FORMAL MODEL OF CORRECTNESS WITHOUT SERIALIZABILITY

Henry F. Korth and Gregory Speegle

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-47 December 1987

Formal Model of Correctness Without Serializability*

Henry F. Korth
Gregory 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 in the model. 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 non-serializable executions that are correct under our model.

1. Introduction

The classical approach to the theory of database concurrency control [Bernstein et al. 1987, Papadimitriou 1986, Eswaran et al. 1976] 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 [Yannakakis 1982] which implies that if transaction systems have no special structure then two phase locking is necessary to ensure serializability. 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

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

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. It is impossible to show that schedules legal under these schemes ensure serializability. Indeed, for any consistency constraint C, it is possible to construct a legal schedule mapping a state satisfying C to one that does not. 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 is a correctness criteria which states 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. Likewise, a transaction will see either all of the updates performed by another transaction, or none of them, thus seeing a consistent database. Therefore, 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. Of course, 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 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 [Papadimitriou 1986, Bernstein et al. 1987]. Multiple versions are simply old values of a data item retained by the system. 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 and leaves the other versions alone. Obviously, any protocol operating on a system with single versions can be simulated by a protocol on a system with multiple versions by having the version function assign only the last version created to any transaction which attempts to read the data item. In fact, the schedules which are multiversion serializable form 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]. The literature contains many articles on nested transaction systems [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]. The fundamental difference between a nested transaction system and an unnested one is the operations allowed to transactions. In an unnested scheme, such as traditional database systems, transactions are modeled as a series of reads and writes. 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, like figure 1.

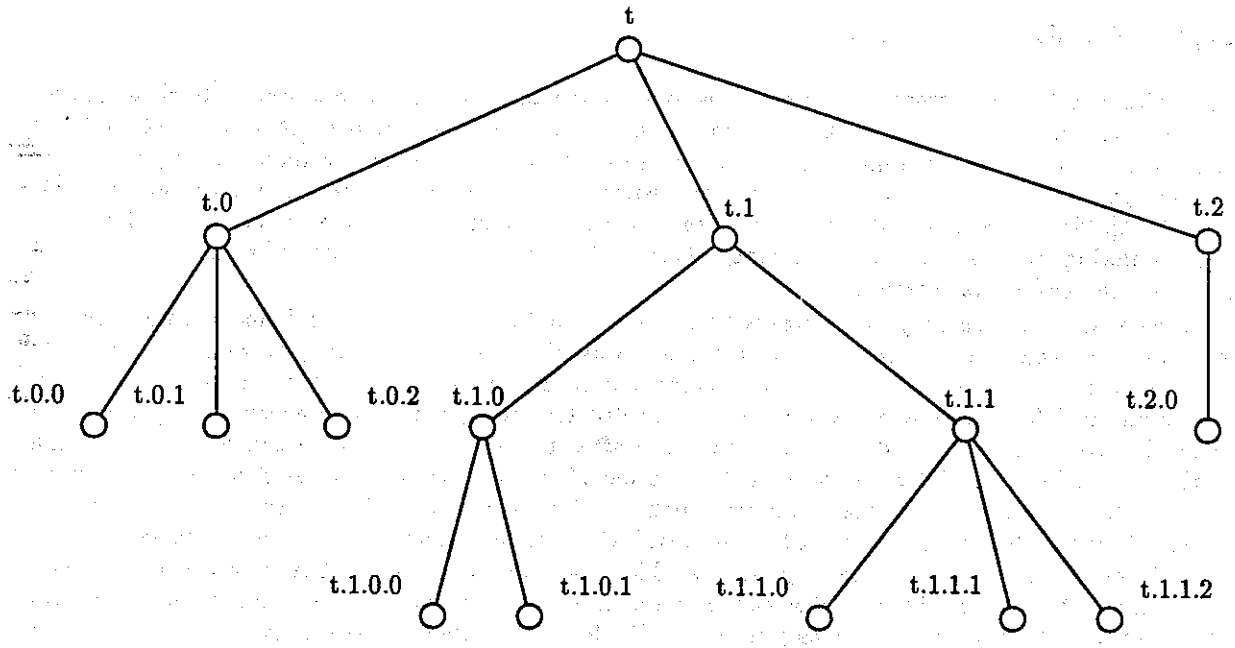

Figure 1

A nested transaction such as the one shown in figure 1, could execute in many different ways. For example, it could execute such that all of the leaves, which are the basic operations, are in a serial order such as t.0.0,t.0.1,...t.2.0. Or, transaction t might have a very interleaved execution. The transaction might have created the first subtransaction, t.0, and performed t.0.0 and t.0.1, when the user realized more work was needed. The user then contacted another user to perform some task, t.1. That user then split this job into two parts, t.1.0 and t.1.1, one of which was given to yet another user. Thus, t.0.2,t.1.0.0,t.1.0.1,t.1.1.0,t.1.1.1, and t.1.1.2 represent the steps of three interleaved transactions. Finally, t.0 and t.1 terminate, and the user of t creates a final subtransaction to perform one more operation. Now t.2 is created and t.2.0 is performed. Obviously, many different executions are possible, but this shows some of the potential for nested transactions.

Note that each subtransaction can have only one parent, but each parent can create many subtransactions. The leaves of the tree, for example t.2.0 and t.1.1.1, are database operations, which cannot be broken down into smaller parts. The root of this nested transaction system is typically a special transaction which contains only nesting operations. A transaction can contain either database access statements, or it can create subtransactions, however, it cannot do both. This does not reduce the modeling ability of nested transactions, however, as any transaction which desires to perform both nesting and access operations can create subtransactions for each of its needed database accesses. In figure 1, this is displayed in the creation of t.2, which only performs one operation, t.2.0.

Another reason for using nested transactions is that they provide a mechanism to allow needed interaction for long duration transactions. These interactions are defined in [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. Note that if all of these top-level transactions contained only database accesses, then the nested transaction system would have the identical properties of an unnested system, the only difference being the presence of the root transaction. 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 at 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]. The semantics of the read statement allow two read operations to access the same data item without conflicting, thus increasing concurrency. 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. Thus, a concurrent execution of transactions is considered to be correct if all of the transactions leave the database in a state which satisfies their postconditions. For example, top-level transactions in the nested structure might have to satisfy the database consistency constraint, however, their subtransactions might have to preserve a predicate which does not leave the database in a consistent state, but a state in which another subtransaction can execute. That subtransaction, or yet another which executes later, then restores the database to a consistent state. By doing this, the database remains consistent across entire long duration transactions, but the smaller pieces of the transaction can achieve greater concurrency. This is similar to a traditional transaction temporarily invalidating the database during its execution, but restoring the database before it terminates.

2.4 Increased Concurrency Versus Increased Overhead

Each of the above features can increase the concurrency of any database, however, the overhead involved in using them is easily avoided in traditional systems with short duration transactions by using simple concurrency control techniques like two-phase locking. In applications with long duration transactions, two-phase locking leads to unacceptably long waiting time as transactions await the release of locks held by other transactions. Alternatives to two-phase locking based on timestamps lead either to long-duration delays (e.g. conservative timestamp ordering [Bernstein et al. 1987]) or to aborts of transactions. Aborts are undesirable when transactions are of long duration since a substantial amount of work is undone. It has been shown by Yannakakis [Yannakakis 1982, Papadimitriou 1986] that the only alternative to two-phase locking is a protocol which imposes a structure on the database entities (e.g. the tree protocol of [Silberschatz and Kedem 1980]). Structuring, however, restricts the order in which transactions must access data, thereby reducing concurrency.

Many applications of long duration transactions do not require serializability, though consistency must be preserved. For example, all that matters in a design application is that the design is correct. The serialization order (if any) of the work of each designer is not relevant. Furthermore, long-duration transactions justify the use of more sophisticated concurrency control techniques even if these techniques have somewhat higher overhead. Multiple versions are required in design applications for reference purposes, so it is easy to justify their use to enhance concurrency. A nested structure fits long-duration transactions well, as has been suggested before [Moss 1985, Kim et al. 1984]. We shall see that the use of preconditions and postconditions allows us to permit a large class of executions which, though correct, are not necessarily serializable.

Since the most costly component of most applications with long-duration transactions is the time spent by humans interacting with the transactions, we feel that it is essential to consider concurrency control

schemes which:

- reduce the number and duration of waits
- reduce the number and affect of aborts
- facilitate collaboration between users

The model we propose in this paper provides a framework for correct, highly concurrent executions of long transactions. The history of every data item is preserved by keeping multiple versions of it. Every transaction is modeled as a nested transaction, complete with potentially parallel subtransactions. Also, each transaction has associated with it a precondition and a postcondition which determines the correctness of individual transactions. This allows us to claim that if all transactions execute correctly, then the system is correct.

Section 3 defines our model. In Section 4, we examine various classes of schedules achievable within our model. This allows us to characterize in a formal sense the increased concurrency provided by each feature of our model. In Section 5, we show the potential of our model in a practical sense by describing a concurrency protocol based on our model.

3. Formal Presentation of the Model

Long duration transactions can benefit from using additional semantics, such as consistency predicates, multiple versions of data items and nesting of transactions. In Section 2, we showed intuitively how these semantics can be used to increase the concurrency of long duration systems. However, a formal model is needed to characterize this increase, and to prove that the database remains consistent with this additional concurrency. Such a formal model must allow representations of nested transactions, multiple versions and database predicates. 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 concept allows us to capture the full scope of the additional semantics and by appropriate restrictions applied to these mappings, represent other transaction models as well. Unfortunately, this technique emphasizes concurrency control over recovery. Our model has a notion of recoverability, which will be discussed later. For now, we focus on concurrency issues.

3.1 Definitions

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 $\bigcup_{e \in E} \mathbf{dom}(e)$ 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 one of the comparison operators, $=, \neq, <, \leq, >$, or $\geq$), and x and y are either database 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, a predicate, 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 and starting in a consistent state, will leave the database in a consistent state.

We represent multiple versions in our model 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 $V_S = \{f : E \rightarrow \bigcup_{e \in E} \mathrm{dom}(e) | \forall e(f(e) \in \mathrm{dom}\ (e) \wedge \exists g \in S\ (g(e) = f(e)))\}$. Let V represent the set of all possible version states, and therefore be the same as 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 also 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. A transaction satisfies its specification 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. This leads to the following definition.

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 transactions $t_i \in T$, t is the parent of t_i. By our definition, any part of t which cannot be divided into subtransactions is a basic operation of the system. Basic operations are usually thought of as read and write accesses to the database, but can include other accesses such as increment and decrement operations [Korth 1983] or complex design update operations [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 predicates in 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 $t = (T, P, I_t, O_t)$ 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 $t_i \in T, \forall e \in E$, either

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

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.

3.2 Proofs of model properties

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]. The proof uses the following lemma which states that finding a correct version assignment for one transaction is an NP-complete problem. Formally, this problem can be represented by asking is $I_{t_i}(X(t_i))$ satisfiable under a given S?

Lemma 1: One transaction version correctness is NP-complete

The one transaction version correctness problem is:

> Given a set E of entities, a transaction $t = (T, P, I_t, O_t)$, a relation R on T, and a database state S, for a given $t_i \in T$, does there exist X such that $I_{t_i}(X(t_i))$ is satisfied?

Proof:

Part 1: One transaction version correctness is in NP.

Step 1: A nondeterministic algorithm guesses $X(t_i)$.

Step 2: $I_{t_i}(X(t_i))$ is evaluated, which is polynomial since the range of X is a version state, which has only one value per data item.

Part 2: One transaction version correctness is NP-hard.

The proof proceeds by transformation of the satisfiability problem. This problem is stated as:

> Given a set U of variables and a predicate C over the variables in U, is there a truth assignment for C?

Step 1: Let $E = U$.

Step 2: Let $S = \{S_0^U, S_1^U\}$, where $\forall e \in E, S_0^U(e) = 0 \wedge S_1^U(e) = 1$.

Step 3: Let $I_{t_i} = C$.

If C is satisfiable, there is an assignment of values 0 or 1 to the variables in U that makes C true. Note that V_S contains version states which have all possible combinations of 0's and 1's for the data items in E, that is V_S represents all possible assignments of values to the variables in U. If C is satisfiable, then there exists an X such that $X(t_i) = v$, $v \in V_S$, and $v(u_k) = 1$ if u_k is true, and $v(u_k) = 0$ if u_k is false. If C is not satisfiable, then X cannot map a version state to t_i, such that $I_{t_i}(X(t_i))$ holds.

Therefore, the satisfiability problem can be transformed into the one transaction version correctness problem. □

Theorem 1: Execution correctness is NP-complete

The execution correctness problem is defined as:

> Given a set E of entities, a transaction $t = (T, P, I_t, O_t)$, with parent(t) =nil, does there exist a correct (R, X)? Or, in other words, $\forall t_i \in T$, does $I_{t_i}(X(t_i)) \wedge O_t(X(t_f))$ hold?

Proof:

Part 1: Execution correctness is in NP.

Step 1: A nondeterministic algorithm guesses X.

Step 2: $\forall t_i \in T$, evaluate $I_{t_i}(X(t_i))$.

Step 3: Evaluate $O_t(X(t_f))$.

Part 2: Execution correctness is NP-hard.

The proof proceeds by showing the one transaction version correctness problem is a sub-problem to this one.

Step 1: Let $T = \{t_1\}$.

Step 2: Let O_t =true.

Therefore, $\forall t_i \in T, (I_{t_i}(X(t_i)) \wedge O_t(X(t_f))) = (I_{t_1}(X(t_1)) \wedge O_t(X(t_f)))$, by step 1. Then $I_{t_1}(X(t_1)) \wedge O_t(X(t_f)) = I_{t_1}(X(t_1))$, by step 2. Thus, (R, X) is correct if $I_{t_1}(X(t_1))$ is satisfiable. □

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]. However, these classes are too restrictive for use in long duration transaction systems. Thus, we present broader correctness classes for use in these systems.

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, as can be seen by the following lemma.

Lemma 2: All view serializable schedules are correct executions

Proof sketch:

For an execution to be correct, all transactions must have a satisfied input predicate, which in the case of view serializable schedules, is the database consistency constraint. Since a view serializable schedule is view equivalent to a serial schedule, all transactions perform reads on data items which they could have read

in the serial schedule. Since it is assumed that all transactions preserve the database consistency constraint, these reads must also be consistent with this predicate. Likewise, the final result of a view serializable schedule is the same as a serial schedule, and thus also satisfies the database consistency constraint, which is the output condition of the database. Therefore, all view serializable schedules are correct executions. □

There are obviously many schedules which are correct executions, but which are not view serializable, since correct executions allows multiple versions and partial orders, which are not part of the standard model. We can apply restrictions to 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. Transactions t_0 and t_f are the initial transaction (which writes the "initial" database) and the final transaction (which reads the "final" database) respectively. 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 lemma 3, this order is formally noted by the function f.

Lemma 3: An execution (R, X) is view 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: $\exists$ 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: $f(t_i) = f(t_j) + 1 \Rightarrow X(t_i) = t_j(X(t_j))$.

Proof:

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

Subpart A: Both schedules must contain the same transactions - true by part 2.

Subpart B: Each transaction reads the same values in both schedules - true by parts 3 and 4.

Subpart C: 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. In this section, multiple versions, predicates and partial orders will be added to the standard model to create new correctness classes. These are the semantics the model is designed to represent.

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:

t_1:	R(x)	W(x)					R(y)	W(y)
t_2:			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 transaction is assumed to execute correctly if its operations are executed in any total order consistent with the partial order given in its implementation (T, P). The serializability constraint means that the transactions themselves must still be serializable. The increased concurrency from such a structure is obvious when a locking protocol is used. 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 to the following.

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

The concept of 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 conjunct. The increased concurrency for this class is derived from the fact the serializable schedules for each conjunct do not have to agree.

It should be noted that we are assuming that no database has an empty consistency constraint. Indeed, for such a database, any schedule would preserve "consistency," and no concurrency control of any form is needed!

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

An important issue concerning the relationship between PWSR and SR is the definition of the predicate. 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 set of transactions in the schedule 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.

t_1:	R(x)	W(x)					R(y)	W(y)
t_2:			R(x)	R(y)	W(y)			

Example 2.

Since this is the same schedule as example 1, the same reasoning holds for why this schedule is not in SR. $PWSR_C$ can decompose this schedule into the two following schedules based on the conjuncts presented here.

t_1:	R(x)	W(x)		
t_2:			R(x)	

Example 3.a

t_1:			R(y)	W(y)
t_2:	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, $\prec$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. It turns out that the only conflicts which exist in MVCSR are reads before writes on the same data item [Papadimitriou 1986, Bernstein et al. 1987]. The class PWCSR simply enforces conflict serializability on each conjunct of the database constraint, and $\prec$CSR allows conflict serializable schedules but permits operations to be placed in a partial order within a transaction.

Again, as with the previous section, if we combine all of the properties of MVCSR, PWCSR, and $\prec$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 2 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

t_1:	R(x)		W(x)	
t_2:		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 ∪ $\prec$CSR ∪ SR)

t_1:	R(y)	W(x)			W(y)
t_2:	R(x)		W(x)	W(y)	

This schedule is not in PWCSR since the decompositions lead to nonserializable schedule. Likewise, 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 ∪ $\prec$CSR ∪ SR)

t_1:	R(x)	W(x)				R(y)	W(y)
t_2:			R(x)	W(x)	R(y)	W(y)	

This schedule is clearly in PWCSR if x and y are in different conjuncts. Likewise, 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

t_1:	R(x)	W(x)				R(y)	W(y)
t_2:			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

t_1:	R(x)	W(x)	
t_2:	W(x)		
t_3:			W(x)

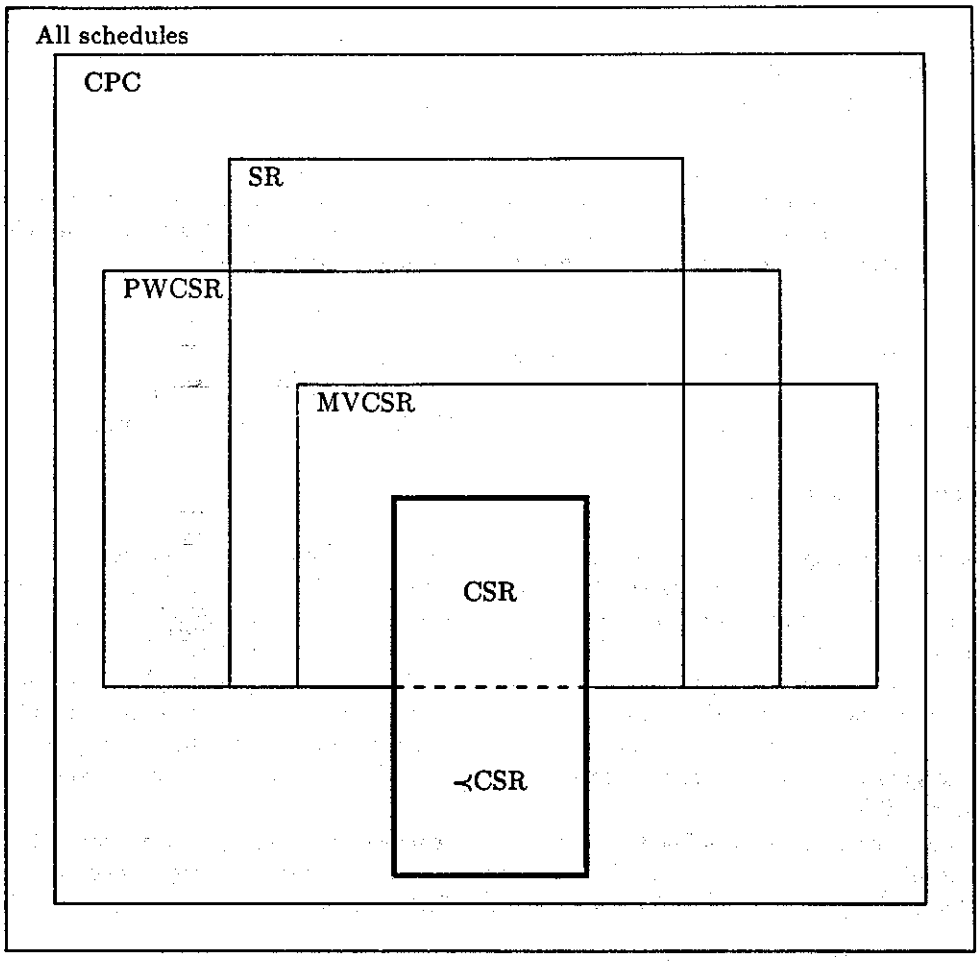

Figure 2

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

t_1:	R(x)				W(y)	
t_2:		W(y)				
t_3:			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

t_1:	R(x)		W(x)
t_2:		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

t_1:	R(x)	W(x)				W(y)			
t_2:			R(x)	W(y)					
t_3:							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

t_1:	R(x)	W(x)			R(y)	W(y)			
t_2:			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. Protocols work on a different level than the model properties presented in Section 3. In that section, the entire computation is given and whatever properties the computation has can be proven by examining this history. This is a static view of the database, since everything has already terminated, and the final result can be examined. For the protocols, the database is an active entity with possibly many transactions executing simultaneously at many different places. The system receives requests from these transactions, and based on the current state of the database, performs a correct action. It is the goal of these protocols to ensure consistency, however that is defined, and allow as much concurrency as possible.

In our model, correctness is defined as the class of correct executions, and a protocol which allows only correct executions is presented here. Other protocols including virtual timestamps and predicatewise two-phase locking [Korth et al 1988] have been re-defined to fit within our model. These will be described in a future paper.

5.1 A Correct Execution Protocol

Long duration transactions in this model 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.

Correct executions are violated due to two conditions: partial order invalidation and input constraint dissatisfaction. In terms of the formal model, either R or X can be incorrect. R can violate the definition of an execution, which says $(t_1, t_2) \in P^+ \Rightarrow (t_2, t_1) \notin R^+$, while X, can either map a non-unique state to a transaction, or $\exists t_i$ such that $I_{t_i}(X(t_i))$ does not hold. For this protocol, partial order invalidation is caused by a transaction reading a version of a data item written by the wrong transaction. This type of error is similar to an non-serializable schedule in that the database could become inconsistent because the transaction is reading the wrong data. Input constraint dissatisfaction occurs when a transaction attempts to execute with data which does not satisfy its input constraint. The execution of the transaction at this point is undefined. Therefore, a concurrency control protocol should prevent both of these conditions.

During the transaction definition phase, the only possible problem is partial order invalidation. Once a transaction has been defined, the next step is to test the partial order to determine if it is still correct. This can be done by building a graph representing the partial order and testing the graph for cycles. If any cycles are found, then the partial order has been violated and the defined transaction is aborted. Another problem could occur if the defined transaction is placed in the partial order before some transaction which has already

committed. That is, $\hat{O}_{t_1} \cap N_{t_2} \neq \emptyset$ and t_2 commits before t_1 is defined. In that case, the partial order would be violated if the new transaction wrote a data item which the committed transaction was supposed to read. There are two ways to solve this problem. Either we must undo the committed transaction, which is possible since the commit is only relative to the parent, or we can prohibit such a construction. Since recovery is not a primary concern with this work, we shall assume the latter option, though we may consider the former in practice. After the partial order has been verified, the transaction is named. One method to name a transaction is to append a number to the name of the parent, which is greater than any previously assigned to a subtransaction, such as is done in figure 1 of Section 2.

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. A pessimistic protocol could require the transaction block at this point until all predecessors have either committed or written every data item in the transactions input set, but this could require an extremely long wait, which our protocol avoids.

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 have been determined for every data item in the input set of the transaction, the system must still select a single version for each data item such that the input constraint is satisfied. This is the second part of the version assignment problem. Since multiple versions exist for each data item, an exhaustive search of all possible combinations would take time exponential in the number of data items. Instead, a heuristic based scheme should be used for selecting versions. Another alternative would be to treat the version selection process as a query, which it very closely resembles, to find the tuples which satisfy the predicate. If a satisfactory SQL-type query could be built from the predicate, then typical database optimizations, like indices, could be used to reduce the time required to find a satisfactory set of versions. Additionally, the expected case should be better than the general case since at least one transaction in every non-leaf will have only one version, to chose for each value. Likewise, the partial-order restrictions and parent-world view also serve to reduce the number of versions which need to be included in the search space. Also remember that any satisfactory set of versions can be used, which increases the number of goal states for the search. Finally, 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 is augmented by 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.

A transaction which consists of subtransaction operations does not acquire read locks, but it does use both write and R_v-locks. A non-leaf transaction is validated in exactly the same way as a database access transaction, complete with acquiring R_v-locks. However, since a non-leaf transaction does not actually perform any read operations, it does not acquire any read-locks. Likewise, a nested transaction does not perform any write steps, but it does release versions. A version is released when the final subtransaction, t_f, terminates. Each subtransaction must verify $O_i(X(t_f))$ before it terminates.

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

Figure 3. Lock Compatibility Matrix

The table needs some explanations since it is not a conventional compatibility matrix. A "true" entry in the table means the lock can be granted, the appropriate entry is made in the entity lock table, and the transaction continues normal operations. 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 as the result of a write. These problems occur because of the optimistic nature of our protocol. Basically, 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 very similar to actions taken during the validation phase. Details of the re-eval procedure are given in figure 4.

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. The only rule for the concurrency control protocol is that a transaction cannot commit until all of its predecessors have committed, all of its subtransactions must have terminated and its output condition is satisfied.

Although we will soon prove this protocol correct, it can be optimized in many ways. For example, we can identify siblings before calling the re-eval procedure, thus reducing the work that procedure has to do. However, we defer these optimizations to a future paper.

```
Re-eval (R,W,e);                                  (* e is the data item being written;
                                                   R is an array of transactions which hold
                                                   read locks or R_v locks on data item e; W is
                                                   the transaction which is writing the new version *)
i:=0
While (R[i].name ≠ nil)                           (* R[i].name is the name of
                                                   the transaction *)
    if (prefix(R[i].name=prefix(W.name))          (*prefix returns the all but the last element
                                                   in the name of a transaction, so this
                                                   checks to see if the transactions are siblings. *)
        if (path(parent(W).P,W.name,R[i].name))   (* path(a,b,c) returns true if there exists a
                                                   path in a partial order (a) from (b) to (c).
                                                   This determines if W is a predecessor to the
                                                   lock holding transaction. *)
            V:= the author of the version read by R[i]
            if (path(parent(W).P,V.name,W.name))  (*this determines if W is a successor
                                                   of V. If it is, then the transaction which
                                                   read from V, should have read from W. *)
                if (R holds an R_v lock)
                    re-assign(R[i])               (* if the transaction held a read for validation
                                                   lock, then it must have its versions re-assigned to
                                                   see if an acceptable set of versions exists which
                                                   includes the version written by W *)
                else
                    abort(R[i])                   (* if the transaction held a read lock,
                                                   then it has already read the data item and must be
                                                   aborted due to partial order invalildation *)
                end if
            end if
        end if
    end if
    i:=i+1;
end while
```

Figure 4

5.2 Proof of Protocol Correctness

In Section 4 a schedule is considered to be a correct execution if every transaction sees a database state which satisfies its input constraint and every transaction leaves the database in a state consistent with its output condition. An execution is considered to be parent-based if all transactions read from either siblings or its parent and no other transactions. All executions legal under the protocol presented in Section 5.1 are both parent-based and correct.

Lemma 4: All executions allowed under this protocol are parent-based executions.

Proof:

This property requires two parts to be proven.

1. $(t_i, t_j) \in P^+ \Rightarrow (t_j, t_i) \notin R^+$, and
2. For all transactions $t = (T, P, I_t, O_t), \forall t_i \in T, \forall e \in E$, either
   a. $(X(t_i))(e) = (X(t))(e)$, or
   b. $\exists t_j \in T$ such that $(t_j, t_i) \in R$ and $(X(t_i))(e) = (t_j(X(t_j)))(e)$.

Part 1:

Follows from the first restriction on transactions in the set D_i.

Part 2:

Follows from the initial condition on transactions in the set D_i. □

It now remains to show that all executions legal under this protocol are correct executions.

Theorem 2: The Concurrency Control Protocol Allows Only Correct Executions

A legal schedule under the protocol is correct if for the transaction $t = (T, P, I_t, O_t)$, where parent(t)=nil, there exists (R, X), such that $\forall t_i \in T, I_{t_i}(X(t_i)) \wedge O_t(X(t_f))$.

Proof:

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. Although this is good from the point of view of increasing concurrency, it may be bad from the point of view of scheduler complexity. We investigated the complexity question both formally and from a practical perspective. We defined classes of schedules in our model and compared these classes with those that exist under the classical model. Although the richest of our classes have an NP-complete recognition problem, this is no worse that in the classical model in which serializability testing is NP-complete. We showed several interesting subclasses for which the recognition problem is polynomial. This was achieved by generalizing the notion of conflict serializability.

To show that our model has potential as a practical scheme for transaction management, we showed that earlier transaction management protocols can be represented easily on our model. We then 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.

The new protocol is not necessarily optimal. Although it offers a high degree of concurrency along with a formal notion of correctness, we expect that still better protocols can be defined using our model. Much future work remains to identify the best performing protocols under our model, and many implementation details are still to be addressed, particularly in the area of efficient recovery from failures. We are currently investigating both implementation questions and extensions to the formal model.

References:

1 Bancilhon, F., Kim, W., Korth, H., "A Model of CAD Transactions," Proceedings of the 11th Conference on Very Large Databases, 1985.

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

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

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

5 Fekete, A., Lynch, N., Merritt, M., Weihl, W., "Nested Transactions and Read/Write Locking," Proceedings of the 6th ACM Symposium on Principles of Database Systems, 1987.
6 Kim, W., Lorie, R., McNabb, D., Plouffe, W., "A Transaction Mechanism for Engineering Design Databases," Proceedings of the 10th Conference on Very Large Databases, 1984.
7 Korth, H., "Locking Primitives in a Database System," Journal of the ACM, 30:1, January 1983, pp. 55-79.
8 Korth, H., Kim, W., Bancilhon, F., "On Long-Duration CAD Transactions" to appear, Information Sciences, 1988.
9 Kung, H., Papadimitriou, C., "An Optimality Theory for Concurrency Control for Databases," Acta Informatica, 1983.
10 Liskov, B., Curtis, D., Johnson, P., Scheifler, R., "Implementation of Argus," Proceedings of the 11th ACM Symposium on Operating Systems Principles, 1987.
11 Lorie, R., Plouffe, W., "Relational Databases for Engineering Data," IBM Research Report, RJ 3847 (43914), 1983.
12 Lynch, N. "Concurrency Control for Resilient Nested Transactions," *Advances in Computing Research - The Theory of Databases*, ed. F. Preparata, 1986.
13 Moss, J. *Nested Transactions - An Approach to Reliable Distributed Computing*, The MIT Press, 1985.
14 Moss, J., Griffeth, N., Graham, M., "Abstraction in Recovery Management," Proceedings of the 12th ACM SIGMOD Conference, 1986.
15 Papadimitriou, C., "The Serializability of Concurrent Database Updates," Journal of the ACM, 26:4, October 1979, pp. 631-653.
16 Papadimitriou, C., *The Theory of Database Concurrency Control*, Computer Science Press, 1986.
17 Papadimitriou, C., Kanellakis, P., "On Concurrency Control by Multiple Versions," Proceedings of the 1st ACM Symposium on Principles of Database Systems, 1982.
18 Silberschatz, A., and Kedem, Z., "Consistency in Hierarchical Database Systems," Journal of the ACM, 27:1, January 1980, pp. 72-80.
19 Weikum, G., Schek, H.-J., "Architectural Issues of Transactions Management in Multi-Level Systems," Proceedings of the 10th Conference on Very Large Databases, 1984.
20 Yannakakis, M., "Issues of Correctness in Database Concurrency Control by Locking," Journal of the ACM, 29:3, July 1982, pp. 718-740.