Concurrency Control in Advanced Database Applications

Naser S. Barghouti and Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027

29 May 1990

Abstract

Concurrency control has been thoroughly studied in the context of traditional database applications such as banking and airline reservations systems. There are relatively few studies, however, that address the concurrency control issues of advanced database applications such as CAD/CAM and software development environments. The concurrency control requirements in such applications are different from those in conventional database applications; in particular, there is a need to support non-serializable cooperation among users whose transactions are long-lived and interactive, and to integrate concurrency control mechanisms with version and configuration control. This paper outlines the characteristics of data and operations in some advanced database applications, discusses their concurrency control requirements, and surveys the mechanisms proposed to address these requirements.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*concurrency*; *transaction processing*; H.2.8 [**Database Management**]: Applications; D.2.6 [**Software Engineering**]: Programming Environments—*interactive*; D.2.9 [**Software Engineering**]: Management—*programming teams*

General Terms: Algorithms, Design, Management

Additional Key Words and Phrases: Concurrency control, design environments, advanced database applications, object-oriented databases, extended transaction models, cooperative transactions, long transactions, relaxing serializability

INTRODUCTION

Many advanced computer-based applications, such as computer-aided design and manufacturing (CAD/CAM), network management, financial instruments trading, medical informatics, office automation, and software development environments (SDEs), are data-intensive in the sense that they generate and manipulate large amounts of data (e.g., all the software artifacts in an SDE). It is desirable to base these kinds of application systems on data management capabilities similar to those provided by database management systems (DBMSs) for traditional data processing. These capabilities include adding, removing, retrieving and updating data from on-line storage, and maintaining the consistency of the information stored in a database. Consistency in a DBMS is maintained if every data item satisfies specific consistency constraints, which are typically implicit in data processing, although known to the implementors of the applications, and programmed into atomic units called *transactions* that transform the database from one consistent state to another. Consistency can be violated by concurrent access by multiple transactions to the same data item. A DBMS solves this problem by enforcing a concurrency control policy that allows only consistency-preserving schedules of concurrent transactions to be executed.

We use the term *advanced database applications* to describe application systems, such as the ones mentioned above, that utilize DBMS capabilities. They are termed *advanced* to distinguish them from traditional database applications, such as banking and airline reservations systems, in which the nature of the data and the operations performed on the data are amenable to concurrency control mechanisms that enforce the classical transaction model. Advanced applications, in contrast, place different kinds of consistency constraints, and, in general, the classical transaction model is not applicable. For example, network management, financial instruments trading and medical informatics may require real-time processing, while CAD/CAM, office automation and SDEs involve long interactive database sessions and cooperation among multiple database users. Conventional concurrency control mechanisms appropriate for traditional applications are not applicable "as is" in these new domains. We are concerned in this paper with the latter class of advanced applications, which involve computer-supported cooperative work, and their requirements are elaborated in section 5.

Some researchers and practitioners question the adoption of terminology and concepts from on-line transaction processing (OLTP) systems for advanced applications. In particular, these researchers feel that the terms ‘‘long transactions’’ and ‘‘cooperating transactions’’ are an inappropriate and misleading use of the term ‘‘transaction’’, since they do not carry the atomicity and serializability properties of OLTP transactions. We agree that atomicity, serializability and the corresponding OLTP implementation techniques are not appropriate for advanced applications. However, the term ‘‘transaction’’ seems to conjure up a nice intuition regarding the needs for consistency, concurrency control and fault recovery, and that some basic OLTP supports such as locks, versions and validation provide a good starting point for implementation of ‘‘long transactions’’ and ‘‘cooperating transactions’’ mechanisms. In any case, nearly all the relevant literature uses the term ‘‘transaction’’, so it is necessary that we do likewise in our survey.

The goals of this paper are to provide a basic understanding of how concurrency control in advanced database applications involving computer-supported cooperative work differs from that in traditional data processing applications, to outline some of the mechanisms used to control concurrent access in these advanced applications, and to point out some problems with these mechanisms. We assume that the reader is somewhat familiar with database concepts, but do not assume in-depth understanding of transaction processing and concurrency control issues. Throughout the paper, we try to define the concepts that we use, give practical examples of the formal concepts, and explain the various mechanisms at an intuitive level rather than a detailed technical level.

The paper is organized as follows. We start with an example to motivate the need for new concurrency control mechanisms. Section 2 describes the data handling requirements of advanced database applications and shows why there is a need for capabilities like those provided by DBMSs. Section 3 gives a brief overview of the consistency problem in traditional database applications and explains the concept of serializability. Section 4 presents the main serializability-based concurrency control mechanisms. Readers who are familiar with conventional concurrency control schemes could skip sections 3 and 4. Section 5 enumerates the concurrency control requirements of advanced database applications. The discussion in that section focuses on software development environments, although many of the problems of CAD/CAM and office automation systems are similar. Sections 6, 7, and 8 survey the various concurrency control mechanisms proposed for this class of advanced database applications. Section 9 discusses some of the shortcomings of these mechanisms.

1 A MOTIVATING EXAMPLE

We motivate the need for extended concurrency control policies by a simple example from the software development domain. Variants of the following example will be used throughout the paper to demonstrate the various concurrency control models.

Two programmers, John and Mary, are working on the same software project. The project consists of four modules A, B, C and D. Modules A, B and C consist of procedures and declarations that comprise the main code of the project; module D is a library of procedures called by the procedures in modules A, B and C. Figure 1 depicts the organization of the project.

When testing the project, two bugs are discovered. John is assigned the task of fixing one bug that is suspected to be in module A, so he "reserves" A and starts working on it. Mary's task is to explore a possible bug in the code of module B and so she starts browsing B after "reserving" it. After a while, John finds out that there is a bug in A caused by bugs in some of the procedures in the library module, so he "reserves" module D to modify a few things in it. After modifying a few procedures in D, John proceeds to compile and test the modified code.

Mary finds a bug in the code of module B and modifies various parts of the module to fix it. Mary now wants to test the new code of B. She is not concerned with the modifications that John

Figure 1: Organization of example project

made in A because module A is unrelated to module B, but she wants to access the modifications that John made in module D because the procedures in D are called in module B and the modifications that John has made to D might have introduced inconsistencies with the code of module B. But since John is still working on modules A and D, Mary will have to access module D at the same time that John is modifying it.

In the above example, if the traditional concurrency control scheme of two-phase locking was used, for example, John and Mary would not be able to access the modules in the manner described above. They would be allowed to concurrently lock module B and module A, respectively, since they work in isolation on these modules. Both of them, however, need to work cooperatively on module D and thus neither of them can lock it. Even if the locks were at the granularity of procedures, they would still have a problem because both John and Mary might need to access the same procedures, in order to recompile D, for example, before releasing the locks (after reaching a satisfactory stage of modification of the code such as the completion of unit testing). Other traditional concurrency control schemes would not solve the problem because they would require the serialization of Mary's work with John's.

The problem might be solved by supporting parallel versions of module D (Mary would access the last compiled version while John works on a new version), but this requires Mary to later retest her code after the new version of D is released. What is needed is a flexible concurrency control scheme that allows cooperation among John and Mary. In the rest of this paper, we explain the basic concepts behind traditional concurrency control mechanisms, show how these mechanisms do not support the needs of advanced applications, and describe several concurrency control mechanisms that provide some of the necessary support.

2 ADVANCED DATABASE APPLICATIONS

Many large multi-user software systems, such as software development environments, generate and manipulate large amounts of data, e.g., in the form of source code, object code, documentation, test suites, etc. Traditionally, users of such systems managed the data they generate either manually or by the help of special-purpose tools. For example, programmers working on a large-scale software project use system configuration management (SCM) tools such as Make [Feldman 79] and RCS [Tichy 85] to manage the configurations and versions of the programs they are developing. Releases of the finished project are stored in different directories manually. The only common interface between all these tools is the file system, which stores project parts in text or binary files regardless of their internal structures. This significantly limits the ability to manipulate these objects in desirable ways, causes inefficiencies as far as storage of collections of objects is concerned, and leaves data, stored as a collection of related files, susceptible to corruption due to incompatible concurrent access.

More recently, researchers have attempted to utilize database technology to uniformly manage all the objects belonging to a system. Design environments, for example, need to store the objects they manipulate (design documents, circuit layouts, programs, *etc.*) in a database and have it managed by a DBMS for several reasons [Bernstein 87; Dittrich et al. 87; Nestor 86; Rowe and Wensel 89]:

1. Data integration: providing a single data management and retrieval interface for all tools accessing the data.
2. Application orientation: organizing data items into structures that capture much of the semantics of the intended applications.
3. Data integrity: preserving consistency and recovery, to ensure that all the data satisfy the integrity constraints required by the application.
4. Convenient access: providing a powerful query language to access sets of data items at a time.
5. Data independence: hiding the internal structure of data from tools so that if that structure is changed, it will have a minimal impact on the applications using the data.

Since there are numerous commercial database systems available, several projects have tried to use them in advanced applications. Researchers discovered quite rapidly, however, that even the most sophisticated of today's DBMSs are inadequate for requirements of advanced applications [Korth and Silberschatz 86; Bernstein 87]. One of the shortcomings of traditional general-purpose databases is the inability to provide flexible concurrency control mechanisms that can support the needs of users in advanced applications. To understand the reasons behind this, we need to explain the concept of serializable transactions that is central to all conventional concurrency control mechanisms.

3 THE CONSISTENCY PROBLEM IN CONVENTIONAL DATABASE SYSTEMS

Database consistency is maintained if each data item in the database satisfies some application-specific consistency constraints. For example, in a distributed airline reservation system, one consistency constraint might be that each seat on a flight can be reserved by only one passenger. It is often the case, however, that not all consistency constraints are known before hand to the designers of general-purpose DBMSs, because of the lack of information about the computations in potential applications.

Given the lack of knowledge about the application-specific semantics of database operations, and the need to design general mechanisms that cut across many potential applications, the best a DBMS can do is to abstract all operations on a database to be either a read operation or a write operation, irrespective of the particular computation. Then it can guarantee that the database is always in a consistent state with respect to reads and writes regardless of the semantics of the particular application. Ignoring the possibility of bugs in the DBMS program and the application program, inconsistent data then results from two main sources: (1) software or hardware failures such as bugs in the operating system or a disk crash in the middle of operations, and (2) concurrent access of the same data item by multiple users or programs.

3.1 The Transaction Concept

To solve these problems, the operations performed by a process that is accessing the database are grouped into sequences called *transactions* [Eswaran et al. 76]. Thus, users would interact with a DBMS by executing transactions. In traditional DBMSs, transactions serve three distinct purposes [Lynch 83]: (1) they are logical units that group together operations that comprise a complete task; (2) they are atomicity units whose execution preserves the consistency of the database; and (3) they are recovery units that ensure that either all the steps enclosed within them are executed, or none are. It is thus by definition that if the database is in a consistent state before a transaction starts executing, it will be in a consistent state when the transaction terminates.

In a multi-user system, users execute their transactions concurrently, and the DBMS has to provide a concurrency control mechanism to guarantee that consistency of data is maintained in spite of concurrent accesses by different users. From the user's viewpoint, a concurrency control mechanism maintains the consistency of data if it can guarantee: (1) that each of the transactions submitted to the DBMS by a user eventually gets executed; and (2) that the results of the computation performed by each transaction are the same whether it is executed on a dedicated system or concurrently with other transactions in a multi-programmed system [Bernstein et al. 87; Papadimitriou 86].

Let us follow up on our previous example to demonstrate the concept of transactions. John and Mary are assigned the task of fixing two bugs that were suspected to be in modules A and B. The first bug is caused by an error in procedure p1 in module A, which is called by procedure p3 in module B (thus fixing the bug might affect both p1 and p3). The second bug is caused by

T_{John}	T_{Mary}
reserve(A)	
modify(p1)	
write(A)	
	reserve(A)
	modify(p2)
	write(A)
reserve(B)	
modify(p3)	
write(B)	
	reserve(B)
	modify(p4)
	write(B)

Time

Figure 2: Serializable schedule

an error in the interface of procedure p2 in module A, which is called by procedure p4 in B. John and Mary agree that John will fix the first bug and Mary will fix the second. John starts a transaction T_{John} and proceeds to modify procedure p1 in module A. After completing the modification, he starts modifying procedure p3 in module B. At the same time, Mary starts a transaction T_{Mary} to modify procedure p2 in module A and procedure p4 in module B.

Although T_{John} and T_{Mary} are executing concurrently, their outcomes are expected to be the same, had each of them been executed on a dedicated system. The overlap between T_{Mary} and T_{John} results in a sequence of actions from both transactions, called a *schedule*. Figure 2 shows an example of a schedule made up by interleaving operations from T_{John} and T_{Mary}. A schedule that gives each transaction a consistent view of the state of the database is considered a consistent schedule. Consistent schedules are a result of synchronizing the concurrent operations of users by allowing only those operations that maintain consistency to be interleaved.

3.2 Serializability

Let us give a more formal definition of a consistent schedule. Since transactions are consistency preserving units, if a set of transactions T_1, T_2, ..., T_n are executed serially (i.e., for every i= 1 to n-1, transaction T_i is executed to completion before transaction T_{i+1} begins), consistency is preserved. Thus, every serial execution (schedule) is correct by definition. We can then establish that a serializable execution (one that is equivalent to a serial execution) is also correct. From the perspective of a DBMS, all computations in a transaction either read or write a data item from the database. Thus, two schedules S1 and S2 are said to be computationally equivalent if [Korth and Silberschatz 86]:

1. The set of transactions that participate in S_1 and S_2 are the same.
2. For each data item Q in S_1, if transaction T_i executes read(Q) and the value of Q read by T_i is written by T_j, then the same will hold in S_2 (i.e., read-write synchronization).
3. For each data item Q in S_1, if transaction T_i executes the last write(Q) instruction, then the same holds also in S_2 (i.e., write-write synchronization).

For example, the schedule shown in figure 2 is equivalent to the serial schedule T_{John}, T_{Mary} (execute T_{John} to completion and then execute T_{Mary}) because: (1) the set of transactions in both schedules are the same; (2) both data items A and B read by T_{Mary} are written by T_{John} in both schedules; and (3) T_{Mary} executes the last write(A) operation and the last write(B) operation in both schedules.

The consistency problem in conventional database systems reduces to that of testing for serializable schedules because it is accepted that the consistency constraints are unknown. Each operation within a transaction is abstracted into either reading a data item or writing it. Achieving serializability in DBMSs can thus be decomposed into two subproblems: read-write synchronization and write-write synchronization, denoted rw and ww synchronization, respectively [Bernstein and Goodman 81]. Accordingly, concurrency control algorithms can be categorized into those that guarantee rw synchronization, those that are concerned with ww synchronization, and those that integrate the two. Rw synchronization refers to serializing transactions in such a way so that every read operation reads the same value of a data item as that it would have read in a serial execution. Ww synchronization refers to serializing transactions so that the last write operation of every transaction leaves the database in the same state as it would have left it in some serial execution. Rw and ww synchronization together result in a consistent schedule.

When more than one transaction is involved in reading and writing the same object at the same time, one of the transactions is guaranteed to complete its task while other transactions must be prevented from executing the conflicting operations until the continuing transaction is complete and a consistent state is guaranteed. Thus, even though a DBMS may not have any information about application-specific consistency constraints, it can guarantee consistency by allowing only serializable executions of concurrent transactions. This concept of serializability is central to all the concurrency control mechanisms described in the next section. If more semantic information about transactions and their operations were available, schedules that are not serializable but that do maintain consistency can be produced. That is exactly what the extended transaction mechanisms discussed later try to achieve.

4 TRADITIONAL APPROACHES TO CONCURRENCY CONTROL

In order to understand why conventional concurrency control mechanisms are too restrictive for advanced applications, it is necessary to be familiar with the basic ideas of the main serializability-based concurrency control mechanisms that have been proposed for, and implemented in, conventional database systems. Most of the mechanisms follow one of five main approaches to concurrency control: two-phase locking, which is the most popular example of locking protocols, timestamp ordering, optimistic concurrency control, multiversion concurrency, and nested transactions, which is relatively orthogonal to the first four mechanisms. In this section, we briefly describe these five approaches. For a comprehensive discussion and survey of the topic, the reader is referred to [Bernstein and Goodman 81] and [Kohler 81].

4.1 Locking Mechanisms

4.1.1 Two-Phase Locking

The two-phase locking mechanism (2PL) introduced by Eswaran *et al.* is now accepted as the standard solution to the concurrency control problem in conventional database systems. 2PL guarantees serializability in a centralized database when transactions are executed concurrently. The mechanism depends on well-formed transactions, which (1) do not relock entities that have been locked earlier in the transaction, and (2) are divided into a growing phase, in which locks are only acquired, and a shrinking phase, in which locks are only released [Eswaran et al. 76]. During the shrinking phase, a transaction is prohibited from acquiring locks. If a transaction tries during its growing phase to acquire a lock that has already been acquired by another transaction, it is forced to wait. This situation might result in deadlock if transactions are mutually waiting for each other's resources.

2PL allows only a subset of serializable schedules. In the absence of information about how and when the data items are accessed, however, 2PL is both necessary and sufficient to ensure serializability by locking [Yannakakis 82]. If we have prior knowledge about the order of access of data items, which is often the case in advanced applications, we can construct locking protocols that are not 2PL but ensure serializability. One such protocol is the tree protocol, which can be applied if there is a partial ordering on the set of items that are accessed by concurrent transactions. To illustrate this protocol, assume that a third programmer, Bob, joined the programming team of Mary and John and is now working with them on the same project. Suppose that Bob, Mary and John want to modify modules A and B concurrently in the manner depicted in schedule S1 of figure 3. The tree protocol would allow this schedule to execute because it is serializable (equivalent to T_{Bob} T_{John} T_{Mary}) even though it does not follow the 2PL protocol (because T_{John} releases the lock on A before it acquires the lock on B). It is possible to construct S1 because all of the transactions in the example access (write) A before B. This information about the access patterns of the three transactions was used to construct the non-2PL schedule shown in the figure. This example demonstrates why 2PL is in fact not appropriate for advanced applications.

Schedule S1:

	T_{John}	T_{Mary}	T_{Bob}
|	lock(A)		
|	read(A)		
|	modify(A)		
|	write(A)		
|	unlock(A)		
|		lock(A)	
|		read(A)	
|		modify(A)	
|		write(A)	
|			lock(B)
|			read(B)
|			modify(B)
|			write(B)
|			unlock(B)
|	lock(B)		
|	read(B)		
|	modify(B)		
|	write(B)		
|	unlock(B)		
|		lock(B)	
|		read(B)	
|		modify(B)	
|		write(B)	
|		unlock(A)	
|		unlock(B)	
V			
Time			

Figure 3: Serializable but not 2PL schedule

4.2 Timestamp Ordering

One of the problems of locking mechanisms is deadlock, which occurs when two or more transactions are mutually waiting for each other's resources. This problem can be solved by assigning each transaction a unique number, called a timestamp, chosen from a monotonically increasing sequence, which is often a function of the time of the day [Kohler 81]. Using timestamps, a concurrency control mechanism can totally order requests from transactions according to the transactions' timestamps [Rosenkrantz et al. 78]. The mechanism forces a transaction requesting to access a resource that is being held by another transaction to either (1) wait until the other transaction that has hold of the requested resource at that time terminates, (2) abort itself and restart if it cannot be granted the request, or (3) preempt the other transaction and get hold of the resource. A scheduling protocol decides which one of these three actions to take after comparing the timestamp of the requesting transaction with the timestamps of conflicting transactions. The protocol must guarantee that a deadlock situation will not arise.

Two of the possible alternative scheduling protocols used by timestamp-based mechanisms are: (1) the WAIT-DIE protocol, which forces a transaction to wait if it conflicts with a running transaction whose timestamp is more recent, or to die (abort and restart) if the running transaction's timestamp is older; and (2) the WOUND-WAIT protocol, which allows a transaction to wound (preempt by suspending) a running one with a more recent timestamp, or forces the requesting transaction to wait otherwise. Locks are used implicitly in this technique since some transactions are forced to wait as if they were locked out.

4.3 Multiversion Timestamp Ordering

The timestamp ordering mechanism above assumes that only one version of a data item exists. Consequently, only one transaction can access a data item at a time. This mechanism can be improved in the case of read-write synchronization by allowing multiple transient versions of a data item to be read and written by different transactions, as long as each transaction sees a consistent set of versions for all the data items that it accesses. This is the basic idea of the multiversion scheme introduced by Reed [Reed 78]. In Reed's mechanism, each transaction is assigned a unique timestamp when it starts; all operations of the transaction are assigned the same timestamp. For each data item *x* there is a set of read timestamps and a set of <*write timestamp, value*> pairs, called transient versions.

The existence of multiple versions eliminates the need for write-write synchronization since each write operation produces a new version and thus can not conflict with another write operation. The only possible conflicts are those corresponding to read-from relationships [Bernstein et al. 87], as demonstrated by the following example.

Let R(x) be a read operation with timestamp TS(R). R(x) is processed by reading the value of the version of *x* whose timestamp is the largest timestamp smaller than TS(R). TS(r) is then added to the set of read timestamps of item *x*. Similarly, let W(x) be a write operation that assigns value *v* to item *x*, and let its timestamp be TS(W). Let interval(W) be the interval from TS(W) to the smallest timestamp of a version of *x* greater than TS(W) (i.e., a version of *x* that was written by another transaction whose timestamp is more recent than TS(W)). A situation like this occurs because of delays in executing operations within a transaction (the write operation might have been the last operation after many other operations in the same transaction). Because of those delays, an operation O_i belonging to transaction T_i might be executed after T_i had started by a period of time. In the meanwhile, other operations from a more recent transaction might have been performed. If any read timestamps lies in the interval (i.e., a transaction has already read a value of *x* written by a more recent write operation than W), then W is rejected (and the transaction is aborted). Otherwise, W is allowed to create a new version of *x*.

4.4 Optimistic Non-Locking Mechanisms

In many applications, locking has been found to constrain concurrency and to add an unnecessary overhead. The locking approach has the following disadvantages [Kung and Robinson 81]:

1. Lock maintenance represents an unnecessary overhead for read-only transactions, which do not affect the integrity of the database.
2. Most of the general-purpose deadlock-free locking mechanisms work well only in some cases but perform rather poorly in other cases. There are no locking mechanisms that provide high concurrency in all cases.
3. When large parts of the database resides on secondary storage, locking of objects that are accessed frequently (referred to as congested nodes), while waiting for secondary memory access, causes a significant decrease in concurrency.
4. Not permitting locks to be unlocked until the end of the transaction, which although not required is always done in practice to avoid cascaded aborts, decreases concurrency.
5. Most of the time it is not necessary to use locking to guarantee consistency since most transactions do not overlap; locking may be necessary only in the worst case.

To avoid these problems, Kung and Robinson presented the concept of "optimistic" concurrency control by introducing two families of concurrency control mechanisms (*serial validation* and *parallel validation*) that do not use locking. They require each transaction to consist of two or three phases: a read phase, a validation phase and possibly a write phase. During the read phase, all writes take place on local copies (also referred to as transient versions) of the records to be written. Then, if it can be established during the validation phase that the changes the transaction made will not cause loss of integrity, i.e., that they are serializable with respect to all committed transactions, the local copies are made global and thus accessible to other transactions in the write phase.

Validation is done by assigning each transaction a timestamp at the end of the read phase and synchronizing using timestamp ordering. The correctness criteria used for validation are based on the notion of serial equivalence. Any schedule produced by this technique ensures that if transaction T_i has a timestamp less than the timestamp of transaction T_j then the schedule is equivalent to the serial schedule T_i followed by T_j. This can be ensured if any one of the following three conditions holds:

1. T_i completes its write phase before T_j starts its read phase.
2. The set of data items written by T_i does not intersect with the set of data items read by T_j, and T_i completes its write phase before T_j starts its write phase.
3. The set of data items written by T_i does not intersect with the set of data items read or written by T_j, and T_i completes its read phase before T_j completes its read phase.

Although optimistic concurrency control allows more concurrency under certain circumstances, it decreases concurrency when the read and write sets of the concurrent transactions overlap. For example, Kung and Robinson's protocol would cause one of transactions in the simple 2PL schedule in figure 2 to be rolled back and restarted. From the viewpoint of advanced applications, the use of rollback as the main mechanism of achieving serializability is a serious disadvantage. Since operations in advanced transactions are generally long-lived (e.g., compiling a module), rolling them back and restarting them wastes all the work that these operations did (the object code produced by compilation). The inappropriateness of rolling back a long transaction in advanced applications is discussed further in section 5.

4.5 Multiple Granularity Locking

All the concurrency control protocols described so far operate on individual data items to achieve synchronization of transactions. It is sometimes desirable, however, to able to access a set of data items as a single unit, e.g., to effectively lock each item in the set in one operation rather than having to lock each item individually. Gray *et al.* presented a *multiple granularity* concurrency control protocol, which aims to minimize the number of locks used while accessing sets of objects in a database [Gray et al. 75]. In their model, Gray *et al.* organize data items in a tree where items of small granularity are nested within larger ones. Each non-leaf item represents the data associated with its descendants. This is different from the tree protocol presented above in that the nodes of the tree (or graph) do not represent the order of access of individual data items but rather the organization of data objects. The root of the tree represents the whole database. Transactions can lock nodes explicitly, which in turn locks descendants implicitly. Two modes of locks were defined: *exclusive* and *shared*. An exclusive (X) lock excludes any other transaction from accessing (reading or writing) the node; a shared (S) lock permits other transaction to read the same node concurrently, but prevents any updating of the node.

To determine whether to grant a transaction a lock on a node (given these two modes), the transaction manager would have to follow the path from the root to the node to find out if any other transaction has explicitly locked any of the ancestors of the node. This is clearly inefficient. To solve this problem, a third kind of lock mode called *intention* lock mode was introduced [Gray 78]. All the ancestors of a node must be locked in intention mode before an explicit lock can be put on the node. In particular, nodes can be locked in five different modes. A non-leaf node is locked in intention-shared (IS) mode to specify that descendant nodes will be explicitly locked in shared (S) mode. Similarly, an intention-exclusive (IX) lock implies that explicit locking is being done at a lower level in an exclusive (X) mode. A shared and intention-exclusive (SIX) lock on a non-leaf node implies that the whole subtree rooted at the node is being locked in shared mode, and that explicit locking will be done at a lower level with exclusive-mode locks. A compatibility matrix for the five kinds of locks is defined as shown in figure 4. The matrix is used to determine when to grant lock requests and when to deny them.

Finally, a multiple granularity protocol based on the compatibility matrix was defined as follows:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Figure 4: Compatibility matrix of granularity locks

1. Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requester.
2. Before requesting an X, SIX or IX lock on a node, all ancestor nodes of the requested node must be held in SIX or IX mode by the requester.
3. Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of a transaction, it should not hold a lock on a node after releasing its ancestors.

The multiple granularity protocol increases concurrency and decreases overhead especially when there is a combination of short transactions with a few accesses and transactions that last for a long time accessing a large number of objects such as audit transactions that access every item in the database. The Orion object-oriented database system provides a concurrency control mechanism based on the multi-granularity mechanism described above [Kim et al. 88; Garza and Kim 88].

4.6 Nested Transactions

A transaction, as presented above, is a set of primitive atomic actions abstracted as read and write operations. Each transaction is independent of other transactions. In practice, there is a need to compose several transactions into one unit (i.e., one transaction) for two reasons: (1) to provide modularity; and (2) to provide finer grained recovery. The recovery issue maybe the more important one, but it is not addressed in detail here since the focus of this paper is on concurrency control. The modularity problem is concerned with preserving serializability when composing two or more transactions. One way to compose transactions is gluing together the primitive actions of all the transactions by concatenating the transactions in order into one big transaction. This preserves consistency but decreases concurrency because the resulting transaction is really a serial ordering of the subtransactions. Interleaving the actions of the transactions to provide concurrent behavior, on the other hand, can result in violation of serializability and thus consistency. What is needed is to execute the composition of transactions as a transaction in its own right, and to provide concurrency control within the transaction.

The idea of nested spheres of control, which is the origin of the nested transactions concept, was first introduced by Davies [Davies 73] and expanded by Bjork [Bjork 73]. Reed presented a comprehensive solution to the problem of composing transactions by formulating the concept of nested transactions [Reed 78]. A nested transaction is a composition of a set of subtransactions; each subtransaction can itself be a nested transaction. To other transactions, the top-level nested transaction is visible and appears as a normal atomic transaction. Internally, however, subtransactions are run concurrently and their actions are synchronized by an internal concurrency control mechanism. The more important point is that subtransactions fail and can be restarted or replaced by another subtransaction independently without causing the whole nested transaction to fail or restart. In the case of gluing the actions of subtransactions together, on the other hand, the failure of any action would cause the whole new composite transaction to fail. In Reed's design, timestamp ordering is used to synchronize the concurrent actions of subtransactions within a nested transaction. Moss designed a nested transaction system that uses locking for synchronization [Moss 85]. Moss's design also manages nested transactions in a distributed system.

Figure 5: Scheduling nested transactions

As far as concurrency is concerned, the nested transaction model presented above does not change the meaning of transactions (in terms of being atomic) and it does not alter the concept of serializability. The only advantage is performance improvement because of the possibility of increasing concurrency at the subtransaction level, especially in a multiprocessor system. To illustrate this, consider transactions T_{John} and T_{Mary} of figure 2. We can construct each as a nested transaction as shown in figure 5. Using Moss's algorithm, the concurrent execution of John's transaction and Mary's transaction will produce the same schedule presented in figure 2. Within each transaction, however, the two subtransactions can be executed concurrently, improving the overall performance.

It should be noted that combinations of optimistic concurrency control, multiversion objects, and nested transactions is the basis for many of the concurrency control mechanisms proposed for advanced database applications. To understand the reasons behind this, we have to address how the nature of data and computations in advanced database applications imposes new requirements on concurrency control. We explore these new requirements in the next section, and then present several approaches that take these requirements into consideration in rest of the paper.

5 CONCURRENCY CONTROL REQUIREMENTS IN ADVANCED DATABASE APPLICATIONS

Serializable executions of transactions with respect to reads and writes on the database, for example, are enforced in conventional transaction management schemes because of the lack of semantic knowledge about the application-specific operations, which leads to the inability to specify or check consistency constraints on data, as well as the desire to provide a general transaction processing scheme that does not depend on application details. But there is nothing that makes a non-serializable schedule inherently inconsistent. If enough information is known about the transactions running concurrently, a non-serializable but consistent schedule can be constructed.

In fact, equating the notions of consistency with serializability causes a significant loss of concurrency in advanced applications, where it is often possible to define specific consistency constraints. Several researchers have thus studied the nature of concurrent behavior in advanced applications, and have arrived at new requirements for concurrency control [Bancilhon et al. 85; Yeh et al. 87]:

1. **Supporting long transactions**: Long-lived operations on objects in design environments (such as compiling and circuit layout) imply that the transactions, in which these operations may be embedded, are also long-lived. Long transactions need different support than short transactions. In particular, blocking a transaction until another commits is rarely acceptable for long transactions. It is worthwhile noting that the problem of long transactions has also been addressed in traditional data processing applications (bank audit transactions, for example).

2. **Supporting user control**: In order to support user tasks that are nondeterministic and interactive in nature, the concurrency control mechanism should provide the user with the ability to start a transaction, interactively execute operations within it, dynamically restructure it, and commit or abort it at any time. The nondeterministic nature of transactions implies that the concurrency control mechanism will not be able to determine whether or not the execution of a transaction will violate database consistency, except by actually executing it and validating its results against the changed database. This might lead to situations in which the user might have invested many hours running a transaction, only to find out later when he wants to commit it that some of the operations he performed within the transaction have violated some consistency constraints; he would definitely oppose the deletion of all his work (by rolling back the transaction) in order to prevent the viola-

tion of consistency. He might, however, be able to explicitly reverse the effects of some operations in order to regain consistency. Thus, what is needed is the provision of more user control over transactions.

3. **Supporting synergistic cooperation**: Cooperation among programmers to develop versions of project components has significant implications on concurrency control. In CAD/CAM systems and SDEs, several users share knowledge collectively and through this knowledge, they are able to continue their work. Furthermore, the activities of two or more users working on shared objects may not be serializable. They may pass the shared objects back and forth in a way that cannot be accomplished by a serial schedule. Also, two users might be modifying two components of the same complex object concurrently, with the intent of integrating these components to create a new version of the complex object, and thus they might need to look at each others' work to make sure that they are not modifying the two components in ways that would make integration difficult. Such cooperation was coined *synergistic interaction* by Yeh *et al.* To insist on serializable concurrency control in design environments might thus decrease concurrency or more significantly actually disallow desirable forms of cooperation among developers.

There has been a flurry of research to develop new approaches to transaction management that meet the requirements of advanced applications. In the rest of the paper, we survey the mechanisms that address the requirements listed above. We categorize these mechanisms into three categories according to which requirement they support best. All the mechanisms that address the problems introduced by long transactions are grouped in one section. Of the mechanisms that address the issue of cooperation, some achieve only coordination of the activities of multiple users while others do allow synergistic cooperation; the two classes of mechanisms are separated into two different sections. Issues related to user control are slightly addressed by mechanisms in both categories, but we did not find any mechanism that provide satisfactory support for user control over transactions.

In addition to the three requirements listed above, many advanced applications require support for complex objects. For example, objects in a software project might be organized in a nested object system (projects consisting of modules that contain procedures), where individual objects are accessed hierarchically. We do not survey mechanisms that support complex objects because we felt that describing these mechanisms would require explaining concepts from object-oriented programming and object-oriented database systems, both of which are outside the scope of this paper. It is worthwhile noting, however, that the complexity of the structure and the size of objects in advanced applications strongly suggest the appropriateness of concurrency control mechanisms that combine and extend multiversion and multiple granularity mechanisms.

It might be interesting to note that many of the ideas implemented in the mechanisms we survey in the rest of the paper have actually been discussed earlier in other contexts. For instance, some of the ideas related to multilevel transactions, long transactions and cooperative transactions were discussed by Davies in [Davies 78].

6 SUPPORTING LONG TRANSACTIONS

Many of the operations performed on data in advanced database applications are long-lived. Some last for several minutes or hours such as compiling code or printing a complete layout of a VLSI chip. When these operations are part of a transaction, they result in a long transaction (LT), which lasts for an arbitrarily long period of time (ranging from hours to weeks). Such transactions occur in traditional domains (e.g., printing the monthly account statements at a bank) as well as in advanced applications, but they are usually an order of magnitude longer in advanced applications. LTs are particularly common in design environments, and the length of their duration causes serious performance problems if these transactions are allowed to lock resources until they commit. Other short or long transactions wanting to access the same resources are forced to wait even though the LT might have finished using the resources. LTs also increase the likelihood of automatic aborts (rollback), in order to avoid deadlock, or in the case of failing validation in optimistic concurrency control.

Two main approaches have been pursued to solve these problems: (1) extending serializability-based mechanisms while still maintaining serializable schedules; and (2) relaxing serializability of schedules containing LTs. These alternative approaches utilize the application-specific semantics of operations in order to increase concurrency. Several examples of each approach are presented in this section. Some of the schemes were proposed to support LTs for traditional DBMSs, but the techniques themselves seem applicable to advanced applications and thus they are discussed in this section rather than earlier.

6.1 Extending Serializability-based Techniques

In traditional transaction processing, all database operations are abstracted into read and write operations. This abstraction is necessary for designing general-purpose concurrency control mechanisms that do not depend on the particulars of applications. Two-phase locking, for example, can be used to maintain consistency in any database system, regardless of the intended application, because it maintains serializability, and thus consistency, of transaction schedules by guaranteeing the atomicity of all transactions.

Given the requirement of supporting long transactions in advanced applications, however, the performance of two-phase locking becomes unacceptable because it would force LTs to lock resources for a long time even after they have finished using them, while blocking other short and long transactions that need to access the same resources. Optimistic mechanisms that use timestamp ordering would cause repeated rollback of transactions given that the rate of conflicts would increase significantly in the context of long transactions. One paradigm for solving the problems introduced by LTs is to make use of any additional information that can be extracted and use that information with one of the traditional techniques, while maintaining the same traditional scheme in case the additional information is not available (i.e., it might be available for some transactions but not for others). This paradigm is the basis for extending both two-phase locking and optimistic concurrency control in order to address the requirements of long transactions.

6.1.1 Altruistic Locking

One piece of information that can be used to increase concurrency is when resources are no longer needed by a transaction, so that they can be released and used by other transactions. This information can be used to allow a long transaction, that otherwise follows a serializable mechanism such as two-phase locking, to conditionally unlock some of its resources so that they can be used by other transactions that meet certain requirements.

One formal mechanism that follows this approach is *altruistic locking*, which is an extension of the basic two-phase locking algorithm [Salem et al. 87]. Altruistic locking makes use of information about access patterns of a transaction to decide which resources it can release. In particular, the technique uses two types of information: (1) Negative access pattern information, which describes objects that will not be accessed by the transaction; and (2) Positive access pattern information, which describes which, and in what order, objects will be accessed by the transaction. Taken together, these two types of information allow long transactions to *release* their resources after they are done with them. Releasing a resource is a conditional unlock operation because it allows other transactions to access the released resource as long as they follow certain restrictions that ensure serializability. The set of all data items that have been locked and then released by an LT is called the *wake* of the transaction.

A *two-phase with release* schedule is then defined as any schedule that adheres to two restrictions:

1. No two transactions hold locks on the same data item simultaneously unless one of them locked and released the object before the other locked it; the later lock-holder is said to be in the wake of the releasing transaction.
2. If a transaction is in the wake of another transaction, it must be completely in the wake of that transaction. This means that if John's transaction locks a data item that has been released by Mary's transaction, then any data item, that is accessed by both John and Mary and that is currently locked by John, must have been released by Mary before it was locked by John.

This definition maintains serializability of transactions without altering the structure of transactions. The mechanism assumes that transactions are programmed and not user-controlled (i.e., the user cannot make up the transactions as he goes along). In the following example, however, we will assume an informal extension to this mechanism that will allow user-controlled transactions.

Consider the project depicted in figure 1 in the introduction where each module in the project contains a number of procedures (subobjects). Suppose that Bob, who joined the programming team of Mary and John, wants to familiarize himself with the code of all the procedures of the project, so he starts a long transaction, T_{Bob}, that accesses all of the procedures, one procedure at a time. Bob needs to access each procedure only once to read it and add some comments about the code of each procedure; as he finishes accessing each procedure he releases it. In the meanwhile, John starts a short transaction, T_{John} that accesses only two procedures, *p1* and then *p2*, from module A. Let us assume that T_{Bob} has already accessed *p2* and released it, and is currently reading *p1*. T_{John} has to wait until T_{Bob} is finished with *p1* and releases it, at which point T_{John} can start accessing *p1* by entering the wake of T_{Bob}, because all the objects that T_{John} needs to access (*p1* and *p2*) are in the wake of T_{Bob}. After finishing with *p1*, T_{John} can start accessing *p2* without delay.

Figure 6: Access patterns of three transactions

Now assume that Mary starts another short transaction, T_{Mary}, that needs to access both *p2* and a third procedure *p3* that is not in the wake of T_{Bob} yet. T_{Mary} can access *p2* after T_{John} terminates, but then it must wait until either *p3* has been accessed by T_{Bob} (i.e., until *p3* enters the wake of T_{Bob}) or until T_{Bob} terminates. If T_{Bob} never accesses *p3*, T_{Mary} is forced to wait until T_{Bob} terminates (which might take a long time since it is a long transaction). To improve concurrency in this situation, Salem *et al.* introduced a mechanism for *expanding* the wake of a long transaction dynamically in order to enable short transactions that are already in the wake of a long transaction to continue running. The mechanism uses the negative access information provided to it in order to only add objects that will not be accessed by the long transactions to its wake. Following up on the example, the expanding mechanism would add *p3* to the wake of T_{Bob} (by issuing a release on *p3* even if T_{Bob} had not locked it). This would allow T_{Mary} to access *p3* and thus continue executing without delay.

Figure 6 depicts the example above. Each data object is represented along the vertical axis whereas time is represented along the horizontal axis. The transactions belonging to Bob, John, and Mary are represented by solid lines. T_{Bob} accesses *p2* at time t_i since the line of T_{Bob} passes through a black dot at the point (t_i, *p2*). As shown, T_{John} is totally in the wake of T_{Bob} while T_{Mary} is not. The transaction manager can dynamically expand the wake of T_{Bob} by adding *p3* to it (as shown by the thick line) and then T_{Mary} would be totally in the wake of T_{Bob}. In this case, the schedule of the three transactions is equivalent to the serial execution of T_{Bob}, followed by T_{John}, followed by T_{Mary}.

The basic advantage of this scheme is its ability to utilize the knowledge that a transaction no longer needs access to a data object that it has locked. It maintains serializability and assumes that the data stored in the database is of the conventional form. Furthermore, if access information is not available, any transaction, at any time, can run under the conventional 2PL protocol without performing any special operations. However, as observed earlier, because of the interactive nature of transactions in design environments, the access patterns of transactions are not predictable. In the absence of this information, altruistic locking reduces to two-phase locking. Altruistic locking also suffers from the problem of cascaded rollbacks; the problem is that when a long transaction aborts, all the short transactions in its wake have to be aborted even if they already terminated.

6.1.2 Snapshot Validation

Altruistic locking assumes two-phase locking as its basis, and thus suffers from the overhead of locking mechanisms as noted in section 4. An alternative approach that avoids this overhead is to assume an underlying validation mechanism. As presented in section 4, validation (also called optimistic) techniques allow concurrent transactions to proceed without restrictions. Before committing a transaction, however, a validation phase, which establishes that the transaction did not produce conflicts with other committed transactions, has to be passed. The main shortcoming of the traditional validation technique is its weak definition of conflict, which causes some transactions, such as those in figure 2, to be restarted unnecessarily (i.e., the transactions might have been actually serializable but the conflict mechanism did not recognize them as such). This is not a serious problem in conventional applications where transactions are short. It is very undesirable, however, to restart a long transaction that has done a significant amount of work. Pradel *et al.* observed that the risk of restarting a transaction can be reduced by distinguishing between *serious conflicts*, which require restart, and *non-serious conflicts*, which do not. One mechanism that uses this approach is called *snapshot validation* [Pradel et al. 86].

Going back to our example, let us assume that Bob, John, and Mary start three transactions T_{Bob}, T_{John} and T_{Mary} simultaneously. T_{Bob} modifies (i.e., writes) procedures *p1* and *p2* during the read phase of T_{John} and T_{Mary} as shown in figure 7. The validation phases of T_{John} and T_{Mary} will thus consider T_{Bob} operations. According to the traditional optimistic concurrency control protocol, both T_{John} and T_{Mary} would have to be restarted because of conflicts; i.e., procedures *p1* and *p2* that they read have been updated by T_{Bob}. T_{John} read *p1*, which was later

Figure 7: Validation conflicts

changed by T_{Bob}, and thus what T_{John} read was out of date. This conflict is "serious" since it violates serializability and must be prevented. In this case, T_{John} has to be restarted, thus reading the updated *p1*. The conflict between T_{Mary} and T_{Bob}, however, is not serious since the concurrent schedule presented in figure 7 is equivalent to the serial schedule of T_{Bob} followed by T_{Mary}. This schedule is not allowed under the traditional protocol, but the snapshot technique allows T_{Mary} to commit because the conflict is not serious.

Pradel *et al.* presented a simple mechanism for determining whether or not conflicts are serious. In the example above, T_{Bob} terminates while T_{Mary} is still in its read phase. Each transaction has a read set that is ordered by the time of access of each object (i.e., if object *p1* is accessed before object *p2* by a transaction, then *p1* appears before *p2* in the read set). When T_{Bob} terminates, T_{Mary} takes note of the termination in its read set. During its validation phase, T_{Mary} has to consider only the objects that were read before T_{Bob} terminated. Any conflicts that occur after that in the read set are considered not serious. Thus, the conflict between T_{Bob} and T_{Mary} regarding procedure *p2* is not serious because T_{Mary} read *p2* after T_{Bob} has terminated.

Pradel *et al.* also analyzed the starvation problem in the conventional optimistic protocol and discovered that the risk of starvation is greater the longer the transaction is. Starvation occurs when a transaction that is restarted because it had failed its validation phase keeps failing its validation phase due to conflicts with other transactions. Starvation is detected after a certain number of trials and restarts. The classical optimistic concurrency control protocol solves the starvation problem by locking the whole database for the starving transaction, thus allowing it to proceed uninterrupted. Such a solution is clearly not acceptable for advanced applications. Pradel *et al.* present an alternative solution based on the concept of a substitute transaction.

If a transaction T_{John} is starving, a substitute transaction ST_{John} is established such that ST_{John} has the same read set and write set of T_{John}. At this point, T_{John} is restarted. ST_{John} simply reads its transaction number (the first thing that any transaction does) and then immediately enters its validation phase. This will force all other transactions to validate against

ST_{John}. Since ST_{John} has the same read and write sets of T_{John}, it will make sure that any other transaction T_j that conflicts with T_{John} would not pass its validation against ST_{John} and thus would have to restart. This "clears the way" for T_{John} to proceed with its execution with a much decreased risk of restart. ST_{John} terminates only after T_{John} commits.

6.1.3 Order-Preserving Serializability for Multilevel Transactions

The two mechanisms presented above extend traditional single-level protocols, in which a transaction is a flat computation made up of a set of atomic operations. In advanced applications, however, most computations are long-duration operations that involve several lower-level suboperations. For example, linking the object code of a program involves reading the object code of all its component modules, accessing system libraries, and generating the object code of the program. Each of these operations might itself involve suboperations that are distinguishable. If traditional single-level protocols are used to ensure atomicity of such long transactions, the lower-level operations will be forced to be executed in serial order, resulting in long delays and a decrease in concurrency.

Beeri, Weikum and Schek have observed that concurrency can be increased if long-duration operations are abstracted into subtransactions that are implemented by a set of lower-level operations. If these lower-level operations are themselves translated into yet more lower-level operations, then the abstraction can be extended to multiple levels [Beeri et al. 88]. This is distinct from the traditional nested transactions model presented in section 4 [Reed 78; Moss 85] in two main respects: (1) a multilevel transaction has a fixed number of levels, of which each two adjacent pairs define a *layer* of the system, whereas nested transactions have no predefined notion of layers; and (2) in contrast to nested transactions where there need not be a notion of abstraction, in a multilevel transaction, the higher the level, the more abstract the state.

These two distinctions lead to a major difference between transaction management for nested transactions and for multilevel transactions. In nested transactions, a single global mechanism must be used because there is no predefined notion of layers. The existence of layers of abstraction in multilevel transactions opens the way to a modular approach to concurrency control where different concurrency control protocols (schedulers) can be applied at different layers of the system; i.e., layer-specific mechanisms can be used. Each of these protocols must ensure serializability with respect to its layer. Unfortunately, not all combinations of concurrency control schedulers lead to correct executions. To illustrate, assume we have a 3-level multilevel transaction and that the protocol between the second and third levels is commutativity-based, i.e., if two adjacent operations at the third level can commute, their order in the schedule can be changed. Changing the order of operations at the third level, however, might change the order of subtransactions at the second level, and since the protocol only considers operations at the third level, it may change the order of operations in such a way so as to result in a non-serializable order of the subtransactions at second level.

The example above shows that serializability is too weak a correctness criterion to use for a "handshake" between the protocols of adjacent layers in a multilevel system, and that it must be extended to take into account the order of transactions at the adjacent layers. Beeri, Bernstein and Goodman introduced the notion of *order-preserving correctness* as the necessary property that layer-specific protocols must use to guarantee consistency [Beeri et al. 86; Beeri et al. 89]. This notion was used earlier in a concurrency control model for multilevel transactions implemented in the DASDBS system [Weikum and Schek 84; Weikum 86]. A combined report on both of these efforts appears in [Beeri et al. 88]

The basic idea of order-preserving serializability is to extend the concept of commutativity which states that order transformation cannot be applied to two operations belonging to the same transaction because that changes the order of execution of operations within a transaction. This notion can be translated to multilevel systems by allowing the order of two adjacent commuting operations to change only if their least common ancestor does not impose an order on their execution. If commuting operations leads to serializing the operations of a subtransaction in one unit (i.e., they are not interleaved with operations of other subtransactions), and thus making it an atomic computation, the tree rooted at the subtransaction can be replaced by a node representing the atomic execution of the subtransaction. The pruning of serial computations, and thus reducing the number of levels in a multilevel transaction by one, is termed *reduction*.

To illustrate, assume that Mary is assigned the task of adding a new procedure *p10* to module A and recompiling the module to make sure that the addition of procedure *p10* did not introduce any compile-time errors. Bob is simultaneously assigned the task of deleting procedure *p0* from module A. Adding or deleting a procedure from module A is an abstraction that is implemented by two operations: updating the attribute that maintains the list of procedures contained in A (i.e., updating the object containing module A), and updating the documentation D to describe the new functionality of the module A after adding/deleting a procedure. Recompiling a module is an abstraction for reading the source code of the module and updating the object containing the module (to update its timestamp, for example). Consider the concurrent execution of T_{Mary} and T_{Bob} in figure 8(a). Although the schedule is not serializable, it is correct because the operations at the lower level can be commuted so as to produce a serializable schedule while preserving the order of the subtransactions at the second level. The results of successive commutations is shown in figures 8 (b) and (c). The result of applying reduction is shown in (d) and the final result of applying commutation to the reduced tree, which is a serial schedule, is shown in (e).

Beeri, Schek and Weikum have shown that order preservation is only a sufficient condition to maintain consistency across layers in a multilevel system. They present a weaker condition, *conflict-based order-preserving serializability*. This condition states that a layer-specific protocol need only preserve the order of conflicting operations of the top level of its layers. For example, consider the schedule in figure 9(a) which shows a concurrent execution of three transactions initiated by Mary, Bob and John. Compiling module A and compiling module B are non-conflicting operations since they do not involve any shared objects. Linking the subsystem containing both A and B, however, conflicts with the other two operations. Although the

Figure 8: Order-Preserving Serializable Schedule

schedule is not order-preserving serializable, it is correct because it could be serialized as shown in figure 9(b) by changing the order of the two compile operations, and since these are non-conflicting subtransactions, the change of order preserves correctness.

Martin [Martin 87] presented a similar model based on the paradigm of nested objects, which models hierarchical access to data by defining a nested object system. Each object in the system exists at a particular level of data abstraction in the system. Operations are specified for objects at all levels where operations at level *i* are specified in terms of operations at level *i-1*. Thus, the execution of operations at level *i* result in the execution of perhaps several *suboperations* at level *i-1*. The objects accessed by suboperations at level *i-1* on behalf of an operation on an object at level *i* are called *subobjects* of the object at level *i*.

Figure 9: Conflict-Based Order-Preserving Serializable Schedule

Martin's model allows two kinds of schedules that are not serializable, *externally serializable* schedules and *semantically verifiable* schedules. Externally serializable computations allow only serializable access to top-level objects while allowing nonserializable access to subobjects. Subobjects are thus left in a state that cannot be produced by any serial execution. Semantically verifiable schedules allow nonserializable access to objects at all levels. Nonserializable behavior can be proven to be correct if the operation semantics at all levels are given and considered. In Martin's model, weakening an object's conflict specification may produce a correct nonserializable schedule. For example, it can be specified that a write operation on a specific object at a specific level does not conflict with a read operation on the same node in figure 9, the scheduler would have allowed the link operation and the compile operations to be commuted. Such a schedule might be considered correct if the semantics of linking the object codes of two modules does not prohibit it from reading different versions of the two modules.

6.2 Relaxing Serializability

The approaches presented in the previous section extend traditional techniques while maintaining serializability as a basis for guaranteeing consistency. Another approach that aims at supporting long transactions is based on relaxing the serializability requirement by using the semantics of application-specific operations. Relaxing serializability increases the level of concurrency in a system of concurrent transactions, and thus improves its performance.

6.2.1 Semantics-Based Concurrency Control

Garcia-Molina observed that by using semantic information, a DBMS can replace the serializability constraint by the semantic consistency constraint [Garcia-Molina 83]. The gist of this approach is that from a user's point of view, not all transactions need to be atomic. Garcia-Molina introduced the notion of *sensitive transactions* to guarantee that users see consistent data on their terminals. Sensitive transactions are those which output only consistent data to the user, and thus must see a consistent database state in order to produce correct data. Not all transactions that output data are sensitive since some users might be satisfied with data that is only relatively consistent. For example, say that a manager of a software project wants to get an idea about the progress of his programming team by browsing the modules and procedures that exist in the project. He might be satisfied with information returned by a read-only transaction that, in order to avoid delays resulting from waiting for in-progress transactions to finish before reading their updates, does not take into consideration the updates being made by these in-progress transactions.

A *semantically consistent schedule*, which transforms the database from one semantically consistent state to another, is one that guarantees that all sensitive transactions obtain a consistent view of the database in the sense that they appear to be atomic transactions with respect to all other transactions.

When the notion of serializability is replaced by that of semantic consistency, it becomes difficult for a general concurrency control mechanism to decide which schedules preserve consistency. Even if all the consistency constraints were given to the system (which is not possible in the general case), there is no way for the concurrency control mechanism to guess which schedules maintain semantic consistency without running the schedules and checking the constraints on the resulting state of the database [Garcia-Molina 83]. Doing that, however, would be equivalent to implementing an optimistic concurrency control scheme which suffers from the problem of rollback. Thus, the concurrency control mechanism must be provided with information about which transactions are compatible with each other (i.e., their operations can be interleaved at certain points without violating semantic consistency). Having the user provide this information, of course, is not a good idea in the general case because it burdens the user with having to understand the details of applications. However, in some applications, this kind of burden might be acceptable in order to avoid the performance penalty of traditional general-purpose mechanisms. If this is the case, the question remains what kind of framework should the user be provided for supplying semantic information that can be used to relax serializability, thus allowing more concurrency among long transactions.

In some advanced applications such as CAD, where the different parts of the design are stored in a project database, it is possible to supply semantic information in the form of integrity constraints on database entities. Design operations incrementally change those entities in order to reach the final design [Eastman 80; Eastman 81]. By definition, full integrity of the design, in the sense of satisfying its specification, exists only when it is complete. Unlike in conventional domains where database integrity is maintained during all quiescent periods, the iterative design process causes the integrity of the design database to be only partially satisfied until the design is complete. There is a need to define transactions that maintain the partial integrity required by design operations. Kutay and Eastman proposed a transaction model that is based on the concept of *entity state*, which describes the degree of integrity satisfied by an entity [Kutay and Eastman 83]

Each entity in the database is associated with a state that is defined in terms of a set of integrity constraints. Like a traditional transaction, an entity state transaction is a collection of actions that read an entity set {e}R and potentially write into an entity set {e}W. Unlike traditional transactions, however, entity state transactions are instances of transaction classes, each of which defines: (1) the set of entities {e}R that instance transactions read; (2) the set of entities {e}W that instance transactions write; (3) the set of constraints $\{c^+\}^B$ that must be satisfied on {e}R and {e}W prior to the invocation of a transaction; (4) the set of constraints $\{c^-\}^D$ that can be violated during the execution of a transaction; and (5) the set of constraints $\{c^+\}^A$ that hold after the transaction execution is completed; and (6) the set of constraints $\{c^-\}^A$ that are violated after the transaction execution is completed. A very simple example of a transaction class is the class of browsing transactions that have all the entities in the database as their read set, and all other sets are empty since these transactions do not transform the database in any way.

The integrity constraints associated with transaction classes define a partial ordering of these classes in the form of a precedence ordering. Transaction classes can thus be depicted as a finite state machine where the violation or satisfaction of specific integrity constraints defines a transition from one database state to another. Based on this, Kutay and Eastman define a concurrency control protocol that detects violations to the precedence ordering defined by the application-specific integrity constraints. Violations are resolved by communication among transactions to negotiate the abortion of one or more of the conflicting transactions. Kutay and Eastman did not provide any details of inter-transaction communication.

6.2.2 Semantic Atomicity

Garcia-Molina presented a framework similar in some respects to Kutay and Eastman's in that it explicitly defines semantics of database operations. Garcia-Molina defines four kinds of semantic information: (1) transaction semantic types; (2) compatibility sets associated with each type; (3) division of transactions into smaller steps (subtransactions); and (4) countersteps to compensate for some of the steps executed within transactions. The first three kinds of information are declarative, while the fourth piece of information consists of a procedural description. Transactions are categorized into types depending on the specifics of the application, in particular, the kinds of data objects, and operations on them, supported by the application. Each transaction type is divided into steps with the assumption that each step must be performed as an indivisible unit. A compatibility set associated with a transaction type defines allowable interleavings between steps of transactions of the particular kind with the same or other kinds of transactions. Countersteps specify what to do in case a step needs to be undone.

Using these definitions, Garcia-Molina defines an alternative concept to atomicity called *semantic atomicity*. A transaction is said to be semantically atomic if all its steps are executed, or if any executed steps are eventually followed by their countersteps. An atomic transaction, in contrast, is one in which all or none of the steps are executed. In the context of a DBMS, the four pieces of information presented above are used by a locking mechanism that uses two kinds of locks: *local locks*, which ensure the atomicity of transaction steps; and *global locks*, which guarantee that the interleavings between transactions do not violate semantic consistency.

Thus, depending on the compatibility sets of different types of transactions, various levels of concurrency can be achieved. In one extreme, if the compatibility sets of all kinds of transactions are empty, the mechanism reverts to a traditional locking mechanism that enforces serializability of the long transactions. In the other extreme, if all transaction types are compatible, the mechanism only enforces the atomicity of the small steps within each transaction, and thus the mechanism reverts to a system of short atomic transactions (i.e., the steps). In advanced applications where this kind of mechanism might be the most applicable, allowable interleavings would be between these two extremes.

In Garcia-Molina's scheme, transactions are statically divided into atomic steps and compatibility sets that define the allowable interleavings with respect to those steps. Thus if transactions of type X are compatible with transactions of types Y and Z, then any two transactions T_i, of type Y, and T_j, of type Z, can arbitrarily interleave their steps with a transaction T_k, of type X. There is thus no distinction between interleaving with respect to Y and interleaving with respect to Z. Lynch observed that it might be more appropriate to have different sets of interleavings (in the form of specific breakpoints) with respect to different transaction types [Lynch 83].

This observation seems to be valid for systems in which activities tend to be hierarchical in nature, for example, software development environments. Transactions in such systems can often be nested into levels, where at each level, transactions that have something in common, in terms of access to data items, are grouped. Level one groups all the transactions in the system while subsequent levels group transactions that are more strongly related to each other. A strong relation between two transactions might be that they often need to access the same objects at the same time in a non-conflicting way. A set of breakpoints (defining interleavings) is then described for each level of the nesting where the higher order sets (for the higher levels) always includes the lower order sets. This results in a total ordering of all sets of breakpoints. This means that the breakpoints that specify interleavings at a level cannot be more restrictive than those that define interleavings at a higher level.

Let us illustrate this concept by following up on our example from the software development domain. To remind the reader, Bob, John and Mary are cooperatively developing a software project. In their development effort, they need to modify objects (code and documentation) as well as get information about the current status of development (e.g., the latest cross-reference information between procedures in modules A and B). Let us suppose that Mary starts two transactions (in two different windows, for example) T_{Mary1} and T_{Mary2} to modify a procedure in module A, and get cross-reference information, respectively; Bob starts a transaction T_{Bob1} to update a procedure in module B; John starts two transactions T_{John1} to modify module A, and T_{John2} to get cross-reference information.

A hierarchy of transaction classes can be set up as shown in figure 10. The top level includes all transactions. Level 2 groups all modification transactions (T_{Mary1}, T_{Bob1} and T_{John1}) together and all cross-reference transactions (T_{Mary2} and T_{John2}) together. Level 3 separates the

Figure 10: Multilevel Transaction Classes

transactions according to which modules they affect; for example, it separates the transactions that modify module A (T_{Mary1} and T_{John1}) from those modifying module B (T_{Bob1}). Level four contains all the singleton transactions. At

Then, the sets of breakpoints are specified by describing the transaction segments between the breakpoints. For example, the top level set might specify that no interleaving is allowed; the second-level set might specify that all modification transactions might interleave at some granularity, and that cross-reference transactions might similarly interleave, but that modification and cross-reference transactions cannot interleave (to guarantee that cross-reference information does not change while a modification transaction is in progress).

The gist of multilevel atomicity is for the concurrency control mechanism to use the sets of breakpoints to provide as much concurrency as defined by the allowed interleaving between these breakpoints at each level. Atomicity with respect to breakpoints and allowed interleaving is maintained at each level. Thus, the mechanism in our example might allow transactions T_{Mary1} and T_{John1} to interleave their steps while modifying module A (i.e., allow some level of cooperation so as not to block out module A for a long time by one of them), but it will not allow T_{Mary1} and T_{John2} to interleave their operations.

6.2.3 Sagas

Both semantic atomicity and multilevel atomicity are theoretical concepts that are not immediately practical. For example, neither Garcia-Molina [Garcia-Molina 83] nor Lynch [Lynch 83] explain how a multilevel atomicity scheme might be implemented (e.g., it is not clear how the user decides on the levels of atomicity and breakpoint sets). Simplifying assumptions are needed to make these concepts practical. One restriction that simplifies the multilevel atomicity concept is to allow only two levels of nesting: the LT at the top level and simple transactions. Making this simplifying restriction, Garcia-Molina and Salem introduced the concept of *sagas* [Garcia-Molina and Salem 87], which are LTs that can be broken up into a collection of sub-transactions that can be interleaved in any way with other transactions.

A saga is not just a collection of unrelated transactions because it guarantees that all its sub-transactions will be completed or they will be compensated (explained shortly). A saga thus satisfies the definition of a transaction as a logical unit; they are similar to Moss's nested transactions and Lynch's multilevel transactions in that respect. Sagas are different from nested transactions, however, in that, in addition to there being only two levels of nesting, they are not atomicity units since sagas may view the partial results of other sagas. By structuring long transactions in this way, non-serializable schedules that allow more concurrency can be produced. Mechanisms based on nested transactions as presented in section 4 produce only serializable schedules.

In traditional concurrency control, when a transaction is aborted for some reason, all the changes that it introduced are undone and the database is returned to the state that existed before the transaction began. This operation is called *rollback*. The concept of rollback is not applicable to sagas because unlike atomic transactions, sagas permit other transactions to change the same objects that its committed sub-transactions have changed. Thus, it would not be possible to restore the database to its state before the saga started without cascaded and transitive aborts of all the committed transactions that viewed the partial results of the aborted transaction. Instead, user-supplied *compensation functions* are executed to compensate for each transaction that was committed at the time of failure or automatic abort.

A compensation function undoes the actions performed by a transaction from a semantic point of view. For example, if a transaction reserves a seat on a flight, its compensation function would cancel the reservation. We cannot say, however, that the database was returned to the state that existed before the transaction started, because in the meantime, another transaction could have reserved another seat and thus the number of seats that are reserved would not be the same as it was before the transaction.

Although sagas were introduced to solve the problem of long transactions in traditional applications, their basic idea of relaxing serializability is applicable to design environments. For example, a long transaction to fix a bug in a design environment can be naturally modeled as a saga that consists of subtransactions to edit a file, compile source code, and run the debugger. These subtransactions can usually be interleaved with subtransactions of other long transactions. The three transactions in figure 9 can be considered as sagas, and the interleavings shown in the figure can be allowed under the sagas scheme. Using compensation functions instead of cascaded aborts is also suitable for advanced applications. For example, if one decided to abort the modifications introduced to a file, one can revert to an older version of the file and delete the updated version.

One shortcoming of sagas, however, is that it limits nesting to two levels. Most design applications are multilevel, that is, several levels of nesting are needed to support high-level operations that are translated into a set of component sub-operations [Beeri et al. 89]. In software development, for example, a high-level operation such as modifying a subsystem translates into a set of operations to modify its component modules, each of which in turn is an abstraction for modifying the procedures that make up the module.

Realizing the multilevel nature of advanced applications, several researchers have proposed models and proof techniques that address multilevel transactions. We have already described three related models in the previous subsection [Weikum and Schek 84; Beeri et al. 89; Beeri et al. 88]. Two other nested transaction models [Walter 84; Kim et al. 84] are described in section 7, since these two models address the issue of groups of users and coordinated changes. We now describe a formal model of correctness without serializability that is based on multilevel transactions.

6.2.4 Conflict Predicate Correctness

Korth and Speegle have presented a formal model that allows mathematical characterization of correctness without serializability [Korth and Speegle 88]. Their model combines three features that lead to enhancing concurrency over the serializability-based models: (1) versions of objects; (2) multilevel transactions; and (3) explicit consistency predicates, similar to Kutay and Eastman's predicates, described earlier. We describe their model intuitively.

The database in Korth and Speegle's model is a collection of entities, each of which has multiple versions (i.e., multiple values), which are persistent and not transient like in the traditional multi-version schemes. A specific combination of versions of entities is termed a *unique database state*. A set of unique database states that involve different versions of the same entities forms one database state, i.e., each database state has multiple versions. The set of all versions that can be generated from a database state is termed the *version state* of the database. A transaction in Korth and Speegle's model is a mapping from a version state to a unique database state, i.e., it transforms the database from one consistent combination of versions of entities to another. Consistency constraints are specified in terms of pairs of input and output predicates on the state of the database. A predicate, which is a logical conjunction of comparisons between entities and constants, can be defined on a set of unique states that satisfy it. Each transaction guarantees that if its input predicate holds when the transaction begins, its output predicate will hold when it terminates.

Instead of implementing a transaction by a set of flat operations, it is implemented by a pair of a set of subtransactions and a partial ordering on these subtransactions. Any transaction that cannot be divided into subtransactions is a basic operation such as read and write. Thus, a transaction in Korth and Speegle's model is a quadruple (T, P, I_t, O_t) where T is the set of subtransactions, P is a partial order on these subtransactions, I_t is the input predicate on the set of all database states, and I_t is the output predicate. The input and output predicates define three sets of data items related to a transaction: the input set, the update set, and the fixed-point set, which is the set of entities not updated by the transaction. Given this specification, Korth and Speegle define a *parent-based execution* of a transaction as a relation on the set of subtransactions *T* that is consistent with the partial order *P*, and which encodes dependencies between subtransactions based on their three data sets. This definition allows independent executions on different versions of database states.

Finally, they define a new multilevel correctness criteria: an execution is correct if at each level, every subtransaction can access a database state that satisfies its input predicate and the result of all of the subtransactions satisfies the output predicate of the parent transaction. But since determining whether an execution is in the class of correct executions is NP-complete, Korth and Speegle consider subsets of the set of correct executions that have efficient protocols. One of these subsets is the *conflict predicate correct* class (CPC) in which the only conflicts that can occur are a read of a data item followed by a write of the same data item (this is the same as in traditional multi-version techniques). In addition, if two data items are in different conjuncts of the consistency predicate, execution order must be serializable only with respect to each conjunct individually; if for each conjunct the execution order is serializable, then the execution is correct. The protocol that recognizes the CPC class creates a graph for each conjunct where each node is a transaction. An arc is drawn between two nodes if one node reads a data item in the conjunct and the other node writes the same data item in the same conjunct. A schedule is correct if the graphs of all conjuncts are acyclic. This class contains executions that could not be produced by any of the mechanisms mentioned above except maybe for sagas.

Time ↓	T_{John}	T_{Mary}
	read(B)	
		read(A)
	write(A)	
		write(A)
		write(B)
	write(B)	

Figure 11: Non-serializable but Conflict Predicate Correct Schedule

Figure 12: Graphs built by CPC protocol

To illustrate, consider the schedule shown in figure 11 (which is adapted from [Korth and Speegle 88]). This schedule is clearly not serializable and is not allowed by any of the traditional protocols. However, suppose that the database consistency constraint is a conjunct of the form *P1 OR P2*, and *P1* is over A while *P2* is over B. In this case, the schedule is in CPC since the data items A and B are in different conjuncts of the database consistency constraint and the graphs for both conjuncts *P1* and *P2* individually, as shown in figure 12, are acyclic.

6.2.5 Commit-Serializability

In many advanced database applications, such as design environments, operations are interactive. The operations a user performs within a transaction might be: (1) of uncertain duration; (2) of uncertain development, i.e., it cannot be predicated which operations the user will invoke a priori; and (3) dependent on other concurrent operations. Both altruistic locking and sagas address only the first and third of these characteristics. They do not address the uncertainty of the development of a transaction (i.e., the user makes it up as he goes along). Specifically, neither sagas nor long transactions in the altruistic locking scheme can be restructured dynamically to reflect a change in the needs of the user. To solve this problem, Pu *et al.* introduced two new operations, *split-transaction* and *join-transaction*, which are used to reconfigure long transactions while in progress [Pu et al. 88]. These two mechanisms are the basis of a concurrency control policy that they now call *commit-serializability* [Kaiser 89].

The basic idea of commit-serializability is that all sets of database actions that are included in a set of concurrent transactions are performed in a schedule that is serializable when the actions are committed. The schedule however is made up of new transactions that result from splitting and joining the original transactions. The new set of transactions may not correspond in a simple way to the originally initiated set. Split-transaction divides an ongoing transaction into two or more serializable transactions by dividing the actions and the resources between the new transactions. The resulting transactions can proceed independently from that point on. More important, the resulting transactions behave as if they had been independent all along, and the original transaction disappears entirely, as if it had never existed. Thus, the split-transaction operation can be applied only when it is possible to generate two serializable transactions.

One advantage of splitting a transaction is the ability to commit one of the new transactions and, therefore, release all of its resources so that they can be used by other transactions. The splitting of a transaction reflects the fact that the user who controlled the original transaction has decided that he is done with some of the resources reserved by the transaction, and this set of resources can be treated as part of a separate transaction. Note that the splitting of a transaction in this case has resulted from new information about the dynamic access pattern of the transaction, i.e., the fact that it no longer needs some resources. This is different from the static access pattern that altruistic locking uses to determine that a resource can be released. Another difference from altruistic locking is that rather than only allowing resources to be released by committing one of the transactions that result from a split, the transactions can proceed in parallel and be controlled by different users. Join-transaction does the reverse operation of merging the results of two or more separate transactions, as if these transactions had always been a single transaction, and releasing their resources atomically.

T_{Mary}	T_{MaryA}	T_{John}
initiate		
read(A)		
read(B)		
write(A)		initiate
		request read(A)
corresponding notify(A)		
split((B), (A))		
	commit(A)	
write(B)		actual read(A)
commit(B)		write(A)
		read(B)
		write(B)
		commit(A, B)

Time

Figure 13: Example of Split-Transaction

To clarify this technique, suppose that both Mary and John start two long transactions T_{Mary} and T_{John} to modify the two modules A and B. After a while, John find out that he needs to access module A. Being notified that T_{John} needs to access module A, Mary decides that she can "give up" the module since she has finished her changes to it, so she splits up T_{Mary} into T_{Mary} and T_{MaryA}. Mary then commits T_{MaryA}, thus committing her changes to A while continuing to retain B. Mary can do that only if the changes committed to A do not depend in any way on the previous or planned changes to B, which might later be aborted. T_{John} can now read A and use it for testing code. Mary independently commits T_{Mary} thus releasing B. T_{John} can then access B and finally commit changes to both A and B. The schedule of T_{Mary}, T_{MaryA} and T_{John} is shown in figure 13.

Commit-serializability relaxes the traditional concept of serializability by allowing transactions to be dynamically restructured. Eventually, the restructuring produces a set of transactions that are serialized. Unlike all the other approaches described earlier in this section, this approach addresses the issue of user control over transactions since it allows users to dynamically restructure their long transactions.

7 SUPPORTING COORDINATION AMONG MULTIPLE DEVELOPERS

When a group of designers works on a large project, there arises a need to coordinate the access of its member designers to the project database. The designers work independently most of the time on the parts of the project they are responsible for, but they need to interact at various points to integrate their work. Thus, a few coordination rules, which moderate the concurrent access to the project database by multiple designers, need to be enforced in order to guarantee that designers do not either duplicate or invalidate the work of other designers.

In this section, we describe some mechanisms that coordinate the efforts of multiple designers. It is important to emphasize that all the mechanisms described in this section fall short of supporting synergistic cooperation among designers in the sense of being able to pass incomplete but relatively stable data objects between them in a non-serializable fashion. It is also important to note that unlike the mechanisms that were presented in the previous section, most of the models presented here were not developed as formal transaction models, but rather as practical systems to support design projects. The behavior of these systems, however, can be formulated in terms of a transaction model, as we do in this section.

7.1 Pessimistic Coordination

The simplest form of supporting coordination among members of a programming team is to control their access to shared files so that only one developer can modify any file at any one time. One approach that has been implemented by widely-used software development tools like version control tools, such as SCCS [Rochkind 75] and RCS [Tichy 85], is the *reserve/deposit* mechanism (also called reserve/replace and checkout/checkin). Each data object is considered to be a collection of different versions, where each version represents the state of the object at some time in the history of its development. The versions are usually stored in the form of a compact representation that allows the full reconstruction of any version, if needed. Once the original version of the object has been created, it becomes *immutable*, i.e., it cannot be modified. Instead, a new version can be created after explicitly reserving the object, which makes a copy of the original version of the object (or the latest version thereafter) and gives the owner of the reservation exclusive access to the copy so that he can modify it and deposit it as a new version.

Other users who need to access the same object must either wait until the new version is deposited, or choose another version, if that exists, and reserve that instead. Thus, two or more users can modify the same object only by creating branches, which create different versions that can coexist. Branching ensures write serializability by guaranteeing that only one write per version of an object exists. The result of consecutive reserves, deposits, and branches is a version tree that records the full history of development of the object. When two branches of the version tree are merged (by manually merging the latest version of each branch into one version), the tree becomes a dag. This scheme is pessimistic since it does not allow access conflicts to occur (rather than allowing them to occur and then correcting them as in optimistic schemes).

The basic reserve/deposit mechanism provides minimal coordination between multiple developers because it does not use semantic information about the objects it manipulates or the operations performed on these objects. It suffers from two main problems as far as concurrency control is concerned. First, it does not support any notion of aggregate or composite objects, forcing the user to reserve and deposit each object individually. This can lead to problems if a programmer reserves several objects, all of which belong to one aggregate object, creates new versions of all of them, makes sure that they are consistent as a set, and then forgets to deposit one of the objects. This will lead to an inconsistent set of versions being deposited. Second, the reserve/deposit mechanism does not provide any support for reserved objects beyond locking them in the public database. Thus, once an object has been reserved by a programmer, it is not controlled by the concurrency control mechanism but by the owner of the reservation who can decide to let other random programmers access that object.

Two mechanisms that provide partial solutions to both of the problems described above are the *conversational transactions* mechanism provided as an extension to System R [Lorie and Plouffe 83; William et al. 81], and the *design transactions* mechanism [Katz and Weiss 84]. In both mechanisms, the database of a design project consists of a public database, which is shared among all designers, and several private databases, each of which is only accessed by a single designer. Each designer starts a long transaction in his private databases that lasts for the duration of the design task he is responsible for. Both models are referred to as the conversational transactions model hereafter.

When a designer needs to access an object from the public database, he requests to check out the object in a particular mode, either to read it, write it, or delete it. This request initiates a short transaction on the public database, which sets a short-lived lock on the object that is requested, and checks if the object has been checked out by another transaction in a conflicting mode. If it has not, the short transaction sets a permanent lock for the duration of the long transaction on the object, copies the object to the specific private database, removes the short-lived lock, and commits. Otherwise, it removes the short-lived lock, notifies the user that he cannot access the object, and aborts. The short-lived locks that are created by check out and check in transactions on the public database are used to prevent other check out or check in transactions from accessing the same object at the same time. The permanent locks prevent long transactions from checking out an object that has already been checked out in an exclusive mode.

All objects that are checked out by a long transaction are checked back in by initiating short check-in transactions on the public database at the end of the long transaction. A check-in transaction copies the object to the public database, and deletes the old version of the object that was locked by the corresponding check-out transaction. The new version of the object does not inherit the long-lived lock from its predecessor. Thus, each conversational transaction ensures that all the objects that it checked out will be checked back in before it commits. This mechanism solves the first problem described above with the reserve/deposit model.

A concurrency control mechanism similar to conversational transactions is used in Smile, a multi-user software development environment [Kaiser and Feiler 87]. Smile adds semantics-based consistency preservation to the conversational transactions model by enforcing global consistency checks before allowing a set of objects to be checked back in. Smile also maintains semantic information about the relations among objects, which enables it to reason about collections of objects rather than individual objects, thus providing more support to composite objects such as modules or subsystems.

Like the conversational transactions model, Smile maintains all information about a software project in a *main database*, which contains the baseline version of a software project. Modification of any part of the project takes place in private databases called *experimental databases*. To illustrate Smile's transaction model, assume that John wants to modify modules A and B; he starts a transaction T_{John} and reserves A and B in an experimental database (EDB_{John}). When a module is reserved, all of its subobjects (e.g., procedures, types, etc.) are also reserved. Reserving A and B guarantees that other transactions will not be able to modify these modules until John has deposited them. Other transactions, however, can read the baseline version of the modules from the main database. John then proceeds to modify the body of the modules. When the modification process is complete, he requests a deposit operation to return the updated A and B to the main database and make all the changes available to other transactions.

But before a set of modules is deposited from an experimental database to the main database, Smile compiles the set of modules together with the unmodified modules in the main database, to make sure that they do not contain errors (i.e., that the set is self-consistent, and that it did not introduce any errors that would prevent integrating it with the rest of the main database). If the compilation succeeds (i.e., no errors are detected), the modules are deposited and T_{John} commits. Otherwise, John is informed of the errors and the deposit operation is aborted. In this case, John has to fix the errors in the modules and repeat the deposit operation when he is done. T_{John} commits only when the set of modules that were reserved are successfully compiled and then deposited.

Smile's model of consistency does not only enforce self-consistency of the set of modules, but it also enforces global consistency with the baseline version of all other modules. Thus, John will not be permitted to make a change to the interface of module A (to the number or types of parameters of a procedure, for example) within EDB_{John} unless he has reserved all other modules that may be affected by the change. For example, if procedure *p1* of module A is called by procedure *p7* of module C, then John has to reserve module C (in addition to A and B, which he has already reserved) before he can modify the interface of *p1*. If another transaction T_{Mary} has module C reserved in another experimental database, EDB_{Mary}, the operation to change *p1* is aborted and T_{John} is forced to either wait until T_{Mary} deposits C, at which point T_{John} can reserve it, or to continue working on another task that does not require C. From this example, it should be clear that by enforcing semantics-based consistency, Smile restricts cooperation even more than both the reserve/deposit model and conversational transactions because two users cannot simultaneously access objects that are semantically related to each other at the interface level.

Although the two-level database hierarchy of Smile and the conversational transactions mechanism provides better coordination support than the basic reserve/deposit model, it does not allow for a natural representation of hierarchical design tasks in which groups of users participate. Supporting such a hierarchy requires a nested database structure similar to the one provided by the multilevel transaction schemes described in the previous section.

7.2 Multilevel Pessimistic Coordination

A more recent system, Infuse, supports a multilevel hierarchy of experimental databases rather than a two-level hierarchy, and relaxes application-specific consistency constraints by enforcing only that modules in an experimental database have to be self-consistent before they are deposited to the parent database [Kaiser and Perry 87]. More global consistency is enforced only when the modules reserved in top level experimental databases are deposited to the main database.

Figure 14: Experimental databases in Infuse

Returning to our example, let us assume that both Bob and Mary are involved in a task that requires modifying modules A and C. Figure 14 depicts the situation. An experimental database ($EDB_{A,C}$), in which both A and C are reserved, is created. Between themselves, they decide that Bob should modify module A while Mary should work on module C. Bob creates a child experimental database (EDB_A) in which he reserves A, and Mary creates EDB_C in which she reserves C. Bob decides that his task requires changing the interface of procedure *p1* by adding a new parameter. At the same time, Mary starts modifying module C in her database (remember that procedure *p7* of module C calls *p1* in module A). After Bob completes his changes, he deposits module A to EDB_A. No errors are detected at that point because Infuse only checks that A is self-consistent. This is possible because Infuse assumes that any data types or objects that are used in the module but not defined in it, must be defined elsewhere. If they are not defined anywhere in the system, the final attempt to deposit into the main database will detect that, Infuse however checks that all uses of a data type or object in the same module are consistent with each other.

Mary then finishes her changes and deposits C. Again no errors are detected at that level. However, when either Bob or Mary proceeds to deposit the modules in $EDB_{A,C}$ to the main database, the compiler reports that modules A and C are not consistent with each other because of the new parameter of procedure *p1*. At that point, either Bob or Mary has to create a child experimental database in which he or she can fix the bug by changing the call to *p1* in procedure *p7*.

Infuse's model allows greater concurrency at the cost of greater semantics-based inconsistency — and the potential need for a later round of changes to re-establish consistency; but serializability is always maintained by requiring sibling EDBs to reserve disjoint subsets of the resources locked by the parent EDB.

7.3 Optimistic Coordination

The coordination models described thus far are pessimistic in that they do not allow concurrent access to the same object in order to prevent any consistency violations that might occur. It is often the case in design efforts, however, that two or more developers within the same team would prefer to access different versions of the same data item concurrently rather than be delayed. Since these developers are familiar with each other's work, they can resolve any conflicts they introduced during their concurrent access by merging the different versions into a consistent version. One approach to providing such an optimistic coordination scheme is to support the notion of consistent sets of versions.

The schemes described above use immutable objects either implicitly or explicitly. None of them, however, supports consistent sets of version. Thus, if a system has been built using the latest version of each object in the database, when new versions of the objects are created, it would be impossible to find out which versions of the objects actually participated in building the system, in case we want to debug that system. There is a need to group sets of versions that are self-consistent into *configurations* that would enable programmers to reconstruct a system using the correct versions of the objects that comprise the system. This notion of configurations is supported by many software development systems(e.g., [Leblang and Chase, Jr. 87]). Thus, immutability of versions of objects in design environments, where objects have multiple versions, reduces the problem of consistency to the problem of explicitly naming the set of consistent versions in configuration objects. This basically solves the problem of reserve/deposit where only ad hoc ways (associating attributes with versions deposited at the same time) can be used to keep track of which versions of different objects belong together. Therefore, transaction mechanisms are responsible for *naming* consistent groups of versions of related objects [Walpole et al. 88a].

7.3.1 Domain Relative Addressing

Walpole *et al.* have addressed the problem of consistency in immutable object systems and introduced a consistency control notion called *domain relative addressing* that supports versions of configuration objects [Walpole et al. 87; Walpole et al. 88b]. Domain relative addressing extends the notion of time relative addressing (multiversion concurrency control) introduced by Reed. Whereas Reed's mechanism synchronizes accesses to objects with respect to their timestamps, domain relative addressing does so with respect to the domain of the data items accessed by the transaction. The database is partitioned into separate consistent domains where each domain (configuration) consists of one version of each of the conceptual objects in a related set.

	T_{John}	T_{Mary}
|		
|	read(p1)	
|		read(p1)
|		read(p2)
|	modify(p1)	
|	write(p1)	
|		modify(p2)
|	read(p2)	
|	modify(p2)	
|	write(p2)	
|		write(p2)
V		
Time		

Figure 15: Domain relative addressing schedule

Figure 16: Maintaining consistency using domain relative addressing

To illustrate this technique, consider the two transactions T_{Bob} and T_{John} of figure 15. The schedule in the figure is disallowed by all the conventional concurrency control schemes. Under domain relative addressing, the schedule in figure 15 is allowed because T_{Bob} and T_{John} operate on different versions of procedures p1 and p2. A similar scenario occurs if Bob wants to modify module A and then modify module B to make it consistent with the updated A. At the same time, John wants to modify B keeping it consistent with A. This can be done if T_{Bob} and T_{John} use different versions of modules A and B as shown in figure 16. This scheme captures the semantics of the operations performed (consistent updates) by maintaining that version A^1 (the original version of module A) is consistent with B^1 (the version of B modified by T_{John}), while A^2 (module A after T_{Bob} has modified it) is consistent with B^2 (new version of B that T_{Bob} has created). All of A^1, A^2, B^1, and B^2 become immutable versions. Domain relative addressing is the concurrency control mechanism used in the Cosmos software development environment [Walpole et al. 88a].

7.3.2 Copy/Modify/Merge

Like Infuse, Sun's Network Software Environment (NSE) supports a nested transaction mechanism that operates on a multilevel hierarchical database structure [Adams et al. 89]. Like Cosmos (and unlike Infuse), NSE supports concurrent access to the same data objects by combining the reserve/deposit model with an extension to the classical optimistic concurrency control policy, thus allowing limited cooperation among programmers. Unlike Cosmos, however, NSE provides some assistance to developers in merging different versions of the same data item.

NSE requires programmers to *acquire* (reserve) copies of the objects they want to modify in an *environment* (not to be confused with a software development environment), where they can modify the copies. Programmers in other environments at the same level cannot access these copies until they are deposited to the parent environment. Environments, however, can have child environments that acquire a subset of their set of copies. Multiple programmers can operate in the same environment where the basic reserve/deposit mechanism is enforced to coordinate their modifications.

Several sibling environments can concurrently acquire copies of the same object and modify them independently, thus creating parallel versions of the same object. To coordinate the deposit of these versions to the parent environment, NSE requires that each environment merge its version (called *reconcile* in NSE's terminology) with the previously committed version of the same object. Thus, the first environment to finish its modifications deposits its version as the new version of the original object in the parent environment, the second environment to finish has to merge its version with the first environment's version, creating a newer version. The third environment to finish will merge its version with this newer version and so on.

Like the optimistic concurrency control (OCC) mechanism, NSE's mechanism allows concurrent transactions (programmers in sibling environments in this case) to simultaneously access private copies of the same object. Before users can make their copies visible to other users (i.e., the write phase in the OCC mechanism), they have to reconcile (validate) the changes they made with the changes that other users in sibling environments have concurrently made on the same object. If conflicts are discovered, rather than rolling back transactions, the users of conflicting updates have to *merge* their changes, producing a new version of the object.

Figure 17: Layered development in NSE

To illustrate this mechanism, assume that the modules of the project depicted in figure 1 in the introduction represent the following: module A comprises the user interface part of the project, module B is the kernel of the project, module C is the database manager, while module D is a library module. The development happens in three layers as shown in figure 17. At the top layer, the environment PROJ-ENV represents the released project. All the objects of the project belong to this environment. At the second level, two environments co-exist: one to develop the user interface, FRONT_END, and the other to develop the kernel, BACK_END. FRONT_END acquires copies of modules A and C, while BACK_END acquires copies of B and C. John works on modifying the front end in his private environment, JOHN, while Mary works on developing the back end in her private environment.

John starts by acquiring module A in order to modify it. He creates a new version of p1 but then finds out that in order to modify p2, he needs to modify p5. Consequently, he acquires p5 into his environment and creates new versions of p2 and p5. Finally he deposits all his changes to FRONT_END, creating new versions of modules A and C as shown in figure 17. Concurrently, Mary acquires module B and modifies it, and deposits the changes to BACK_END. Mary can then test her code in BACK_END. Let us suppose that before Mary starts testing her code, John finishes testing his code and deposits all of his changes to the top-level environment, creating a new version of the project and making all of his changes visible to everybody. Before testing her code, Mary can check to see if any of the code that is relevant to her (modules B and C) has been changed by some other programmer. NSE provides a command, *resync*, to do that automatically on demand. *Resync* will inform Mary that John has changed procedure p5. At this point, Mary can decide to acquire John's new version and proceed to test her code.

In another scenario, the exact same series of actions as above occur except that Mary discovers that she needs to modify procedure p5 in C, so she acquires it. In this case, after the *resync* command informs her that John has already deposited a new version of p5, Mary has to merge her new version with John's. This is done by invoking a special editor that facilitates the merging process. Merging produces a new version of p5, which Mary can use to test her code. Finally, she can deposit all of her code, creating a new version of the whole project.

7.3.3 Backout and Commit Spheres

Both Infuse and NSE implicitly use the concept of nested transactions, and they enforce a synchronous interaction between a transaction and its child subtransactions, in which control flows from the parent transaction to the child subtransaction. Subtransactions can access only data items that the parent transaction can access, and they commit their changes only to their parent transaction. A more general model is needed in order to support a higher level of coordination among transactions. Walter observed that there are three aspects that define the relationship between a parent transaction and child subtransaction [Walter 84]: (1) the interface aspect; (2) the dependency aspect; and (3) the synchronization aspect.

The interface between a parent transaction and a child subtransaction can either be single-request, i.e., the parent requests a query from the child and waits until the child returns the result, or *conversational*, i.e., the control changes between the parent that issues a sequence of requests and the child that answers these requests. A conversational interface, in which values are passed back and forth between the parent and the child, necessitates grouping the parent and child transactions in the same rollback domain because if the child transaction is aborted (for any reason) in the middle of a conversation, not only does the system have to rollback the changes of the child transaction, but the parent transaction has to be rolled back to the point before the conversation began. In this case, the two transactions are said to belong to the same *backout sphere*. A backout sphere includes all transactions that are involved in chain of conversations and requires that backing out (rollback) of all transactions in the sphere if any one of them is backed out. A single-request interface, which is what the traditional nested transaction model supports, does not require rolling back the parent because the computation of the child transaction does not affect the computation in-progress in the parent transaction.

The dependency aspect concerns a child transaction's ability to commit its updates independently of when its parent transaction commits. If a child is independent of its parent, then it is said to be in a different *commit sphere*. Any transaction within a commit sphere can commit only if all other transactions in its sphere also commit. If a child, which is in a different commit sphere from its parent, commits, then the parent must either remember that the child committed (e.g., by writing the committed values in its variables), or it must be able to execute the child transaction again if the parent is restarted.

The synchronization aspect concerns the ability to support the concurrent execution of the parent transaction and its subtransactions. Such concurrency can occur if the child subtransaction is called from the parent transaction asynchronously (i.e., the parent continues its execution and fetches the results of the child subtransaction at a later time). In this case, both the parent and the child may attempt to access the same data items at the same time, and thus the need for synchronization. If the child is called synchronously (i.e., the parent waits until the child terminates), then it can safely access the data items locked by its parent.

Given these three aspects, Walter presented a nested transaction model in which each subtransaction has three attributes that must be defined when it is created. The first attribute, reflecting the interface criterion, can be set to either COMMIT or NOCOMMIT. The dependency attribute is set to either BACKOUT or NOBACKOUT, and the third attribute, reflecting the synchronization mode, is set to either SYNC or NOSYNC. The eight combinations of these attributes define levels of coordination between a transaction and its subtransactions. For example, a subtransaction created with the attributes COMMIT, BACKOUT, SYNC, is independent of its parent since it possesses its own backout sphere, its own commit sphere and it can access data items not locked by its parent.

Walter observes that it is possible to define all other nested transaction models in his model. Moss' model for example is defined as creating subtransactions with attributes set to BACKOUT, NOCOMMIT, SYNC. Beeri, Schek and Weikum's multilevel transaction model [Beeri et al. 88], described in the previous section, supports the combination COMMIT, BACKOUT, NOSYNC. No synchronization is needed between a transaction and its subtransactions because they operate at two different levels of abstraction (e.g., if locking is used, different levels would use different types of locks).

The models presented in this section support limited cooperation among teams of developers mainly by coordinating their access to shared data. Both NSE and Cosmos allow two or more environments to acquire copies of the same object, modify them and merge them. NSE also provides programmers with the ability to set *notification* requests on particular objects so that they are informed of other programmers who acquire or reconcile these objects. Infuse provides a notion of *workspaces* that cut across the hierarchy to permit grouping of an arbitrary set of experimental databases. This "cutting across" enables users to look at the partial results of other users' work, for the purpose of early detection of inconsistencies. None of the models described so far, however, support all the requirements of synergistic cooperation among teams of developers.

8 SUPPORTING SYNERGISTIC COOPERATION

In the previous section, we addressed the issue of coordinating the access of a group of developers to the shared project database. Although this coordination is often all what is needed for small groups of developers, it is not sufficient when a large team of designers work on a large-scale design project. The teams are often subdivided into several groups, each responsible for a part of the design task. Members of each group usually cooperate to complete the part they are responsible for. In this case, there is a need to support cooperation among members of the same group, as well as coordination of the efforts of multiple groups. The mechanisms described in the previous section address the coordination part, but most of them do not support any form of cooperation.

Supporting synergistic cooperation necessitates relying on sharing the collective knowledge of designers. For example, in an SDE, it is common to have several programmers cooperate on developing the same subsystem. Each programmer becomes an "expert" in a particular part of the subsystem, and it is only through the sharing of the expertise of all the programmers that the subsystem is integrated and completed. In such a cooperative design environment, the probability of conflicting accesses to shared data is relatively high because it is often the case that several users, with overlapping expertise, are working on related tasks concurrently. Note that in an SDE there is overlapping access to executables and status information even if not to source code.

However, many of the conflicts that occur in design environments are not serious in the sense that they can be tolerated by users. In particular, designers working closely together often need to exchange incomplete designs, knowing that they might change shortly, in order to coordinate the development of various parts of the design. A DBMS supporting such an environment should not obstruct this kind of cooperation by disallowing concurrent access to shared objects or non-serializable interaction.

Instead, the concept of database consistency preservation needs to be refined along the lines of the previous section to allow non-serializable cooperative interaction. Such a refinement can be based on four observations [Bancilhon et al. 85]: (1) design efforts are usually partitioned into separate projects, where each project is developed by a team of designers; (2) available workstations provide multiple windows, in which multiple tasks can be executed concurrently by the same designer; (3) projects are divided into subtasks were a group of designers, each working on a subtask, have a great need to share data among them; and (4) in complex design projects, some subtasks are contracted to other design groups (subcontractors) that have only limited access to the main project's database.

In this section, we present two models that aim at defining the underlying primitives needed for the implementation of cooperative concurrency control mechanisms. We then describe four mechanisms, two from the CAD/CAM community and two from the SDE domain, that use combinations of these primitives to implement cooperative concurrency control policies. It is worthwhile to note that much of the work described in this section is very recent, and some of it is preliminary. We believe, however, that the models presented here provide a good sample of the research efforts under way in the area of cooperative transaction models.

8.1 Cooperation Primitives

In order to address the four observations listed above, there is a need to introduce two new primitives that can be used by mechanisms supporting cooperation. The first primitive is *notification* (mentioned in the previous section), which enables developers to monitor what is going on as far as access to particular objects in the database is concerned. The second is the concept of a *group* of cooperating developers who are working on the same task (or at least related tasks), and thus need to cooperate among themselves much more than with other groups.

8.1.1 Interactive Notification

One approach to maintaining consistency, while still allowing some kind of cooperation, is to support notification and interactive conflict resolution rather than enforcing serialization [Yeh et al. 87]. To do this, the Gordion database system provides a notification primitive that can be used in conjunction with other primitives (such as different lock modes) to implement cooperative concurrency control policies [Ege and Ellis 87]. Notification alerts users about "interesting" events such as breaking a non-exclusive lock.

Two policies that use notification in conjunction with non-exclusive locks and versions were implemented in the Gordion system; these are *immediate notification* and *delayed notification*. Immediate notification alerts the user of any conflict (attempts to access an object that has a non-exclusive lock on it or out of which a new version is being created by another user) as soon as the conflict occurs. Delayed notification alerts the user of all the conflicts that have occurred only when one of the conflicting transactions attempts to commit. Conflicts are resolved by instigating a ‘‘phone call’’ between the two parties with the assumption that they can interact (hence the name interactive notification) to resolve the conflict.

Yeh *et al.* analyzed the performance of these two policies and concluded that the performance of these two protocols is better than the conventional approach, mainly because of the avoidance of nested waiting [Yeh et al. 87]. These policies incorporate humans as part of the conflict resolution algorithm. This, on the one hand, enhances concurrency in advanced applications where many of the tasks are interactive. But on the other hand, it can also degrade consistency because humans might not really resolve conflicts, resulting in inconsistent data. Like the sagas model, this model also burdens the user with knowledge about semantics of applications. This points to the need for incorporating some intelligent tools, similar to NSE's merge tool, to help the user resolve conflicts.

8.1.2 The Group Paradigm

Since developers of a large project often work in small teams, each responsible for a specific task, there is a need to formally define the kinds of interactions that can happen among members of the same team as opposed to interactions between teams. Abbadi and Toueg defined the concept of a group as a set of transactions, that when executed transforms the database from one consistent state to another [El Abbadi and Toueg 89]. Groups, like nested transactions, is a higher level abstraction than a transaction. Abbadi and Toueg presented the group paradigm to deal with consistency of replicated data in an unreliable distributed system, where they hierarchically divide the problem of achieving serializability into two simpler ones: (1) a local policy that ensures a total ordering of all transactions within a group; and (2) a global policy that ensures correct serialization of all groups.

There are significant differences between groups and nested transactions. A nested transaction is designed *a priori* in a structured manner as a single entity that may invoke subtransactions, which may themselves invoke other subtransactions. Groups do not have any *a priori* assigned structure and no predetermined precedence ordering imposed on the execution of transactions within a group. Another difference is that the same concurrency control policy is used to ensure synchronization among nested transactions at the root level and within each nested transaction. Groups, however, could use different local and global policies (an optimistic local policy, for example, and a 2PL global policy).

Although the group paradigm was introduced to model inter-site consistency in a distributed database system, it can be used to model teams of developers, where each team is modeled as a group with a local concurrency control policy that supports synergistic cooperation. A global policy can then be implemented to coordinate the efforts of the various groups. Of course, the local policies and the global policy have to be compatible in the sense that they do not contradict each other. Toueg and Abbadi do not sketch the compatibility requirements between global and local policies.

Dowson and Nejmeh have applied the group concept to model teams of programmers. The have introduced the notion of *visibility domains*, which models groups of programmers executing nested transactions on immutable objects [Dowson and Nejmeh 89]. A visibility domain is a set of users who can share the same data items. Each transaction has a particular visibility domain associated with it. Any member of a visibility domain of a transaction may start a subtransaction on the copy of data that belongs to the transaction. The only criterion for data consistency is that the visibility domain of a transaction be a subset of the visibility domain of its parent.

8.2 Cooperating Transactions

Variations of the primitives defined above have been the basis for several concurrency control mechanisms that provide various levels of cooperation. In this section we present four mechanisms that support some form of synergistic cooperation. Two of the mechanisms were designed for CAD environments while the other two were designed for SDEs. The notion of cooperation in SDEs is similar to that in CAD. There are differences, however, that arise from the differences in the structure of projects in the two domains. It seems that CAD projects are more strictly organized than software development projects, with a more stringent division of tasks, and with less sharing among tasks.

Designers working on the same subtask might need unconstrained cooperation, while two designers working on different tasks of the same project might need more constrained cooperation between them. Designers working on two different projects (although within the same division, for example) might be content with traditional transaction mechanisms that enforce isolation. In software development, programmers working on different projects might still need shared access to libraries, and thus might need more cooperation than is provided by traditional mechanisms even if their tasks are unrelated.

One approach to provide such support is to divide users (designers or programmers) into groups, and provide each group with a range of lock modes, similar to the range provided by Observer, that allow various levels of isolation and cooperation among multiple users in the same group, and among different groups. Specific policies that allow cooperation can then be implemented by the environment using the knowledge about user groups and lock modes. In this section, we describe four mechanisms that are based on the group concept. It is worth mentioning that the four mechanisms avoid using blocking to synchronize transactions, thus eliminating the problem of deadlock.

8.2.1 Group-Oriented CAD Transactions

An extension to the conversational transactions model, described in section 7, that provides such a range of lock modes, is the *group-oriented model* [Klahold et al. 85]. Unlike the conversational transactions model, which sets long-lived locks on objects that are reserved in a private database for a long period of time (until they are deposited to the public database), the group-oriented model avoids the problem of long-lived locks.

The model categorizes transactions into *group transactions* (GT) and *user transactions* (UT), where any UT is a subtransaction of a GT, and provides primitives to define groups of users with the intention of assigning each GT a user group. Each user group develops a part of the project in a *group database*. A GT reserves objects from the public database into the group database of the user group it was assigned. Within a group database, individual designers create their own user database, and they invoke UTs to reserve objects from the group database to their user database.

In the group-oriented model, user groups are isolated from each other (i.e., one user group cannot see the work of another user group until the work is deposited in the public database). Group transactions are thus serializable. Within a group transaction, several user transactions can run concurrently. These transactions, however, are serializable unless users intervene to make them cooperate in a non-serializable schedule. The basic mechanism provided for relaxing serializability is a version concept that allows parallel development (branching) and notification (being told that intermediate results exist), two requirements for synergistic cooperation. Versions are derived, deleted, and modified explicitly by a designer only after being locked. Not all locks are exclusive, however. Some of the lock modes allow more than one designer to cooperate on modifying the same version.

The model supports five lock modes: (1) read-only, which makes a version available only for reading; (2) read/derive, which allows the owner of the lock to read a version of an object non-exclusively by allowing concurrent reads of the same version, and derivation of a new version of the object in parallel with the read operations; (3) shared derivation, which allows the owner to read a version of an object, and derive a new version of it, while allowing parallel reads of the same version of the object and derivation of different new versions by other users; (4) exclusive derivation, which allows the owner of the lock to read a version of an object and derive a new version, and allows only parallel reads of the original version; and (5) exclusive lock, which allows the owner to read, modify and derive a version, and allows no parallel operations on the locked version.

Using these lock modes, several designers can cooperate on developing the same design object. The exclusive lock modes allow for isolation of development efforts (as in traditional transactions), if that is what is needed. To guarantee consistency of the database, designers are only allowed to access objects as part of a transaction. Each transaction in the group-oriented model is two-phase, consisting of an acquire phase and a release phase. Locks can only be strengthened (i.e., converted into a more exclusive mode) in the acquire phase, and weakened (converted into a more flexible lock) in the release phase. If a user requests a lock on a particular object and the object is already locked with an incompatible lock, the request is rejected and the initiator of the requesting transaction is informed of the rejection. This avoids the problem of deadlock, which is caused by blocking transactions that request unavailable resources. The initiator of the transaction is notified when the object he requested becomes available for locking.

In addition to this flexible locking mechanism, the model provides a read operation that breaks any lock by allowing a user to read any version, knowing that it might have been changed or is about to be changed. This operation provides the designer (more often a manager of a design effort) the ability to observe the progress of development of a design object, without affecting the designers doing the development.

8.2.2 Cooperating CAD Transactions

Like the group-oriented model, the *cooperating CAD transactions* model, introduced by Bancilhon, Kim and Korth, envisions a design workspace to consist of a global database that contains a public database for each project and private databases of active designers' transactions [Bancilhon et al. 85]. Traditional two-phase locking is used to synchronize access to shared data among different projects in the database. Within the same project, however, each designer is responsible for a well-defined subtask and he invokes a long transaction to complete the subtask.

All the designers of a single project participate in one cooperating transaction, which is the set of all long transactions initiated by those designers. All the short-duration transactions invoked by all the designers within the same cooperating transaction are serialized as if they were invoked by one designer. Thus, if a designer invokes a short-transaction (within his long transaction) that conflicts with another designer's short transaction, one of them has to wait only for the duration of the short transaction. Each cooperating transaction encapsulates a complete design task. Some of the subtasks within a design task can be "subcontracted" to another group instead of being implemented by members of the project. In this case, a special cooperating transaction called a *client/subcontractor* transaction is invoked for that purpose. Each client/subcontractor transaction can invoke other client/subcontractor transactions leading to a hierarchy of such transactions spawned by a single client (designer). This notion is similar to Infuse's hierarchy of experimental databases, discussed in the previous section.

A cooperating transaction is thus a nested transaction that preserves some consistency constraints defined as part of the transaction. Each subtransaction (itself a cooperating transaction) in turn preserves some integrity constraints (not necessarily the same ones as its parent transaction). The only requirement here is that subtransactions have weaker constraints than their ancestors but not vice versa. Thus, the integrity constraints defined at the top level of a cooperating transaction imply all the constraints defined at lower levels. At the lowest levels of the nested transaction are the database operations, which are atomic sequences of physical instructions such as reading and writing of a single data item.

To replace the conventional concept of a serializable schedule for a nested transaction, Bancilhon, Kim and Korth define the notion of an *execution* of a cooperating transaction to be a total order of all the operations invoked by the subtransactions of the cooperating transaction that is compatible with the partial orders imposed by the different levels of nested transactions. A *protocol* is a set of rules that restrict the set of admissible executions. Thus, if the set of rules are strict enough, they would allow only serializable executions. The set of rules can, however, allow non-serializable, and even incorrect, executions.

8.2.3 Transaction Groups

In order to allow members of the same group to cooperate and to monitor changes in the database, there is a need to provide concurrency control mechanisms with a range of lock modes of varying exclusiveness. The *transaction groups* model proposed for the ObServer system replaces classical locks with <lock mode, communication mode> pairs to support the implementation of a nested framework for cooperating transactions [Skarra and Zdonik 89; Fernandez and Zdonik 89]. A transaction group (TG) is defined as a process that controls the access of a set of cooperating transactions (members of the transaction group) to objects from the object server. Since a TG can include other TGs, a tree of TGs is composed.

Within each TG, member transactions and subgroups are synchronized according to an *input protocol* that defines some semantic correctness criteria appropriate for the application. The criteria are specified by semantic patterns, and enforced by a recognizer, which makes sure that a lock request from a member transaction matches an element in the set of locks that the group may grant its members, and a conflict detector, which makes sure that a request to lock an object in a certain mode does not conflict with the locks already held on the object.

If a transaction group member requests an object that is not currently locked by the group, the group has to request a lock on the object from its parent. Because the input protocol (which control access to objects) of the parent might be different from that of the child group, the group might have to transform its request lock into a different lock mode accepted by the parent's input protocol. The transformation is carried out by an *output protocol*, which consults a lock translation table to determine how to transform a lock request into one that is acceptable by the parent group.

The lock modes provided by Observer indicate whether the transaction intends to read or write the object and whether it permits reading while another transaction writes, writing while other transactions read, and multiple writers of the same objects. The communication modes specify whether the transaction wants to be notified if another transaction needs the object or if another transaction has updated the object. Transaction groups and the associated locking mechanism provide suitable low-level primitives for implementing a variety of concurrency control policies.

To illustrate, consider the following example. Mary and John are assigned the task of updating modules M and N that are strongly related (i.e., procedures in them call each other, and type dependencies exist between the two modules), while Bob is assigned responsibility for updating the documentation of the project. Obviously, Mary and John need to cooperate while updating the modules whereas Bob only needs to access the final result of the modification of both modules in order to update the documentation. Two transaction groups are defined, *TG1* and *TG2*. TG1 has T_{Bob} and TG2 as its members, and TG2 has T_{John} and T_{Mary} as its members. The output protocol of TG2 states that changes made by the transactions within TG2 are committed to TG1 only when all the transactions of TG2 have either committed or aborted. The input protocol of TG2 accepts lock modes that allow T_{Mary} and T_{John} to cooperate (e.g., see

Figure 18: Transaction Groups

partial results of their updates to the modules) while isolation is maintained within TG1 (to prevent T_{Bob} from accessing the partial results of the transactions in TG2. This arrangement is depicted in figure 18.

8.2.4 Participant Transactions

The transaction groups mechanism defines groups in terms of their access to database objects in the context of a nested transaction system. Another approach is to define a group of users as *participants* in a specific set of transactions, meaning that these transactions need not appear to have been performed in some serial order with respect to these participants [Kaiser 90]. A set of transactions, with a particular set of participants, is called a *domain*[1]. Other users remain *observers*, and this set of transactions must appear serial to these users. Participation is always with respect to some specific set of transactions. A particular user may be a participant for some transactions and an observer for others that access the same objects.

A user can nest subtransactions to carry out subtasks or to consider alternatives. All such subtransactions may be part of an implicit domain, with the one user as sole participant. Alternatively, one or more explicit domains — perhaps with multiple participants — may be created for subsets of the subtransactions. In the case of an implicit domain, there is no requirement for serializability among the subtransactions. However, such a subtransaction must appear atomic with respect to any participants, other than the controlling user, in the parent transaction's domain.

[1]The word domain means different things in participant transactions, visibility domains, and domain relative addressing.

The domain in which a user participates would typically be the set of transactions associated with the members of a cooperating group of users working towards a common goal. However, there is no implication that all the transactions in the domain commit together, or even that all of them commit (some may abort). Thus it is misleading to think of the domain as a top-level transaction, with each user's transaction as a subtransaction, although this is likely to be a frequent case in practice.

Each transaction is associated with zero or one particular domains at the time it is begun. A transaction that is not placed in any domain is the same as a classical (but interactive) transaction, with no participants except the one user. Such a transaction must be serializable with respect to all other transactions in the system. A transaction is placed in a domain in order to non-serializably share partial results with other transactions in the same domain, but it must be serializable with respect to all transactions not in the domain.

	T_{John}	T_{Mary}
|	begin(X)	
|	access(A)	
|	read(B)	begin(X)
|	write(B)	access(C)
|		read(B)
|	access(A)	write(B)
|	read(B)	
|	write(B)	
|		commit(B,C)
|	read(B)	
|	write(B)	
|	commit(A,B)	
V		
Time		

Figure 19: Example Participation Schedule

To illustrate, say a domain X is defined to respond to a particular modification request, and programmers Mary and John begin transactions T_{Mary} and T_{John} associated with X. Assume that an *access* operation is either a read or a write operation. The schedule shown in Figure 19 is not serializable according to any of the conventional concurrency control mechanisms. T_{Mary} reads the updates that T_{John} made to module B that are written but are not yet committed by T_{John}, modifies parts of module B, and then commits. T_{John} continues to modify modules A and B after T_{Mary} has committed. Since Mary and John participate in the same domain X, this is legal with respect to T_{Mary} and T_{John}, and serializable according to the participation transactions mechanism.

Now say that Bob starts a transaction T_{Bob}that is not associated with domain X, and the

T_{John}	T_{Mary}	T_{Bob}
begin(X)		begin
modify(A)		modify(C)
modify(B)	begin(X)	
read(C)	access(D)	
write(C)	read(B)	
modify(A)	write(B)	
read(B)		
modify(B)		
	commit(B,D)	
modify(B)		
		read(B)

Time

Figure 20: Example Participation Conflict

sequence of events shown in Figure 20 happens. Bob first modifies module C. This by itself would be legal, since T_{Bob} thus far could be serialized before T_{John}(but not after). But then T_{Bob} attempts to read the module B committed by T_{Mary}. This would be illegal, and thus is not permitted even though T_{Mary} was committed. T_{Mary} cannot be serialized before T_{Bob}, and thus before T_{John}, because T_{Mary} reads the uncommitted changes to module B written by T_{John}. In fact, T_{Mary} cannot be serialized either before or after T_{John}. This would not be a problem if it were never necessary to serialize T_{Mary} with any transactions outside the domain. Mary's update to module B would be irrelevant if John committed his final update to B before any transactions outside the domain accessed B. Thus the serializability of transactions within a participation domain need be enforced only with respect to what is actually observed by the users who are not participants in the domain.

9 SUMMARY AND DISCUSSION

This paper investigates concurrency control issues for a class of advanced database applications involving computer-supported cooperative work, concentrating primarily on software development environments, although the requirements of CAD/CAM environments and office automation are similar. The differences between concurrency control requirements in these advanced applications and traditional data processing applications are discussed, and several new mechanisms and policies that address these differences are presented. Many of these have not yet been implemented in any system. This is due to two factors: (1) many are theoretical frameworks rather than practical schemes; and (2) many of the more pragmatic schemes are so recent that there has not been a sufficient period of time to design and implement even prototype systems. Table 21 summarizes the discussions of the previous sections by indicating whether or not each mechanism or policy addresses long transactions, user control and cooperation, and naming a system, if any, in which the ideas have been implemented.

Mechanism	System	Long Trans.	User Control	Cooperation
Altruistic Locking	N/A	Yes	No	Limited
Snapshot Validation	N/A	Yes	No	Limited
Order-Preserving Transactions	DASDBS	Yes	No	Limited
Entity-State Transactions	N/A	Yes	No	Limited
Semantic Atomicity	N/A	Yes	No	Limited
Multilevel Atomicity	N/A	Yes	No	Yes
Sagas	N/A	Yes	No	Limited
Conflict-Based Serializability	N/A	Yes	No	Limited
Commit-Serializability	N/A	Yes	Yes	Limited
Reserve/Deposit	RCS	No	No	Limited
Conversational Transactions	System R	Limited	Limited	No
Multilevel Coordination	Infuse	Yes	Yes	Limited
Domain Relative Addressing	Cosmos	Yes	No	Limited
Copy/Modify/ Merge	NSE	Yes	Yes	Limited
Multiple Commit Points	N/A	Yes	Yes	Limited
Interactive Notification	Gordion	No	Limited	Limited
Visibility Domains	N/A	Yes	Limited	Yes
Group-Oriented CAD Trans.	N/A	Yes	Limited	Yes
Cooperating CAD Transactions	Orion	Yes	Limited	Yes
Transaction Groups	ObServer II	Limited	Limited	Yes
Participant Transactions	N/A	Yes	Limited	Yes

Figure 21: Advanced database systems and their concurrency control schemes

There are four other concerns that extended transaction models for advanced applications should also address. These are: (1) the interface to and requirements for the underlying DBMS; (2) the interface to the application tools and environment kernel; (3) the end-user interface; and (4) the environment/DBMS administrator’s interface. In a software development environment, for example, there are a variety of tools that need to retrieve different kinds of objects from the database. A tool that builds the executable code of the whole project might access the most recent version of all objects that are of type *code*. Another tool, for document preparation, accesses all the objects of type *document* or of type *description* in order to produce a user manual. There might be several relationships between documents and code (a document describing a module may have to be modified if the code of the module is changed, for instance). Users collaborating on a project invoke tools as they go along in their sessions, which might result in tools being executed concurrently. In such a situation, the transaction manager, which controls concurrent access to the database, must "understand" how to provide each user and each tool with access to a consistent set of objects that they operate on, where consistency is defined according to the needs of the application. The transaction manager must mediate retrieval and storage of these objects in the underlying database.

A problem that remains unresolved is the lack of performance metrics that evaluate the proposed policies and mechanisms in terms of the efficiencies of both implementation and use. We have not come across any empirical studies that investigate the needs of developers working together on the same project and how different concurrency control schemes might affect the development process and the productivity of developers. It might very well be that some of the schemes that appear adequate theoretically will turn out to be very inefficient and/or unproductive for the purposes of a particular family of software engineering practices. But it is not clear how to define appropriate measures.

Another problem is that most of the notification schemes are limited to (1) attaching the notification mechanism to the locking primitives and (2) notifying human users, generally about the availability of resources. These schemes assume that only the human user is active and that the database is just a repository of passive objects. It is important, however, for the DBMS of an advanced application to be active in the sense that it be able to monitor the activities in the database and automatically perform some operations in response to changes made to the database (this is what the database community calls *triggers* [Stonebraker et al. 88]). Notification must be expanded, perhaps in combination with triggers, to detect a wide variety of database conditions, to consider indirect as well as direct consequences of database updates, and to notify appropriate monitor and automation elements provided by the software development environment.

In addition to supporting automation, advanced applications like SDEs typically provide the user with capabilities to execute queries about the status of the development process. By definition, this requires access to the internal status of in-progress tasks or transactions, perhaps restricted to distinguished users such as managers. If a manager of a design project needs to determine the exact status of the project in terms of what has been completed (and how far the schedule has slipped!), the database management system must permit access to subparts of tasks that are still in-progress and not yet committed. Furthermore, the queries must be precise in reflecting a consistent state of the database, in the sense that no other activities may be concurrently writing the database (the brief lack of concurrency in this case may be deemed acceptable to fulfill managerial goals).

One key reason why traditional concurrency control mechanisms are too restrictive for advanced applications is that they do not make use of the available semantics. Many of the extended transaction models presented in this paper do use some kind of information about transactions, such as their access patterns, and about users, such as which design group they belong to. Most, however, do not define or use the semantics of the task that a transaction is intended to perform, or the semantics of database operations in terms of when an operation is applicable, what effects it has and what implications it has for leading to future database operations. Consequently, these mechanisms capture only a subset of the interactions possible in advanced applications. One approach to solving this problem is to define a formal model that can characterize the whole range of interactions among transactions. This approach was pursued in developing the ACTA framework, which is capable of specifying both the structure and behavior of transactions, as well as concurrency and recovery properties [Chrysanthis and Ramamritham 90].

Although all of the extended transaction models presented in the paper address at least one of the concurrency control requirements, which include supporting long duration transactions, user control over transactions and cooperation among multiple users, none of them supports all requirements. For example, some mechanisms that support long transactions, such as altruistic locking, do not support user control. Some mechanisms that support user control, such as optimistic coordination, do not directly support cooperation. All three requirements must be fulfilled for the class of advanced applications considered here, those involving computer-supported cooperative work.

ACKNOWLEDGMENTS

We would like to thank Terrance Boult and Soumitra Sengupta for reviewing earlier versions of this paper. We would also like to thank the anonymous referees who provided many detailed suggestions and corrections. Barghouti is supported in part by the Center for Telecommunications Research. Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research. The authors can be contacted at naser@cs.columbia.edu and kaiser@cs.columbia.edu.

REFERENCES

[Adams et al. 89] Adams, E. W., Honda, M., and Miller, T. C. Object Management in a CASE Environment. Proceedings of the 11th International Conference on Software Engineering, IEEE Computer Society Press, May, 1989, pp. 154-163.

[Bancilhon et al. 85] Bancilhon, F., Kim, W., and Korth, H. A Model of CAD Transactions. Proceedings of the 11th International Conference on Very Large Data Bases, Morgan Kaufmann, August, 1985, pp. 25-33.

[Beeri et al. 86] Beeri, C., Bernstein, P. A., and Goodman, N. A Model for Concurrency in Nested Transaction Systems. Technical Report TR-86-03, The Wang Institute for Graduate Studies, Tyngaboro, MA, March, 1986.

[Beeri et al. 88] Beeri, C., Schek, H. -J., and Weikum, G. Multilevel Transaction Management, Theoretical Art or Practical Need? Advances in Database Technology - EDBT ’88, 1988.

[Beeri et al. 89] Beeri, C., Bernstein, P. A., and Goodman, N. ‘‘A Model for Concurrency in Nested Transaction Systems .’’ *Journal of the ACM 36*, 1, 1989.

[Bernstein 87] Bernstein, P. Database System Support for Software Engineering -- An Extended Abstract. Proceedings of the 9th International Conference on Software Engineering, March, 1987, pp. 166-178.

[Bernstein and Goodman 81] Bernstein, P., and Goodman, N. ‘‘Concurrency Control in Distributed Database Systems.’’ *ACM Computing Surveys 13*, 2, June 1981, pp. 185-221.

[Bernstein et al. 87] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[Bjork 73] Bjork, L. A. Recovery Scenario for a DB/DC System. Proceedings of the 28th ACM National Conference, ACM Press., August, 1973, pp. 142-146.

[Chrysanthis and Ramamritham 90] Chrysantis. P. K., and Ramamritham, K. . Acta: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data, ACM Press, May, 1990, pp. 194-203.

[Davies 73] Davies, C. T. Recovery Semantics for A DB/DC System. Proceedings of the 28th ACM National Conference, ACM Press., August, 1973, pp. 136-141.

[Davies 78] Davies, C. T. ‘‘Data Processing Spheres of Control.’’ *IBM System Journal 17*, 2, 1978, pp. 179-198.

[Dittrich et al. 87] Dittrich, K., Gotthard, W., and Lockemann, P. ‘‘DAMOKLES -- The Database System for the UNIBASE Software Engineering Environment.’’ *IEEE Bulletin on Database Engineering 10*, 1, March 1987, pp. 37-47.

[Dowson and Nejmeh 89] Dowson, M., and Nejmeh, B. Nested Transactions and Visibility Domains. Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases, ACM Press, February, 1989, pp. 36-38. Position paper.

[Eastman 80] Eastman, C. System Facilities for CAD Databases. Proceedings of the 17th ACM Design Automation Conference, ACM Press, June, 1980, pp. 50-56.

[Eastman 81] Eastman, C. Database Facilities for Engineering Design. Proceedings of the IEEE, IEEE Computer Society Press, October, 1981, pp. 1249-1263.

[Ege and Ellis 87] Ege, A., and Ellis, C. A. . Design and implementation of Gordion, an Object Base Management System. Proceedings of the 3rd International Conference on Data Engineering, IEEE Computer Society Press, February, 1987, pp. 226-234.

[El Abbadi and Toueg 89] El Abbadi, A. and Toueg, S. ''The Group Paradigm for Concurrency Control Protocols.'' *IEEE Transactions on Knowledge and Data Engineering 1*, 3, September 1989, pp. 376-386.

[Eswaran et al. 76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I. ''The Notions of Consistency and Predicate Locks in a Database System.'' *Communications of the ACM 19*, 11, November 1976, pp. 624-632.

[Feldman 79] Feldman, S. I. ''Make — A Program for Maintaining Computer Programs.'' *Software — Practice & Experience 9*, 4, April 1979, pp. 255-265.

[Fernandez and Zdonik 89] Fernandez, M. F., and Zdonik, S. B. Transaction Groups: A Model for Controlling Cooperative Work. Proceedings of the 3rd International Workshop on Persistent Object Systems: Their Design, Implementation and Use, January, 1989, pp. 128-138.

[Garcia-Molina 83] Garcia-Molina, H. ''Using Semanti c Knowledge for Transaction Processing in a Distributed Database.'' *ACM Transactions on Database Systems 8*, 2, June 1983, pp. 186-213.

[Garcia-Molina and Salem 87] Garcia-Molina, H., and Salem, K. SAGAS. Proceedings of the ACM SIGMOD 1987 Annual Conference, ACM Press, May, 1987, pp. 249-259.

[Garza and Kim 88] Garza, J., and Kim, W. Transaction Management in an Object-Oriented Database System. Proceedings of the ACM SIGMOD International Conference on the Management of Data, ACM Press, June, 1988, pp. 37-45.

[Gray 78] Gray, J. Notes On Database Operating Systems. IBM Research Report RJ2188, IBM Research Laboratory, San Jose, CA, 1978.

[Gray et al. 75] Gray, J., Lorie R., and Putzolu, G. Granularity of Locks and Degrees of Consistency in a Shared Database. IBM Research Report RJ1654, IBM Research Laboratory, San Jose, CA, 1975.

[Kaiser 89] Kaiser, G. E. A Marvelous Extended Transaction Processing Model. Proceedings of the 11th World Computer Conference IFIP Congress '89, Elsevier Science Publishers B.V., August, 1989, pp. 707-712.

[Kaiser 90] Kaiser, G. E. A Flexible Transaction Model for Software Engineering. Proceedings of the 6th International Conference on Data Engineering, IEEE Computer Society Press, February, 1990, pp. 560-567.

[Kaiser and Feiler 87] Kaiser, G. E., and Feiler, P. H. Intelligent Assistance without Artificial Intelligence. Proceedings of the 32nd IEEE Computer Society International Conference, IEEE Computer Society Press, February, 1987, pp. 236-241.

[Kaiser and Perry 87] Kaiser, G. E., and Perry, D. E. Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution. Proceedings of the Conference on Software Maintenance, September, 1987, pp. 108-114.

[Katz and Weiss 84] Katz, R., and Weiss, S. Design Transaction Management. Proceedings of the ACM IEEE 21st Design Automation Conference, IEEE Computer Society Press, June, 1984, pp. 692-693.

[Kim et al. 84] Kim, W., Lorie, R. A., McNabb, D., and Plouffe, W. A Transaction Mechanism for Engineering Databases. Proceedings of the 10th International Conference on Very Large Data Bases, Morgan Kaufmann, August, 1984, pp. 355-362.

[Kim et al. 88] Kim, W., Ballou, N., Chou, H., and Garza, J. Integrating an Object-Oriented Programming System with a Database System. Proceedings of the 3rd International Conference on Object Oriented Programming Systems, Languages and Applications, September, 1988, pp. 142-152.

[Klahold et al. 85] Klahold, P.,Schlageter, G., Unland, R., and Wilkes, W. A Transaction Model Supporting Complex Applications in Integrated Information Systems. Proceedings of the ACM SIGMOD International Conference on the Management of Data, ACM Press, May, 1985, pp. 388-401.

[Kohler 81] Kohler, W. ‘‘A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems.’’ *ACM Computing Surveys 13*, 2, June 1981, pp. 149-183.

[Korth and Silberschatz 86] Korth, H., and Silberschatz, A. *Database System Concepts.* McGraw-Hill Book Company, New York, NY, 1986.

[Korth and Speegle 88] Korth, H., and Speegle, G. Formal Model of Correctness Without Serializability. Proceedings of the ACM SIGMOD International Conference on the Management of Data, ACM Press, June, 1988, pp. 379-386.

[Kung and Robinson 81] Kung, H., and Robinson, J. ‘‘On Optimistic Methods for Concurrency Control.’’ *ACM Transactions on Database Systems 6*, 2, June 1981, pp. 213-226.

[Kutay and Eastman 83] Kutay, A., and Eastman, C. Transaction Management in Engineering Databases. Proceedings of the Annual Meeting of Database Week; Engineering Design Applications, IEEE Computer Society Press, May, 1983, pp. 73-80.

[Leblang and Chase, Jr. 87] Leblang, D. B., and Chase, R. P., Jr. ‘‘Parallel Software Configuration Management in a Network Environment.’’ *IEEE Software 4*, 6, November 1987, pp. 28-35.

[Lorie and Plouffe 83] Lorie, R., and Plouffe, W. Complex Objects and Their Use in Design Transactions. Proceedings of the Annual Meeting of Database Week; Engineering Design Applications, IEEE Computer Society Press, May, 1983, pp. 115-121.

[Lynch 83] Lynch, N. A. ‘‘Multilevel Atomicity — A New Correctness Criterion for Database Concurrency Control.’’ *ACM Transactions on Database Systems 8*, 4, December 1983, pp. 484-502.

[Martin 87] Martin, B. Modeling Concurrent Activities with Nested Objects. Proceedings of the 7th International Conference on Distributed Computing Systems, IEEE Computer Society Press, September, 1987, pp. 432-439.

[Moss 85] Moss, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge, MA, 1985.

[Nestor 86] Nestor, J. R. Toward a Persistent Object Base. In *Advanced Programming Environments*, Conradi, R., Didriksen, T. M., and Wanvik, D. H., Eds., Springer-Verlag, Berlin, 1986, pp. 372-394.

[Papadimitriou 86] Papadimitriou, C. *The Theory of Database Concurrency Control.* Computer Science Press, Rochville, MD, 1986.

[Pradel et al. 86] Pradel, U., Schlageter, G., and Unland, R. Redesign of Optimistic Methods: Improving Performance and Availability. Proceedings of the 2nd International Conference on Data Engineering, IEEE Computer Society Press, February, 1986, pp. 466-473.

[Pu et al. 88] Pu, C., Kaiser, G., and Hutchinson, N. Split Transactions for Open-Ended Activities. Proceedings of the 14th International Conference on Very Large Databases, Morgan Kaufmann, August, 1988, pp. 26-37.

[Reed 78] Reed, R. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. Thesis, MIT Laboratory of Computer Science, September 1978. MIT Technical Report 205

[Rochkind 75] Rochkind, M. J. ‘‘The Source Code Control System.’’ *IEEE Transactions on Software Engineering SE-1*, 1975, pp. 364-370.

[Rosenkrantz et al. 78] Rosenkrantz, D., Stearns, R., and Lewis P. ‘‘System level concurrency control for distributed database systems.’’ *ACM Transactions on Database Systems 3*, 2, June 1978, pp. 178-198.

[Rowe and Wensel 89] *1989 ACM SIGMOD Workshop on Software CAD Databases*, Napa, CA, February, 1989.

[Salem et al. 87] Salem, K., Garcia-Molina, H., and Alonso, R. Altruistic Locking: A Strategy for Coping with Long Lived Transactions. Proceedings of the 2nd International Workshop on High Performance Transaction Systems , September, 1987, pp. 19.1 - 19.24.

[Skarra and Zdonik 89] Skarra, A. H., and Zdonik, S. B. Concurrency Control and Object-Oriented Databases. In Kim, W., and Lochovsky, F. H., Ed., *Object-Oriented Concepts, Databases, and Applications*, ACM Press, New York, NY, 1989, pp. 395-421.

[Stonebraker et al. 88] Stonebraker, M., Katz, R., Patterson, D. and Ousterhout, J. The Design of XPRS. Proceedings of the 14th International Conference on Very Large Databases, Morgan Kaufmann, August, 1988, pp. 318-330.

[Tichy 85] Tichy, W. F., ‘‘RCS — A System for Version Control.’’ *Software — Practice and Experience 15*, 7, July 1985, pp. 637-654.

[Walpole et al. 87] Walpole, J., Blair, G., Hutchison, D., and Nicol, J. ‘‘Transaction mechanisms for distributed programming environments.’’ *Software Engineering Journal 2*, 5, September 1987, pp. 169-177.

[Walpole et al. 88a] Walpole, J., Blair, G., Malik, J., and Nicol, J. Maintaining Consistency in Distributed Software Engineering Environments. Proceedings of the 8th International Conference on Distributed Computing Systems, IEEE Computer Society Press, June, 1988, pp. 418-425.

[Walpole et al. 88b] Walpole, J., Blair, G., Malik, J., and Nicol, J. A Unifying Model for Consistent Distributed Software Development Environments. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM Press, November, 1988, pp. 183-190.

[Walter 84] Walter, B. Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications. Proceedings of the 10th International Conference on Very Large Data Bases, Morgan Kaufmann, August, 1984, pp. 161-171.

[Weikum 86] Weikum, G. A Theoretical Foundation of Multi-level Concurrency Control. Proceedings of the 5th ACM Symposium on Principles of Database Systems, ACM Press, March, 1986, pp. 31-42.

[Weikum and Schek 84] Weikum, G., and Schek, H. -J. Architectural Issues of Transaction Management in Multi-Level Systems. Proceedings of the 10th International Conference on Very Large Data Bases, Morgan Kaufmann, August, 1984.

[William et al. 81] Williams, R., et al. R^*: An Overview of the Architecture. IBM Research Laboratory, San Jose, CA, December, 1981.

[Yannakakis 82] Yannakakis, M. ‘‘Issues of Correctness in Database Concurrency Control by Locking.’’ *Journal of the ACM 29*, 3, July 1982, pp. 718-740.

[Yeh et al. 87] Yeh, S., Ellis, C., Ege, A., and Korth, H. Performance Analysis of Two Concurrency Control Schemas for Design Environments. Technical Report STP-036-87, MCC, Austin, TX, June, 1987 .

Table of Contents

List of Figures