An Assertional Characterization of Serializability*

E. Robert McCurley
Fred B. Schneider

TR 89-1039
September 1989

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

*This material is based on work supported by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under grant no. CCR-8701103, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

An Assertional Characterization of Serializability*

E. Robert McCurley
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853

September 28, 1989

Abstract

Serializability is usually defined operationally in terms of sequences of operations. This paper gives another definition of serializability—in terms of sequences of states. It also shows how this definition can be used to prove correctness of solutions to the concurrency control problem.

1 Introduction

A *database system* is a computer system that stores information. *Consistency constraints* restrict system states to those that are meaningful; *transactions* are designed so that each individually transforms the database from

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

one consistent state to another. For example, in a database for a banking application, a consistency constraint might relate cash-on-hand to the sum of the account balances; a transaction for a deposit would adjust both an account balance and cash-on-hand, preserving the consistency constraint.

During concurrent execution of transactions, operations can interleave in ways that leave the database in an inconsistent state. Avoiding such states is called the *concurrency control problem.* The traditional solution to this problem is based on *serializability* [EGLT76], which asserts that transactions executing concurrently "behave like" they are run serially, one after another. If, when run in isolation, each transaction transforms the database from one consistent state to another and if transactions are serializable, then concurrent execution of those transactions will also transform the database from one consistent state to another. Thus, implementing serializability solves the concurrency control problem.

Serializability is usually defined (operationally) in terms of sequences, called *schedules*, that list operations in the order they run. This would seem to preclude use of programming methodologies based on assertions about states—so called *assertional reasoning*. It is tempting to regard this as a fundamental limitation of such methodologies. In this paper, we show this view to be erroneous. We give an assertional characterization of serializability and show how this definition can be used to prove correctness of solutions to the concurrency control problem.

We proceed as follows. In Section 2, we present a database system model and give a formal definition of serializability. Two ways to specify serializability using formulas of a Hoare-style programming logic are given in Section 3. In Section 4, we discuss possible extensions to our database system model and their implications. We conclude, in Section 5, with a comparison of our definition and previous ones.

2 Serializability

2.1 System Model

A database system Σ can be represented by a triple $\langle V, C, T\rangle$, where V is a set of variables, C is a predicate on V, and

$$T: \quad [\tau_0 \parallel \dots \parallel \tau_{N-1}]$$

is a concurrent program in which each *transaction* τ_i is a program that references only variables of V.

Variables in V represent the state of the database, including but not limited to the part that actually contains application data.[1] C is the consistency constraint of the database. A state is *consistent* iff C is true in that state. Each transaction τ_i is assumed to terminate in a consistent state when run alone starting in one.

An example of a database system is given in Figure 1. Σ_0 models a database that maintains an unordered collection of records, represented by variable s of type set. Variables $in_0, \ldots, in_{N-1}$ are sequences, containing records that have or will be inserted into s, and consistency constraint C_0 requires s to contain a subset of these. Each transaction Add_i adds the records of in_i to s. Variable t_i of Add_i is local to the transaction, hence excluded from V_0. In the guard of the loop, $|in_i|$ denotes the length of the sequence in_i. Angle brackets "$\langle$" and "$\rangle$" surround operations that execute atomically.

$$
\begin{array}{rll}
\Sigma_0 = \langle & V_0: & s, in_0, \ldots, in_{N-1}, \\
& C_0: & s \subseteq (in_0 \cup \cdots \cup in_{N-1}), \\
& T_0: & [Add_0 \mid\mid \cdots \mid\mid Add_{N-1}] \ \rangle
\end{array}
$$

$$
\begin{array}{ll}
Add_i: & \langle\, t_i := 0 \,\rangle; \\
& \textbf{do } t_i \neq |in_i| \rightarrow \\
& \quad \langle\, s := s \cup in_i(t_i) \,\rangle; \\
& \quad \langle\, t_i := t_i + 1 \,\rangle \\
& \textbf{od}
\end{array}
$$

Figure 1: Database System Σ_0.

2.2 Serializability

Serializability of $\Sigma = \langle V, C, T \rangle$ can be understood in terms of an abstract database system $\Sigma' = \langle V', C', T' \rangle$ where V' and C' are the same as V and C of Σ, and

$$T': \quad [\tau'_0 \mid\mid \ldots \mid\mid \tau'_{N-1}] \ , \tag{1}$$

is the concurrent program in which each τ'_i is $\langle\, \tau_i \,\rangle$, a transaction that executes the same operations as τ_i but as a single atomic operation. Executions

[1] A commit flag is an example of an element of V that does not contain application data.

of T' are *serial*, meaning that transactions are executed one after another without interleaving.

Recall that serializability asserts that every (potentially interleaved) execution of T "behaves like" some (serial) execution of T'. Since if one system T implements another one T', then every behavior of T can be viewed as a behavior of T', serializability of T is equivalent to T implementing T' [Lam88].

2.3 Defining "Implements"

In [Pri87, GP85], a formal definition in terms of predicate transformers is given for when one program implements another. This definition is a generalization of that presented in [Hoa72] and can be summarized as follows. Let S' be a program operating on variables X, and let P be a predicate on X such that S' is guaranteed to terminate when run in an initial state satisfying P. Let S be a program on variables Y (disjoint from X), which is intended to implement S'.

Programs S' and S are called the *abstract* and *concrete* programs, respectively. The correspondence between the states of these programs is represented by a predicate I on X and Y, called a *coupling invariant*. It often takes the form $X = F(Y)$, where F is a function that maps any state of S to the state of S' that it implements.

The property that S implements S' is formalized as

$$(P \wedge I) \Rightarrow wp(S, wp^*(S', I)). \tag{2}$$

Here, $wp(S, R)$ denotes the *weakest precondition* of S with respect to R [Dij76], the set of states in which any execution of S is guaranteed to terminate in a state satisfying R, and $wp^*(S', R)$ denotes the *angelic weakest precondition* of S' with respect to R, the set of states in which *some* execution of S' will terminate in a state satisfying R.[2] Thus, (2) specifies a correspondence between the effect of concrete program S and that of the abstract program S', when both programs are viewed as predicate transformers. In particular, (2) asserts that concrete program S changes its variables in a way that is consistent—as defined by I—with some execution of the abstract program it implements.

[2]The relationship between wp and wp^* is as follows:

$$wp(S, true) \Rightarrow (wp^*(S, R) \Leftrightarrow \neg wp(S, \neg R)).$$

For deterministic S, $wp(S, R) \Leftrightarrow wp^*(S, R)$.

2.4 Serializability as "Implements"

Serializability of $\Sigma = \langle V, C, T \rangle$ can be formalized in terms of $\Sigma' = \langle V', C', T' \rangle$ using (2) by taking T' of (1) to be the abstract program and T to be the concrete program. To do so, however, variables V' in C' and T' must be replaced by fresh variables $v'_0, \ldots, v'_{M-1}$ disjoint from V. Henceforth, we assume that V and V' are disjoint.

For P in (2) we take C' and for I we take $\bigwedge_{v_i \in V} v_i = v'_i$ giving the following characterization of serializability.

Definition: Σ is serializable if and only if

$$(C' \wedge \bigwedge_{v_i \in V} v_i = v'_i) \Rightarrow wp(T, wp^*(T', \bigwedge_{v_i \in V} v_i = v'_i)). \tag{3}$$

Some simplification of (3) is possible. Since $\tau'_0, \ldots, \tau'_{N-1}$ run serially in T', any execution of T' will be equivalent to some execution of a sequential program ρ in the set

$$\mathcal{S}(T'): \ \{\phi_0; \ldots; \phi_{N-1} \mid \phi_i \text{ is a transaction of } T', 0 \leq i < N\}.$$

Thus, for any predicate R,

$$wp^*(T', R) \Leftrightarrow \bigvee_{\rho \in \mathcal{S}(T')} wp^*(\rho, R). \tag{4}$$

Substituting the right-hand side of (4) into (3) gives the equivalent formula

$$(C' \wedge \bigwedge_{v_i \in V} v_i = v'_i) \Rightarrow wp(T, \bigvee_{\rho \in \mathcal{S}(T')} wp^*(\rho, \bigwedge_{v_i \in V} v_i = v'_i)). \tag{5}$$

When transactions of T' are deterministic, (3) can be further simplified. Each $\rho \in \mathcal{S}(T')$ is now deterministic. Since wp and wp^* are equivalent for deterministic programs, (5) is equivalent to

$$(C' \wedge \bigwedge_{v_i \in V} v_i = v'_i) \Rightarrow wp(T, \bigvee_{\rho \in \mathcal{S}(T')} wp(\rho, \bigwedge_{v_i \in V} v_i = v'_i)). \tag{6}$$

3 Proof Techniques for Serializability

3.1 Hoare's Logic

Verifying that a database system Σ satisfies any of (3), (5) or (6) presents a difficult problem: the weakest precondition of a concurrent program such as T is a complicated predicate that is difficult to evaluate. Hoare's logic [Hoa69] provides an alternative and often more tractable formalism for reasoning about concurrent programs. A *triple* is a formula

$$\{P\}S\{Q\} \tag{7}$$

where S is a program and P and Q are predicates on variables of S. Predicates P and Q are called *assertions* with P designated the *precondition* and Q the *postcondition* of S.

The triple (7) has the following interpretation:

> If execution of S begins in a state satisfying P and S terminates, then the state reached will satisfy Q.

Since this interpretation implies nothing about the termination of S, a triple specifies *partial correctness*. Axioms and inference rules of Hoare's logic for a simple sequential programming language can be found in [Hoa69]. Additional axioms and rules for concurrent programs are given in [OG76].

3.2 Effective Criteria Serializability

The relationship between a triple and wp is

$$Q \Rightarrow wp(S,R) \qquad \text{iff} \qquad \{Q\}S\{R\} \quad \text{and} \quad term(S,Q) \tag{8}$$

for any predicates Q and R and any program S, where $term(S,Q)$ specifies that S is guaranteed to terminate when started in a state satisfying Q. Thus, our definition of serializability in Section 2.4 and formulas (3), (5) and (6) imply the following theorem and corollary.

Theorem 1 Let $\Sigma = \langle V, C, T\rangle$ be a database system and Σ' a corresponding abstract system. Σ is serializable if and only if

$$\textbf{T1.1:}\quad \{C' \wedge \bigwedge_{v_i \in V} v_i = v_i'\}\, T\, \{\bigvee_{\rho \in S(T')} wp^*(\rho, \bigwedge_{v_i \in V} v_i = v_i')\},$$

$$\textbf{T1.2:}\quad term(T, C' \wedge \bigwedge_{v_i \in V} v_i = v_i').$$

Corollary 2 If the transactions of T' are deterministic, then Σ is serializable if and only if

C2.1: $\{C' \wedge \bigwedge_{v_i \in V} v_i = v_i'\}\, T\, \{\bigvee_{\rho \in \mathcal{S}(T')} wp(\rho, \bigwedge_{v_i \in V} v_i = v_i')\}$,

C2.2: $term(T, C' \wedge \bigwedge_{v_i \in V} v_i = v_i')$.

The conditions of Theorem 1 and Corollary 2 provide effective criteria for verifying—using assertional reasoning—serializability of database systems. Validity of T1.1 and C2.1 can be established using the logic of [OG76]; validity of T1.2 and C2.2 can be established using temporal logic [Pnu81, MP81].

3.3 An Example

Returning to the example of Figure 1, note that transactions of Σ_0' are deterministic since those of Σ_0 are. Thus, Corollary 2 can be used to verify serializability of Σ_0 as follows. We first prove condition C2.1,

$$\{C_0' \wedge \bigwedge_{v_i \in V_0} v_i = v_i'\}\, T_0\, \{\bigvee_{\rho \in \mathcal{S}(T_0')} wp(\rho, \bigwedge_{v_i \in V_0} v_i = v_i')\}. \tag{9}$$

We abbreviate the formal proof of (9) with the following *proof outline* [OG76]:

$$\begin{array}{l}
\{C_0' \wedge s = s' \wedge (\forall i\colon 0 \leq i < N\colon in_i = in_i')\} \\
\langle \Delta_0, \ldots, \Delta_{N-1} := \emptyset, \ldots, \emptyset \rangle; \\
\{I0 \wedge (\forall i\colon 0 \leq i < N\colon \Delta_i = \emptyset)\} \\
[PO(Add_0) \parallel \ldots \parallel PO(Add_{N-1})] \\
\{I0 \wedge (\forall i\colon 0 \leq i < N\colon \Delta_i = in_i)\} \\
\{\bigvee_{\rho \in \mathcal{S}(T_0')} wp(\rho, \bigwedge_{v_i \in V_0} v_i = v_i')\}
\end{array} \tag{10}$$

Here, each $PO(Add_i)$ is the proof outline shown in Figure 2 below, and

$$\begin{array}{ll}
I0\colon & (\forall i\colon 0 \leq i < N\colon \emptyset \subseteq \Delta_i \subseteq in_i) \\
 & \wedge\, s = (s' \cup \bigcup_{0 \leq i < N} \Delta_i) \wedge (\forall i\colon 0 \leq i < N\colon in_i = in_i')
\end{array}$$

is an assertion that remains true throughout execution of T_0. Variables $\Delta_0, \ldots, \Delta_{N-1}$ are *auxiliary variables* [OG76, McC89]. They have type set and represent the difference between s and s'. Since they are used for purposes of proof only, they need not be implemented.

In verifying serializability of Σ_0, we next prove condition C2.2 of Corollary 2—that T_0 terminates when started in a state satisfying $C_0' \wedge \bigwedge_{v_i \in V_0} v_i =$

$$
\begin{array}{l}
\{ I0 \wedge \Delta_i = \emptyset \} \\
\langle\, t_i := 0 \,\rangle ; \\
\{ I0 \wedge 0 \leq t_i \leq |in_i| \wedge \Delta_i = in_i(0..t_i - 1) \} \\
\textbf{do } t_i \neq |in_i| \rightarrow \\
\qquad \{ I0 \wedge 0 \leq t_i < |in_i| \wedge \Delta_i = in_i(0..t_i - 1) \} \\
\qquad \langle\, s, \Delta_i := s \cup in_i(t_i), \Delta_i \cup in_i(t_i) \,\rangle ; \\
\qquad \{ I0 \wedge 0 \leq t_i < |in_i| \wedge \Delta_i = in_i(0..t_i) \} \\
\qquad \langle\, t_i := t_i + 1 \,\rangle \\
\qquad \{ I0 \wedge 0 \leq t_i \leq |in_i| \wedge \Delta_i = in_i(0..t_i - 1) \} \\
\textbf{od} \\
\{ I0 \wedge \Delta_i = in_i \}
\end{array}
$$

Figure 2: $PO(Add_i)$

v_i'. Since the loop in each Add_i executes exactly $|in_i|$ iterations, Add_i executes a bounded number of operations, each of which is an assignment that is guaranteed to terminate. Consequently, each Add_i terminates, and it follows that T_0 terminates.

By Corollary 2, therefore, Σ_0 is a serializable database system. Notice that an explicit concurrency control mechanism was not needed to achieve this serializability, even though transactions shared access to variable s. This, then, illustrates how our work can be used to prove correctness of solutions to the concurrency control problem when the semantics of individual transactions contribute to the solution.

The preceding example is misleading. Although effective, the criteria given by Theorem 1 and its corollary are not practical for verifying serializability of a database system. For all but the simplest database systems, the assertions used will be too large for a proof of T1.1 or C2.1 to be tractable, due to the number and complexity of serial executions $\rho \in \mathcal{S}(T')$. For example, consider the database system Σ_1 of Figure 3. Σ_1 is obtained from Σ_0 by adding variables $out_0, \ldots, out_{N-1}$ and transactions $List_0, \ldots, List_{N-1}$ that write the contents of s to these variables. The consistency constraint has been strengthened to require that contents written by a *List* transaction contain all or none of the elements being added by an *Add* transaction. This has made it necessary to synchronize transactions in order to prevent listing s when only part of some in_i has been added. Synchronization is accomplished using locking [KS79, Kor83]. Transactions synchronize by acquiring

$$
\begin{array}{llll}
\Sigma_1 = \langle & V_1: & \multicolumn{2}{l}{s, in_0, \ldots, in_{N-1}, out_0, \ldots, out_{N-1},} \\
& C_1: & \multicolumn{2}{l}{C_0 \wedge (\forall i,j\colon 0 \le i,j < N\colon out_i \cap in_j = in_j \vee out_i \cap in_j = \emptyset)} \\
& & \multicolumn{2}{l}{\wedge (\forall i,M\colon 0 \le i < N\colon \neg locked(Add_i, M) \wedge \neg locked(List_i, M))} \\
& T_1: & \multicolumn{2}{l}{[Add_0 \parallel \cdots \parallel Add_{N-1} \parallel List_0 \parallel \cdots \parallel List_{N-1}] \rangle}
\end{array}
$$

$$
\begin{array}{llll}
Add_i\colon & \langle\, \mathbf{lock(S)} \rangle; & List_i\colon & \langle\, \mathbf{lock(X)} \rangle; \\
& \langle\, t_i := 0 \rangle; & & \langle\, out_i := s \,\rangle; \\
& \mathbf{do}\ t_i \ne |in_i| \rightarrow & & \langle\, \mathbf{unlock(X)} \rangle \\
& \quad \langle\, s := s \cup in_i(t_i) \rangle; & & \\
& \quad \langle\, t_i := t_i + 1 \rangle & & \\
& \mathbf{od}; & & \\
& \langle\, \mathbf{unlock(S)} \rangle & &
\end{array}
$$

Figure 3: Database System Σ_1.

and releasing a lock, which can have either shared or exclusive mode, denoted by **S** and **X**, respectively. An **X**-lock is incompatible with other locks, so a transaction attempting to acquire either an **S**-lock or an **X**-lock will block until no other transaction holds an **X**-lock.[3] To ensure that transactions terminate when run in isolation, the consistency constraint requires that transactions hold no locks initially. We denote the fact that a transaction τ_i holds an M-lock by the predicate $locked(\tau_i, M)$.

Corollary 1 requires

$$\{C_1' \wedge \bigwedge_{v_i \in V_1} v_i = v_i'\}\ T_1\ \{\bigvee_{\rho \in \mathcal{S}(T_1')} wp(\rho, \bigwedge_{v_i \in V_1} v_i = v_i')\}$$

to be valid. In the postcondition of the analogous triple (9) for T_0, all disjuncts $wp(\rho, \bigwedge_{v_i \in V_0} v_i = v_i')$ were equivalent. For Σ_1 of Figure 3, the number of different disjuncts will be exponential in N, making it painful to verify.

3.4 Simpler Criteria for Serializability

When transactions are deterministic (as they are in Σ_0 and Σ_1), simpler criteria for serializability can be formulated by promoting abstract transactions from their passive role in the postcondition of C2.1 to a more active

[3] The semantics of operations on s have allowed shared-mode locks to be used in transactions Add_i, even though they modify s.

one. Let τ be a transaction of Σ and let τ' be the corresponding transaction of an abstract system for Σ. An *augmentation* of τ is a program τ^* obtained by substituting $\langle \alpha;\tau' \rangle$ for some atomic operation α that runs exactly once in any terminating execution of τ. An augmentation of T is a program

$$T^*: \quad [\tau_0^* \parallel \cdots \parallel \tau_{N-1}^*]$$

in which each τ_i^* is an augmentation of τ_i. The following theorem shows that serializability can be characterized using triples for augmentations.

Theorem 3 Let $\Sigma = \langle V, C, T \rangle$ be a database system with deterministic transactions and let $\Sigma' = \langle V', C', T' \rangle$ be an abstract system for Σ. Let T^* be an augmentation of T using transactions of T'. Σ is serializable if

$$\textbf{T3.1:} \quad \{C' \wedge \bigwedge_{v_i \in V} v_i = v_i'\}\ T^*\ \{\bigwedge_{v_i \in V} v_i = v_i'\},$$

$$\textbf{T3.2:} \quad \mathit{term}(T, C' \wedge \bigwedge_{v_i \in V} v_i = v_i').$$

Proof Due to Corollary 2, it suffices to show that conditions T3.1 and T3.2 imply C2.1 and C2.2. Since T3.2 and C2.2. are identical, it suffices to show T3.1 and T3.2 imply C2.1.

Assume T3.1 holds and consider a terminating execution of T that starts in a state satisfying the precondition of C2.1. Any such execution can be formally represented by a history of the form

$$\sigma: \quad s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots s_{M-1} \xrightarrow{\alpha_M} s_M,$$

where s_0 is the initial state, and $s_{i-1} \xrightarrow{\alpha_i} s_i$ denotes that atomic action α_i transforms state s_{i-1} to state s_i. To show that C2.1 is valid, it suffices to show that s_M satisfies the postcondition of C2.1.

For any execution σ of T, the construction of T^* implies that there is a corresponding history

$$\sigma^*: \quad s_0^* \xrightarrow{\alpha_1^*} s_1^* \xrightarrow{\alpha_2^*} s_2^* \ldots s_{M-1}^* \xrightarrow{\alpha_M^*} s_M^*$$

of T^* in which $s_0^* = s_0$ and α_i^* is either α_i or $\langle \alpha_i;\tau' \rangle$, where τ' is the abstract transaction used to form the τ^* that contains α_i^*. Validity of T3.1 implies that s_M^* satisfies $\bigwedge_{v_i \in V} v_i = v_i'$.

Since V and V' are disjoint, the abstract transactions in σ^* can be pulled from their atomic operations and permuted with operations α_i to obtain a history

$$\sigma': \quad s'_0 \xrightarrow{\alpha_1} s'_1 \xrightarrow{\alpha_2} s'_2 \ldots s'_{M-1} \xrightarrow{\alpha_M} s'_M \xrightarrow{\tau'_0} s'_{M+1} \ldots s'_{M+N-1} \xrightarrow{\tau'_{N-1}} s'_{M+N}$$

of T followed by T' in which $s'_0 = s_0$, $s'_{M+N} = s^*_M$ and $\tau'_0, \ldots, \tau'_{N-1}$ are the abstract transactions of T' in the order they appear in σ^*.

Transactions are assumed to be deterministic, and the subsequence of σ' from s'_0 to s'_M has the same initial state and operations as the original execution sequence σ. Consequently, $s'_M = s_M$. The subsequence of σ' from s'_M to s'_{M+N} is one of the $\rho \in \mathcal{S}(T')$. Since $s'_{M+N} = s^*_M$ satisfies $\bigwedge_{v_i \in V} v_i = v'_i$, s'_M satisfies $wp(\rho, \bigwedge_{v_i \in V} v_i = v'_i)$, from which it follows that s_M satisfies the postcondition of C2.1. $\square$

The conditions given by Theorem 3 are simpler to verify than those of Theorem 2 or Corollary 2, due to shorter assertions in the proof of T3.1. However, this simplicity has been acquired at the expense of completeness. The conditions of Theorem 1 and its corollary are equivalent to serializability, while those of Theorem 3 only imply it. An example of a serializable database system for which T3.1 cannot be proven is given in [McC88].

3.5 Example Revisited

Theorem 3 can be used to prove Σ_1 of Figure 3 serializable, as follows. The following augmentations of each Add_i and $List_i$ are used:

Add^*_i: $\langle$ **lock(S)** $\rangle$;
$\langle\, t_i := 0 \,\rangle$;
do $t_i \neq |in_i| \rightarrow$
  $\langle\, s := s \cup in_i(t_i) \,\rangle$;
  $\langle\, t_i := t_i + 1 \,\rangle$
od;
$\langle$ **unlock(S)**; $Add'_i \rangle$

$List^*_i$: $\langle$ **lock(X)** $\rangle$;
$\langle\, out_i := s; List'_i \,\rangle$;
$\langle$ **unlock(X)** $\rangle$

We first prove T3.1,

$$\{C'_1 \wedge \bigwedge_{v_i \in V_1} v_i = v'_i\}\ T^*_1\ \{\bigwedge_{v_i \in V} v_i = v'_i\},$$

using the proof outline

$$
\begin{array}{ll}
\{\, C_1' \wedge \bigwedge_{v_i \in V_1} v_i = v_i' \} & \\
\langle\, \Delta_0,\ldots,\Delta_{N-1} := \emptyset,\ldots,\emptyset \rangle; & \\
\{ I1 \wedge (\forall i\colon\ 0 \le i < N\colon\ \Delta_i = \emptyset)\} & \\
[PO(Add_0^*) \parallel \ldots \parallel PO(Add_{N-1}^*) \parallel & (11) \\
\ PO(List_0^*) \parallel \ldots \parallel PO(List_{N-1}^*)] & \\
\{ I1 \wedge (\forall i\colon\ 0 \le i < N\colon\ \Delta_i = \emptyset)\} & \\
\{ \bigwedge_{v_i \in V_1} v_i = v_i' \} &
\end{array}
$$

In (11), $PO(Add_i^*)$ and $PO(List_i^*)$ are the proof outlines shown in Figures 4 and 5, and $I1$ is

$$
\begin{array}{l}
I0 \\
\wedge\ (\forall i\colon\ 0 \le i < N\colon\ out_i = out_i') \\
\wedge\ (\forall i\colon\ 0 \le i < N\colon\ \Delta_i \ne \emptyset \Rightarrow locked(Add_i, \mathbf{S})) \\
\wedge\ (\forall i,j\colon\ 0 \le i,j < N\colon\ \neg(locked(Add_i, \mathbf{S}) \wedge locked(List_j, \mathbf{X}))) \\
\wedge\ (\forall i,j\colon\ 0 \le i \ne j < N\colon\ \neg(locked(List_i, \mathbf{X}) \wedge locked(List_j, \mathbf{X})))
\end{array}
$$

Auxiliary variables $\Delta_0,\ldots,\Delta_{N-1}$ play the same role here as in the previous example. The proof of (11) is straightforward and is omitted here.

Condition T3.2 requires T_1 to terminate when started in a state satisfying $C_1' \wedge \bigwedge_{v_i \in V_1} v_i = v_i'$. The argument for this is analogous to that used to prove termination of Σ_0 except for the possibility of deadlock introduced by the addition of locking. Deadlock is impossible, however, since locking is two-phase [EGLT76].

4 Extensions

4.1 Modes of Termination

The database system model presented in Section 2.1 ignores certain aspects of actual database systems. One of these is the potential for transactions to abort. An aborting transaction typically executes a recovery protocol in which operations are run that undo the effects of its changes to the database, thereby giving the effect that it never ran.

We can incorporate this mode of transaction termination into our system model by including in each transaction τ_i an operation modeling its recovery protocol and including in V a Boolean variable $commit_i$, initially false, that τ_i sets to true if and only if it terminates without executing its recovery protocol. This change in the system model necessitates a change in the

$\{I1 \wedge \Delta_i = \emptyset \wedge \neg locked(Add_i, \mathbf{S})\}$
$\langle\, \mathbf{lock(S)} \,\rangle;$
$\{I1 \wedge \Delta_i = \emptyset \wedge locked(Add_i, \mathbf{S})\}$
$\langle\, t_i := 0 \,\rangle;$
$\{I1 \wedge 0 \le t_i \le |in_i| \wedge \Delta_i = in_i(0..t_i - 1) \wedge locked(Add_i, \mathbf{S})\}$
do $t_i \ne |in_i| \rightarrow$
$\quad \{I1 \wedge 0 \le t_i < |in_i| \wedge \Delta_i = in_i(0..t_i - 1) \wedge locked(Add_i, \mathbf{S})\}$
$\quad \langle\, s, \Delta_i := s \cup in_i(t_i), \Delta_i \cup in_i(t_i) \,\rangle;$
$\quad \{I1 \wedge 0 \le t_i < |in_i| \wedge \Delta_i = in_i(0..t_i) \wedge locked(Add_i, \mathbf{S})\}$
$\quad \langle\, t_i := t_i + 1 \,\rangle$
$\quad \{I1 \wedge 0 \le t_i \le |in_i| \wedge \Delta_i = in_i(0..t_i - 1) \wedge locked(Add_i, \mathbf{S})\}$
od;
$\{I1 \wedge \Delta_i = in_i \wedge locked(Add_i, \mathbf{S})\}$
$\langle\, \mathbf{unlock(S)}; \Delta_i := \emptyset; Add_i' \,\rangle$
$\{I1 \wedge \Delta_i = \emptyset \wedge \neg locked(Add_i, \mathbf{S})\}$

Figure 4: $PO(Add_i^*)$

definition of serializability as well. Our definition specifies in (3) that every execution of T corresponds to some execution of T'. However, there can be executions of T in which transactions abort for reasons that are not encountered in a serial execution (e.g., deadlock), and no execution of T' will correspond to these.

This problem is circumvented by chosing

$$T'': \quad [\tau_0'' \;||\; \cdots \;||\; \tau_{N-1}'']$$

as the abstract concurrent program for Σ' (instead of T'), where each transaction

$$\tau_i'': \quad \langle\, \mathbf{if}\ true \rightarrow \tau_i' \,[]\, true \rightarrow \mathbf{skip}\ \mathbf{fi} \,\rangle$$

executes one of τ_i' or **skip** when run, the choice being made nondeterministically. If **skip** is selected, all variables, including $commit_i'$, will be left unchanged, giving the same effect as if τ_i' ran but aborted. Every execution in $\mathcal{S}(T'')$ will now be equivalent to some serial execution ρ in the set

$$\mathcal{SS}(T'): \; \{\phi_0; \ldots; \phi_{k-1} \,|\, 0 \le k \le N,\ \phi_i \text{ is a transaction of } T',\ 0 \le i < k\}.$$

$$\{I1 \wedge \neg locked(List_i, \mathbf{X})\}$$
$$\langle\, \mathbf{lock(X)} \rangle;$$
$$\{I1 \wedge locked(List_i, \mathbf{X})\}$$
$$\langle\, out_i := s;\ List_i' \rangle;$$
$$\{I1 \wedge locked(List_i, \mathbf{X})\}$$
$$\langle\, \mathbf{unlock(X)} \rangle$$
$$\{I1 \wedge \neg locked(List_i, \mathbf{X})\}$$

Figure 5: $PO(List_i^*)$

There are several consequences of replacing T' by T''. One is that $\mathcal{S}(T')$ must be replaced by $\mathcal{SS}(T')$ in (5) and formulas derived from it. Consequently, an implementation of Σ that aborts every transaction will be serializable under our definition, violating constraints on transaction progress that are often assumed of databases. This can be avoided by specifying these constraints in addition to serializability.

Another consequence of using abstract transactions of T'' is that even when transactions of T are deterministic, those of T'' are not, and consequently cannot be used in augmentations to prove serializability as described in Section 3.4. Transactions of T' can still be used in augmentations, however, as long as each τ_i' is restricted to positions where it runs if and only if τ_i commits. The proof of Theorem 3 is virtually unchanged by this restriction, guaranteeing that serializability is still ensured by conditions T3.1 and T3.2.

4.2 Views

In our definition of serializability, the choice of the coupling invariant reflects an implicit assumption that transaction behavior is characterized by the entire system state. This may be too strong. Parts of the state of a real database system will be invisible to users of the system and need not be included when considering behavior. For example, the set s of database Σ_1 might be implemented using an array a that stores elements in contiguous locations. The order of elements in this array should not be considered when determining whether or not execution is serializable.

The visible aspects of the database system state are an abstraction of the system state. This can be modeled by using a function on system states that maps indistinguishable states to a common abstract representation. We call such a function a *view function.* We can incorporate a view function f into our definition of serializability by replacing the original coupling invariant $\bigwedge_{v_i \in V} v_i = v_i'$ by $\bigwedge_{v_i \in V} f(v_i) = f(v_i')$ in (3), together with formulas derived from it.

5 Discussion

We have defined serializability in terms of concurrent execution of transactions implementing serial execution. By choosing an assertional characterization of "implements", serializability was expressed using Hoare's logic. This makes it possible to verify concurrency control mechanisms using that logic.

There are many other definitions of serializability [Pap86]. What most of these definitions have in common is that serializability is defined as a property of system *schedules*, sequences of operations resulting from particular system executions. Schedule-based definitions of serializability fall into two broad categories based on how schedule behavior is characterized: *state based* and *conflict based*.

In state-based definitions, system behavior is described in terms of how schedules transform one state to another. Definitions differ with respect to the parts of the state considered significant. A schedule is *final-state* serializable if it and some serial schedule transform identical initial states to final states that agree on the value of all shared variables. A schedule is *view* serializable if the final states agree on the values obtained by read operations as well. Both final-state and view serializable schedules can be expressed by our definition of serializability by suitable choice of system variables.

Conflict-based definitions of serializability describe behavior somewhat indirectly, using *conflict relations* (also known as *dependency relations*) on operations of the schedule. An operation α_1 *conflicts* with another operation α_2 in the same schedule (written $\alpha_1 < \alpha_2$) if (i) the operations are from different transactions, (ii) α_1 precedes α_2, and (iii) α_1 and α_2 do not commute with each other (i.e., the same initial state can produce different final states when the operations are run in different orders). The conflict relation on operations is extended to one on transactions. This, then, is used to determine the set of serial schedules exhibiting the same behavior: a schedule is *conflict serializable* if it and some serial schedule have the same conflict relation on transactions.

$$\begin{array}{rll} \Sigma_2 = \langle & V_2\colon & \langle x, y, out, b \rangle, \\ & C_2\colon & 0 \le b \le 1 \wedge x + y = 100, \\ & T_2\colon & [\mathit{UpdateXY} \parallel \mathit{ListX}]\ \rangle \end{array}$$

$$\begin{array}{lll} \mathit{UpdateXY}\colon & \alpha_0\colon & \langle x, y := x + b * 17, y - \overline{b} * 17 \rangle; \\ & \alpha_1\colon & \langle x, y := x + \overline{b} * 17, y - b * 17 \rangle \\ \mathit{ListX}\colon & \beta_0\colon & \langle out := x \rangle \end{array}$$

Figure 6: Database System Σ_2.

Because schedule-based definitions of serializability consider operation sequences and system states independently, some database systems considered serializable under our definition are not considered serializable by the schedule-based definitions above. The (somewhat contrived) database system Σ_2 of Figure 6 is an example. There, variables x, y, and *out* hold integer values, while b holds a binary value. Transaction *UpdateXY* subtracts 17 from y and adds it to x, using the value of b to control the order in which this occurs. ($\overline{b}$ denotes the complement of b.) Using the techniques of Section 3, it is not difficult to prove Σ_2 serializable according to our definition.

Consider the schedule

$$\alpha_0; \beta_0; \alpha_1. \tag{12}$$

Note that the value read by *ListX* depends on the state in which execution begins: *out* gets the same value as it does from the serial schedule $\alpha_0;\alpha_1;\beta_0$ if $b = 1$ initially, while *out* gets the same value as from $\beta_0;\alpha_0;\alpha_1$ if $b = 0$. Thus, schedule (12) is neither final-state nor view serializable since these require (12) to "behave like" one of the serial schedules for all initial states. Nor is (12) conflict serializable, since the conflict relation on *UpdateXY* and *ListX* has a cycle. Thus, although Σ_2 is serializable according to our definition, it is not serializable by the schedule-based definitions of serializability.

In [Cas81], a definition of serializability similar to ours is given. This definition uses operators of Concurrent Dynamic Logic (CDL) instead of weakest preconditions to express the "implements" relationship between T and its serial model T'. Our definition is more general than the CDL definition, however, because the coupling invariant can be written in terms of a view function. Our definition also provides more useful criteria for verifying serializability, since Hoare's logic offers a variety of formal techniques for deriving and verifying triples that CDL currently lacks.

Acknowledgment

David Gries and Phil Hutto read and provided helpful comments on an earlier version of this paper.

References

[Cas81] M. Casanova. *The Concurrency Control Problem for Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, New York, 1981.

[Dij76] 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.

[GP85] D. Gries and J. Prins. A new notion of encapsulation. In *Proceedings SIGPLAN Symposium on Language Issues in Programming Environments*, pages 131–139, Jun. 1985.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Kor83] 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.

[Lam88] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, Jan. 1988.

[McC88] E. R. McCurley. *An Assertional Characterization of Serializability and Locking*. PhD thesis, Cornell University, 1988.

[McC89] R. McCurley. A valid rule for auxiliary variable transformations. Technical Report GIT-ICS-89/11, Georgia Institute of Technology, 1989.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs, part I: The temporal framework. Technical Report STAN-CS-81-836, Stanford University, 1981.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

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

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pri87] J. F. Prins. *Partial Implementations in Program Derivation.* PhD thesis, Cornell University, 1987.