INTRA-TRANSACTION CONCURRENCY CONTROL AND THE NT/PV MODEL*

Henry F. Korth and Gregory D. Speegle

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

TR-90-26 August 1990

* Work partially supported by a grant from the IBM Corporation, and TARP grant 4355.

Intra-Transaction Concurrency Control and the NT/PV Model*

Henry F. Korth

Gregory D. Speegle

Department of Computer Sciences

University of Texas at Austin

Austin, TX 78712

Abstract

In nested transactions systems, it is possible for each level of a transaction to have its own correctness criteria. This results in a complex, multilevel concurrency control problem. We propose an approach to this problem called the intra-transaction concurrency control paradigm for handling multilevel concurrency control applications. This paradigm is based on defining a restricted scope for the concurrency control protocols used at each level of nesting. As such, the process of deriving multilevel protocols is simplified. To demonstrate this, two example protocols, one for the class conflict serializability (CSR) and one for the class conflict predicate correct (CPC) [KS88] are presented.

1 Introduction

Typical database applications, such as banking and airline reservation systems, consist of short transactions which access a small number of data items and execute in seconds. Such applications require executions to be equivalent to an execution in which only one transaction executes at a time. This correctness criterion is called serializability.

However, databases are now being used in applications such as computer-aided design (CAD), computer-aided software engineering (CASE), office information systems and knowledge bases. These new applications consist of long-duration transactions which can require weeks to execute. Correctness of such transactions is based on the state of the database when the transaction terminates, and is often independent of the order of execution. Thus the requirement of a serial execution as a basis for correctness is unnecessary for these applications, and therefore serializability is an overly strict correctness criterion.

*Work partially supported by a grant from the IBM Corporation, and TARP grant 4355.

Submitted to the Seventh International Conference on Data Engineering, Kobe, Japan, April 8-12, 1991.

Therefore, current long-duration transaction systems use informal correctness criteria and ultimately rely on the users to manipulate the database into a correct state [DG87, HL82, KLMP84]. Not only do these systems fail to have a mechanism for ensuring correctness, but they also must rely on the user's ability to understand concurrent activity as well as their actual project. Although these systems do allow the performance needed for the new database applications, they do not provide sufficient correctness criteria.

The NT/PV model [KS88, KS90] is an approach which uses formal correctness criteria while providing reasonable performance for long-duration transaction systems. It includes features required by long-duration transaction systems, but which are unused in typical database applications. These features are nested transactions, explicit predicates and multiple versions.

The combination of these features opens new questions for concurrency control. One of these questions involves the use of different correctness criteria for different sets of transactions. In [KKB88] a case is made for using different criteria in CAD databases. They propose using two-phase locking to resolve conflicts between top-level design projects, while more relaxed protocols are used to allow designers to cooperate within a project. If we generalize their proposal to allow each transaction to define its own correctness criteria, then *multilevel concurrency control* is achieved.

A transaction system operating under multilevel concurrency control could conceivably have a different type of transaction manager for each transaction. Due to the complexity of the code of such a system, this would be impractical. However, it is realistic to limit a system to only a few protocols whose code is shared by several transaction manager instances. Thus, a system may have one protocol controlling top-level transactions, another handling cooperation among designers within a project, yet another to provide isolation between designers within a project, and a fourth to provide complex operation atomicity. At different places within the transaction, these protocols must cooperate to ensure correctness. There are two approaches to such a system. The first is to build an ad-hoc protocol which can cooperate with exactly one other protocol, such as a version of two-phase locking which can cooperate with a version of timestamps. The other approach is to develop a paradigm for handling general cooperation between transaction managers.

We have taken the second approach in developing the intra-transaction concurrency control paradigm. This paradigm assigns the resolution of each conflict to a unique transaction manager instance. This is accomplished by restricting the scope of every transaction manager instance to the subtransactions of a single transaction. Conflicts at lower levels in the transaction hierarchy are assumed to be resolved by transaction manager instances associated with lower levels. We present a protocol, called the L-VC protocol that exploits the intra-transaction concurrency control paradigm for the class CPC [KS88].

The rest of this paper is divided into four sections. Section 2 provides an overview of the NT/PV model. Section 3 presents multilevel concurrency control and the intra-transaction concurrency control paradigm. Section 4 presents the L-VC protocol for the class CPC. Finally, Section 5 concludes this paper with some observations on intra-transaction concurrency control, the NT/PV model and long-duration transaction

systems.

2 The NT/PV Model

In this section, each of the parts of the NT/PV model is presented with informal descriptions. These parts, nested transactions, explicit predicate and multiple versions, combine to capture the semantics of long-duration transaction systems. For more details, see [KS88].

2.1 Nested Transactions

Nested transactions are used in [Mos85] as a model for presenting protocols for distributed databases. They are used to describe the interactions between designers in computer-aided design applications in [BKK85, KKB88]. Nested transactions are an important extension of the standard database model, allowing greater semantics to be captured, and greater concurrency to be achieved.

Nested transactions are based on the concept of the *subtransaction*. A one-level subtransaction is a collection of read and write steps which the rest of the transaction views as a single atomic action. In general, a subtransaction can contain not only read and write steps, but also other subtransactions, thus changing a transaction from a sequence of steps into a hierarchy of subtransactions.

The two advantages of using nested transactions are greater performance and refined recovery. In the standard database model, every transaction has a total order for all of its steps. With nested transactions, each subtransaction can execute concurrently with each other and the parent transaction, thus increasing performance. In the standard data model, any problem that requires an abort causes the entire transaction to be aborted. With nested transactions, only the subtransaction containing the problem must be aborted.

2.2 Explicit Predicates

All database concurrency control is based on the notion that it is possible to determine whether or not a database is correct. It is also required that if a transaction executes independently on a consistent database, it will terminate with the database in a state which also satisfies the database consistency constraint. Thus, the correctness of serial schedules in the standard transaction model relies on the correctness of each transaction program. This, in turn, requires that each transaction programmer understand the (often implicitly defined) database consistency constraint.

Another alternative is to state explicitly, using predicate logic, what it means for a database to be correct. Correctness is then based on the values of the database entities. The values can then be placed into the formulas, and the correctness of the database can be determined.

In [KKB88], this explicit consistency constraint is put into conjunctive normal form. Thus, there exists a set of conjuncts, all of which must be satisfied for the database to be correct. For a conjunct to be satisfied,

any one of a set of boolean formulas on data items must be true with respect to the current database state. With such a database consistency constraint, a new type of correctness can be defined. If for all conjuncts, a schedule restricted to the data items appearing in the conjunct is serializable, the schedule is *predicatewise serializable* [KS88]. Note that the equivalent serial orders for each conjunct can be different. Note also that every serializable schedule is predicatewise serializable.

2.3 Multiple Versions

In the standard database model, every data item in the database has exactly one value which may change as updates are made. If instead of writing over the old value of a data item, a new version of the data item is created containing the new value while saving the old value, it is possible to achieve greater concurrency [Pap86, BHG87]. This happens because a transaction might need to read the old value, and if it has been destroyed, corrective action must be taken. A system which keeps old data values for the data items uses multiple versions.

The greater concurrency of multiple versions is not free. One of the mechanisms required by multiple versions is some method to determine which of many versions a transaction should read. Typically, a version manager tries to assign to transactions versions which satisfy the correctness criteria. There exist many examples of version managers which can be used to ensure serializability [Pap86, BHG87].

2.4 Definitions

A transaction can be thought of as a mapping from database states to database states. The domain and range of this mapping are characterized by a pair (I, O) where I and O are predicates on database states. I is the *input condition* and O is the *output condition* for the transaction. Together, these predicates form the *specification* of the transaction. A transaction satisfies its specification if for every database state which satisfies the input condition, the transaction will leave the database in a state which satisfies its output condition. A transaction specification will maintain database consistency if $O \rightarrow \mathcal{DB}$, where $\mathcal{DB}$ is the database consistency constraint. In the standard database model it is assumed that $I = O = \mathcal{DB}$.

In the NT/PV model, transactions can be nested. We represent this by associating with a transaction t a pair (T, P) where T is a set of transactions or operations and P is a partial order on T. A nested transaction is a relation over the set of all database states. That is, for a given database state, the transaction will leave the database in one of a set of database states. It is a relation instead of a function because two unordered subtransaction could update the same data item. Thus, the result produced by the transaction is determined by the run time execution order of the subtransactions. A nested transaction maintains the database consistency constraint if every possible execution of the transaction leaves the database in a consistent state.

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 [Kor83] or complex design update operations [KLMP84].

We denote a transaction t by a four-tuple (T, P, I, O), with T, P, I, and O as defined above. We denote a read (resp. write) access by t_i to an entity e by $R_i(e)$ (resp. $W_i(e)$). We use $A_i(e)$ when the distinction between read and write access does not matter. We use $t.T$ to denote the set of subtransactions of t, and $t.P$ to denote the partial order on T. Likewise, $t.I$ and $t.O$ are the precondition and postcondition of t respectively.

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 remains undefined, it may be helpful to think of this relation as representing the run-time dependencies between the subtransactions. 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 1 *An execution of a transaction $t = (T, P, I, O)$ 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 database state. If $t_k \in T$, then $X(t_k)$ is the input state of t_k.*

Note that the mapping X allows multiple versions to exist since for distinct transactions t_i and t_j, the version assigned may be different.

For some applications, it is useful to be able to compare elements of transactions at levels below the current transaction. This is accomplished by recursively taking the union of the subtransactions of all of the transactions in a level. For this definition, we assume that a leaf in the hierarchy is its own subtransaction.

Definition 2 *Let $\mathcal{T}_0 = T$. Then, $\forall i, i > 0, \mathcal{T}_i = \bigcup_{\pi \in \mathcal{T}_{i-1}} \pi.T$.*

We use R_i to denote the execution relation of the elements in $\mathcal{T}_i$. Note that $\mathbf{T}_\infty$ is the set of database access steps, and therefore, R_∞ is the execution order of the database steps. We use the relation $\prec$ to indicate that one step executes before another. The $\preceq$ relation is used in order to indicate that two steps are adjacent in the execution.

3 A Correctness Criterion for Nested Transactions

Nested transactions structure the operations of a transaction into a hierarchy, where significant units of work are grouped into subtransactions. Such a structure may be of arbitrary depth, and thus requires concurrency

control protocols to manage multiple levels of transactions. Most previous work on nested transactions [Mos85, KS88, BBG89, BR90] assume that a single concurrency control mechanism is in force at all levels of the nesting. The CAD model of [KKB88] alludes to the possible need for different mechanisms at different levels. In that paper, the top-level transactions represent separate projects which require isolation, while all transactions below the top-level represent design activities which require collaboration. Thus, to implement the model of [KKB88], the root transaction must impose a protocol on the top-level transactions which ensures serializability, but each top-level transaction must impose a weaker protocol on the cooperating transactions which comprise it. Such protocols are proposed in [KKB88].

Within the NT/PV model it is possible to generalize the concurrency control requirements of [KKB88]. In the NT/PV model, each transaction can define a correct execution for its children. Since this flexibility is not limited to top-level transactions, it is possible for every transaction within the NT/PV model to be executing under a different correctness criteria. This situation is called *multilevel concurrency control.*

3.1 Multilevel Correctness

Under multilevel correctness, each transaction $t = (T, P, I, O)$ has restrictions placed on its execution by its transaction manager. The set of all relations over subtransactions which satisfy these restrictions is denoted $t.TM$. Formally, $t.TM$ is a set of relations over $t.T \times t.T$ such that all elements in $t.TM$ satisfy a set of constraints and a set of conditions. A constraint is a formula of the form $A_i(e) \prec A_j(e) \rightarrow (t_i, t_j) \in R$, where $A_i(e) \in t_i.\mathbf{T}_\infty$, $A_j(e) \in t_j.\mathbf{T}_\infty$, $i \neq j$, and $\prec$ is the precedes operator. A condition is a restriction on the structure of R. Possible conditions on R include acyclic, totally ordered, and empty. For example, the condition for a conflict serializable (CSR) execution is that R is acyclic and the set of constraints on R is $\{W_i(e) \prec A_j(e) \rightarrow (t_i, t_j) \in R, A_i(e) \prec W_j(e) \rightarrow (t_i, t_j) \in R\}$. For simplicity, we will consider only transaction managers which have the condition that R is acyclic.

An execution (R, X) of a transaction $t = (T, P, I, O)$ is *one-level correct* if it uses only one version of the database and $R \in t.TM$. An execution (R, X) of a transaction $t = (T, P, I, O)$ is *multilevel correct* if t is one-level correct and $\forall t_i \in t.T$, t_i is multilevel correct. Note that all transactions containing only database access steps are defined to be one-level correct.

Multilevel correctness serves as a good basis for correctness in transaction systems requiring multilevel concurrency control. However, there are some problems with developing protocols for multilevel correct executions. First, since different transactions can be executing under different correctness criteria, determining which protocol should be followed for a particular step is non-trivial. For instance, assume that a top-level transaction is executing under two-phase locking, but its subtransactions execute according to basic timestamp order. If a step of a subtransaction conflicts with a sibling subtransaction and another top-level transaction, is the protocol for two-phase locking or timestamps or both used to resolve the conflict? For an execution to be multilevel correct, both protocols must be followed. Therefore, the schedules allowed under

multilevel concurrency control would be the intersection of all schedules allowed by all protocols involved in resolving a particular conflict. Thus, instead of increasing concurrency, improperly designed multilevel protocols can actually *decrease* concurrency. Secondly, the protocols must be able to execute in the presence of other protocols. One way to do this is to develop an instance of each protocol to work with an instance of every other protocol. For example, there would be an instance of two-phase locking which could be used with timestamps, and an instance of timestamps which could be used with two-phase locking. This approach requires each protocol to be more complicated than under single level concurrency control, and limits the possible combinations of protocols and correctness criteria.

The solution to this problem is not to require multilevel correct executions but, rather, to consider multilevel correct executions to be the basis for correct executions in transaction systems using multilevel concurrency control, much like serial schedules are the basis for correctness in the standard database model. Similarly to the standard database model, correct executions can now be defined as *equivalent* to a multilevel correct execution. One useful equivalence for multilevel concurrency control effectively isolates transactions from all transactions except its parent and its siblings. Since one-level correctness is based on the interleaving of the siblings, one-level correctness can now be determined without concern over the correctness of other transactions, or even the correctness of the siblings themselves, since that is determined by the protocols at lower levels. This equivalence is exploited in the *intra-transaction concurrency control* paradigm, which we describe below.

3.2 Intra-Transaction Concurrency Control

The intra-transaction concurrency control paradigm is a technique for defining the interface between protocols for different transactions. Under the intra-transaction concurrency control paradigm, each nested transaction is assigned a version of the database which becomes local to the transaction. The value of a data item in this local database is either the value of the data item in the local database of the parent, or a value written by some sibling of the transaction. The subtransactions of the transaction then execute upon this local database as if it were the database. These subtransactions can then execute in isolation from all of the actions performed by any non-sibling transaction in the system. Of course, the parent is not isolated, and therefore restrictions placed on the parent could influence the execution of the subtransactions.

However, as a result of this isolation, the protocols used under intra-transaction concurrency control do not have to interact directly with the other protocols. Each protocol is in effect only for the local version of the database assigned to its transaction. This simplifies the protocols and allows the combinations of protocols to be determined by the application and not by what the database supports.

Additionally, since only the subtransactions can access the local database, the protocol can consider all transactions to be three levels deep. The root of each transaction structure is the transaction associated with the protocol. The second level is composed of the subtransactions the protocol regulates. The third

level is a subset of the database access steps which compose the subtransactions. This subset is the initial read and final write of each data item by any subtransaction. A step $A(e)$ is an initial read for a transaction t_i if $A(e) \in t_i.T_\infty$ and for all other steps $A'(e) \in t_i.T_\infty, (A(e), A'(e)) \in t_i.R_\infty$. A step $W(e)$ is a final write for a transaction t_i if $W(e) \in t_i.T_\infty$ and for all other write steps $W'(e) \in t_i.T_\infty, (W'(e), W(e)) \in t_i.R_\infty$. Note that a blind write can be considered the initial "read" of a data item. All other steps will be regulated by protocols associated with transactions at lower levels in the structure.

Clearly, the version management system under intra-transaction concurrency control is very important. For an arbitrary read step, the version assigned to it is the current version of the data item in the local database associated with the parent of the transaction. Of course, this version could have been inherited from earlier ancestors of the transaction, but it becomes distinct from those versions. All write steps are assigned to this version as well. Old versions of data items are immediately discarded, unless the local protocol uses multiple versions.

Intra-transaction concurrency control is a relaxation of multilevel correctness. For example, assume there exists a transaction t with two subtransactions, t_1 and t_2. Let the condition of $t.TM$ be that R is acyclic. Let the constraint of $t.TM$ be $(A_i(e), A_j(e)) \in R_\infty \rightarrow (t_i, t_j) \in R$, where $A_i(e)$ (respectively $A_j(e)$) is an access of data item e by some subtransaction of t_i (respectively t_j). Assume that t_1 contains two subtransactions t_A and t_B, and that t_2 contains two subtransactions t_C and t_D. For simplicity, assume that $t_1.TM$ and $t_2.TM$ allow all executions. Let $A_i^j(e)$ be a step in subsubtransaction j of subtransaction i. Then $R_\infty = R_1^A(e)W_1^A(e)R_2^C(e)W_2^C(e)R_1^B(e)W_2^D(e)$ is allowed under intra-transaction concurrency control, but does not satisfy the constraint.

This execution is allowed by intra-transaction concurrency control because the initial read ($R_1^A(e)$) and final write ($W_1^A(e)$) of data item e by transaction t_1 both occur before the initial read ($R_2^C(e)$) and final write ($W_2^D(e)$) by transaction t_2 on data item e. However, $(t_1, t_2) \in R$ since $R_1^A(e) \prec W_2^C(e)$ and $(t_2, t_1) \in R$ since $W_2^C(e)$ precedes $R_1^B(e)$. Yet, this execution is acceptable if intra-transaction concurrency control is used since the step $R_1^B(e)$ would be assigned the version written by $W_1^A(e)$. Thus, this execution of t_1 and t_2 is *view equivalent* to the constraint, in that t_1 does not read any value written by t_2.

In the remainder of this paper we make the following minimum assumptions about transaction managers:

- The transaction manager enforces the constraint $W_\alpha(e) \prec R_\beta(e) \rightarrow (t_\alpha, t_\beta) \in R$
- The transaction manager enforces the constraint $R_\alpha(e) \prec W_\beta(e) \rightarrow (t_\alpha, t_\beta) \in R$
- The transaction manager enforces the condition R is acyclic.

Theorem 1 *If all of the transaction managers are subject to the above conditions, and an execution (R, X) of a transaction $t = (T, P, I, O)$ is allowed under intra-transaction concurrency control, then the execution is view equivalent to a multilevel correct execution of t.*

3.3 An Example Protocol

One of the primary advantages of the intra-transaction concurrency control paradigm is the capability for using standard database model protocols in a multiple level environment without modification. To demonstrate this, the following protocol requires the root transaction to use two-phase locking while every top-level transaction requires their subtransactions to execute according to basic timestamp order. The following example execution is not allowed under either two-phase locking or timestamps, but it is in CSR and it is allowed under intra-transaction concurrency control.

Let the transaction system be defined as $\mathbf{T} = (\{t_0, t_1, t_2, t_f\}, \emptyset, \mathcal{DB}, \mathcal{DB})$, where t_0 is the imaginary initial transaction which writes the initial database state and t_f is the imaginary final transaction which reads the final database state. Let $t_1 = (\{t_A, t_B\}, \emptyset, \mathcal{DB}, \mathcal{DB})$, where t_A and t_B are subtransactions which must execute in timestamp order with $TS(t_A) < TS(t_B)$. Let $t_2 = (\{t_C, t_D, t_E\}, \emptyset, \mathcal{DB}, \mathcal{DB})$ with $TS(t_C) < TS(t_D) < TS(t_E)$. Let $t_A = (\{R(e_0)\}, \emptyset, \mathcal{DB}, \mathcal{DB})$ and let $t_B = (\{R(e_3), R(e_0)\}, ((R(e_0), R(e_3))), \mathcal{DB}, \mathcal{DB})$. Let $t_C = (\{W(e_1), W(e_2), W(e_3)\}, ((W(e_1), W(e_3)), (W(e_3), W(e_2))), \mathcal{DB}, \mathcal{DB})$. Let $t_D = (\{W(e_1), W(e_2)\}, ((W(e_1), W(e_2)), \mathcal{DB}, \mathcal{DB})$. Let $t_E = (\{W(e_1)\}, \emptyset, \mathcal{DB}, \mathcal{DB})$. Let the execution order of the database steps be $R_B(e_0), W_C(e_1), W_D(e_1), R_A(e_0), W_E(e_1), W_C(e_3), R_B(e_3), W_C(e_2), W_D(e_2)$.

The intra-transaction concurrency control paradigm for two-phase locking at the root and basic timestamp order at the next level down allows this execution. The following three steps illustrate this point:

- $W_C(e_1)$: This step must be processed by both transaction managers since it is the initial access (and therefore is considered to be the initial read) of data item e_1 by both transaction t_2 and subtransaction t_C. The intra-transaction version manager assigns the initial version of e_1 to this step. Transaction t_2 is granted a write lock on the data item. The write step of subtransaction t_C is allowed by the timestamp protocol. Since this step is a final write of e_1 by subtransaction t_C, the intra-transaction version manager updates that version. Note that the version assigned to t_2 is not updated at this time.

- $R_B(e_3)$: This step is also the initial read of e_3 by both t_B and t_1. Therefore, it is assigned the value assigned to t_1. In this case, assuming that t_2 released the lock on e_3 immediately after the write by t_C, that would be the value written by the step $W_C(e_3)$. Note that if t_2 has not released the lock on e_3, the step would be blocked. However, subtransaction t_A could continue to execute.

- $W_D(e_1)$: Since this step is neither the initial read or final write of e_1 for t_2, the two-phase locking protocol is ignored. The write is allowed because the timestamp protocol allows it. Note that if the final write for t_2 had already been performed, then t_2 could have unlocked this data item. However, this write would still be allowed, since the effects would be contained within t_2, and thus the execution would still be view equivalent to a correct execution.

Note that this execution would not be allowed under two-phase locking because t_D would have to unlock data item e_1 between the third and fifth step, but it would have to lock data item e_2 after the eighth step.

By the rules of two-phase locking, this is not allowed. Note also that this execution would not be allowed under basic timestamps. In that protocol, transaction t_C would be assigned a higher timestamp than t_B. Thus, when t_B attempts to perform $R(e_3)$, the transaction is aborted because the read timestamp of the transaction is less than the write timestamp of the data item.

4 A Protocol for Conflict Predicate Correct Executions

The NT/PV model contains many features which are useful in increasing concurrency control. Multiple versions, explicit predicates and nested transactions have all be shown to allow executions which are not serializable in the strictest since, but which are equivalent to a serial schedule by a more general notion of equivalence. Note that each of these features exploits a different aspect of a database system. Multiple versions exploits the implementation of the database, while predicates exploit the semantics of the database and nested transactions exploit the semantics of a transaction. Therefore, it should be possible to gain all of the benefits of these features in one correctness criteria.

The correctness class conflict predicate correct [KS88] (CPC) exploits all of the features of the NT/PV model while still retaining the conflict ordering properties of criteria like conflict serializability. CPC exploits multiple versions in order to reduce conflicts from read-write, write-read and write-write, to only read-write. CPC exploits explicit predicates in order to reduce the potential for cycles in the execution. This is accomplished by requiring the database consistency constraint to be in conjunctive normal form. It is known [KKB88] that if the restriction of the execution to the data items in each conjunct is serializable, then the database consistency constraint is preserved, even if the execution is not serializable. Finally, CPC exploits nested transactions by allowing multilevel correctness criteria. If an execution is a CPC execution, then the restriction of the execution of the top-level transactions to the data items in each conjunct is a multiversion conflict serializable execution. However, each of the lower-level transactions must only execute according to the partial order of their parent.

The class conflict predicate correct is an excellent source for an example of multilevel concurrency control because the criteria for top-level transactions is different from the criteria for the root transaction. The root transaction needs a protocol to ensure the acyclicity of each multiversion conflict graph, one for every conjunct. Each of the top-level transactions requires a protocol to ensure that their subtransaction hierarchies execute according to the defined partial orders. Thus, two protocols are needed. The first one for the root transaction and the other for all lower-level transactions. When these protocols are combined, it must be the case that only conflict predicate correct executions are allowed. The intra-transaction concurrency control paradigm defines the cooperation between the protocols. In this section, we present a protocol that allows a large subset of the executions which are view equivalent to CPC executions. It is called the L-VC protocol, which means locking for versions and conjuncts.

		held	
		R	W
requested	R	$\Leftrightarrow$	$\Rightarrow$
	W	$\Rightarrow$	$\Leftrightarrow$

Figure 1: Lock Compatibility Matrix for Root Transaction

4.1 Protocol for the Root Transaction

The protocol for the root transaction must efficiently manage versions and data items in conjuncts. In order to do this, constrained sharing [AEA90] is exploited. Constrained sharing is a relaxation of two-phase locking which allows traditionally conflicting lock modes to be held on the same data item. For example, two transactions can hold a write lock on the same data item. This enhancement is achieved by requiring that transactions holding conflicting locks on data items do so in an *ordered shared* relationship. This relationship requires that the transactions access the data in the same order as which they acquired the locks. Additionally, a transaction may unlock a data item only if for all data items the transaction holds locks on, no current lock holder held the lock when the transaction first acquired the lock. By using these two restrictions, [AEA90] presents a locking protocol which allows all conflict serializable schedules (the class CSR).

The protocol for the root transaction exploits constrained sharing for a different reason. The subtransactions of the root transaction are supposed to execute such that the restriction of the execution to the data items in each conjunct is a multiversion conflict serializable (MVCSR) execution (see e.g. [Pap86, BHG87]). Under MVCSR, if a read step precedes a write step, then they conflict. However, if the write step precedes the read step, then the read operation can be considered to occur before or after the write step. Unfortunately, under two-phase locking, it is impossible to determine when a step will occur after a lock has been received. Thus, all read steps must conflict with all write steps. Constrained sharing resolves this problem by allowing read locks and write locks to be ordered shared. Also, for the L-VC protocol, a W lock on a data item only allows a transaction to write a data item, but not to read it. In order for a transaction to both read and write a data item, it must hold both locks. A transaction must release all locks held on a data item at the same time. Figure 1 shows the lock compatibility matrix for the root transaction. Note that the compatibility matrix is three-valued, unlike standard lock compatibility matrices which are two-valued (true and false). The values allowed are shared, denoted $\Leftrightarrow$, ordered shared, denoted $\Rightarrow$, and conflicting, denoted $\not\Leftrightarrow$.

The L-VC protocol exploits the fact a transaction must be two-phase only with respect to the data items in a given conjunct. This is exactly the same requirement as in predicatewise two-phase locking presented in [KKB88]. Thus a transaction can unlock a data item in one conjunct, and then later lock a data item in a different conjunct. This is allowed since the database consistency constraint is preserved if all conjuncts are preserved, and if no data items are shared by the conjuncts, the preservation of one conjunct is independent

Begin Transaction
Initialize transaction parameters T and P

Create Subtransaction$\{name, before, after\}$
insert $(before, name)$ into partial order $t_i.P$.
insert $(name, after)$ into partial order $t_i.P$.

Lock$\{$*data-item,mode*$\}$
Consult lock compatibility matrix (Figure 1)
If result is $\Leftrightarrow$ then allow lock
If result is $\not\Leftrightarrow$ then block t_i
If result is $\Rightarrow$ then insert t_i into an ordered list of lock holders on *data-item*

Read$\{$*data-item*$\}$
Read lock must be held
If ordered shared mode then wait until previous lock holders have executed
Assign version of *data-item* according to version management protocol

Write$\{$*data-item,value*$\}$
Write lock must be held
If ordered shared mode then wait until previous lock holders have executed
Create version of *data-item* according to version management protocol

Unlock$\{$*data-item*$\}$
If ordered shared mode then wait until previous lock holders have released all locks
Unblock transactions waiting on released lock on first-in/first-out basis

Figure 2: Operations for subtransaction t_i of the Root Transaction

of the preservation of another.

Note that here we present the protocol for concurrency control without the version manager. This follows the concept presented in [AS89] of treating the version manager and the concurrency control mechanisms as distinct protocols. The version assignment protocol is presented separately in Section 4.5

4.2 Lower-Level Transaction Protocol

The protocol for each transaction below the root is exactly the same. Each protocol ensures that the subtransactions execute according to the partial order. Note that if only one version is allowed, then all descendents of a subtransaction must execute before any descendent of a successor subtransaction executes. This protocol relaxes this constraint in two ways. First, intra-transaction concurrency control allows all steps other than the initial read and final write of the data item to be ignored in determining the execution order of the subtransactions. Second, an execution is correct even if interleaving occurs as long as each read step reads from the correct write step. Note that both of these relaxations together allow an execution to be correct if it is view equivalent to an execution which is consistent with the partial order.

If a subtransaction reads a data item, it is assigned a version. The intra-transaction concurrency control

Create Subtransaction {*name,before,after*}
insert (*before,name*) into partial order $t_i.P$
insert (*name,after*) into partial order $t_i.P$

Write{*data-item,value*}
Create version of *data-item* with author t_i and value *value*

Read{*data-item*}
Select all predecessors of t_i such that the predecessor wrote a version of *data-item*
If no such predecessor then assign initial version
If predecessors exist then begin
save the set of predecessors,
perform topological sort of partial order
assign version for which author is last in the topological sort
end

Figure 3: Operations for subtransaction t_i of a Top-Level Transaction

version manager assigns the set of versions created by the siblings of this transaction plus the version assigned to its parent. The protocol version manager then selects the versions written by the predecessors of the subtransaction, as determined by the partial-order of the parent of the subtransaction. The set of all predecessors which wrote a version is saved for later use. If no such version exists, then the protocol version manager assigns to the read step the version assigned to the parent transaction. If a subtransaction writes a data item, it creates a new version with the name of the subtransaction included with the version. Figure 3 provides pseudo-code for these operations.

Once a subtransaction attempts to commit, the validation phase of the protocol begins. A subtransaction is validated if the following three criteria are satisfied:

1. The subtransaction has terminated
2. All of the predecessors are valid
3. No predecessor wrote a data item after the subtransaction read the data item

The first criteria will be fulfilled by all subtransactions being validated, since the subtransaction will not be validated until it attempts to commit. The second criteria is required because if a predecessor has not been validated, then it could write a data item after the validation of the subtransaction has completed and invalidate the subtransaction. The third criteria is checked by selecting all predecessors which have written a value for a data item read by the subtransaction. If this set contains predecessors not in the set saved at the time of the read step, then the subtransaction cannot be validated. Note that this must be checked for every data item read by the transaction.

4.3 Proof of Protocol Correctness

It now remains to show that all executions allowed by the L-VC protocol are view equivalent to CPC executions. In order to do this, two things must be shown. First, that for every conjunct, the multiple version conflict graph over top-level transactions is acyclic, and second that every transaction executes according to its partial order. The first requirement is maintained by the root level transaction manager, while the second property is maintained by the transaction managers for all lower-level transactions.

Theorem 2 *If an execution is allowed by the L-VC protocol, it is view equivalent to a CPC execution.*

Proof:
We must show that if an execution is allowed by the L-VC protocol, then the execution is view equivalent to an execution which is consistent with the partial orders of the subtransactions and which has for every conjunct, an acyclic multiple version conflict graph.

First, assume that there exists some conjunct such that the restriction of the execution to the data items in the conjunct produces a cyclic multiple version conflict graph. WLOG, assume this cycle is of the form, $t_0, t_1, \ldots, t_{n-1}, t_0$. Note that we will use $i+1$ for $i+1$ mod n. For every pair of transactions t_i, t_{i+1}, there exists a data item e_i such that $R_i(e_i) \in t_i.T_\infty$ and $W_{i+1}(e_i) \in t_{i+1}.T_\infty$, and $(R_i(e_i), W_{i+1}(e_i)) \in R_\infty$. Note that the set of all such e_i must appear in the same conjunct. By intra-transaction concurrency control, each such read must be the first read on the data item by any database access step in the hierarchy of the transaction. Likewise, the write step must be the last such write step. Therefore, by the lock compatibility matrix for the root level transaction manager, t_i must lock e_i before t_{i+1} locks it. Therefore, by the lock release rule of [AEA90], t_i must unlock e_i before t_{i+1} unlocks e_{i+1}. Therefore, by the form of the cycle, t_0 must unlock e_0 before t_{n-1} unlocks e_{n-1}, which must happen before t_0 unlocks e_0. Thus, we have a contradiction.

Therefore, all executions allowed by this protocol are view equivalent to an execution for which each conjunct has an acyclic multiple version conflict graph.

Now assume that the execution contains a transaction, T, such that T is composed of two subtransactions, t_i and t_j, such that $(t_i, t_j) \in T.P$ but some step $\alpha_j \in t_j.T_\infty$ executes before some step $\alpha_i \in t_i.T_\infty$. If α_j does not access the data item accessed by α_i, then this execution is correct because the two transactions do not depend on each other. If α_j reads the data item and α_i reads the data item, then the execution is correct because each transaction will execute exactly the same whether or not the other is executing. If α_j writes the data item and α_i reads it, then the execution is also correct because the version manager of the top-level transaction protocol only assigns versions to t_i which are written by predecessors of t_i. Thus, the execution of t_i and t_j are still correct.

The only remaining case is when α_j reads a data item and α_i writes it. The ordering of a_i and a_j is relevant only if each step is either the first read on the data item by any database access step in the hierarchy

of the transaction or the last write step in the hierarchy. However, in this case, t_j will be aborted in the validation phase of t_j. This occurs because by intra-transaction concurrency control, α_i will be writing a version which is visible to the protocol controlling t_j. Thus t_j will fail test three of the validation criteria. Thus this execution cannot occur. Thus, all executions of t_i and t_j will be view equivalent to the partial order of T.

Therefore, every execution allowed under the L-VC protocol is view equivalent to a CPC execution. □

Note that if intra-transaction concurrency control is used for the lower-level transactions, then all executions allowed for them are not only view equivalent to multilevel correct executions, but they are multilevel correct executions. This follows since the partial order is maintained if for each data item, all write steps by a predecessor precede all read steps by a successor. By intra-transaction concurrency control, this holds since the final write (and therefore all other writes) of a predecessor must precede the initial read (and therefore all other reads) of the successor. However, view equivalence is still needed when the top-level transactions are included in the execution.

4.4 Analysis of the L-VC Protocol

Since the L-VC protocol uses the intra-transaction concurrency control paradigm, the set of executions it allows is incomparable to the set of CPC executions. Recall that an execution is in CPC if for every conjunct, the multiversion conflict graph of the top-level transactions is acyclic. Since these top-level transactions may be nested transactions, it is possible for operations to act upon values of data items which are internal with respect to the top-level transaction. Under intra-transaction concurrency control, some of these executions are allowed since they are view equivalent to a CPC execution, but the execution itself can violate the restrictions of CPC.

For example, recall the earlier example of two transactions, t_1 and t_2, where t_1 contains two subtransactions t_A and t_B, and t_2 contains two subtransactions t_C and t_D. Assume this time that the transactions are top-level transactions executing under CPC. The following execution R_∞ is allowed under intra-transaction concurrency control, but is not a CPC-execution. Again, let $A_i^j(e)$ be a step in subtransaction j of transaction i. Let $R_\infty = R_1^A(e)W_1^A(e)R_2^C(e)W_2^C(e)R_1^B(e)W_2^D(e)$. Since $(R_1^A(e), W_2^C(e)) \in R_\infty$, then $(t_1, t_2) \in R$ for the root transaction. Likewise, since $(R_1^B(e), W_2^D(e)) \in R_\infty$ then $(t_2, t_1) \in R$ for the root transaction. Therefore R for the root transaction contains a cycle under the constraints of CPC, and the execution is not a CPC execution. However, note that the execution is allowed under the L-VC protocol since only the steps $R_\infty = R_1^A(e)W_1^A(e)R_2^C(e)W_2^D(e)$ are considered. Clearly, this execution generates an acyclic graph under CPC.

Likewise, there exist schedules which are in CPC, but which are not allowed by the protocol. For example, consider an example where the only transactions in the system are two flat transactions, t_1 and t_2, where both transactions read one data item and write a second data item. Assuming that both data items, e and

e', are in the same conjunct, then the following execution is in CPC, but is not allowed by the protocol. Let $R_\infty = W_1(e)R_2(e)W_2(e')R_1(e')$. By the rules for constrained sharing in [AEA90], neither t_1 nor t_2 can release their locks, so this execution cannot terminate. However, by the definition of CPC [KS88], since the execution relation R is empty, it is therefore acyclic, and thus this execution is in CPC.

Note that this example schedule is allowed by a multiversion timestamp algorithm. Thus, it would seem reasonable to replace the protocol for top-level transactions with a multiversion timestamp protocol [Ree83]. However, the following execution is allowed by the L-VC protocol presented here and is not allowed by the multiversion timestamp protocol. Again assuming two flat transactions, t_1, t_2 are the only top-level transactions in the system. Let $R_\infty = R_1(e)R_2(e)W_1(e)$. Since t_1 executes first, it is assigned the lower timestamp. Therefore, when $W_1(e)$ is performed, the transaction is aborted since $R_2(e)$ should have read the version created by this step instead of the original version.

4.5 Version Management for the L-VC Protocol

Theorem 2 states that the L-VC protocol allows only executions which are view equivalent to executions with acyclic multiversion conflict graphs. The next step in proving correctness for multiversion protocols is to show that all executions allowed by the protocol are equivalent to a one-version serial execution of the transactions. However, in order to do this, the version control protocol must be defined.

Many version control protocols can be used with the L-VC protocol, such as multiversion timestamping [Ree83]. Below, a new version control protocol, called the write-order version control protocol is presented. This protocol assings versions such that the order imposed on the execution by the L-VC protocol is maintained by the version assignment function. In order to do this, write steps by transactions must be ordered consistently with the other accesses.

Before the write-order protocol can be presented, some additional definitions about executions are needed. A data item is *marked* if the last access to the data item is a read step. At the beginning of the execution, all data items are not marked. The version of data item e written by transaction t is denoted $V_t(e)$. $V_t(e)$ is a *fresh* version at a given point during an execution if for all other write steps on e, denoted $W'(e)$, and all read steps on e, denoted $R(e)$, $W_t(e) \prec R(e) \rightarrow W'(e) \prec R(e)$. The set of all fresh versions relative to a given version is defined as the *version set* of $V_t(e)$, and is denoted V_e^t. Formally, $V_e^t = \{V_{t'}(e) | \forall R(e), R(e) \prec W_{t'}(e) \leftrightarrow R(e) \prec W_t(e)\}$. A version $V_t(e)$ is the *freshest* version if $V_t(e)$ is a fresh version and for every version $V_{t'}(e)$ in the version set for $V_t(e)$, if t' and t lock a data item with a result of ordered shared, then t must access the data item after t', and if t and t' are unrelated by the L-VC protocol, then t locked e for write after t' locked e for write.

Intuitively, a version set corresponds to the set of versions which are written between any two read steps on a data item. This is important since the L-VC protocol ensures that if a write step by transaction t precedes a read step by transaction t' on a data item, then for all data items which t writes (respectively,

reads) and t' reads (writes), the write (read) by t will precede the read (write) by t'. The same holds true if t reads a data item before t' writes it. Therefore, between any two read steps will exist a set of versions which should be assigned to the second read step. Fresh versions are the versions which can be assigned to a read step, and the freshest version is the version which will be assigned.

The write-order protocol has three rules. One concerns the assigning of versions to read steps. The second rule creates new versions for write steps, and the third rule regulates when transactions can unlock data items.

- For a read step on data item e performed by transaction t
  - mark e
  - assign freshest version of e to $R_t(e)$
- For a write step on data item e performed by transaction t
  - if e is marked, then mark all fresh versions as stale
  - create new version of e with author t
  - unmark e
- For an unlock of a data item e by transaction t
  - If there exists a data item e' such that $V_t(e')$ is the freshest version of e', or if $V_t(e')$ is now stale but it was the freshest version in its version set, then delay the unlock until all authors of versions in the version set of $V_t(e')$ have unlocked e'
  - Else, allow the unlock step

Thus, the write-order version control protocol assigns all read steps to the freshest version of a data item. The freshest version corresponds to the version which would be created after all other versions according to the order of the transactions established by the L-VC protocol. The unlock rule is used to enforce this order, and is the same as the unlock rule for constrained sharing [AEA90].

Theorem 3 *If an execution (R, X) of a transaction $t = (T, P, I, O)$ is allowed under the L-VC protocol using the write-order version control protocol, then R is view equivalent to a one-version one-level serial execution of the top-level transactions.*

Proof:

For brevity, we will use R to denote the execution (R, X). By the definition of intra-transaction concurrency control and Theorem 2, if all initial read steps return the same values in R as in a one-level serial execution of the top-level transactions, then all non-initial read steps return the same values. Therefore, it

remains to show that all initial read steps return the same values in R as in a serial execution of the top-level transactions.

Consider the following graphs. The mvcg of R is from theorem 2. A write-read graph contains one node for each transaction and an arc exists from transaction t_i to t_j if there exists some data item e such that t_i wrote e before t_j read it. By arguments identical to those used in the proof of theorem 2 for the mvcg, the wrg of R is acyclic. Likewise construct a freshest version graph (fvg) by having one node for each transaction and an arc from t_i to t_j if there exists some data item e such that $V_{t_i}(e)$ and $V_{t_j}(e)$ are in the same version set and V_{t_j} is the freshest version in the version set. Let the order graph have one node for each transaction and contain the union of edge sets of the mvcg, wrg and fvg.

We show that a serial execution S of transaction t exists such that the order graph for S is the same as for R.

Part 1:

Assume there exists a cycle in the order graph. WLOG, let it be of the form $t_0, t_1, \ldots, t_n, t_0$. We will use the notation $i+1$ for $i+1 \bmod n$. An edge appears in the order graph only if at least one of the following is true for R: $(R_i(e), W_{i+1}(e)) \in R^*$ (arc is from the mvcg); $(W_i(e), R_{i+1}(e)) \in R^*$ (arc is from the wrg); $V_{t_i}(e)$ and $V_{t_{i+1}}(e)$ are in the same version set and $V_{t_{i+1}}(e)$ is the freshest version in the set. For the first two cases, by the lock release rule of constrained sharing, it must be the case that t_i must unlock e_i before t_{i+1} unlocks any data item. For the last case, by the unlock rule of the write-order protocol, it must be the case that t_i must unlock e_i before t_{i+1} unlocks any data item. Therefore, by arguments identical to those in theorem 2, this requires t_0 to unlock e_0 before t_0 unlocks e_0. This is a contradiction. Therefore, the order graph is acyclic.

Thus, the top-level transactions can be ordered according to the order graph. If two transactions are unrelated in the order graph, then they can be placed arbitrarily, with respect to each other, in the serial execution. Therefore, all serial executions so generated obey the properties required of the serial execution S.

Part 2:

Assume that there exists some initial read step of transaction t_i on data item e, denoted $R_i(e)$, which returns a different value in S and R. Therefore, there exists some write step (possibly written by the imaginary transaction which writes the initial values of all data items) $W_k(e)$ which writes the value returned by $R_i(e)$ in R, but another write $W_j(e)$ which writes the value returned by $R_i(e)$ in S. Therefore, either $R_i(e) \prec W_k(e)$ or $W_k(e) \prec W_j(e) \prec R_i(e)$ in S.

If $R_i(e) \prec W_k(e)$, then there exists an arc from t_i to t_k in the mvcg for S. By the construction of S, such an arc must exist in R. Since $R_i(e)$ read from $W_k(e)$ in R, there is an arc from $t_k(e)$ to $t_i(e)$ in the wrg of R. By arguments identical to those presented in part 1, the wrg and the mvcg of R are consistent. This is a contradiction. Therefore, $R_i(e)$ cannot precede $W_k(e)$.

Therefore, $W_k(e) \prec W_j(e) \prec R_i(e)$ in S. There are four possible cases which allow this construction in S. Either there is a path from t_k to t_j in one of the three graphs which combine to make the order graph, or t_k and t_j are arbitrarily ordered this way.

Case 1: t_k and t_j are arbitrarily ordered. Since both t_k and t_j both wrote data item e, either the write steps are in different version sets or they both wrote versions which were not the freshest in the set. Since $R_i(e)$ is assigned the version written by t_k, $W_k(e)$ write the freshest version. Therefore, the write steps must be in distinct version sets. By the definition of a version set, there exists some read step which precedes one of the writes but not the other. Let this read step be $R_m(e)$. Thus, in R the execution is of the form $W_k(e) \prec R_m(e) \prec W_j(e)$ or $W_j(e) \prec R_m(e) \prec W_k(e)$. However, in either case, the order graph would relate the write steps. Therefore, the transactions must be ordered by the order graph.

Case 2: There exists an arc from t_k to t_j in the mvcg for S. Therefore, $t_k \prec t_j$ in the mvcg for R. Therefore, there exists a read step in t_k and a write step in t_j, both on data item e' such that $R_k(e') \prec W_j(e')$. By the L-VC protocol, t_k must unlock e' before t_j unlocks any data item. If $W_k(e)$ and $W_j(e)$ are both in the same version set, since $W_k(e)$ is the freshest version in the set, t_k cannot unlock any data item until t_j unlocks e. Therefore, neither transaction can unlock any data items, and one must be aborted. If the versions are in different version sets, then there exists a read step $R_m(e)$ such that $W_k(e) \prec R_m(e) \prec W_j(e)$ or $W_j(e) \prec R_m(e) \prec W_k(e)$ in R. Clearly, $W_j(e) \prec R_m(e) \prec W_k(e)$ yields a cycle in the order graph and therefore cannot occur. Therefore, $W_k(e) \prec R_m(e) \prec W_j(e)$ in R. By the write-order version control protocol, $R_i(e)$ must also precede $W_j(e)$, or $R_i(e)$ would not be assigned $W_k(e)$ (i.e.., $W_k(e)$ would not be the freshest version). Therefore, t_i would precede t_k in the wrg of R, the wrg of S, the order graph and S. Therefore, $R_i(e)$ could not read from $W_j(e)$ in S, a contradiction.

Case 3: There exists an arc from t_k to t_j in the wrg for S. This proof is isomorphic to case 2, making the appropriate substitution of $W_k(e') \prec R_j(e')$ for $R_k(e') \prec W_j(e')$.

Case 4: There exists an arc from t_k to t_j in the fvg for S. Therefore, there exists some data item e' such that t_k and t_j both had versions in the same version set and t_j wrote the freshest version. Clearly, the fvg would contain a cycle if $W_k(e)$ and $W_j(e)$ were both also in the same version set, since $W_k(e)$ is the freshest version in that set. By arguments similar to those presented in case 1, $W_k(e)$ and $W_j(e)$ could also not be in distinct version sets.

Therefore, $R_i(e)$ cannot read the version written by $W_j(e)$. Therefore $R_i(e)$ cannot return a value other than that written by $W_k(e)$. This is a contradiction.

Therefore, all initial read steps return the same values in R and S. Therefore, all read steps return the same values in R and S. Therefore, all executions allowed by the L-VC protocol using the write-order version control protocol are view equivalent to a one-version serial execution of the top-level transactions. □

5 Conclusion

The concept of a nested transaction is valuable in modeling long-duration transactions. It is also clear that concurrency control techniques should be developed which exploit the different correctness criteria possessed by different transactions. However, in order to do this, a formal notion of multilevel correctness is needed.

We provide this formal notion by the defining transaction managers which are associated with each transaction. Each transaction manager is a set of constraints on the steps of each subtransaction and a condition on the execution order of the subtransactions themselves. Typically, such transaction managers are implemented as protocols. The problem with multilevel concurrency control is the interaction between these transaction managers. To solve this problem, we propose the intra-transaction concurrency control paradigm.

Under intra-transaction concurrency control, a transaction manager is only responsible for the initial reads and the final writes performed by the subtransactions which comprise the transaction associated with the transaction manager. It is assumed that all other operations are internal to the lower-level subtransactions, and therefore have other transaction managers to ensure their correctness. The intra-transaction concurrency control paradigm is used to derive a protocol for the class conflict predicate correct (CPC).

Although the constraints and conditions cover the standard conflict classes, there are many classes which are not covered. These classes, such as view-serializability-based classes and predicate-based classes [KS88], can also benefit from intra-transaction concurrency control. As for multilevel correctness criteria, intra-transaction concurrency control represents only one possible paradigm for exploiting the rich concurrency possible when multiple notions of correctness are combined in a single transaction.

In advanced database applications, the criteria for correctness for each transaction is unique. Thus, with complex interactions being performed by long-duration systems, acceptable performance and correctness cannot be obtained without some notion of multilevel concurrency. However, multilevel concurrency control is not easy to implement. Therefore, intra-transaction concurrency control is needed as a mechanism to develop protocols acceptable for these applications. The concept of intra-transaction correct establishes a minimum level which can be used to define criteria acceptable for these applications.

We have not considered the relationship between our protocols for concurrency and the issue of failure recovery in this paper. Clearly, the standard undo/redo approach is not sufficient. Rather, we need to employ a semantically richer technique such as the compensating transaction model of [KLS90].

References

[AEA90] D. Agrawal and A. El Abbadi. Locks with constrained sharing. In *9th ACM Symposium on Principles of Database Systems*, 1990.

[AS89] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *SIGMOD International Conference on Management of Data*, 1989.

[BBG89] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 1989.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BKK85] F. Bancilhon, W. Kim, and H. Korth. A model of CAD transactions. In *Proceedings of the 11th International Conference on Very Large Databases*, 1985.

[BR90] B.R. Badrinath and K. Ramamritham. Performance evaluation of semantics-based multilevel concurrency control protocols. In *SIGMOD International Conference on Management of Data*, 1990.

[DG87] H.C. Du and S. Ghanta. A framework for efficient IC/VLSI CAD databases. In *Proceedings of the 13th International Conference on Very Large Databases*, 1987.

[HL82] R. Haskin and R. Lorie. On extending the functions of a relational database system. In *SIGMOD International Conference on Management of Data*, 1982.

[KKB88] H. Korth, W. Kim, and F. Bancilhon. On long-duration CAD transactions. *Information Sciences*, October 1988.

[KLMP84] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A transaction mechanism for engineering design databases. In *Proceedings of the 10th International Conference on Very Large Databases*, 1984.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. Formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Databases*, 1990. to appear.

[Kor83] H. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[KS88] H. Korth and G. Speegle. Formal model of correctness without serializability. In *SIGMOD International Conference on Management of Data*, 1988.

[KS90] H. Korth and G. Speegle. Long duration transactions in software design projects. In *6th International Conference on Data Engineering*, 1990.

[Mos85] J. E. B. Moss. *Nested Transactions - An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

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

[Ree83] D. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

Appendix A: Multilevel Correctness of Executions under Intra-Transaction Concurrency Control

Theorem 1: *If all transaction managers enforce the constraints* $W_\alpha(e) \prec R_\beta(e) \rightarrow (t_\alpha, t_\beta) \in R$, *and* $R_\alpha(e) \prec W_\beta(e) \rightarrow (t_\alpha, t_\beta) \in R$, *and the condition that* R *is acyclic, and an execution* (R, X) *of a transaction* $t = (T, P, I, O)$ *is allowed under intra-transaction concurrency control then it is view equivalent to a multilevel correct execution of* t.

Proof:

The proof proceeds by induction on the number of levels in the nested transaction hierarchy. The base case is where the transaction consists of database access steps only. By the definition of one-level correct, all executions of such a transaction are one-level correct, and therefore are multilevel correct.

The inductive hypothesis is that for all transactions of k levels or fewer, every execution allowed under intra-transaction concurrency control is view equivalent to some multilevel correct execution of the transaction. It must be shown that for all transactions of $k+1$ levels, the theorem holds.

Let $t = (T, P, I, O)$ be a transaction in the NT/PV model with at least $k+1$ levels such that the execution (R, X) of t is correct under intra-transaction concurrency control. We must show that the execution is view equivalent to some multilevel correct execution of t. Therefore, we must show that the execution is view equivalent to a one-level correct execution of t, consisting of a multilevel correct execution for each of the subtransactions of t. The inductive hypothesis proves that the restriction of the execution to each top-level subtransaction is view equivalent to a multilevel correct execution of the subtransaction, since they can have no more than k levels each.

Therefore it remains to show that for some execution $R' \in t.TM$, every read step in R'_∞ returns the value returned by the read step in R_∞. Let $R_i^I(e)$ denote the initial read of data item e by transaction t_i and $W_i^F(e)$ denote the final write of data item e by transaction t_i. Below, an algorithm is given to construct the required execution (R', X') of t. The correctness of the algorithm is proved subsequently.

1. Order all initial reads and final writes of all subtransactions of t such that the value returned by an initial read step on data item e is the value written by the last write step on data item e which precedes it.

2. For all subtransactions $t_i \in t.T$, insert non-initial read steps into the execution as close to the beginning of the execution as possible, such that

- The order of the steps in the equivalent multilevel correct execution of subtransaction t_i
- If there exists an ordered pair $(t_j, t_i) \in R$ and there exists a constraint of the form $R_\alpha \prec A_\beta \rightarrow (t_\alpha, t_\beta) \in R$ and there exists a step $A_j(e)$ already in R'_∞, then the non-initial read must follow the step $A_j(e)$. Note that the step $A_j(e)$ may be either a read or a write step.
- If the insertion of the non-initial read step, $R_i(e)$ creates a cycle in R', then, while preserving their relative order to each other, move the following set S of steps to immediately after $R_i(e)$,
  - If $A_j(e)$ follows $R_i^I(e)$ and precedes the inserted non-initial read step $R_i(e)$ and there exists the constraint $A_\alpha \prec R_\beta \rightarrow (t_\alpha, t_\beta) \in R$ and $A_j(e)$ is the same type of database access as A_α and (t_j, t_i) is in the cycle, then $A_j(e)$ is in S.
  - If $A_m(e')$ is between some element of S and $R_i(e)$ and there exists some element $A_n(e'')$ of S such that $(t_n, t_m) \in R$, then include $A_m(e')$.

3. For all subtransactions $t_i \in t.T$, insert non-final write steps into the execution as close to the end of the execution as possible, such that
   - The order of the steps in the equivalent multilevel correct execution of subtransaction t_i
   - If there exists an ordered pair $(t_i, t_j) \in R$ and there exists a constraint of the form $A_\alpha \prec W_\beta \rightarrow (t_\alpha, t_\beta) \in R$ and there exists a step $A_j(e)$ already in R'_∞, then the non-final write must precede the step. Note again that the step $A_j(e)$ may be either a read or a write step.
   - If the insertion of the non-final write step, $W_i(e)$ creates a cycle in R', then, while preserving their relative order to each other, move the following set S of steps to immediately before $W_i(e)$,
     - If $A_j(e)$ precedes $W_i^F(e)$ and follows the inserted non-final write step $W_i(e)$ and there exists the constraint $W_\alpha \prec A_\beta \rightarrow (t_\alpha, t_\beta) \in R$ and $A_j(e)$ is the same type of database access as A_β and (t_i, t_j) is in the cycle, then $A_j(e)$ is in S.
     - If $A_m(e')$ is between some element of S and $W_i(e)$ and there exists some element $A_n(e'')$ of S such that $(t_m, t_n) \in R$, then include $A_m(e')$.

It now must be shown that the execution (R', X') exists (that is, the construction specified by the algorithm is possible), that it is in $t.TM$ and that it is view equivalent to (R, X). The first two properties are shown in Part 1; view equivalence is proved in Part 2.

Part 1: Proof that the execution (R', X') exists and R' is acyclic.

The proof of part 1 proceeds by induction on the number of non-initial read and final write steps in the execution.

If there are no non-initial read steps or final write steps, then the execution (R', X') consists of only initial reads and final writes. Such an execution must exist since, in intra-transaction concurrency control,

the initial reads and final writes are ordered according to $t.TM$. Likewise, the execution is acyclic by the restrictions on the condition of $t.TM$.

The inductive hypothesis is that for an execution (R, X) with κ non-initial reads and non-final writes, there exists an execution (R', X') constructed as above, and R' is acyclic.

Assume that (R, X) has $\kappa+1$ non-initial reads and non-final writes. Assume that the $\kappa+1st$ such step of the execution is a non-initial read step and it cannot be inserted in the execution (the case for the non-final writes is similar). Let t_i denote the transaction for this step. Therefore, there must exist a step $A_i(e')$ which must execute after $R_i(e)$ by the transaction order of t_i, but there exists another step $A_j(e)$ such that $A_j(e)$ follows $A_i(e')$ and there exists an ordered pair $(t_j, t_i) \in R$ and there exists a constraint of the form $A_\alpha \prec W_\beta \rightarrow (t_\alpha, t_\beta) \in R$. In other words, either the conflict semantics of the execution or the transaction order must be violated.

By the inductive hypothesis, the definition of a constraint and the existence of $(t_j, t_i) \in R$, it must be the case that the initial read of e by transaction t_i must also follow $A_j(e)$. Therefore, $R_i^I(e)$ must follow $A_i(e')$. However, since $R_i(e)$ must precede $A_i(e')$, then it must precede $R_i^I(e)$. By the definition of initial read, this is not possible. Therefore non-initial reads can be inserted into the execution.

Now assume that the insertion of the $\kappa + 1st$ non-initial read or non-final write of the execution causes a cycle in (R', X'). As before, assume the step is a non-initial read step of transaction t_i. By the inductive hypothesis, such a cycle must include t_i. Consider the case where all steps $A_j(e)$ which satisfy constraints which cause a cycle in R' either precede $R_i^I(e)$ or follow $R_i(e)$. Therefore, all pairs of transactions added to R' by the insertion of $R_i(e)$ to R already exist in R because of the existence of $R_i^I(e)$. By the inductive hypothesis, no such cycles can exist.

Now consider the case where R'_∞ is $R_i^I(e) \ldots A_j(e) \ldots R_i(e)$. By the construction of (R', X'), if a cycle exists, a set of the data items which follow $R_i^I(e)$ but precede $R_i(e)$ will be moved to follow $R_i(e)$. Note that the order of these steps will be preserved. Denote the sequence of steps moved as M. Denote those steps not moved as M'. After the move, R'_∞ is $R_i^I(e) \ldots R_i(e) M$. Assume this move creates a cycle. Thus, moving the elements in M adds a set of ordered pairs to R'. These ordered pairs must be of the form $(t_{M'}, t_M)$ where t_M is a transaction which has a step in M and $t_{M'}$ is a transaction which has a step in M, denoted M'.

By the inductive hypothesis of Part 1, if no new elements are added by moving the steps, then the move does not generate a cycle. Assume some elements are added to R' by the move. Thus, there must be a constraint of the form $A_\alpha \prec A_\beta \rightarrow (t_\alpha, t_\beta) \in R$ and some access in M', denoted $A_{m'}$, is the same access as A_α, and some access in M, denoted A_m, is the same access as A_β and $(t_{m'}, t_m) \notin R'$ before the move. Therefore, it must be the case that $A_{m'}$ followed A_m before the move. It also must be the case that $(t_m, t_{m'}) \notin R'$ before the move, or $A_{m'}$ would be in M. However, since all such elements added R' are of the form $(t_{M'}, t_M)$, $(t_m, t_{m,'})$ cannot be added, so no cycle can be created.

The proof for the non-final write steps is similar. Therefore, we conclude that (R', X') exists and R' is

acyclic under the constraints for $t.TM$. Thus, (R', X') is a one-level correct execution for t.

Part 2: (R', X') is view equivalent to (R, X).

Since both executions are over the same subtransactions, all that needs to be shown is that the read steps return the same values in both executions.

Assume this is not the case. Therefore, there exists some read step in subtransaction t_i such that $R_i(e)$ returns a different value in the two executions. Under intra-transaction concurrency control, the value returned by a read step in (R, X) is either the value returned by the initial read step or the value written by the write step which precedes it in the equivalent multilevel correct execution of t_i.

By step 1 of the construction of R', all of the initial read steps returned the same values in both executions. Now assume that a final write step was moved past an initial read step, or that an initial read step was moved ahead of a final write step in the execution of R'. We will discuss only the case of the write step being moved past the read step, because the other case is similar. Thus, the movement is caused by the insertion of a read step into the execution. Let $R_i^I(e)$ be the initial read step which returns the value of $W_j^F(e)$ in R. Therefore, there is no constraint of the form $W_\alpha \prec R_\beta \rightarrow (t_\alpha, t_\beta) \in R$, because if there were, $R_i^I(e)$ would also have been moved. Likewise, there can be no constraint of the form $R_\alpha \prec R_\beta \rightarrow (t_\alpha, t_\beta) \in R$, since $R_j^I(e)$ must also precede $R_i^I(e)$. This would cause $(t_j, t_i) \in R$, and $R_i^I(e)$ would be moved. However, this means the insertion of a read step can only conflict with steps which follow it in the execution. Since the initial read step also conflicts with those steps, no cycle can be generated, and thus $R_i^I(e)$ follows $W_j^F(e)$.

Now consider the case where the step $W_n^F(e)$ is moved between $R_i^I(e)$ and $W_j^F(e)$, where $R_i^I(e)$ returns the value of $W_j^F(e)$ in R. Clearly, no constraints of the form $W_\alpha \prec W_\beta \rightarrow (t_\alpha, t_\beta) \in R$, can be in the transaction manager because $W_j^F(e)$ would also be moved. However, it is also the case that no constraints of the form $R_\alpha \prec W_\beta \rightarrow (t_\alpha, t_\beta) \in R$, can be in the transaction manager either. Recall that R_∞ is $R_a^I(e) \ldots A_b(e) \ldots R_a(e)$ for anything to be moved. However, in this case we also know that R_∞ is $R_a^I(e'') \ldots R_j^I(e) \ldots R_n^I(e) \ldots A_b(e) \ldots W_j^F(e) \ldots W_n^F(e) \ldots R_a(e'') \ldots R_i^I(e)$. Therefore, if the constraint $R_\alpha \prec W_\beta \rightarrow (t_\alpha, t_\beta) \in R$, held, then there would be a cycle in R', which contradicts the construction of R'. Since all transaction managers must support this condition, this is a contradiction. Therefore, the initial reads return the same values in each execution.

Non-initial read steps in R return the value written by the most recent write step in its own transaction, or the Let $R_i(e)$ be a step of transaction t_i such that $R_i(e)$ returns a different value in R and R'. Therefore, R'_∞ is either $R_i^I(e) \ldots W_j(e) \ldots R_i(e)$ or $R_i^I(e) \ldots W_i(e) \ldots W_j(e) \ldots R_i(e)$. In either case, a cycle must exist in R'_∞ from the constraints $R_\alpha \prec W_\beta \rightarrow (t_\alpha, t_\beta) \in R$, and $W_\alpha \prec R_\beta \rightarrow (t_\alpha, t_\beta) \in R$, which all transaction managers must enforce. This contradicts the construction of R'.

Thus, all read steps in the execution R return the same values as in a one-level correct execution. Therefore, (R', X') is view equivalent to (R, X). □