Formal Model of Correctness Without Serializability*

Henry F. Korth, Gregory D. Speegle

Department of Computer Sciences
University of Texas
Austin, TX 78712-1188

Abstract
In the classical approach to transaction processing, a concurrent execution is considered to be correct if it is equivalent to a non-concurrent schedule. This notion of correctness is called *serializability*. Serializability has proven to be a highly useful concept for transaction systems for data-processing style applications. Recent interest in applying database concepts to applications in computer-aided design, office information systems, etc. has resulted in transactions of relatively long duration. For such transactions, there are serious consequences to requiring serializability as the notion of correctness. Specifically, such systems either impose long-duration waits or require the abortion of long transactions. In this paper, we define a transaction model that allows for several alternative notions of correctness without the requirement of serializability. After introducing the model, we investigate classes of schedules for transactions. We show that these classes are richer than analogous classes under the classical model. Finally, we show the potential practicality of our model by describing protocols that permit a transaction manager to allow correct non-serializable executions.

1. Introduction

The classical approach to the theory of database concurrency control is based on an uninterpreted consistency constraint. Transactions are required to map consistent states of the database to consistent states. A concurrent execution of a set of transactions (a schedule) is correct if it is equivalent to a serial (non-concurrent) execution. This notion of correctness is called *serializability*. Since no explicit use is made of the database consistency constraint except for the assumption that transactions preserve consistency, serializability is necessary if one is to prove that a schedule preserves consistency. The formal theory of serializability is well-developed and appears in [Bernstein et al. 1987, Papadimitriou 1986] as well as elsewhere.

The class of serializable schedules is too rich for a practical transaction management systems for several reasons including the following:

- testing for serializability is NP-complete [Papadimitriou 1979]
- included among the serializable schedules are schedules that present several obstacles to crash recovery (allowance of cascading rollbacks and non-recoverable schedules).

On the other hand, the class of serializable schedules turns out to be too restrictive for long duration transactions. This follows from a theorem of Yannakakis[1982] which implies that if transaction systems have no special structure then two phase locking is necessary to ensure serializability, unless non-lock-based techniques are used. Two phase locking imposes the constraint that a transaction may not issue a lock request after its first unlock request. Thus locks must be held in general for a substantial fraction of the duration of a transaction. For long-duration transactions, this leads to long duration waiting for locks. Alternative techniques for ensuring serializability use transaction aborts. However, aborts of long duration transactions are highly undesirable since large amounts of work done by users can be lost.

Researchers and developers of database systems for CAD, office information systems, software development environments, etc. have dealt with this problem by implementing ad-hoc concurrency control mechanisms modeled after human behavior in collaborative projects. These methods [Kim et al. 1984, Lorie and Plouffe 1983] provide tools to assist users in minimizing the adverse effects of concurrency. However, they do not allow a complete mathematical characterization of correctness. Although schemes of this sort have met the needs of several practical systems, a formal notion of correct schedules that meets the requirements of CAD and CAD-like applications is desirable. It is exactly such a model that we propose in this paper.

2. An Introduction to the Model

Serializability requires that the schedule of operations performed by concurrently executing transactions is equivalent to a serial execution of these transactions [Eswaran et al. 1976]. Serializable schedules have many qualities which are "good" in some sense. If all of the transactions preserve the consistency of the database, then serializable schedules also keep the database consistent. Users of systems which enforce serializability know their tasks will see a consistent database and their tasks will execute such that if the tasks finish, they will have been unaffected by other concurrent transactions. This also means the amount of help one user can give another, as in the case of cooperating transactions where two designers work together to complete a project, is very limited. Although this is acceptable in traditional database systems, long duration transactions need to be able to interact in order to complete their tasks. Therefore, the goal of a long duration transaction system should be to allow schedules which permit interaction among transactions while still maintaining consistency.

Our model includes three features for enhancing concurrency that are not part of the traditional

* Research partially supported by NSF grant DCR-850-7224, and a grant from the IBM Corporation

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and / or specific permission.

© 1988 ACM 0-89791-268-3/88/0006/0379 $1.50

model:

- versions
- nested transactions
- explicit consistency predicates

We introduce each of these features informally in this section and define our model formally in the next section.

2.1 Multiple Versions

One method for improving concurrency is the use of multiple versions for concurrency control. Whenever a transaction attempts to perform a read operation, an algorithm called the "version function" assigns one of the values of the data item to the transaction. Whenever a transaction attempts to write a data item, the system creates a new version of the data item with the new value. The set of multiversion serializable schedules is a proper superset of the set of serializable schedules [Papadimitriou and Kanellakis 1982]. Since versions must be supported in a design environment anyway, it is desirable to take advantage of them to enhance concurrency.

2.2 Nested Transactions

A second technique for improvements in modeling long duration transaction systems is nested transactions [Moss 1985, Beeri et al. 1986, Fekete et al. 1987, Lynch 1986, Liskov et al. 1987, Moss et al. 1986, Bancilhon et al. 1985, Korth et al. 1988, Weikum and Schek 1984]. With nested transactions, some transactions contain only primitive operations such as reads and writes, but others contain complex operations which contain multiple primitive operations themselves. These complex operations can be viewed as subtransactions to the transaction which contains it. This subtransaction becomes a child of its creating transaction. If we extend this concept, we allow subtransactions to contain subtransactions of their own, thereby creating trees of transactions. Therefore, nested transactions are frequently represented as a tree.

Nested transactions provide a mechanism to allow needed interaction for long duration transactions [Bancilhon et al. 1985]. The subtransactions created by the root, called top-level transactions, are distinct design operations which require only minimal interaction, but which maintain database consistency. The tree structure of nested transactions allows designers to operate in a hierarchy which resembles the design process. A user can request work from another user by creating a new subtransaction for that user. That work then becomes part of the requesting user's transaction, so it can be easily incorporated into the user's design. Finally, greater concurrency can be achieved with nested transactions by allowing subtransactions to execute in parallel and by allowing schedules which are non-serializable at one level but are equivalent to some serial schedule at a higher level [Beeri et al. 1986].

2.3 Explicit Consistency Predicates

The third additional element of this model is explicit semantics of database systems. Such semantics can increase the concurrency of the system by redefining the notion of conflict. It must be assumed that any two operations conflict if there is no information indicating otherwise. The most common example of using semantics is defining accesses to be either a read or a write of a data item, but other examples can be found in [Korth 1983]. Our goal in this paper is the use of explicit consistency predicates for increasing concurrency.

An earlier use of consistency predicates is [Bancilhon et at 1985], whose model defines an invariant for a transaction such that if the transaction operates correctly, then the consistency constraint for the entire database is still correct. We generalize that notion from an invariant to a precondition and a postcondition. The precondition defines a database state which is needed for the transaction to execute correctly, while the postcondition describes the state of the database after the transaction has executed, assuming the transaction is run by itself.

3. Formal Presentation of the Model

Our formal model is based on the concept of representing transactions as mappings from one database state to another, with schedules becoming compositions of mappings. This allows us to capture additional semantics and, by appropriate restrictions, to represent other transaction models as well. Unfortunately, this technique emphasizes concurrency control over recovery. Our model has a notion of recoverability, but for now, we focus on concurrency issues.

Let E denote the set of all entities in the database, and $\forall e \in E$, let $\mathbf{dom}(e)$ denote the domain of entity e. In the standard database model, each entity is assigned one value from its domain. The collection of all such values determines the state of the database.

Definition : A *unique state*, S^U, is a one-to-one mapping with a domain E and range $\bigcup_{e \in E} \mathrm{dom}(e)$ such that $\forall e, S^U(e) \in \mathbf{dom}(e)$.

Transactions in the standard model change the database from one unique state to another. Thus, transactions can be modeled as functions on unique states. Let D^U represent the set of all unique states.

Definition : A standard transaction is a mapping from D^U to D^U.

Although this definition of a transaction is very general, it does require some constraints on the actions performed by a transaction. For example, a transaction cannot update an entity to an element not in the domain of the entity, as there can be no unique state S^U which can perform that mapping. Note also that this definition of a transaction does not require the preservation of any kind of consistency constraint. The notion of a database consistency constraint is useful, but before it can be defined, the concept of a predicate on a set of unique states must be established.

Definition : If P is a predicate on unique states, we extend P to a set of unique states, denoted $P(S)$, where $P(S) = \{S^U | S^U \in S \wedge P(S^U)\}$. Clearly, $P(S) \subseteq D^U$.

We shall assume all of our predicates are formed from atoms and clauses. An *atom* is a comparison $x \theta y$, where θ is a comparison operator, and x and y are either entities or constants. A disjunctive-clause is a disjunction (or) of atoms. We consider only predicates in conjunctive normal form, that is a conjunction of disjunctive clauses.

Definition : A predicate P is in *conjunctive normal form* if $P = \bigwedge_{i=0}^{n-1} C_i$ where each $C_i = \bigvee_{j=0}^{m-1} L_j$ where each L_j is an atom.

It is easy to show that all predicates can be expressed in conjunctive normal form. We now introduce a term for the sets of data items which appear in disjunctive clauses of a predicate.

Definition : Let $P = \bigwedge_{i=0}^{n-1} C_i$. Let x_i denote the set of data items mentioned in an atom in C_i. Each such x_i is an *object*. The set of all objects in a predicate, $\{x_0, x_1, \ldots, x_{n-1}\}$ is denoted by $\hat{P}$.

We represent database consistency constraints as predicates on database states and define what it means for a transaction to maintain the database consistency constraint.

Definition : If C is the database consistency constraint, then a standard transaction *maintains consistency* if it is a mapping from $C(D^U)$ to $C(D^U)$.

Thus a standard transaction can be thought of as a program t, which guarantees that if $C(S^U)$ holds when the transaction begins then $C(t(S^U))$ holds when the transaction terminates. This is the basic assumption of standard database models. Transactions executing in a serial order maintain C.

We represent multiple versions by allowing a set of unique states to form a database state.

Definition : A *database state* S is a set of unique states S_i^U. The set of all database states S is D.

Although this captures all possible versions of a given data item, it does not represent all possible combinations of versions which a transaction might access. That is represented by the version state which is associated with each database state.

Definition : The *version state* of the database is the set of all versions which can be generated from a database state S, and is denoted V_S, where $V_S = \{f : E \rightarrow \bigcup_{e \in E} \mathbf{dom}(e) | \forall e(f(e) \in \mathbf{dom}\ (e) \wedge \exists g \in S\ (g(e) = f(e)))\}$. V denotes the set of all possible version states, that is, V_D.

V_S is a collection of value assignments to database entities such that some unique state in S makes the same assignment. However, the assignments made to a version state can be drawn from different unique states. Note that if $v \in V_S$, then $v(e)$ returns a value of some version of e, and that all v satisfy the definition of a unique state. Note also that if $|S| = 1$ and $S^U \in S$, then $V_S = \{S^U\}$. The presence of versions changes the definition of a transaction.

Definition : A *transaction*, t, is a mapping from D to D^U, such that $\exists v \in V_S$ such that $t(\{v\}) = t(S)$. The *result* of a transaction, t, applied to state S is the state $S \cup t(S)$.

In other words, transactions are equivalent to a mapping from a version state to a unique state.

The second concept added to the standard model concerns pre- and post- conditions of transactions. A transaction can be viewed as a program which will leave the database in a certain state given that it is not interleaved with other transactions, and if the database is in a certain state when it begins. These predicates are the *specification* of the transaction.

Definition : A *specification* for a transaction t is a pair (I_t, O_t) where I_t and O_t are predicates on D. Every entity read by t must appear in I_t. (I_t, O_t) is satisfied by t if $\forall S \in I_t(D), t(S) \in O_t(D)$.

The third extension involves nested transactions. Nested transactions can be thought of as a lower-level implementation of their parent.

Definition : An *implementation* of a transaction t is a pair (T, P) where T is a set of transactions or operations and P is a partial order on T.

Thus for all $t_i \in T$, t is the parent of t_i. Any part of t which cannot be divided into subtransactions is a basic operation of the system. Basic operations are usually read and write, but can include other operations such as increment and decrement [Korth 1983] or complex design update [Kim et al. 1984]. Frequently, we shall give both a specification (I_t, O_t) and an implementation (T, P) for t in a four-tuple (T, P, I_t, O_t).

We define three sets of data items related to a transaction: the input set, the update set and the fixed-point set. The fixed-point set is simply the set of all data items which the transaction does not update. We also define the object set of a transaction, which is based on the specification of a transaction.

Definition : Let t be (T, P, I_t, O_t). The ***input set***, N_t is the set of data items in appearing in I_t. The *fixed-point set*, $F_t = \{e | e \in E \wedge \forall S^U \in D^U, S^U(e) = (t(\{S^U\}))(e)\}$. The ***update set***, $U_t = E - F_t$. The *object set*, $\hat{t} = \bigcup_{i=0}^{n-1} (\hat{O}_{t_i})$ where $\{t_0, t_1, \ldots, t_{n-1}\}$ are the subtransactions of t.

It is now possible to define an execution of a transaction. Such an execution must include a relation on the subtransactions which is consistent with P, the partial order. Although the semantics for including a relationship between subtransactions are not yet defined, it may be helpful intuitively to think of this relation as representing a "reads from" graph. Also needed in the execution is some notion of the state of the database before a transaction begins to execute. This is required to check that transactions fulfill their specifications.

Definition : An *execution* of a transaction t, where $t = (T, P, I_t, O_t)$, is a pair (R, X) where $R \subseteq T \times T$ is a relation on T such that $(t_i, t_j) \in P^+ \Rightarrow (t_j, t_i) \notin R^+$, where P^+ and R^+ are the transitive closure of P and R respectively, and X is a mapping from T to a version state $v \in V_S$. If $t_k \in T$, then $X(t_k)$ is called the ***input state*** of t_k.

This definition places no restrictions on X and only a proscriptive constraint on R. We add semantics to our initial definition of execution by requiring that R encode "dependencies" among the $X(t_i)$ and that each state $X(t_i)$ "depend" upon $X(t)$, the input state of the parent transaction.

Definition : A *parent-based* execution (R, X) of $t = (T, P, I_t, O_t)$ is an execution such that for each $t_i \in T$, with an execution $(R_i, X_i), \forall e \in E$, either

- $(X_i(t_i))(e) = (X(t))(e)$
- $\exists t_j \in T$, with execution (R_j, X_j) such that $(t_j, t_i) \in R$ and $(X_i(t_i))(e) = (t_j(X_j(t_j)))(e)$.

Note that (t_i, t_j) or (t_j, t_i) is not required for R^+ even if $N_{t_i} \cap U_{t_j} \neq \emptyset$. This allows for independent executions which can happen in multiple version systems. The final state of an execution can be defined as the state of the database after every transaction has executed. By using a pseudo-transaction, t_f, the final state can be defined as follows:

Definition : The *final state* of an execution is $X(t_f)$ where $\forall t_i \in T, (t_i, t_f) \in R^+ \wedge E = N_{t_f}$.

Likewise, we can define a pseudo-transaction t_0 which creates the initial state of the database. However, the initial state should only apply to the root transaction, while the final state can apply to any transaction.

Definition : The *initial state* of an execution can be denoted as $t_0(S)$ where $\forall t_i \in T, (t_0, t_i) \in R^+ \wedge E = U_{t_0} \wedge$ parent(parent(t_0)) =nil.

Correctness can now be defined for executions of transactions.

Definition : An execution (R, X) of a transaction $t = (T, P, I_t, O_t)$ is *correct* if $\forall t_i \in T, I_{t_i}(X(t_i)) \wedge O_t(X(t_f))$.

In other words an execution is correct if every subtransaction can access a database state which satisfies its input condition and the result of all of the subtransactions satisfies the output condition of the transaction. We can extend this notion of correctness to both the ancestors and descendants of a given transaction, thus producing multi-level correctness criteria. More importantly, this correctness criteria can be applied to the root transaction, thus ensuring that the entire database system executes correctly.

It is not hard to show that determining if a given execution is correct is an NP-complete problem. This is analogous to determining if a given schedule is serializable in traditional database models, which is also NP-complete [Papadimitriou 1979]. This motivates us to consider in Section 4 subsets of the set of correct executions that have efficient protocols, yet offer a high degree of concurrency.

4. Correctness Classes

Classical concurrency control theory has developed a number of classes of correct executions for transaction systems. These classes include view serializability, conflict serializability, multiversion serializability and others [Bernstein et al. 1987, Papadimitriou 1986]. Since these classes are too restrictive for use in long duration transaction systems, we present broader correctness classes. The broadest class we propose, is the set of all schedules which are correct executions. A protocol which allows a subset of such schedules is presented in the next section. This class contains a large number of schedules, including the set of all view serializable schedules.

There are obviously many schedules which are correct executions, but which are not view serializable, since the set of correct executions allows multiple versions and partial orders, which are not part of the standard model. We can apply restrictions to the set of correct executions so that we allow only standard-model, view serializable executions.

4.1 View Serializability

The standard model consists of a root (T, P, I, O) where $T = \{t_0, t_1, \ldots, t_n, t_f\}$, P is the empty order and both I and O are the database consistency constraint C. For each $t_j \in T$, $t_j = (a_j, p_j, i_j, o_j)$ where $a_j \subseteq \{read, write\} \times E$, p_j is a total order on a_j, and both i_j and o_j are the database consistency constraint C. Note that a_j represents the smallest operations in the standard model, and therefore cannot be broken down into subtransactions. Note also that t_f is as we defined earlier. The database is also restricted such that the database state is always a unique state, so $|S| = 1$. This is accomplished in the standard model by having every write operation overwrite the previous value of the entity.

In order to formalize the concept of serializability in our model, some representation of the total order of the transactions is needed. This total order must be consistent with the partial order defined in R, and is the basis for the reads performed by a transaction. In what follows, this order is denoted by the function f.

An execution (R, X) is view serializable if all of the following hold:

1) the database system conforms to the standard model;
2) $\forall t_i \in T, \exists s \neq t_i$ such that $(t_i, s) \in R$ and $\exists r \neq t_i$ such that $(r, t_i) \in R$;
3) $\exists$ a 1-1, onto mapping $f : T \rightarrow \{0, \ldots, |T| - 1\}$ such that $f(t_i) < f(t_j) \Rightarrow (t_j, t_i) \notin R$;
4) $f(t_i) = f(t_j) + 1 \Rightarrow X(t_i) = t_j(X(t_j))$.

Such an execution is view equivalent to a serial schedule executing in increasing value of f:

- Both schedules must contain the same transactions - true by part 2.
- Each transaction reads the same values in both schedules - true by parts 3 and 4.
- The database is in the final state after both schedules - since the final state is $X(t_f)$, true by parts 3 and 4.

Executions with these properties are view serializable since any interleaving of operations which satisfies these properties is acceptable. For example, the database never has to be the state $t_j(X(t_j))$ so long as $X(t_i)$ can see the equivalent of that state. Thus, updates can be performed on data items not in N_{t_i} and $X(t_i)$ can still see $t_j(X(t_j))$.

4.2 Extensions to View Serializability

Although view serializable schedules are a basis for correctness in standard database systems, other classes also exist. These classes are derived by adding semantics, whereby more schedules can be considered correct by the concurrency control algorithms.

Multiple Versions

Multiple versions are very easy to represent in our model. Everything remains the same except that we relax the requirement $|S| = 1$. Obviously, this new class, MVSR, allows more schedules than SR, since we can restrict MVSR to SR by the previous condition on S, and the following schedule is not allowed in SR, but is in MVSR:

```
t1:  R(x)W(x)                    R(y)W(y)
t2:              R(x)R(y)W(y)
```

Example 1

Intuitively, this schedule is not equivalent to t_1, t_2 since t_1 reads y from t_2 and it is not equivalent to t_2, t_1 since t_2 reads x from t_1. However, X maps the version state $v = t_0(S)$ to t_2 and the version state $v = t_2(X(t_2))$ to t_1, thus allowing multi-version serializability.

Partial Order Serializability

Partial order serializability, $\prec$SR, results from allowing the operations of transactions to happen in a partial order instead of a total order. A scenario can exist where an item required by a transaction is locked, thus causing a standard transaction to wait. However, if partial orders are used, the transaction can access a different, available data item.

Also, partial order serializability enables us to extend the notion of serializability to multiple levels. Top-level transactions must remain serializable, but lower-level operations must execute consistent with the partial order of its parent. When these lower level operations are actually transactions, non-serializable schedules can be generated. This is similar to the work of [Beeri et al. 1986]. Partial order serializability is represented in our model by changing the standard model as follow:

The model consists of a root (T, P, I, O) where $T = \{t_0, t_1, \ldots, t_f\}$, P is the empty order and both I and O are the database consistency constraint C. For each $t_j \in T$, $t_j = (a_j, p_j, i_j, o_j)$ where $a_j \subseteq \{read, write\} \times E$ or $a_j = \{t_{j_0}, t_{j_1}, \ldots, t_{j_f}\}$ where each t_{j_k} is defined similarly to t_j, p_j is a partial order on a_j, and both i_j and o_j are the database consistency constraint C.

Predicatewise Serializability

Predicatewise serializability, PWSR, is derived from a protocol called predicatewise two-phase locking, presented in [Korth et al. 1988]. The basic idea is that if the database consistency constraint is in conjunctive normal form, we can maintain the consistency constraint by enforcing serializability only with respect to data items which share a disjunctive clause. The increased concurrency for this class is derived from the fact the serializable schedules for each disjunctive clause do not have to agree.

Predicatewise serializable schedules can be represented in this model as follows.

The database consistency constraint C, is written as a predicate P, by our definition of a predicate. The standard model is used to represent the database. Then, $\forall x_i \in \hat{P}$ where x_i is an object, we use the following definitions.

Definition : For a transaction $t = (T, P, I_t, O_t)$, the set of subtransactions which mention x_i is $T^{x_i} = \{t_i | t_i \in T \wedge x_i \cap (N_{t_i} \cup U_{t_i}) \neq \emptyset\}$.

Definition : For an execution (R, X) of a transaction t, the restriction of a partial order R by an object x_i is $R^{x_i} = \{(r, s) | (r, s) \in R^+ \wedge r \in T^{x_i} \wedge s \in T^{x_i}\}$.

Definition : An execution (R, X) is predicatewise serializable if

1: the database system conforms to the standard model;

2: $\forall t_i \in T, \exists s \neq t_i$ such that $(t_i, s) \in R$ and $\exists r \neq t_i$ such that $(r, t_i) \in R$;

3: $\forall x_i \in \hat{P}, \exists$ a 1-1, onto mapping $f : T^{x_i} \rightarrow \{0, \ldots, |T^{x_i}| - 1\}$ such that $f(t_i) < f(t_j) \Rightarrow (t_j, t_i) \notin R^{x_i}$;

4: $f(t_i) = f(t_j) + 1 \Rightarrow \forall e \in x_i, (X(t_i))(e) = (t_j(X(t_j)))(e)$.

We define the class $PWSR_C$ as the schedules allowed under PWSR for a given predicate C. Clearly, for all predicates, any schedule which is in SR is in $PWSR_C$, since the projection of a serializable schedule on to a subset of the entities in the database is serializable. Likewise, there exist conjuncts such that a schedule is in $PWSR_C$, but not in SR. For example, assume a two item database where one conjunct contains atoms which are only over the data item x, and another conjunct contains atoms which are only over the data item y. Then the following schedule is in $PWSR_C$, but not in SR.

```
t1:  R(x)W(x)                    R(y)W(y)
t2:            R(x)R(y)W(y)
```

Example 2.

This is the same schedule that we showed in example 1 not to be in SR. $PWSR_C$ can decompose this schedule into the two following schedules based on the conjuncts presented here.

```
t1:  R(x)W(x)
t2:            R(x)
```

Example 3.a

```
t1:                        R(y)W(y)
t2:            R(y)W(y)
```

Example 3.b

Both of these are clearly serializable, since they are in fact, serial schedules.

Predicate Correct

Predicate correct, denoted PC, is the class obtained by combining all of the extensions to serializability presented in this section. A system which is predicatewise correct allows multiple versions of data items, multiple levels with partial orders, and predicatewise serializability for its correctness criteria. This class is very broad, encompassing all of the others, yet it still represents a restriction of correct executions.

Definition : An execution (R, X) is predicatewise correct if

1: the database system conforms to the model of Section 4;

2: $\forall t_i \in T, \exists s \neq t_i$ such that $(t_i, s) \in R$ and $\exists r \neq t_i$ such that $(r, t_i) \in R$;

3: $\forall x_i \in \hat{P}, \exists$ a 1-1, onto mapping $f : T^{x_i} \rightarrow \{0, \ldots, |T^{x_i}| - 1\}$ such that $f(t_i) < f(t_j) \Rightarrow (t_j, t_i) \notin R^{x_i}$;

4: $f(t_i) = f(t_j) + 1 \Rightarrow \forall e \in x_i, (X(t_i))(e) = (t_j(X(t_j)))(e)$.

Due to the partial order, schedules can be predicatewise correct but not in PWSR or MVSR. Likewise, the use of multiple versions increases the choices available to transactions and the use of predicates reduces the conflicts between transactions to allow schedules not in $\prec$SR.

The unfortunate consequence of predicatewise correct, is that determining if a schedule is in PC is NP-complete. It is obvious that determining if a schedule is view serializable is a subproblem to this one, with the reductions applied to this class being the expansions used to gain MVSR, ≺SR, and PWSR. However, just as SR can be reduced to an efficient class, conflict serializability, we can reduce PC to an efficient class, conflict predicate correct.

4.3 Conflict Predicate Correct

CPC can be built by using the same extensions which we applied to the class SR by applying them to the class conflict serializability, denoted CSR. A schedule is conflict serializable if it conflict equivalent to any serial schedule. Two schedules are conflict equivalent if their conflicting steps are in the same order. Under the standard model, two steps conflict if they are on the same data item and at least one of them is a write. The class MVCSR is defined as all schedules which can be considered to be conflict serializable given that we can use multiple versions of data items. The only conflicts which exist in MVCSR are reads before writes on the same data item [Papadimitriou 1986]. The class PWCSR simply enforces conflict serializability on each conjunct of the database constraint, and ≺CSR allows conflict serializable schedules but permits operations to be placed in a partial order within a transaction.

If we combine all of the properties of MVCSR, PWCSR, and ≺CSR, we achieve the class CPC. An important property of CPC is that determining if a schedule is CPC can be done efficiently. The only conflicts which still exist in CPC are a read of a data item followed by a write on that data item, just as in MVCSR. However, in CPC, if two data items are in different conjuncts, then the execution order of the transactions does not have to be the same for the transactions. Thus, the techniques in [Papadimitriou 1986] for showing a schedule to be in MVCSR can be repeated for each conjunct. This technique creates a graph where each node is a transaction, and an arc is drawn from A to B if A performs a read step and B performs a write step on the same entity. In this case, each graph corresponds to a single conjunct, and the arc is drawn only if the data item accessed by A and B is in the conjunct. A schedule is MVCSR iff the graph is acyclic, and consequently, a schedule is CPC iff all of the graphs are acyclic. Since testing for acyclicity is efficient for 1 graph, it remains efficient for n graphs, where n is unrelated to the number of nodes or edges, as it is here.

In order to get a better understanding of the classes involved in CPC, Figure 1 presents the relationships between all of the various subclasses. Examples of schedules are given for each non-empty region of the diagram.

1. Non-CPC

```
t1: R(x)      W(x)
t2:      R(x)      W(x)
```

Intuitively, this schedule is not in CPC because none of its decompositions by conjuncts (which are exactly this schedule) can be serialized by a version function. This is because either t_1 should read from t_2 or t_2 should read from t_1 in a serial schedule, and this does not happen here.

2. CPC − (PWCSR∪MVCSR∪ ≺CSR ∪ SR)

```
t1: R(y)      W(x)           W(y)
t2:      R(x)      W(x)W(y)
```

This schedule is not in PWCSR since the decompositions lead to nonserializable schedule. Likewise,

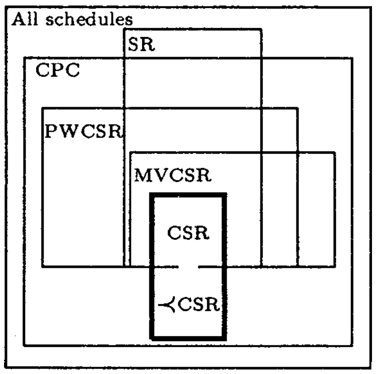

Figure 1: Classes of Schedules

it is not in MVCSR because the read of y done by t_1 conflicts with the write of y done by t_2 and similarly, t_2 conflicts with t_1 on x. However, the schedule could be in CPC if the database consistency constraint placed x and y in different conjuncts. The restricted schedules are in MVCSR.

3. PWCSR − (MVCSR ∪ ≺CSR ∪ SR)

```
t1: R(x)W(x)                R(y)W(y)
t2:         R(x)W(x)R(y)W(y)
```

This schedule is in PWCSR if x and y are in different conjuncts. The schedule is clearly not in SR or MVCSR since either serial schedule would cause a transaction to read a data item from the other transaction, which it did not in this schedule.

4. (PWCSR ∩ MVCSR) − SR

```
t1: R(x)W(x)            R(y)W(y)
t2:         R(x)R(y)W(y)
```

This is the example schedule given in the previous section. The arguments for it being in MVSR and PWSR hold for MVCSR and PWCSR.

5. SR − PWCSR

```
t1: R(x)      W(x)
t2:      W(x)
t3:                W(x)
```

This schedule is equivalent to the serial schedule t_1, t_2, t_3, however, it is not conflict serializable and cannot be decomposed by any non-empty predicate.

6. SR − MVCSR

```
t1: R(x)               W(y)
t2:      W(y)
t3:           R(y)W(x)      W(y)
```

This schedule is view equivalent to the serial schedule t_1, t_2, t_3, but a conflict exists between transactions t_1 and t_3 on both x and y, keeping this schedule out of MVCSR.

7. MVCSR − PWCSR

```
t1: R(x)      W(x)
t2:      W(x)
```

Clearly, this schedule remains unserializable for all non-empty predicates, since t_2 cannot be "moved" to before or after t_1 by flipping operations. However, if the final read is of the version created by t_2, then this schedule is equivalent to the serial schedule t_1, t_2.

8. (SR ∩ MVCSR) − CSR

```
t1: R(x)W(x)          W(y)
t2:         R(x)W(y)
t3:                        W(x)
```

This schedule is multi-version conflict serializable to the schedule t_1, t_2, t_3, by having the final read of y be the version created by t_2. It is also serializable to the schedule t_2, t_1, t_3. However, this schedule is not conflict serializable since no exchanges can move t_2 to either before or after t_1.

9. CSR

```
t1:  R(x) W(x)      R(y) W(y)
t2:            R(x)           R(y) W(y)
```

Note that all conflicts are resolved in the same order for both x and y in this example schedule.

5. Transaction Management

Given a model for long duration transactions and correctness classes for it, the next step is to develop protocols to allow schedules in the classes. We present a protocol which allows only correct executions in this section. Other protocols including virtual timestamps and predicatewise two-phase locking [Korth et al. 1988] have been re-defined to fit within our model, but will be described in a future paper.

Long duration transactions in our protocol can be thought of as consisting of four parts. The first phase, called *transaction definition*, occurs when an active transaction defines a subtransaction. The system obtains the input constraint, output condition, and place in the partial order, of the defined transaction. The second phase is the *transaction validation* phase. This phase is concerned with assigning appropriate versions to the previously defined transaction. The third phase is called the *transaction execution* phase. During this phase, a transaction performs all of its operations. The fourth phase deals with the end of all transactions, commit, abort or whatever else is needed, and is the *transaction termination* phase. Correctness, defined as correct executions, must be maintained throughout all of these phases.

During the transaction validation phase, the system finds versions of data items which satisfy the input constraint of the transaction. For each data item in the input constraint, the transaction places a R_v (read for validation) lock on the data item in order to protect the transaction from updates performed by other transactions during this phase. The system then performs a two-part process in order to assign correct versions to the transaction. The first part determines the set of versions for each data item which can be read without causing partial order invalidation. To do this, a set of transactions, D_i is associated with each data item d_i in the input constraint of the transaction being validated, for simplicity, called t_i. A transaction t_j is in D_i if parent(t_i) =parent(t_j) *unless:*

1. $(t_i, t_j) \in P^+$, or
2. $d_i \notin U_{t_j}$, or
3. $\exists t_k$ such that $(d_i \in U_{t_k}) \wedge (\{(t_j, t_k), (t_k, t_i)\} \subseteq P^+)$.

Basically, every sibling is considered to be in the set unless the transaction is a successor to the transaction being verified, it does not write the data item which corresponds to the set, or there exists another transaction which comes between this transaction and the transaction being verified and that transaction writes a version of the data item. Note that transactions which might yet write the data item are not considered in the set D_i. By not considering the transactions which might later write a version of the data item, the protocol is making the optimistic assumption that such transactions will not write a new version which the transaction must read.

After the sets of transactions have been determined, each element in the set is checked to see if it is a predecessor of the transaction being verified. If one of the transactions is a predecessor, then the rest of the transactions are removed from the set, and the version written by the predecessor is the only one allowed to the transaction. Otherwise, any of the versions written by any member of the set, or the version assigned to the parent, can be assigned to the transaction.

Once the set of allowable versions has been determined for every entity in the input set of the transaction, the system must select a single version for each entity such that the input constraint is satisfied. Since multiple versions exist for each entity, an exhaustive search of all possible combinations would take exponential time. Instead, a heuristic based scheme should be used for selecting versions. Even if substantial effort is expended in version selection, the avoidance of one long duration wait is likely to justify this overhead.

During the transaction execution phase, the transactions issue read and write requests which the concurrency control protocol augments to ensure safe concurrent execution. A read request requires upgrading a R_v-lock to a R-lock. If the transaction does not have a R_v-lock on the data item, then the read is rejected. The lock compatibility matrix is then consulted for granting R-lock requests, so a transaction can either be granted the lock, or temporarily blocked on some writing transaction. After the lock is granted, the transaction can read from the version assigned to it. A write request from a transaction does not require a transaction to hold a read lock, and therefore, can never fail (although it is possible for a transaction to abort due to its read lock on a data item it is writing). As soon as the write is completed, the write lock is released. Once the write lock is released, all read-lock requests which were blocked by the write are allowed to continue into the re-evaluation process. A write creates a new version of the data item, which is immediately available to all siblings of the writing transaction.

We can now construct a lock compatibility matrix for this model. Note that all locks are placed on the entity, not on a version of the entity. Let W represent a write lock request, R_v represent a read for validation lock, R represent a read lock.

			held	
		R_v	R	W
	R_v	true	true	false
requested	R	true	true	false
	W	re-eval	re-eval	true

Figure 2. Lock Compatibility Matrix

A "true" entry in the table means the lock can be granted. As expected in multiple version systems, this result occurs except when a read operation conflicts with a write. A "false" entry in the table means the lock cannot be granted and the transaction becomes blocked on this data item. The blocking time involved here is small, because write locks are held only for the duration of the write operation, not for the duration of the entire transaction. Once a transaction becomes unblocked on a data item, the re-evaluation routine is called, as if the matrix result had been "re-eval". A "re-eval" result on the table means the transaction should be interrupted and its input constraint should be re-evaluated based on the new version written by one of its predecessors.

The purpose of the re-evaluation procedure is to correct problems which might occur due to the optimistic nature of our protocol. The re-evaluation procedure checks to see if a transaction with a read lock on an entity should have read the version most recently created instead of the one previously assigned to it. Although a transaction holding a read lock will have to be aborted, the protocol can try to salvage a transaction which holds a R_v-lock by changing the versions which have been assigned to it. This salvaging occurs in a procedure called re-assign, which may change any version assignment as long as the transaction has not read the data item. The goal of this procedure is to re-establish both the partial order and the input constraint, and is similar to actions taken during the validation phase.

During the transaction termination phase, the concurrency control protocol operates just as if the transaction were still in the execution phase. Other protocols for handling commits and aborts interact with the transaction at this point, but until the transaction commits, it can be aborted by the concurrency control protocol. However, any such abort will be the result of another transaction, as a transaction in the termination phase cannot perform any operations other than a commit or an abort.

To show that all executions legal under this protocol are correct executions, we provide the following proof sketch. A legal schedule under the protocol is correct if for the transaction $t = (T, P, I_t, O_t)$, where parent(t)=nil, $\exists\, (R, X)$, such that $\forall t_i \in T, I_{t_i}(X(t_i)) \wedge O_t(X(t_f))$. Initially, the input constraint condition is preserved by the version assignment function. No transaction is allowed to execute unless its input constraint is satisfied. If a transaction is assigned new versions during the re-evaluation process, then it must also wait until the input constraint is satisfied. The output condition requirement is met by the termination phase of the protocol, since no transaction can commit if its output condition is not satisfied.

6. Conclusions

We have presented an extension to the classical transaction model design to support the requirements of long-duration, interactive transactions. By showing how to represent both the classical model and the model of Bancilhon, et al. [1985] using the notation of our model, we demonstrated that our model is compatible with existing transaction theory. The features we have added to the classical model include pre- and post- conditions that describe transaction behavior to the transaction manager, a partial interpretation of these conditions based on the notion of *objects* that allows for efficient protocols, direct support for multiple versions, and the integration of our predicate-based notions of correctness with the nested transaction theory of Moss [1985] and Beeri, et al. [1986].

Our model allows a much richer class of schedules than the classical model. We defined classes of schedules in our model and compared these classes with those that exist under the classical model.

To show that our model has potential as a practical scheme for transaction management, we gave a new protocol that imposes moderate overhead but offers a high degree of potential concurrency for transactions in a cooperative processing environment as might exist in CAD.

References:

Bancilhon, F., Kim, W., Korth, H., "A Model of CAD Transactions," *Proc. 11 Conf. on Very Large Databases*, 1985.

Beeri, C., Bernstein, P.A., Goodman, N., "A Model for Concurrency in Nested Transaction Systems," Report TR-86-03, Wang Institute, 1986.

Bernstein, P.A., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

Eswaran, K., Gray, J., Lorie, R., Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System," *C. ACM*, 19:11, November 1976 pp. 624-633.

Fekete, A., Lynch, N., Merritt, M., Weihl, W., "Nested Transactions and Read/Write Locking," *Proc. 6 ACM Symposium on Principles of Database Systems*, 1987.

Kim, W., Lorie, R., McNabb, D., Plouffe, W., "A Transaction Mechanism for Engineering Design Databases," *Proc. 10th Conf. on Very Large Databases*, 1984.

Korth, H., "Locking Primitives in a Database System," *J. ACM*, 30:1, January 1983, pp. 55-79.

Korth, H., Kim, W., Bancilhon, F., "On Long-Duration CAD Transactions" to appear, *Information Sciences*, 1988.

Liskov, B., Curtis, D., Johnson, P., Scheifler, R., "Implementation of Argus," *Proc. 11 ACM Symposium on Operating Systems Principles*, 1987.

Lorie, R., Plouffe, W., "Relational Databases for Engineering Data," IBM Research Report, RJ 3847 (43914), 1983.

Lynch, N. "Concurrency Control for Resilient Nested Transactions," *Advances in Computing Research - The Theory of Databases*, ed. F. Preparata, 1986.

Moss, J. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

Moss, J., Griffeth, N., Graham, M., "Abstraction in Recovery Management," *Proc. 12th ACM SIGMOD Conference*, 1986.

Papadimitriou, C., "The Serializability of Concurrent Database Updates," *J. ACM*, 26:4, October 1979, pp. 631-653.

Papadimitriou, C., *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

Papadimitriou, C., Kanellakis, P., "On Concurrency Control by Multiple Versions," *Proc. ACM Symposium on Principles of Database Systems*, 1982.

Weikum, G., Schek, H.-J., "Architectural Issues of Transactions Management in Multi-Level Systems," *Proc. 10 Conf. on Very Large Databases*, 1984.

Yannakakis, M., "Issues of Correctness in Database Concurrency Control by Locking," *J. ACM*, 29:3, July 1982, pp. 718-740.