Concurrency Control in Advanced Database Applications

NASER S. BARGHOUTI AND GAIL E. KAISER

Department of Computer Science, Columbia University, New York, New York 10027

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

General Terms: Algorithms, Design, Management

Additional Key Words and Phrases: Advanced database applications, concurrency control, cooperative transactions, design environments, extended transaction models, long transactions, object-oriented databases, 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 database is maintained if every data item satisfies specific consistency constraints. These are typically implicit in data processing in the sense they are 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 to the same data item by multiple transactions. 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

CONTENTS

systems, such as the ones mentioned above, that use DBMS capabilities. They are termed *advanced* to distinguish them from traditional database applications such as banking and airline reservations systems. In traditional applications, 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, have different kinds of consistency constraints, and, in general, the classical transaction model is not applicable. For example, applications like network management and medical informatics may require real-time processing. Others like

CAD/CAM and office automation involve long, interactive database sessions and cooperation among multiple database users. Conventional concurrency control mechanisms are not applicable "as is" in these new domains. This paper is concerned with the latter class of advanced applications, which involve computer-supported cooperative work. The requirements of these applications 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 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. The term *transaction*, however, provides a nice intuition regarding the need for consistency, concurrency control, and fault recovery. Basic OLTP concepts such as locks, versions, and validation provide a good starting point for the implementation of long transactions and cooperating transactions. In any case, nearly all the relevant literature uses the term *transaction*. We do likewise in our survey.

The goals of this paper are to provide a basic understanding of the difference between concurrency control in advanced database applications and in traditional data processing applications, to outline mechanisms used to control concurrent access in these advanced applications, and to point out some problems with these mechanisms. We assume the reader is familiar with database concepts but do not assume an in-depth understanding of transactions and concurrency control issues. Throughout the paper we define the concepts we use and give practical examples of them. We explain the mechanisms at an intuitive level rather than at a detailed technical level.

The paper is organized as follows. Section 1 presents 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 may wish to skip Sections 3 and 4. Section 5 enumerates the concurrency control requirements of advanced database applications. It 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 and concludes with a summary of the mechanisms.

1. 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 are 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. He "reserves" A and starts
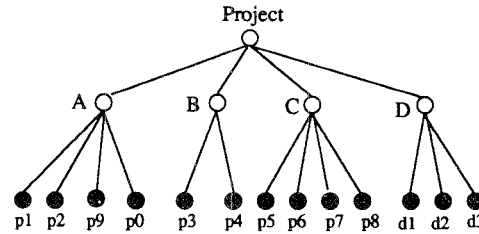

Figure 1. Organization of example project.

working on it. Mary's task is to explore a possible bug in the code of module B, so she starts browsing B after "reserving" it. After a while, John finds there is a bug in A caused by bugs in some of the procedures in the library module, so he "reserves" module D. 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 then wants to test the new code of B. She is not concerned with the modifications John made in A because module A is unrelated to module B. She does, however, want to access the modifications John made in module D because the procedures in D are called in module B. The modifications John made to D might have introduced inconsistencies to the code of module B. But since John is still working on modules A and D, Mary will either have to access module D at the same time John is modifying it or wait until he is done.

In the above example, if the traditional concurrency control scheme of two-phase locking was used, for example, John and Mary would not have been 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, for example, to recompile

D. The locks are released only 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 also 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 of module D while John works on a new version. This requires Mary to retest her code after the new version of D is released, which is really unnecessary. What is needed is a flexible concurrency control scheme that allows cooperation between 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 multiuser software systems, such as software development environments, generate and manipulate large amounts of data. SDEs, for example, generate and manipulate source code, object code, documentation, test suites, and so on. Traditionally, users of such systems manage the data they generate either manually or by the use of special-purpose tools. For example, programmers working on a large-scale software project use system configuration management tools such as Make [Feldman 1979] and RCS [Tichy 1985] 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 among all these tools is the file system, which stores project components in text or binary files regardless of their internal structures. This significantly limits the ability to manipulate these objects in desirable ways. It also causes inefficiencies in the storage of collections of objects

and leaves data, stored as a collection of related files, susceptible to corruption due to incompatible concurrent access.

Recently, researchers have attempted to use database technology to manage the objects belonging to a system uniformly. 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 1987; Dittrich et al. 1987; Nestor 1986; Rowe and Wensel 1989]:

(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 all the data satisfy the integrity constraints required by the application.

(4) Convenient access. Providing a powerful query language to access multiple sets of data items at a time.

(5) Data independence. Hiding the internal structure of data from tools so that if the structure is changed, it will have a minimal impact on the applications using the data.

Since there are numerous commercial DBMSs 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 advanced applications [Bernstein 1987; Korth and Silberschatz 1986]. One of the shortcomings of traditional general-purpose DBMSs is their inability to provide flexible concurrency control mechanisms. To understand the reasons behind this, we need to explain the concepts of *transactions* and *serializability*. These two concepts are central to all conventional concurrency control mechanisms.

3. CONSISTENCY PROBLEMS IN CONVENTIONAL DBMSs

Database consistency is maintained if every data item in the database satisfies the application-specific consistency constraints. For example, in an airline reservations 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 the consistency constraints are not known beforehand to the designers of general-purpose DBMSs. This is due to the lack of information about the computations in potential applications and the semantics of database operations in these applications. Thus, the best a DBMS can do is abstract each database operation to be either a read operation or a write operation, irrespective of the particular computation. Then it can guarantee the database is always in a consistent state with respect to reads and writes, independent of the semantics of the particular application.

Ignoring the possibility of bugs in the DBMS program and the application program, inconsistent data result from two main sources: software or hardware failures such as bugs in the operating system or a disk crash in the middle of operations and 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 program accessing the database are grouped into sequences called *transactions* [Eswaran et al. 1976]. Users interact with a DBMS by executing transactions. In traditional DBMSs, transactions serve three distinct purposes [Lynch 1983]: (1) They are logical units that group together operations comprising 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 multiuser system, users execute their transactions concurrently. The DBMS must 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 that each of the transactions submitted to the DBMS by a user eventually gets executed and that the results of the computations performed by each transaction are the same whether it is executed on a dedicated system or concurrently with other transactions in a multiprogrammed system [Bernstein et al. 1987; Papadimitriou 1986].

Let us follow up our previous example to demonstrate the transaction concept. John and Mary are now 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 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 to $p1$, 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 as they would have been 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
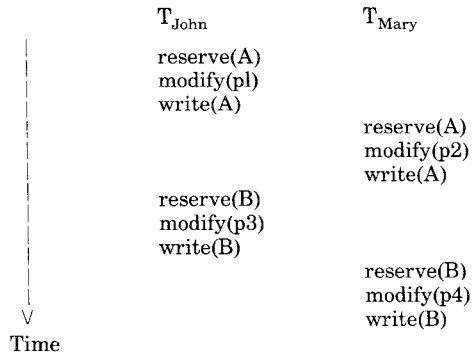
T_{John} T_{Mary}

reserve(A)
modify(pl)
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.

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 transactions 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. A schedule is consistent if the transactions comprising the schedule are executed serially. In other words, a schedule consisting of transactions $T_1, T_2, \ldots, T_n$ is consistent if for every $i = 1$ to $n - 1$, transaction T_i is executed to completion before transaction T_{i+1} begins. We can then establish that a serializable execution, one that is equivalent to a serial execution, is also consistent. 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 1986]:

(1) The set of transactions that participates in S_1 and S_2 is 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, 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 write(Q) be-

fore T_j executes write(Q), the same will hold in S_2 (i.e., write–write synchronization).

For example, the schedule shown in Figure 2 is computationally equivalent to the serial schedule T_{John}, T_{Mary} (execute T_{John} to completion then execute T_{Mary}) because the set of transactions in both schedules are the same, both data items A and B read by T_{Mary} are written by T_{John} in both schedules, and T_{Mary} executes both write(A) and write(B) after T_{John} 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 one. 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 1981]. 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. The rw synchronization refers to serializing transactions in such a way that every read operation reads the same value of a data item as it would have read in a serial execution. The ww synchronization refers to serializing transactions so the last write operation of every transaction leaves the database in the same state as it would have left it in a serial execution. The rw and ww synchronizations together result in a consistent schedule.

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 is available, schedules that are not serializable but do maintain can could be permitted. This is exactly the goal of the *extended transaction* mechanisms discussed later.

4. TRADITIONAL APPROACHES TO CONCURRENCY CONTROL

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 in conventional DBMSs. Most of the mechanisms follow one of four main approaches to concurrency control: two-phase locking (the most popular example of locking protocols), timestamp ordering, multiversion timestamp ordering, and optimistic concurrency control. Some mechanisms add multiple granularities of locking and nesting of transactions. In this section, we briefly describe these approaches. There have been a few comprehensive discussions and surveys of traditional concurrency control mechanisms, including Bernstein and Goodman [1981] and Kohler [1981]; a book has also been written on the subject [Bernstein et al. 1987].

4.1 Locking Mechanisms

4.1.1 Two-Phase Locking

The two-phase locking mechanism (2PL) introduced by Eswaran et al. [1976] is now accepted as the standard solution to the concurrency control problem in conventional DBMSs. 2PL guarantees serializability in a centralized database when transactions are executed concurrently. The mechanism depends on well-formed transactions, which do not relock entities that have been locked earlier in the transaction and are divided into a growing phase in which locks are only acquired and a shrinking phase, in which locks are only released. 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.

4.1.2 Tree Protocol

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 1982]. In advanced applications, it is often the case that the DBMS has prior knowledge about the order of access of data items. The DBMS can use this information to ensure serializability by using locking protocols that are not 2PL. One such protocol is the tree protocol, which can be applied if there is a partial ordering on the set of data items accessed by concurrent transactions [Silberschatz and Kedem 1980]. To illustrate this protocol, assume a third programmer, Bob, joined the programming team of Mary and John and is now working with them on the same project. Suppose 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 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 is the basis for allowing the non-2PL schedule shown in the figure.

4.2 Timestamp Ordering

One of the problems of locking mechanisms is the potential for deadlock. Deadlock 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
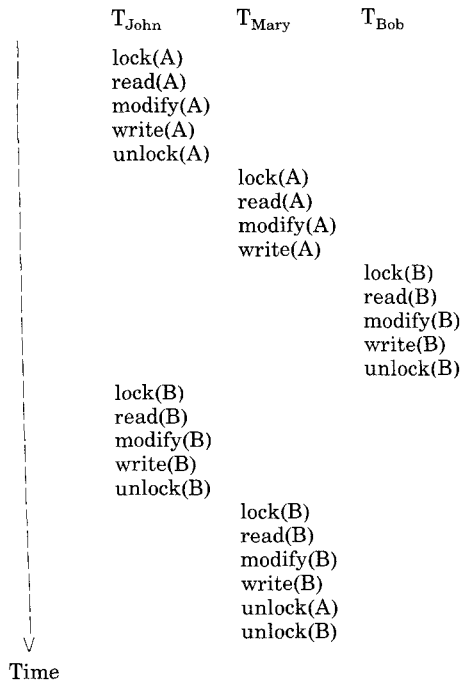
Schedule S1:

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)	

Time

Figure 3. Serializable but not 2PL schedule.

sequence. This sequence is often a function of the time of day [Kohler 1981]. Using timestamps, a concurrency control mechanism can totally order requests from transactions according to the transactions' timestamps [Rosenkrantz et al. 1978]. The mechanism forces a transaction T_1 requesting to access a data item x that is being held by another transaction T_2 to wait until T_2 terminates, abort itself and restart if it cannot be granted access to x, or preempt T_2 and get hold of x. A scheduling protocol decides which one of these three actions to take after comparing the timestamps of T_1 and T_2.

Two of the possible alternative scheduling protocols used by timestamp-based mechanisms are 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 the WOUND-WAIT protocol, which al-

lows 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 both protocols since some transactions are forced to wait as if they were locked out. Both protocols guarantee that a deadlock situation will not arise.

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 restriction can be relaxed by allowing multiple transactions to read and write different versions of the same data item as long as each transaction sees a consistent set of versions for all the data items it accesses. This is the basic idea of the first multiversion time-stamp ordering scheme introduced by Reed [1978]. In Reed's mechanism, each transaction is assigned a unique time-stamp when it starts; all operations of the transaction are assigned the same timestamp. In addition, each data item x has a set of transient versions, each of which is a $\langle writetimestamp, value \rangle$ pair, and a set of read timestamps. If a transaction reads a data item, the transaction's timestamp is added to the set of read timestamps of the data item. A write operation, if permitted by the concurrency control protocol, causes the creation of a new transient version with the same time-stamp as that of the transaction requesting the write operation. The concurrency control mechanism operates as follows:

Let T_i be a transaction with time-stamp TS(i), and let $R(x)$ be a read operation requested by T_i [i.e., $R(x)$ will also be assigned the timestamp TS(i)]. $R(x)$ is processed by reading a value of the version of x whose timestamp is the largest timestamp smaller than TS(R) (i.e., the latest value written before T_i started). TS(i) is then added to the set of read timestamps of x. Read operations are always permitted. Write operations,

in contrast, might cause a conflict. Let T_j be another transaction with timestamp TS(j), and let W(x) be a write operation requested by T_j that assigns value v to item x. W(x) will be permitted only if other transactions with a more recent timestamp than TS(j) have not read a version of x whose timestamp is greater than TS(j).

A situation like this can occur because of the delays in executing operations within a transaction. That is, an operation O_j belonging to transaction T_j is executed a specified period of time after T_j has started. Meanwhile, other operations from a transaction with a more recent time-stamp might have been performed. In order to detect such situations, let *interval*(W) be the interval from TS(j) to the smallest timestamp of a version of x greater than TS(j) (i.e., a version of x that was written by a transaction whose timestamp is more recent than T_j's timestamp). If any read time-stamps lie in the interval [i.e., a transaction has already read a value of x written by a more recent write operation than W(x)], then W(x) is rejected (and the transaction is aborted). Otherwise, W(x) is allowed to create a new version of x with timestamp TS(j).

The existence of multiple versions eliminates the need for write–write synchronization since each write operation produces a new version and thus cannot conflict with another write operation. The only possible conflicts are those corresponding to read-from relationships [Bernstein et al. 1987], as demonstrated by the protocol above.

4.4 Optimistic Nonlocking 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 1981]:

(1) Lock maintenance represents an unnecessary overhead for read-only transactions, which do not affect the integrity of the database.

(2) There are no locking mechanisms that provide high concurrency in all cases. Most of the general-purpose, deadlock-free locking mechanisms work well only in some cases but perform rather poorly in other cases.

(3) When large parts of the database reside 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 released except at 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 cases.

To avoid these disadvantages, Kung and Robinson [1981] presented the concept of "optimistic" concurrency control. 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 violate serializability with respect to all committed transactions, the local copies are made global. Only then, in the write phase, do these copies become accessible to other transactions.

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 older than the timestamp of transaction T_j, 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 times 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 concurrent transactions overlap. For example, Kung and Robinson's protocol would cause one of the 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 for maintaining consistency is a serious disadvantage. Since operations in advanced applications are generally long-lived (e.g., compiling a module), rolling them back and restarting them wastes all the work these operations did (the object code produced by compilation). The in-appropriateness of rolling back a long transaction in advanced applications is discussed further in Section 5.

4.5 Multiple Granularity Locking

The concurrency control protocols described so far operate on individual data items to synchronize transactions. It is sometimes desirable, however, to be able to access a set of data items as a single unit. Gray et al. [1975] presented a *multiple granularity* concurrency control protocol that aims to minimize the number of locks used while accessing sets of objects in a database. In their model, Gray et al. organize data items in a tree where small items are nested within larger ones. Each nonleaf item represents the data associated with its descendants. This

is different from the tree protocol presented above in that the nodes of the tree 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 kinds of locks are 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 transactions to read the same node concurrently but prevents any updating of the node.

To determine whether to grant a lock on a node to a transaction, 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 an *intention* lock was introduced [Gray 1978]. 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 nonleaf 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 exclusive (X) mode. A shared and intention-exclusive (SIX) lock on a nonleaf 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 shown in Figure 4. The matrix is used to determine when to grant lock requests and when to deny them.

Gray et al. defined the following multiple granularity protocol based on the compatibility matrix:

(1) A transaction T_i can lock a node in S or IS mode only if all ancestors of the node are locked in either IX or IS mode by T_i.

	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.

(2) A transaction T_i can lock a node in X, SIX, or IX mode only if all the ancestors of the node are locked in either SIX or IX mode by T_i.

(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 the transaction, the transaction should not hold a lock on a node after releasing the locks on its ancestors.

The multiple granularity protocol increases concurrency and decreases overhead. This is especially true 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 [Garza and Kim 1988; Kim et al. 1988].

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 all 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 may be the more important one, but it is not addressed in detail here since the focus of this paper is on concur-

rency 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 [1973] and expanded by Bjork [1973]. Reed [1978] presented a comprehensive solution to the problem of composing transactions by formulating the concept of nested transactions. A nested transaction is a composition of a set of subtransactions; each subtransaction can itself be a nested transaction. To other transactions, only 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 a subtransaction can fail and be restarted or replaced by another subtransaction 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 1985].

As far as concurrency is concerned, the nested transaction model presented above
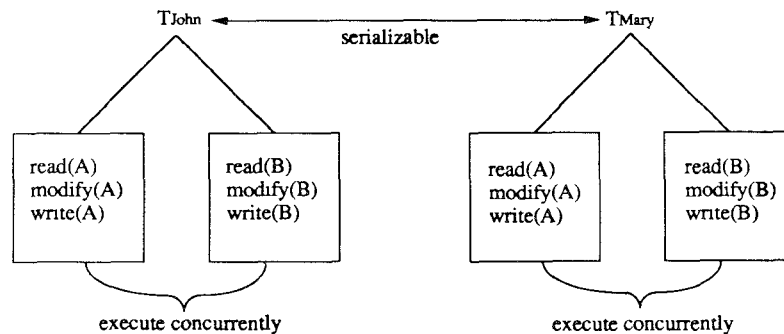
Figure 5. Scheduling nested transactions.

does not change the meaning of transactions (in terms of being atomic). It also does not alter the concept of serializability. The only advantage of nesting 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 many of the concurrency control mechanisms proposed for advanced database applications are based on combinations of optimistic concurrency control, multiversion objects, and nested transactions. To understand the reasons behind this, we must first address the concurrency control requirements of advanced database applications. We explore these requirements in Section 5; in the rest of the paper, we present several approaches that take these requirements into consideration.

5. CONCURRENCY CONTROL REQUIREMENTS IN ADVANCED DATABASE APPLICATIONS

Traditional DBMSs enforce serializable executions of transactions with respect to

read and write operations because of the lack of semantic knowledge about the application-specific operations. This leads to the inability to specify or check semantic consistency constraints on data. But there is nothing that makes a nonserializable schedule inherently inconsistent. If enough information is known about the transactions and operations, a nonserializable but consistent schedule can be constructed. In fact, equating the notions of consistency with serializability causes a significant loss of concurrency in advanced applications. In these applications, it is often possible to define specific consistency constraints. The DBMS can use these specifications rather than serializability as a basis for maintaining consistency. Several researchers have studied the nature of concurrent behavior in advanced applications and have arrived at new requirements for concurrency control [Bancilhon et al. 1985; Yeh et al. 1987]:

(1) Supporting long transactions. Operations on objects in design environments (such as compiling source code or circuit layout) are often long-lived. If these operations are embedded in transactions, these transactions, unlike traditional ones, will also be long-lived. Long transactions need different support than traditional 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 (e.g., bank audit transactions).

(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 or she wants to commit the work that some of the operations performed within the transaction violated some consistency constraints. The user would definitely oppose deleting all of the work (by rolling back the transaction). He or she might, however, be able to reverse the effects of some operations explicitly in order to regain consistency. Thus, there is a need to provide more user control over transactions.

(3) Supporting synergistic cooperation. Cooperation among programmers to develop project components has significant implications on concurrency control. In CAD/CAM systems, SDEs, and other design environments, several users might have to exchange knowledge (i.e., share it collectively) in order to be able to continue their work. The activities of two or more users working on shared objects may not be serializable. The users 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 parts of the same object concurrently, with the intent of integrating these parts to create a new version of the object. In this case, they might need to look at each others' work to make sure they are not modifying the two parts in a way that would make their integration difficult. This kind of sharing and exchanging knowledge was termed *synergistic interaction* by Yeh et al. To insist on serializable concurrency control in design environments might thus decrease concurrency or, more significantly, actually prevent 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 only 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, whereas others allow synergistic cooperation. The two classes of mechanisms are separated into two sections. Issues related to user control are briefly addressed by mechanisms in both categories, but we did not find any mechanism that provides satisfactory support for user control over transactions in advanced applications.

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 describing these mechanisms would require explaining concepts of object-oriented programming and object-oriented data-

base 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.

Many of the ideas implemented in the mechanisms we survey in the rest of the paper have 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 [1978].

6. SUPPORTING LONG TRANSACTIONS

Many of the operations performed on data in advanced database applications are long-lived. Some, such as compiling code or printing a complete layout of a VLSI chip, last for several minutes or hours. 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. 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) to avoid deadlock or in the case of failing validation in optimistic concurrency control.

Two main approaches have been pursued to solve these problems: extending serializability-based mechanisms while still maintaining serializable schedules and relaxing serializability of schedules containing LTs. These alternative approaches use 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 pertinent to advanced applications and thus are discussed in this section.

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 (2PL), for example, can be used to maintain consistency in any database system, regardless of the intended application. This is true because 2PL maintains serializability, and thus consistency, of transaction schedules by guaranteeing the atomicity of all transactions.

The performance of 2PL, however, is unacceptable for advanced applications because it forces LTs to lock resources for a long time even after they have finished using these resources. In the meantime, other transactions that need to access the same resources are blocked. Optimistic mechanisms that use timestamp ordering also suffer from performance problems when applied to long transactions. These mechanisms cause repeated rollback of transactions when the rate of conflicts increases significantly, which is generally the case in the context of long transactions.

One approach for solving the problems introduced by LTs is to extract semantic information about transactions and operations and use that information to extend traditional techniques. The extended technique should revert back to the traditional scheme in case the additional information is not available (i.e., it might be available for some transactions but not for others). This approach 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 they can be released and used by other transactions. This information can be used to allow a long transaction, which otherwise follows a serializable mechanism such as two-phase locking, to release some of its resources conditionally. These resources can then be used by other transactions given that they satisfy 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. 1987]. 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: negative access pattern information, which describes objects that will not be accessed by the transaction and 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. The set of all data items that have been locked and then released by an LT is called the *wake* of the transaction. Releasing a resource is a conditional unlock operation because it allows other transactions to access the released resource as long as they follow the restrictions stated in the protocol below, which ensures serializability.

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

(1) No two transactions can hold locks on the same data item simultaneously unless one of them has locked and released the object before the other locks 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, 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.

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

Consider again the example in Figure 1, where each module in the project contains a number of procedures (subobjects). Suppose Bob, who joined the programming team of Mary and John, wants to familiarize himself with the code of all the procedures of the project. Bob starts a long transaction, T_{Bob}, that accesses all of the procedures, one procedure at a time. He needs to access each procedure only once to read it and add some comments about the code; as he finishes accessing each procedure he releases it. In the meantime, John starts a short transaction, T_{John}, that accesses only two procedures, $p1$ then $p2$, from module A. Assume 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 that point T_{John} can start accessing $p1$ by entering the wake of T_{Bob}. T_{John} will be allowed to enter the wake of T_{Bob} (i.e., to be able to access $p1$) because all of the objects 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 since it has already been released by T_{Bob}.
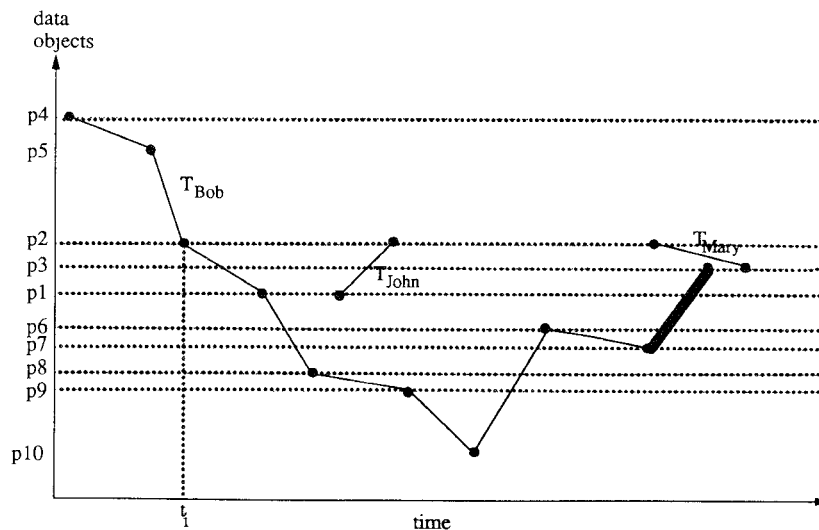
Figure 6. Access patterns of three transactions.

Now assume Mary starts another short transaction, T_{Mary}, that needs to access both $p2$ and a third procedure $p3$ that is not yet in the wake of T_{Bob}. 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 $p3$ (Bob changes his mind about viewing $p3$), 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. [1987] 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 add objects that will not be accessed by the long transaction to its wake. Continuing the example, the 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. Time is represented along the horizontal axis. The transactions belonging to Bob, John, and Mary are represented by solid lines. For example, T_{Bob} is represented by a solid line that passes through several black dots. Each black dot represents a data object (horizontal dotted lines connect black dots to objects they stand for). T_{Bob} accesses $p4$, $p5$, $p2$, $p1$, $p8$, $p9$, $p10$, $p6$ and $p7$ (the thick line extending to $p3$ is not part of T_{Bob}). T_{Bob} accesses $p2$ at time t_1 as indicated by the black dot at point $(t_1, p2)$ in the graph. T_{John} is in the wake of T_{Bob} totally because every object accessed by T_{John} ($p1$ and $p2$) was accessed before by T_{Bob}. This is not the case with T_{Mary}. In order to allow T_{Mary} to execute, the transaction expand the wake of T_{Bob} by adding $p3$ to it (as shown by the thick line), 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 altruistic locking is its ability to use the knowledge that a transaction no longer needs access to a data object it has locked. It maintains serializability and assumes the data stored in the database are of the conventional form. Furthermore, if access information is not available, any trans-
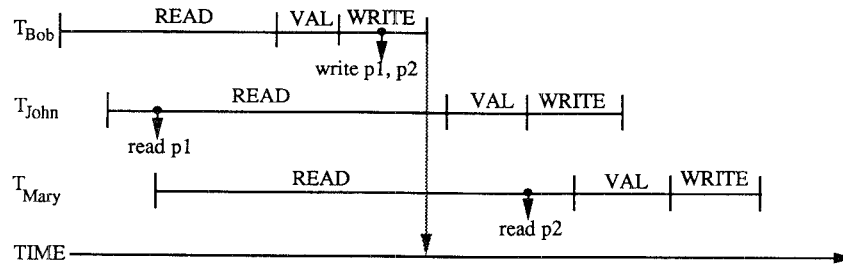
Figure 7. Validation conflicts.

action, at any time, can run under the conventional 2PL protocol without performing any special operations. As observed earlier, however, 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: When a long transaction aborts, all the short transactions in its wake have to be aborted even if they have 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 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 has to be passed in order to establish that the transaction did not produce conflicts with other committed transactions. The main shortcoming of the traditional validation technique is its weak definition of conflict. Because of this weak definition some transactions, such as those in Figure 2, are restarted unnecessarily. In other words, the transactions might actually have been serializable but the conflict mechanism did not recognize them as such. This is not a serious prob-

lem 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. [1986] observed that the risk of restarting a transaction can be reduced by distinguishing between *serious conflicts*, which require restart, and *nonserious conflicts*, which do not. They introduced a mechanism called *snapshot validation* that uses this approach.

Going back to our example, assume 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 phase of T_{John} and T_{Mary} will thus consider operations in T_{Bob}. According to the traditional optimistic concurrency control protocol, both T_{John} and T_{Mary} would have to be restarted because of conflicts. Procedures $p1$ and $p2$ that they read have been updated by T_{Bob}. T_{John} read $p1$, which was later changed by T_{Bob}; 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 to read 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. [1986] 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. For example, if object $p1$ is accessed before object $p2$ by the same transaction, then $p1$ appears before $p2$ in the transaction's 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 point in the read set are not considered 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. [1986] also analyzed the starvation problem in the conventional optimistic protocol and discovered that the longer the transaction, the greater the risk of starvation. 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. [1986] 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 created 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 any transaction does), 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 as 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 continue 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 et al. [1988] 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, the abstraction can be extended to multiple levels. This is distinct from the traditional nested transactions model presented in Section 4 in two main respects: (1) A multilevel transaction has a predefined number of levels, of which each two adjacent pairs defines a *layer* of the system, whereas nested transactions have no predefined notion of layers. (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 operations.

These two distinctions lead to a major difference between transaction management for nested transactions and for multilevel transactions. In nested trans-

actions, 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. Different concurrency control protocols (schedulers) are applied at different layers of the system. More specifically, layer-specific concurrency control protocols can be used. Each of these protocols must ensure serializability with respect to its layer. In addition, a protocol at one layer should not invalidate the protocols at higher levels. In other words, the protocols at all layers of the multilevel transaction should work together to produce a correct execution (schedule).

Unfortunately, not all combinations of concurrency control protocols lead to correct executions. To illustrate, assume we have a three-level necessary transaction and the protocol between the second and third levels is commutativity-based. This means that 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. 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 nonserializable order of the subtransactions at the second level.

The example above shows that serializability is too weak a correctness criterion to use for the "handshake" between the protocols of adjacent layers in a multilevel system. The correctness criteria must be extended to take into account the order of transactions at the adjacent layers. Beeri et al. [1986, 1989] introduced the notion of *order-preserving correctness* as the necessary property that layer-specific protocols must use to guarantee consistency. This notion was used earlier in a concurrency control model for multilevel transactions implemented in the DASDBS system [Weikum 1986; Weikum and Schek 1984]. A combined report on both of these efforts appears in Beeri et al. [1988].

The basic idea of order-preserving serializability is to extend the concept of commutativity. Commutativity states that order transformation of two operations belonging to the same transaction can be applied if and only if the two operations commute (i.e., the order of their execution with respect to each other is immaterial). This notion can be translated to multilevel systems by allowing the order of two adjacent 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. Pruning serial computations and thus reducing the number of levels in a multilevel transaction by one is termed *reduction*.

To illustrate, assume Mary is assigned the task of adding a new procedure $p10$ to module A and recompiling the module to make sure the addition of procedure $p10$ does 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 module A after adding or deleting a procedure. Recompiling a module is an abstraction for reading the source code of the module and updating the object containing the module (e.g., to update its timestamp and modify the object code). Consider the concurrent execution of T_{Mary} and T_{Bob} in Figure 8a. 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.
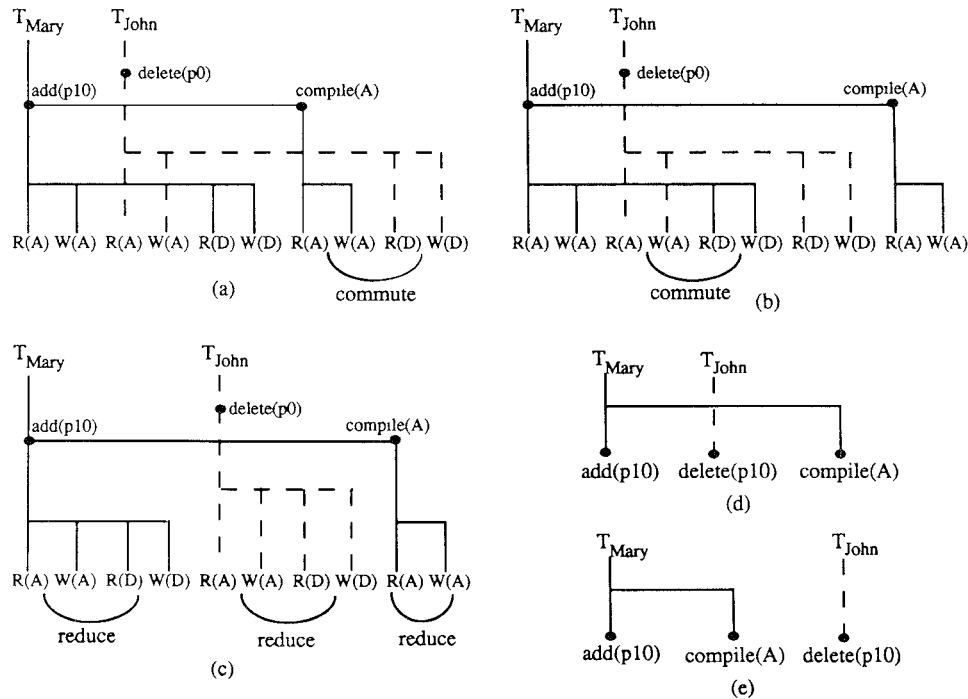
Figure 8. Order-preserving serializable schedule.

The results of successive commutations are shown in Figures 8b and 8c. The result of applying reduction is shown in Figure 8d, and the final result of applying commutation to the reduced tree, which is a serial schedule, is shown in Figure 8e.

Beeri et al. [1988] have shown that order preservation is only a sufficient condition to maintain consistency across layers in a multilevel system. They present a weaker necessary 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 9a, which shows a concurrent execution of three transactions initiated by Mary, Bob, and John. Compiling module A and compiling module B are nonconflicting 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 schedule is not order-preserving serializable, it is correct because it could be serialized, as shown in Figure 9b, by changing the order of the two compile operations. Since these are nonconflicting subtransactions, the change of order preserves correctness.

Martin [1987] 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. Operations at level i are specified in terms of operations at level $i - 1$. Thus, the execution of operations at level i results 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.

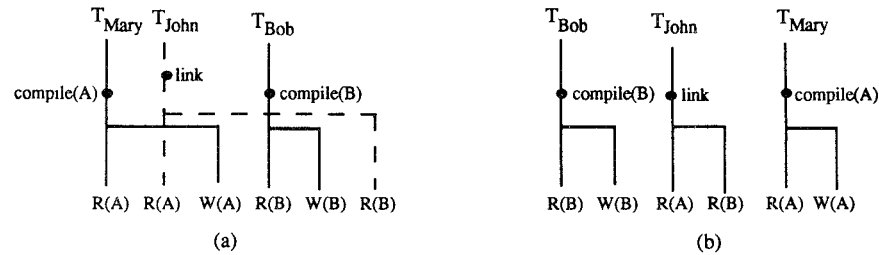Martin's model allows two kinds of schedules that are not serializable—

Figure 9. Conflict-based, order-preserving serializable schedule.

externally serializable schedules and *semantically verifiable* schedules. Externally serializable schedules allow only serializable access to top-level objects while allowing nonserializable access to subobjects. Subobjects may be 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 semantics of operations 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, in Figure 9 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. The scheduler would have then 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 code of two modules does not prohibit the linker from reading different versions of the two modules.

6.2 Relaxing Serializability

The approaches presented in Section 6.1 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 either data or application-specific operations. Relaxing serializability allows more concurrency and thus improves the performance of a system of concurrent transactions.

The semantics-based mechanisms can be divided into two main groups [Skarra and Zdonik 1989]. One group defines concurrency properties on abstract data types; the other defines concurrency properties on the transactions themselves. The first group of mechanisms [Herlihy and Weihl 1988; Weihl 1988] constrains interleaving of concurrent transactions by considering conflicts between operations defined on typed objects. Describing the details of this group of mechanisms requires an overview of abstract data types and object-oriented systems. Since these concepts are outside the scope of this paper, we have chosen to limit our discussion to the mechanisms in the second group, which use semantics of transactions rather than typed objects.

6.2.1 Semantics-Based Concurrency Control

Garcia-Molina [1983] observed that by using semantic information about transactions, a DBMS can replace the serializability constraint with the semantic consistency constraint. 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 that must 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 are only relatively consistent. For example, suppose Bob wants to get an idea about the progress of his programming team. He starts a transaction T_{Bob} that browses the modules and procedures of the project. Meanwhile, John and Mary have two in-progress transactions, T_{John} and T_{Mary}, respectively, that are modifying the modules and procedures of the project. Bob might be satisfied with information returned by a read-only transaction that does not take into consideration the updates being made by T_{John} and T_{Mary}. This would avoid delays that would result from having T_{Bob} wait for T_{John} and T_{Mary} to finish before reading the objects they updated.

A *semantically consistent schedule* is one that transforms the database from one *semantically* consistent state to another. It does so by guaranteeing that all sensitive transactions obtain a consistent view of the database. Each sensitive transaction must appear to be an atomic transaction with respect to all other transactions.

It is more difficult to build a general concurrency control mechanism that decides which schedules preserve semantic consistency than it is to build one that recognizes serializable schedules. Even if all the consistency constraints were given to the DBMS (which is not possible in the general case), there is no way for the concurrency control mechanism to determine a priori which schedules maintain semantic consistency. The DBMS must run the schedules and check the constraints on the resulting state of the database in order to determine if they maintain semantic consistency [Garcia-Molina 1983]. Doing that, however, would be equivalent to implementing an optimistic concurrency control scheme that suffers from the problem of rollback. To avoid rollback, the concurrency control mechanism must be provided with information about which transactions are compatible with each other.

Two transactions are said to be compatible if their operations can be interleaved at certain points without violating semantic consistency. Having the user provide this information is not feasible in the general case because it burdens the user with having to understand the details of applications. In some applications, however, 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 user still has to be provided with a framework for supplying information about the compatibility of transactions.

Garcia-Molina presented a framework that explicitly defines the semantics of database operations. He 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; the fourth piece of information consists of a procedural description. Transactions are categorized into types. The type of a transaction is determined by the nature of the operations it performs on objects. Each transaction type defines the steps that make up a transaction of that type. The steps are asumed to be atomic. 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 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. Compatibility sets define the allowable interleavings with respect to those steps. Thus, if transactions of type X are compatible with transactions of types Y and Z, 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 [1983] observed that it might be more appropriate to have different sets of interleavings with respect to different transaction types. More specifically, it would be useful if for every two transaction types X and Y, the DBMS is provided with information about the set of breakpoints at which the steps of a transaction of type X can be interleaved with the steps of a transaction of type Y. A breakpoint specifies a point between two operations within a transaction at which one or more operations of another transaction can be executed.

Lynch's 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. Each level groups transactions that have something in common in terms of access to data items. Level one groups all the transactions in the system, whereas 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 nonconflicting way. A set of breakpoints is then described for each level. The higher-order sets (for the higher levels) always include the lower order sets. This results in a total ordering of all sets of breakpoints. This means the breakpoints that specify interleavings at any level cannot be more restrictive than those that define interleavings at a higher level.

Let us illustrate this concept by continuing our example from the software development domain. Recall that 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). Suppose Mary starts two transactions (e.g., in two different windows), 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 for this example 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 transactions according to which modules they affect. Level 3 separates the transactions that modify module A (T_{Mary1} and T_{John1}) from those that modify module B (T_{Bob1}). Level 4 contains all the singleton transactions.

The sets of breakpoints for these levels

Level 1 T_{Mary1} T_{Mary2} T_{Bob1} T_{John1} T_{John2} More Allowable Interleavings

Level 2 T_{Mary1} T_{Bob1} T_{John1} T_{Mary2} T_{John2}

Level 3 T_{Mary1} T_{John1} T_{Bob1} T_{Mary2} T_{John2}

Level 4 T_{Mary1} T_{John1} T_{Bob1} T_{Mary2} T_{John2}
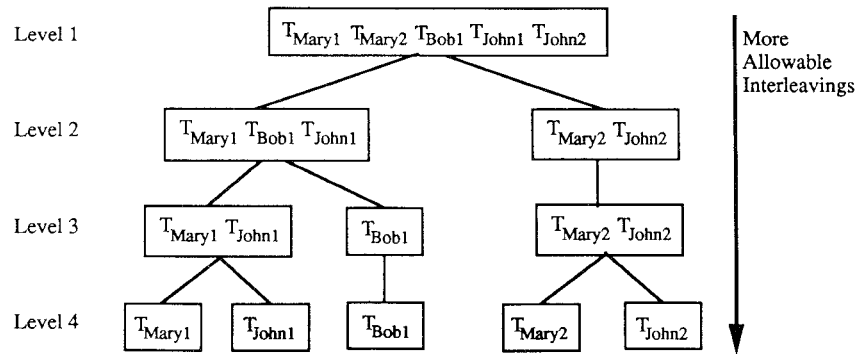
Figure 10. Multilevel transaction classes.

can be specified by describing the transaction segments between the breakpoints. For example, the top-level set might specify that no interleaving is allowed and the second-level set might specify that all modification transactions might interleave at some granularity and that cross-reference transactions might similarly interleave. The two types of transactions, however, cannot interleave their operations. This guarantees that cross-reference information does not change while a modification transaction is in progress.

The concurrency control mechanism can then use the sets of breakpoints to provide as much concurrency as permitted 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 degree 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.

Breakpoints is not the only way to provide semantic information about 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 1980, 1981]. By definition, full integrity of the design, in the sense of satisfying its specification, exists only when the design 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 [1983] proposed a transaction model that is based on the concept of *entity state*.

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 reads a set of entities and potentially writes into a set of entities. Unlike traditional transactions, however, entity state transactions are instances of transaction classes. Each class defines (1) the set of entities that instance transactions read, (2) the set of entities that instance transactions write, (3) the set of constraints that must be satisfied on the read and write entity sets prior to the invocation of a transaction, (4) the set of constraints that can be violated during the execution of an instance transaction, (5) the set of constraints that hold after the execution of the transaction is completed, and (6) the set of constraints that is violated after the transaction execution is com-

pleted. A simple example of a transaction class is the class of transactions that have all the entities in the database as its read set. 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 [1983] 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 details of intertransaction communication.

6.2.2 Sagas

Semantic atomicity, multilevel atomicity, and entity-based integrity constraints are theoretical concepts that are not immediately practical. For example, neither Garcia–Molina [1983] nor Lynch [1983] explain how a multilevel atomicity scheme might be implemented. 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 [1987] introduced the concept of *sagas*, which are LTs that can be broken up into a collection of subtransactions 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 subtransactions will be completed or they will be *compensated* (explained shortly). A saga thus satisfies

the definition of a transaction as a logical unit; a saga 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, nonserializable 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 subtransactions have changed. Thus, it would not be possible to restore the database to its state before the saga started without cascaded 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 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 could revert to an older version of the file and delete the updated version.

One shortcoming of sagas, however, is that they limit nesting to two levels. Most design applications require several levels of nesting to support high-level operations composed of a set of suboperations [Beeri et al. 1989]. 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 already described three related models in Section 6.1.3 [Beeri et al. 1988, 1989; Weikum and Schek 1984]. Two other nested transaction models [Kim et al. 1984; Walter 1984] are described in Section 7, since these two models address the issue of groups of users and coordinated changes. We now will describe a formal model of correctness without serializability that is based on multilevel transactions.

6.2.3 Conflict Predicate Correctness

Korth and Speegle [1988] have presented a formal model that allows mathematical characterization of correctness without serializability. 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. These features are similar to Kutay and Eastman's [1983] predicates described earlier. We describe Korth and Speegle's model at an intuitive level.

The database in Korth and Speegle's model is a collection of entities, each of which has multiple versions (i.e., multiple values). The versions are persistent and not transient like in the traditional multiversion 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. In other words, 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. Thus, a transaction 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 sets 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 ordering on these subtransactions, I_t is the input predicate on the set of all database states, and O_t is the output predicate. The input and output predicates define three sets of

data items related to a transaction: (1) the input set, (2) the update set, and (3) 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. The relation encodes dependencies between subtransactions based on their three data sets. This definition allows independent executions on different versions of database states.

Finally, Korth and Speegle 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 and 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* (CPC) class 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 multiversion 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, 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 for sagas.

Korth and Speegle [1990], recognizing that the practicality of their model was in question, applied the model to a realistic example from the field of computer-
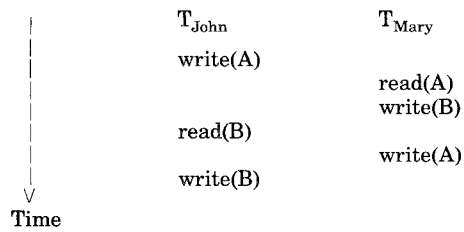
T_{John}	T_{Mary}
write(A)	
	read(A)
	write(B)
read(B)	
	write(A)
write(B)	

Time

Figure 11. Nonserializable but conflict-predicate-correct schedule.

aided software engineering (CASE). Rather than using the same example they presented, we use another example here. Consider the schedule shown in Figure 11 (which is adapted from [Korth and Speegle 1988]). This schedule is clearly not serializable and is not allowed by any of the traditional protocols. Suppose, however, that the database consistency constraint is a conjunct of the form $P1$ OR $P2$, where $P1$ is over A while $P2$ is over B. This occurs when A and B are not related to each other (i.e., the value of B does not depend on A and vice versa). 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 are acyclic, as shown in Figure 12. In other words, the schedule is correct because both T_{John} and T_{Mary} access A in a serializable manner and also access B in a serializable manner.

6.2.4 Dynamic Restructuring of Transactions

In many advanced database applications, such as design environments, operatio are interactive. The operations a use performs within a transaction might be (1) of uncertain duration, (2) of uncertain development (i.e., it cannot be predicted 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. Specifically,

Figure 12. Graphs built by CPC protocol.

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. [1988] introduced two new operations, *split-transaction* and *join-transaction*, which are used to reconfigure long transactions while in progress.

The basic idea is that all sets of database actions included in a set of concurrent transactions are performed in a schedule that is serializable when the actions are committed. The schedule, however, may include new transactions that result from splitting and joining the original transactions. Thus, the committed set of transactions may not correspond in a simple way to the originally initiated set. A 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 in order to release all of its resources so 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 he or she is done with some of the resources reserved by the transaction. These 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 (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 results from a split, the split-transactions can proceed in parallel and be controlled by different users. A 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.

To clarify this technique, suppose 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 finds that he needs to access module A. Being notified that T_{John} needs to access module A, Mary decides she can "give up" the module since she finished her changes to it. Therefore, she splits 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.

The split-transaction and join-transaction operations relax the traditional concept of serializability by allowing transactions to be dynamically restructured. Eventually, the restructuring
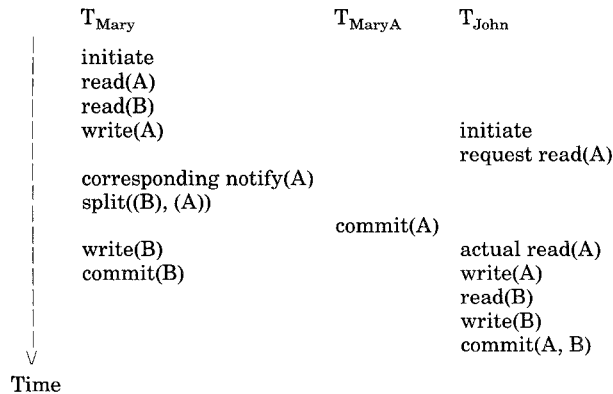
```
T_Mary                        T_MaryA      T_John

|   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)
V
Time
```

Figure 13. Example of split-transaction.

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 restructure their long transactions dynamically. This is most useful when opportunities for limited exchange of data among transactions arise while they are in progress. The split and join operations can be combined with the other techniques discussed in Section 8 to provide for the requirements of cooperation.

7. SUPPORTING COORDINATION AMONG MULTIPLE DEVELOPERS

When a small group of developers works on a large project, a need arises to coordinate the access of its members to the database in which project components are stored. Most of the time, the developers work independently on the parts of the project for which they are responsible, 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 developers, need to be enforced to guarantee that one developer does not duplicate or invalidate the work of other developers.

In this section we describe mechanisms that coordinate the efforts of members of a group of developers. It is important to

emphasize that all the mechanisms described in this section fall short of supporting synergistic cooperation in the sense of being able to pass incomplete but relatively stable data objects between developers in a nonserializable fashion. It is also important to note that unlike the mechanisms presented in Section 6, most of the models presented here were not developed as formal transaction models but rather as practical systems to support design projects, mostly software development efforts. The behavior of these systems, however, can be formulated in terms of transaction models, as we do in this section.

7.1 Version and Configuration Management

The simplest form of supporting coordination among members of a development team is to control the access to shared files so only one developer can modify any file at any one time. One approach that has been implemented by widely used version control tools like the Source Code Control System (SCCS) [Rochkind 1975] and the Revision Control System (RCS) [Tichy 1985] is the *checkout/ checkin* mechanism (also called reserve/ replace and reserve/deposit). Each data object is considered to be a collection of different versions. 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*, which means it cannot be modified. Instead, a new version can be created after explicitly reserving the object. The reservation 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 he or she 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 reserve another version, if that exists. Thus, two or more users can modify the same object only by working on two parallel versions, creating branches in the version history. Branching ensures write serializability by guaranteeing that only one writer 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 on the same version (rather than allowing them to occur then correcting them as in optimistic schemes). It is optimistic, however, in the sense that it allows multiple parallel versions of the same object to be created even if these versions are conflicting. The conflicts are resolved manually when the users merge their versions.

The basic checkout/checkin mechanism provides minimal coordination between multiple developers. It does not use semantic information about the objects or the operations performed on these objects. The model 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 conceptu-

ally to one aggregate object, creates new versions of all of them, makes sure they are consistent as a set, 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 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. The owner of the reservation can decide to let other programmers access that object.

The first problem results from not keeping track of which versions of objects are consistent with each other. For example, if each component (object) of a software system has multiple versions, it would be impossible to find out which versions of the objects actually participated in producing a particular executable version being tested. Further, a programmer cannot tell which versions of different objects are consistent with each other. There is a need to group sets of versions consistent with each other into *configurations*. This 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., Apollo Domain Software Engineering Environment (DSEE) [Leblang and Chase, Jr. 1987]). A recent survey by Katz [1990] gives a comprehensive overview of version and configuration management systems.

Supporting configurations 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 checkout/checkin 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.

7.1.1 Domain Relative Addressing

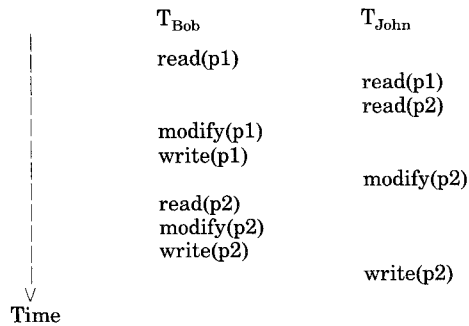Walpole et al. [1987, 1988a] addressed the problem of consistency in configura-

T_{Bob} T_{John}

read(p1)

           read(p1)
           read(p2)

modify(p1)
write(p1)

           modify(p2)

read(p2)
modify(p2)
write(p2)

           write(p2)

Time

Figure 14. Domain-relative addressing schedule.

Figure 15. Maintaining consitency using domain-relative addressing.

tion management systems and introduced a concurrency control notion called *domain-relative addressing* that supports versions of configuration objects. Domain relative addressing extends the notion of Reed's time-relative addressing (multiversion concurrency control) described in Section 4.3. 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.

To illustrate this technique, consider the two transactions T_{Bob} and T_{John} of Figure 14. By all the conventional concurrency control schemes, the schedule in the figure is disallowed. Under domain-relative addressing, the schedule in Figure 14 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 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 15. 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 module B modified by T_{John}), while A^2 (module A
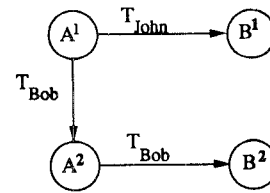
after T_{Bob} has modified it) is consistent with B^2 (the new version of module B that T_{Bob} has created). All of A^1, A^2, B^1, and B^2 become immutable versions. Domain-relative addressing is the concurrency control mechanism used in the Cosmos software development environment [Walpole et al. 1988b].

7.2 Pessimistic Coordination

Although domain-relative addressing solves the problem of configurations of objects, it does not address the second problem of the checkout/checkin model, which is concurrency control support for the checked-out objects. Two mechanisms that provide partial solutions to this problem are the *conversational transactions* mechanism provided as an extension to System R [Lorie and Plouffe 1983; William et al. 1981] and the *design transactions* mechanism [Katz and Weiss 1984]. Although the models differ in their details, they are similar as far as concurrency control is concerned. Thus, we refer to both models as the conversational transactions model. In this model, the database of a design project consists of a public database and several private databases. The public database is shared among all designers, whereas each private database is accessed only by a single designer. Each designer starts a long transaction in his or her private database that lasts for the duration of the design task.

When the long transaction needs to access an object in the public database, it 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. The

short transaction first sets a short-lived lock on the object, then 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 on the object for the duration of the long transaction. Before the short transaction commits, it copies the object to the specific private database and removes the short-lived lock. If the object has been checked out by another long transaction, the short transaction removes the short-lived lock, notifies the user that he or she cannot access the object, and aborts. The short-lived lock that was created by the short transaction on the public database prevents other short 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 checkin transactions on the public database at the end of the long transaction. A checkin transaction copies the object to the public database and deletes the old version of the object that was locked by the corresponding checkout 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 multiuser software development environment [Kaiser and Feiler 1987]. 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 in. Smile also maintains semantic information about the relations among objects, which enables it to reason about collections of objects rather than individual objects. It thus provides 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 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) 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.

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. The compilation verifies that the set of modules is self-consistent and did not introduce any errors that would prevent integrating it with the rest of the main database. If the compilation succeeds, 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 was reserved is successfully compiled then deposited.

Smile's model of consistency not only enforces self-consistency of the set of modules, 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 (e.g., to the number or types of parameters of a procedure) 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, John has to reserve module C (in addition to modules 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 module C, at which point T_{John} can reserve it, or to continue working on another task that does not require module C. From this example, it should be clear that by enforcing semantics-based consistency, Smile restricts cooperation even more than the conversational transactions because two users cannot simultaneously access objects semantically related to each other at the interface level.

Although the two-level database hierarchy of Smile and the conversational transactions mechanism provide better coordination support than the basic checkout/checkin 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 Section 6.

7.2.1 Multilevel Pessimistic Coordination

A more recent system, Infuse, supports a multilevel, rather than a two-level, hierarchy of experimental databases. Infuse relaxes application-specific consistency constraints by requiring only that modules in an experimental database be self-consistent before they are deposited to the parent database [Kaiser and Perry 1987]. More global consistency is enforced only when the modules reserved in top-level experimental databases are deposited to the main database.

Returning to our example, assume both Bob and Mary are involved in a task that requires modifying modules A and C; Figure 16 depicts the situation. They refers to create an experimental database
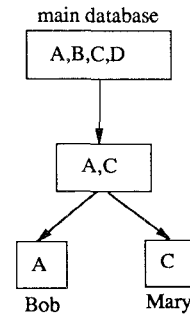

main database

Figure 16. Experimental databases in Infuse.

in which both modules A and C are reserved ($EDB_{A,C}$). Bob and Mary decide that Bob should modify module A and Mary should work on module C. Bob creates a child experimental database in which he reserves module A (EDB_A). Mary creates EDB_C in which she reserves module C. Bob decides 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. Recall 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 any data types or procedures 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 only 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 module C. Again no errors are detected at that level. When either Bob or Mary attempts to deposit the modules in $EDB_{A,C}$ to the main database, however, 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 must 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 reestablish consistency. But serializability is always maintained by requiring sibling EDBs to reserve disjoint sub sets of the resources locked by the parent EDB.

7.3 Optimistic Coordination

The coordination models presented in Section 7.2 are pessimistic in that they do not allow concurrent access to the same object in order to prevent any consistency violations that might occur. They are thus more restrictive in that sense than the configuration and version management schemes. It is often the case in design efforts, however, that two or more developers within the same team prefer to access different versions of the same object concurrently. Since these developers are typically familiar with each other's work, they can resolve any conflicts they introduce during their concurrent access by merging the different versions into a single consistent version. Rather than supporting multiple versions in a flat database, however, software development environments to provide a hierarchical structure like Infuse's.

7.3.1 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. 1989]. Like Cosmos (and unlike Infuse), NSE supports concurrent access to the same data objects. NSE combines the checkout/checkin model with an extension to the classical optimistic concurrency control policy, thus allowing limited cooperation among programmers. Unlike the checkout/checkin model and 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 can, however, 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 access private copies of the same object simultaneously. 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 other users in sibling environments have concurrently made on the same objects. 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.

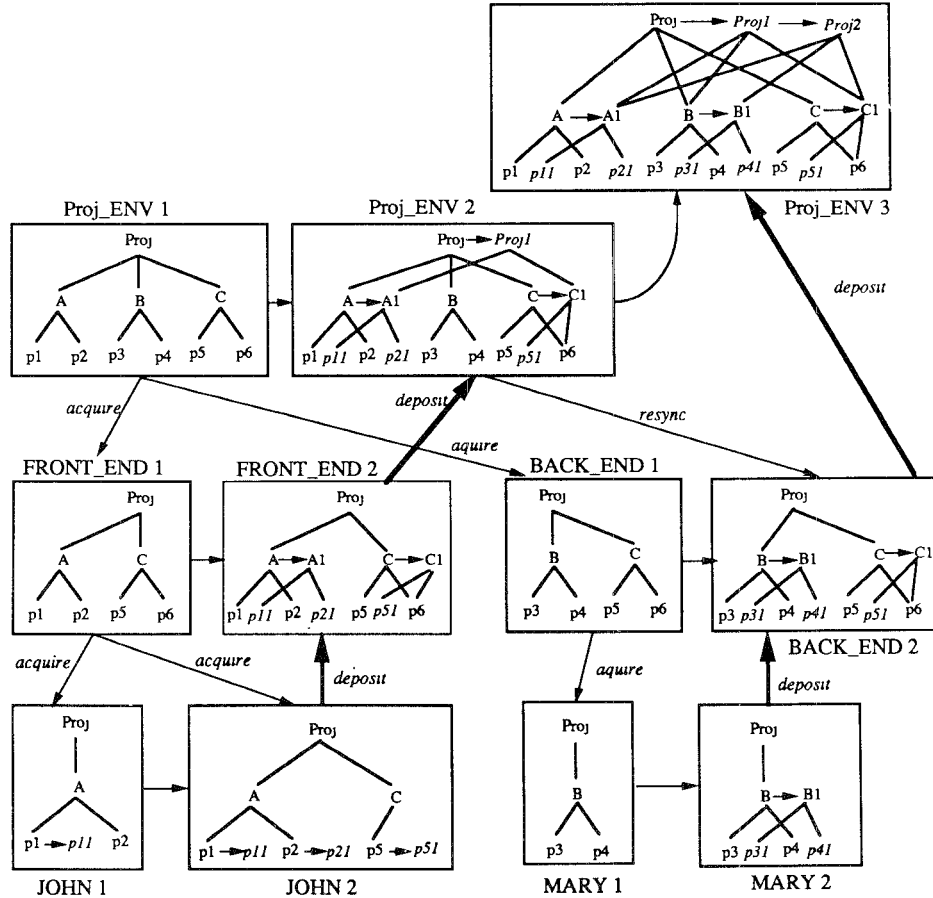To illustrate this mechanism, assume the modules of the project depicted in

Figure 17. Layered development in NSE.

Figure 1 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, and 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 coexist: 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; 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 acquires 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, modifies it, and deposits the changes to BACK_END. Mary can then test her code in BACK_END.

Suppose 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 another 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 occurs except that Mary discovers she needs to modify procedure *p5* in module 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.

7.3.2 Backout and Commit Spheres

Both Infuse and NSE implicitly use the concept of nested transactions; they also 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 the data items the parent transaction can access, and they commit their changes only to their parent transaction. A more general model is needed to support a higher level of coordination among transactions. Walter [1984] observed that there are three aspects that define the relationship between a parent transaction and a child subtransaction: the interface aspect, the dependency aspect, and the synchronization aspect.

The interface between a parent transaction and a child subtransaction can either be single-request, that is, the parent requests a query from the child and waits until the child returns the result or *con-*

versational, that is, 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 roll back 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 involved in a chain of conversations and requires 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, 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 in a different commit sphere than its parent commits, then the parent must either remember the child committed (e.g., by writing the committed values in its variables) or 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, thus the need for synchronization. If the child is called synchronously (i.e., the parent waits until the child terminates), it can safely access the data items locked by its parent.

Given these three aspects, Walter [1984] presented a nested transaction model in which each subtransaction has three attributes that must be defined when the transaction 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 NOBACK-OUT, 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 and its own commit sphere and it can access data items not locked by its parent.

Walter claims 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 et al.'s [1988] multilevel transaction model described in Section 6.1.3 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 we described 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 they are informed when

other programmers acquire or reconcile these objects. Infuse provides a notion of *workspaces* that cuts 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 under certain circumstances for the purpose of early detection of inconsistencies. None of the models described so far, however, supports all the requirements of synergistic cooperation among teams of developers.

8. SUPPORTING SYNERGISTIC COOPERATION

In Section 7 we addressed the issue of coordinating the access of a group of developers to the shared project database. Although this coordination is often all that is needed for small groups of developers, it is not sufficient when a large number of developers works on a large-scale design project [Perry and Kaiser 1991]. The developers are often subdivided into several groups, each responsible for a part of the design task. Members of each group usually cooperate to complete their part. 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 Section 7 address the coordination issue, 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 components.

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 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 nonserializable interaction.

Instead, the concept of database consistency preservation needs to be refined along the lines of Section 7 to allow nonserializable cooperative interaction. Such a refinement can be based on four observations [Bancilhon et al. 1985]: (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 where a group of designers, each working on a subtask, has a great need to share data among themselves and (4) in complex design projects, some subtasks are contracted to other design groups (subcontractors) that have limited access to the projects's main 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 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 Section 7), 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. The members of a group usually work on the same task (or at least related tasks) and thus need to cooperate among themselves much more than with members of 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 enforce serialization [Yeh et al. 1987]. 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 1987]. Notification alerts users about "interesting" events such as an attempt to lock an object that has already been locked in an exclusive mode.

Two policies that use notification in conjunction with nonexclusive locks and versions were implemented in the Gordion system: *immediate notification* and *delayed notification* [Yeh et al. 1987]. Immediate notification alerts the user of any conflict (attempt to access an object that has an exclusive lock on it or from 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.

These policies incorporate human beings as part of the conflict resolution algorithm. This, on the one hand, enhances concurrency in advanced applications, where many of the tasks are interactive. On the other hand, it can also degrade consistency because human beings might not really resolve conflicts, which could result in inconsistent data. Like the sagas model, this model burdens the user with knowledge about the 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, there is a need to define formally the kinds of interactions that can happen among members of the same team as opposed to interactions between teams. El Abbadi and Toueg [1989] defined the concept of a *group* as a set of transactions that, when executed, transforms the database from one consistent state to another. They presented the group paradigm to deal with consistency of replicated data in an unreliable distributed system. They hierarchically divide the problem of achieving serializability into two simpler ones: a local policy that ensures a total ordering of all transactions within a group and a global policy that ensures correct serialization of all groups.

Groups, like nested transactions, are an aggregation of a set of transactions. There are significant differences, however, 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 do not have 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 (e.g., an optimistic local policy and a 2PL global policy).

The group paradigm was introduced to model intersite consistency in a distributed database system. It can also 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. The local policies and the global policy have to be compatible in the sense that they do not contradict each other. Toueg and El Abbadi do not sketch the compatibility requirements between global and local policies.

Dowson and Nejmeh [1989] applied the group concept to model teams of programmers. They introduced the notion of *visibility domains*, which model groups of programmers executing nested transactions on immutable objects. A visibility domain is a set of users that 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; the other two were designed for SDEs. The notion of cooperation in SDEs is similar to that in CAD. Differences, however, 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.

In both cases, designers working on the same subtask might need unconstrained cooperation, whereas two designers working on different tasks of the same project might need more constrained cooperation. CAD 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 providing such support is to divide users (designers or programmers) into groups. Each group is then provided with a range of lock modes that allows various levels of isolation and cooperation among multiple users in the same group and between 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. All four mechanisms avoid using blocking to synchronize transactions, thus eliminating the problem of deadlock.

8.2.1 Group-Oriented CAD Transactions

One approach, called the *group-oriented model*, extends the conversational transactions model described in Section 7 [Klahold et al. 1985]. Unlike the conversational transactions model, the group-oriented model does not use long-lived locks on objects in the public database. The conversational transactions model sets long-lived locks on objects that are checked out from the public database until they are checked back in to the public database.

The group-oriented model categorizes transactions into *group transactions* (GT) and *user transactions* (UT). Any UT is a subtransaction of a GT. The model also 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 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. 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 are serializable unless users intervene to make them cooperate in a nonserializable schedule. The basic mechanism provided for relaxing serializability is a version concept that allows parallel development (branching) and notification. Versions are derived, deleted and modified explicitly by a designer only after being locked in any one of a range of lock modes.

The model supports five lock modes on a version of an object: (1) read only, which makes a version available only for reading; (2) read–derive, which allows multiple users either to read the same version or derive a new version from it; (3) shared derivation, which allows the owner to both read the version and derive a new version from it, while allowing parallel reads of the same version 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-phased, consisting of an acquire phase and a release phase. Locks can only be strengthened (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 later when the object he or she 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 be about to be changed. This operation provides the designer (more often a manager of a design effort) with 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, et al. [1985], envisions a design workspace to consist of a global database that contains a public database for each project and private databases for active designers' transactions. 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 invokes a long transaction to complete a well-defined subtask for which he is responsible.

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 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 or her 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 Section 7.2.1.

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. Thus, the integrity constraints defined at the top level of a cooperating transaction imply all the constraints defined at lower levels. At the lowest level 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 et al. [1985] 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 execu-
tions. Thus, if the set of rules is strict
enough, they would allow only serializ-
able executions. The set of rules can,
however, allow nonserializable, 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 mecha-
nisms with a range of lock modes of vary-
ing 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 frame-
work for cooperating transactions [Zdonik
1989; Skarra and Zdonik 1989]. A trans-
action group (TG) is defined as a process
that controls the access of a set of cooper-
ating 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 accord-
ing to an *input protocol* that defines some
semantic correctness criteria appropriate
for the application. The criteria are spec-
ified by semantic patterns and enforced
by a recognizer and a conflict detector.
The recognizer ensures that a lock re-
quest from a member transaction
matches an element in the set of locks
that the group may grant its members.
The conflict detector ensures that a re-
quest to lock an object in a certain mode
does not conflict with the locks already
held on the object.

If a transaction group member re-
quests an object that is not currently
locked by the group, the group has to
request a lock on the object from its par-
ent. The input protocol of the parent
group, which controls access to objects,
might be different from that of the child
group. Therefore, the child group might
have to transform its requested lock into
a different lock mode accepted by the
parent's input protocol. The transforma-
tion is carried out by an *output protocol*,
which consults a lock translation table to
determine how to transform a lock re-
quest 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 transac-
tion writes, writing while other transac-
tions read, and multiple writers of the
same object. 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 ex-
ample. Mary and John are assigned the
task of updating modules A and B, which
are strongly related (i.e., procedures in
the modules call each other, and type
dependencies exist between the two mod-
ules), while Bob is assigned responsibil-
ity for updating the documentation of the
project. 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 trans-
action 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 mem-
bers. 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 partial results of their updates to the
modules) while isolation is maintained
within TG1 (to prevent T_{Bob} from access-
ing the partial results of the transactions
in TG2). This arrangement is depicted in
Figure 18.

8.2.4 Participant Transactions

The transaction groups mechanism de-
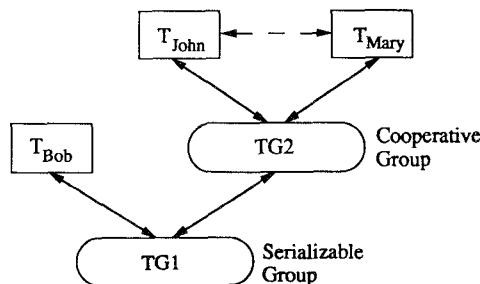fines groups in terms of their access to

Figure 18. Transaction groups.

database objects in the context of a nested transaction system. Another approach is to define a group of transactions as *participants* in a specific *domain*[1] [Kaiser 1990]. Participant transactions in a domain need not appear to have been performed in some serial order with respect to each other. The set of transactions that is not a participant in a domain is considered an *observer* of the domain. The set of observer transactions of a domain must be serialized with respect to the domain. A particular transaction may be a participant in some domains and an observer of others whose transactions access the same objects.

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

The domain in which a transaction participates would typically be the set of transactions associated with the members of a cooperating group of users working toward a common goal. Unlike

---

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

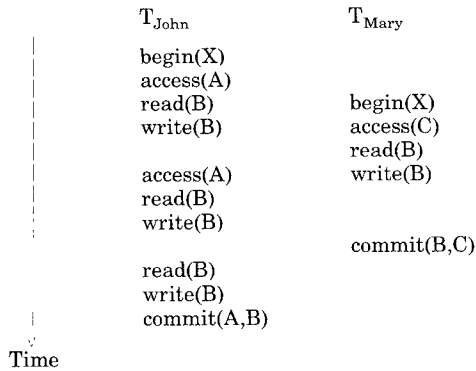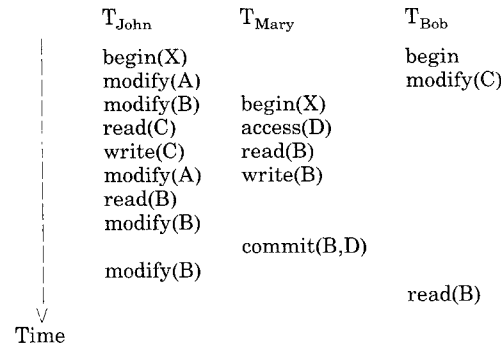transaction groups, however, there is no implication that all the transactions in the domain commit together or even that all of them commit (some may abort). Thus, it is misleading to think of the domain as a top-level transaction, with each user's transaction as a subtransaction, although in practice this is likely to be a frequent case. The transaction groups mechanism described above is thus a special case of participant transactions.

Each transaction is associated with zero or one particular domain at the time it is initiated. A transaction that does not participate in any domain is the same as a classical (but interactive) transaction. 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 share partial results with other transactions in the same domain nonserializably, but it must be serializable with respect to all transactions not in the domain.

To illustrate, say a domain X is defined to respond to a particular modification request, and programmers Mary and John start transactions T_{Mary} and T_{John} that participate in X. Assume 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 T_{John} made to module B that are written but are not yet committed by T_{John}, modifies parts of module B, then commits. T_{John} continues to modify modules A and B after T_{Mary} has committed. Since T_{Mary} and T_{John} participate in the same domain X, the schedule is legal according to the participant transactions mechanism.

Now suppose Bob starts a transaction T_{Bob} that is an observer of domain X. Assume the 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 module B, which has been modified and committed by T_{Mary}. This would be illegal even though

```
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)

Time
```

Figure 19. Participation schedule.

```
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. Participation conflict.

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 not 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 module B before any transactions outside the domain accessed module B. Thus, the serializability of transactions within a participation domain needs be enforced only with respect to what is actually observed by the users who are not participants in the domain.

9. SUMMARY

This paper investigates concurrency control issues for a class of advanced database applications involving computer-supported cooperative work. We concentrate 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: Many are theoretical frameworks rather than practical schemes, and 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 1 summarizes the discussions in Sections 6 to 8 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.

There are four other concerns that extended transactions models for advanced applications should address: (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 is 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 of type *code*. Another tool, for document preparation, accesses all 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

Table 1. Advanced Database Systems and Their Concurrency Control Schemes

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
Split-transaction, join-transaction	N/A	Yes	Yes	Limited
Checkout/checkin	RCS	No	No	Limited
Domain-relative addressing	Cosmos	Yes	No	Limited
Conversational transactions	System R	Limited	Limited	No
Multilevel coordination	Infuse	Yes	Yes	Limited
Copy/modify/merge	NSE	Yes	Yes	Limited
Backout and commit spheres	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	Limited	Limited	Yes
Participant transactions	N/A	Yes	Limited	Yes

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 upon which they operate, where consistency is defined according to the needs of the application.

A problem that remains unsolved is the lack of performance metrics by which to evaluate the proposed policies and mechanisms in terms of the efficiencies of both implementation and use. We have encountered only one empirical study [Yeh et al. 1987] that investigates 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 be that some of the schemes that appear adequate theoretically will turn out to be inefficient or unproductive for the purposes of a particular application. But is is not clear how to define appropriate measures.

Another problem is that most of the notification schemes are limited to attaching the notification mechanism to the locking primitives and 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 (i.e., what the database community calls *triggers* [Stonebraker et al. 1988]). 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. Some queries may, however, require a consistent state of the database in the sense that no other activities may be concurrently writing to 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 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 1990].

Although all of the extended transaction models presented in this 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

REFERENCES

ADAMS, E. W., HONDA, M., AND MILLER, T. C. 1989. Object managment in a CASE environment. In *Proceedings of the 11th International Conference on Software Engineering* (May) IEEE Computer Society Press, pp. 154–163.

BANCILHON, F., KIM, W., AND KORTH, H. 1985. A model of CAD transactions. In *Proceedings of the 11th International Conference on Very Large Databases*, (August), Morgan Kaufmann, pp. 25–33.

BEERI, C., BERNSTEIN, P. A , AND GOODMAN, N. 1986 A model for concurrency in nested transaction systems. Tech Rep. TR-86-03, The Wang Institute for Graduate Studies, Tyngaboro, Mass.

BEERI, C , BERNSTEIN, P. A., AND GOODMAN, N. 1989. A model for concurrency in nested transaction systems. J/2, *ACM 36*, 230–269.

BEERI, C., SCHEK, H.-J., AND WEIKUM, G. 1988. Multilevel transaction management: Theoretical art or practical need? *Adv. Database Tech* (Mar), pp. 134–154 Published as Lecture Notes in Computer Science #303, G. Goos and J. Hart Manis, Eds

BERNSTEIN, P. A. 1987. Database systems support for software engineering: An extended ab-

stract. In *Proceedings of the 9th International Conference on Software Engineering* (March), IEEE Computer Society Press, pp. 166–178.

BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June), 185–221.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass.

BJORK, L. A. 1973. Recovery scenario for a DB/DC system. In *Proceedings of the 28th ACM National Conference* (Aug.), ACM Press, pp. 142–146.

CHRYSANTHIS, P. K., AND RAMAMRITHAM, K. 1990. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data* (May), ACM Press, pp. 194–203.

DAVIES, C. T. 1973. Recovery semantics for a DB/DC system. In *Proceedings of the 28th ACM National Conference* (Aug.), ACM Press, pp. 136–141.

———. 1978. Data processing spheres of control. *IBM Syst. J. 17*, 2, 179–198.

DITTRICH, K., GOTTHARD, W., AND LOCKEMANN, P. 1987. DAMOKLES: The database system for the UNIBASE software engineering environment. *IEEE Bullet. Database Eng. 10*, 1 (March), 37–47.

DOWSON, M., AND NEJMEH, B. 1989. Nested transactions and visibility domains. In *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases* (Feb.), ACM Press, pp. 36–38. Position paper.

EASTMAN, C. 1980. System facilities for CAD databases. In *Proceedings of the 17th ACM Design Automation Conference* (June), ACM Press, pp. 50–56.

———. 1981. Database facilities for engineering design. In *Proceedings of the IEEE* (Oct.), IEEE Computer Society Press, pp. 1249–1263.

EGE, A., AND ELLIS, C. A. 1987. Design and implementation of Gordion, an object-based management system. In *Proceedings of the 3rd International Conference on Data Engineering* (Feb.), IEEE Computer Society Press, pp 226–234.

EL ABBADI, A. AND TOUEG, S. 1989. The group paradigm for concurrency control protocols. *IEEE Trans. Knowledge Data Eng. 1*, 3 (Sept.), 376–386.

ESWARAN, K., GRAY, J., LORIE, R., AND TRAIGER, I. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–632.

FELDMAN, S. I. 1979. Make: A program for maintaining computer programs. *Softw. Pract. Exper. 9*, 4 (Apr.), 255–265.

FERNANDEZ, M. F., AND ZDONIK, S. B. 1989. Trans-

action groups: A model for controlling cooperative work. In *Proceedings of the 3rd International Workshop on Persistent Object Systems: Their Design, Implementation, and Use* (Jan.), Morgan-Kaufman pp. 128–138.

GARCIA-MOLINA, H. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8*, 2 (June), 186–213.

GARCIA-MOLINA, H., AND SALEM, K. 1987. SAGAS. In *Proceedings of the ACM SIGMOD 1987 Annual Conference* (May), ACM Press, pp. 249–259.

GARZA, J., AND KIM, W. 1988. Transaction management in an object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (June), ACM Press, pp. 37–45.

GRAY, J. 1978. Notes on database operating systems. IBM Res. Rep. RJ2188, IBM Research Laboratory, San Jose, Calif.

GRAY, J., LORIE, R., AND PUTZOLU, G. 1975. Granularity of locks and degrees of consistency in a shared database. IBM Res. Rep. RJ1654, IBM Research Laboratory, San Jose, Calif.

HERLIHY, M. P., AND WEIHL, W. E. 1988. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems* (Mar.), ACM Press, pp. 201–210.

KAISER, G. E. 1990. A flexible transaction model for software engineering. In *Proceedings of the 6th International Conference on Data Engineering* (Feb.), IEEE Computer Society Press, pp. 560–567.

KAISER, G. E., AND FEILER, P. H. 1987. Intelligent assistance without artificial intelligence. In *Proceedings of the 32nd IEEE Computer Society International Conference* (Feb.), IEEE Computer Society Press, pp. 236–241.

KAISER, G. E., AND PERRY, D. E. 1987. Workspaces and experimental databases: Automated support for software maintenance and evolution. In *Proceedings of the Conference on Software Maintenance* (Sept.), IEEE Computer Society Press, pp. 108–114.

KATZ, R. H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv. 22*, 4 (Dec.), 375–408.

KATZ, R. H., AND WEISS, S. 1984. Design transaction management. In *Proceedings of the ACM IEEE 21st Design Automation Conference* (June), IEEE Computer Society Press, pp. 692–693.

KIM, W., LORIE, R. A., McNABB, D., AND PLOUFFE, W. 1984. A transaction mechanism for engineering databases. In *Proceedings of the 10th International Conference on Very Large Databases* (Aug.), Morgan Kaufmann, pp. 355–362.

KIM, W., BALLOU, N., CHOU, H., AND GARZA, J. 1988. Integrating an object-oriented pro-

gramming system with a database system. In *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications* (Sept.), ACM Press, pp. 142–152.

KLAHOLD, P., SCHLAGETER, G., UNLAND, R., AND WILKES, W. 1985. A transaction model supporting complex applications in integrated information systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (May), ACM Press, pp. 388–401.

KOHLER, W. 1981 A survey of techniques for synchronization and recovery in decentralized computer systems *ACM Comput. Surv. 13*, 2 (June), 149–183.

KORTH, H., AND SILBERSCHATZ, A. 1986. *Database System Concepts*. McGraw–Hill, New York.

———. 1990. Long-duration transactions in software design projects. In *Proceedings of the 6th International Conference on Data Engineering* (Feb.), IEEE Computer Society Press, pp 568–574.

KORTH, H., AND SPEEGLE, G. 1988. Formal model of correctness without serializability. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (June), ACM Press, pp. 379–386.

KUNG, H., AND ROBINSON, J. 1981. On optimistic methods for concurrency control *ACM Trans. Database Syst. 6*, 2 (June), 213–226.

KUTAY, A., AND EASTMAN, C. 1983. Transaction management in engineering databases. In *Proceedings of the Annual Meeting of Database Week: Engineering Design Applications* (May), IEEE Computer Society Press, pp. 73–80.

LEBLANG, D. B., AND CHASE, R P., JR. 1987. Parallel software configuration management in a network environment *IEEE Softw 4*, 6 (Nov.), 28–35.

LORIE, R , AND PLOUFFE, W 1983. Complex objects and their use in design transactions. In *Proceedings of the Annual Meeting of Database Week; Engineering Design Applications* (May), IEEE Computer Society Press, pp 115–121.

LYNCH, N. A. 1983 Multilevel atomicity: A new correctness criterion for database concurrency control. *ACM Trans. Database Syst. 8*, 4 (Dec.), 484–502

MARTIN, B. 1987 Modeling concurrent activities with nested objects. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Sept.), IEEE Computer Society Press, pp 432–439.

MOSS, J. E. B. 1985. *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge, Mass.

NESTOR, J. R. 1986. Toward a persistent object base In *Advanced Programming Environments*, Conradi, R., Didriksen, T. M., and Wanvik, D H , eds. Springer–Verlag, Berlin, pp. 372–394.

PAPADIMITRIOU, C 1986. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Md.

PERRY, D., AND KAISER, G. 1991. Models of software development environments *IEEE Trans. Softw Eng. 17*, 3 (Mar.)

PRADEL, U , SCHLAGETER, G , AND UNLAND, R. 1986. Redesign of optimistic methods: Improving performance and availability In *Proceedings of the 2nd International Conference on Data Engineering* (Feb.), IEEE Computer Society Press, pp. 466–473.

PU, C., KAISER, G., AND HUTCHINSON, N. 1988. Split transactions for open-ended activities In *Proceedings of the 14th International Conference on Very Large Databases* (Aug.), Morgan Kaufmann, pp. 26–37.

REED, R. 1978 Naming and synchronization in a decentralized computer system Ph.D. dissertation, MIT Laboratory of Computer Science, MIT Tech. Rep. 205.

ROCHKIND, M. J 1975. The source code control system *IEEE Trans. Softw. Eng. SE-1*, 364–370

ROSENKRANTZ, D., STEARNS, R., AND LEWIS, P. 1978 System-level concurrency control for distributed database systems *ACM Trans. Database Syst 3*, 2 (June), 178–198

ROWE, L. A , AND WENSEL, S. 1989. *1989 ACM SIGMOD Workshop on Software CAD Databases*, Napa, Calif

SALEM, K., GARCIA–MOLINA, H., AND ALONSO, R. 1987. Altruistic locking: A strategy for coping with long-lived transactions In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems* (Sept), Amdahl Corporation, IBM Corporation, pp. 19.1–19 24.2RSilberschatz, A., and Kedem, Z. Consistency in hierarchical database systems. *J. ACM 27*, 1 (Jan), 72–80

SKARRA, A. H , AND ZDONIK, S. B. 1989. Concurrency control and object-oriented databases In Kim, W., and Lochovsky, F. H., eds , *Object-Oriented Concepts, Databases, and Applications*, ACM Press, New York, pp 395–421

STONEBRAKER, M., KATZ, R , PATTERSON, D., AND OUSTERHOUT, J. 1988. The design of XPRS. In *Proceedings of the 14th International Conference on Very Large Databases* (Aug.), Morgan Kaufmann, pp 318–330

TICHY, W. F. 1985. RCS: A system for version control. *Softw. Pract. Exper. 15*, 7 (July), 637–654

WALPOLE, J., BLAIR, G., HUTCHISON, D., AND NICOL, J. 1987. Transaction mechanisms for distributed programming environments. *Softw. Eng. J. 2*, 5 (Sept.), 169–177

WALPOLE, J., BLAIR, G., MALIK, J., AND NICOL, J. 1988a A unifying model for consistent distributed software development environments. In *Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on*

Practical Software Development Environments (Nov.), ACM Press, pp. 183–190.

——. 1988b. Maintaining consistency in distributed software engineering environments. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (June), IEEE Computer Society Press, pp. 418–425.

WALTER, B. 1984. Nested transactions with multiple commit points: An approach to the structuring of advanced database applications. In *Proceedings of the 10th International Conference on Very Large Databases* (Aug.), Morgan Kaufmann, pp. 161–171.

WEIHL, W. 1988. Commutativity-based concurrency control for abstract data types (Preliminary Report). In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences* (Jan.), IEEE Computer Society Press, pp. 205–214.

WEIKUM, G. 1986. A theoretical foundation of multilevel concurrency control. In *Proceedings*

of the 5th ACM Symposium on Principles of Database Systems (Mar.), ACM Press, pp. 31–42.

WEIKUM, G., AND SCHEK, H.-J. 1984. Architectural issues of transaction management in multilevel systems. In *Proceedings of the 10th International Conference on Very Large Databases* (Aug.), Morgan Kaufmann.

WILLIAMS, R., DANIELS, D., HASS, L., LAPIS, B., LINDSAY, B., NG, P., OBERMARCLE, R., SELINGER, P., WALKER, A., WILMS, P., AND YOST, R. 1981. R:* An overview of the architecture In *Improving Database Usability and Responsiveness*, Scheuermann, P., Ed. Academic Press, NY, pp. 1–27.

YANNAKAKIS, M. 1982. Issues of correctness in database concurrency control by locking. *J. ACM 29*, 3 (July), 718–740

YEH, S., ELLIS, C., EGE, A., AND KORTH, H. 1987. Performance analysis of two concurrency control schemas for design environments. Tech. Rep. STP-036-87, MCC, Austin, Tex.