manitoba, 1994.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, 1990.

[Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* . MIT Press, 1989.

[WS91] G. Weikum and H. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models*, chapter 13, pages 516–553. Morgan Kaufmann Publishers, 1991.

[WYC93] K. Wu, P. Yu, and M. Chen. Dynamic Finite Versioning: An Effective Versioning Approach to Concurrent Transaction and Query Processing. *1063-6382/93 IEEE*, pages 577–586, 1993.

[Zap93] M. Zapp. Concurrency Control in Object-Based Systems. Technical Report technical report: 93–03, University of Manitoba, July 1993.

[ZB93a] 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.

[ZB93b] M.E. Zapp and K. Barker. On Concurrency Control in Object Bases. In *Mid-Continent Information Systems Conference (MISC'93)*, pages 91–97, Fargo, USA, May 1993.

[ZB93c] M.E. Zapp and K. Barker. The Serializability of Transaction in Object Bases. In *Proceedings of the International Conference on Computers and Information*, pages 428–432, Sudbury, Canada, May 1993.