# An Assertional Characterization of Serializability and Locking*

Ernest R. McCurley
Ph.D. Thesis

88-894
January 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This material is based on work supported in part by the Office of Naval Research under contract N00014-86-K-0092 and the National Science Foundation under Grant No. CCR-8701103. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of the Office of Naval Research or the Nationa Science Foundation.

# AN ASSERTIONAL CHARACTERIZATION OF SERIALIZABILITY AND LOCKING

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ernest Robert McCurley

January 1988

An Assertional Characterization of Serializability and Locking

Ernest Robert McCurley, Ph.D.

Cornell University 1988

The problem of synchronizing transactions in a database system so that concurrent execution transforms the system from one consistent state to another is called the Concurrency Control Problem. Over the past 20 years, a property of concurrent execution called serializability has evolved as a universal paradigm for solving the Concurrency Control Problem. Up until now, most work on serializability has been characterized by an emphasis on sequences of operations. Researchers studying programming logics and methodologies have developed a different approach to characterizing the semantics of concurrent programs. This approach is called assertional reasoning, and emphasizes the system state instead of sequences of operations. This dissertation describes the extension of the formalisms and tools of assertional reasoning to the Concurrency Control Problem.

Proposed is a definition of serializability that generalizes previous definitions in many respects. Two methods are described by which this definition of serializability can be specified in an assertional programming logic using formulas called proof outlines. As a consequence of specifying serializability with proof outlines, it becomes

possible to formally verify serializability. The use of an assertional programming logic eliminates the need to explicitly consider transaction interleavings, simplifying verification. Another consequence of specifying serializability with proof outlines is the ability to derive synchronization protocols for serializability. This possibility is realized in the form of a method for deriving locking protocols from assertional specifications. The method is based on a novel view of locking, in which locks held by transactions reflect properties of the system state. Using this method, semantic information available during the derivation process can be used to obtain locking protocols permitting greater concurrency among transactions than locking protocols obtained by more traditional methods. Examples are given throughout the dissertation to illustrate the methods described.

# Biographical Sketch

Rob McCurley was born on July 5th, 1958, in a suburb of Los Angeles, California, third child and only son of loving parents Ernest and Alice McCurley. In his first year of life, he moved with his parents to Atlanta, Georgia, where he grew up in the heart of the New South. He graduated from Peachtree High School in the spring of 1976.

In the fall of 1976, he enrolled at the University of Georgia. There, he studied mathematics and fell in love with his future wife, Janet Bless. The university awarded him a B.S. in Mathematics, magna cum laude, in spring of 1980.

He enrolled as a graduate student at Cornell University in the fall of 1980, where he studied the science of computers and learned the true meaning of winter. On the happy day of March 27, 1982, he and Janet were married. Cornell awarded him a M.S. in Computer Science in 1983.

To Janet.

# Acknowledgements

I would like to thank Fred Schneider for his immeasurable contributions to both the form and content of this dissertation. As my advisor, Fred has spent much time and effort trying to teach me how research is conducted. I hope he realizes how much I appreciate his patience. I would also like to thank the other members of my special committee, John Hopcroft and Richard Shore, for their time and helpful comments on early drafts of this dissertation.

I am grateful for the support of the Office of Naval Research under contract N00014-86-K-0092 and for the support of the National Science Foundation under Grant No. CCR-8701103 during the course of this research.

There are many people I would like to thank for their friendship. In particular, I would like to thank Tim and Jeanne Lansing for moral support, traveler's assistance, and restraint in harassing me; Doug and Maureen Howe for many pleasurable evenings of dinner and conversation; and Jacob Aizikowitz for listening to my musings and grumblings about research and other topics.

Finally, I would like to thank my family, especially my parents. Though they must have thought I was crazy at times, they loved and supported me when I doubted myself. However, the person I am most grateful to is my wife Janet. Her love and encouragement have meant everything to me; without them, I would have given up long ago.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

Many computer applications involve information that must be stored, retrieved, and modified. For example, a bank must maintain customer account balances and update them as deposits and withdrawals are made; a university must record information about course offerings and student grades.

*Database systems* are computer systems that store and maintain large amounts of information. Information in a database system is typically stored on magnetic disk storage devices rather than in primary memory because of their high capacity for data storage and relative involatility. It is accessed through one or more processors connected to these storage devices.

Information stored in a database can be viewed as modeling some aspect of the application it supports. For example, a banking database system might store a list of numeric values to model balances of customer accounts. As events, such as deposits and withdrawals, transform the application state, the database state is transformed accordingly by running programs called *transactions*.

The correspondence between an application and a database system imposes certain restrictions on the database system state. A bank might require account balances to be non-negative, which restricts the stored values that model these balances to be non-negative. Restrictions imposed by an application on the database system state are called *consistency constraints*. A consistency constraint can be thought of as a predicate on the database system, although in practice such predicates are often too complex to be written explicitly. States that satisfy the consistency constraint are called *consistent states*.

Database systems are started in a consistent state and transactions are constructed so that they model the events to which they correspond, thereby guaranteeing that each transaction individually will transform the system from one consistent state to another. A *serial execution* of transactions is one in which transactions are executed one at a time, starting one only after the preceding one competes. By a simple inductive argument on the number of transactions, any serial execution will transform the system from one consistent state to another.

*Concurrent execution* of transactions, in which one or more transactions are started before previous ones complete, has an advantage over over serial execution. In some systems, a large portion of transaction execution time is spent waiting for responses from relatively slow I/O devices (such as a user terminals or storage devices). By running transactions concurrently, the time that one transaction spends waiting can be used to run operations from another transaction that is not waiting, thereby increasing the rate at which transactions are processed.

Unfortunately, without synchronization, concurrently executed transactions can interleave in ways that leave the database in an inconsistent state. The problem of synchronizing transactions so that concurrent execution transforms the system from one consistent state to another is called the *Concurrency Control Problem.* Over the past 20 years, a property of concurrent execution called *serializability* has evolved as the basis for solving the Concurrency Control Problem. Until now, however, most work on serializability has been characterized by an emphasis on *sequences of operations.* The definition of serial execution of transactions is an example of this style of characterization. The view that locking protocols exclude operations from executing concurrently is another example.

A different approach to analyzing the semantics of both sequential and concurrent programs has been developed by researchers studying programming logics and methodology. The approach is called *assertional reasoning* and emphasizes *system states* rather than operation sequences. This thesis describes the application of assertional reasoning to database systems. We give an assertional characterization of serializability; it generalizes previous definitions of serializability. Our approach to defining serializability not only allows the correctness of synchronization protocols for serializability to be proven formally, but also allows semantics of an application to be incorporated into the *derivation* of synchronization protocols that allow a high degree of concurrency among transactions. We illustrate this benefit by giving an assertional characterization of locking and a method for deriving locking protocols from specifications.

## 1.1 Consistency and Concurrency

A simple example illustrates the Concurrency Control Problem. Consider a database system that models bank accounts numbered from 0 to $N$. The database stores account balances as values in an array $a[0..N]$, with $a[i]$ holding the balance of account number $i$. Another variable $ba$ holds the value of bank assets. As is typical of database systems, these values are stored on magnetic disk.

Disk drives typically provide two types of operations for accessing values: *read* and *write*. Let $r(x,t)$ denote a read operation that copies the value of $x$ (stored on disk) into a computer memory location denoted $t$; let $w(x,e)$ denote a write operation that evaluates expression $e$ involving values in computer memory and copies the resulting value back to $x$ on disk.

The requirement that bank assets match the amount deposited in accounts induces a consistency constraint $ba = \sum_{0 \leq i \leq N} a[i]$, specifying that $ba$ equals the sum of values in $a[0..N]$. As customers make deposits and transfer funds within accounts, transactions must be run to update the values in $a[0..N]$ and $ba$ while leaving the system in a consistent state.

Using read and write operations, the transaction $DEP(a[i],x)$ of Figure 1.1 increments $a[i]$ and $ba$ to reflect a deposit of amount $x$ to that account. The transaction reads the balance of $a[i]$ into memory and writes the updated balance back, subsequently updating $ba$ in the same way to ensure that the consistency constraint will hold afterwards. In a similar manner, transaction $INT(a[j],y)$ of Figure 1.1 increments $a[j]$ and $ba$ by $y * a[j]$, reflecting the accumulation of interest at rate $y$ by account $j$.

$$\begin{array}{llll} DEP(a[i],x): & r(a[i],t0); & INT(a[j],y): & r(a[j],t2); \\ & w(a[i],t0+x); & & w(a[j],t2+y*t2); \\ & r(ba,t1); & & r(ba,t3); \\ & w(ba,t1+x) & & w(ba,t3+y*t2) \end{array}$$

Figure 1.1: Deposit and Interest Transactions.

Suppose that a deposit of $d$ to account $s$ is made at about the same time interest at rate $r$ is begin credited to that account. If $DEP(a[s],d)$ and $INT(a[s],r)$ run concurrently and without synchronization, transaction operations can interleave in the following order:

$$\begin{array}{ll} \sigma 0: & r(a[s],t0); \\ & r(a[s],t2); \\ & w(a[s],t0+d); \\ & w(a[s],t2+r*t2); \\ & r(ba,t1); \\ & w(ba,t1+d); \\ & r(ba,t3); \\ & w(ba,t3+r*t2). \end{array}$$

A sequence of transaction operations like $\sigma 0$ that denotes an interleaving resulting from concurrent execution is called a *schedule*. The use of the statement composition operator ";" between operations allows the schedule to be viewed as a sequential program having the same effect as the particular concurrent execution it is modeling. When concurrent execution produces schedule $\sigma 0$, the update $w(a[s],t0+d)$ by $DEP(a[s],d)$ is overwritten by $INT(a[s],r)$, effectively losing the deposit into $a[s]$. As a consequence, $\sigma 0$ will leave $ba = d + \sum_{0 \le i \le N} a[i]$, an inconsistent state.

## 1.2 Serializable Schedules

As $\sigma 0$ illustrates, not all schedules in which transactions interleave transform a database system from a consistent to an inconsistent state. One type of schedule that preserves consistency is a serializable schedule. A *serializable schedule* is one that "behaves like" some *serial schedule*—a schedule that denotes a serial execution of transactions.[1] Since serial execution of transactions transforms a database from one consistent state to another, execution resulting in a serializable schedule will do so as well.

An example of a serializable schedule of $DEP(a[s],d)$ and $INT(a[s],r)$ is

$$\begin{aligned}\sigma 1:\ & r(a[s],t0);\\ & w(a[s],t0+d);\\ & r(a[s],t2);\\ & w(a[s],t2+r*t2);\\ & r(ba,t1);\\ & w(ba,t1+d);\\ & r(ba,t3);\\ & w(ba,t3+r*t2).\end{aligned}$$

For any given initial values of $a[0..N]$ and $ba$, $\sigma 1$ leaves the same final values as the serial schedule

[1] We describe more formally what it means for one schedule to "behave like" another in Section 1.3.4.

$\sigma 2$: $r(a[s], t0);$
$w(a[s], t0 + d);$
$r(ba, t1);$
$w(ba, t1 + d);$
$r(a[s], t2);$
$w(a[s], t2 + r * t2);$
$r(ba, t3);$
$w(ba, t3 + r * t2).$

The consistency-preserving properties of serializable schedules imply that the Concurrency Control Problem can be solved by synchronizing transactions to ensure that every schedule is serializable.

## 1.3 Related Work

A great deal of research has been published about serializability. Several different database models have been considered and several different definitions of serializability have been proposed.

### 1.3.1 Operation Types

One way in which database system models differ is in the types of operations that can be used to construct transactions. Many models [BBGLS83,BG83,BG81,BSW79, GW82,G83,G78,P79,R83,SLR76,TS85,Y84] assume that transactions are constructed from read and write operations as the ones described in Section 1.1 were. This reflects the use of storage devices, such as disks, that implement these operations in hardware. More recently, models have been devised for systems that support operations other than read and write. For example, a model with operations that traverse and manipulate search structures is considered in [GS85] and one with operations on abstract data types such as queues and sets is considered in [SS84] and [W84]. The model of [K83] does not place any restrictions at all on the operations from which transactions are constructed. Although system models with a greater variety of operations tend to make analysis of concurrent execution more complicated than in read-write models, they do make it possible to describe more accurately the semantics of concurrent execution.

Many models also make assumptions about the way operations are ordered within transactions. In [P79] and [BSW79], for example, transactions consist of a single read operation followed by a single write operation, each of which accesses several variables at the same time. In [Y84], transactions can contain several read and write operations but the operations are assumed to be ordered so that no transaction writes to the same variable twice or reads a variable it has previously written. Such restrictions on transaction structure simplify analysis.

In addition to assumptions about organization, different models make different assumptions about the degree to which semantics of individual operation are known. In [P79], only the set of variables accessed by a write operation is considered when analyzing its behavior; the function used to compute the value it stores is left unspecified. The same is true in [Y84]. In contrast, the models of [SS84] and [W84] specify not only the variables that operations access, but also details of how these operations transform these variables from one state to another. As with restrictions on operation type and order, weaker assumptions about operation semantics simplify analysis of concurrent execution. However, models that make stronger assumptions about semantic information allow use of this information when deriving synchronization for serializability, usually allowing more concurrency than models that make weaker assumptions.

### 1.3.2 Transaction Synchronization

Another area of difference in various database system models is the way in which transaction synchronization is represented. In some models, synchronization is *implicit*—transactions do not execute synchronizing operations directly but send *requests* for operations to a system process called a *scheduler* [P79]. The scheduler considers the history of requests when deciding whether to delay or grant a pending request. An example of implicit synchronization is *timestamp ordering* [BG81], in which a *timestamp* is assigned to each transaction as it begins to execute. Each request submitted to the scheduler is marked with the timestamp of the transaction submitting it. The scheduler then uses these timestamps to order requests. In other database system models, transaction synchronization is *explicit*—synchronizing operations appear among transaction operations for manipulating data.

### 1.3.3 Locking

A form of synchronization used in many database system models is *locking* [G78,K83, Y84,KS79]. In database systems that use locking for synchronization, transactions *acquire* and *release* entities called *locks*. In some systems using locking, transactions explicitly execute operations to acquire and release locks, while in others, locks are acquired and released implicitly as transactions execute operations. A *locking protocol* characterizes how locks can be used to synchronize transactions. A locking protocol specifies

- a set of possible *modes*, or types, that locks can have,
- a *compatibility relation* indicating what locks can be held concurrently, and
- a set of *locking rules* transactions must follow when acquiring and releasing locks.

Synchronization results from mediation of lock acquisition and release requests according to the compatibility relation, delaying requests that are inconsistent with the compatibility relation.

Previous research on synchronization in database systems has focused on developing locking protocols that allow as much concurrency as possible among transactions, but restrict possible schedules to serializable ones. Several protocols have been proposed. One area of difference between them is the set of lock modes assumed. The set of lock modes is usually derived from the set of operations that the database system model permits. Each lock mode typically specifies the operation with which it is associated and the process that has acquired it.

Lock compatibility relations have traditionally been derived from the semantics of the operations with which they are associated. Exclusive locks are used whenever the net effect of concurrently executing transactions can depend on how operations of a particular type interleave. For example, the value left in $a[s]$ by transactions $DEP(a[s],d)$ and $INT(a[s],r)$ of Section 1.1 depends on how the write operations $w(a[s],t0+d)$ and $w(ba,t1+d)$ in $DEP$ interleave with $w(a[s],t2+r*t2)$ and $w(ba,t3+r*t2)$ in $INT$. Consequently, the lock mode associated with write operations would by exclusive. If both transactions consisted of only read operations, every interleaving would produce the same result, which implies that the lock mode associated with read operations need not be exclusive.

Locking rules for acquiring and releasing locks generally require a transaction to have acquired and not yet released a lock for an operation before it can execute that operation. Several additional restrictions on lock acquisition and release have been proposed. For example, lock acquisition and release is often required to be *two-phase* [EGLT76], which means that a transaction never acquires additional locks once it has released any lock. This divides transaction execution into a lock acquiring phase and a lock releasing phase.

A two-phase locking rule is shown to be sufficient to guarantee only serial schedules for the model used in [EGLT76]. The necessity of two-phase locking in the absence of restrictions on transaction structure is also discussed there. In models where more is known about the structure of access to data, locking rules that are not two-phase have been proposed. In [BS77], for example, a protocol for transactions that traverse and modify B-trees that does not obey the two-phase restriction on lock acquisition and release is presented. This approach is generalized in [GS85] to obtain locking rules that are not two-phase when transaction operations are structured to traverse more general types of linked data structures.

### 1.3.4 Definitions of Serializability

As with database system models, several different definitions of serializability have been proposed. These definitions differ primarily in the formal definition of when a schedule "behaves like" a serial schedule.

One of the earliest formal definition of serializability, found in [EGLT76], falls into a class of definitions that has subsequently been called *conflict serializability* [P86]. In conflict serializability, behaviors of schedules are compared according to certain *conflict relations* that they induce. A schedule $\sigma$ induces the conflict relation $CR_\sigma$ on pairs of operations in $\sigma$, where $(a_i, a_j) \in CR_\sigma$ if and only if $a_i$ and $a_j$ are from different transactions, $a_i$ appears before $a_j$ and both operations cannot be run in the other order and produce the same result. The conflict relation $CR_\sigma$ is extended to transactions by defining $(\tau_i, \tau_j) \in CR_\sigma$ if and only if $(a_i, a_j) \in CR_\sigma$ for some operation $a_i$ from $\tau_i$ and $a_j$ from $\tau_j$.

A conflict relation $CR_\sigma$ reflects the potential for one transaction to influence the behavior of another in the concurrent execution represented by the schedule $\sigma$. Thus, the behavior of two schedules can be compared by comparing there associated conflict relations. Two schedules $\sigma$ and $\sigma'$ are *conflict equivalent* if and only if $CR_\sigma$ and $CR_{\sigma'}$ are the same relations on transactions. A schedule $\sigma$ is *conflict serializable* if and only if it is conflict equivalent to some serial schedule $\sigma'$. An equivalent definition sometimes given is that $\sigma$ is conflict serializable if and only if $CR_\sigma$ is acyclic, since this ensures at least one serial schedule shares the same conflict relation.

As the second formulation of conflict serializability illustrates, whether or not a particular schedule is conflict serializable depends directly on the strength of the conflict relation: the more conflicting operations there are in a schedule $\sigma$, the more likely $CR_\sigma$ is to contain a cycle and hence fail to be conflict serializable. For this reason, operation semantics are used to define conflict relations that relate as few operations as possible. In models with read and write operations, the conflict relation is defined so that $(a_i, a_j) \in CR_\sigma$ whenever $a_i$ and $a_j$ reference the same variable and at least one is a write operation. This is because pairs of operations on different variables or pairs of operations that only read the same variable cannot influence each other. In models such as [K83] and [GS85], the greater degree to which operation semantics are specified permits weaker conflict relations to be specified. For example, two operations that change the value of the same variable do not necessarily conflict as they would if they were simply considered to be instances of write operations.

Another class of serializability definitions involves those that compare schedules on the basis of how they transform a system from one state to another. This class is sometimes subdivided into *final-state serializability* and *view serializability* [P86]. In both of these subclasses, a schedule "behaves like" another if and only if both transform identical initial states to identical final states. However, final-state and view serializability differ as to what portion of the system state is used to compare the effect of schedules.

In the definition of final-state serializability found in [K83], system states are compared according to the value of only those variables that are shared by transactions. However, it is argued in [Y84] that final-state serializability is inappropriate for models in which transactions contain read operations because it ignores the values copied into a transaction's local storage by read operations and does not take into account the possibility that transactions might read inconsistent values and behave erratically or present inconsistent output to users of the database system. View serializability is therefore proposed in [Y84] as a more appropriate definition of serializability. When comparing the effect of schedules, view serializability includes in the system state the values read by transactions in addition to the values of shared variables.

In [P79], the relationship between conflict and view serializability is explored. It is proven that every conflict-serializable schedule is also view serializable. In light of this result, view serializability would seem to be a preferable definition because of its generality. However, it is also shown in [P79] that the complexity of the general problem of deciding whether a schedule is view serializable as a function of its length is NP-complete. This makes it improbable that efficient algorithms can be constructed for synchronizing arbitrary sets of transactions. Because a schedule can be determined to be conflict serializable in time polynomial in its length, conflict serializability is more often used in practice as the basis for concurrency control.

### 1.3.5 Alternatives to Serializability

Some have suggested that requiring every schedule to be equivalent to some serial schedule is too strict a requirement for database systems (e.g. [L76]). The Concurrency Control Problem requires only that transactions transform the database system from one consistent state to another. Every serializable schedule will accomplish this, but in some cases there may be non-serializable schedules that do so as well.

An alternative is proposed in [G83]. There, every schedule is required to be *semantically consistent* rather than serializable. A schedule $\sigma$ is semantically consistent if

- $\sigma$ transforms the system from one consistent state to another, and
- there is a serial schedule $\sigma'$ such that for every initial consistent state, $\sigma$ and $\sigma'$ leave the same values in a specified set of *RS variables* (for *R*equiring *S*erializability).

Since the RS variables need not include every variable of the consistency constraint, the first requirement is not redundant. This approach has the advantage of allowing more concurrency than if schedules are required to be serializable. A simple example of a database system that models an airline reservation system is given in [G83] to illustrate this. The system contains four variables $SX$, $SY$, $TX$ and $TY$. The value of $SX$ denotes the number of passengers on a flight $FX$, while $TX$ denotes the type of plane scheduled to handle that flight: either "small" or "large". Variables $SY$ and $TY$ denote the same information for a flight $FY$. The consistency constraint for the system is

$$(SX \geq 100 \Rightarrow TX = \text{“large”}) \wedge (SY \geq 100 \Rightarrow TY = \text{“large”}).$$

The RS variables are $SX$ and $SY$.

Two transactions are considered, one that reserves a seat on both flights:

$RXY$: $R1$: Increment $SX$ by 1. If $SX \geq 100$, change $TX$ to "large".
$R2$: Increment $SY$ by 1. If $SY \geq 100$, change $TY$ to "large".

and one that cancels a seat on both flights:

$CXY$: $C1$: Decrement $SX$ by 1.
$C2$: Decrement $SY$ by 1.

Suppose that both $RXY$ and $CXY$ run concurrently in an initially consistent state with $SX = SY = 99$ and $TX = TY =$ "small" producing the schedule

$\sigma 3$: $R1, C1, C2, R2$.

This schedule will leave $SX = SY = 99$, $TX =$ "large" and $TY =$ "small", which is also a consistent state. It also leaves the RS variables $SX$ and $SY$ with the same values as either of the two possible serial schedules. Consequently, $\sigma 3$ is semantically consistent. However, $\sigma 3$ is not serializable under any of the definitions described previously and would not be allowed in a database system requiring serializability. Thus, an advantage of replacing serializability by the weaker requirement of semantic consistency is that more concurrency among transactions is possible. A disadvantage of this approach is that analysis of synchronization requirements for semantic consistency can be more complicated than for serializability because of the details of the consistency constraint and operation semantics that must be considered.

## 1.4 Reasoning About Concurrency

A database system can be viewed as a *concurrent program*—a collection of sequential programs that run concurrently. Properties of concurrent programs can be viewed in terms of *safety* and *liveness*. A safety property is one that specifies that one of a given set of "bad" states is never reached. An example of a safety property is *partial correctness*, which says that execution that begins in one of a given set of initial states does not terminate in a state outside of a given set of final states. A liveness property is one that specifies that some set of "good" states are eventually reached. An example of a liveness property is *termination*, which says that execution that begins in one of a given set of initial states eventually terminates.

As the schedules considered previously indicate, execution of a concurrent program can produce any of a number of different interleavings of its constituent operations. The interleavings that are possible depend on the *atomic operations* that constitute the concurrent program. An atomic operation is one that indivisibly runs to completion once started. Following [L80], an atomic operation that executes program text $S$ is denoted $\langle S \rangle$. In the sequel, we will write $S$ in Guarded Command Notation [D76] but require that $S$ is deterministic.

For all but the simplest concurrent programs, the number of different schedules is apt to be too large for safety and liveness properties to be verified by considering every possible schedule. To address this problem, a more tractable approach to reasoning about concurrent programs has been developed. It is based on the use of a formal logical system relating program behavior to predicates on program states.

Proof Outline Logic [SA87] is one programming logic for expressing and proving safety properties of concurrent programs. A ***proof outline*** is a formula

$$\{Q\}\, S\, \{R\}$$

where $Q$ and $R$ are predicates on the system state and are called ***assertions***; $S$ is an ***annotated program***, a program in which each atomic operation $\langle \alpha \rangle$ is preceded by zero or more assertions. An assertion that immediately precedes $\langle \alpha \rangle$ in the proof outline is called the ***precondition*** of $\langle \alpha \rangle$ and is denoted $pre(\langle \alpha \rangle)$. An assertion that immediately follows $\langle \alpha \rangle$ is called the ***postcondition*** of $\langle \alpha \rangle$ and is denoted $post(\langle \alpha \rangle)$.

A proof outline $\{Q\}\, S\, \{R\}$ specifies the safety property that if $S$ is started at some atomic operation $\langle \alpha \rangle$ in a state that satisfies $pre(\langle \alpha \rangle)$, then at any point reached during execution, the state will satisfy the assertion or assertions that appears at that point. Proof Outline Logic provides a set of axioms and inference rules for inferring valid proof outlines. These axioms and rules include those of Predicate Logic [S67] along with axioms and rules given in [SA87] that are specific to Proof Outline Logic. A summary of these rules can be found in Appendix A of this dissertation.

Dijkstra's *weakest precondition* predicate transformer [D76], a function that maps one assertion to another, is often used in conjunction with Proof Outline Logic to reason about programs. For $S$ an operation or program and $R$ a predicate, the predicate $wp(S,R)$ (read *the weakest precondition of $S$ with respect to $R$*) denotes the largest set of states in which execution of $S$ is guaranteed to terminate leaving $R$ true. From the semantics of proof outlines and $wp$, it follows that

$$\{wp(S,R)\}\, S\, \{R\}$$

is a valid proof outline for any $S$ and $R$. Thus, a precondition of $S$ that allows a given postcondition $R$ to be asserted can be computed using $wp$. A summary of general properties of $wp$ along with rules for computing $wp(S,R)$ can be found in Appendix B.

*Assertional reasoning* is the name given to the style of characterizing program semantics in terms of assertions on the program state. When compared to other approaches to reasoning about concurrent programs, an apparent disadvantage of assertional reasoning is the level of detail at which the analysis is carried out. Of course, this is also an advantage since it is possible to capture detailed semantic information that is ignored in other formal systems. Another advantage of assertional reasoning is that properties of concurrent programs are often specified most naturally in terms of properties of the states reached during execution. For example, a solution to the Concurrency Control Problem requires every execution that begins in a consistent state to leave a consistent state. This is an assertional property since it specifies a property of the system state (consistency) at points during execution (before and after). Yet another advantage of assertional reasoning is the ability not only to *prove* that a given program satisfies a particular specification, but also to *derive* programs from their specification, using the inference rules of the logic to motivate refinement of the program.

## 1.5 Overview of Dissertation

This thesis describes the application of assertional reasoning to the Concurrency Control Problem. Chapter 2 presents a new definition of serializability that is based on assertional reasoning and generalizes previously proposed definitions in several respects. A method for using Proof Outline Logic to specify and prove that database systems satisfy this definition of serializability is then presented. Chapter 3 presents an assertional view of locking and describes a method for deriving locking protocols. This method is then used to derive synchronization for a database system modeling a simple banking application. Finally, Chapter 4 summarizes the thesis and draws some conclusions from the research presented here.

# Chapter 2

# Serializability

As discussed in Section 1.3, there is no standard system model or definition of serializability. In this chapter, we describe the system model used in the remainder of this dissertation. We then propose a definition of serializability that generalizes previous definitions of serializability in several ways. Finally, we demonstrate how this definition can be formulated in Proof Outline Logic.

## 2.1 Database System Model

Concurrent execution of a set of transactions $\tau_0, \ldots, \tau_{N-1}$ is denoted

$$\textbf{cobegin}\ \tau_0 \parallel \cdots \parallel \tau_{N-1}\ \textbf{coend}. \tag{2.1}$$

A necessary condition for an atomic operation $\langle S \rangle$ in (2.1) to run is that it be *enabled*, which means that the control point before $\langle S \rangle$ has been reached and the system state be one in which $S$ will run to completion. During execution of (2.1), however, it is possible for more than one operation to be enabled at the same time. Consequently, a *scheduling* *policy* must be given to specify how operations are selected for execution from among those that are enabled. We will assume in the remainder of this dissertation that concurrent execution of transactions follows a *weakly fair* scheduling policy [SA87]—no operation that becomes and remains enabled will be forever delayed.

Execution of (2.1) terminates when every transaction has terminated. A transaction $\tau_i$ can terminate in two ways. One is for $\tau_i$ to *complete* by executing an operation $\langle end(\tau_i)\rangle$ and halting. The other way in which $\tau_i$ can terminate is to *abort*. A transaction aborts when conditions such as deadlock, system failure, or unexpected input make it undesirable or impossible for it to complete. $\tau_i$ aborts by executing an operation $\langle abort(\tau_i)\rangle$ and halting. Operation $\langle abort(\tau_i)\rangle$ typically implements *recovery operations* to cancel the effect of operations previously executed by $\tau_i$.

A database system $\Sigma$ can be specified by a 4-tuple $\langle V, C, T, \equiv\rangle$, where $V = \langle v_0, \ldots, v_n\rangle$ is a vector of variables, $C$ is a predicate on $V$, $T = \{\tau_0, \ldots, \tau_{N-1}\}$ is a set of sequential programs, and $\equiv$ is an equivalence relation on the domain of $V$ (the cross product of the domains of the variables of $V$). Variables of $V$ characterize the state of $\Sigma$. Any system state can be written as a vector of constants $V' = \langle v_0', \ldots, v_n'\rangle$, where each $v_i'$ is the value of the corresponding variable $v_i$ in that state. For any predicate $P$ on $V$, $P$ is true in state $V'$ if and only if $P^{V}_{V'} = true$.[1] Predicate $C$ in the specification of $\Sigma$ is a predicate that implies the consistency constraint of $\Sigma$; a state is *consistent* if $C$ is true in that state.

Each $\tau_i \in T$ models a transaction of $\Sigma$. An *execution* of $\Sigma$ is an execution of the concurrent program

[1] $P^{v_0,\ldots,v_n}_{e_0,\ldots,e_n}$ denotes the result obtained by simultaneously replacing all occurrences of $v_i$ by the corresponding $e_i$.

$$\textbf{cobegin}\ \tau_0 \parallel \cdots \parallel \tau_{N-1}\ \textbf{coend}, \tag{2.2}$$

and a *schedule* of $\Sigma$ is the sequence of atomic operations resulting from a terminating execution of (2.2). We assume that each $\tau_i$ will complete leaving $C$ true when executed in isolation starting with $C$ true. For each $\tau_i \in T$, we will assume that $V$ contains a Boolean variable $cf_i$, called the *completion flag* of $\tau_i$, such that $cf_i = true$ if and only if $\tau_i$ has completed. This models information that is typically found in system logs.

The equivalence relation $\equiv$ in the specification of $\Sigma$ is a binary relation on the domain of $V$ and partitions the states of $\Sigma$ into equivalence classes. Each equivalence class contains the states that cannot be distinguished from one another by the application supported by $\Sigma$. This provides an abstraction that hides aspects of the system state that are irrelevant to the application being supported. To limit the amount of information that can be hidden, $\equiv$ is requires to satisfy two *adequacy constraints*:

AC1. For all system states $V'$ and $V''$,

$$(V' \equiv V'') \Rightarrow (\forall i\colon\ 0 \le i < N\colon\ cf_i' = cf_i'').$$

AC2. For all system states $V'$ and $V''$,

$$(V' \equiv V'') \Rightarrow (C^V_{V'} \Leftrightarrow C^V_{V''}).$$

AC1 ensures that states in which different sets of transactions have completed are distinguishable. AC2 ensures that consistent states and inconsistent states are distinguishable.

An example a database specified as a 4-tuple is $\Sigma0$ of Figure 2.1. $\Sigma0$ models an application in which a series of independent events move elements one at a time from the

$$
\begin{aligned}
&\Sigma 0 = \langle\, V0, C0, T0, \equiv_0 \rangle \\
&\quad V0 = \langle q0, q1, x_0, \ldots, x_{N-1}, cf_0, \ldots, cf_{N-1} \rangle, \\
&\quad C0 = (q1 \cdot q0 = Q \wedge |q0| \geq (\#\, k\colon\ 0 \leq k < N\colon\ cf_k = \mathit{false})), \\
&\quad T0 = \{\tau_0, \ldots, \tau_{N-1}\}, \\
&\qquad \tau_i = S1_i\colon\ \langle x_i, q0 := q0(0), q0(1..) \rangle; \\
&\qquad\qquad S2_i\colon\ \langle q1 := q1 \cdot x_i \rangle; \\
&\qquad\qquad S3_i\colon\ \langle \mathit{end}(\tau_i) \rangle \\
&\quad (V0' \equiv_0 V0'') \Leftrightarrow (q0' = q0'' \wedge q1' = q1'' \wedge \bigwedge_{0 \leq k \leq N} cf_k' = cf_k'')
\end{aligned}
$$

Figure 2.1: Database System $\Sigma 0$.

head of one queue to the rear of another, as in a factory where parts are transferred from one assembly line to another. $V0$ contains two sequence variables $q0$ and $q1$ modeling the two queues, and a variable $x_i$ for each $\tau_i$ in $T0$ to hold the item removed from $q0$ and not yet appended to $q1$. The following notation is used for sequence variables:

| | |
|---|---|
| $\lvert s \rvert$ | the number of elements in $s$. |
| $s(i)$ | the $ith$ element of $s$ for $0 \leq i < \lvert s \rvert$. |
| $s(i..j)$ | the subsequence of consecutive elements from the $ith$ to the $jth$ for $0 \leq i \leq j < \lvert s \rvert$ (and the empty sequence if $j < i$). |
| $s(i..)$ | an abbreviation for $s(i..\lvert s \rvert - 1)$. |
| $s1 \cdot s2$ | the catenation of $s1$ and $s2$. |

The conjunct $q1 \cdot q0 = Q$ in the consistency constraint $C0$ specifies that queue elements are not lost in the transfer, while the second conjunct[2] $|q0| \geq (\#\, k\colon\ 0 \leq k <$

[2] $(\#\, i\colon\ R\colon\ P)$ denotes the number of values $i$ in range $R$ that satisfy $P$.

$N$: $cf_k = false$) specifies that $q0$ contains enough elements for every transaction that has not completed to remove one. Each transaction $\tau_i \in T0$ models the transfer of one element from the first queue to the second, using a local variable $x_i$ for temporary storage of the element removed. For simplicity, we assume that transactions of $\Sigma 0$ terminate only by completing. The operation $S1_i$ moves the first element of $q0$ into $x_i$, and $S2_i$ then moves it to the rear of $q1$; $S3_i$ has no effect other than ensuring that $cf_i = true$.

The equivalence relation $\equiv_0$ specifies that two states $V0'$ and $V0''$ are equivalent when each of $q0$ and $q1$ contain the same sequence of elements in both states, and the same transactions have completed. The values of temporary variables $x_0, \ldots, x_{N-1}$ are ignored by $\equiv_0$ since the particular order in which transactions run is insignificant in this application.

## 2.2 Serializability

Recall that $wp(S, R)$ denotes the set of states in which execution of $S$ will terminate leaving $R$ true. Using $wp$, it is possible to formalize the property that a schedule $\sigma$ "behaves like" like a serial schedule.

**Definition 2.2.1 (Serializable Schedule)** Let $\Sigma = \langle V, C, T, \equiv \rangle$ be a database system and let $SER(T)$ denote the set of serial schedules for $\Sigma$, each schedule consisting of zero or more transactions of $T$. Let $\widetilde{V}$ be a vector of new variables each having the same domain as the corresponding one in $V$. A schedule $\sigma$ of $\Sigma$ is a *serializable schedule* of $\Sigma$ if and only if:

$$\models (C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (\bigvee_{\sigma' \in SER(T)} wp(\sigma', V \equiv \widetilde{V})).$$

□

Definition 2.2.1 can be interpreted as follows. A state satisfying the antecedent is one satisfying the consistency constraint and in which execution of $\sigma$ is guaranteed to terminate in a state indistinguishable under $\equiv$ from $\widetilde{V}$. A state that satisfies the consequent is one in which execution of at least one serial schedule of $\Sigma$ is guaranteed to terminate in a state indistinguishable under $\equiv$ from $\widetilde{V}$. Thus, the implication specifies that any consistent state in which execution of $\sigma$ terminates in a state indistinguishable from $\widetilde{V}$ is one in which execution of at least one serial schedule $\sigma'$ of $\Sigma$ terminates in a state indistinguishable from $\widetilde{V}$. From the assumption that $\equiv$ satisfies adequacy constraint AC1, it follows that the states reached by $\sigma$ and $\sigma'$ will have the same set of completed transactions. From the assumption that $\equiv$ satisfies adequacy constraint AC2, it will follow that the state reached by $\sigma$ will satisfy the consistency constraint if and only if the state reached by $\sigma'$ does. Since $\sigma'$ is a serial schedule that starts in a consistent state, it will always leave a consistent state, and consequently the state reached by $\sigma$ will be consistent.

For an example of a schedule that is serializable according to Definition 2.2.1, consider $\Sigma 0$ of Figure 2.1. When $N = 2$, execution of $\Sigma 0$ can produce the schedule

$\sigma 4$: $S1_0; S1_1; S2_0; S2_1; S3_0; S3_1$.

Consider the serial schedule

$\sigma 5$: $S1_0; S2_0; S3_0; S1_1; S2_1; S3_1$.

Using the rules for computing $wp$ (see Appendix B for a summary), it can be shown that

$$
\begin{aligned}
&(C0 \wedge wp(\sigma 4, V0 \equiv_0 \widetilde{V0})) = \\
&\quad q1 \cdot q0 = Q \wedge |q0| \geq (\# k\colon\ 0 \leq k < 2\colon\ cf_k = \mathit{false}) \wedge q0(2..) = \widetilde{q0} \\
&\quad \wedge q1 \cdot q0(0) \cdot q0(1) = \widetilde{q1} \wedge \mathit{true} = \widetilde{cf}_0 = \widetilde{cf}_1
\end{aligned}
$$

and

$$
\begin{aligned}
&wp(\sigma 5, V0 \equiv_0 \widetilde{V0}) = \\
&\quad q0(2..) = \widetilde{q0} \wedge q1 \cdot q0(0) \cdot q0(1) = \widetilde{q1} \wedge \mathit{true} = \widetilde{cf}_0 = \widetilde{cf}_1
\end{aligned}
$$

From this it follows that

$$\models (C0 \wedge wp(\sigma 4, V0 \equiv_0 \widetilde{V0})) \Rightarrow wp(\sigma 5, V0 \equiv_0 \widetilde{V0})$$

and since $\sigma 5 \in SER(T0)$,

$$\models (C0 \wedge wp(\sigma 4, V0 \equiv_0 \widetilde{V0})) \Rightarrow (\bigvee_{\sigma' \in SER(T0)} wp(\sigma', V0 \equiv_0 \widetilde{V0})).$$

Therefore, $\sigma 4$ is serializable according to our definition.

Definition 2.2.1 for serializable schedules can be extended to obtain a definition for serializable database systems.

**Definition 2.2.2 (Serializable System)** Database system $\Sigma = \langle V, C, T, \equiv \rangle$ is a *serializable system* if and only if concurrent execution of transactions that begins with $C$ true always terminates and every resulting schedule is serializable. □

Note that although the schedule $\sigma 4$ is serializable, $\Sigma 0$ is not a serializable system since

$$\sigma 6\colon\ S1_0; S1_1; S2_1; S2_0; S3_0; S3_1$$

is not a serializable schedule.

## 2.3 Serializability with Proof Outlines

Definition 2.2.1 characterizes serializability using *wp*. It is also possible to characterize serializability using proof outlines. Two benefits result from such a formulation. The first is that Proof Outline Logic then can be used to verify formally the serializability of a database system. The second, explored more fully in Chapter 3, is that it becomes possible to derive synchronization protocols that ensure serializability.

A Proof Outline Logic characterization of serializability is formulated by introducing auxiliary variables and operations on them that allow the behavior of serial schedules to be characterized by assertions. Let $\Sigma = \langle V, C, T, \equiv \rangle$ be a database system with variables

$$V = \langle v_0, \ldots, v_n \rangle$$

and transactions

$$T = \{\tau_0, \ldots, \tau_{N-1}\}.$$

Define a vector of new variables

$$\widehat{V} = \langle \widehat{v}_0, \ldots, \widehat{v}_n \rangle,$$

with each new variable $\widehat{v}_k$ having the same type as the corresponding variable $v_k$ in $V$. Each $\widehat{v}_k$ is called the ***shadow variable*** corresponding to $v_k$. With these shadow variables, construct a set of new transactions

$$\widehat{T} = \{\widehat{\tau}_0, \ldots, \widehat{\tau}_{N-1}\},$$

where each $\widehat{\tau}_i$ is obtained from $\tau_i \in T$ by replacing all reference to $v_k \in V$ by a reference to the corresponding $\widehat{v}_k \in \widehat{V}$. Each $\widehat{\tau}_i$ is called the ***shadow transaction*** corresponding to $\tau_i$.

Let $SER(\widehat{T})$ denote the set of serial schedules consisting of zero or more transactions of $\widehat{T}$. The isomorphism between $V$ and $\widehat{V}$ and between each $\tau_i \in T$ and $\widehat{\tau}_i \in \widehat{T}$ implies that for any serial schedule $\sigma' \in SER(T)$, there is a serial schedule $\widehat{\sigma} \in SER(\widehat{T})$ that transforms $\widehat{V}$ in the same way that $\sigma$ transforms $V$. This isomorphism between schedules of $SER(T)$ and $SER(\widehat{T})$ makes it possible to construct a proof outline that is valid if and only if $\sigma$ satisfies Definition 2.2.1.

**Theorem 2.3.1 (Schedule Serializability with Proof Outlines)** Schedule $\sigma$ of database system $\Sigma = \langle V, C, T, \equiv \rangle$ is a serializable schedule if and only if

$$
\begin{aligned}
SS0(\sigma): \quad & \{C \wedge V = \widehat{V}\} \\
& \sigma \\
& \{\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})\}
\end{aligned}
$$

is valid. □

**Proof of Theorem 2.3.1** From the interpretation of $SS0(\sigma)$ and of the weakest precondition predicate transformer, $SS0(\sigma)$ is a valid proof outline if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, true)) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})). \tag{2.3}$$

Thus, the theorem follows if $\sigma$ is a serializable schedule of $\Sigma$ if and only if (2.3). This is proven in Lemma 2.3.4 proven below. □

The proof of Lemma 2.3.4 will frequently rely on inferences that are justified by the following two lemmas. The first states that substituting subformulas of $A$ with equivalent ones results in a formula that is equivalent to $A$.

**Lemma 2.3.2** Let $A'$ be obtained from $A$ by replacing some occurrences of $B_1$, ..., $B_n$ by $B'_1$, ..., $B'_n$ respectively. If

$$\models B_1 \Leftrightarrow B'_1, \ldots, \models B_n \Leftrightarrow B'_n,$$

then

$$\models A \text{ if and only if } \models A'.$$

□

**Proof of Lemma 2.3.2** By induction on the structure of $A$. See [S67] for details. □

The second lemma characterizes the distributivity of $wp$ over conjunction with a predicate $B$ when $B$ does not contain variables referenced by $S$.

**Lemma 2.3.3** For any program $S$ and predicates $A$ and $B$, if $S$ does not change any variable of $B$, then

$$\models (wp(S,A) \wedge B) \Leftrightarrow wp(S, A \wedge B).$$

□

**Proof of Lemma 2.3.3** By definition, $wp(S,B)$ represents the set of all states such that execution of $S$ begun in any one of them is guaranteed to terminate in a state satisfying $B$. Since $S$ does not change any variable of $B$, then $wp(S,B)$ is the set of states in which $S$ is guaranteed to terminate and in which $B$ is true. Thus,

$$\models (wp(S, true) \wedge B) \Leftrightarrow wp(S, B). \tag{2.4}$$

By Predicate Logic, $wp(S, A)$ can be conjoined to both sides of (2.4) giving

$$\models (wp(S, A) \wedge wp(S, true) \wedge B) \Leftrightarrow (wp(S, A) \wedge wp(S, B)). \tag{2.5}$$

Distributivity of Conjunction from Appendix B implies that

$$wp(S, A) \wedge wp(S, true)) \Leftrightarrow wp(S, A \wedge true).$$

Substituting the right side for the left in (2.5) gives

$$\models (wp(S, A \wedge true) \wedge B) \Leftrightarrow (wp(S, A) \wedge wp(S, B)). \tag{2.6}$$

Distributivity of Conjunction also implies that

$$wp(S, A) \wedge wp(S, B)) \Leftrightarrow wp(S, A \wedge B).$$

Substituting the right side for the left in (2.6) gives

$$\models (wp(S, A \wedge true) \wedge B) \Leftrightarrow (wp(S, A \wedge B)). \tag{2.7}$$

Since $(A \wedge true) \Leftrightarrow A$,

$$\models (wp(S, A) \wedge B) \Leftrightarrow wp(S, A \wedge B).$$

$\square$

Using these lemmas, the equivalence of the serializability of $\sigma$ and the validity of (2.3) can be proven.

**Lemma 2.3.4** $\sigma$ is a serializable schedule of $\Sigma$ if and only if (2.3). $\square$

**Proof of Lemma 2.3.4** By Definition 2.2.1, $\sigma$ is serializable under $\equiv$ if and only if

$$\models (C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (\bigvee_{\sigma' \in SER(T)} wp(\sigma', V \equiv \widetilde{V})). \qquad (2.8)$$

From Predicate Logic,

$$\models P \Leftrightarrow (\forall \widehat{V}:\ V = \widehat{V} \Rightarrow P^{V}_{\widehat{V}})$$

for any predicate $P$. Taking

$$P = [\bigvee_{\sigma' \in SER(T)} wp(\sigma', V \equiv \widetilde{V})],$$

and applying Lemma 2.3.2, (2.8) if and only if

$$\models (C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (\forall \widehat{V}:\ V = \widehat{V} \Rightarrow (\bigvee_{\sigma' \in SER(T)} wp(\sigma', V \equiv \widetilde{V}))^{V}_{\widehat{V}}). \qquad (2.9)$$

From the construction of the shadow transactions and definition of $SER(\widehat{T})$,

$$\models (\bigvee_{\sigma' \in SER(T)} wp(\sigma', V \equiv \widetilde{V}))^{V}_{\widehat{V}} \Leftrightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})).$$

Thus, (2.9) if and only if

$$\models (C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (\forall \widehat{V}:\ V = \widehat{V} \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))). \qquad (2.10)$$

From Predicate Logic, when variables of $\widehat{V}$ are not free in $P$,

$$\models (P \Rightarrow (\forall \widehat{V}:\ Q)) \Leftrightarrow (\forall \widehat{V}:\ P \Rightarrow Q).$$

Taking

$$P = [C \wedge wp(\sigma, V \equiv \widetilde{V})] \quad \text{and}$$
$$Q = [V = \widehat{V} \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))],$$

(2.10) if and only if

$$\models (\forall \widehat{V}\colon\ (C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (V = \widehat{V} \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})))). \tag{2.11}$$

From Predicate Logic,

$$\models (\forall \widehat{V}\colon\ P) \text{ if and only if } \models P.$$

Taking

$$P = [(C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (V = \widehat{V} \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})))],$$

(2.11) if and only if

$$\models (C \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (V = \widehat{V} \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))). \tag{2.12}$$

From Predicate Logic,

$$\models [P \Rightarrow (Q \Rightarrow R)] \Leftrightarrow [(P \wedge Q) \Rightarrow R].$$

Taking

$$\begin{aligned} P &= [C \wedge wp(\sigma, V \equiv \widetilde{V})], \\ Q &= [V = \widehat{V}] \quad \text{and} \\ R &= [\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})], \end{aligned}$$

(2.12) if and only if

$$\models (C \wedge wp(\sigma, V \equiv \widetilde{V}) \wedge V = \widehat{V}) \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})). \tag{2.13}$$

By the commutativity of conjunction in the antecedent, (2.13) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})). \tag{2.14}$$

From Predicate Logic,

$$\models ((P \wedge Q) \Rightarrow R) \Leftrightarrow ((P \wedge Q) \Rightarrow (Q \wedge R)).$$

Taking

$$\begin{aligned} P &= [C \wedge V = \widehat{V}], \\ Q &= [wp(\sigma, V \equiv \widetilde{V})] \quad \text{and} \\ R &= [\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})], \end{aligned}$$

(2.14) if and only if

$$\begin{aligned} &\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \\ &\quad (wp(\sigma, V \equiv \widetilde{V}) \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))). \end{aligned} \tag{2.15}$$

Since $\sigma$ does not reference any free variable of $(\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))$, it follows by Lemma 2.3.3 that

$$\models (wp(\sigma, V \equiv \widetilde{V}) \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})))$$

$$\Leftrightarrow$$

$$wp(\sigma, V \equiv \widetilde{V} \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))).$$

Thus, (2.15) if and only if

$$\begin{aligned} &\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \\ &\quad wp(\sigma, V \equiv \widetilde{V} \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))). \end{aligned} \tag{2.16}$$

Since conjunction distributes over disjunction,

$$(V \equiv \widetilde{V} \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))) \Leftrightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} V \equiv \widetilde{V} \wedge wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})).$$

Thus, (2.16) if and only if

$$\begin{aligned} &\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \\ &\quad wp(\sigma, (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} V \equiv \widetilde{V} \wedge wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V}))). \end{aligned} \tag{2.17}$$

Using the property that $\widehat{\sigma}$ does not modify any variable in $V \equiv \widetilde{V}$ and Lemma 2.3.3,

$$\models (V \equiv \widetilde{V} \wedge wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V})) \Leftrightarrow wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V} \wedge V \equiv \widetilde{V})$$

for each $\widehat{\sigma} \in SER(\widehat{T})$. Thus, (2.17) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \\ wp(\sigma, (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv \widetilde{V} \wedge V \equiv \widetilde{V}))). \tag{2.18}$$

Because $\equiv$ is an equivalence relation, it is transitive and symmetric. From this it follows that

$$(\widehat{V} \equiv \widetilde{V} \wedge V \equiv \widetilde{V}) \Leftrightarrow (\widehat{V} \equiv V \wedge V \equiv \widetilde{V}).$$

Thus, (2.18) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \\ wp(\sigma, (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V \wedge V \equiv \widetilde{V}))). \tag{2.19}$$

Using the property that $\widehat{\sigma}$ does not modify variables of $V \equiv \widetilde{V}$ and Lemma 2.3.3,

$$\models wp(\widehat{\sigma}, \widehat{V} \equiv V \wedge V \equiv \widetilde{V}) \Leftrightarrow (V \equiv \widetilde{V} \wedge wp(\widehat{\sigma}, \widehat{V} \equiv V)).$$

Thus, (2.19) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \\ wp(\sigma, (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} V \equiv \widetilde{V} \wedge wp(\widehat{\sigma}, \widehat{V} \equiv V))). \tag{2.20}$$

Since conjunction distributes over disjunction,

$$(\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} V \equiv \widetilde{V} \wedge wp(\widehat{\sigma}, \widehat{V} \equiv V)) \Leftrightarrow (V \equiv \widetilde{V} \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))).$$

Thus, (2.20) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \qquad (2.21)$$
$$wp(\sigma, V \equiv \widetilde{V} \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))).$$

Since $wp$ satisfies the property of Distributivity of Conjunction,

$$\models wp(\sigma, V \equiv \widetilde{V} \wedge (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V)))$$
$$\Leftrightarrow$$
$$(wp(\sigma, V \equiv \widetilde{V}) \wedge wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))).$$

Thus, (2.21) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow \qquad (2.22)$$
$$(wp(\sigma, V \equiv \widetilde{V}) \wedge wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))).$$

From Predicate Logic,

$$((P \wedge Q) \Rightarrow (Q \wedge R)) \Leftrightarrow ((P \wedge Q) \Rightarrow R).$$

Taking

$$\begin{aligned} P &= [C \wedge V = \widehat{V}], \\ Q &= [wp(\sigma, V \equiv \widetilde{V})] \quad \text{and} \\ R &= [wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))], \end{aligned}$$

(2.22) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V)). \qquad (2.23)$$

From Predicate Logic,

$$\models P \text{ if and only if } \models P_V^{\widetilde{V}}$$

for any predicate $P$. Taking

$$P = [(C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))],$$

(2.23) if and only if

$$\models [(C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv \widetilde{V})) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))]^{\widetilde{V}}_{V}. \tag{2.24}$$

Since $\widetilde{V}$ does not occur in $C \wedge V = \widehat{V}$ or in $wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V))$, (2.24) if and only if

$$\models (C \wedge V = \widehat{V} \wedge [wp(\sigma, V \equiv \widetilde{V})]^{\widetilde{V}}_{V}) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V)). \tag{2.25}$$

Since[3] $\widetilde{V}$ does not occur in $\sigma$,

$$wp(\sigma, V \equiv \widetilde{V})^{\widetilde{V}}_{V} \Leftrightarrow wp(\sigma, V \equiv V).$$

Thus, (2.25) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, V \equiv V)) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V)). \tag{2.26}$$

Since $\equiv$ is an equivalence relation, it is reflexive. Thus,

$$\models (V \equiv V) \Leftrightarrow \mathit{true}.$$

Thus, (2.26) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma, \mathit{true})) \Rightarrow wp(\sigma, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, \widehat{V} \equiv V)). \tag{2.27}$$

By identity, (2.27) if and only if (2.3).

□

[3] This can be proven by induction on the structure of $\sigma$.

Theorem 2.3.1 characterizes serializability in terms of proof outlines the serializability of a particular schedule of a database system. This result can be extended to obtain a similar characterization of the serializability of an entire database system.

**Theorem 2.3.5 (System Serializability with Proof Outlines)** Database system $\Sigma = \langle V, C, T, \equiv \rangle$ is a serializable system if and only if execution of $\Sigma$ terminates when started with $C \wedge V = \widehat{V}$ true and

$$SD0(\Sigma):\quad \begin{array}{l} \{C \wedge V = \widehat{V}\} \\ \textbf{cobegin}\ \tau_0 \parallel \cdots \parallel \tau_{N-1}\ \textbf{coend} \\ \{\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})\} \end{array}$$

is valid. □

**Proof of Theorem 2.3.5** Since the variables of $\widehat{V}$ do not occur in transactions of $\Sigma$, execution of $\Sigma$ terminates when started with $C \wedge V = \widehat{V}$ true if and only if execution of $\Sigma$ terminates when started with $C$ true. The interpretation of proof outlines and the semantics of **cobegin** imply that $SD0(\Sigma)$ is valid if and only if

$$SS0(\sigma):\quad \{C \wedge V = \widehat{V}\}\sigma\{\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})\}$$

is valid for every schedule $\sigma$ of $\Sigma$. By Theorem 2.3.1, $SS0(\sigma)$ is valid if and only if $\sigma$ is a serializable schedule. The theorem follows immediately from the definition of a serializable system. □

The hypotheses of Theorem 2.3.5 suggest a method for proving a database system serializable.

**Method 2.3.6 (Proving System Serializability)** To prove that a system $\Sigma = \langle V, C, T, \equiv \rangle$ is serializable:

1. **Introduce Shadow Variables and Shadow Transactions.** Define shadow variables $\widehat{V}$ and construct shadow transactions $\widehat{T}$ corresponding to the variables $V$ and transactions $T$ of $\Sigma$.

2. **Prove $SD0(\Sigma)$.** Prove that

$$
\begin{aligned}
SD0(\Sigma):\quad & \{C \wedge V = \widehat{V}\} \\
& \textbf{cobegin}\ \tau_0 \parallel \cdots \parallel \tau_{N-1}\ \textbf{coend} \\
& \{\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})\}
\end{aligned}
$$

   is valid.

3. **Prove Termination.** Prove that execution of $\Sigma$ terminates when started with $C \wedge V = \widehat{V}$ true.

□

## 2.4 An Example

We now present an example of the application of Method 2.3.6. As pointed out in Section 2.2, $\Sigma 0$ is not a serializable system. However, a serializable system can be constructed from $\Sigma 0$ by synchronizing transactions using a simplified version of the conservative timestamp ordering protocol in [BG81].

In conservative timestamp ordering, a unique integer *timestamp* is assigned to each transaction as it begins to run. A *version number* associated with each shared variable holds the timestamp of the last transaction to access it. An operation from transaction $\tau_i$ can access $v$ if it satisfies the following conditions.

TS1. The timestamp of $\tau_i$ is greater than the version number of $v$.

TS2. No transaction $\tau_j$ with a timestamp less than that of $\tau_i$ will later attempt to access $v$.

Since each transaction sets the version number of $v$ to its timestamp when accessing it, condition TS1 implies that the timestamp of a transaction $\tau_i$ accessing $v$ is greater than that of another transaction $\tau_j$ that accesses $v$ immediately before $\tau_i$. One consequence of this is that transactions are guaranteed to access $v$ in an order consistent with that of their timestamps. Another consequence is that the version number of $v$ is monotonically non-decreasing. Therefore, if a transaction $\tau_j$ finds TS1 false when attempting to access $v$, TS1 will subsequently remain false and prevent $\tau_j$ from completing. To avoid this possibility, condition TS2 requires $\tau_i$ to wait before accessing $v$ until all transaction attempting to access $v$ and having smaller timestamps have done so.[4] The result is that transaction satisfying TS1 and TS2 will access $v$ in ascending timestamp order without aborting.

We model the assignment of timestamps and synchronization of operations according to version numbers as follows. Let *clock*, $vq0$ and $vq1$ be integer variables holding the global clock and version numbers of $q0$ and $q1$. For each $\tau_i \in T0$, let $ts_i$ be an integer variable holding the timestamp of $\tau_i$ and let *clock* be an integer variable holding the value of the clock. To model the selection of a timestamp by $\tau_i$, the operation

$$S0_i: \ \langle \, clock, ts_i := clock + 1, clock + 1 \, \rangle$$

[4]Many timestamp protocols relax the second condition and abort transactions trying to access variables with version numbers greater than their timestamp. However, these protocols require older versions to be maintained for recovery purposes and also require additional machinery to cope with the possibility of cascading aborts [BG81]. We choose the more restrictive protocol so that the proof of correctness is not obscured by these additional complexities.

is added to $\tau_i$ before $S1_i$. To model the update of version numbers when $\tau_i$ accesses $q0$ and $q1$,

$$vq0 := ts_i$$

is added to $S1_i$ and

$$vq1 := ts_i$$

is added to $S2_i$.

To denote synchronization that delays an operation $S$ until a condition $B$ becomes true, we enclose $S$ in a guarded command of the form[5]

$$\langle \mathbf{if}\ B \rightarrow S\ \mathbf{fi} \rangle.$$

The following lemma provides a guard $B$ for $S1_i$ that ensures that the transactions of $T0$ satisfy conditions TS1 and TS2 for accessing $q0$.

**Lemma 2.4.1 (Timestamp Condition)** Transactions $\tau_0, \ldots, \tau_{N-1}$ satisfy conditions TS1 and TS2 for accessing $q0$ if each $S1_i$ is delayed until $vq0 + 1 = ts_i$. □

**Proof of Lemma 2.4.1** Consider operation $S1_i$ in transaction $\tau_i$. Suppose that $S1_i$ does not run until $vq0 + 1 = ts_i$. Since

$$vq0 + 1 = ts_i \Rightarrow ts_i > vq0,$$

then delaying each $S1_i$ until $vq0 + 1 = ts_i$ ensures that the timestamp of each $\tau_i$ is greater than $vq0$ when $\tau_i$ accesses $q0$. This is what is required by condition TS1.

[5]Guarded Command Notation semantics specify that $\mathbf{if}\ B \rightarrow S\ \mathbf{fi}$ executes $S$ if started with $B$ true and will fail to terminate if started with $B$ false. Since atomic operations run to completion once started, execution of $\langle \mathbf{if}\ B \rightarrow S\ \mathbf{fi} \rangle$ delays until $B$ becomes true.

Now consider operation $S1_j$ in a transaction $\tau_j$ $j \neq i$ that runs after $S1_i$. Note that assignment $vq0 := ts_i$ in $S1_i$ leaves $vq0 \geq ts_i$. Since

$$vq0 + 1 = ts_j \Rightarrow ts_j > vq0,$$

then $ts_j > vq0$ will be true when $S1_j$ runs. The assignment $vq0 := ts_i$ in $S1_i$ ensures that $vq0 \geq ts_i$ after $S1_i$ runs. Since

$$(vq0 + 1 = ts_j \wedge vq0 \geq ts_i) \Rightarrow ts_j > ts_i,$$

then delaying each $S1_i$ until $vq0 + 1 = ts_i$ ensures that the timestamp of $\tau_j$ cannot be less than the timestamp of $\tau_i$ when $S1_j$ runs after $S1_i$. This is required by condition TS2. □

By Lemma 2.4.1, access to $q0$ will satisfy TS1 and TS2 if $vq0 + 1 = ts_i$ is chosen as the guard for each $S1_i$. By a similar analysis, it can be shown that access to $q1$ will satisfy TS1 and TS2 if $vq1 + 1 = ts_i$ is chosen as the guard for each $S2_i$. Because of the synchronization that has been added to transactions, the conjunct

$$vq0 = vq1 = clock$$

has been added to the consistency constraint $C1$ to ensure that transactions complete when executed in isolation starting in a consistent state. In addition, the definition of $\equiv_1$ has been changed to ensure that continues to satisfy adequacy constraint AC2. This gives the synchronized database system $\Sigma 1$ of Figure 2.2.

We can now apply Method 2.3.6 to prove that $\Sigma 1$ is serializable. First we define shadow variables

$$\widehat{V1}: \quad \widehat{q0}, \widehat{q1}, \widehat{x}_0, \ldots, \widehat{x}_{N-1}, \widehat{cf}_0, \ldots, \widehat{cf}_{N-1}, \widehat{clock}, \widehat{vq0}, \widehat{vq1}, \widehat{ts}_0, \ldots, \widehat{ts}_{N-1}$$

$$\Sigma 1 = \langle V1, C1, T1, \equiv_1 \rangle$$

$$V1 = \langle q0, q1, x_0, \ldots, x_{N-1}, cf_0, \ldots, cf_{N-1}, clock, vq0, vq1, ts_0, \ldots, ts_{N-1} \rangle$$

$$C1 = \{ q1 \cdot q0 = Q \wedge |q0| \geq (\# k:\ 0 \leq k < N:\ cf_k = false) \wedge vq0 = vq1 = clock \},$$

$$T1 = \{\tau'_0, \ldots, \tau'_{N-1}\},$$

$$\begin{aligned}
\tau'_i = S0_i:&\quad \langle clock, ts_i := clock + 1, clock + 1 \rangle; \\
S1_i:&\quad \langle \mathbf{if}\ vq0 + 1 = ts_i \rightarrow x_i, q0, vq0 := q0(0), q0(1..), ts_i\ \mathbf{fi} \rangle; \\
S2_i:&\quad \langle \mathbf{if}\ vq1 + 1 = ts_i \rightarrow q1, vq1 := q1 \cdot x_i, ts_i\ \mathbf{fi} \rangle; \\
S3_i:&\quad \langle end(\tau'_i) \rangle
\end{aligned}$$

$$(V1' \equiv_1 V1'') \Leftrightarrow (q0' = q0'' \wedge q1' = q1'' \wedge \bigwedge_{0 \leq k \leq N} cf'_k = cf''_k \\ \wedge\ vq0' = vq0'' \wedge vq1' = vq1'' \wedge clock' = clock'')$$

Figure 2.2: Synchronized Database System $\Sigma 1$

corresponding to the variables of $V1$ and construct shadow transactions

$$\begin{aligned}
\widehat{\tau}'_i:&\quad \langle \widehat{clock}, \widehat{ts}_i := \widehat{clock} + 1, \widehat{clock} + 1 \rangle; \\
&\quad \langle \mathbf{if}\ \widehat{vq0} + 1 = \widehat{ts}_i \rightarrow \widehat{x}_i, \widehat{q0}, \widehat{vq0} := \widehat{q0}(0), \widehat{q0}(1..), \widehat{ts}_i\ \mathbf{fi} \rangle; \\
&\quad \langle \mathbf{if}\ \widehat{vq1} + 1 = \widehat{ts}_i \rightarrow \widehat{q1}, \widehat{vq1} := \widehat{q1} \cdot \widehat{x}_i, \widehat{ts}_i\ \mathbf{fi} \rangle; \\
&\quad \langle end(\widehat{\tau}'_i) \rangle
\end{aligned}$$

corresponding to each $\tau'_i$.

Next, we prove that

$$\begin{aligned}
SD0(\Sigma 1):\quad & \{ C1 \wedge V1 = \widehat{V1} \} \\
& \mathbf{cobegin}\ \tau'_0 \parallel \cdots \parallel \tau'_{N-1}\ \mathbf{coend} \\
& \{ \bigvee_{\widehat{\sigma} \in SER(\widehat{T1})} wp(\widehat{\sigma}, V1 \equiv_1 \widehat{V1}) \}
\end{aligned}$$

is valid. To do this, we first construct the full proof outline[6]

[6] A full proof outline is one in which every atomic operations is preceded and followed by at least one assertion.

$FSD0(\Sigma 1)$:

$$\{\, C1 \wedge V1 = \widehat{V1}\}$$
$$\langle\, CLOCK_0, VQ0_0, VQ1_0, \ldots, CLOCK_{N-1}, VQ0_{N-1}, VQ1_{N-1} := 0, \ldots, 0\rangle;$$
$$\{I \wedge \bigwedge_{0 \leq k < N} 0 = VQ1_k = VQ0_k = CLOCK_k\}$$
$$\textbf{cobegin } PO(\tau'_0) \parallel \cdots \parallel PO(\tau'_{N-1}) \textbf{ coend}$$
$$\{\, \widehat{vq0} = \widehat{vq1} = \widehat{clock} \wedge q0 = \widehat{q0}(N\,..) \wedge q1 = \widehat{q1}\cdot(\widehat{q0}(0\,..\,N-1)) \wedge \bigwedge_{0 \leq k < N} cf_k = true \wedge vq0 = vq1 = clock = \widehat{clock} + N\}$$

where each $PO(\tau'_i)$ is the proof outline for $\tau'_i$ shown in Figure 2.3. In each $\tau'_i$, auxiliary variables[C73,OG76] $CLOCK_i$, $VQ0_i$ and $VQ1_i$ are used to record whether $\tau'_i$ has incremented $clock$, $vq0$, and $vq1$, respectively.

Each assertion contains the invariant

$$I\colon\; I0 \wedge I1 \wedge I2 \wedge I3 \wedge I4.$$

The first conjunct

$$\begin{aligned} I0\colon\;\; & \widehat{clock} = \widehat{vq0} = \widehat{vq1} \wedge (\forall k\colon\; 0 \leq k < N\colon\; 0 \leq VQ1_k \leq VQ0_k \leq CLOCK_k \leq 1) \\ & \wedge clock = \widehat{clock} + \sum_{0 \leq k < N} CLOCK_k \wedge vq0 = \widehat{vq0} + \sum_{0 \leq k < N} VQ0_k \\ & \wedge vq1 = \widehat{vq1} + \sum_{0 \leq k < N} VQ1_k \end{aligned}$$

specifies that $\widehat{clock}$, $\widehat{vq0}$ and $\widehat{vq1}$ remain equal, and bounds the values of $clock$, $vq0$ and $vq1$ in terms of the values of the auxiliary and shadow variables. The second conjunct

$$\begin{aligned} I1\colon\;\; & |q0| \geq N - \sum_{0 \leq k < N} VQ0_k \wedge |\widehat{q0}| \geq N \\ & \wedge q0 = \widehat{q0}((\sum_{0 \leq k < N} VQ0_k)\,..) \wedge q1 = \widehat{q1}\cdot[\widehat{q0}(0\,..(\sum_{0 \leq k < N} VQ1_k)-1)] \end{aligned}$$

bounds the size of $q0$ and $\widehat{q0}$ and specifies in terms of the auxiliary variables the elements that have been transferred from $q0$ to $q1$. The third conjunct

$$I2: \bigwedge_{0 \leq k < N} CLOCK_k = 1 \Rightarrow ts_k \leq clock$$
$$\wedge \bigwedge_{0 \leq j \neq k < N} (CLOCK_j = 1 \wedge CLOCK_k = 1) \Rightarrow ts_j \neq ts_k$$

specifies that different transactions choose different timestamps, while the fourth and fifth conjuncts

$$I3: (\forall v: vq0 < v \leq clock: (\exists k: 0 \leq k < N: CLOCK_k = 1 \wedge VQ0_k = 0 \wedge v = ts_k))$$

and

$$I4: (\forall v: vq1 < v \leq vq0: (\exists k: 0 \leq k < N: VQ0_k = 1 \wedge VQ1_k = 0 \wedge v = ts_k))$$

specify that some transaction $\tau'_k$ has a timestamp $ts_k = v$ for every value $v$ between $vq0+1$ and $clock$ or between $vq1+1$ and $vq0$.

The proof $FSD0(\Sigma 1)$ is a straightforward application of the axioms and rules of Proof Outline Logic, and is omitted here.

From $FSD0(\Sigma 1)$, $SD0(\Sigma 1)$ can be inferred as follows. From $FSD0(\Sigma 1)$, the proof outline

$$\begin{aligned}
&\{ C1 \wedge V1 = \widehat{V1} \} \\
&\textbf{cobegin}\ \tau'_0 \parallel \cdots \parallel \tau'_{N-1}\ \textbf{coend} \\
&\{ q0 = \widehat{q0}(N..) \wedge q1 = \widehat{q1} \cdot (\widehat{q0}(0..N-1)) \wedge \bigwedge_{0 \leq k < N} cf_k = true \}
\end{aligned} \tag{2.28}$$

can be inferred using the Assertion Deletion Rule followed by the Auxiliary Variable Deletion Rule. It can be shown by induction on $N$ that

$$\begin{aligned}
&wp(\hat{\tau}_0; \ldots; \hat{\tau}_{N-1}, V1 \equiv_1 \widehat{V1}) \\
&\quad = (\widehat{vq0} = \widehat{vq1} = \widehat{clock} \wedge q0 = \widehat{q0}(N..) \wedge q1 = \widehat{q1} \cdot (\widehat{q0}(0..N-1)) \\
&\qquad \wedge \bigwedge_{0 \leq k < N} cf_k = true \wedge vq0 = vq1 = clock = \widehat{clock} + N), \\
&\quad = post(FSD0(\Sigma 1)).
\end{aligned}$$

$PO(\tau_i')$:

$\{I \wedge vq1 \leq vq0 \leq clock \wedge CLOCK_i = 0 \wedge VQ0_i = 0 \wedge VQ1_i = 0\}$

$S0_i$: $\langle clock, ts_i, CLOCK_i := clock + 1, clock + 1, 1\rangle$;

$\{I \wedge vq1 \leq vq0 < ts_i \leq clock \wedge CLOCK_i = 1 \wedge VQ0_i = 0 \wedge VQ1_i = 0\}$

$S1_i$: $\langle \mathbf{if}\ vq0 + 1 = ts_i \rightarrow x_i, q0, vq0, VQ0_i := q0(0), q0(1..), ts_i, 1\ \mathbf{fi}\rangle$;

$\{I \wedge vq1 < ts_i \leq vq0 \leq clock \wedge x_i = \widehat{q0}(ts_i - (vq0 + 1) + \sum_{0 \leq k < N} VQ0_k)$

$\wedge\ CLOCK_i = 1 \wedge VQ0_i = 1 \wedge VQ1_i = 0\}$

$S2_i$: $\langle \mathbf{if}\ vq1 + 1 = ts_i \rightarrow q1, vq1, VQ1_i := q1 \cdot x_i, ts_i, 1\ \mathbf{fi}\rangle$;

$\{I \wedge ts_i \leq vq1 \leq vq0 \leq clock \wedge CLOCK_i = 1 \wedge VQ0_i = 1 \wedge VQ1_i = 1\}$

$S3_i$: $\langle end(\tau_i')\rangle$

$\{I \wedge ts_i \leq vq1 \leq vq0 \leq clock \wedge cf_i = true \wedge CLOCK_i = 1 \wedge VQ0_i = 1 \wedge VQ1_i = 1\}$

Figure 2.3: Proof Outline $PO(\tau_i')$.

Since $\hat{\tau}_0; \ldots; \hat{\tau}_{N-1} \in SER(\widehat{T1})$,

$$wp(\hat{\tau}_0; \ldots; \hat{\tau}_{N-1}, V1 \equiv_1 \widehat{V1}) \Rightarrow \bigvee_{\hat{\sigma} \in SER(\widehat{T1})} wp(\hat{\sigma}, V1 \equiv \widehat{V1}),$$

and $SD0(\Sigma 1)$ can be inferred from (2.28) using the Rule of Consequence.

Finally, we must show that execution of $\Sigma 1$ terminates when started with $C1 \wedge V1 = \widehat{V1}$. Recall that we have assumed concurrent execution of transactions to be weakly fair. The following lemma provides a general strategy for proving termination under this assumption, and will be used here and in subsequent examples.

**Lemma 2.4.2 (Termination Under Weak Fairness)** If concurrent execution of transactions is weakly fair, execution of any database system $\Sigma$ will terminate if the following two conditions are satisfied.

T1. Every execution of $\Sigma$ consists of a bounded number of atomic operations.

T2. As long as execution of $\Sigma$ has not terminated, there is at least one enabled atomic operation.

□

**Proof of Lemma 2.4.2** Suppose $\Sigma$ has not terminated. Condition T2 guarantees that there must be at least one enabled atomic operation $S$. If no operation runs, then $S$ will be forever delayed in spite of the fact that it is enabled, which would violate the assumption of weak fairness. Thus, some operation will run as long as execution of $\Sigma$ has not terminated. Condition T1 states that there is a bound on the number of operations that can run before $\Sigma$ terminates. From this it follows that $\Sigma$ will eventually terminate. □

Thus, we can prove that execution of $\Sigma 1$ terminates when started with $C1 \wedge V1 = \widehat{V1}$ true by showing that execution of $\Sigma 1$ satisfies conditions T1 and T2 when started with $C1 \wedge V1 = \widehat{V1}$ true.

**Theorem 2.4.3** Execution of $\Sigma 1$ satisfies conditions T1 and T2 of Lemma 2.4.2 when started with $C1 \wedge V1 = \widehat{V1}$ true. □

**Proof of Theorem 2.4.3** Since each $\tau_i'$ of $\Sigma 1$ contains only four atomic operations and does not contain any loops, $\Sigma 1$ trivially satisfies condition T1. Now, we show that $\Sigma 1$ satisfies condition T2. Suppose execution of $\Sigma 1$ has not terminated. Then there must be at least one atomic operation $S$ such that control point preceding $S$ has been reached. Suppose that $S$ is $S0_i$ for some transaction $\tau_i^!$. The states in which $S0_i$ will run to completion are those that satisfy $wp(S0_i, true)$. Since

$$wp(S0_i, true) = true,$$

$S0_i$ will be enabled when it is reached. Thus, condition T2 will be satisfied when $S0_i$ has been reached.

Suppose that $S1_i$ has been reached. Note that

$$\begin{aligned} pre(S1_i) &\Rightarrow (I1 \wedge VQ0_i = 0), \\ &\Rightarrow |q0| > 0. \end{aligned}$$

Thus, $q0$ contains at least one element when $S1_i$ has been reached. In addition,

$$\begin{aligned} pre(S1_i) &\Rightarrow (I3 \wedge vq0 < clock), \\ &\Rightarrow (I3 \wedge vq0 < vq0+1 \leq clock), \\ &= ((\forall v\colon\ vq0 < v \leq clock\colon \\ &\qquad (\exists k\colon\ 0 \leq k < N\colon\ CLOCK_k = 1 \wedge VQ0_k = 0 \wedge v = ts_k)) \\ &\quad \wedge vq0 < vq0+1 \leq clock). \end{aligned}$$

Since $vq0 < vq0+1 \leq clock$ implies that $vq0+1$ satisfies the range of the universal quantifier in $I3$, the quantified expression can be instantiated with $vq0+1$ substituted for $v$. Thus,

$$pre(S1_i) \Rightarrow (\exists k\colon\ 0 \leq k < N\colon\ CLOCK_k = 1 \wedge VQ0_k = 0 \wedge vq0+1 = ts_k) \wedge |q0| > 0.$$

It follows from the interpretation of $FSD0(\Sigma1)$ that when execution of $\Sigma1$ starts with $C1 \wedge V1 = \widehat{V1}$ and reaches $S1_i$, there is some $\tau_k^!$ such that $CLOCK_k = 1 \wedge VQ0_k = 0 \wedge vq0+1 = ts_k \wedge |q0| > 0$. Since $CLOCK_k = 1 \wedge VQ0_k = 0$ implies that the control point before $S1_k$ has been reached and

$$(vq0+1=ts_k \wedge |q0|>0) \Rightarrow wp(S1_k, true),$$

then $S1_k$ in $\tau'_k$ will be enabled when $S1_i$ is reached. Thus, condition T2 will be satisfied when $S1_i$ has been reached.

Suppose that $S2_i$ has been reached. Note that

$$\begin{aligned} pre(S2_i) &\Rightarrow (I4 \wedge vq1 < vq0), \\ &\Rightarrow (I3 \wedge vq1 < vq1+1 \leq vq0), \\ &= ((\forall v\colon\ vq1 < v \leq vq0\colon \\ &\qquad\quad (\exists k\colon\ 0 \leq k < N\colon\ VQ0_k = 1 \wedge VQ1_k = 0 \wedge v = ts_k)) \\ &\quad\ \wedge vq1 < vq1+1 \leq vq0). \end{aligned}$$

Since $vq1 < vq1+1 \leq vq0$ implies that $vq1+1$ satisfies the range of the universal quantifier in $I4$, the quantified expression can be instantiated with $vq1+1$ substituted for $v$, from which it follows that

$$pre(S2_i) \Rightarrow (\exists k\colon\ 0 \leq k < N\colon\ VQ0_k = 1 \wedge VQ1_k = 0 \wedge vq1+1 = ts_k).$$

It follows from the interpretation of $FSD0(\Sigma 1)$ that when execution of $\Sigma 1$ starts with $C1 \wedge V1 = \widehat{V1}$ and reaches $S2_i$, there is some transaction $\tau'_k$ such that $VQ0_k = 1 \wedge VQ1_k = 0 \wedge vq1+1 = ts_k$. Since $VQ0_k = 1 \wedge VQ1_k = 0$ implies that the control point before $S2_k$ has been reached and

$$vq1+1 = ts_k \Rightarrow wp(S2_k, true),$$

then $S2_k$ in $\tau'_k$ will be enabled when $S2_i$ has been reached. Thus condition T2 will be satisfied when $S2_i$ has been reached.

Finally, suppose that $S3_i$ has been reached. Since

$$wp(S3_i, true) = true,$$

$S3_i$ will be enabled and condition T2 will be satisfied.

Thus, execution of $\Sigma 1$ satisfies conditions T1 and T2 of Lemma 2.4.2 when started with $C1 \wedge V1 = \widehat{V1}$ true, and consequently will terminate. $\square$

## 2.5 A More Tractable Method

The preceding example $\Sigma 1$ with Method 2.3.6 is misleading in one respect. Because of the uniformity of transactions in $T1$, all serializable executions of $\Sigma 1$ leave the same values in $q0$ and $q1$. For an arbitrary serializable database system $\Sigma$, however, the size of assertions in the proof of $SD0(\Sigma)$ can be proportional to the number of different serial schedules. If $\Sigma$ contains $N$ transactions, there are $\sum_{0 \leq k \leq N} \binom{N}{k}$ possible serial schedules, a number that quickly grows intractably large.

A more tractable method of proving database systems serializable in Proof Outline Logic can be obtained by moving shadow transactions from the postcondition of the proof outline into the transactions themselves. This is accomplished by constructing an *augmented system*. For $\Sigma = \langle V, C, T, \equiv \rangle$, let $\Sigma^* = \langle V^*, C, T^*, \equiv \rangle$ be the database system in which $V^*$ is the vector obtained by concatenating $V$ and $\widehat{V}$, and $T^* = \{\tau_0^*, \ldots, \tau_{N-1}^*\}$ is a set of ***augmented transactions*** in which each $\tau_i^*$ constructed[7] by replacing some $\langle S_i \rangle$ in $\tau_i$ by $\langle S_i ; \widehat{\tau}_i \rangle$.

For $\sigma^*$ a schedule of $\Sigma^*$, let $\sigma^* |_V$ be the schedule of transactions in $T$ obtained by deleting the operations on $\widehat{V}$ from $\sigma^*$, and let $\sigma^* |_{\widehat{V}}$ be the schedule obtained by deleting the operations on $V$ from $\sigma^*$. Note that $\sigma^* |_V$ is a schedule of $\Sigma$ that transforms variables of $V$ in exactly the same way as $\sigma^*$, and $\sigma^* |_{\widehat{V}}$ is the sequential composition

[7] We define nested angle brackets $\langle \ldots \langle S_k \rangle \ldots \rangle$ to be equivalent to $\langle \ldots S_k \ldots \rangle$.

of shadow transactions that contains $\widehat{\tau}_i$ if and only if $\tau_i$ completes in $\sigma^*|_V$. If each $\tau_i^*$ has been constructed so that operation $\langle S_i;\widehat{\tau}_i\rangle$ runs if and only if $\tau_i$ completes, then the serializability of $\sigma^*|_V$ will follow from the equivalence of $V$ and $\widehat{V}$ after $\sigma^*$ runs.

**Theorem 2.5.1 (Schedule Serializability with Proof Outlines II)** Let $\Sigma^*$ be an augmented system for database system $\Sigma = \langle V, C, T, \equiv \rangle$ in which each $\tau_i^*$ has been constructed so that $\langle S_i;\widehat{\tau}_i\rangle$ runs if and only if $\tau_i$ completes. Let $\sigma^*$ be a schedule of $\Sigma^*$. If

$$\begin{aligned} SS1(\sigma^*)\colon\; & \{ C \wedge V = \widehat{V} \} \\ & \sigma^* \\ & \{ V \equiv \widehat{V} \} \end{aligned}$$

is valid, then $\sigma^*|_V$ is a serializable schedule of $\Sigma$. $\square$

**Proof of Theorem 2.5.1** $SS1(\sigma^*)$ is valid if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*, true)) \Rightarrow wp(\sigma^*, V \equiv \widehat{V}). \tag{2.29}$$

By Lemma 2.3.4, $\sigma^*|_V$ is a serializable schedule of $\Sigma$ if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V, true)) \Rightarrow wp(\sigma^*|_V, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})). \tag{2.30}$$

Thus, the theorem will follow if it can be shown that (2.29) implies (2.30). This is proven as follows.

Since $V$ and $\widehat{V}$ are disjoint, every operation of $\sigma^*|_V$ commutes with every operation of $\sigma^*|_{\widehat{V}}$. From this it follows that

$$\models wp(\sigma^*, R) \Leftrightarrow wp(\sigma^*|_V; \sigma^*|_{\widehat{V}}, R)$$

for any predicate $R$. Due to this and Lemma 2.3.2, (2.29) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, true)) \Rightarrow wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, V \equiv \widehat{V}). \tag{2.31}$$

By Lemma 2.3.3 and Predicate Logic,

$$\begin{aligned} wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, true) &\Leftrightarrow wp(\sigma^*|_V, wp(\sigma^*|_{\widehat{V}}, true)), \\ &\Leftrightarrow wp(\sigma^*|_V, true \wedge wp(\sigma^*|_{\widehat{V}}, true)), \\ &\Leftrightarrow wp(\sigma^*|_V, true) \wedge wp(\sigma^*|_{\widehat{V}}, true). \end{aligned}$$

Thus, (2.31) if and only if

$$\begin{aligned} &\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V, true) \wedge wp(\sigma^*|_{\widehat{V}}, true)) \\ &\quad \Rightarrow wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, V \equiv \widehat{V}). \end{aligned} \tag{2.32}$$

Since conjunction is commutative, (2.32) if and only if

$$\begin{aligned} &\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_{\widehat{V}}, true) \wedge wp(\sigma^*|_V, true)) \\ &\quad \Rightarrow wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, V \equiv \widehat{V}). \end{aligned} \tag{2.33}$$

By Predicate Logic, $(C \wedge V = \widehat{V}) \Rightarrow C^V_{\widehat{V}}$, and because $\sigma^*|_{\widehat{V}}$ is a serial schedule of shadow transactions,

$$C^V_{\widehat{V}} \Rightarrow wp(\sigma^*|_{\widehat{V}}, true).$$

Thus,

$$\begin{aligned} &\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_{\widehat{V}}, true) \wedge wp(\sigma^*|_V, true)) \\ &\quad \Leftrightarrow (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V, true)). \end{aligned}$$

Thus, (2.33) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V, true)) \Rightarrow wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, V \equiv \widehat{V}). \tag{2.34}$$

By definition,

$$wp(\sigma^*|_V;\sigma^*|_{\widehat{V}}, V \equiv \widehat{V}) \Leftrightarrow wp(\sigma^*|_V, wp(\sigma^*|_{\widehat{V}}, V \equiv \widehat{V})).$$

Thus, (2.34) if and only if

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V, true)) \Rightarrow wp(\sigma^*|_V, wp(\sigma^*|_{\widehat{V}}, V \equiv \widehat{V})). \tag{2.35}$$

By construction of $T^*$, $\sigma^*|_{\widehat{V}} \in SER(\widehat{T})$. From this it follows that

$$wp(\sigma^*|_{\widehat{V}}, V \equiv \widehat{V}) \Rightarrow (\bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V}))$$

and by monotonicity of $wp$,

$$wp(\sigma^*|_V, wp(\sigma^*|_{\widehat{V}}, V \equiv \widehat{V})) \Rightarrow wp(\sigma^*|_V, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})).$$

By Predicate Logic, if $A'$ is obtained from $A$ by replacing some occurrence of $B$ by $B'$, and $\models B \Rightarrow B'$, then $\models A \Rightarrow A'$. If $\models A \Rightarrow A'$, then $\models A$ implies $\models A'$. From this it follows that (2.35) implies

$$\models (C \wedge V = \widehat{V} \wedge wp(\sigma^*|_V, true)) \Rightarrow wp(\sigma^*|_V, \bigvee_{\widehat{\sigma} \in SER(\widehat{T})} wp(\widehat{\sigma}, V \equiv \widehat{V})). \tag{2.36}$$

By identity, (2.36) if and only if (2.30). □

Theorem 2.5.1 can be extended to obtain a tractable proof outline whose validity implies that every schedule of $\Sigma$ is serializable.

**Theorem 2.5.2 (System Serializability with Proof Outlines II)** Let $\Sigma^*$ be an augmented system for $\Sigma$ in which each $\tau_i^*$ has been constructed so that $\langle S_i; \widehat{\tau}_i \rangle$ runs if and only if $\tau_i$ completes. $\Sigma$ is a serializable system if execution of $\Sigma^*$ terminates when started with $C \wedge V = \widehat{V}$ true and

$SD1(\Sigma^*)$: $\{ C \wedge V = \widehat{V} \}$
**cobegin** $\tau_0^* \,||\, \cdots \,||\, \tau_{N-1}^*$ **coend**
$\{ V \equiv \widehat{V} \}$

is valid. □

**Proof of Theorem 2.5.2** Since variables $\widehat{V}$ do not occur in transactions of $\Sigma$, execution of $\Sigma$ terminates when started with $C$ true if execution of $\Sigma^*$ terminates when started with $C \wedge V = \widehat{V}$ true. The interpretation of proof outlines and the semantics of **cobegin** imply that $SD1(\Sigma^*)$ is valid if and only if

$$SS1(\sigma^*): \quad \{C \wedge V = \widehat{V}\}\sigma^*\{V \equiv \widehat{V}\}$$

is valid for every schedule $\sigma^*$ of $\Sigma^*$. By Theorem 2.5.1, the validity of $SS1(\sigma^*)$ implies that $\sigma^* \mid_V$ is a serializable schedule. Since every schedule $\sigma^* \mid_V$ is a schedule of $\Sigma$, the validity of $SD1(\Sigma^*)$ implies that every schedule of $\Sigma$ is serializable, and the theorem follows by Definition 2.2.2 of a serializable system. □

Theorem 2.5.2 serves as the basis for a simpler method of proving the serializability of a database system.

**Method 2.5.3 (Proving System Serializability II)** To prove that a system $\Sigma = \langle V, C, T, \equiv \rangle$ is serializable:

1. **Introduce Shadow Variables and Transactions.** Define shadow variables $\widehat{V}$ and construct shadow transactions $\widehat{T}$ corresponding to the variables $V$ and and transactions $T$ of $\Sigma$.

2. **Form Augmented System.** Construct an augmented system $\Sigma^* = \langle V^*, C, T^*, \equiv \rangle$ in which one operation $\langle S_i \rangle$ in each $\tau_i \in T$ is replaced by $\langle S_i; \widehat{\tau}_i \rangle$ and $\widehat{\tau}_i$ runs if and only if $\tau_i$ completes.

3. **Prove $SD1(\Sigma^*)$.** Prove that

$$SD1(\Sigma^*):\quad \{\, C \wedge V = \widehat{V} \}$$
$$\textbf{cobegin}\ \tau_0^* \,||\, \cdots \,||\, \tau_{N-1}^*\ \textbf{coend}$$
$$\{\, V \equiv \widehat{V} \}$$

is valid.

4. **Prove Termination.** Prove execution of $\Sigma^*$ terminates when started with $C \wedge V = \widehat{V}$ true.

□

## 2.6 Examples of the Second Method

We now present two examples that use Method 2.5.3 to prove serializability. In the first example, we give an alternate proof that $\Sigma 1$ of Figure 2.2 is serializable. In the second example, we prove that replacing equivalence relation $\equiv_0$ of $\Sigma 0$ by one that reflects the semantics of a different application results in a database system that is serializable without any synchronization at all.

### 2.6.1 An Alternate Proof of Serializability for $\Sigma 1$

As the first step of Method 2.5.3, we define shadow variables

$$\widehat{V1}:\ \widehat{q0}, \widehat{q1}, \hat{x}_0, \ldots, \hat{x}_{N-1}, \widehat{cf}_0, \ldots, \widehat{cf}_{N-1}, \widehat{clock}, \widehat{vq0}, \widehat{vq1}, \widehat{ts}_0, \ldots, \widehat{ts}_{N-1}$$

corresponding to the variables of $V1$ and construct shadow transactions

$$
\begin{aligned}
&\Sigma 1^* = \langle\, V1^*, C1, T1^*, \equiv_1 \rangle, \\
&\quad V1^* = V1 \cdot \widehat{V1}, \\
&\quad T1^* = \{\tau_0^*, \ldots, \tau_{N-1}^*\} \\
&\qquad \tau_i^* = S0_i\!: \ \langle\, clock, ts_i := clock + 1, clock + 1; \widehat{\tau}_i' \rangle; \\
&\qquad\qquad S1_i\!: \ \langle \mathbf{if}\ vq0 + 1 = ts_i \ \rightarrow\ x_i, q0, vq0 := q0(0), q0(1..), ts_i \ \mathbf{fi} \rangle; \\
&\qquad\qquad S2_i\!: \ \langle \mathbf{if}\ vq1 + 1 = ts_i \ \rightarrow\ q1, vq1 := q1 \cdot x_i, ts_i \ \mathbf{fi} \rangle; \\
&\qquad\qquad S3_i\!: \ \langle\, end(\tau_i^*) \rangle
\end{aligned}
$$

Figure 2.4: Augmented Database System $\Sigma 1^*$

$$
\begin{aligned}
\widehat{\tau}_i'\!: \ & \langle\, \widehat{clock}, \widehat{ts}_i := \widehat{clock} + 1, \widehat{clock} + 1 \,\rangle; \\
& \langle \mathbf{if}\ \widehat{vq0} + 1 = \widehat{ts}_i \ \rightarrow\ \widehat{x}_i, \widehat{q0}, \widehat{vq0} := \widehat{q0}(0), \widehat{q0}(1..), \widehat{ts}_i \ \mathbf{fi} \rangle; \\
& \langle \mathbf{if}\ \widehat{vq1} + 1 = \widehat{ts}_i \ \rightarrow\ \widehat{q1}, \widehat{vq1} := \widehat{q1} \cdot \widehat{x}_i, \widehat{ts}_i \ \mathbf{fi} \rangle; \\
& \langle\, end(\widehat{\tau}_i') \rangle
\end{aligned}
$$

corresponding to each $\tau_i'$.

Next, we construct the augmented system $\Sigma 1^*$ of Figure 2.4. Each $\tau_i^*$ is constructed from $\tau_i'$ by replacing $S0_i$ with $\langle S0_i; \widehat{\tau}_i \rangle$. In this position, shadow transactions will execute in timestamp order. By our assumption that transactions always complete, $S0_i$ is reached if and only if $\tau_i'$ of $\Sigma 1$ completes, as required.

Next, we prove that

$$
\begin{aligned}
SD1(\Sigma 1^*)\!: \ & \{\, C1 \wedge V1 = \widehat{V1} \} \\
& \mathbf{cobegin}\ \tau_0^* \parallel \cdots \parallel \tau_{N-1}^* \ \mathbf{coend} \\
& \{\, V1 \equiv_1 \widehat{V1} \}
\end{aligned}
$$

is valid. To do this, we first construct the full proof outline

$FSD1(\Sigma1^*)$:

$$\{\, C1 \wedge V1 = \widehat{V1}\}$$
$$\langle LQ1, CLOCK_0, VQ0_0, VQ1_0, \ldots, CLOCK_{N-1}, VQ0_{N-1}, VQ1_{N-1}$$
$$:= |q1|, 0, \ldots, 0\rangle;$$
$$\{I \wedge (\forall k\colon\ 0 \le k < N\colon\ 0 = VQ1_k = VQ0_k = CLOCK_k)\}$$
$$\textbf{cobegin } PO(\tau_0^*) \,||\, \cdots \,||\, PO(\tau_{N-1}^*) \textbf{ coend}$$
$$\{\, q0 = \widehat{q0} \wedge q1 = \widehat{q1} \wedge \bigwedge_{0 \le k \le N} cf_k = \widehat{cf}_k \wedge vq0 = \widehat{vq0} \wedge vq1 = \widehat{vq1} \wedge clock = \widehat{clock}\}$$

where each $PO(\tau_i^*)$ is the proof outline for $\tau_i^*$ that shown in Figure 2.5. Auxiliary variables $CLOCK_i$, $VQ0_i$ and $VQ1_i$ are used again to indicate when $\tau_i^!$ has incremented *clock*, $vq0$, and $vq1$, respectively, and an additional auxiliary variable $LQ1$ is used to record the initial length[8] of $q1$.

Each assertion contains the invariant

$$I\colon\ I0 \wedge I1 \wedge I2 \wedge I3 \wedge I4.$$

Here, the first conjunct

$$\begin{aligned} I0\colon\ & \widehat{clock} = \widehat{vq0} = \widehat{vq1} \wedge (\forall k\colon\ 0 \le k < N\colon\ 0 \le VQ1_k \le VQ0_k \le CLOCK_k \le 1) \\ & \wedge clock = \widehat{clock} \wedge vq0 + \sum_{0 \le k < N} CLOCK_k = \widehat{vq0} + \sum_{0 \le k < N} VQ0_k \\ & \wedge vq1 + \sum_{0 \le k < N} CLOCK_k = \widehat{vq1} + \sum_{0 \le k < N} VQ1_k \end{aligned}$$

specifies that $\widehat{clock}$, $\widehat{vq0}$ and $\widehat{vq1}$ remain equal, a result of executing shadow transactions atomically, and bounds values of *clock*, $vq0$ and $vq1$ in terms of the shadow and auxiliary variables. The second conjunct

[8] Note that consistency constraint $C1$ does *not* imply that initially $q1$ is empty.

$$\begin{aligned}
I1\colon\ & |q0| \geq N - \sum_{0 \leq k < N} VQ0_k \wedge |\widehat{q0}| \geq N - \sum_{0 \leq k < N} CLOCK_k \\
& \wedge LQ1 \geq 0 \wedge |\widehat{q1}| = LQ1 + \sum_{0 \leq k < N} CLOCK_k \\
& \wedge q0 = [\widehat{q1}((LQ1 + \sum_{0 \leq k < N} VQ0_k)..)] \cdot \widehat{q0} \\
& \wedge q1 \cdot [\widehat{q1}((LQ1 + \sum_{0 \leq k < N} VQ1_k)..)] = \widehat{q1}
\end{aligned}$$

bounds the size of sequences $q0$, $\widehat{q0}$ and $\widehat{q1}$ and specifies in terms of the auxiliary variables the elements that have been transferred from $q0$ to $q1$. The third conjunct

$$\begin{aligned}
I2\colon\ & \bigwedge_{0 \leq k < N} CLOCK_k = 1 \Rightarrow ts_k \leq clock \\
& \wedge \bigwedge_{0 \leq j \neq k < N} (CLOCK_j = 1 \wedge CLOCK_k = 1) \Rightarrow ts_j \neq ts_k
\end{aligned}$$

specifies that different transactions choose different timestamps, while the fourth and fifth conjuncts

$$I3\colon\ (\forall v\colon\ vq0 < v \leq clock\colon\ (\exists k\colon\ 0 \leq k < N\colon\ CLOCK_k = 1 \wedge VQ0_k = 0 \wedge v = ts_k))$$

and

$$I4\colon\ (\forall v\colon\ vq1 < v \leq vq0\colon\ (\exists k\colon\ 0 \leq k < N\colon\ VQ0_k = 1 \wedge VQ1_k = 0 \wedge v = ts_k))$$

specify that there is a transaction $\tau'_k$ with timestamp $ts_k = v$ for every value $v$ between $vq0 + 1$ and $clock$ or between $vq1 + 1$ and $vq0$.

The proof $FSD1(\Sigma1^*)$ is a straightforward application of the axioms and rules of Proof Outline Logic, and is omitted here. $SD1(\Sigma1^*)$ can be inferred from $FSD1(\Sigma1^*)$ using the Assertion Deletion Rule followed by the Auxiliary Variable Deletion Rule.

Finally, we show that concurrent execution terminates when started in a state satisfying $pre(SD1(\Sigma1^*))$. This proof is exactly the same as the proof of termination in Section 2.4, so we will not repeat it here.

$PO(\tau_i^*)$:

$\{I \wedge vq1 \le vq0 \le clock \wedge CLOCK_i = 0 \wedge VQ0_i = 0 \wedge VQ1_i = 0\}$

$S0_i$: $\langle clock, ts_i, CLOCK_i := clock+1, clock+1, 1; \hat{\tau}_i^! \rangle$;

$\{I \wedge vq1 \le vq0 < ts_i \le clock \wedge CLOCK_i = 1 \wedge VQ0_i = 0 \wedge VQ1_i = 0 \wedge \widehat{cf}_i = true\}$

$S1_i$: $\langle \mathbf{if}\ vq0+1 = ts_i \rightarrow x_i, q0, vq0, VQ0_i := q0(0), q0(1..), ts_i, 1\ \mathbf{fi} \rangle$;

$\{I \wedge vq1 < ts_i \le vq0 \le clock \wedge x_i = \widehat{q1}(LQ1 + ts_i - (vq0+1) + \sum_{0 \le k < N} VQ0_k)$
$\wedge\ CLOCK_i = 1 \wedge VQ0_i = 1 \wedge VQ1_i = 0 \wedge \widehat{cf}_i = true\}$

$S2_i$: $\langle \mathbf{if}\ vq1+1 = ts_i \rightarrow q1, vq1, VQ1_i := q1 \cdot x_i, ts_i, 1\ \mathbf{fi} \rangle$;

$\{I \wedge ts_i \le vq1 \le vq0 \le clock \wedge CLOCK_i = 1 \wedge VQ0_i = 1 \wedge VQ1_i = 1 \wedge \widehat{cf}_i = true\}$

$S3_i$: $\langle end(\tau_i^!) \rangle$

$\{I \wedge ts_i \le vq1 \le vq0 \le clock \wedge CLOCK_i = 1 \wedge VQ0_i = 1 \wedge VQ1_i = 1 \wedge cf_i = \widehat{cf}_i\}$

Figure 2.5: Proof Outline $PO(\tau_i^!)$.

### 2.6.2 Sequence Variables with Set Semantics

Database systems in which variables are instances of abstract datatypes are considered in [SS84], where it is shown that by ignoring parts of the state that do not produce visible differences in the values of the abstract datatypes implemented, a larger set of schedules can be considered serializable. This view of serializability can be formalized in our system model by using the equivalence relation. We illustrate this by considering the transactions of the database system $\Sigma 0$ described in Section 2.1 in a context in which sequence variables $q0$ and $q1$ are viewed as implementing *sets*.

Recall, $\Sigma 0$ models an application in which a series of independent events move elements of $q0$ to $q1$. Suppose $q0$ and $q1$ are treated as unordered collections instead of as queues. Database system $\Sigma 2$ of Figure 2.6 models this situation. Note that the

$$\Sigma 2 = \langle V2, C2, T2, \equiv_2 \rangle$$
$$V2 = \langle q0, q1, x_0, \ldots, x_{N-1}, cf_0, \ldots, cf_{N-1} \rangle,$$
$$C2 = (\{q0\} \cup \{q1\} = Q \wedge |q0| \geq (\#\, k\colon\ 0 \leq k < N\colon\ cf_k = \mathit{false})),$$
$$T2 = \{\tau_0, \ldots, \tau_{N-1}\},$$
$$\tau_i = S1_i\colon\ \langle x_i, q0 := q0(0), q0(1..) \rangle;$$
$$S2_i\colon\ \langle q1 := q1 \cdot x_i \rangle;$$
$$S3_i\colon\ \langle \mathit{end}(\tau_i) \rangle$$
$$(V2' \equiv_2 V2'') \Leftrightarrow (\{q0'\} = \{q0''\} \wedge \{q1'\} = \{q1''\} \wedge \bigwedge_{0 \leq k < N} cf'_k = cf''_k)$$

Figure 2.6: Database System $\Sigma 2$.

variables $V2$ and transactions $T2$ of $\Sigma 2$ are the same as $V0$ and $T0$ of $\Sigma 0$. However, the consistency constraint $C0$ of $\Sigma 0$ has been replaced by the weaker constraint

$$C2\colon\ \{q0\} \cup \{q1\} = Q \wedge |q0| \geq (\#\, k\colon\ 0 \leq k < N\colon\ cf_k = \mathit{false})$$

and the equivalence relation $\equiv_0$ has been replaced by the weaker relation

$$(V2' \equiv_2 V2'') \Leftrightarrow (\{q0'\} = \{q0''\} \wedge \{q1'\} = \{q1''\} \wedge \bigwedge_{0 \leq k < N} cf'_k = cf''_k)$$

to reflect that the order of elements within $q0$ and $q1$ is no longer significant.

For any initial state satisfying $C2$, any schedule $\sigma$ of $T2$ will leave a consistent state in which $q0$ and $q1$ contain the same elements as they would after some serial schedule $\sigma'$, although $\sigma$ and $\sigma'$ might order the elements of $q1$ differently. Consequently, every schedule of $\Sigma 2$ will be serializable under Definition 2.2.1, as is easily proven using Method 2.5.3.

First, we introduce shadow variables

$$\widehat{V2}\colon\ \langle \widehat{q0}, \widehat{q1}, \hat{x}_0, \ldots, \hat{x}_{N-1}, \widehat{cf}_0, \ldots, \widehat{cf}_{N-1} \rangle$$

$$\begin{aligned}
&\Sigma 2^* = \langle V2^*, C2, T2^*, \equiv_2 \rangle,\\
&\quad V2^* = V2 \cdot \widehat{V2},\\
&\quad \widehat{V2} = \langle \widehat{q0}, \widehat{q1}, \hat{x}_0, \ldots, \hat{x}_{N-1}, \widehat{cf}_0, \ldots, \widehat{cf}_{N-1} \rangle,\\
&\quad T2^* = \{\tau_0^*, \ldots, \tau_{N-1}^*\}\\
&\qquad \tau_i^* = S1_i\text{:}\ \ \langle x_i, q0 := q0(0), q0(1..); \hat{\tau}_i \rangle;\\
&\qquad\qquad\ \ S2_i\text{:}\ \ \langle q1 := q1 \cdot x_i \rangle;\\
&\qquad\qquad\ \ S3_i\text{:}\ \ \langle end(\tau_i^*) \rangle
\end{aligned}$$

Figure 2.7: Augmented Database System $\Sigma 2^*$.

and construct shadow transactions

$$\begin{aligned}
\hat{\tau}_i\text{:}\ \ &\langle \hat{x}_i, \widehat{q0} := \widehat{q0}(0), \widehat{q0}(1..) \rangle;\\
&\langle \widehat{q1} := \widehat{q1} \cdot \hat{x}_i \rangle;\\
&\langle end(\hat{\tau}_i) \rangle
\end{aligned}$$

for each $\tau_i$ of $\Sigma 2$.

Next, we construct augmented system $\Sigma 2^*$ of Figure 2.7.

Next, we prove that

$$\begin{aligned}
SD1(\Sigma 2^*)\text{:}\ \ &\{\, C2 \wedge V2 = \widehat{V2} \}\\
&\textbf{cobegin}\ \tau_0^* \parallel \cdots \parallel \tau_{N-1}^*\ \textbf{coend}\\
&\{\, V2 \equiv \widehat{V2} \}
\end{aligned}$$

is valid. This follows by the Assertion Deletion and Auxiliary Variable Deletion Rules from the validity of the full proof outline

$$
\begin{aligned}
PO(\tau_i^*)\colon\ & \{I \wedge D_i = \{\}\} \\
& S1_i\colon\ \langle x_i, q0, D_i := q0(0), q0(1..), \{q0(0)\}; \hat{\tau}_i \rangle; \\
& \{I \wedge D_i = \{x_i\} \wedge \widehat{cf}_i = true\} \\
& S2_i\colon\ \langle q1, D_i := q1 \cdot x_i, \{\} \rangle; \\
& \{I \wedge D_i = \{\} \wedge \widehat{cf}_i = true\} \\
& S3_i\colon\ \langle end(\tau_i^*) \rangle \\
& \{I \wedge D_i = \{\} \wedge cf_i = \widehat{cf}_i\}
\end{aligned}
$$

Figure 2.8: Proof Outline for $PO(\tau_i^*)$ for $\tau_i^*$ of $\Sigma 2^*$.

$$
\begin{aligned}
& FSD1(\Sigma 2^*)\colon \\
& \quad \{\, C2 \wedge V2 = \widehat{V2}\} \\
& \quad \langle D_0, \ldots, D_{N-1} := \{\}, \ldots, \{\} \rangle; \\
& \quad \{I \wedge \bigwedge_{0 \le k < N} D_k = \{\}\} \\
& \quad \textbf{cobegin}\ PO(\tau_0^*) \parallel \cdots \parallel PO(\tau_{N-1}^*)\ \textbf{coend} \\
& \quad \{\{q0\} = \{\widehat{q0}\} \wedge \{q1\} = \{\widehat{q1}\} \wedge \bigwedge_{0 \le k < N} cf_k = \widehat{cf}_k\}
\end{aligned}
$$

where each $PO(\tau_i^*)$ is the proof outline for $\tau_i^*$ given in Figure 2.8. Auxiliary variables $D_i$ have been added to indicate the elements of $Q$ that have been deleted from $q0$ but not yet added to $q1$. Each assertion contains the invariant

$$
I\colon\ q0 = \widehat{q0} \wedge (\{q1\} \cup \bigcup_{0 \le k < N} D_k) = \{\widehat{q1}\} \wedge |\widehat{q0}| \ge (\# k\colon\ 0 \le k < N\colon\ \widehat{cf}_k = false).
$$

The proof of $FSD1(\Sigma 2^*)$ is straightforward and therefore is omitted here. From $FSD1(\Sigma 2^*)$, $SD1(\Sigma 2^*)$ can be inferred by applying the Assertion Deletion Rule followed by the Auxiliary Variable Deletion Rule.

Finally, we must show that execution of $\Sigma 2^*$ terminates when started with $C2 \wedge V2 = \widehat{V2}$ true. To do this, we use Lemma 2.4.2, which states that under the assumption that concurrent execution of transactions is weakly fair, execution of $\Sigma 2^*$ will terminate if conditions T1 and T2 are satisfied.

**Theorem 2.6.1** When started with $C2 \wedge V2 = \widehat{V2}$ true, execution of $\Sigma 2^*$ satisfies conditions T1 and T2. □

**Proof of Theorem 2.6.1** Since each $\tau_i$ of $\Sigma 2^*$ contains only four atomic operations and does not contain any loops, execution of $\Sigma 2$ trivially satisfies condition T1.

Suppose that execution of $\Sigma 2^*$ has not terminated. Then there must be at least one atomic operation $S$ such that control point preceding $S$ has been reached. Since $FSD1(\Sigma 2^*)$ is valid, $pre(S)$ will be true when $S$ is reached. Since

$$pre(S) \Rightarrow wp(S, true)$$

for every $S$ in $FSD1(\Sigma 2^*)$, then $S$ will be enabled when it is reached, and condition T2 will be satisfied. □

## 2.7 Incompleteness of the Second Method for Proving Serializability

The characterization of serializability in terms of proof outlines given by Theorem 2.3.5 is complete. This is because the property that database system $\Sigma$ is serializable is equivalent to the properties specified by the theorem's hypotheses, namely (i) $SD0(\Sigma)$ is a valid proof outline and (ii) execution of $\Sigma$ begun in a state satisfying $pre(SD0(\Sigma))$ is guaranteed to terminate. Because it is derived from Theorem 2.3.5, Method 2.3.6 for proving the serializability of database systems is complete relative to the method with which validity of $SD0(\Sigma)$ and termination of $\Sigma$ are proven.

$$\begin{aligned}
&\Sigma 3 = \langle V3, C3, T3, \equiv_3 \rangle \\
&\quad V3 = \langle x_0, x_1, x_2, x_3, rx_0, rx_1, rx_2, cf_0, cf_1, cf_2 \rangle, \\
&\quad C3 = true, \\
&\quad T3 = \{\tau_0, \tau_1, \tau_2\}, \\
&\qquad \tau_i = S1_i\colon\ \langle rx_i := x_i \rangle; \\
&\qquad\qquad S2_i\colon\ \langle x_{i+1} := rx_i + 1 \rangle; \\
&\qquad\qquad S3_i\colon\ \langle end(\tau_i) \rangle \\
&\quad (V3' \equiv_3 V3'') \Leftrightarrow (V3' = V3'').
\end{aligned}$$

Figure 2.9: Database System $\Sigma 3$.

In contrast, the characterization of serializability in terms of proof outline that is given by Theorem 2.5.2 is not complete—there are serializable database systems $\Sigma$ for which it is not possible to construct an augmented system $\Sigma^*$ such that $SD1(\Sigma^*)$ is valid. For such systems, it will be impossible to use Method 2.5.3 to prove serializability. An example of such a system is $\Sigma 3$ of Figure 2.9.

$\Sigma 3$ is serializable, but there is no way to construct an augmented system $\Sigma 3^*$ such that $SD1(\Sigma 3^*)$ is valid. To see this, assume the contrary. Thus, assume there is an augmented system $\Sigma 3^*$ for which $SD1(\Sigma 3^*)$ is valid. Consider the schedule

$$\sigma 7^*\colon\ S1_0; S1_1; S2_0; S3_0; S1_2 : S2_1; S3_1; S2_2; S3_2$$

of $\Sigma 3^*$. In $\sigma 7^*$, note that $S1_1$, which reads $x_1$, precedes $S2_0$, which writes $x_1$. Consequently, $rx_1$ will be left holding the initial value of $x_1$. Likewise, $S1_2$, which reads $x_2$, precedes $S2_1$, which writes $x_2$, and so $rx_2$ will be left holding the initial value of $x_2$.

From the validity of $SD1(\Sigma 3^*)$ and the interpretation of proof outlines, it follows that

$$\{C3 \wedge V3 = \widehat{V3}\}\sigma 7^* \{V3 = \widehat{V3}\} \tag{2.37}$$

is valid. This implies that shadow transactions $\widehat{\tau}_0$, $\widehat{\tau}_1$ and $\widehat{\tau}_2$ in $\sigma 7^*$ run in an order that leaves $\widehat{rx}_1$ and $\widehat{rx}_2$ storing the initial values of $\widehat{x}_1$ and $\widehat{x}_2$, respectively.

Since the last operation of $\tau_0^*$ precedes the first operation of $\tau_2^*$ in $\sigma 7^*$, $\widehat{\tau}_0$ will run before $\widehat{\tau}_2$, regardless of how the augmented system $\Sigma 3^*$ is constructed. Thus, $\sigma 7^*|_{\widehat{V3}}$ must be one of the schedules $\widehat{\tau}_0;\widehat{\tau}_2;\widehat{\tau}_1$, $\widehat{\tau}_0;\widehat{\tau}_1;\widehat{\tau}_2$ or $\widehat{\tau}_1;\widehat{\tau}_0;\widehat{\tau}_2$. In the first and second cases, $\widehat{rx}_1$ will be left holding a value one greater than the initial value of $\widehat{x}_1$, while in the third case, $\widehat{rx}_2$ will be left holding a value one greater than the initial value of $\widehat{x}_2$. This contradicts the values of $\widehat{rx}_1$ and $\widehat{rx}_2$ inferred from the validity of $SD1(\Sigma 3^*)$. Thus, $SD1(\Sigma 3^*)$ cannot be valid.

Incompleteness of Method 2.5.3 arises because shadow transactions can model only limited serial behavior when they are used to construct an augmented system. In any schedule $\sigma^*$ of an augmented system $\Sigma^*$, each shadow transaction $\widehat{\tau}_i$ runs during execution of the augmented transaction $\tau_i^*$ that contains it. If $\tau_i^*$ and $\tau_j^*$ do not interleave with each other in $\sigma^*$, then the order in which $\widehat{\tau}_i$ and $\widehat{\tau}_j$ run will be the same as that of $\tau_i^*$ and $\tau_j^*$. For this reason, the proof outline $SD1(\Sigma^*)$ of Method 2.5.3 specifies that every schedule $\sigma$ of the original system $\Sigma$ behaves like a serial schedule $\sigma'$ in which the order of transactions is consistent with the order of non-interleaved transaction in $\sigma$. Database system $\Sigma 3$ demonstrates that not every serializable database system satisfies this property, and consequently not every database system can be proven serializable using Method 2.5.3. In spite of this, the tractability Method 2.5.3 compared to Method 2.3.6 makes in preferable in situations where it suffices.

## 2.8 Discussion

### 2.8.1 Comparing System Models

By constructing transactions from appropriately chosen atomic operations, the system model presented in Section 2.1 can model any of the database system models described in Section 1.3. For example, a system implementing read and write operations such as those used to construct the transactions of Figure 1.1 can be modeled by using an atomic operation $\langle\, t := a[i] \,\rangle$ to denote each read operation $r(a[i], t)$ and $\langle\, a[i] := e \,\rangle$ to denote each write operation $w(a[i], e)$.

Explicit synchronization is represented in our database system model by including synchronizing operations among the atomic operations from which transactions are constructed. Implicit synchronization is modeled in one of two ways: either by introducing a scheduler process to which transactions make operation requests, or by modifying transaction operations to perform the function of the scheduler themselves. This second approach was illustrated in Section 2.4, when the synchronized database system $\Sigma 1$ was constructed from $\Sigma 0$.

### 2.8.2 Comparing Definitions of Serializability

In Section 1.3, we divided definitions of serializability into two classes, those characterizing schedule behavior in terms of conflict relations and those characterizing it in terms of state transformations. Definitions 2.2.1 and 2.2.2 generalize in two ways definitions in the second class. One generalization results from the inclusion of the equivalence relation for database system state equality. This allows various criteria by which previous definitions of serializability compare system states to be represented. For example,

$$
\begin{aligned}
&\Sigma 4 = \langle V4, C4, T4, \equiv_4 \rangle \\
&\quad V4 = \langle x, y, b, z, cf_0, cf_1 \rangle, \\
&\quad C4 = (x + y = 1000), \\
&\quad T4 = \{\tau_0, \tau_1\} \\
&\qquad \tau_0 = A1: \ \langle \textbf{if } b \rightarrow y := y + 17 \,[\!]\, \neg b \rightarrow x := x - 17 \textbf{ fi} \rangle; \\
&\qquad \phantom{\tau_0 = {}} A2: \ \langle \textbf{if } b \rightarrow x := x - 17 \,[\!]\, \neg b \rightarrow y := y + 17 \textbf{ fi} \rangle; \\
&\qquad \phantom{\tau_0 = {}} A3: \ \langle \mathit{end}(\tau_0) \rangle \\
&\qquad \tau_1 = B1: \ \langle z := x \rangle; \\
&\qquad \phantom{\tau_1 = {}} B2: \ \langle \mathit{end}(\tau_1) \rangle \\
&\quad (V4' \equiv_4 V4'') \Leftrightarrow (x' = x'' \wedge y' = y'' \wedge b' = b'' \wedge z' = z'' \wedge cf_0' = cf_0'' \wedge cf_1' = cf_1'')
\end{aligned}
$$

Figure 2.10: Database System $\Sigma 4$.

final-state serializability can be represented by Definition 2.2.2 by choosing $\equiv$ so that $V' \equiv V''$ if and only if $V'$ and $V''$ agree on the final values of every shared variable of the system; view serializability can be represented by adding auxiliary variables to transactions to record the value obtained by read operations and choosing $\equiv$ so that $V' \equiv V''$ if and only if $V'$ and $V''$ agree on the final values of both the shared variables and the added auxiliary variables.

A second source of generality in Definitions 2.2.1 and 2.2.2 results from the use of *wp* to compare the way in which schedules transform the system from one state to another. A schedule that is final-state or view serializable is required to "behave like" a particular serial schedule; a schedule serializable according to Definition 2.2.1 is allowed to "behave like" different serial schedules depending on the initial state. To see this, consider $\Sigma 4$ of Figure 2.10. In particular, consider the non-serial schedule

$\sigma 8$: $A1; B1; B2; A2; A3$

of $\tau_0$ and $\tau_1$.

The two possible serial schedules of $\Sigma 4$ are

$\sigma 9$: $A1; A2; A3; B1; B2$

and

$\sigma 10$: $B1; B2; A1; A2; A3$.

Schedule $\sigma 8$ is neither final state nor view serializable because neither $\sigma 9$ nor $\sigma 10$ individually produces the same final state as $\sigma 8$ for *every* consistent initial state. In particular, $\sigma 8$ produces different values of $z$ depending on whether $b = true$ or $b = false$ initially.

According to Definition 2.2.1, $\sigma 8$ is serializable if and only if

$$\models (C4 \wedge wp(\sigma 8, V4 \equiv_4 \widetilde{V4})) \Rightarrow (wp(\sigma 9, V4 \equiv_4 \widetilde{V4}) \vee wp(\sigma 10, V4 \equiv_4 \widetilde{V4})). \quad (2.38)$$

Computing $wp(\sigma 9, V4 \equiv_4 \widetilde{V4})$ and $wp(\sigma 10, V4 \equiv_4 \widetilde{V4})$ using the rules of Appendix B gives

$$\begin{aligned} &wp(\sigma 9, V4 \equiv_4 \widetilde{V4}) \\ &\quad = (x - 17 = \tilde{x} \wedge y + 17 = \tilde{y} \wedge b = \tilde{b} \wedge x - 17 = \tilde{z} \wedge true = \widetilde{cf}_0 \wedge true = \widetilde{cf}_1) \end{aligned}$$

and

$$\begin{aligned} &wp(\sigma 10, V4 \equiv_4 \widetilde{V4}) \\ &\quad = (x - 17 = \tilde{x} \wedge y + 17 = \tilde{y} \wedge b = \tilde{b} \wedge x = \tilde{z} \wedge true = \widetilde{cf}_0 \wedge true = \widetilde{cf}_1). \end{aligned}$$

Computing $wp(\sigma 8, V4 \equiv_4 \widetilde{V4})$ gives

$$
\begin{aligned}
& wp(\sigma 8, V4 \equiv_4 \widetilde{V4}) \\
& \quad = [(b \Rightarrow (x-17=\tilde{x} \wedge y+17=\tilde{y} \wedge b=\tilde{b} \wedge x=\tilde{z} \wedge true=\widetilde{cf}_0 \wedge true=\widetilde{cf}_1) \\
& \quad\quad \wedge (\neg b \Rightarrow (x-17=\tilde{x} \wedge y+17=\tilde{y} \wedge b=\tilde{b} \wedge x-17=\tilde{z} \wedge \\
& \quad\quad\quad true=\widetilde{cf}_0 \wedge true=\widetilde{cf}_1))], \\
& \quad = [(b \Rightarrow wp(\sigma 10, V4 \equiv_4 \widetilde{V4})) \wedge (\neg b \Rightarrow wp(\sigma 9, V4 \equiv_4 \widetilde{V4}))].
\end{aligned}
$$

For any predicates $P$ and $Q$, it follows tautologically that

$$[C4 \wedge (b \Rightarrow P) \wedge (\neg b \Rightarrow Q)] \Rightarrow [P \vee Q].$$

Taking $P$ to be $wp(\sigma 10, V4 \equiv_4 \widetilde{V4})$ and $Q$ to be $wp(\sigma 9, V4 \equiv_4 \widetilde{V4})$, (2.38) follows trivially. Thus, $\sigma 8$ is serializable according to Definition 2.2.1.

One previous definition of serializability that is similar to ours can be found in [C81]. Like Definition 2.2.2, the definition of [C81] characterizes system behavior by the way in which the system state is transformed. Our definition and that of [C81] also share the property that final states are compared using an equivalence relation on states, although the equivalence relations that can be specified in [C81] are limited to those having the form

$$(V' \equiv V'') \Leftrightarrow (U' = U'')$$

for some vector of variables $U$ containing a subset of those appearing in $V$. subset of those that can be specified in our definition.

However, a more significant difference between the two definitions of serializability is in the formalism chosen to describe the way in which concurrent execution transforms the system from one state to another. Instead of using $wp$ to describe program semantics as we do here, the definition of [C81] uses an extension of Dynamic Logic [FL79,H79] called Concurrent Dynamic Logic [P87] to reason formally about concurrent program semantics. The main advantage using *wp* in Definitions 2.2.1 and 2.2.2 is that it provides a better foundation for translating Definition 2.2.2 into Proof Outline Logic, and Proof Outline Logic is more familiar to programmers than Concurrent Dynamic Logic.

# Chapter 3

# Deriving Locking Protocols

Using Method 2.3.6 or Method 2.5.3 of Chapter 2, a proof that a database system $\Sigma$ is serializable is partitioned into a proof of a safety property (partial correctness specified by $SD0(\Sigma)$ or $SD1(\Sigma^*)$) and a proof of a liveness property (termination). A benefit of specifying the safety property associated with serializability as a proof outline is that it becomes possible to verify formally synchronization protocols.

In this chapter, we illustrate a second benefit of partitioning the proof of serializability in this way—the ability to *derive* synchronization protocols for serializability, using obligations that arise in the proof of correctness to motivate the introduction of synchronizing operations. We start with a general discussion of how synchronization requirements can be derived while constructing a proof outline. We then present an assertional characterization of locking that allows these synchronization requirements to be satisfied by using locking operations.

## 3.1 Proofs of Concurrent Programs

Before considering the specific problem of deriving synchronization to ensure serializability of database systems, we first examine how proof outlines for concurrent programs are constructed. Note that both $SD0(\Sigma)$ of Method 2.3.6 and $SD1(\Sigma^*)$ of Method 2.5.3 have the general form

$$PO(\Sigma):\ \ \{Q\}\,\mathbf{cobegin}\ \tau_0 \mid\mid \cdots \mid\mid \tau_{N-1}\ \mathbf{coend}\,\{R\}. \tag{3.1}$$

In Proof Outline Logic, $PO(\Sigma)$ is inferred in two steps. First, the **cobegin** Rule (summarized with other rules and axioms of Proof Outline Logic in Appendix A) is applied to obtain a full proof outline

$$FPO(\Sigma):\ \ \{Q\}\,\mathbf{cobegin}\ PO(\tau_0) \mid\mid \cdots \mid\mid PO(\tau_{N-1})\ \mathbf{coend}\,\{R\}. \tag{3.2}$$

Then, the Assertion Deletion Rule is applied to delete intermediate assertions from $FP0(\Sigma)$ to obtain $PO(\Sigma)$.

The **cobegin** Rule requires $FPO(\Sigma)$ to satisfy four hypotheses. The first hypothesis ensures that each $PO(\tau_i)$ itself is a valid proof outline:

$H1$: $PO(\tau_0),\ldots,PO(\tau_{N-1})$ are valid proof outlines.

The second hypothesis of the **cobegin** Rule ensures that the precondition of each $PO(\tau_i)$ will be true when concurrent execution starts:

$H2$: $Q \Rightarrow (pre(PO(\tau_0)) \wedge \ldots \wedge pre(PO(\tau_{N-1})))$.

The third hypothesis ensures that $R$ will be true when concurrent execution terminates with the postcondition of each $PO(\tau_i)$ true:

$H3$: $(post(PO(\tau_0)) \wedge \ldots \wedge post(PO(\tau_{N-1}))) \Rightarrow R$.

The last hypothesis of the **cobegin** Rule is called *interference freedom* [OG76]. For $\alpha$ an atomic operation and $A$ an assertion in $FPO(\Sigma)$, $\alpha$ is *parallel* to $A$ (denoted $\alpha \mid\mid A$) if $\alpha$ occurs in one transaction and $A$ occurs in the proof outline of another. Interference freedom ensures that no atomic operation invalidates an assertion to which it is parallel:

$H4$: $(\forall \alpha, A\colon \alpha \mid\mid A\colon NI(\alpha, A)\colon \ \{pre(\alpha) \wedge A\}\alpha\{A\})$.

When $NI(\alpha, A)$ is not valid for some $\alpha$ parallel to $A$, we say that $\alpha$ ***interferes with*** $A$.

A full proof outline satisfying the hypotheses of the **cobegin** Rule can be constructed by a step-wise derivation in which sequential proof outlines are chosen to satisfy some of the hypotheses initially, and are transformed in a series of steps until they satisfy the remaining hypotheses [SA87].

**Method 3.1.1 (Deriving Full Proof Outlines)** To derive a full proof outline

$$FPO(\Sigma)\colon \ \{Q\}\textbf{cobegin}\, PO(\tau_0) \mid\mid \cdots \mid\mid PO(\tau_{N-1})\,\textbf{coend}\{R\}$$

that satisfies the hypotheses of the **cobegin** Rule, do the following.

1. **Construct Sequential Proof Outlines.** Construct valid sequential proof outlines $PO(\tau_0),\ldots,PO(\tau_{N-1})$ in satisfying hypotheses H1 and H2.

2. **Eliminate Interference.** While hypothesis H4 remains unsatisfied, do the following.

   (a) Enumerate and check the interference freedom formulas.

   (b) Choose an invalid $NI(\alpha, A)$ for $\alpha$ in $PO(\tau_i)$ and $A$ in $PO(\tau_j)$ and do one of the following.

- **Strengthen** $pre(\alpha)$. Replace $pre(\alpha)$ by a stronger assertion[1] $pre(\alpha)'$ such that the interference freedom formula $\{pre(\alpha)' \wedge A\}\,\alpha\,\{A\}$ is valid, strengthening assertions that precede $pre(\alpha)'$ as necessary to ensure that $PO(\tau_i)$ remains valid.
- **Weaken** $A$. Replace $A$ by a weaker assertion[2] $A'$ such that the interference freedom formula $\{pre(\alpha) \wedge A'\}\,\alpha\,\{A'\}$ is valid, weakening assertions the follow $A'$ as necessary to ensure $PO(\tau_j)$ remains valid.

3. Check that the resulting proof outlines satisfy hypothesis H3.

□

## 3.2 Interference and Synchronization

Even when $PO(\Sigma)$ of Equation 3.1 is not valid, it is often possible to derive synchronization that ensures that $PO(\Sigma)$ is valid by examining where constructing $FPO(\Sigma)$ with Method 3.1.1 fails.

It will always be possible to construct proof outlines $PO(\tau_0),\ldots,PO(\tau_{N-1})$ that satisfy hypotheses H1 and H2 as specified by the first step of the method. Suppose that in the second step, an invalid triple $NI(\alpha,A)$ for $\alpha$ in $PO(\tau_i)$ and $A$ in $PO(\tau_j)$ is discovered. Two options are available: replacing $pre(\alpha)$ by the stronger assertion $pre(\alpha) \wedge (\neg A \vee wp(\alpha,A))$ or replacing $A$ by the weaker assertion $A \vee post(\alpha)$. However, the other hypotheses of the **cobegin** Rule effectively limit the strength of $pre(\alpha)$ and

[1] $pre(\alpha)'$ is stronger than $pre(\alpha)$ if $pre(\alpha)' \Rightarrow pre(\alpha)$ and $pre(\alpha) \not\Rightarrow pre(\alpha)'$.
[2] $A'$ is weaker than $A$ if $A \Rightarrow A'$ and $A' \not\Rightarrow A$.

the weakness of $A$.

In particular, hypothesis H2 limits how strong $pre(PO(\tau_i))$ can be made. Because H1 requires $PO(\tau_i)$ to remain valid, the strength of $pre(PO(\tau_i))$ effectively limits the strength of $pre(\alpha)$ and other assertions that follow $pre(PO(\tau_i))$. In a similar manner, hypothesis H3 limits how weak $post(PO(\tau_j))$ can be made, and consequently how weak $A$ and other assertions preceding $post(PO(\tau_j))$ can be made. Because of these limitations, it is possible to reach a point in Method 3.1.1 at which an invalid interference freedom formula $NI(\alpha, A)$ has been identified, but $pre(\alpha)$ cannot strengthened or $A$ weakened enough to eliminate this interference without making it impossible to satisfy one of the hypotheses H1 through H3.

Such conflicts can be overcome if a method of selectively strengthening assertions can be found. With such a method, $pre(\alpha)$ could be strengthened enough to eliminate interference while assertions that precede it are left weak enough to ensure that other hypotheses remain satisfied. Likewise, $A$ could be made weak enough to eliminate interference while assertions that follow $A$ are strengthened to ensure that other hypotheses remain satisfied.

In the remainder of this chapter, we will demonstrate how locking can be used to implement synchronization required to do this. We first show how locking can be used to ensure that concurrent execution of transactions preserves a certain type of invariant, called an *exclusion invariant*. We then show how the problem of strengthening assertions selectively can be reduced to one of preserving invariants of this type.

## 3.3 Exclusion Invariants

A locking protocol $\Lambda$ can be specified as a triple $\langle M, LC, R\rangle$, where $M$ is a set of lock modes, $LC$ is the lock compatibility relation on these modes, and $R$ is the set of rules that transactions must follow when acquiring and releasing locks. A lock with mode "$M$" is denoted $\ell_{[M]}$. Locking protocols described in the literature often use locks that are associated with system variables. In read-write locking protocols, for example, each read or write lock is associated with a particular variable or set of variables. Locks associated with variables can be formulated in our notation by including associated variables in the mode of the lock. For example, read and write locks on $x$ can be denoted $\ell_{[R(x)]}$ and $\ell_{[W(x)]}$, respectively.

The set of locks held by a transaction $\tau_i$ is denoted $ls_i$. Lock compatibility relation $LC$ is a predicate on the lock sets of the transactions of $\Sigma$. To add locks to $ls_i$, $\tau_i$ *acquires* them with the operation

$$\mathbf{acq}(\ell_{[M_0]},\ldots,\ell_{[M_k]}),$$

and to remove locks from $ls_i$, $\tau_i$ *releases* them with the operation

$$\mathbf{rel}(\ell_{[M_0]},\ldots,\ell_{[M_k]}).$$

The predicate

$$ls_i \supseteq \{\ell_{[M_0]},\ldots,\ell_{[M_k]}\}$$

is true if and only if $\tau_i$ has acquired locks $\ell_{[M_0]},\ldots,\ell_{[M_k]}$ but not yet released them.

A database system synchronized with a locking protocol can be denoted by a pair $\langle\Lambda,\Sigma\rangle$. Here, $\Lambda = \langle M, LC, R\rangle$ is locking protocol and $\Sigma = \langle V, C, T, \equiv\rangle$ is a database system such that $V$ contains the lock set $ls_i$ for each transaction $\tau_i \in T$, $C \Rightarrow LC$ and each transaction $\tau_i \in T$ follows the rules of $R$.

The *local state* of a transaction $\tau_i$ is the part of the system state that only $\tau_i$ can modify. For example, variables that only $\tau_i$ can modify are components of the local state of $\tau_i$, as is the value of the program counter of $\tau_i$. A predicate $LP$ is *local* to $\tau_i$ if $LP$ is a predicate on the local state of $\tau_i$.

An *exclusion invariant* is a predicate of the form

$$XI: \ \neg(LP \wedge LQ),$$

where $LP$ and $LQ$ are predicates local to different transactions $\tau_i$ and $\tau_j$, respectively. Locking implements synchronization that prevents sections of different transactions from interleaving with each other. If we view locking assertionally, locking protocols can be used to preserve exclusion invariants. This is accomplished by giving modes and rules that couple the local state of transactions to the sets of locks they hold, and lock compatibility relations that provide the necessary exclusion. The following theorem suggests a way to do this.

**Theorem 3.3.1** Let $LP$ and $LQ$ be local predicates of transactions $\tau_i$ and $\tau_j$ of $\Sigma$, and let "$M0$" and "$M1$" be modes of the locking protocol $\Lambda$ of $\Sigma$. Then

$$LP \Rightarrow ls_i \supseteq \{\ell_{[M0]}\}, \tag{3.3}$$

$$LQ \Rightarrow ls_j \supseteq \{\ell_{[M1]}\} \tag{3.4}$$

and

$$\neg(ls_i \supseteq \{\ell_{[M0]}\} \wedge ls_j \supseteq \{\ell_{[M1]}\}) \tag{3.5}$$

imply

$$\neg(LP \wedge LQ).$$

$\square$

**Proof of Theorem 3.3.1** By Predicate Logic, (3.3) and (3.4) imply

$$(LP \wedge LQ) \Rightarrow (ls_i \supseteq \{\ell_{[M0]}\} \wedge ls_j \supseteq \{\ell_{[M1]}\}). \tag{3.6}$$

Since $A \Rightarrow B$ if and only if $\neg B \Rightarrow \neg A$, (3.6) implies

$$\neg(ls_i \supseteq \{\ell_{[M0]}\} \wedge ls_j \supseteq \{\ell_{[M1]}\}) \Rightarrow \neg(LP \wedge LQ).$$

From this it follows that (3.3), (3.4) and (3.5) imply

$$\neg(LP \wedge LQ).$$

$\square$

From Theorem 3.3.1 follows a method for using locking to guarantee an exclusion invariant.

**Method 3.3.2 (Guaranteeing Exclusion Invariants)** Let $\Sigma$ be a database system synchronized under locking protocol $\Lambda = \langle M, LC, R \rangle$, and let $LP$ and $LQ$ be local predicates of transactions $\tau_i$ and $\tau_j$, respectively, of $\Sigma$. The transactions of $\Sigma$ can be synchronized to ensure that

$$XI\colon\ \neg(LP \wedge LQ)$$

remains true by doing the following.

1. **Introduce Lock Modes.** Add new lock modes "$M0$" and "$M1$" to $M$.

2. **Strengthen Lock Compatibility.** Strengthen $LC$ so that

$$LC \Rightarrow \neg(ls_i \supseteq \{\ell_{[M0]}\} \wedge ls_j \supseteq \{\ell_{[M1]}\}).$$

3. **Strengthen Rules.** Add rules to $R$ that ensure that

$$LPI: \quad LP \Rightarrow ls_i \supseteq \{\ell_{[M0]}\}$$

and

$$LQI: \quad LQ \Rightarrow ls_j \supseteq \{\ell_{[M1]}\}$$

remain true at all times.

□

## 3.4 Using Locking to Strengthen Assertions Selectively

We now return to our original goal, which was to synchronize transactions so that an assertion $P$ in a proof outline $PO(\tau_i)$ can be strengthened selectively. Without loss of generality, assume that $P$ is to be replaced by a stronger assertion $P'$ such that $P' \Rightarrow (P \wedge B)$. For certain choices of $B$, the problem of replacing $P$ by $P'$ can be reduced to one of guaranteeing a set of exclusion invariants, as we now show.

**Lemma 3.4.1** Let

$$FPO(\Sigma):\ \{Q\}\mathbf{cobegin}\ PO(\tau_0) \parallel \cdots \parallel PO(\tau_{N-1})\ \mathbf{coend}\{R\}$$

be a full proof outline for a database system $\Sigma$ and let $P$ be an assertion in one of the $PO(\tau_i)$. Let $LP$ be a predicate local to $\tau_i$ such that

$$P \Rightarrow LP \tag{3.7}$$

and for each $0 \leq j \neq i < N$ let $LQ_j$ be a predicate local to $\tau_j$ such that

$$P \Rightarrow (B \vee \bigvee_{0\leq j\neq i<N} LQ_j). \tag{3.8}$$

Then

$$(P \wedge \bigwedge_{0\leq j\neq i<N} \neg(LP \wedge LQ_j)) \Rightarrow (P \wedge B).$$

□

**Proof of Lemma 3.4.1** From (3.7) and (3.8) it follows that

$$P \Rightarrow (LP \wedge (B \vee \bigvee_{0\leq j\neq i<N} LQ_j)). \tag{3.9}$$

Since conjunction distributes over disjunction, (3.9) implies

$$P \Rightarrow ((LP \wedge B) \vee \bigvee_{0\leq j\neq i<N} (LP \wedge LQ_j)). \tag{3.10}$$

Because

$$(\bigvee_{0\leq j\neq i<N} (LP \wedge LQ_j)) \Leftrightarrow \neg(\bigwedge_{0\leq j\neq i<N} \neg(LP \wedge LQ_j))$$

and because disjunction is commutative, (3.10) is equivalent to

$$P \Rightarrow (\neg(\bigwedge_{0\leq j\neq i<N} \neg(LP \wedge LQ_j)) \vee (LP \wedge B)). \tag{3.11}$$

By Predicate Logic, (3.11) is equivalent to

$$(P \wedge \bigwedge_{0 \leq j \neq i < N} \neg(LP \wedge LQ_j)) \Rightarrow (LP \wedge B),$$

from which it follows that

$$(P \wedge \bigwedge_{0 \leq j \neq i < N} \neg(LP \wedge LQ_j)) \Rightarrow B. \tag{3.12}$$

Because $P$ appears in the antecedent of (3.12) and $P \Rightarrow P$, (3.12) implies

$$(P \wedge \bigwedge_{0 \leq j \neq i < N} \neg(LP \wedge LQ_j)) \Rightarrow (P \wedge B).$$

□

Provided local predicates $LP$ and $LQ_j$ satisfying hypotheses (3.7) and (3.8) can be found, Method 3.3.2 can be used to strengthen $P$ with each of the exclusion invariants $\neg(LP \wedge LQ_j)$. The resulting assertion $P'$ will imply $P \wedge \bigwedge_{0 \leq j \neq i < N} \neg(LP \wedge LQ_j)$, and therefore by Lemma 3.4.1 will satisfy $P' \Rightarrow (P \wedge B)$. This gives the following method for using locking to strengthen assertions.

**Method 3.4.2 (Selectively Strengthening Assertions)** Let

$$FPO(\Sigma){:}\quad \{Q\}\,\mathbf{cobegin}\; PO(\tau_0) \parallel \cdots \parallel PO(\tau_{N-1})\; \mathbf{coend}\{R\}$$

be a full proof outline for database system $\Sigma$ and let $P$ be an assertion in one of the $PO(\tau_i)$. To strengthen $P$ to an assertion $P'$ such that $P' \Rightarrow (P \wedge B)$, do the following.

1. **Choose Local Predicates.** Choose a predicate $LP$ local to $\tau_i$ such that

$$P \Rightarrow LP$$

and for each $0 \leq j \neq i < N$ choose a predicate $LQ_j$ local to $\tau_j$ such that

$$P \Rightarrow (B \vee \bigvee_{0 \leq j \neq i < N} LQ_j).$$

2. **Guarantee Exclusion Invariants.** Use Method 3.3.2 to strengthen $\Lambda$ so that

$$\neg(LP \wedge LQ_j)$$

is invariant for each $0 \leq j \neq i < N$. Take $P' = (P \wedge \bigwedge_{0 \leq j \neq i < N} \neg(LP \wedge LQ_j)$.

□

Using Method 3.4.2 to help eliminate interference by selectively strengthening assertions, our method for proving the serializability of a database system (Method 2.5.3) can be transformed into a method for deriving synchronization that ensures it is serializable.

**Method 3.4.3 (Deriving Synchronization for Serializability)** To derive a locking protocol $\Lambda = \langle M, LC, R \rangle$ for database system $\Sigma = \langle V, C, T, \equiv \rangle$ such that $\langle \Lambda, \Sigma \rangle$ is serializable, do the following.

1. **Approximate $\langle \Lambda, \Sigma \rangle$.** Choose an initial version of $\Lambda$ and modify $\Sigma$ as necessary to follow $\Lambda$.

2. **Define Shadow Variables and Transactions.** Define shadow variables $\widehat{V}$ and shadow transactions $\widehat{T}$ corresponding to the variables $V$ and and transactions $T$ of $\Sigma$.

3. **Form Synchronized Augmented System.** Construct an augmented system $\Sigma^*$.

4. **Derive Synchronization.** Strengthen the locking protocol $\Lambda$ so that it becomes possible to construct the valid, full proof outline

$$\begin{array}{ll} FSD1(\Sigma^*)\text{:} & \{\, C \wedge V = \widehat{V}\} \\ & \textbf{cobegin}\ PO(\tau_0^*) \,||\, \cdots \,||\, PO(\tau_{N-1}^*)\ \textbf{coend} \\ & \{\, V \equiv \widehat{V}\} \end{array}$$

as follows.

   (a) **Construct Sequential Proof Outlines.** Construct valid proof outlines $PO(\tau_0^*)$,...,$PO(\tau_{N-1}^*)$ satisfying hypotheses H1 and H2.

   (b) **Eliminate Interference.** While hypothesis H4 remains unsatisfied, do the following

      i. Enumerate and check the interference freedom formulas.

      ii. Choose an invalid $NI(\alpha, A)$ for $\alpha$ in $PO(\tau_i)$ and $A$ in $PO(\tau_j)$ and do one of the following.

         - **Strengthen** $pre(\alpha)$**.** If possible, replace $pre(\alpha)$ by stronger assertion $pre(\alpha)'$ such that $\{pre(\alpha)' \wedge A\}\alpha\{A\}$ is valid, strengthening assertions that precede $pre(\alpha)'$ enough that $PO(\tau_i)$ remains valid as required by hypothesis H1 but not enough that hypothesis H2 is invalidated. If hypotheses H1 and H2 prohibit strengthening $pre(\alpha)$ enough to eliminate interference, use Method 3.4.2 to selectively strengthen $pre(\alpha)$ without invalidating these hypotheses.

         - **Weaken** $A$**.** If possible, replace $A$ by a weaker assertion $A'$ such that the interference freedom formula $\{pre(\alpha) \wedge A'\}\alpha\{A'\}$ is valid, weakening assertions that follow $A'$ enough that $PO(\tau_j)$ remains valid as required by hypothesis H1 but not enough that hypothesis H3 becomes unattainable. If hypotheses H1 and H3 prohibit weakening $A$ enough to eliminate interference, use Method 3.4.2 to selectively strengthen an assertion following $A$ so that $A$ can be weakened without invalidating these hypotheses.

(c) Check that the resulting proof outlines satisfy hypothesis H3.

5. **Infer $SD1(\Sigma^*)$.** Using the Assertion Deletion Rule, Infer

$$\begin{array}{ll} SD1(\Sigma^*): & \{C \wedge V = \widehat{V}\} \\ & \textbf{cobegin}\ \tau_0^* \mid\mid \cdots \mid\mid \tau_{N-1}^*\ \textbf{coend} \\ & \{V \equiv \widehat{V}\} \end{array}$$

from $FSD1(\Sigma^*)$.

6. **Prove Termination.** Prove that execution of $\Sigma^*$ terminates when started with $C \wedge V = \widehat{V}$ true.

□

## 3.5 An Example

To illustrate use of Method 3.4.3 to derive locking protocols for serializability, we consider a database system that supports a simple banking application. In deriving synchronization for serializability, we will illustrate another point as well. This example was first used in [L76] to argue that serializability is inappropriate as a correctness criteria because of the restrictions it imposes on concurrency among transactions. We will show that choosing an equivalence relation more accurately reflecting the semantics of the banking application makes it possible to derive a locking protocol that does ensure serializability while at the same time ensures a high degree of concurrency among transactions.

Database system $\Sigma5$ of Figure 3.1 models a simple banking application. Variables $V5$ include an array $a[0..N]$ that models account balances. For $j > 0$, array element $a[j]$ holds the balance of a customer account; $a[0]$ holds the balance of a dummy account that is equal to minus the sum of customer accounts, modeling the bank's financial obligation to its customers. This implies a consistency constraint that the elements of $a$ sum to zero.

Transaction $\tau_0$ models an auditor that inspects account balances to determine if funds have embezzled. The auditor accomplishes this by copying account balances into a ledger $l$ for inspection at a later time. $V5$ contains an array $l[0..N]$ modeling the ledger used by the auditor to record account balances. Transaction $\tau_1$ models a sequence of deposits, withdrawals and transfers. (A deposit to or withdrawal from an account $a[j]$ would be implemented by a transfer between that account and the bank's account $a[0]$.) Here, $\tau_1$ is shown performing only two such updates, to simplify analysis. To ensure that array references by $\tau_1$ are within $a$'s range of subscripts, $C5$ bounds variables $c0$, $d0$, $c1$ and $d1$.

States equivalent under $\equiv_5$ are those in which corresponding elements of $a[0..N]$ and variables $c0$, $d0$, $c1$ and $d1$ have the same value, and in which elements of $l[0..N]$ sum to the same value.[3] The latter property of $\equiv_5$ reflects that only the sum of ledger

[3]This value is 0 for consistent states.

$$
\begin{aligned}
&\Sigma 5 = \langle V5, C5, T5, \equiv_5 \rangle,\\
&\quad V5 = \langle a[0..N], l[0..N], k, c0, d0, t0, c1, d1, t1, cf_0, cf_1 \rangle,\\
&\quad C5 = (N \geq 0 \wedge 0 = \sum_{0 \leq j \leq N} a[j] \wedge 0 \leq c0 < d0 \leq N \wedge 0 \leq c1 < d1 \leq N),\\
&\quad T5 = \{\tau_0, \tau_1\},\\
&\qquad \tau_0 = A0\text{:}\ \ \langle k, l[0] := 0, a[0] \rangle;\\
&\qquad\qquad \mathbf{do}\ k \neq N \ \rightarrow\\
&\qquad\qquad\qquad A1\text{:}\ \ \langle k, l[k+1] := k+1, a[k+1] \rangle\\
&\qquad\qquad \mathbf{od};\\
&\qquad\qquad A2\text{:}\ \ \langle end(\tau_0) \rangle,\\
&\qquad \tau_1 = T0\text{:}\ \ \langle a[c0], a[d0] := a[c0] + t0, a[d0] - t0 \rangle;\\
&\qquad\qquad T1\text{:}\ \ \langle a[c1], a[d1] := a[c1] + t1, a[d1] - t1 \rangle;\\
&\qquad\qquad T2\text{:}\ \ \langle end(\tau_1) \rangle,\\
&\quad (V5' \equiv_5 V5'')\\
&\qquad \Leftrightarrow (a'[0..N] = a''[0..N] \wedge \sum_{0 \leq j \leq N} l'[j] = \sum_{0 \leq j \leq N} l''[j]\\
&\qquad\qquad \wedge\ c0' = c0'' \wedge d0' = d0'' \wedge c1' = c1'' \wedge d1' = d1'' \wedge cf_0' = cf_0'' \wedge cf_1' = cf_1'')
\end{aligned}
$$

Figure 3.1: Database System $\Sigma 5$ for an Idealized Banking Application.

entries is significant in the context of the banking application.

When $\tau_0$ and $\tau_1$ run concurrently, it is possible for $T0$ or $T1$ in $\tau_1$ to credit an account $c$ that has already been recorded in $l[0..N]$ and debit an account $d$ that has not yet been recorded. If this occurs, then $l[0..N]$ will not sum to zero when $\tau_0$ completes, and the auditor's ledger will incorrectly reflect that funds have been embezzled. To prevent this, we use Method 3.4.3 to derive a synchronized system $\langle \Lambda 6, \Sigma 6 \rangle$ that is serializable.

First, we choose a trivial locking protocol $\Lambda 6$ with no rules and add lock sets to the

$$\Lambda 6 = \langle M6, LC6, R6 \rangle,$$
$$M6 = \{\},$$
$$LC6 = \mathit{true},$$
$$R6 = \{\}$$
$$\Sigma 6 = \langle V6, C6, T6, \equiv_6 \rangle,$$
$$V6 = \langle a[0..N], l[0..N], k, c0, d0, t0, c1, d1, t1, cf_0, cf_1, ls_0, ls_1 \rangle,$$
$$C6 = C5$$
$$T6 = T5$$
$$(V6' \equiv_6 V6'')$$
$$\Leftrightarrow (a'[0..N] = a''[0..N] \wedge \sum_{0 \le j \le N} l'[j] = \sum_{0 \le j \le N} l''[j] \wedge cf_0' = cf_0'' \wedge cf_1' = cf_1''$$
$$\wedge\, c0' = c0'' \wedge d0' = d0'' \wedge c1' = c1'' \wedge d1' = d1'' \wedge ls_0' = ls_0'' \wedge ls_1' = ls_1'')$$

Figure 3.2: Synchronized Database System $\langle \Lambda 6, \Sigma 6 \rangle$.

variables of $\Sigma 5$ to give the synchronized database system $\langle \Lambda 6, \Sigma 6 \rangle$ of Figure 3.2.

Next, we define shadow variables

$$\widehat{V6} = \langle \hat{a}[0..N], \hat{l}[0..N], \hat{k}, \widehat{c0}, \widehat{d0}, \widehat{t0}, \widehat{c1}, \widehat{d1}, \widehat{t1}, \widehat{cf}_0, \widehat{cf}_1, \widehat{ls}_0, \widehat{ls}_1 \rangle$$

corresponding to those of $V6$, and shadow transactions

$$\begin{aligned}
&\hat{\tau}_0 = \langle \hat{k}, \hat{l}[0] := 0, \hat{a}[0] \rangle; \\
&\qquad \mathbf{do}\ \hat{k} \neq N \rightarrow \\
&\qquad\qquad \langle \hat{k}, \hat{l}[\hat{k}+1] := \hat{k}+1, \hat{a}[\hat{k}+1] \rangle \\
&\qquad \mathbf{od}; \\
&\qquad \langle \mathit{end}(\hat{\tau}_0) \rangle
\end{aligned}$$

and

$$
\begin{aligned}
&\Sigma 6^* = \langle\, V6^*, C6, T6^*, \equiv_6 \rangle,\\
&\quad V6^* = V6 \cdot \widehat{V6},\\
&\quad T6^* = \{\tau_0^*, \tau_1^*\},\\
&\qquad \tau_0^* = A0\text{:}\ \ \langle\, k, l[0] := 0, a[0] \,\rangle;\\
&\qquad\qquad \mathbf{do}\ k \neq N \ \rightarrow\\
&\qquad\qquad\qquad A1\text{:}\ \ \langle\, k, l[k+1] := k+1, a[k+1] \,\rangle\\
&\qquad\qquad \mathbf{od};\\
&\qquad\qquad A2\text{:}\ \ \langle\, end(\tau_0); \widehat{\tau}_0 \,\rangle\\
&\qquad \tau_1^* = T0\text{:}\ \ \langle\, a[c0], a[d0] := a[c0] + t0, a[d0] - t0 \,\rangle;\\
&\qquad\qquad T1\text{:}\ \ \langle\, a[c1], a[d1] := a[c1] + t1, a[d1] - t1 \,\rangle;\\
&\qquad\qquad T2\text{:}\ \ \langle\, end(\tau_1); \widehat{\tau}_1 \,\rangle
\end{aligned}
$$

Figure 3.3: Augmented Database System $\Sigma 6^*$.

$$
\begin{aligned}
\widehat{\tau}_1 = &\ \langle\, \widehat{a}[\widehat{c0}], \widehat{a}[\widehat{d0}] := \widehat{a}[\widehat{c0}] + \widehat{t0}, \widehat{a}[\widehat{d0}] - \widehat{t0} \,\rangle;\\
&\ \langle\, \widehat{a}[\widehat{c1}], \widehat{a}[\widehat{d1}] := \widehat{a}[\widehat{c1}] + \widehat{t1}, \widehat{a}[\widehat{d1}] - \widehat{t1} \,\rangle;\\
&\ \langle\, end(\widehat{\tau}_1) \,\rangle
\end{aligned}
$$

corresponding to $\tau_0$ and $\tau_1$. With these we form the augmented system $\langle \Lambda 6, \Sigma 6^* \rangle$ of Figure 3.3.

Next, we strengthen the locking protocol $\Lambda 6$ so that a valid full proof outline $FSD1(\Sigma 6^*)$ can be constructed from which $SD1(\Sigma 6^*)$ can be inferred. We present the derivation of $\Lambda 6$ as a succession of versions of $\langle \Lambda 6, \Sigma 6^* \rangle$ and $FSD1(\Sigma 6^*)$. Each version follows from the previous by a change to the database that makes progress towards satisfying the hypotheses of the **cobegin** Rule.

As a proof outline for the initial version $\langle \Lambda 6, \Sigma 6^* \rangle$ of Figure 3.3 we construct

$FSD1(\Sigma 6^*)$:

$\{ C6 \wedge V6 = \widehat{V6} \}$

**cobegin** $PO(\tau_0^*) \mid\mid PO(\tau_1^*)$ **coend**

$$\{ a[0..N] = \hat{a}[0..N] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \hat{l}[j] \wedge cf_0 = \widehat{cf}_0 \wedge cf_1 = \widehat{cf}_1$$
$$\wedge\, c0 = \widehat{c0} \wedge d0 = \widehat{d0} \wedge c1 = \widehat{c1} \wedge d1 = \widehat{d1} \wedge ls_0 = \widehat{ls}_0 \wedge ls_1 = \widehat{ls}_1 \}$$

where $PO(\tau_0^*)$ and $PO(\tau_1^*)$ are the proof outlines of Figures 3.4 and 3.5. Each assertion of $PO(\tau_0^*)$ and $PO(\tau_1^*)$ contains the invariant

$$I0: \quad \widehat{C6} \wedge c0 = \widehat{c0} \wedge d0 = \widehat{d0} \wedge t0 = \widehat{t0} \wedge c1 = \widehat{c1} \wedge d1 = \widehat{d1} \wedge t1 = \widehat{t1}$$
$$\wedge\, cf_0 = \widehat{cf}_0 \wedge cf_1 = \widehat{cf}_1.$$

It is easy to verify that $FSD1(\Sigma 6^*)$ satisfies hypotheses H1 and H2 of the **cobegin** Rule, so we omit the details here.

Next, we enumerate and check the interference freedom formulas. When this is done, we find that $NI(T0, post(A0))$, $NI(T0, pre(A1))$ and $NI(T0, post(A1))$ are invalid because $T0$ can make the conjunct

$$\left( \sum_{0 \le j \le k} l[j] + \sum_{k < j \le N} a[j] \right) = \sum_{0 \le j \le N} \hat{a}[j]$$

of the loop invariant $P0$ false by transferring funds between an account in $a[0..k]$ that has already been audited and an account in $a[k+1..N]$ that has not. For the same reason, $NI(T1, post(A0))$, $NI(T1, pre(A1))$ and $NI(T1, post(A1))$ are also invalid.

It is not possible to weaken $P0$ by deleting this conjunct because doing so would make it impossible to obtain a postcondition $post(PO(\tau_0^*))$ strong enough to satisfy hypothesis H3 of the **cobegin** Rule. Consequently, we strengthen $pre(T0)$ and $pre(T1)$.

Since

$$\{pre(T0) \wedge \neg(c0 \le k < d0) \wedge P0\}\ T0 \{P0\}$$

$PO(\tau_0^*)$:

$$\{I0 \wedge \sum_{0\le j\le N} a[j] = \sum_{0\le j\le N} \hat{a}[j]\}$$

$A0$: $\langle k, l[0] := 0, a[0] \rangle$;

$$\{I0 \wedge P0\colon\ (\sum_{0\le j\le k} l[j] + \sum_{k<j\le N} a[j]) = \sum_{0\le j\le N} \hat{a}[j]\}$$

**do** $k \ne N \rightarrow$

$\{I0 \wedge P0 \wedge k \ne N\}$

$A1$: $\langle k, l[k+1] := k+1, a[k+1] \rangle$

$\{I0 \wedge P0\}$

**od**;

$$\{I0 \wedge \sum_{0\le j\le N} l[j] = \sum_{0\le j\le N} \hat{a}[j]\}$$

$A2$: $\langle end(\tau_0); \hat{\tau}_0 \rangle$

$$\{I0 \wedge \sum_{0\le j\le N} l[j] = \sum_{0\le j\le N} \hat{l}[j]\}$$

Figure 3.4: Version 1 of $PO(\tau_0^*)$.

$PO(\tau_1^*)$: $\{I0 \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}$

$T0$: $\langle a[c0], a[d0] := a[c0] + t0, a[d0] - t0 \rangle$;

$$\{I0 \wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0] + t0 \wedge a[d0] = \hat{a}[d0] - t0\}$$

$T1$: $\langle a[c1], a[d1] := a[c1] + t1, a[d1] - t1 \rangle$;

$$\{I0 \wedge \bigwedge_{j\ne c0,d0,c1,d1} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0] + t0 \wedge a[d0] = \hat{a}[d0] - t0$$
$$\wedge\, a[c1] = \hat{a}[c1] + t1 \wedge a[d1] = \hat{a}[d1] - t1\}$$

$T2$: $\langle end(\tau_1); \hat{\tau}_1 \rangle$

$$\{I0 \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}$$

Figure 3.5: Version 1 of $PO(\tau_1^*)$.

and

$$\{pre(T1) \wedge \neg(c1 \leq k < d1) \wedge P0\}\ T1\ \{P0\}$$

are both valid, interference by $T0$ and $T1$ can be eliminated by replacing $pre(T0)$ and $pre(T1)$ by stronger assertions $pre(T0)'$ and $pre(T1)'$ such that

$$pre(T0)' \Rightarrow (pre(T0) \wedge \neg(c0 \leq k < d0))$$

and

$$pre(T1)' \Rightarrow (pre(T1) \wedge \neg(c1 \leq k < d1)).$$

However, strengthening $pre(T0)$ and $pre(T1)$ in this way would require replacing $pre(PO(\tau_1^*))$ by a stronger assertion that implies both $\neg(c0 \leq k < d0)$ and $\neg(c1 \leq k < d1)$. Since neither $C6$ nor $V6 \equiv_6 \widehat{V6}$ in $pre(FSD1(\Sigma 6^*))$ imply these predicates, strengthening $pre(T0)$ and $pre(T1)$ in this way would violate hypothesis H2. Consequently, we use Method 3.4.2 to strengthen $pre(T0)$ and $pre(T1)$ selectively.

To facilitate application of the method, we introduce a Boolean array $At[0..N]$ of auxiliary variables local to $\tau_0^*$ and another Boolean array $In[0..N]$ of auxiliary variables local to $\tau_1^*$. Elements of $At$ are used to indicate in the local state of $\tau_0^*$ the value of $k$ at points where assertions that are interfered with appear. This is accomplished by adding assignments to $\tau_0^*$ that ensure

$$A \Rightarrow At[k]$$

for every assertion $A$ in $PO(\tau_0^*)$ that contains $P0$.

In a similar manner, elements of $In$ are used to indicate in the local state of $\tau_1^*$ the indices $j$ in the range $c0 \leq j < d0$ at the point preceding $T0$, and those in the range $c1 \leq j < d1$ at the point preceding $T1$. This is accomplished by adding to $\tau_1^*$ assignments that ensure

$$pre(T0) \Rightarrow \bigwedge_{c0 \leq j < d0} In[j]$$

and

$$pre(T1) \Rightarrow \bigwedge_{c1 \leq j < d1} In[j].$$

This gives the second version of $\langle \Lambda 6, \Sigma 6^* \rangle$, where $\Lambda 6$ remains unchanged from Figure 3.2 and $\Sigma 6^*$ is shown in Figure 3.6. The proof outline for this version is

$FSD1(\Sigma 6^*)$:
$\{ C6 \wedge V6 = \widehat{V6} \}$
$\langle At[0], In[0], \ldots, At[N], In[N] := false, \ldots, false \rangle$;
$\{ C6 \wedge V6 = \widehat{V6} \wedge \bigwedge_{0 \leq j \leq N} \neg At[j] \wedge \neg In[j] \}$
**cobegin** $PO(\tau_0^*) \parallel PO(\tau_1^*)$ **coend**
$\{ a[0..N] = \hat{a}[0..N] \wedge \sum_{0 \leq j \leq N} l[j] = \sum_{0 \leq j \leq N} \hat{l}[j] \wedge cf_0 = \widehat{cf}_0 \wedge cf_1 = \widehat{cf}_1$
$\wedge\, c0 = \widehat{c0} \wedge d0 = \widehat{d0} \wedge c1 = \widehat{c1} \wedge d1 = \widehat{d1} \wedge ls_0 = \widehat{ls}_0 \wedge ls_1 = \widehat{ls}_1 \}$

where $PO(\tau_0^*)$ and $PO(\tau_1^*)$ are the proof outlines of Figures 3.7 and 3.8.

With the introduction of these auxiliary variables,

$$\{pre(T0) \wedge \bigwedge_{c0 \leq j < d0} \neg At[j] \wedge A\}\, T0 \{A\}$$

and

$$\{pre(T1) \wedge \bigwedge_{c1 \leq j < d1} \neg At[j] \wedge A\}\, T1 \{A\}$$

now are valid for every assertion $A$ containing $P0$. Thus, we can prevent $T0$ and $T1$ from interfering with assertions containing $P0$ by strengthening $pre(T0)$ and $pre(T1)$ to assertions $pre(T0)'$ and $pre(T1)'$ such that

$\Sigma 6^* = \langle\, V6^*, C6, T6^*, \equiv_6 \rangle,$

$\quad V6^* = V6 \cdot \widehat{V6},$

$\quad T6^* = \{\tau_0^*, \tau_1^*\},$

$\quad\quad \tau_0^* = \langle\, At[0] := \mathit{true} \,\rangle;$

$\quad\quad\quad A0\colon\ \langle\, k, l[0] := 0, a[0] \,\rangle;$

$\quad\quad\quad \mathbf{do}\ k \neq N\ \rightarrow$

$\quad\quad\quad\quad \langle\, At[k+1] := \mathit{true} \,\rangle;$

$\quad\quad\quad\quad A1\colon\ \langle\, k, l[k+1] := k+1, a[k+1] \,\rangle;$

$\quad\quad\quad\quad \langle\, At[k-1] := \mathit{false} \,\rangle$

$\quad\quad\quad \mathbf{od};$

$\quad\quad\quad \langle\, At[N] := \mathit{false} \,\rangle;$

$\quad\quad\quad A2\colon\ \langle\, \mathit{end}(\tau_0); \widehat{\tau}_0 \,\rangle$

$\quad\quad \tau_1^* = \langle\, In[c0], \ldots, In[d0-1] := \mathit{true}, \ldots, \mathit{true} \,\rangle;$

$\quad\quad\quad T0\colon\ \langle\, a[c0], a[d0] := a[c0] + t0, a[d0] - t0 \,\rangle;$

$\quad\quad\quad \langle\, In[c0], \ldots, In[d0-1] := \mathit{false}, \ldots, \mathit{false} \,\rangle;$

$\quad\quad\quad \langle\, In[c1], \ldots, In[d1-1] := \mathit{true}, \ldots, \mathit{true} \,\rangle;$

$\quad\quad\quad T1\colon\ \langle\, a[c1], a[d1] := a[c1] + t1, a[d1] - t1 \,\rangle;$

$\quad\quad\quad \langle\, In[c1], \ldots, In[d1-1] := \mathit{false}, \ldots, \mathit{false} \,\rangle;$

$\quad\quad\quad T2\colon\ \langle\, \mathit{end}(\tau_1); \widehat{\tau}_1 \,\rangle$

Figure 3.6: Version 2 of $\Sigma 6^*$.

$PO(\tau_0^*)$:

$\{I0 \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} a[j] = \sum_{0 \le j \le N} \hat{a}[j]\}$

$\langle\, At[0] := \mathbf{true}\,\rangle$;

$\{I0 \wedge At[0] \wedge \bigwedge_{0 < j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} a[j] = \sum_{0 \le j \le N} \hat{a}[j]\}$

$A0$: $\langle\, k, l[0] := 0, a[0]\,\rangle$;

$\{I0 \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0\!: \ (\sum_{0 \le j \le k} l[j] + \sum_{k < j \le N} a[j]) = \sum_{0 \le j \le N} \hat{a}[j]\}$

**do** $k \ne N \rightarrow$

$\{I0 \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0 \wedge k \ne N\}$

$\langle\, At[k+1] := \mathbf{true}\,\rangle$;

$\{I0 \wedge At[k] \wedge At[k+1] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0 \wedge k \ne N\}$

$A1$: $\langle\, k, l[k+1] := k+1, a[k+1]\,\rangle$;

$\{I0 \wedge At[k-1] \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0\}$

$\langle\, At[k-1] := \mathbf{false}\,\rangle$

$\{I0 \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0\}$

**od**;

$\{I0 \wedge At[N] \wedge \bigwedge_{0 \le j < N} \neg At[j] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \hat{a}[j]\}$

$\langle\, At[N] := \mathbf{false}\,\rangle$;

$\{I0 \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \hat{a}[j]\}$

$A2$: $\langle\, \mathbf{end}(\tau_0); \hat{\tau}_0\,\rangle$

$\{I0 \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \hat{l}[j]\}$

Figure 3.7: Version 2 of $PO(\tau_0^*)$.

$$
\begin{aligned}
PO(\tau_1^*)\text{:}\quad & \{I0 \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}\\
& \langle In[c0],\ldots,In[d0-1] := true,\ldots,true\,\rangle;\\
& \{I0 \wedge \bigwedge_{c0\le j<d0} In[j] \wedge \bigwedge_{\neg(c0\le j<d0)} \neg In[j] \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}\\
& T0\text{:}\quad \langle\, a[c0],a[d0] := a[c0]+t0, a[d0]-t0\,\rangle;\\
& \{I0 \wedge \bigwedge_{c0\le j<d0} In[j] \wedge \bigwedge_{\neg(c0\le j<d0)} \neg In[j] \wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j]\\
& \quad \wedge\, a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0\}\\
& \langle In[c0],\ldots,In[d0-1] := false,\ldots,false\,\rangle;\\
& \{I0 \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j]\\
& \quad \wedge\, a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0\}\\
& \langle In[c1],\ldots,In[d1-1] := true,\ldots,true\,\rangle;\\
& \{I0 \wedge \bigwedge_{c1\le j<d1} In[j] \wedge \bigwedge_{\neg(c1\le j<d1)} \neg In[j] \wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j]\\
& \quad \wedge\, a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0\}\\
& T1\text{:}\quad \langle\, a[c1],a[d1] := a[c1]+t1, a[d1]-t1\,\rangle;\\
& \{I0 \wedge \bigwedge_{c1\le j<d1} In[j] \wedge \bigwedge_{\neg(c1\le j<d1)} \neg In[j] \wedge \bigwedge_{j\ne c0,d0,c1,d1} a[j] = \hat{a}[j]\\
& \quad \wedge\, a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0 \wedge a[c1] = \hat{a}[c1]+t1\\
& \quad \wedge\, a[d1] = \hat{a}[d1]-t1\}\\
& \langle In[c1],\ldots,In[d1-1] := false,\ldots,false\,\rangle;\\
& \{I0 \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{j\ne c0,d0,c1,d1} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0]+t0\\
& \quad \wedge\, a[d0] = \hat{a}[d0]-t0 \wedge a[c1] = \hat{a}[c1]+t1 \wedge a[d1] = \hat{a}[d1]-t1\}\\
& T2\text{:}\quad \langle\, end(\tau_1); \hat{\tau}_1\,\rangle\\
& \{I0 \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}
\end{aligned}
$$

Figure 3.8: Version 2 of $PO(\tau_1^*)$.

$$pre(T0)' \Rightarrow (pre(T0) \wedge \neg At[j])$$

for every $c0 \leq j < d0$ and

$$pre(T1)' \Rightarrow (pre(T1) \wedge \neg At[j])$$

for every $c1 \leq j < d1$. We do this using Method 3.4.2.

To use Method 3.4.2 to strengthen $pre(T0)$ with a particular $\neg At[j]$, we must choose a predicate $LP$ local to $\tau_1^*$ such that

$$pre(T0) \Rightarrow LP.$$

Since

$$pre(T0) \Rightarrow In[j]$$

for each $c0 \leq j < d0$, we choose $LP = In[j]$.

As the next step of Method 3.4.2, we must choose a predicate $LQ$ local to $\tau_0^*$ such that

$$pre(T0) \Rightarrow (\neg At[j] \vee LQ).$$

Since

$$pre(T0) \Rightarrow (\neg At[j] \vee At[j])$$

tautologically, we choose $LQ = At[j]$.

As the last step of strengthening $pre(T0)$, we strengthen the locking protocol $\Lambda 6$ to guarantee that

$$\neg(In[j] \wedge At[j])$$

is invariant. Following Method 3.3.2 for establishing exclusion invariants, we define modes "$IN[j]$" and "$AT[j]$", strengthen $LC6$ so that

$$LC6 \Rightarrow \neg(ls_1 \supseteq \{\ell_{[IN[j]]}\} \wedge ls_0 \supseteq \{\ell_{[AT[j]]}\})$$

and add rules to $R6$ that ensure

$$LIN_j\text{:}\quad In[j] \Rightarrow ls_1 \supseteq \{\ell_{[IN[j]]}\}$$

and

$$LAT_j\text{:}\quad At[j] \Rightarrow ls_0 \supseteq \{\ell_{[AT[j]]}\}$$

remain true.

Method 3.4.2 is used to strengthen $pre(T1)$ with $\neg At[j]$ in a similar manner. Since

$$pre(T1) \Rightarrow In[j]$$

and

$$pre(T1) \Rightarrow (\neg At[j] \vee At[j])$$

for each $c1 \leq j < d1$, we choose $LP = In[j]$ and $LQ = At[j]$. Strengthening $\Lambda6$ to ensure that $\neg(In[j] \wedge At[j])$ is accomplished exactly as before.

Repeating for each appropriate value of $j$ the steps described above for strengthening $pre(T0)$ and $pre(T1)$ with $\neg At[j]$ results in the third version of $\Lambda6$, shown in Figure 3.9. Since $c0$, $d0$, $c1$ and $d1$ are not known in advance, we have made the modes, lock compatibility constraint and rules of $\Lambda6$ general enough for any possible values. The rules of $R6$ have been abbreviated by the invariants they require to remain true.

$$\begin{aligned}
&\Lambda 6 = \langle M6, LC6, R6 \rangle, \\
&\quad M6 = \{AT[0], IN[0], \ldots, AT[N], IN[N]\}, \\
&\quad LC6 = \bigwedge_{0 \le j \le N} \neg(ls_1 \supseteq \{\ell_{[IN[j]]}\} \wedge ls_0 \supseteq \{\ell_{[AT[j]]}\}), \\
&\quad R6 = \{LIN_0, LAT_0, \ldots, LIN_N, LAT_N\}.
\end{aligned}$$

Figure 3.9: Version 3 of $\Lambda 6$.

Having strengthened $\Lambda 6$, we must also modify $\Sigma 6^*$ to ensure that it continues to follow the locking protocol. The rules of $R6$ require each $LAT_j$ to remain true, and so $\tau_0^*$ must hold $\ell_{[IN[j]]}$ whenever $In[j]$ is true. Since each $At[j]$ is false before the **cobegin**, $\tau_0^*$ satisfies these rules initially. To ensure that $\tau_0^*$ continues to satisfy $R6$, we add an operation to acquire $\ell_{[AT[j]]}$ at the point where $At[j]$ becomes true. The rules of $R6$ also require each $LIN_j$ to remain true, which implies that $\tau_1^*$ must hold $\ell_{[AT[j]]}$ whenever $At[j]$ is true. Since each $In[j]$ is false before the **cobegin**, $\tau_1^*$ satisfies these rules when it starts. To ensure that $\tau_1^*$ continues to satisfy them, we add an operation to acquire $\ell_{[IN[j]]}$ at the point where $In[j]$ becomes true.

Addition of these acquire operations requires that the consistency constraint $C6$ also be strengthened to ensure that transactions complete when started in a consistent state. We accomplish this by strengthening the consistency constraint to

$$C6'\!: \quad C6 \wedge ls_0 = ls_1 = \{\}.$$

To ensure that they leave a consistent state when started in one, $\tau_0^*$ and $\tau_1^*$ must release every lock they acquire. To promote concurrency, we place release operations to that locks are released as early as possible.

Since $LAT_j$ is true whenever $At[j]$ is false, we add operations to $\tau_0^*$ to release each

$\ell_{[AT[j]]}$ at the point where $At[j]$ becomes false. Likewise, we add operations to $\tau_1^*$ to release each $\ell_{[IN[j]]}$ at the point where $In[j]$ becomes false. This gives the third version of $\Sigma 6^*$ in Figure 3.10 and completes the third version of $\langle \Lambda 6, \Sigma 6^* \rangle$. The proof outline for this version is

$$
\begin{array}{l}
FSD1(\Sigma 6^*): \\
\quad \{\, C6' \wedge V6 = \widehat{V6} \} \\
\quad \langle\, At[0], In[0], \ldots, At[N], In[N] := \mathit{false}, \ldots, \mathit{false} \,\rangle; \\
\quad \{\, C6' \wedge V6 = \widehat{V6} \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \neg In[j] \} \\
\quad \textbf{cobegin}\ PO(\tau_0^*) \mid\mid PO(\tau_1^*)\ \textbf{coend} \\
\quad \{\, a[0..N] = \hat{a}[0..N] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \hat{l}[j] \wedge cf_0 = \widehat{cf}_0 \wedge cf_1 = \widehat{cf}_1 \\
\quad\quad \wedge\, c0 = \widehat{c0} \wedge d0 = \widehat{d0} \wedge c1 = \widehat{c1} \wedge d1 = \widehat{d1} \wedge ls_0 = \widehat{ls}_0 \wedge ls_1 = \widehat{ls}_1 \}
\end{array}
$$

where $PO(\tau_0^*)$ and $PO(\tau_1^*)$ are the proof outlines of Figures 3.11 and Figure 3.12. Each assertion of $PO(\tau_0^*)$ and $PO(\tau_1^*)$ contains the stronger invariant

$$
\begin{array}{ll}
I1: & \widehat{C6'} \wedge c0 = \widehat{c0} \wedge d0 = \widehat{d0} \wedge t0 = \widehat{t0} \wedge c1 = \widehat{c1} \wedge d1 = \widehat{d1} \wedge t1 = \widehat{t1} \\
 & \wedge\, cf_0 = \widehat{cf}_0 \wedge cf_1 = \widehat{cf}_1 \wedge \widehat{ls}_0 = \widehat{ls}_1 = \{\} \\
 & \wedge \bigwedge_{0 \le j \le N} LIN_j \wedge \bigwedge_{0 \le j \le N} LAT_j .
\end{array}
$$

When the interference freedom formulas are enumerated and checked again, we find every formula $NI(\alpha, A)$ to be valid. Thus, $FSD1(\Sigma 6^*)$ satisfies hypothesis H4 of the **cobegin** Rule. We have been careful to preserve hypotheses H1 and H2, and it is easy to verify that

$$(\mathit{post}(PO(\tau_0^*)) \wedge \mathit{post}(PO(\tau_1^*))) \Rightarrow \mathit{post}(FSD1(\Sigma 6^*))$$

as required by hypothesis H3. Thus, $FSD1(\Sigma 6^*)$ is valid.

As the next step of Method 3.4.3, we must infer

$$
\begin{aligned}
&\Sigma 6^* = \langle\, V6^*, C6', T6^*, \equiv_6 \rangle,\\
&\quad V6^* = V6 \cdot \widehat{V6},\\
&\quad C6' = C6 \wedge ls_0 = ls_1 = \{\},\\
&\quad T6^* = \{\tau_0^*, \tau_1^*\},\\
&\qquad \tau_0^* = \langle\, \mathbf{acq}(\ell_{[AT[0]]}); At[0] := \mathit{true}\,\rangle;\\
&\qquad\qquad A0\colon\ \langle\, k, l[0] := 0, a[0]\,\rangle;\\
&\qquad\qquad \mathbf{do}\ k \neq N\ \rightarrow\\
&\qquad\qquad\qquad \langle\, \mathbf{acq}(\ell_{[AT[k+1]]}); At[k+1] := \mathit{true}\,\rangle;\\
&\qquad\qquad\qquad A1\colon\ \langle\, k, l[k+1] := k+1, a[k+1]\,\rangle;\\
&\qquad\qquad\qquad \langle\, At[k-1] := \mathit{false}; \mathbf{rel}(\ell_{[AT[k-1]]})\,\rangle\\
&\qquad\qquad \mathbf{od};\\
&\qquad\qquad \langle\, At[N] := \mathit{false}; \mathbf{rel}(\ell_{[AT[N]]})\,\rangle;\\
&\qquad\qquad A2\colon\ \langle\, \mathit{end}(\tau_0); \widehat{\tau}_0\,\rangle\\
&\qquad \tau_1^* = \langle\, \mathbf{acq}(\ell_{[IN[c0]]}, \ldots, \ell_{[IN[d0-1]]}); In[c0], \ldots, In[d0-1] := \mathit{true}, \ldots, \mathit{true}\,\rangle;\\
&\qquad\qquad T0\colon\ \langle\, a[c0], a[d0] := a[c0] + t0, a[d0] - t0\,\rangle;\\
&\qquad\qquad \langle\, In[c0], \ldots, In[d0-1] := \mathit{false}, \ldots, \mathit{false}; \mathbf{rel}(\ell_{[IN[c0]]}, \ldots, \ell_{[IN[d0-1]]})\,\rangle;\\
&\qquad\qquad \langle\, \mathbf{acq}(\ell_{[IN[c1]]}, \ldots, \ell_{[IN[d1-1]]}); In[c1], \ldots, In[d1-1] := \mathit{true}, \ldots, \mathit{true}\,\rangle;\\
&\qquad\qquad T1\colon\ \langle\, a[c1], a[d1] := a[c1] + t1, a[d1] - t1\,\rangle;\\
&\qquad\qquad \langle\, In[c1], \ldots, In[d1-1] := \mathit{false}, \ldots, \mathit{false}; \mathbf{rel}(\ell_{[IN[c1]]}, \ldots, \ell_{[IN[d1-1]]})\,\rangle;\\
&\qquad\qquad T2\colon\ \langle\, \mathit{end}(\tau_1); \widehat{\tau}_1\,\rangle
\end{aligned}
$$

Figure 3.10: Version 3 of $\Sigma 6^*$.

$$SD1(\Sigma6^*):\quad \{C6 \wedge V6 = \widehat{V6}\}\,\textbf{cobegin}\ \tau_0^* \parallel \tau_1^*\ \textbf{coend}\{\,V6 \equiv_6 \widehat{V6}\}$$

from $FSD1(\Sigma6^*)$. This is accomplished by first applying the Assertion Deletion Rule to obtain

$$\begin{array}{l} \{\,C6 \wedge V6 = \widehat{V6}\} \\ In[0], At[0], \ldots, In[N], At[N] := \mathit{false}, \ldots, \mathit{false}; \\ \textbf{cobegin}\ \tau_0^* \parallel \tau_1^*\ \textbf{coend} \\ \{\,V6 \equiv_6 \widehat{V6}\} \end{array}$$

and then applying the Auxiliary Variable Deletion Rule to delete assignments to elements of $In$ and $At$.

Finally, we must prove that execution of $\Sigma6^*$ terminates when started with $C6 \wedge V6 = \widehat{V6}$ true. To do this, we use Lemma 2.4.2, which states that under the assumption that concurrent execution of transactions is weakly fair, execution of $\Sigma6^*$ will terminate if the following two conditions are satisfied.

T1. Every execution of $\Sigma6^*$ consists of a bounded number of atomic operations.

T2. As long as execution of $\Sigma6^*$ has not terminated, there is at least one enabled atomic operation.

**Theorem 3.5.1** When started with $C6 \wedge V6 = \widehat{V6}$ true, execution of $\Sigma6^*$ satisfies conditions T1 and T2. □

**Proof of Theorem 3.5.1** Note that the number of iterations of the loop in $\tau_0^*$ is bounded by $N$. Since every other operation is executed as most once, execution of $\Sigma6^*$ satisfies condition T1.

Suppose that execution of $\Sigma6^*$ has not terminated. There are three possibilities:

$PO(\tau_0^*)$:

$\{I0 \wedge ls_0 = \{\} \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} a[j] = \sum_{0 \le j \le N} \widehat{a}[j]\}$

$\langle \mathbf{acq}(\ell_{[AT[0]]}); At[0] := \mathit{true} \rangle$;

$\{I0 \wedge ls_0 = \{\ell_{[AT[0]]}\} \wedge At[0] \wedge \bigwedge_{0 < j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} a[j] = \sum_{0 \le j \le N} \widehat{a}[j]\}$

$A0$: $\langle k, l[0] := 0, a[0] \rangle$;

$\{I0 \wedge ls_0 = \{\ell_{[AT[k]]}\} \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j]$

$\wedge\, P0$: $(\sum_{0 \le j \le k} l[j] + \sum_{k < j \le N} a[j]) = \sum_{0 \le j \le N} \widehat{a}[j]\}$

**do** $k \ne N \rightarrow$

$\{I0 \wedge ls_0 = \{\ell_{[AT[k]]}\} \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0 \wedge k \ne N\}$

$\langle \mathbf{acq}(\ell_{[AT[k+1]]}); At[k+1] := \mathit{true} \rangle$;

$\{I0 \wedge ls_0 = \{\ell_{[AT[k]]}, \ell_{[AT[k+1]]}\} \wedge At[k] \wedge At[k+1] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0$

$\wedge\, k \ne N\}$

$A1$: $\langle k, l[k+1] := k+1, a[k+1] \rangle$;

$\{I0 \wedge ls_0 = \{\ell_{[AT[k-1]]}, \ell_{[AT[k]]}\} \wedge At[k-1] \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0\}$

$\langle At[k-1] := \mathit{false}; \mathbf{rel}(\ell_{[AT[k-1]]}) \rangle$

$\{I0 \wedge ls_0 = \{\ell_{[AT[k]]}\} \wedge At[k] \wedge \bigwedge_{0 \le j \ne k \le N} \neg At[j] \wedge P0\}$

**od**;

$\{I0 \wedge ls_0 = \{\ell_{[AT[N]]}\} \wedge At[N] \wedge \bigwedge_{0 \le j < N} \neg At[j] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \widehat{a}[j]\}$

$\langle At[N] := \mathit{false}; \mathbf{rel}(\ell_{[At[N]]}) \rangle$;

$\{I0 \wedge ls_0 = \{\} \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \widehat{a}[j]\}$

$A2$: $\langle \mathit{end}(\tau_0); \widehat{\tau}_0 \rangle$

$\{I0 \wedge ls_0 = \{\} \wedge \bigwedge_{0 \le j \le N} \neg At[j] \wedge \sum_{0 \le j \le N} l[j] = \sum_{0 \le j \le N} \widehat{l}[j]\}$

Figure 3.11: Version 3 of $PO(\tau_0^*)$.

$PO(\tau_1^*)$: $\{I0 \wedge ls_1 = \{\} \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}$

$\langle \mathbf{acq}(\ell_{[IN[c0]]},\ldots,\ell_{[IN[d0-1]]}); In[c0],\ldots,In[d0-1] := true,\ldots,true\rangle;$

$\{I0 \wedge ls_1 = \{\ell_{[IN[c0]]},\ldots,\ell_{[IN[d0-1]]}\} \wedge \bigwedge_{c0\le j<d0} In[j] \wedge \bigwedge_{\neg(c0\le j<d0)} \neg In[j]$
$\wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}$

$T0$: $\langle a[c0], a[d0] := a[c0]+t0, a[d0]-t0\rangle;$

$\{I0 \wedge ls_1 = \{\ell_{[IN[c0]]},\ldots,\ell_{[IN[d0-1]]}\} \wedge \bigwedge_{c0\le j<d0} In[j] \wedge \bigwedge_{\neg(c0\le j<d0)} \neg In[j]$
$\wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0\}$

$\langle In[c0],\ldots,In[d0-1] := false,\ldots,false; \mathbf{rel}(\ell_{[IN[c0]]},\ldots,\ell_{[IN[d0-1]]})\rangle;$

$\{I0 \wedge ls_1 = \{\} \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0]+t0$
$\wedge a[d0] = \hat{a}[d0]-t0\}$

$\langle \mathbf{acq}(\ell_{[IN[c1]]},\ldots,\ell_{[IN[d1-1]]}); In[c1],\ldots,In[d1-1] := true,\ldots,true\rangle;$

$\{I0 \wedge ls_1 = \{\ell_{[IN[c1]]},\ldots,\ell_{[IN[d1-1]]}\} \wedge \bigwedge_{c1\le j<d1} In[j] \wedge \bigwedge_{\neg(c1\le j<d1)} \neg In[j]$
$\wedge \bigwedge_{j\ne c0,d0} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0\}$

$T1$: $\langle a[c1], a[d1] := a[c1]+t1, a[d1]-t1\rangle;$

$\{I0 \wedge ls_1 = \{\ell_{[IN[c1]]},\ldots,\ell_{[IN[d1-1]]}\} \wedge \bigwedge_{c1\le j<d1} In[j] \wedge \bigwedge_{\neg(c1\le j<d1)} \neg In[j]$
$\wedge \bigwedge_{j\ne c0,d0,c1,d1} a[j] = \hat{a}[j] \wedge a[c0] = \hat{a}[c0]+t0 \wedge a[d0] = \hat{a}[d0]-t0$
$\wedge a[c1] = \hat{a}[c1]+t1 \wedge a[d1] = \hat{a}[d1]-t1\}$

$\langle In[c1],\ldots,In[d1-1] := false,\ldots,false; \mathbf{rel}(\ell_{[IN[c1]]},\ldots,\ell_{[IN[d1-1]]})\rangle;$

$\{I0 \wedge ls_1 = \{\} \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}$

$T2$: $\langle end(\tau_1); \hat{\tau}_1\rangle$

$\{I0 \wedge ls_1 = \{\} \wedge \bigwedge_{0\le j\le N} \neg In[j] \wedge \bigwedge_{0\le j\le N} a[j] = \hat{a}[j]\}$

Figure 3.12: Version 3 of $PO(\tau_1^*)$.

- One of $\tau_0^*$ or $\tau_1^*$ has reached an operation $S_i$ that is not an acquire operation.
- One of $\tau_0^*$ or $\tau_1^*$ has reached an acquire operation $S_i$ and the other has terminated.
- Both $\tau_0^*$ and $\tau_1^*$ have reached acquire operations $S_i$ and $S_j$.

Assume the first case. Since $FSD1(\Sigma6^*)$ is valid, $pre(S_i)$ will be true when $S_i$ is reached. For every $S_i$ in $FSD1(\Sigma6^*)$ that is not an acquire operation,

$$pre(S_i) \Rightarrow wp(S_i, true).$$

Thus, $S_i$ is enabled.

Assume the second case. Since both transactions release every lock they acquire, the lock set of the terminated transaction will be empty. This implies that the acquire operation $S_i$ is enabled.

Assume the last case. Without loss of generality, assume that $S_i$ is an operation of $\tau_0^*$ and $S_j$ is an operation of $\tau_1^*$. Note that the lock set of $\tau_1^*$ is empty when an acquire operation in $\tau_1^*$ has been reached. This implies that $S_i$ is enabled.

In each case, at least one operation is enabled. Thus, $\Sigma6^*$ satisfies condition T2. □

Thus, database system $\langle \Lambda6, \Sigma6 \rangle$ of Figure 3.13, obtained by deleting auxiliary and shadow variables from $\langle \Lambda6, \Sigma6^* \rangle$, is serializable.

# 3.6 Discussion

## 3.6.1 Comparing Locking Protocols

Database system locking protocols are usually specified operationally. Lock modes typically correspond to the types of operations from which transactions are constructed

$$
\begin{aligned}
&\Lambda 6 = \langle M6, LC6, R6 \rangle, \\
&\quad M6 = \{AT[0], IN[0], \ldots, AT[N], IN[N]\}, \\
&\quad LC6 = \bigwedge_{0 \le j \le N} \neg(ls_1 \supseteq \{\ell_{[IN[j]]}\} \wedge ls_0 \supseteq \{\ell_{[AT[j]]}\}), \\
&\quad R6 = \{LIN_0, LAT_0, \ldots, LIN_N, LAT_N\}. \\
&\Sigma 6 = \langle V6, C6', T6, \equiv_6 \rangle, \\
&\quad C6' = C6 \wedge ls_0 = ls_1 = \{\}, \\
&\quad T6 = \{\tau_0, \tau_1\}, \\
&\qquad \tau_0 = \langle \mathbf{acq}(\ell_{[AT[0]]}) \rangle; \\
&\qquad\qquad A0\colon\ \langle k, l[0] := 0, a[0] \rangle; \\
&\qquad\qquad \mathbf{do}\ k \neq N \rightarrow \\
&\qquad\qquad\qquad \langle \mathbf{acq}(\ell_{[AT[k+1]]}) \rangle; \\
&\qquad\qquad\qquad A1\colon\ \langle k, l[k+1] := k+1, a[k+1] \rangle; \\
&\qquad\qquad\qquad \langle \mathbf{rel}(\ell_{[AT[k-1]]}) \rangle \\
&\qquad\qquad \mathbf{od}; \\
&\qquad\qquad \langle \mathbf{rel}(\ell_{[AT[N]]}) \rangle; \\
&\qquad\qquad A2\colon\ \langle end(\tau_0) \rangle \\
&\qquad \tau_1 = \langle \mathbf{acq}(\ell_{[IN[c0]]}, \ldots, \ell_{[IN[d0-1]]}) \rangle; \\
&\qquad\qquad T0\colon\ \langle a[c0], a[d0] := a[c0] + t0, a[d0] - t0 \rangle; \\
&\qquad\qquad \langle \mathbf{rel}(\ell_{[IN[c0]]}, \ldots, \ell_{[IN[d0-1]]}) \rangle; \\
&\qquad\qquad \langle \mathbf{acq}(\ell_{[IN[c1]]}, \ldots, \ell_{[IN[d1-1]]}) \rangle; \\
&\qquad\qquad T1\colon\ \langle a[c1], a[d1] := a[c1] + t1, a[d1] - t1 \rangle; \\
&\qquad\qquad \langle \mathbf{rel}(\ell_{[IN[c1]]}, \ldots, \ell_{[IN[d1-1]]}) \rangle; \\
&\qquad\qquad T2\colon\ \langle end(\tau_1) \rangle
\end{aligned}
$$

Figure 3.13: Serializable Database System $\langle \Lambda 6, \Sigma 6 \rangle$.

and the compatibility relation typically specifies that modes associated with operations of a given type are exclusive when the outcome of transaction execution is influenced by the order in which operations of this type interleave. Rules for acquiring and releasing locks almost always require a transaction to hold a lock when executing an associated operation.

In contrast, the locking protocols derived using the method of this chapter are specified assertionally. Lock modes correspond to predicates about the system state, and lock compatibility relations forbid different transactions from simultaneously holding locks when the associated predicates should not be simultaneously true. Rules for acquiring and releasing locks enforce a coupling between the state of a transaction and the set of locks it holds by requiring a transaction to hold a lock with a given mode whenever the associated predicate is true.

### 3.6.2 Locks and Local State

Locking protocols derived using Method 3.3.2 associate the locks held by a transaction $\tau_i$ with its local state through rules that require invariants of the form

$$LPI:\quad LP \Rightarrow ls_i \supseteq \{\ell_{[M]}\}$$

to remain true. Since no other transaction can modify the local state of $\tau_i$ or change the contents of its lock set, the requirement that $LP$ is local to $\tau_i$ ensures that $LPI$ is not interfered with. This property simplifies the task of synchronizing $\tau_i$ so that $LPI$ remains true. In addition, this property makes Method 3.3.2 appropriate for synchronizing transactions to eliminate interference, since it avoids introducing additional interference in the process.

Section 3.5 demonstrated that the search for appropriate local predicates required when using Method 3.4.2 to strengthening assertions can lead to introduction of local auxiliary variables to capture relevant properties of the local state of transactions. While this may seem somewhat cumbersome, it does tend to make explicit the points at which locks should be acquired and released.

# Chapter 4

# Concluding Remarks

## 4.1 Summary and Discussion

This dissertation has addressed two fundamental problems that arise in the context of database systems: the characterization of serializability and the construction of locking protocols to synchronize concurrently executing transactions. In contrast to the use of operational reasoning that has dominated previous research on these problems, we have used assertional reasoning to analyze the semantics of concurrent execution of transactions. As a result of this effort, we have been able to apply to database systems the tools and techniques that have been developed for reasoning assertionally about more general types of concurrent programs. This has lead to insight into semantics of serializability, provided new methods for specifying and proving the serializability of database systems, and suggested new ways of constructing locking protocols for database synchronization.

In Chapter 2, we presented a formal definition of serializability. A unique feature of this definition is an equivalence relation with which final states reached by schedules are compared. The equivalence relation, which can be derived from the application supported by the database, makes explicit the way in which the effects of different schedules are reflected in the system state. It does this by partitioning the set of systems states into equivalence classes, each containing states that are indistinguishable by the supported application. The inclusion of this relation as a parameter of the definition can be viewed as a generalization of previous definitions, which make implicit assumptions about aspects of the system state that are relevant.

Our initial characterization of serializability shared with previous ones the property that the serializability of a database system is defined in terms of the serializability of each of its possible schedules. Because of the potentially enormous number of different schedules possible in a typical database system, it also shares with previous definitions the property of limited utility as a practical basis for verifying the serializability of database systems. For this reason, we turned to proof outlines to obtain a more useful characterization of serializability.

Proof outlines provide a way to reason formally about a concurrent program without considering every possible interleaving of its operations. We presented two characterizations of serializability in terms of proof outlines. The first was equivalent to our original definition; the second was strictly weaker, specifying a property that only implied serializability under the original definition. Translation of serializability into proof outlines was made possible using shadow variables and transactions to model serial schedule behavior in the system state. Our two characterizations of serializability with proof outlines differ concerning how these shadow variables and transactions were used to accomplish this.

Our first characterization of serializability with proof outlines used shadow transactions within assertions to specify the set of states reachable by serial executions of a database system. This necessitated a proof outline with a postcondition of size proportional to the number of different serial schedules for that system. This number, though smaller than the number of all types of schedules, can be large enough in many situations that the proof outlines used to specify and prove serializability grow unwieldy.

Our second characterization of serializability avoided this problem by moving the shadow transactions from assertions into transactions themselves, where they run serially along side other transaction operations. This makes it possible to characterize serializability with simpler assertions, because serial behavior is captured implicitly in the state of the shadow variables as the shadow transactions run.

Our use of proof outlines to characterize serializability provides not only a way to characterize and reason assertionally about serializability, but also provides a framework in which synchronization to ensure serializability can be derived from the proof outlines that specify it. We explored this possibility in Chapter 3, where we described a method for deriving locking protocols for database systems. Our method is built upon an assertional characterization of locking: locks are associated with predicates on the system state and lock compatibility is induced by restrictions on configurations of states during concurrent execution of transactions. This is different from the traditional view of locking, in which locks are associated with operation types and lock compatibility is motivated by restrictions on the types of operations that can run concurrently.

Using our method, locking protocols are derived using full proof outlines for transactions. Since full proof outlines contain assertions before and after each operation, information about the context in which operations run is available while deriving synchronization. This information can be used to identify those interleavings of operations that do not violate serializability, and incorporated into the derived locking protocol to increase concurrency among transactions that follow it. Locking protocols derived operationally are not able to capitalize on such information.

## 4.2 Topics for Further Research

In our database system model, we have assumed that database systems execute a fixed, finite number of transactions concurrently. Such a model is appropriate for special purpose databases that support applications in which the set of transactions necessary can be determined in advance. It is not as appropriate for systems in which new transactions are introduced and executed as time passes. Further research is needed to determine the extent to which the results of this dissertation can be applied to these types of database systems.

Serializability is an instance of a type of virtual atomicity that appears in areas of concurrent programming other than database systems. An example of one such situation is described in [HW86], where concurrent processes access instances of abstract datatypes by invoking abstract operations. To simplify the design of processes in such systems, processes are constructed under the assumption that individual abstract operation invocations run atomically. When abstract operations run concurrently, however, the operations from which they are composed can interleave to violate these assumptions. To prevent this, abstract operations are synchronized to guarantee a property called *linearizability* that ensures every concurrent execution is equivalent to one in which abstract operations run indivisibly. Analysis of linearizability and other situations requiring virtual atomicity is warranted to see if the assertional tools developed in this dissertation are useful.

# Appendix A

# Axioms and Inference Rules of Proof Outline Logic

**skip Axiom.**

$$\{R\}\,\mathbf{skip}\,\{R\}$$

**Assigment Axiom.**

Let $\overline{x} = x_0, \ldots, x_N$ be a vector of simple variables (i.e. not elements of records or arrays) and let $\overline{e} = e_0, \ldots, e_N$ be a vector equal in length to $\overline{x}$ of expresions in which the types of each $e_i$ and $x_i$ are the same.

$$\{R^{\overline{x}}_{\overline{e}}\}\,\overline{x} := \overline{e}\,\{R\}$$

**Acquire Axiom.**

For $\mathbf{acq}(\ell_{[m_0]},\ldots,\ell_{[m_k]})$ an acquire operation in $\tau_i$,

$$\begin{array}{l}\{(LC \Rightarrow R)^{ls_i}_{ls_i \cup \{\ell_{[m_0]},\ldots,\ell_{[m_k]}\}}\}\\ \mathbf{acq}(\ell_{[m_0]},\ldots,\ell_{[m_k]})\\ \{R\}\end{array}$$

**Release Axiom.**

For $\mathbf{rel}(\ell_{[m_0]},\ldots,\ell_{[m_k]})$ a release operation in $\tau_i$,

$$\begin{array}{l}\{(LC \Rightarrow R)^{ls_i}_{ls_i - \{\ell_{[m_0]},\ldots,\ell_{[m_k]}\}}\}\\ \mathbf{rel}(\ell_{[m_0]},\ldots,\ell_{[m_k]})\\ \{R\}\end{array}$$

**Statement Composition Rule.**

$$\frac{\{P\}\,S0\,\{Q\},\ \{Q\}\,S1\,\{R\}}{\{P\}\,S0;\{Q\}\,S1\,\{R\}}$$

**if Rule.**

$$\frac{\{B0 \wedge Q\}\,S0\,\{R\},\ldots,\{Bn \wedge Q\}\,Sn\,\{R\}}{\begin{array}{l}\{Q\}\\ \mathbf{if}\ B0 \rightarrow \{B0 \wedge Q\}\,S0\,\{R\}\\ \vdots\\ \,[\,]\ Bn \rightarrow \{Bn \wedge Q\}\,Sn\,\{R\}\\ \mathbf{fi}\\ \{R\}\end{array}}$$

**do Rule.**

$$\frac{\{B0 \wedge I\}\,S0\{I\}, \ldots, \{Bn \wedge I\}\,Sn\{I\}}{\begin{array}{l}\{I\} \\ \textbf{do}\; B0 \rightarrow \{B0 \wedge I\}\,S0\{I\} \\ \vdots \\ \,[\,]\, Bn \rightarrow \{Bn \wedge I\}\,Sn\{I\} \\ \textbf{od} \\ \{I \wedge \neg B0 \wedge \ldots \wedge \neg Bn\}\end{array}}$$

**Rule of Consequence.**

$$\frac{\{Q\}\,S\,\{R\},\; Q' \Rightarrow Q,\; R \Rightarrow R'}{\{Q'\}\,S\,\{R'\}}$$

**Assertion Deletion Rule.**

Let $S''$ be the result of deleting one or more assertions from annotated program $S'$.

$$\frac{\{Q\}\,S'\{R\}}{\{Q\}\,S''\{R\}}$$

**Atomicity Rule.**

$$\frac{\{Q\}\,S'\{R\}}{\{Q\}\langle S'\rangle\{R\}}$$

**Auxiliary Variable Deletion Rule.**

Let $AV$ be a set of auxiliary variables in annotated program $S$ and let $S\,|_{\overline{AV}}$ be the annotated program obtained by deleting from $S$ all assignments to variables of $AV$. If $Q$, $R$ and the assertions of $S$ do not mention any variable in $AV$, then

$$\frac{\{Q\}\,S\,\{R\}}{\{Q\}\,S\,|_{\overline{AV}}\{R\}}$$

**cobegin Rule.**

Let $PO_0,\ldots,PO_N$ be full proof outlines (ones in which at least one assertion preceeds and follows each atomic operation.) Define $\alpha \parallel A$ if and only if $\alpha$ is an atomic operation in one proof outline $PO_i$ and $A$ is an assertion in another proof outline $PO_j$.

$$\frac{\begin{array}{ll} H0\colon & PO_0,\ldots,PO_{N-1}, \\ H1\colon & Q \Rightarrow (pre(PO_0) \wedge \ldots \wedge pre(PO_{N-1})), \\ H2\colon & (post(PO_0) \wedge \ldots \wedge post(PO_{N-1})) \Rightarrow R, \\ H3\colon & (\forall \alpha, A\colon\ \alpha \parallel A\colon\ NI(\alpha, A)\colon\ \ \{pre(\alpha) \wedge A\}\alpha\{A\}) \end{array}}{\{Q\}\,\mathbf{cobegin}\ PO_0 \parallel PO_1 \parallel \cdots \parallel PO_{N-1}\ \mathbf{coend}\{R\}}$$

# Appendix B

# The Weakest Precondition Predicate Transformer

The *weakest precondition of $S$ with respect to $R$*, denoted $wp(S,R)$, represents the set of all states such that execution of $S$ begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying $R$ [G81]. $wp$ satisfies the following properties.

**Law of the Excluded Miracle.**

$$wp(S,false) = false.$$

**Distributivity of Conjunction.**

$$(wp(S,Q) \wedge wp(S,R)) \Leftrightarrow wp(S,Q \wedge R).$$

**Law of Monotonicity.**

If $Q \Rightarrow R$ then $wp(S,Q) \Rightarrow wp(S,R)$.

**Distributivity of Disjuntion.**

$$(wp(S,Q) \vee wp(S,R)) \Rightarrow wp(S,Q \vee R).$$

**Distributivity of Disjuntion for Deterministic $S$.**

$$(wp(S,Q) \vee wp(S,R)) \Leftrightarrow wp(S,Q \vee R).$$

**skip Axiom.**

$$wp(\mathbf{skip},R) = R.$$

**Assignment Axiom.**

Let $\overline{x} = x_0,\ldots,x_N$ be a vector of simple variables (i.e. not elements of records or arrays) and let $\overline{e} = e_0,\ldots,e_N$ be a vector equal in length to $\overline{x}$ of expresions in which the types of each $e_i$ and $x_i$ are the same. Define $DOM(e_0,\ldots,e_N)$ to be the predicate that describes the set of all states in which each $e_i$ is well-defined.

$$wp(x_0,\ldots,x_N := e_0,\ldots,e_N, R) = DOM(e_0,\ldots,e_N) \wedge R^{x_0,\ldots,x_N}_{e_0,\ldots,e_N}.$$

**if Rule.**

Let $IF$ denote

$$
\begin{aligned}
&\textbf{if } B_0 \rightarrow S_0 \\
&[\,]\, B_1 \rightarrow S_1 \\
&\vdots \\
&[\,]\, B_N \rightarrow S_N \\
&\textbf{fi}
\end{aligned}
$$

and let $BB$ denote

$$B_0 \vee B_1 \vee \cdots \vee B_n.$$

$$wp(IF, R) = (DOM(BB) \wedge BB \wedge \bigwedge_{0 \leq j \leq N} B_i \Rightarrow wp(S_i, R)$$

**do Rule.**

Let $DO$ denote

$$
\begin{aligned}
&\textbf{do } B_0 \rightarrow S_0 \\
&[\,]\, B_1 \rightarrow S_1 \\
&\vdots \\
&[\,]\, B_N \rightarrow S_N \\
&\textbf{od}
\end{aligned}
$$

and let $BB$ denote

$$B_0 \vee B_1 \vee \cdots \vee B_n.$$

Define

$$H_0(R) = \neg BB \wedge R.$$

and

$$H_k(R) = H_0(R) \vee wp(IF, H_{k-1}(R))$$

for $k > 0$.

$$wp(DO, R) = (\exists k\colon\ 0 \le k\colon\ H_k(R)).$$

**Composition Rule.**

$$wp(S1; S2, R) = wp(S1, wp(S2, R)).$$

# Bibliography


[BBGLS83] C. Beeri, P. Bernstein, N. Goodman, M. Lai, and D. Shasha. A concurrency control theory for nested transactions. In *Second Annual ACM Symposium on Principles of Distributed Computing*, pages 45–62, ACM, 1983.

[BG81] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, Jun. 1981.

[BG83] P. Bernstein and N. Goodman. Concurrency control for multiversion database systems. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[BSW79] P. Bernstein, D. Shipman, and W. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3):203–216, May 1979.

[C73] M. Clint. Program proving: coroutines. *Acta Informatica*, 2:50–63, 1973.

[C81] M. Casanova. *The Concurrency Control Problem for Database Systems.* *Lecture Notes in Computer Science*, Springer-Verlag, New York, New York, 1981.

[D76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Inglewood, New Jersey, 1976.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[FL79] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[G78] J. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: an Advanced Course*, Springer-Verlag, 1978.

[G81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, New York, 1981.

[G83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, Jun. 1983.

[GS85] N. Goodman and D. Shasha. Semantically-based concurrency control for search structures. In *Principles of Distributed Systems*, pages 8–19, ACM, 1985.

[GW82] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, Jun. 1982.

[H79] D. Harel. *First Order Dynamic Logic*. Volume 68 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, New York, 1979.

[HW86] M. P. Herlihy and J. M. Wing. *Axioms for Concurrent Objects*. Technical Report CMU-CS-86-154, Carnegie-Mellon University, Oct. 1986.

[K83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, Jan. 1983.

[KS79] Z. Kedem and A. Silberschatz. Controlling concurrency using locking. In *IEEE Foundations of Computer Science*, pages 274–285, IEEE, 1979.

[L76] L Lamport. *Towards a Theory of Correctness for Multi-user Data Base Systems*. Technical Report CA-7610-0712, Massachusetts Computer Associates, Oct. 1976.

[L80] L Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[P79] C. H. Papadimitriou. Serializability of concurrent updates. *Journal of the ACM*, 26(4):631–653, Oct. 79.

[P86] C. H. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[P87] D. Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34(2):450–479, Apr. 87.

[R83] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, Feb. 1983.

[S67] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[SA87] F. B. Schneider and G. R. Andrews. Concurrent programming: centralized and distributed. In *Current Trends in Concurrency*, Springer-Verlag, 1987.

[SLR76] R. E. Stearns, P. M. Lewis, and D. J. Rosenkrantz. Concurrency control for database systems. In *17th Symposium on Foundations of Computer Science*, pages 19–32, 1976.

[SS84] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, Aug. 1984.

[TS85] A. Tuzhilin and P. Spirakis. A semantic approach to correctness of concurrent transaction executions. In *Proceedings of Principles of Distributed Computing*, pages 85–95, 1985.

[W84] W. E. Weihl. *Specification and Implementation of Atomic Data Types.* Ph.D. dissertation, Massachusetts Institute of Technology, 1984.

[Y84] M. Yannakakis. Serializability by locking. *Journal of the ACM*, 31(2):227–244, Apr. 1984.