A Flexible Transaction Model for Software Engineering

Gail E. Kaiser*

Columbia University
Department of Computer Science
New York, NY 10027
212-854-3856
kaiser@cs.columbia.edu

CUCS-445-89

10 July 1989

Abstract

It is generally recognized that the classical transaction model, providing atomicity and serializability, is too strong for certain application areas since it unnecessarily restricts concurrency. We are concerned with supporting cooperative work in multi-user design environments, particularly teams of programmers cooperating to develop and maintain software systems. We present an extended transaction model that meets the special requirements of software engineering projects, describe possible implementation techniques, and discuss a number of issues regarding the incorporation of such a model into multi-user software development environments.

*Supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

keywords: concurrency control, extended transaction models, software development environments

1. Introduction

The focus of this paper is how to coordinate activities among groups of individuals, who are cooperating to achieve common goals in the context of design environments based on database management systems; we are specifically concerned with teams of programmers using a shared software development environment. The primary innovations of this research are (1) a flexible transaction model suitable for the software development process and (2) the first steps toward a technology for applying the model to building multi-user software development environments. We use the term "transaction" loosely to refer to any protocol that supports failure recovery, concurrency control, data consistency, and user-initiated rollback with respect to (possibly nested) units of work appropriate to the application. Previously proposed protocols are informal, address only a subset of these capabilities, and/or support units of work that are too small [Rowe 89].

Our goal is to demonstrate that the transaction, thus defined, is the proper mechanism for activity coordination among and between teams of programmers, and therefore should form part of the foundation of multi-user software development environments. This goal has been generally recognized in the database research community [Neuhold 88], but has not previously been achieved. Our approach to this problem is to formalize a new, *flexible* transaction model consisting of the following features:

1. **Decomposition**, using *nested transactions*, a previously existing notion for permitting simultaneous serialized activities within a transaction [Moss 85].
2. **Fault tolerance**, using *savepoints*, a previously existing mechanism for preventing crashes from causing complete undo of the current subtransaction [Gray 81].
3. **Persistent versions**, a previously existing mechanism for exploring alternatives, maintaining history, and increasing concurrency (e.g., [Walpole 88a]).
4. **Activity Interaction**, using *commit-serializability*, supported by two new transaction processing operations we have developed: split-transaction and join-transaction [Pu 88]. Commit-serializability denotes that all sets of database operations are in fact serializable when committed, even though these transactions may not correspond in a simple way to those transactions that were begun.

   For example, in response to a modification request, programmer Alice starts her own transaction, in which she reads modules M and N, updates, compiles, and links N together with old object code for M. Another programmer Bob requests to read N. Alice is done with N but not M, so she *splits* her transaction into two, one of which commits the update to N. Bob now can read N; he recompiles and uses N for testing his own changes. Bob commits or aborts his transaction, including any new changes to N, and Alice independently commits or aborts her update to M.
5. **Programmer Interaction**, using *participation domains* (aka participant

transactions), a new formalism for relaxing the classical intent of serializability that a set of transactions appear to have been performed in some serial order with respect to every user. Certain users are designated as participants in a specific set of transactions (the domain), meaning these transactions need not appear to have been performed in some serial order with respect to these participants, who may view uncommitted updates. All other users must observe a serial order.

Again considering the example, programmer Alice starts her transaction, reads modules M and N, updates, compiles, and links N together with old object code for M. Programmer Bob is a participant in the same set of transactions as Alice, so he reads N; this does not require N to be committed, as with commit-serializability above. Bob recompiles and uses N for testing his own changes. Bob commits or aborts his work and Alice commits or aborts her transaction. If Alice aborts and thus rolls back the update to N, Bob's transaction will be notified and Bob may request for it to be rolled back.

Although dimensions 1, 2 and 3 have previously appeared individually, 4 and 5 are novel approaches of this work, as is the integration of these concepts. In this paper, we concentrate on the flexible transaction model, particularly concurrency control issues. We have previously introduced the split-transaction and join-transaction operations, which form the basis for commit-serializability, but participation domains appears for the first time in this paper. We plan to implement the flexible transaction model as part of the kernel of a generic software development environment architecture, to be able to produce a range of multi-user environments. Our preliminary design for application of commit-serializability alone to our existing Marvel software development environment is presented elsewhere [Kaiser 89], while incorporation of the full flexible transaction model into a multi-user software development environment is still in its early stages.

We start by sketching the notion of user-controlled transactions, necessary for cooperative work in for many application areas apart from software engineering, but perhaps unfamiliar to readers versed in traditional transaction applications such as banking and airline reservations. We then outline previous work in applying transactions to software engineering. We briefly discuss commit-serializability, and then present participation domains. Finally, we consider a range of implementation techniques, including the combination of commit-serializability and/or persistent versions with participation domains, resulting in the concurrency control component of our new flexible transaction model. We do not address fault tolerance in this paper.

2. Extending Transactions to Software Engineering

We consider the main contribution of the classical transaction model to be the principle of *conservation of consistency* in the database. A transaction that meets its specification will take the database from one consistent state into another consistent state. The vast majority of existing transaction processing systems support only *programmed* transactions, where a program is written in advance (or compiled from a single end-user query) to undertake one or more transactions.

Our extension to the classical transaction model is motivated by *user-controlled* transactions (e.g., [Skarra 86, Pausch 88]). Our goal is to conserve the consistency of the database extended for software development, manipulated by both programs (software development tools such as compilers and debuggers) and programmers. A user-controlled transaction starts when a human user gives a begin-transaction command to the user interface of the transaction manager (part of a software development or other multi-user design environment). The user may then carry out any number of activities that read and update objects, including initiating nested subtransactions, which may be either user-controlled or programmed.

User-controlled transactions are open-ended, i.e., users do not pre-declare all the objects to be manipulated at the start of transactions, but acquire the objects as they are needed. In our application, the users are allowed to exchange information with other user-controlled transactions. Our extended transaction manager monitors these user interactions; depending on their impact on object consistency, the extended transaction manager may prevent an exchange, alert the users involved about the conflict, request the users to resolve the conflict immediately, or postpone the conflict in the hope of resolving it later. A user-controlled transaction ends when the user gives either the commit-transaction command, in which case the updates are committed, or the abort-transaction command, in which case the updates are rolled back. In both cases, the extended transaction manager checks the object consistency and may request the user to resolve any remaining conflicts.

The above paragraph is an oversimplification in that we do *not* intend for the human users to be directly involved in resolving most of the conflicts detected by the transaction manager. This added burden of user-controlled transactions makes it mandatory to embed our flexible transaction model in an advanced software development environment kernel such as Marvel [Kaiser 88a, Kaiser 88b] or Arcadia [Osterweil 87, Taylor 88]. We envision that such an

environment will be able model interactions with the transaction manager as well as interactions with tools, and thus the environment will perform most conflict resolution on behalf of the users.

Under no circumstances, however, should the transaction manager or the environment automatically roll back a user-controlled transaction, throwing away possibly many days of work, to resolve either consistency conflicts or deadlocks. Under the direction of the environment, or in some cases a human user, the extended transaction manager may partially roll back a user-controlled transaction to a savepoint. The notion of automatically restarting is restricted to programmed transactions. For the same reason, recovery from crashes will resume from the most recent savepoint.

3. Previous Work

Transactions were invented for databases [Eswaran 76], and have recently been added to operating systems [Spector 88]. Many properties of a transaction (fault tolerance, controlled concurrent access to data, commitment of a consistent set of changes, explicitly requested abort, and nested activities) are essential for multi-user software development environments, as well as other cooperative work such as CAD/CAM [Bancilhon 85]. An archetypical example would be enclosing within a transaction all activities of a programming team responding to a modification request for a deployed software product. These activities — different programmers browsing and editing overlapping sets of source files, compiling and linking, executing test cases and generating traces, etc. — could take weeks and affect substantial portions of the system. Other programmers not on the modification request team should not see the system in an inconsistent state, but must be able to continue their own programming work.

Classical transactions are based on two properties: *atomicity* for crash recovery and *serializability* for concurrency control. Atomicity requires that either the entire transaction appears to have completed, or that the entire transaction appears to have never started. Serializability requires that all committed transactions appear to have executed in some serial order. These properties cause problems if the model is naively applied to software development (or other cooperative work). If a transaction aborts due to a failure, *all* changes are undone, perhaps throwing away hours of work. Concurrency control techniques that ensure isolation and the appearance of serial order are too restrictive: a programmer would be prevented from editing some file simply because another programmer had previously *read* the file but has not yet

finished his programming transaction; programmers would not be able to release certain resources to programmers cooperating on the same subsystem while continuing to use other resources that are part of the same activity.

Existing software engineering tools provide some of the needed facilities, such as serialized access to individual files and creation of parallel versions (e.g., RCS [Tichy 85]), system build (e.g., Make [Feldman 79]), and checkpointing and undo/redo [Archer 84]. The crippling problem is that these mechanisms operate only on individual files (or the collection of individual files required for a system build), rather than on the complete set of resources updated during the software development activity. A few environments, such as Smile [Kaiser 87a] developed by the Gandalf project at Carnegie Mellon University, Imperial Software Technology's IStar [Dowson 87], and our own Infuse [Kaiser 87b], do publish sets of resources but use ad hoc methods rather than a formal model, and thus their mechanisms are not easily adapted to other systems.

Sun Microsystems's Network Software Environment (NSE) *copy/modify/merge* paradigm [Adams 89] is more formal: it is based on optimistic concurrency control [Kung 81], but when conflicts are detected during validation the transaction's updates must be merged with the previously committed version rather than rolled back. Unfortunately, merging has been formalized only for trivial programming languages [Horwitz 88], and makes sense for only a few types of objects such as source code and modification request logs, but not for others like object code and execution traces.

The University of Lancaster's Cosmos system [Walpole 88b, Walpole 88a] supports *domain relative addressing* with respect to persistent versions; this formalization is based on Reed's multiple-version implementation of serializability [Reed 78]. Each Cosmos transaction has its own domain, a set of immutable versions; another transaction can update the same object at the same time, by creating another version branch. Although merging is not required by the model per se, it is still necessary to resolve the divergent versions.

The *transaction groups* concept proposed for the ObServer system at Brown University [Skarra 89, Fernandez 89] defines a nested framework for cooperating transactions. Within a transaction group, member transactions and subgroups are synchronized according to some semantic correctness criteria appropriate for the application. The criteria are specified by

semantic patterns, and enforced by a recognizer and conflict detector, which must be constructed for each application. The implementation of transaction groups is supported by replacing classical locks with non-restrictive lock mode, communication mode pairs. The lock mode indicates whether the transaction intends to read or write the object and whether it permits reading while another transaction writes, writing while other transactions read and multiple writers of the same objects. The communication mode specifies 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 extended transaction models.

Dowson and Nejmeh have sketched a model called *visibility domains* [Dowson 89], which could probably be implemented using transaction groups. A visibility domain is defined in terms of access control to data. Any user who has access rights to a particular object may initiate a transaction that manipulates the object; any concurrent transactions operating on the object share the same shadow copy of the object, and serializability is not enforced among these transactions. As the authors admit, the details of the scheme remain to be elaborated. For example, it is unclear what it means for a transaction to commit: Does this make its updates visible to users without access rights to the updated objects? What if its update to an object was overwritten by an as yet uncommitted transaction? What if its update to an object was overwritten by a now aborted transaction? There are many other questions.

We have surveyed [Barghouti 89] many additional concurrency control models applied to other cooperative design environments such as for CAD/CAM, as well as for software engineering, but have found none with the completeness and wide applicability of the classical transaction model. A new, *flexible* transaction model is necessary to make integrated multi-user software development environments practical.

We have developed a flexible transaction model that supports both programmed and user-controlled transactions, but our primary concern is meeting the requirements of user-controlled transactions for software engineering applications. In this context, we have developed the notion of *commit-serializability* (summarized in Section 4) to capture object consistency in the classical sense and the notion of *participation domains* to manage object consistency in an extended sense (explained in Section 5).

4. Commit-Serializability

The term *commit-serializability* originally denoted the property that all committed transactions are in fact serializable in the classical sense (the definition of serializable is relaxed in the context of participation domains, as described in the next section), but these transactions may not correspond in a simple way to those transactions that were begun. In particular, transactions may be divided during operation and parts committed separately in such ways that the original transactions are not recognizable. Consider two in-progress transactions T_1 and T_2. T_1 is divided under program or user control into A and B, and shortly thereafter A commits while B continues. T_2 may view the committed updates of A, some of which were made by T_1 before the division, and then itself commits. B may then view the committed updates of T_2 before it commits. T_2, A and B are serializable, but T_1 and T_2 are not. It is misleading to think of T_1 as a nested transaction, with A and B as its subtransactions, since A and B are new top-level transactions (that is, at the same level as T_1). The original transaction T_1 in effect disappears, and in particular is neither committed nor aborted.

Commit-serializability is supported by two transaction processing operations that we have developed, *split-transaction* and *join-transaction*, in addition to the standard begin-transaction, commit-transaction and abort-transaction operations. The split-transaction operation supports the kind of division described above; the inverse join-transaction operation merges a completed transaction into an in-progress transaction to commit their results together.

```
Split-Transaction(
    A: ( AReadSet, AWriteSet, AProcedure ),
    B: ( BReadSet, BWriteSet, BProcedure ))

Join-Transaction(S: TID)
```

Figure 4-1: Split-Transaction and Join-Transaction

The split and join operations take the arguments shown in Figure 4-1. When the split-transaction operation is invoked during a transaction T, the transaction is split into two new transactions A and B. Either A or B, or neither, could obtain the transaction identifier of T — thus there is no distinction between saying T splits into A and B or T splits into T and S. TReadSet consists of all objects read by T but not updated and TWriteSet consists of all objects updated by T (alternatively, TReadSet could be all objects locked for reading by T and

TWriteSet all objects locked for writing, whether or not they had actually been read or written). The split-transaction operation divides TReadSet, not necessarily disjointly, into AReadSet and BReadSet. TWriteSet is divided disjointly into AWriteSet and BWriteSet.

This restriction on AWriteSet and BWriteSet guarantees serializability between the transactions A and B if both eventually commit. It is too strong, however, for the special case where A immediately commits, say, using a variant operation *split-transaction-and-commit*, or otherwise is guaranteed to commit before B and not introduce any other conflicts. In this case, objects in AWriteSet may also appear in either BReadSet or BWriteSet. In the case of a programmed transaction, AProcedure and BProcedure indicate the code for each new transaction to execute following the split. In the case of user-controlled transactions, these two parameters may be replaced by the userIDs of the two users (perhaps the same user) who will now interactively perform the operations within the new transactions.

Transaction $Alice_1$	Transaction $Alice_2$	Transaction Bob
begin		
read(M.c)		
read(N.c)		
write(N.c)		
read(O.c)		
write(O.c)		
write(N.o)		
write(O.o)		begin
read(M.o)		access(X)
read(N.o)		access(Y)
read(O.o)		access(Z)
		attempts read(N.c)
corresponding notification of attempted read(N.c)		
split((M,O), (N))		
	begin	
	commit(N)	
access(M)		*actual* read(N.c)
access(O)		write(N.o)
commit(M,O)		access(X)
		access(Y)
		access(Z)
		commit(N,X,Y, Z)

Figure 4-2: Example Split Schedule

Say a programmer Alice has begun a user-controlled transaction, read modules M, N and O and updated modules N and O. She has compiled the changed N and O, linked them together with the old object code for M, and is in the process of debugging. Another programmer Bob, as part of his own user-controlled transaction, attempts to access module N. Alice is notified of this attempt by the transaction manager. Since Alice is fairly sure she is done making changes to N, but needs to continue work on M and O, she splits her transaction $Alice_1$ into two transactions $Alice_1$ and $Alice_2$. She commits $Alice_2$, to commit the update to N, and then continues her work in $Alice_1$. Bob then reads N, decides to use this new version rather than the old one for testing his own changes to other modules, recompiles N and tests his subsystem. Later Bob commits his transaction, with any new updates to N, and Alice separately commits $Alice_1$, updating M and O. The corresponding schedule is depicted in Figure 4-2. (The c attribute of an object represents its source code and the o attribute its object code.)

It is possible to invoke an abort-transaction operation on a new transaction A or B resulting from a split-transaction. This does not automatically abort the other transaction, since they are now independent. In the case where B is user-controlled, however, if B is still ongoing when A aborts, it may be desirable to notify B that A has aborted and give B the option of subsequently aborting. So if for some reason the $Alice_2$ transaction is not able to commit, due perhaps to conflicts with other transactions in the system, Alice (the human) should be notified regarding the problem.

When the join-transaction operation is invoked during a transaction T, the target transaction S must be ongoing. Alternatively, two transactions T and S can be joined into a new transaction R; the distinction does not matter. TWriteSet and SWriteSet must be disjoint, as must be the pairs TReadSet, SWriteSet and TWriteSet, SReadSet. TReadSet and TWriteSet are added to SReadSet and SWriteSet, respectively, and S may continue or commit. The disjoint restrictions are necessary to ensure consistency within the ongoing transaction S, that is, T and S must not have accessed the same objects in a non-serializable manner, but is not required for serializability of the eventually committed S with respect to all other transactions. This could be weakened to allow T and S to have accessed objects non-serializably, by requiring a user-controlled merge as in NSE as part of the join.

Say a programmer Alice has read modules M, N and O and updated modules N and O. She has compiled the changed N and O, linked them together with the old object code for M, and

Transaction $Alice_1$	Transaction $Alice_2$	Transaction Bob
begin		
read(M.c)		
read(N.c)		
write(N.c)		
read(O.c)		
write(O.c)		
write(N.o)		
write(O.o)		begin
read(M.o)		access(X)
read(N.o)		access(Y)
read(O.o)		access(Z)
join(Bob)		
	begin	access(M)
	access(P)	access(X)
	access(Q)	access(N)
	access(R)	access(Y)
	commit(P,Q,R)	access(O)
		access(Z)
		commit(M,N,O, X,Y,Z)

Figure 4-3: Example Join Schedule

completed debugging. Another programmer Bob is working on other changes to the same subsystem. Since Alice is done, she joins her transaction $Alice_1$, with resources M, N and O, to Bob, so all changes to the subsystem will be published together when Bob eventually commits. Alice then goes on to her next task, initiating a new transaction $Alice_2$. The schedule for this example is displayed in Figure 4-3.

There is some subtlety regarding the circumstances under which a split-transaction or join-transaction could or should be performed during a software development activity. Commit-serializability as described above is concerned only with preserving read/update semantics, but it might be appropriate for a particular software development environment to consider more specific knowledge of the tasks involved, for example, commutativity of operations [Weihl 88], non-interference of different types of operations [Martin 87], dependencies among operations carried out in apparently conflicting transactions [Salem 87], or the possibility of undoing committed transactions using compensation functions [Garcia-Molina 87]. We currently handle only read/update semantics.

Special difficulties arise in advanced environments because transactions may interact with activities automatically initiated by the environment as opposed to a human user (effectively programmed transactions). It is questionable whether these activities should be treated as subtransactions and/or whether they should be treated as if directly controlled by the human user, since the user is not necessarily aware that they are executing in the background. We currently treat them as subtransactions, but assume human cognizance of their operation. These issues, as well as implementation concerns, are discussed in more detail elsewhere [Pu 88, Kaiser 89].

5. Participation Domains

The intent of serializability is that the entire set of transactions committed over the lifetime of a system must appear to have been executed in some serial order with respect to every external observer, even though the actual execution of the transactions has been interleaved and/or concurrent. Further, aborted transactions must appear never to have executed at all. The external observers may include programs, but prior to this work have always been assumed to include any human end-users interacting with the system. We have developed a new semantics of serializability where certain users can be designated as *participants* in a specific subset of the transactions, meaning these transactions need not appear to have been performed in some serial order with respect to these participants. A set of transactions, with a particular set of participants, is called a *domain* (this is not the same thing as a Cosmos domain or a visibility domain, although all these concepts are related). Other users remain *observers*, and this subset of the transactions must appear serial to these users. Participation is always with respect to some specific domain, so a particular user may be a participant for some domains and an observer for others within the same system.

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

The domain in which a user participates would typically be the set of transactions associated with the members of a cooperating group of users working towards a common goal. However, there is no implication that all the transactions in the domain commit together, or even that all of them commit (some may abort, split or join). Thus it is not correct to think of the domain as a top-level transaction, with each user's transaction as a subtransaction, although this is likely to be a frequent case in practice.

Each transaction is associated with zero or more particular domains at the time it is begun. A transaction that is not placed in any domain is the same as a classical (but user-controlled) transaction, with no participants except the one user. Such a transaction must be serializable with respect to all other transactions in the system. A transaction is placed in exactly one domain in order to non-serializably share objects with other transactions in the same domain, but it must be serializable with respect to all transactions not in the domain. A transaction may be placed in multiple domains because the intent is to carry out a subtask common to multiple goals. One big challenge is to prevent ''leakage'' between two domains where one or more users are participants in both, but the rest of the users are participants in only one and observers with respect to the other. Consider the simplest case, where there is only one user who participates in both of two domains, while all other users participating in one are observers for the other. In the general case, this requires that the dual-domain transaction be serializable with respect to the other transactions in both domains.

Say a domain D is defined to respond to a particular modification request, and programmers Alice and Bob begin transactions associated with D. The schedule shown in Figure 5-1 is not serializable according to the classical transaction model. Bob's transaction reads objects N and O written but not yet committed by Alice, modifies N, and then commits. Alice reads the version of N.c written by Bob, and then writes a new version N.c at the same time that Bob's transaction commits the previous version of N.c. Since Alice and Bob participate in the same domain D, this is legal with respect to Alice and Bob, and serializable according to our flexible transaction model.

But say there is another user, Charlie, whose transaction is not associated with domain D, as shown in Figure 5-2. Charlie reads O.c and writes O.c and O.o before Alice accesses O. This by itself would be legal, since Charlie's transaction thus far could be serialized before Alice's (but not after). But then Charlie reads the N.c committed by Bob. This is illegal. Bob cannot be

Transaction Alice	Transaction Bob
begin(D)	
read(M.c)	
read(N.c)	
write(N.c)	
read(O.c)	
write(O.c)	
write(N.o)	
write(O.o)	begin(D)
read(M.o)	read(M.o)
read(N.o)	read(N.o)
read(O.o)	read(O.o)
	read(N.c)
	write(N.c)
read(N.c)	write(N.o)
write(N.c)	commit(M,N,O)
write(N.o)	
read(M.o)	
read(N.o)	
read(O.o)	
commit(M,N,O)	

Figure 5-1: Example Participation Schedule

serialized before Charlie, and thus before Alice, because Bob reads the uncommitted N.c written by Alice. In fact, Bob's transaction cannot be serialized either before or after Alice's. This would not be a problem if it were never necessary to serialize Bob's transaction with any transactions outside the domain. His update to N would be irrelevant if Alice committed her final update to N before any transactions outside the domain accessed N. Thus the serializability of transactions within a participation domain need be enforced only with respect to what is actually observed by the users who are not participants in the domain.

6. Implementation Issues

Enforcement of serializability across participation domains may use locking, timestamps, optimistic methods, as for the classical transaction model, or borrow from one of the new mechanisms surveyed in Section 3. The requirement is to ensure that no transactions are committed unless they are serializable, using this new semantics, with respect to every previously committed transaction that is not associated with the same domain.

Using locking, the conflict in the example above would never have occurred, because Alice

Transaction Alice	Transaction Bob	Transaction Charlie
begin(D)		begin
read(M.c)		read(O.c)
read(N.c)		write(O.c)
write(N.c)		write(O.o)
read(O.c)		
write(O.c)		
write(N.o)		
write(O.o)	begin(D)	
read(M.o)	read(M.o)	
read(N.o)	read(N.o)	
read(O.o)	read(O.o)	
	read(N.c)	
	write(N.c)	
read(N.c)	write(N.o)	
write(N.c)	commit(M,N,O)	
write(N.o)		read(N.c)
read(M.o)		
read(N.o)		
read(O.o)		
commit(M,N,O)		

Figure 5-2: Example Participation Conflict

would not have been able to access O until after Charlie's transaction committed. Waiting for a lock to become available is unacceptable for most software development activities, but can be ameliorated by forking a new persistent version for Alice (leading to a later merging problem, as in NSE and Cosmos). Another problem with classical read and write locks is that Bob would not be able to access M, N and O as shown in the previous example (Figure 5-1). This could be solved using the non-restrictive locks of the transaction groups paradigm or one of the notify schemes common to existing multi-user software development environments (e.g., as in Apollo's DSEE [Leblang 84] or Biin's SMS (aka Gypsy) [Cohen 88]).

Using timestamps in the conflict example, either Charlie's transaction or Alice's would have the earlier timestamp. If Charlie's timestamp is earlier, then Alice reads Charlie's update to O, but the conflict is detected when Charlie attempts to read N — if single-version timestamps are used. Using multiple-version timestamps, Charlie reads an old version of N, so there is no conflict. If Alice's timestamp is earlier, then there is an unnecessary conflict in the single-version case when Alice attempts to read O. This can be solved using persistent versions.

Using optimistic concurrency control would reduce to essentially the copy/modify/merge scheme, where the later transaction to commit must merge any conflicting objects. The primary distinction would be that participants in the same domain would share the same shadow objects, as suggested for visibility domains.

Another implementation scheme would be to combine the non-restrictive locks and application-specific criteria suggested for transaction groups with the opportunities for intervention offered by advanced software development environments. In this context, one potential solution to the example problem would be to allow Alice's read of O but then have the environment delay Charlie's read of N until Alice commits her transaction. But this does not work for our semantics, because Charlie's and Alice's transactions still would not be serializable.

Instead, Charlie could be requested by the environment to commit his transaction, without accessing N, to force serialization of Charlie's transaction before Alice's. Charlie could decline to commit early, and not be able to access N, or Charlie could commit and then start up a new transaction in which to read N. Or he could execute the split-transaction command, to commit his updates to the offending objects (in this case O) without losing the context of his in-progress transaction. This decision could be made by the software development environment rather than by the human user in some cases. Another possibility would be for the environment to postpone the commitment of Bob's transaction, to prevent such conflicts a priori. Bob's transaction would be forced to commit immediately before Alice's, which overwrites its update. Or the join-transaction command could be applied to merge Bob's transaction into Alice's, and thus permit Bob to initiate another task. Charlie reads the old version of N, and as long as his transaction commits before Alice's, serializability is maintained.

7. Conclusions

The primary incentive behind participation domains as the basis for our flexible transaction model is that serialization conflicts can arise only across domains, never within domains, by definition. The intent is that most conflicts would normally arise between transactions associated with the same domain, if domains are carefully defined to represent appropriate software development activities. The users controlling these transactions are cooperating on the same group task and thus very likely to share objects. Users working on unrelated tasks are assumed to very rarely share objects, and in these few cases the cost of creating parallel versions,

with the latter merging problem, or of forcing an early commit (which can be done via split-transaction) or a late commit (via join-transaction) is acceptable.

We are currently considering an extension to participant domains that would remove the symmetry of domains, to permit one transaction to participate with respect to a second transaction without the second being allowed to access the uncommitted updates of the first. This would be useful for software development managers, who must oversee the work of many programmers working on their perhaps unrelated tasks. A mechanism of this form is already implemented in our Infuse software development environment, but has not yet been formalized. We plan to implement our full flexible transaction model in our Marvel system, rather than Infuse, since Marvel is a generic software development environment kernel and not restricted to any particular programming languages or tools. Our goal is to construct a range of multi-user software development environments based on our new transaction model.

Acknowledgments

Calton Pu collaborated with the author on the development of the split-transaction and join-transaction operations, with input from Norm Hutchinson. The development of participation domains and the consequent flexible transaction model was influenced by discussions with Nasser Barghouti, Geoff Clemm, Dan Duchamp, Stu Feldman, Dewayne Perry, Steve Popovich, Calton Pu, Soumitra Sengupta, Ian Thomas, Peter Wegner and Stan Zdonik. Participation domains were originally called *participant transactions*, but discussions with Mark Dowson led to renaming of the model.

References

[Adams 89] Evan W. Adams, Masahiro Honda and Terrence C. Miller.
Object Management in a CASE Environment.
In *11th International Conference on Software Engineering*, pages 154-163.
Pittsburgh PA, May, 1989.

[Archer 84] James E. Archer, Jr., Richard Conway and Fred B. Schneider.
User Recovery and Reversal in Interactive Systems.
ACM Transactions on Programming Languages and Systems 6(1):1-19,
January, 1984.

[Bancilhon 85] Francois Bancilhon, Won Kim and Henry Korth.
A Model of CAD Transactions.
In *11th International Conference on Very Large Databases*, pages 25-33.
Stockholm, August, 1985.

[Barghouti 89] Naser S. Barghouti.
Concurrency Control in Advanced Database Applications.
Technical Report CUCS-425-89, Columbia University Department of Computer Science, April, 1989.

[Cohen 88] Ellis S. Cohen, Dilip A. Soni, Raimund Gluecker, William M. Hasling, Robert W. Schwanke and Michael E. Wagner.
Version Management in Gypsy.
In Peter Henderson (editor), *ACM SIGSoft/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 201-215. ACM Press, Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[Dowson 87] Mark Dowson.
Integrated Project Support with IStar.
IEEE Software 4(6):6-15, November, 1987.

[Dowson 89] Mark Dowson and Brian Nejmeh.
Nested Transactions and Visibility Domains.
In *1989 ACM SIGMOD Workshop on Software CAD Databases*, pages 36-38. Napa CA, February, 1989.
Position paper.

[Eswaran 76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

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

[Fernandez 89] Mary F. Fernandez and Stanley B. Zdonik.
Transaction Groups: A Model for Controlling Cooperative Work.
In *3rd International Workshop on Persistent Object Systems*. Queensland, Australia, January, 1989.

[Garcia-Molina 87] Hector Garcia-Molina and Kenneth Salem.
SAGAS.
In *ACM SIGMOD 1987 Annual Conference*, pages 249-259. San Francisco CA, May, 1987.
Special issue of *SIGMOD Record*, 16(3), December 1987.

[Gray 81] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolo and Irving Traiger.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[Horwitz 88] Susan Horwitz, Jan Prins and Thomas Reps.
Integrating Non-Interfering Versions of Programs.
In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 133-145. San Diego, CA, January, 1988.

[Kaiser 87a] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *32nd IEEE Computer Society International Conference*, pages 236-241. San Francisco CA, February, 1987.

[Kaiser 87b] Gail E. Kaiser and Dewayne E. Perry.
Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution.
In *Conference on Software Maintenance*, pages 108-114. Austin TX, September, 1987.

[Kaiser 88a] Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich.
Intelligent Assistance for Software Development and Maintenance.
IEEE Software :40-49, May, 1988.

[Kaiser 88b] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler and Robert W. Schwanke.
Database Support for Knowledge-Based Engineering Environments.
IEEE Expert 3(2):18-32, Summer, 1988.

[Kaiser 89] Gail E. Kaiser.
A Marvelous Extended Transaction Processing Model.
In *11th World Computer Conference IFIP Congress '89*. Elsevier Science Publishers B.V., San Francisco CA, August, 1989.
In press. Available as Columbia University Department of Computer Science CUCS-404-88.

[Kung 81] H. T. Kung and John Robinson.
On Optimistic Methods for Concurrency Control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

[Leblang 84] David B. Leblang and Robert P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *SIGSoft/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, April, 1984.
Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[Martin 87] Bruce E. Martin.
Modeling Concurrent Activities with Nested Objects.
In *7th International Conference on Distributed Computing Systems*, pages 432-439. West Berlin, West Germany, September, 1987.

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

[Neuhold 88] Erich Neuhold and Michael Stonebraker (eds.).
Future Directions in DBMS Research.
Technical Report TR-88-001, International Computer Science Institute, Berkeley CA, May, 1988.

[Osterweil 87] Leon Osterweil.
Software Processes are Software Too.
In *9th International Conference on Software Engineering*, pages 1-13.
Monterey CA, March, 1987.

[Pausch 88] Randy Pausch.
Adding Input and Output to the Transactional Model.
PhD thesis, Carnegie Mellon University, August, 1988.
CMU-CS-88-171.

[Pu 88] Calton Pu, Gail E. Kaiser and Norman Hutchinson.
Split-Transactions for Open-Ended Activities.
In *14th International Conference on Very Large Data Bases*, pages 26-37.
Los Angeles CA, August, 1988.

[Reed 78] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.
MIT LCS TR-205.

[Rowe 89] Lawrence A. Rowe and Sharon Wensel (ed.).
1989 ACM SIGMOD Workshop on Software CAD Databases.
February, 1989

[Salem 87] Kenneth Salem, Hector Garcia-Molina and Rafael Alonso.
Altruistic Locking: A Strategy for Coping with Long Lived Transactions.
In *2nd international Workshop on High Performance Transaction Systems.*
Pacific Grove CA, September, 1987.

[Skarra 86] Andrea H. Skarra, Stanley B. Zdonik and Stephen P. Reiss.
An Object Server for an Object-Oriented Database System.
In *1986 International Workshop on Object-Oriented Database Systems*, pages 196-204. Pacific Grove, CA, September, 1986.

[Skarra 89] Andrea H. Skarra and Stanley B. Zdonik.
Concurrency Control and Object-Oriented Databases.
Object-Oriented Concepts, Databases, and Applications.
ACM Press, New York, 1989, pages 395-421.

[Spector 88] A. Z. Spector, R. Pausch and G. Bruell.
Camelot, A Flexible, Distributed Transaction Processing System.
In *33rd IEEE Computer Society International Conference*, pages 432-437.
San Francisco CA, March, 1988.

[Taylor 88] Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil and Alex L. Wolf.
Foundations for the Arcadia Environment Architecture.
In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1-13. Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[Tichy 85] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[Walpole 88a] J. Walpole, G.S. Blair, J. Malik and J.R. Nicol.
A Unifying Model for Consistent Distributed Software Development Environments.
In *ACM SIGSOFT/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 183-190. Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[Walpole 88b] J. Walpole, G.S. Blair, J. Malik and J.R. Nicol.
Maintaining Consistency in Distributed Software Engineering Environments.
In *8th International Conference on Distributed Computing Systems*, pages 418-425. San Jose, June, 1988.

[Weihl 88] William E. Weihl.
Commutativity-Based Concurrency Control for Abstract Data Types (Preliminary Report).
In *21st Annual Hawaii International Conference on System Sciences*, pages 205-214. Kona, HI, January, 1988.