Semantic Conditions for Correctness at Different Isolation Levels *

Arthur J. Bernstein, Philip M. Lewis, Shiyong Lu
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA
{*art, pml, shiyong*}*@cs.sunysb.edu*

Abstract

Many transaction processing applications execute at isolation levels lower than SERIALIZABLE in order to increase throughput and reduce response time. The problem is that non-serializable schedules are not guaranteed to be correct for all applications. The semantics of a particular application determines whether that application will run correctly at a lower isolation level, and in practice it appears that many applications do. Unfortunately, we know of no analysis technique that has been developed to test an application for its correctness at a particular level. Apparently decisions of this nature are made on an informal basis. In this paper we describe such a technique in a formal way.

We use a new definition of correctness, semantic correctness, which is weaker than serializability, to investigate the correctness of such executions. For each isolation level, we prove a condition under which transactions that execute at that level will be semantically correct. In addition to the ANSI/ISO isolation levels of READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ, we also prove a condition for correct execution at the READ COMMITTED with first-committer-wins (a variation of READ COMMITTED) and at the SNAPSHOT isolation level. We assume that different transactions can be executing at different isolation levels, but that each transaction is executing at least at the READ UNCOMMITTED level.

1. Introduction

Serializability is the correctness criterion generally used in the literature to determine a schedule's correctness. Such a criterion is clearly inappropriate, however, in determining the correctness of schedules that are produced when a application is run at an isolation level lower than SERIALIZABLE since such schedules might no longer serializable. Hence, in this paper, we use a correctness criterion proposed in [6] and [3] (and further developed in [4]), called *semantic correctness*, which requires that the interleaved execution of a set of transactions have the same *semantic* effect as a serial schedule of the same transactions. The semantic correctness condition for a transaction schedule is based on the conditions developed in [11] for the correct execution of an arbitrary concurrent program.

The ANSI/ISO standard [1] defines three isolation levels lower than SERIALIZABLE: READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ. Database systems frequently use a locking protocol to implement these levels. In addition, at least one major database vendor uses SNAPSHOT isolation implemented through a combination of locking and multiversion techniques. An excellent analysis of these levels and a proposed locking implementation for each is given in [2].

In this paper we assume the locking implementation described in [2] and analyze the way transactions can be interleaved at each isolation level. The semantic correctness of a transaction schedule depends on the pattern of interleaving. By taking this pattern into account it is possible to greatly reduce the semantic analysis called for in [11] that is required to demonstrate the correctness of general concurrent programs. We present conditions for semantic correctness for each isolation level based on the allowable interleaving patterns at that level. These conditions provide the formal basis that underlies the informal reasoning that justifies the use of non-serializable isolation levels. They allow the application designer to choose the lowest isolation level for each type of transaction of an application in order to achieve high performance transaction processing.

2. Semantic correctness

When transactions are executed at isolation levels lower than SERIALIZABLE, the interleaving might no longer be serializable and correctness depends on what kind of interleaving can occur and what the transactions are doing.

*This paper is based upon work supported by NSF grant CCR-9402415.

Hence we need a way of describing what each transactions does: its semantics. The semantics of a transaction, T_i, can be formally characterized by the triple

$$\{I_i \wedge B_i \wedge (\overline{x_i} = \overline{\mathcal{X}_i})\}\ T_i\ \{I_i \wedge Q_i\} \qquad (1)$$

I is the *consistency constraint* of the database, and I_i represents those conjuncts of I required for the correct execution of T_i. For example, the consistency constraint of a banking database might assert that *all* accounts have non-negative balances. However, the correct execution of a transaction that accesses a particular account requires only that the balance of *that* account be non-negative. I_i represents the conjunction of those conjuncts of I that are required for the correct execution of T_i. I_i is also a postcondition of T_i since we require that any conjunct of I that is made false during the execution of T_i is returned to the true state when T_i terminates. It follows that $I \equiv \bigwedge_{i=1}^{n} I_i$, where n is the number of transaction types in the system. B_i describes all conditions that T_i assumes to be true of the arguments passed to it. For example, if T_i is a deposit transaction and dep is the parameter representing the money to be deposited, then B_i might assert $dep \geq 0$.

Q_i is called the *result* and asserts that T_i has performed its intended function. Continuing the above example, if T_i deposits into an account whose balance is bal, we need to assert as a postcondition of T_i that the final balance is dep more than the initial balance. In order to refer to the initial balance in the postcondition, we introduce a logical variable, $\overline{\mathcal{X}_i}$ whose sole purpose is to record the initial value of a database variable, x_i, whose value is changed by T_i. In the example, we characterize T_i with the triple

$$\{bal >= 0 \wedge dep \geq 0 \wedge bal = BAL\}\ T_i$$
$$\{bal >= 0 \wedge bal = BAL + dep\}$$

(1) goes beyond the consistency requirement placed on a transaction by asserting that not only must T_i move the database from one consistent state to another, but that only a subset of the consistent states are acceptable when the transaction terminates. (1) can be regarded as a formal restatement of the specification of T_i. We can demonstrate that T_i is correct by proving that (1) is a theorem using a formal system such as that of [10]. [1]

To overcome the limit of serializability and to increase performance, in [3] we propose a new correctness criterion, called semantic correctness. A schedule, Sch, of transactions is *semantically correct* if

$$\{I\}\ Sch\ \{I \wedge Q_{Sch}\} \qquad (2)$$

is true. First, a semantically correct schedule must maintain the consistency of the database, as indicated by the fact that I is a pre- and postcondition of Sch. A semantically correct schedule must also transform the database to a state that reflects the cumulative results of all the transactions in Sch in some order. We denote the assertion that describes that set of states by Q_{Sch}, the *cumulative result*. The relationship between Q_{Sch} and the results, Q_i, of the individual transactions is described in [4]. In essence, Sch is semantically correct if its postcondition is the same as the postcondition of a serial schedule of the same set of transactions, where the serial order is the order of transaction completion in Sch. For example, if Sch consists of several deposit transactions on some bank account, Q_{Sch} might assert that the final balance is greater than the initial balance by an amount equal to the sum of the deposits.

As illustrated in [3], semantic correctness is weaker than serializability, and it allows schedules that result in states that could not have been reached in any serial schedule. A semantically correct schedule can perform significantly better than any equivalent serial schedule [5].

A proof of (1) can be abbreviated by an annotated program in which each (atomic) statement of T_i, $S_{i,j}$, is preceded by an assertion, $P_{i,j}$, its precondition, describing the state of the system at the time $S_{i,j}$ is eligible for execution. Hence, assertions are associated with control points and we say that an assertion and its corresponding control point are *active* if the following statement is eligible for execution. Each assertion states some condition on the values of items in T_i's workspace and in the database. If $S_{i,j}$ is executed starting in a state where $P_{i,j}$ is true, the next assertion, $P_{i,j+1}$, will be true of the state when $S_{i,j}$ terminates. Hence, if when each statement is executed its precondition is true, the postcondition of the transaction will be true when the transaction terminates.

The major issue with concurrency is *invalidation*: if the execution of the statements of T_i and T_k $(i \neq k)$ are interleaved, $P_{i,j}$ might not be true of the database state when $S_{i,j}$ is initiated. Thus, if $S_{k,l}$ is executed when $P_{i,j}$ is active and true, it might transform the state to one in which $P_{i,j}$ is false. If this occurs, we say that $S_{k,l}$ has *invalidated* $P_{i,j}$. For example, the execution of the statement $x := x + 1$ will invalidate the assertion $x = y$, but not the assertion $x > y$. If, during execution of T_i, an active assertion is invalidated, its subsequent behavior will be unpredictable and semantic correctness is not guaranteed. A sufficient condition to ensure that no invalidation occurs is that, for all preconditions $P_{i,j}$ and all statements $S_{k,l}$, the triple [11]:

$$\{P_{i,j} \wedge P_{k,l}\}\ S_{k,l}\ \{P_{i,j}\} \qquad (3)$$

is a theorem. A similar condition must be specified for Q_i.

If (3) cannot be proven, we say that the proofs of T_i and T_k *interfere* with one another and in particular, $S_{k,l}$ *interferes with* $P_{i,j}$. Hence there is a *possibility* of invalidation at run time if the interleaving actually occurs. Interference is static. It is a property of the proofs of the transactions.

[1] although this is not generally done.

Invalidation is dynamic. It is a result of interference if the interfering operation is executed when the interfered with assertion is active. Hence, interference does not necessarily lead to invalidation. For example, in [4] each transaction is decomposed into atomic isolated steps. A concurrency control is used to prevent those step interleavings that lead to invalidation from happening, thus ensuring that all schedules are semantically correct.

In [4] we apply the results of [11] to a schedule, Sch, of transactions and show that Sch is semantically correct if for all transactions, T_i, in Sch

1. $P_{i,j}$ is true when $S_{i,j}$ is executed and
2. Q_i is not invalidated by any step of a transaction interleaved in Sch with T_i.

By extension, we say that a particular transaction, T_i, is semantically correct in Sch if these conditions apply to $P_{i,j}$ and Q_i.

As indicated by (3), checking for non-interference involves examining each statement and each assertion of all transactions. This checking requires a significant amount of work. For example, in a system of K transaction types, each containing N operations, $(KN)^2$ possible triples must be checked. When the isolation levels of transactions are taken into account, however, the number of triples that must be checked is greatly reduced since the locking discipline that implements the levels prevents certain interleavings from occurring. Hence, although a particular triple (3) may not be a theorem, it cannot result in invalidation if the locking discipline prevents $S_{k,l}$ from being executed when $P_{i,j}$ is active. A major goal of this paper is to determine, for each isolation level, which triples must be checked. This dramatically reduces the amount of analysis. For example, for SMAPSHOT isolation only K^2 triples must be checked, regardless of the number of operations per transaction.

3. Semantic conditions for conventional databases

In this section we present conditions which, for each isolation level, enumerate the non-interference theorems that must be demonstrated in order to ensure semantic correctness. We first consider conventional databases and then relational databases. In conventional databases, no database items are deleted or inserted, and each item is referred to by name in read or write statements. In relational databases, predicates are used in SQL statements to specify the database items they access.

3.1. Model

A transaction program accesses local variables (in its workspace) and database variables using the following constructs.

1. assignment statement. There are three kinds of assignment statements: a read statement, which atomically assigns the value of a database item to a local variable, a write statement, which atomically assigns the value of a local variable to a database item, and a local assignment statement, which does not involve any database items.
2. conditional statement. We assume that the condition is constructed from local variables.
3. loop statement. We assume that the condition is constructed from local variables.

Local variables will be denoted X, Y and database variables will be denoted x, y. We use the notation $sp(P, S_{i,k})$ to denote the strongest postcondition of $S_{i,k}$ that can be asserted when it is executed starting in a state that satisfies P. We assume the protocols given in [2] are used to implement all isolation levels.

3.2. Semantic condition for READ UNCOMMITTED

The locking implementation for READ UNCOMMITTED [2] requires that transactions obtain long-term write locks on items that they write,[2] but no read locks are acquired on items that they read. Long term locks are held until the transaction completes. The following theorem states a condition under which a transaction will execute correctly at READ UNCOMMITTED.

Theorem 1 *A transaction, T_i, that executes at READ UNCOMMITTED will execute semantically correctly if each write statement (including those that rollback a transaction) in every transaction does not interfere with I_i, the postcondition of every read statement in T_i, and Q_i.*

Proof: See Appendix B. $\mathcal{Q.E.D.}$

Since a transaction executing at READ UNCOMMITTED can read uncommitted data, it is necessary to consider the interference caused by write statements that rollback any transaction.

Suppose a transaction executing at READ UNCOMMITTED executes a statement, s, that updates data item x. The fact that the transaction will then hold a long term write lock on x implies that no write statement, s', accessing x in a concurrent transaction can be interleaved after s, and hence s' cannot possibly invalidate $post(s)$ (or any subsequent assertions).

[2] Database systems often prohibit transactions at this level from updating the database. We ignore this restriction here.

Example 1: The elements of array $cust$ are records describing a merchant's customers, and an integrity constraint of the database, I_c, asserts this fact. Two transaction types access the array. The $Mailing\_List$ transaction type scans the array and prints a mailing label. The specification of the transaction requires only that each printed label contains a valid name and address. The New_Order transaction type enters a new record into the array if the customer placing the new order is not described by an existing record. Each time a label is printed by a $Mailing\_List$ transaction we would like to assert "The printed label contains a valid name and address". Neither I_c nor this assertion is interfered with by either the insertion of a new record by an instance of New_Order or the deletion of a record if that instance is rolled back. Hence $Mailing\_List$ transactions can run at READ UNCOMMITTED. The database might include a second array in which each record describes an order that has been placed. A transaction that analyzes last year's orders can run at READ UNCOMMITTED since only this year's orders are subject to modification.

3.3. Semantic condition for READ COMMITTED

The locking implementation of READ COMMITTED [2] requires that transactions obtain long-term write locks on items that they write and short-term read locks on items that they read. A short term lock is released when the operation completes. The following theorem states a condition under which a transaction will execute correctly at READ COMMITTED.

Theorem 2 *A transaction, T_i, executed at READ COMMITTED will execute semantically correctly if each transaction does not interfere with the postcondition of every read statement in T_i, and with Q_i.*

Proof: See Appendix B. $\mathcal{Q.E.D.}$

Example 2: If the specification of the $Mailing\_List$ transaction is strengthened to require that all labels refer to customers, $Mailing\_List$ transactions must be run at least at the READ COMMITTED level since the new postcondition of $Mailing\_List$ "The printed label refers to a customer" is interfered with by the update statement that deletes an entry in $cust$ if New_Order is rolled back.

In addition to $cust$, the database might contain an array emp, with one record for each employee. $emp[i].rate$ is the i^{th} employee's hourly rate, $emp[i].num\_hrs$ is the number of hours that employee has worked so far this week, and $emp[i].sal$ is that employee's accumulated salary for the week. A conjunct of the integrity constraint, I_{sal} asserts that for all records in emp, "$emp[i].rate * emp[i].num\_hrs = emp[i].sal$".

The granularity of locking is at the level of records. An instance of transaction type $Hours$ is executed at the end of each workday to record the number of hours worked by an employee that day. It executes one write statement to increment $emp[i].num\_hrs$ and another to update the accumulated salary. Hence, although the two write statements together preserve I_{sal}, individually they do not. A second transaction type, $Print\_Records$ causes the records to be printed. Its specifications require that each printed record is a consistent snapshot of that employee's information at the time the record is printed. This specification makes it necessary that $Print\_Record$ be run at a level no lower than READ COMMITTED for two reasons: (1) Only committed information can be printed. (2) It follows from Theorem 2 that, in order to ensure that the snapshot of $emp[i]$ is consistent, the write statements of $Hours$ must be seen as an atomic unit by $Print\_Records$. The specification does not require that all printed records come from the same committed snapshot of emp. Hence, it is not necessary to prohibit instances of $Hours$ from updating records that have been printed while $Print\_Records$ is printing other records. As a result the long term read locks that would be acquired on each record printed if $Print\_Records$ were run at REPEATABLE READ are not required.

3.4. Semantic condition for READ COMMITTED with first-committer-wins

The *READ COMMITTED with first-committer-wins* isolation level is an extension of READ COMMITTED with one feature from the SNAPSHOT isolation level. Transactions obtain long-term write locks on items that they write and short-term read locks on items that they read. In addition, if T_1 writes a data item and commits between the times that T_2 has read and attempts to write the item, T_2 will be aborted (first-committer-wins). READ COMMITTED with first-committer-wins is easily and often implemented in relational databases by running an application at the READ COMMITTED level and encoding, (perhaps using sequence numbers) in the UPDATE statements of the application, checks to determine whether the data item to be updated has changed since it was read. The isolation level is also supported by a number of vendors. Some vendors call this level READ COMMITTED with optimistic reads.

Theorem 3 *A transaction, T_i, executed at READ COMMITTED with first-committer-wins will execute semantically correctly if each transaction does not interfere with the postconditions of those read statements in T_i that are not followed by a write statement on the same item, and with Q_i.*

Proof: See Appendix B. $\mathcal{Q.E.D.}$

Note that if transaction T_i writes all the data items it reads, then when T_i commits, T_i has effectively held long term read locks on the data items that it read, and hence in

this case, READ COMMITTED with first-committer-wins is equivalent to REPEATABLE READ. We give an example of correct execution at the READ COMMITTED with first-committer-wins level in Section 6.

3.5. Semantic condition for REPEATABLE READ

The locking implementation of the REPEATABLE READ isolation level [2] requires that a transaction acquire long-term read and write locks on the data items that it accesses. The only problem at the REPEATABLE READ level is the possibility of phantoms [8]. Since phantoms do not occur in conventional (non-relational) databases, REPEATABLE READ ensures serializability and hence semantic correctness. Thus we have the theorem:

Theorem 4 *Under the conventional database model, a transaction executed at REPEATABLE READ executes semantically correctly.*

3.6. Semantic condition for SNAPSHOT isolation

SNAPSHOT isolation is not one of the ANSI/ISO standard isolation levels but is implemented in at least one commercial DBMS. The implementation of SNAPSHOT isolation given in [2] does not use locks. Instead, it uses a multiversion concurrency control that satisfies each read request made by transaction T_i with values from the version of the database, called a *snapshot*, that reflects the effect of all committed transactions at the time T_i was started. Hence read requests never wait. Writing is deferred until the transaction commits. T_i can be committed as long as no other transaction that committed after T_i's first read has updated a data item that T_i has also updated. This mechanism is referred to as *first committer wins*, because the first transaction that has updated a particular data item and requests to commit is allowed to do so, while concurrent transactions that have updated that item are ultimately aborted. Thus *first committer wins* has at least the effect of a long-term write locks on the items written.

We model a transaction T_i at the SNAPSHOT isolation level as two isolated atomic steps: a *read step* followed by a *write step*. The read step reads a snapshot of the database that reflects the effect of all committed transactions at the time T_i was started. The write step is the remainder of the transaction. The step boundary reflects the fact that other transactions can commit while T_i is active, creating new versions of the database that might invalidate assertions that T_i has made about the database based on its snapshot. If T_i commits, its write step must commute with the write steps of these other transactions because they must have written to disjoint data items. Note that the postcondition of the snapshot does not necessarily state that the value of a data item in a snapshot is equal to the most recent committed value of that data item. It only needs to be strong enough to support the proof of the transaction [6].

Theorem 5 *A schedule produced under SNAPSHOT isolation is semantically correct if, given any two transactions T_i and T_j from the schedule, either:*

1. *T_i's write set intersects T_j's write set or*
2. *T_j does not interfere with the postcondition of the read step of T_i, and with Q_i.*

Proof: The read step of T_i reads the snapshot of the database that reflects the effect of all committed transactions at the time T_i was started. This snapshot either reflects the whole result of T_j or it does not reflect any result of T_j. Thus, when we reason about the semantic correctness of T_i in a schedule that includes T_i and T_j, T_j can be considered as a single isolated unit. If (1) applies, then either T_i or T_j will be aborted and has no effect. If not, then using Lemma 1, Lemma 2 and condition 2 of the theorem, it follows that no assertion in T_i will be invalidated by T_j. Note that since T_j preserves I, the precondition of T_i is not interfered with by T_j. $\diamond$

Example 3: Suppose we have two types of withdraw transactions, $Withdraw\_sav(i, w)$ and $Withdraw\_ch(i, w)$, which withdraw w from the i^{th} depositor's savings and checking accounts, respectively. Savings and checking account information is held in arrays $acct\_sav$ and $acct\_ch$ respectively and a conjunct of the integrity constraint, I_{bal} requires that $acct\_sav[i].bal + acct\_ch[i].bal \geq 0$. An annotated version of the $Withdraw\_sav$ program is given in Figure 1. The annotation for $Withdraw\_ch$ is similar.

Sav and Ch are local variables. The postcondition of the read step of $Withdraw\_sav$ is interfered with by the write step of $Withdraw\_ch$. Hence, the theorem states that a concurrent schedule of the two transactions might not be semantically correct. A schedule in which the write step is interleaved between the read and write step of the other exhibits write skew [2]. Note that although this same precondition is also interfered with by another instance of $Withdraw\_sav$, a concurrent schedule in which two instances of $Withdraw\_sav$ are interleaved is semantically correct because the first-committer-wins rule implies that one of them will be aborted (this is reflected in the second condition of the theorem). Finally, a $Deposit\_sav$ ($Deposit\_ch$) transaction, which adds money to $acct\_sav$ ($acct\_ch$) does not interfere with the postcondition of the read step of $Withdraw\_sav$. In this case, their concurrent execution is semantically correct (this is reflected in the third condition of the theorem).

$Withdraw\_sav(i, w)$
BEGIN TRANSACTION
$\{acct\_sav[i].bal + acct\_ch[i].bal \geq 0$
$\wedge\ Sav = Sav_0\}$
$Sav := acct\_sav[i].bal;$
$Ch := acct\_ch[i].bal;$
$\{acct\_sav[i].bal + acct\_ch[i].bal \geq 0$
$\wedge\ acct\_sav[i].bal + acct\_ch[i].bal \geq$
$Sav + Ch \wedge Sav = Sav_0\}$
if $(Sav + Ch \geq w)\ then$
$\{acct\_sav[i].bal + acct\_ch[i].bal \geq 0 \wedge$
$acct\_sav[i].bal + acct\_ch[i].bal \geq$
$Sav + Ch \wedge Sav + Ch \geq w$
$\wedge\ Sav = Sav_0\}$
$acct\_sav[i].bal := Sav - w;$
fi
$\{acct\_sav[i].bal + acct\_ch[i].bal \geq 0$
$\wedge (acct\_sav[i].bal = Sav_0 - w)\}$
END TRANSACTION

Figure 1. Withdraw from savings account

4. Semantic conditions for relational databases

In adapting the conditions for semantic correctness to relational databases, we must deal with database operations that involve predicates. The read statement is now the SELECT and its postcondition might assert that it read all the tuples that satisfy a certain predicate. Similarly, the write statements are UPDATE, INSERT, and DELETE and their postconditions might assert that they wrote, inserted, or deleted all the tuples that satisfy a certain predicate.

Interference now takes new forms. For example, the postcondition of a SELECT statement might assert that the statement read all the tuples that satisfy a predicate, P. That assertion can be interfered with by another transaction that inserts a phantom tuple that also satisfies P.

Phantoms can occur in connection with write statements as well as in connection with the SELECT. Thus, the postcondition of an UPDATE that asserts that the value of all tuples satisfying P have been updated can be interfered with by an INSERT that inserts a phantom tuple that satisfies P. That interference might not cause invalidation of the predicate, however, if the locking policy prevented the INSERT from executing after the UPDATE had taken place.

The locking policy for implementing the ANSI isolation levels discussed in [2] states that all "write locks on data items and predicates (are) long duration". Thus when an UPDATE, INSERT, or DELETE statement refers to a predicate, that predicate is write-locked for the duration of the transaction, and phantoms cannot be inserted into that predicate. Most DBMSs do not implement predicate locking, but instead use a locking protocol (perhaps consisting of some combination of table locks and index locks) that is equivalent to, or stronger than, predicate locking. We assume in what follows that the DBMS uses such a locking protocol. Then Theorems 1, 2, 3 remain valid for relational databases. A more complete treatment of the relational case is contained in a full version of this paper [7].

Theorem 4 can be restated for relational databases by considering the possibility of phantoms. At REPEATABLE READ, the long term read locks obtained on tuples read by a SELECT statement block the execution of a statement in a concurrent transaction that attempts to delete or update such tuples. Hence, the postcondition of the SELECT statement cannot be invalidated by a transaction that attempts to delete or update such a tuple. As a result we get the following theorem. Its proof can be found in [7].

Theorem 6 *For a transaction, T_i, executed at REPEATABLE READ, let $S_{i,j}$ be an arbitrary SELECT in T_i. T_i will execute semantically correctly if each transaction does not interfere with Q_i and either (1) does not interfere with the postcondition of $S_{i,j}$, or (2) includes DELETE or UPDATE statements whose predicates intersect the predicate of $S_{i,j}$.*

For SNAPSHOT isolation, we model a transaction in the same way as we did for conventional databases. Theorem 5 remains valid for relational databases. A proof can be found in [7].

5. Choosing an isolation level

Given the set of transactions types of an application, the problem faced by the application designer is to determine, for each type, the lowest isolation level at which a transaction of that type can execute correctly. Since SNAPSHOT isolation is not generally offered in the context of the other isolation levels, we do not consider it in what follows.

Using the previous results it follows that while we determine the isolation level at which to execute transaction, T_1, we do not have to be concerned about the level of other transactions. Specifically, it we are performing an interference analysis to determine the correctness of executing T_1 at READ UNCOMMITTED, we must consider the interference effects of each write of another transaction, T_2, individually, regardless of the level of T_2. Similarly, if we are considering executing T_1 at any higher level, we consider the interference effects of the whole transaction T_2 as an atomic isolated unit, regardless of the level of T_2. Then a procedure for determining the lowest isolation level at which each transaction can execute semantically correctly is: for each transaction, T_i, in the set, consider the isolation levels READ UNCOMMITTED, READ COMMIT-

TED, REPEATABLE READ, and SERIALIZABLE in sequence and return the first at which execution is semantically correct.

6. an example

To motivate the conditions for semantic correctness in a relational setting, consider a business application that accesses a schema with the following three tables (primary keys are underlined):

ORDERS(order_info, cust_name, deliv_date, done)
CUST(cust_name, address, #orders)
MAXDATE(maximum_date)

A conjunct of I, I_o, asserts that each row of ORDER describes an order and done is true if that order has been delivered. MAXDATE is a table containing a single row that satisfies a second conjunct, I_{max}, that asserts that maximum_date is the maximum delivery date for any order in ORDERS.

This application has four transaction types shown in Figures 2 through 5. Each figure shows an annotation of a transaction program indicating the pre- and postconditions of the transaction and the postcondition of each SELECT statement. These are the critical assertions. The purpose of the figures is to display the critical assertions; the code is just sketched.

Mailing_List (Figure 2): This transaction scans CUST and prints a label using cust_name and address. The specification of the transaction places no condition on the labels printed. Since no critical assertion is interfered with by any single write statement in any of the transaction types, this transaction will execute correctly at READ UNCOMMITTED.

$Mailing\_List()$
BEGIN TRANSACTION
    {$true$}
  SELECTcust_name, address FROM CUST
    {*Returned data contains names and addresses*}
    Print labels using returned names and addresses
    {*Labels have been printed*}
END TRANSACTION

Figure 2. Prints out a mailing list

New_Order (Figure 3): This transaction inserts a new order into ORDERS and, if this is the first order for *customer*, inserts a new tuple into CUST. In order to keep the delivery truck busy, a business rule asserts that there can be no gaps in the sequence of delivery dates: there must be at least one order to be delivered on each date up to some maximum date which is the delivery date of the last outstanding order. A conjunct of the integrity constraint of the database, which we call "no_gaps", asserts this fact. However, there can be more than one order for any particular delivery date. Furthermore, the number of orders for a particular customer in ORDERS must be equal to the value of the #orders field of that customer's tuple in CUST. We refer to this integrity constraint as "$order\_consistency$". The intermediate assertion I'_{max} in Figure 3 asserts that maximum_date is one greater than the latest delivery date in ORDERS. Thus New_Order reads the value of maximum_date in MAXDATE into the workspace variable *maxdate*; and increments maximum_date in MAXDATE by 1. If the customer is new it inserts the tuple $(customer,\ address,\ 1)$ into CUST; otherwise it increments #orders in the customer's tuple in CUST. [3] Finally, it inserts $(order\_info,\ customer,\ maxdate + 1,\ false)$ into ORDERS.[4]

Since no critical assertion is interfered with by any transaction type, this transaction can run at READ COMMITTED. The transaction cannot run at READ UNCOMMITTED because, for example, the no_gap assertion that is a conjunct of assertions in a New_Order transaction, T_1, is interfered with by the rollback statement of another New_Order transaction, T_2, that deletes a tuple from ORDERS (it might leave a gap in delivery dates below the delivery date selected by T_1).

Suppose an additional business rule is imposed: there must be *exactly* one order with a particular delivery date for each date up to some maximum. The "no_gap" conjunct of the integrity constraint is replaced by the conjunct "$one\_order\_per\_day$" which asserts the new requirement. The same *New_Order* transaction can be used to enforce this rule if it is run at READ COMMITTED with first-committer-wins. At READ COMMITTED the INSERT into ORDERS in the New_Order transaction interferes with the conjunct $one\_order\_per\_day$ in the postcondition of the SELECT. However, since New_Order updates MAXDATE after reading it, $one\_order\_per\_day$ cannot be invalidated at the READ COMMITTED with first-committer-wins isolation level. Also note the postcondition of the whole transaction is not interfered with by any transaction type, and thus this transaction can run at READ COMMITTED with first-committer-wins.

[3] The postcondition of New_Order in Figure 3 indicates that the inserted tuple has a particular value in the #orders field. Since the value will change as the customer adds new orders, in order to avoid interference the postcondition should actually be weakened to assert that *at commit time* this tuple was an element of CUST .

[4] Since the value of done will subsequently change, the comments in the previous footnote apply.

$New\_Order(customer,\ address,\ order\_info)$
BEGIN TRANSACTION
$\{no\_gap \wedge order\_consistency \wedge I_{max}\}$
SELECT maximum_date FROM MAXDATE
INTO : $maxdate$
$\{no\_gap \wedge order\_consistency$
$\wedge I_{max} \wedge (maxdate \leq$ maximum_date$)\}$
UPDATE MAXDATE SET
maximum_date = : $maxdate + 1$
SELECT COUNT(*) INTO : $custcount$ FROM
ORDERS WHERE cust_name = : $customer$
$\{no\_gap \wedge order\_consistency \wedge I'_{max}$
$\wedge (maxdate \leq$ maximum_date$) \wedge$
$(custcount = 0) \Rightarrow (customer\ is\ new)\}$
if (custcount == 0) then
INSERT INTO CUST
VALUES (: $customer$, : $address$, 1)
else
UPDATE CUST SET #orders = :$custcount$+1
WHERE cust_name = : $customer$
fi
INSERT INTO ORDERS VALUES
(: $order\_info$, : $customer$, : $maxdate + 1$, $false$)
$\{no\_gap \wedge order\_consistency \wedge I_{max} \wedge$
$(customer,\ address,\ custcount + 1) \in cust$
$\wedge (order\_info,\ customer,\ maxdate + 1,\ false)$
$\in orders\}$
END TRANSACTION

Figure 3. Processes a new order

Delivery (Figure 4): This transaction delivers an order. Thus $Delivery$ scans ORDERS to find all the orders that are due today and updates the done attributes in the orders to be delivered to $true$.

The postcondition of the SELECT statement of a $Delivery$ transaction is interfered with by another $Delivery$ transaction. Thus this transaction type cannot execute at READ COMMITTED. However, if the transaction is executed at REPEATABLE READ, the selected tuples are read locked after the SELECT statement is executed. Hence a $Delivery$ transaction would not be allowed to update these tuples and the assertion could not be invalidated. Thus this transaction meets the condition for correct execution at the REPEATABLE READ isolation level.

Audit (Figure 5): This transaction checks that the integrity constraint $order\_consistency$ is true. Thus $Audit$ scans ORDERS and counts the number of orders registered for a particular customer; reads the tuple for that customer in CUST and compares #orders with the count.

$Delivery(today)$
BEGIN TRANSACTION
$\{I_o\}$
SELECT order_info INTO : $buff$ FROM ORDERS
WHERE deliv_date = : $today$ AND done = FALSE
$\{I_o \wedge$ *returned values are undelivered orders to be delivered today*$\}$
while ((ord_inf := next in $buff$)
UPDATE ORDERS SET done = TRUE
WHERE order_info = : ord_inf
$\{I_o \wedge$ (*tuples in* ORDERS *describing orders due today have* done = TRUE)$\}$
END TRANSACTION

Figure 4. Delivers an order

$Audit(customer)$
BEGIN TRANSACTION
$\{I_o\}$
SELECT COUNT(*) INTO : $count1$ FROM ORDERS
WHERE cust_name = : $customer$
$\{I_o \wedge$ (*count1 = the number of tuples in* ORDERS *for customer*)$\}$
SELECT #orders INTO : $count2$ FROM CUST
WHERE cust_name = : $customer$
$\{I_o \wedge$ (*count1 = the number of tuples in* ORDERS *for customer*) $\wedge$ (*count2 = the value of #orders in* CUST *for customer*)$\}$
$retv := (count1 == count2);$
$\{I_o \wedge (retv = order\_consistency)\}$
END TRANSACTION

Figure 5. Produces accounting information

This transaction must run at the SERIALIZABLE level because the postconditions of both SELECT statements might be interfered with by a New_Order transaction that inserts a (phantom) new order. Note that this transaction does not satisfy the second half of the condition for correct execution at REPEATABLE READ because tuple locks do not prevent the insertion of a phantom new order.

7. Conclusion and future work

We have used semantic correctness as the criterion to investigate the correctness of schedules at different isolation levels. Specifically, for each isolation level, we prove a condition under which transactions that execute at that level will be semantically correct. This technique also clarifies the relationship between interference and invalidation. Interference does not necessarily lead to invalidation because

the underling locking scheme might prevent the offending interleavings from happening. Furthermore, an assertion that is interfered with can often be replaced by a stronger assertion that is not interfered with. In that case, the weaker assertion is not invalidated.

We are planning to use our theorems to analyze the TPC-C^{tm} Benchmark transactions and run them at a combination of isolation levels to evaluate the performance.

References

[1] ANSI X3.135-1992. American national standard for information systems–database languages–sql, November 1992.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, San Jose, CA, May 1995.

[3] A. J. Bernstein, D. S. Gerstl, W. H. Leung, and P. M. Lewis. Design and performance of an assertional concurrency control system. In *Proceeding of International Conference on Data Engineering*, 1998.

[4] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. A concurrency control for step-decomposed transactions. *Information Systems*, 1999. Accepted for publication.

[5] A. J. Bernstein, D. S. Gerstl, P. M. Lewis, and S. Lu. Using transaction semantics to increase performance. In *the Eighth International Workshop on High Performance Transaction Systems (HPTS)*, Pacific Grove, California, USA, Sept. 1999.

[6] A. J. Bernstein and P. M. Lewis. High performance transaction systems using transaction semantics. *Distributed and Parallel Databases*, 4(1), Jan. 1996.

[7] A. J. Bernstein, P. M. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. Technical report, SUNY at Stony Brook, Stony Brook, 1999. TR 99/22.

[8] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), Nov. 1976.

[9] D. Gries, editor. *The Science Of Programming*. Springer-Verlag New York Inc., 1981.

[10] Hoare, C. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), October 1969.

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

A. Proofs of lemmas

Lemma 1: *Let $S_{i,k} : X := e$ be a local assignment statement of transaction T_i, and $S_{j,h}$ be a write statement of another transaction T_j. Suppose $S_{i,k}$ is characterized by the triple $\{P\}\ S_{i,k}\ \{Q\}$, where $Q \equiv sp(P, S_{i,k})$. If $S_{j,h}$ does not interfere with P, then $S_{j,h}$ does not interfere with Q.*

Proof: The strongest postcondition, Q, of $S_{i,k}$ with precondition P is given by [9]: $Q \equiv \exists v(P_v^X \bigwedge X = e_v^X)$. Since $S_{j,h}$ does not interfere with P, the triple

$$\{P \bigwedge P'\}\ S_{j,h}\ \{P\} \tag{4}$$

is a theorem, where P' is the precondition of $S_{j,h}$. Since X is a local variable of T_i, transaction T_j cannot change its value, it follows from (4) that

$$\{P_v^X \bigwedge P'\}\ S_{j,h}\ \{P_v^X\} \tag{5}$$

where v is an arbitrary value of X. Our goal is to show that

$$\{Q \bigwedge P'\}\ S_{j,h}\ \{Q\} \tag{6}$$

Suppose q is an arbitrary state satisfying $Q \wedge P'$, and q' is the state that results after $S_{j,h}$ is executed starting in q. We would like to show that q' satisfies Q. Let v_0 be a value of X that makes Q true in q. Then $P_{v_0}^X$ is true in q and from (5) $P_{v_0}^X$ is true in q' as well. Furthermore, since X is a local variable and $e_{v_0}^X$ only involves local variables, $X = e_{v_0}^X$ is still true of q'. Hence, Q is true of q'. Since q is an arbitrary state satisfying $Q \wedge P'$, it follows that (6) is a theorem. $\diamond$

Lemma 2: *Let $S_{i,k} : x := X$ be a write statement of transaction T_i, and $S_{j,h} : y := Y$ be a write statement of another transaction T_j. Suppose x and y are two distinct database variables and $S_{i,k}$ is characterized by the triple $\{P\}\ S_{i,k}\ \{Q\}$ where $Q \equiv sp(P, S_{i,k})$. If $S_{j,h}$ does not interfere with P, then $S_{j,h}$ does not interfere with Q.*

Proof: The strongest postcondition, Q, of $S_{i,k}$ with precondition P is given by [9]: $Q \equiv \exists v(P_v^x \bigwedge x = X)$. Since $S_{j,h}$ does not change the values of x and X, we can prove this lemma using the same technique that we used in Lemma 1. $\diamond$

B. Proofs of theorems

Theorem 1 *A transaction, T_i, that executes at READ UNCOMMITTED will execute semantically correctly if each write statement (including those that rollback a transaction) in every transaction does not interfere with I_i, the postcondition of every read statement in T_i, and Q_i.*

Proof: Consider an arbitrary execution path of transaction T_i and label the control points along this path α_1, α_2, ..., α_n. Let P_{α_k} be the assertion associated with α_k in the proof (1) of T_i. In the following, we show that, when each control point, α_k, of the path is active, the state of the system, denoted by $state(\alpha_k)$, satisfies an assertion P'_{α_k} such that: (1) $P'_{\alpha_k} \Longrightarrow P_{\alpha_k}$, and (2) for each write statement, $S_{j,h}$, of transaction T_j, either $S_{j,h}$ does not interfere with P'_{α_k}, or if it does, it will not invalidate P'_{α_k}.
The proof is by induction on k.

1. **Induction basis** $k = 1$: Let $P'_{\alpha_1} \equiv P_{\alpha_1}$. Since α_1 is the first control point $P_{\alpha_1} \equiv P_{i,1}$. By the conditions of the theorem, $P_{i,1}$ is not interfered with by $S_{j,h}$.

2. **Induction hypothesis**: For all control points α_i in the execution path $\alpha_1 \cdots \alpha_m$, $state(\alpha_i)$ satisfies an assertion P'_{α_i} that satisfies (1) and (2).

3. **Induction step**: We need to exhibit an assertion, $P'_{\alpha_{m+1}}$, satisfying (1) and (2). Consider all possible control point transitions from α_m to α_{m+1}:

   (a) T_i executes a read statement. By the conditions of the theorem, $P_{\alpha_{m+1}}$ is not interfered with by $S_{j,h}$.

   (b) T_i executes a write statement $stmt$, that writes the same database item as $S_{j,h}$. Let $P'_{\alpha_{m+1}} \equiv sp(P'_{\alpha_m}, stmt)$. Since $P'_{\alpha_m} \Longrightarrow P_{\alpha_m}$, it follows that $P'_{\alpha_{m+1}} \Longrightarrow P_{\alpha_{m+1}}$. Furthermore, $P'_{\alpha_{m+1}}$ cannot be invalidated by $S_{j,h}$ because if $stmt$ has executed, T_i has acquired a write lock on the data item written by $stmt$. Thus, $S_{j,h}$ cannot be executed until T_i terminates. (Note in this case, it is possible that $S_{j,h}$ interferes with $P'_{\alpha_{m+1}}$.)

   (c) T_i executes a write statement, $stmt$, that writes a database item that is distinct from the item written by $S_{j,h}$. Let $P'_{\alpha_{m+1}} \equiv sp(P'_{\alpha_m}, stmt)$. Since $P'_{\alpha_m} \Longrightarrow P_{\alpha_m}$, it follows that $P'_{\alpha_{m+1}} \Longrightarrow P_{\alpha_{m+1}}$. Furthermore, since P'_{α_m} is not interfered with by $S_{j,h}$, it follows from Lemma 2 that $P'_{\alpha_{m+1}}$ is not interfered with by $S_{j,h}$.

   (d) T_i executes a local assignment, $stmt$. Let $P'_{\alpha_{m+1}} = sp(P'_{\alpha_m}, stmt)$. Since $P'_{\alpha_m} \Longrightarrow P_{\alpha_m}$, it follows that $P'_{\alpha_{m+1}} \Longrightarrow P_{\alpha_{m+1}}$. Since P'_{α_m} is not interfered with by $S_{j,h}$, it follows from Lemma 1 that $P'_{\alpha_{m+1}}$ is not interfered with by $S_{j,h}$.

   (e) T_i enters the $THEN$ body of a conditional statement with guard G. Let $P'_{\alpha_{m+1}} \equiv P'_{\alpha_m} \wedge G$. Since $P'_{\alpha_m} \Longrightarrow P_{\alpha_m}$, it follows that $(P'_{\alpha_m} \wedge G) \Longrightarrow (P_{\alpha_m} \wedge G)$. Since P'_{α_m} is not interfered with by $S_{j,h}$ and G only involves local variables, $(P'_{\alpha_m} \wedge G)$, is not interfered with by $S_{j,h}$.

   (f) T_i enters the $ELSE$ body of a conditional statement with guard G. Let $P'_{\alpha_{m+1}} = P'_{\alpha_m} \wedge \neg G$. The argument is the same as the previous case.

   (g) T_i enters (or re-enters) the body of a while loop with guard G. Let $P'_{\alpha_{m+1}} = P'_{\alpha_m} \wedge G$. Since $P'_{\alpha_m} \Longrightarrow P_{\alpha_m}$, it follows that $(P'_{\alpha_m} \wedge G) \Longrightarrow (P_{\alpha_m} \wedge G)$, i.e., $P'_{\alpha_{m+1}} \Longrightarrow P_{\alpha_{m+1}}$. Since P'_{α_m} is not interfered with by $S_{j,h}$ and G only involves local variables, $P'_{\alpha_{m+1}}$ is not interfered with by $S_{j,h}$.

   (h) T_i exits from a $while$ loop with guard G. Let $P'_{\alpha_{m+1}} = P'_{\alpha_m} \wedge \neg G$. The argument is the same as the previous case..

Thus when T_i commits, Q'_i will be true of the final state where $Q'_i \Longrightarrow (I_i \wedge Q_i)$. As one of the conditions of the theorem, $I_i \wedge Q_i$ is not interfered with by $S_{j,h}$. Hence none of the assertions of T_i will be invalidated. Since the proof is done for an arbitrary execution of T_i and an arbitrary write statement of T_j, the semantic correctness of T_i is guaranteed. $\diamond$

Theorem 2 *A transaction, T_i, executed at READ COMMITTED will execute semantically correctly if each transaction does not interfere with the postcondition of every read statement in T_i, and with Q_i.*

Proof: Since the isolation level of T_j is at least READ UNCOMMITTED, T_j will hold a long term write lock on any item it writes. Since at the READ COMMITTED level, T_i uses short term read locks, T_i cannot read any item written by T_j until T_j terminates. Thus, when we reason about the semantic correctness of a schedule that includes T_i, as far as T_i is concerned, T_j can be considered a single isolated unit. Using the same reasoning as employed in the proof of Theorem 1, we can prove that no assertion of T_i can be invalidated by T_j. Note that since T_i preserves I, the only conjunct of the postcondition of T_i that can be interfered with by T_j is Q_i. $\diamond$

Theorem 3 *A transaction, T_i, executed at READ COMMITTED with first-committer-wins will execute semantically correctly if each transaction does not interfere with the postconditions of those read statements in T_i that are not followed by a write statement on the same item, and with Q_i.*

Proof: Consider any (annotated) read statement, $\{P\}\ X := x\ \{Q\}$, in T_i which has a following write statement on x. Then $sp(P,\ X := x) \equiv \exists v(P^X_v) \bigwedge (X = x)$ Assuming P is not interfered with by T_j, then, using Lemma 1, only the second conjunct, $X = x$, can be invalidated. Since T_i commits, no concurrent transaction writes x and the above postcondition is not invalidated. With this observation in mind, the proof follows in a manner similar to the proof for the READ COMMITTED isolation level. $\diamond$