Cooperative Transactions for Multi-User Environments

Gail E. Kaiser

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
212-939-7081/fax:212-666-0140
kaiser@cs.columbia.edu

CUCS-006-93
March 1993 (revised December 1993)

Abstract

This chapter surveys extended transaction models proposed to support long duration, interactive and/or cooperative activities in the context of multi-user software development and CAD/CAM environments. Many of these are variants of the checkout model, which addresses the long duration and interactive nature of the activities supported by environments but still isolates environment users, making it difficult for them to collaborate while their activities are in progress. However, a few cooperative transaction models have been proposed to facilitate collaboration, usually while maintaining some guarantees of consistency.

To appear as a chapter in Won Kim (ed.), *Modern Database Management: Object-Oriented and Multidatabase Technologies*, ACM Press.

1. Introduction

Conventional database transactions guarantee atomicity in two senses: concurrency atomicity, for consistent concurrent access, and failure atomicity, for consistent persistence. Unfortunately, these atomicity properties severely limit the applicability of the transaction concept with respect to modern database applications such as software development and CAD/CAM environments, even though these applications still need consistent concurrent access and persistence.

From a transactional perspective, multi-user environments are characterized by:

- *Long duration*: Activities consisting of sequences of database accesses may last from minutes to months. Rollback of long transactions, either for concurrency control or failure recovery, is generally unacceptable due to the economic and morale costs of lost work.
- *Interactive control*: Users choose the actions within their activities as they go along. It is usually inconvenient to plan transaction schedules *a priori*, and automatic ‘‘redo’’ of activities is often infeasible.
- *Cooperation among users*: Users share partial results of their activities while still in progress. Serializability is sometimes incompatible with consistency, when concerted effort is necessary to take the database from one semantically consistent state to another.

One widely accepted approach to long and interactive activities is the ‘‘checkout’’ model, in which users copy objects from the shared repository to private work areas for manipulation. A user might reserve these objects exclusively until a later ‘‘checkin’’, or multiple users might later ‘‘merge’’ their updates to the same (or semantically related) objects. Neither mechanism, however, directly addresses cooperative activities — so users are tempted to communicate outside system control. This chapter surveys proposed approaches to all three problems, with particular emphasis on supporting cooperation. We are primarily concerned with concurrency control and do not explicitly address failure recovery, although recovery of some sort is usually required. Most of the papers mentioned are relatively recent, but note that many of their ideas were foreshadowed in the 1970’s [Davies 73, Davies 78].

The reader is assumed to be familiar with the conventional atomic transaction model [Bernstein *et al.* 87] and its main implementation mechanisms, such as two-phase locking [Eswaran *et al.* 76], multi-version timestamp ordering [Reed 78], optimistic validation [Kung and Robinson 81], multi-granularity locking [Gray *et al.* 75], and nested transactions [Moss 82]. We start with a small motivating example, which will be referred to throughout the chapter. Next, we explain the checkout model in more detail and consider some extensions, which provide a range of functionality for coordinating the long duration, interactive activities of multiple environment users (i.e., only the first two requirements above). This is followed by a brief synopsis of a generic model that supplies primitives for constructing a range of checkout-like nested transaction models. We then describe some proposals that address synergistic cooperation among the users while their work is in progress (thus addressing all three of the requirements). A table summarizes all the models surveyed, and then the chapter ends with a brief discussion of research directions.

2. Motivating Example

Two programmers, Alice and Bob, are developing the same program consisting of three modules: X, Y and Z. Modules X and Y consist of procedures and declarations that comprise the main code of the program; module Z is a library of procedures called in modules X and Y. When testing the program, two bugs are discovered. Bob is assigned the task of fixing one bug that is suspected to be in module X. He starts working on X. Alice's task is to explore a possible bug in the code of module Y, so she starts browsing Y. After a while, Bob finds that the bug in X is caused by bugs in some of the procedures in the library module. After editing Z, Bob proceeds to compile and test the modified code.

Alice finds a bug in module Y and modifies various parts of the module's code to fix it. Alice now wants to test the new Y. She is not concerned with the modifications that Bob made in X because she believes they are unrelated to her problem. However, she wants to access Bob's changes to module Z because some procedures in Z are called from Y: Bob's edits might have introduced semantic inconsistencies with Y's code. But since Bob is still working on modules X and Z, Alice will either have to access module Z at the same time that Bob is working on it or wait until he is done. Let's assume waiting is unacceptable, because then Alice would be idle until Bob finished his task.

If traditional concurrency control based on two-phase locking were used, Bob and Alice would not be able to access module Z at the same time. They could concurrently lock modules Y and X, respectively, since they work in isolation on these modules. But these users need to work *cooperatively* on module Z, and thus neither of them can lock it for the duration of their tasks. Recall that Bob writes Z and Alice reads it, and write and read locks are normally considered incompatible. Even if the locks were at the finer granularity of individual procedures, they would still have a problem because Bob and Alice need to access the same procedures (e.g., to recompile the module).

A variant of multi-version timestamp ordering might seem to solve the problem, by supporting parallel versions of module Z. Alice would access the previously compiled version while Bob works on a new version. But this would require Alice to later retest her code after the new version of Z is released, resulting in unnecessary work, at least as undesirable as the waiting suggested above. Or optimistic validation could be employed, implicitly introducing parallel versions accessed by Bob and Alice. Here the read/write conflict would be detected only after Bob's work was completed, rolling back (throwing away) his changes! Any other mechanism for implementing conventional transactions would also run into difficulties, because it would necessarily enforce serializability of Alice's and Bob's tasks.

Since this example is trivial, there are many obvious ‘‘work-arounds’’. But few scale up to even medium scale systems with perhaps hundreds of modules and tens of programmers. The next section discusses the checkout model and several extensions, variants of which are typically used in practice — even though they have serious limitations with respect to cooperation while work is in progress.

3. Coordination

When a small team of environment users works together on a project, the members of the team work autonomously, for the most part, but trust the others to act in a reasonable way. There are only a few rules that need to be enforced to maintain smooth interactions among the members of a small team. The environment should provide a means of orchestrating the interactions of the users, with the goal that information and effort is neither lost nor duplicated as a result of their simultaneous activities. In

particular, it is necessary to *coordinate* the concurrent access to the shared repository in which the project components are stored [Perry and Kaiser 91].

3.1. Basic Checkout Model

The basic checkout model, in tandem with versions and configurations, are supported by numerous commercially marketed tools for software development (e.g., Adele [Estublier *et al.* 84], DSEE [Leblang and Chase 87], SMS [Schwanke *et al.* 89]). [Katz 90] gives a comprehensive overview of version and configuration systems oriented towards CAD/CAM environments. Most of these provide some of the capabilities outlined here, but in the text we cite only representative (and often early) examples.

3.1.1. Versions

The simplest form of coordination among members of a design or development team is to control the access to shared objects so that only one user can modify any particular object at a time. The *checkout/checkin* approach has been implemented by widely-used version control tools like SCCS [Rochkind 75] and RCS [Tichy 85]. Each object is considered to be a collection of multiple *versions*. A version represents the state of the object at some time in the history of its development. The versions of an object are usually stored together in a compact representation that allows the reconstruction of any specific version when it is needed.

A version may become *immutable*, which means that it can no longer be modified. Instead, a new successor version can be created after checking out (reserving) the object. The initial reservation makes a copy of the indicated version and gives the owner of the reservation exclusive access to the copy so that he or she can modify it and check it in (deposit it) as a new version. Other users who need to access the same object must wait until the new version has been deposited (and the reservation released) or reserve another version. Two or more users can modify the same object only by working on parallel versions, creating *branches* in the version history. Branching ensures write-serializability among the versions of an object. The result of consecutive reserves, deposits and branches is a version tree that records the full history of development of the single conceptual object. Branches of the version tree can be merged to combine the changes with respect to the least common ancestor along two or more paths.

This scheme is pessimistic since it does not allow access conflicts to occur on what is intended to be the same version (rather than allowing them to occur and then correcting them as in optimistic schemes, see section 3.3). It is conceptually optimistic, on the other hand, in the sense that it allows multiple parallel versions of the same object to be created even though it is known that these versions will have to be merged later. That is, it assumes any semantic conflicts between version branches can be adequately resolved (usually by one user manually constructing a new version to reflect the changes introduced in several versions, although some progress has been made in automatic merging [Horwitz *et al.* 89]).

Considering our example, Bob and Alice would checkout modules X and Y, respectively; Bob would later checkout module Z when he discovered it was also needed for his task. Since Alice does not need to modify Z, she could use whatever commands are provided by the version management tool to access the most recently checked in version of Z in read-only mode and simply release her reservation when done, so a new version branch would not be created. As in the multi-version timestamp ordering approach mentioned in section 2, Alice would initially test with the old code for Z and then repeat her tests after

Bob checked in his new version of Z. The unnecessary work might go even further, though, since Alice could find errors in Z during her testing and repair them herself in a parallel version (in which case she does create a new branch) — even though Bob may already be fixing the same problems! Alternatively, Alice could negotiate with Bob (outside control of the tool), asking him to checkin his changes to Z while maintaining his reservation of X. Then Alice could use the new version of Z for her testing. But this may result in incompatible versions of X and Z, since there is nothing forcing Bob to eventually checkin a consistent version of X, or even checkin X at all as opposed to releasing his reservation.

The basic checkout model provides minimal coordination between multiple users. Like the classical transaction model, it does not rely on semantic information about the objects or the computations 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 sub-object individually. This can lead to problems if a user reserves several objects all of which belong conceptually to one aggregate object, creates new versions of each of them, makes sure that they are consistent as a set, and then neglects to deposit some of the objects. Second, the reserve/deposit mechanism often does not provide any control over reserved objects beyond locking them in the public database. Thus, once an object has been reserved by a user, it may become available outside concurrency control mechanism. The owner of the reservation can decide to let other users access and even modify that object (e.g., through permissions granted on the private work area *via* the file system).

Itasca (previously ORION-2 [Kim *et al.* 91]) solves the second problem, through a distributed database incorporating *private databases*, into which objects are checked out from the shared database, as well as a query facility that distinguishes between private and shared databases. There is no access to objects outside of system control. However, Itasca's support for composite objects, in the sense of maintaining references between objects, does not attach any semantics to the references. In particular, it allows sub-objects of a composite object to be reserved and deposited independently by the same or different users, so the first problem remains.

3.1.2. Configurations

The key omission is not keeping track of which versions of objects are consistent with each other. For example, if each component (object) of a program has multiple versions, it would be impossible to find out which versions of the components actually contributed to producing a particular executable that is being tested. It is necessary to group sets of versions that are consistent with each other or otherwise used together into *configurations*.

[Walpole *et al.* 88a] introduced *domain relative addressing*, which supports versions of configurations by extending [Reed 78]'s notion of time relative addressing (multi-version concurrency control). Whereas Reed's algorithm synchronizes accesses to objects with respect to their timestamp, domain relative addressing does so with respect to their ''domain''. A domain is essentially a configuration, consisting of one selected version for each of a related set of objects. Domains are consistent, by definition, since a (long) transaction reads only those versions in its input domain and writes new versions generating an output domain [Walpole *et al.* 88b]. Even when multiple transactions employ the same input domains concurrently, they produce distinct output domains.

To illustrate this approach, consider the two long transactions T_{Alice} and T_{Bob} of figure 3-1. The schedule

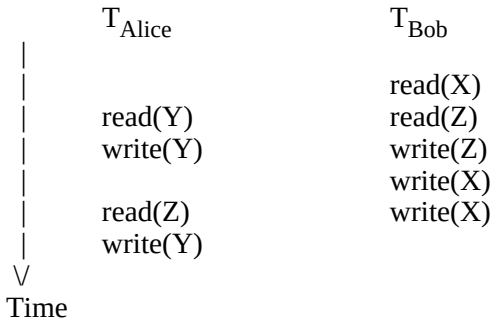

Figure 3-1: Domain Relative Addressing Schedule

shown is serializable, with T_{Alice} either before T_{Bob} or vice versa. Applying domain relative addressing, however, the checkout operation is implicitly invoked by the first read of an object, but checkin is triggered only on explicit commit and has thus not occurred yet (it is not automatically invoked by a write). So T_{Alice} is necessarily serialized before T_{Bob}, meaning she reads the old version of Z before Bob's changes. Although we still have the retesting issue, it is not possible for Bob to introduce a semantic inconsistency by checking in only his changes to Z (or X) and aborting his changes to X (or Z). Instead, an entire configuration representing all checked out versions must be deposited atomically, as a new version of the configuration.

3.2. Semantic Coordination

Configuration management assumes that the versions of objects contained in the same configuration are semantically consistent, just like the classical transaction model assumes that a transaction's actions takes the database objects it accesses from one semantically consistent state to another. But neither enforces any particular integrity constraints, because those are necessarily specific to the application.

3.2.1. Enforcing Consistency

Smile [Kaiser and Feiler 87] is a multi-user software development environment for C programming. As in Itasca, Smile maintains private work areas for individual users as well as a shared repository, and supports global queries encompassing both the user's private database and the shared database. As in domain relative addressing, Smile requires the entire contents of a private work area (called an *experimental database*) to be checked in atomically. Unlike domain relative addressing, however, it does not support versions of either objects or configurations. Instead, there is always one public configuration in the *main database* consisting of the only public version of each object, and any number of private configurations containing private versions that have not yet been deposited.

Smile's novel aspect is that it adds semantics-based consistency enforcement to the checkout model. The database contains C program components, e.g., functions, type definitions and variable declarations, collected into modules corresponding roughly to a C source file plus a C header file. The module is the unit of checkout/checkin, and additional modules can be added to the user's experimental database at any time. When the user requests checkin of his or her experimental database, Smile analyzes all its modules using the Lint tool [Johnson 78], recompiles them, and links them together with the unmodified modules

in the main database. If any errors are detected during this process, the checkin operation is automatically aborted; the user is expected to repair the problems and try again.

Considering our example, Bob starts a long transaction T_{Bob}, which creates a new experimental database EDB_{Bob}, and copies module X into this database. He later requests to add module Z to his experimental database. Now X and Z are exclusively reserved to T_{Bob} and no other transaction can also reserve them. Other long transactions, however, can continue to read the baseline versions of these modules from the main database. For example, Alice can connect directly to the main database to perform read-only queries, or can start her own update transaction on Y in EDB_{Alice}. Bob then proceeds to edit the bodies of X and Z. When the modification process is complete, he requests a deposit operation to copy the updated X and Z to the main database and thus make his changes available to other users.

Smile analyzes and compiles the new X and Z together with Y in the main database (i.e., a module in an experimental database might import from a module in the main database). The baseline might have changed while T_{Bob} was in operation, if Alice checked in an experimental database in the meantime, overwriting the previous baseline. If the analysis and compilation both succeed, the modules are deposited and T_{Bob} commits. Otherwise, Bob is informed of the errors and the deposit is aborted; Bob has to fix the errors in his modules and repeat the deposit operation when he is ready. The commit of T_{Bob} does not automatically delete EDB_{Bob}, so the user can continue to manipulate its copies of the modules, if desired, but it is not possible to checkin anything further from EDB_{Bob} into the main database or checkout any other modules from the main database into EDB_{Bob}.

Smile not only enforces consistency among the objects accessed within the same long transaction, but it also enforces global consistency with respect to all other objects. The description above considers only *a posteori* consistency checking, at the time of (intended) deposit, after the changes have already been made. To minimize unsuccessful deposits, with their recompilation overhead, Smile also enforces consistency in an *a priori* manner: a user is not permitted to change the interface of a module unless he or she has reserved all other modules that depend on that interface — or, actually, on the portion of the interface intended to be changed (in the sense of ‘‘smart recompilation’’ [Tichy 86]). For example, say function f of module Z is called by function g of module Y. The internal code of f can be changed once Z has been reserved, but Bob also has to reserve module Y before he can modify f’s external signature (parameter and return types). If another long transaction T_{Alice} has already reserved module Y in another experimental database, EDB_{Alice}, the operation to edit f’s signature is aborted. T_{Bob} is forced to either wait until T_{Alice} deposits Y, at which point T_{Bob} can reserve it as an addition to EDB_{Bob}, or to work on another task that does not affect Z. From this example, it should be clear that enforcing semantics-based consistency leads Smile to restrict concurrency possibly even more than the previously discussed variants of the checkout model.

3.2.2. Multi-Level Consistency

Infuse [Kaiser *et al.* 89] is another multi-user environment for C programming. As in Smile, copies of modules are checked out into experimental databases representing work areas; unlike Smile, Infuse’s experimental databases are not necessarily private to a single user. Instead, Infuse supports a *multi-level hierarchy*: all modules are reserved from the main database into the same top-level experimental database shared among all users, from which modules may be reserved into child experimental databases representing groups of users, which may in turn have their own child experimental databases for

subgroups, until eventually reaching leaf experimental databases where actual changes are normally made by individuals. The top-level experimental database corresponds roughly to a top-level nested transaction.

Infuse further extends Smile from static (compilation-time) to dynamic (execution-time) consistency checking. Within a given experimental database, no checking is initiated until all of its child experimental databases have been successfully deposited. Then the locally reserved modules are compiled and linked against *stub* modules. The interfaces of stub modules represent the corresponding modules from the baseline (or reserved elsewhere in the hierarchy), but their functionality is sufficient only for testing of the locally reserved modules — for example, a function defined by a stub module might perform no computation but only prompt the user for its return value. The modules reserved in an experimental database cannot be deposited into the parent experimental database until they have passed an appropriate test suite. The entire system is compiled and tested in the top-level experimental database, before deposit of the changed modules into the main database to replace the baseline.

Let's reconsider our example, where Bob is assigned module X and Alice module Y. Either user creates a top-level experimental database $EDB_{X,Y}$, into which X and Y are reserved. Bob creates a child experimental database EDB_X in which he reserves X and Alice creates EDB_Y in which she reserves Y. Again, Bob later adds Z, this time to both the top-level experimental database, now $EDB_{X,Y,Z}$ and his private area $EDB_{X,Z}$. Bob again changes the interface of function f, e.g., by adding a new parameter. Unlike Smile, Infuse permits Bob to make this change even while Alice is simultaneously modifying module Y (recall that function g of module Y calls f in module Z). After Bob completes his changes, he deposits the contents (X and Z) of $EDB_{X,Z}$ into $EDB_{X,Y,Z}$. This operation automatically initiates recompilation and testing using local stub modules, if that had not been performed previously.

Alice separately finishes her changes and deposits Y (EDB_Y) into $EDB_{X,Y,Z}$. Semantic checking is performed only with respect to Y in isolation, so no errors are found at this point. However, when either Bob or Alice attempts to deposit the modules in $EDB_{X,Y,Z}$ to the main database, the compiler reports that modules Y and Z are not consistent with each other because of function f's new parameter. At that point, either Bob or Alice must create a child experimental database EDB_Y' (the previous EDB_Y has been discarded) in which he or she can fix the problem, by modifying the call to f in function g. Infuse thus allows greater concurrency than Smile at the cost of potentially greater inconsistency — and the need for a later round of changes to re-establish consistency.

Infuse provides additional features useful for large scale projects. For example, authorized users can issue read-only queries, called *transcendent transactions* [Kaiser and Perry 91], against any subset of the experimental databases in the hierarchy. This permits a manager to keep tabs on the progress of the work. Infuse also provides a facility whereby users can define *workspaces* [Kaiser and Perry 87] that temporarily group together any two or more experimental databases anywhere in the hierarchy, for the purpose of early consistency checking among the modules represented by those databases. This is akin to the various group models explained in section 5.2, in the sense that the normal isolation among long transactions is relaxed among members of a group, but Infuse workspaces permit the groupings to be formed and destroyed on the fly as needed.

3.3. Optimistic Coordination

The extended transaction models presented so far severely restrict concurrent access to the same object. There is usually read-only access to previously checked in versions, but no more than one transaction can modify the same version at the same time. Although a sequence of parent transactions reserves a module in Infuse, the module can be edited only in a leaf transaction (i.e., one with no children of its own). It is often the case in software development efforts, however, that two or more users in the same group need to change the same object concurrently. Since these users are typically familiar with each other's plans, they are assumed to be able to resolve any conflicts they introduce during their concurrent accesses by *merging* their changes into a single consistent version. Note this is different from domain relative addressing, where transactions may read the same version but always write distinct versions.

Like Infuse, the Network Software Environment (NSE) [Honda 88] operates on a hierarchical database structure. Each node of this structure is called an *environment* (not to be confused with a software development or CAD/CAM environment). An environment presents a read-only view of (potentially) the entire file system but requires updates to be made on private copies of files. Rather than generating permanent configurations as in domain relative addressing, NSE's concurrency control policy permits multiple sibling environments to checkout simultaneously what is conceptually the same version of an object and, when conflicts are discovered at checkin, merge their updates back into the shared file system.

The first environment to finish its work on a particular file deposits its copy as the new version of the file in its parent environment, the second child environment to finish has to merge its copy with the first environment's version, creating a newer version. The third environment to commit will merge its copy with this newer version and so on. This is distinct from creating branching versions, since all the checked in copies add versions to the original branch. (NSE uses the SCCS tool internally to manage these versions.) In effect, the validation phase of the conventional optimistic concurrency control scheme is modified to replace rollback of conflicting transactions with merging of updates. [Adams *et al.* 89] refers to this approach as *copy/modify/merge*.

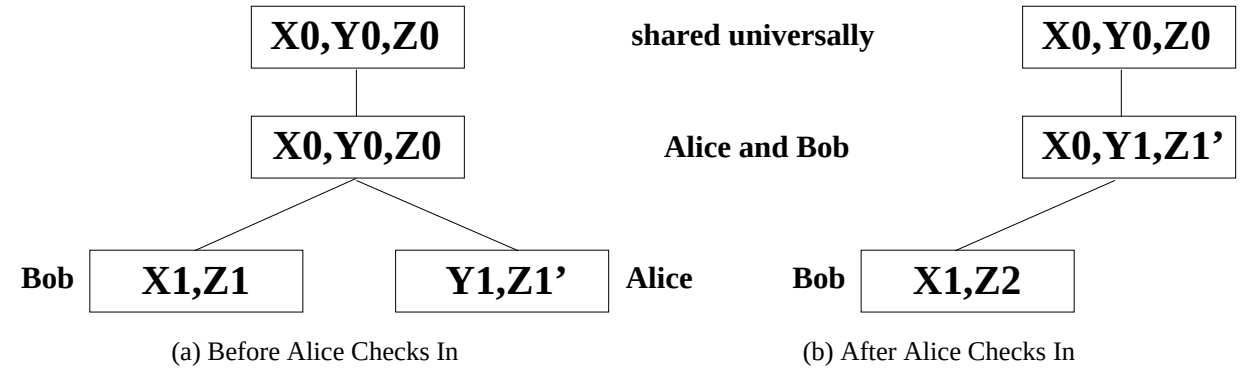

Figure 3-2: NSE Environments

Illustrating this approach, either Bob or Alice creates a top-level environment $ENV_{X,Y}$ to fix X and Y. Bob then creates his own child environment ENV_X to edit X. Z is added later, resulting in $ENV_{X,Y,Z}$ and $ENV_{X,Z}$. Alice controls her own child environment, ENV_Y. (Note that there is nothing preventing Z

from being checked out at the same time as X, but here we follow the scenario of section 2.)

When Alice needs to compile and link the program for testing, she normally accesses the old version of Z in $ENV_{X,Y,Z}$. However, as in the basic checkout model of section 3.1.1, she can negotiate directly with Bob to retrieve his in-progress copy of Z or to ask him to deposit it early. Say she uses the old version, finds some bugs tickled by her own changes to Y, and decides to fix them herself. In NSE, she can checkout Z for updates even though Bob already has it checked out, generating $ENV_{Y,Z}$. But this introduces a race to determine which of Alice and Bob checks in Z first. The ‘‘loser’’ becomes responsible for correctly merging their changes, and it is still necessary for Alice to retest Y using the resulting version of Z. An instance where Alice wins the race and Bob merges the two variants is illustrated in figure 3-2.

4. Modeling Coordination

Domain relative addressing, Smile and Infuse all support an explicit atomic transaction model in the sense that a (long) transaction begins, work proceeds within that transaction, and eventually the transaction either commits or aborts. Either all the changes made during the transaction are retained or none of them are. In contrast, the basic checkout model, Itasca and NSE all allow individual objects to be reserved and deposited independently, with no formal notion of maintaining consistency among multiple objects. Walter [Walter 84] proposes a middle ground with the basic structure of nested transactions but a more flexible relationship between parent and child transactions.

In his model, the interface between a parent transaction and a child transaction can either be single-request, where the parent in effect issues a query or update to the child and waits until the child returns the full result atomically, or *conversational*, meaning the control alternates between the parent that generates a series of requests and the child that answers these requests individually. The conversational interface necessitates grouping the parent and child transactions in the same *backout sphere* because if the child transaction is aborted (for any reason) in the middle of a conversation, not only does the system have to rollback the changes of the child transaction, but the parent transaction has to be rolled back (or backed out) to the point before the entire conversation began (not just the current request). In general, a backout sphere includes all transactions that are involved in a chain of conversations, where all of them must be backed out if any one of them is.

In contrast, the single-request interface supported by the traditional nested transaction model does not require rolling back the parent because the child transaction’s computation does not affect the computation in progress in the parent transaction. In this framework, the basic checkout model can be viewed as supporting a multiple-request interface where there can be multiple outstanding requests at the same time, each of the requests concerns only a single object, and a child transaction multi-tasks among computations towards answering the pending requests. A child transaction might never abort *in toto* but can rollback its changes to an individual object since its last deposit of that object.

In addition to backout spheres, Walter also proposed the idea of *commit spheres*. Any transaction within a commit sphere can commit only if all other transactions in its sphere also commit. One can imagine extended transaction models where child transactions are not necessarily in the same commit sphere as their parent (e.g., the transactions resulting from the split operation introduced in section 5.1.2 are not in the same commit sphere), so a child could commit persistently even though its parent aborts.

Finally, Walter investigated the *synchronization* between parent and child transactions. His single-request and conversational interfaces as described above assume synchronous interaction, as does our suggested multiple-request extension, but asynchronous communication between the parent and child is also plausible. Here computation continues in the parent transaction while the child transaction is also active. Thus there arises the need for controlling concurrent access to objects shared between the parent and child, *via* locking or some other scheme.

Given these three aspects — interface, dependency and synchronization — Walter presented a nested transaction model in which each subtransaction has three attributes defined when it is created. The first attribute, reflecting the interface criterion, can be set to either `BACKOUT` or `NOBACKOUT`, with `BACKOUT` indicating that the child transaction is associated with its own backout sphere and `NOBACKOUT` meaning that it shares a backout sphere with its parent transaction. The dependency attribute is set to either `COMMIT` or `NOCOMMIT`, 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 `BACKOUT`, `COMMIT` and `SYNC` is independent of its parent since it possesses its own backout sphere and its own commit sphere, and locking is needed to synchronize access to objects shared with its parent.

Walter claims that it is possible to define all other nested transaction models within his model. The classical nested transaction model, for example, is defined as creating subtransactions with attributes set to `BACKOUT`, `NOCOMMIT` and `SYNC`. Infuse and NSE arguably could both be defined using `BACKOUT`, `NOCOMMIT` and `NOSYNC`, even though their behavior is substantially different — Walter's model is apparently insufficient for distinguishing between them. In particular, Infuse's interface attribute is `BACKOUT` because the single-request interface inherently places the parent and child in different backout spheres. NSE's interface attribute is also `BACKOUT` because the multiple-request interface places the parent and child in different backout spheres with respect to each distinct request; that is, releasing a reservation without checkin has no effect on previous checkins of the same file by either the same or different child environments. Neither Infuse or NSE allows a subtransaction to commit its changes to the shared repository without depositing through the multiple levels of ancestors, thus `NOCOMMIT`. Finally, there is no synchronization (`NOSYNC`) between a parent and child transaction, in Infuse because there cannot be any work done in a parent experimental database until all the children have committed and thus the synchronization issue does not arise, and in NSE because concurrent updates to the same file are permitted (and merged later).

5. Cooperation

As the size of a project grows, the interactions among the team members increase both in number and in complexity. Although small teams can allow a great deal of freedom, assuming cooperation is effected by frequent direct communications outside system control, larger populations require complicated rules and regulations with their attendant restrictions on individual freedom. Large teams are typically subdivided into several groups, each responsible for a part of the design or development task. Members of a group then cooperate with each other to complete their parts, so it is necessary for the environment to support cooperation among members of the same group, as well as coordination of multiple groups [Perry and Kaiser 91]. The mechanisms described in section 4 generally handle the former well but are relatively weak regarding the latter. (Although Infuse was intended for large teams divided into smaller groups, it

focuses on the coordination among groups without much concern for the cooperation within groups, so is described in section 3 with the other approaches concentrating on coordination.)

5.1. Cooperation Primitives

Walter's primitives described in section 4 form a framework for *reasoning about* different transaction models, but the imprecision in distinguishing Infuse from NSE points to some difficulty regarding how they might be used to *implement* such models. However, others have proposed primitives that are, at least in principle, intended to be employed directly in implementing new transactional systems.

5.1.1. Notification

One approach to maintaining consistency, while still allowing some cooperation, is to support notification and interactive conflict resolution rather than enforcing serializability. The Gordion database system [Ege and Ellis 87] provides a notification primitive that can be used in conjunction with other primitives to implement cooperative concurrency control policies. Notifications alert users about ''interesting'' events such as an attempt to lock an object that has already been locked in an exclusive mode or to create a new version of an object that is already checked out.

Two policies that use notification in conjunction with non-exclusive locks and versions were implemented in the Gordion system: *immediate notification* and *delayed notification* [Yeh *et al.* 87]. Immediate notification alerts the affected users of any attempt at conflicting access as soon as the conflict occurs. Delayed notification alerts the users of all the conflicts that have occurred only when one of the conflicting (long) transactions attempts to commit. (NSE supports an intermediate point, through its `resync` command, to request any pending notifications on demand.) Gordion resolves conflicts by instigating a ''phone call'' between the two parties with the assumption that they can interact directly to resolve the conflict (it might be more practical to send electronic mail, or allow the user to have specified an appropriate action for each class of notification in advance [Leblang and Chase 84]).

These policies incorporate humans as part of the conflict resolution algorithm. On the one hand, this enhances concurrency when many of the tasks are interactive. On the other hand, it can degrade consistency because humans cannot always be relied on to correctly resolve conflicts. It might be preferable for the environment to support some form of automatic consistency checking, as in Smile and Infuse, or intelligent tools, such as NSE's merge utility (see section 3).

5.1.2. Dynamic Restructuring

Two new operations, `split-transaction` and `join-transaction`, have been proposed for restructuring long, interactive transactions while they are in progress [Kaiser and Pu 92]. The basic idea is that all sequences of database accesses that are included in a set of concurrent transactions are performed in a schedule that is serializable at the point when the transactions 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` operation divides an ongoing transaction into two or more serializable transactions by dividing the actions and the resources (e.g., locked objects) between the new transactions.

The resulting transactions can proceed independently from that point on, perhaps controlled by different users, and behave as if they had been independent all along. The original transaction disappears entirely, as if it had never existed. The `split-transaction` operation can be applied only when it is possible to generate two transactions that are serializable with each other as well as all other transactions.

One application of `split-transaction` is to commit one of the new transactions in order to release all of its resources so that they can be acquired by other transactions [Pu *et al.* 88]. In this case, the splitting of a transaction reflects the fact that the user who controlled the original transaction has decided that he is done with some of the resources reserved by his transaction — so these resources can be treated as held by a separate transaction that now commits. The splitting of a transaction generally results from new information determined while the transaction is in progress, in this case the dynamic access pattern of the transaction (the fact that it no longer needs some resources). Thus it would not have been possible to initiate the eventually split transactions as independent top-level transactions *a priori* (as in [Garcia-Molina and Salem 87], where a ‘‘saga’’ consists of a sequence of conventional transactions and interleaving among sagas is permitted at transaction boundaries).

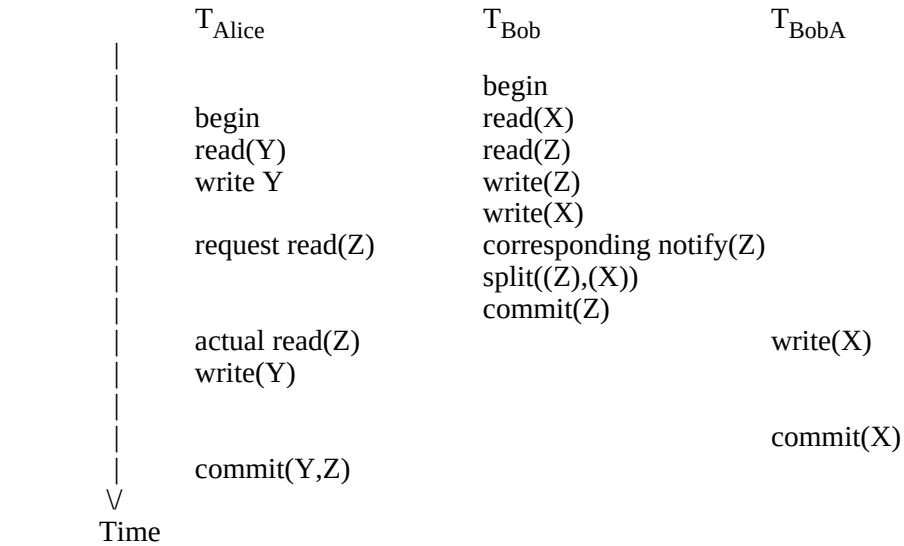

Figure 5-1: Split-Transaction Schedule

To clarify this technique, suppose that Alice and Bob start long transactions T_{Bob} and T_{Alice} to modify the modules X and Y, respectively. Again, T_{Bob} later decides to also edit module Z and, after a while, Alice discovers that she too needs to access Z. We assume environmental support for notification similar to that described above. On being notified that T_{Alice} would like to see the new code of module Z, Bob decides that he can ‘‘give up’’ that module since he has finished his changes to it. So he splits T_{Bob} into T_{Bob} and T_{BobA}, and then commits T_{BobA}, thus committing his changes to Z while retaining X. Bob should do this only if the changes to Z do not depend in any way on the previous or planned changes to X, which might later be aborted. Alice can now read Z and use it for testing her code, and eventually commits T_{Alice}, releasing Z as well as Y. This schedule is shown in figure 5-1.

`Join-transaction` performs the reverse operation of `split-transaction`, merging the ongoing work of two or more independent transactions as if these transactions had always been a single transaction. A `split-transaction` followed by a `join-transaction` in one of the newly separated transactions can be used to transfer resources among particular in-progress transactions without even temporarily making the resources available to other transactions. The above example could therefore be modified to transfer Z from Bob to Alice without committing it, e.g., if it is not in a final (i.e., self-consistent) state but instead Alice intends to take over the work of editing it.

5.2. Group Transactions

Most extended transaction models intended to encompass synergistic cooperation, i.e., concerted effort among multiple concurrent transactions towards a common goal, are based on some notion of a *group*. Transactions within a group can cooperate in ways not permitted for transactions outside the group, or among groups. Perhaps it is not coincidence that most group-based approaches also extend the lock modes available far beyond the conventional read and write.

5.2.1. Group-Oriented Model

The *group-oriented model* [Klahold *et al.* 85] categorizes long transactions into *group transactions* (GTs) and *user transactions* (UTs). Every UT is a subtransaction of some GT. The model provides primitives to define groups of users, with the intention of associating each GT with a user group. A GT reserves objects from the public database into the corresponding group database, within which individual users create their own user databases and invoke UTs to reserve objects from the group database into their user database. The structure is similar to Smile, but there are two levels of experimental databases rather than just one; note that exactly two levels are permitted, not the arbitrary hierarchies of Infuse and NSE (see section 3).

Groups are isolated from each other, i.e., one user group cannot see the work of another user group until the relevant group database has been deposited into the public repository. GTs are thus serializable with respect to each other. Within a group transaction, multiple UTs can run concurrently and are serializable unless users intervene to cooperate in a non-serializable schedule. The basic mechanism provided for relaxing serializability is a version facility that supports parallel development (branching) and notification. Versions can be derived, deleted and modified by a user 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 to either read the same version or derive a new (modifiable) version from it; (3) shared derivation, which allows the owner of the lock to both read the version and derive a new version, while allowing parallel reads of the same version and derivation of different new versions by other users; (4) exclusive derivation, which allows the owner 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 operations by other users on that version.

Users can access objects only as part of a (long) transaction. Each transaction is two-phase, consisting of an acquire phase and a release phase, as in the conventional transaction model. Locks can be

strengthened (i.e., converted into a more exclusive mode) only during the acquire phase, and weakened (converted into a more flexible lock) only during the release phase. If a transaction requests a lock on a particular object and the object is already locked with an incompatible lock by another transaction, the request is rejected and the initiator of the requesting transaction is informed of the rejection. This avoids the problem of deadlock, which would be caused by blocking transactions that request unavailable resources. Instead, the user is notified later when the object becomes available.

The group-oriented transaction model also provides a special read operation that breaks any lock by allowing a user to read a version, knowing that it might soon be changed. This gives the user the ability to observe the progress of a task without impinging on the task's progress. In the scenario of our motivating example, Alice might take advantage of this facility to read Bob's (changing) version of the library module Z as needed for her compilation and testing of Y. She would realize that Z will continue to change and may not even be internally consistent (e.g., it might not compile successfully), but perhaps would prefer the recent snapshot of Z to the old version as it was prior to when Bob began his task.

5.2.2. Transaction Groups

The ObServer database system [Hornick and Zdonik 87] replaces classical locks with a rich set of lock modes and communication modes that can be paired, in principle, to support an implementation framework for cooperative transactions. The lock modes indicate whether the transaction intends to read or write the object and whether it is willing to read while another transaction writes, write while other transactions read, or accept multiple writers of the same object. The communication modes specify whether the transaction wants to be notified if another transaction requests a specific lock on the object or if another transaction has updated the object.

A *transaction group* [Fernandez and Zdonik 89] is a process that controls database access by a set of cooperating transactions and transaction (sub)groups (collectively members of the transaction group). Within each transaction group, member transactions and subgroups are synchronized according to an *input protocol* that defines some semantic correctness criteria appropriate for the application. The criteria are specified by semantic patterns, and enforced by a recognizer and a conflict detector. The recognizer ensures that a lock request from a member transaction matches an element in the set of locks that the group may grant to its members. The conflict detector ensures that a request to lock an object in a certain mode does not conflict with the locks already held on the object.

If a transaction group member requests an object that is not currently locked by its group, the group must request a lock on the object from its parent (if any, otherwise the database itself). The input protocol of the parent group might be different from that of the child group, as in the constraints of the cooperating CAD transactions model. In that case, the child group must transform its requested lock mode into a different mode accepted by the parent's input protocol. The transformation is carried out by an *output protocol*, which consults a lock translation table to determine how to transform a lock request into one that is acceptable to the parent group.

To illustrate, consider the example depicted in figure 5-2. Bob and Alice are together assigned the task of updating modules X, Y and Z, while Charlie is responsible for updating the documentation of the project. Alice and Bob need to cooperate while working on their modules (recall the sharing of Z from previous sections) whereas Charlie only needs to access the final result of the modifications in order to verify that

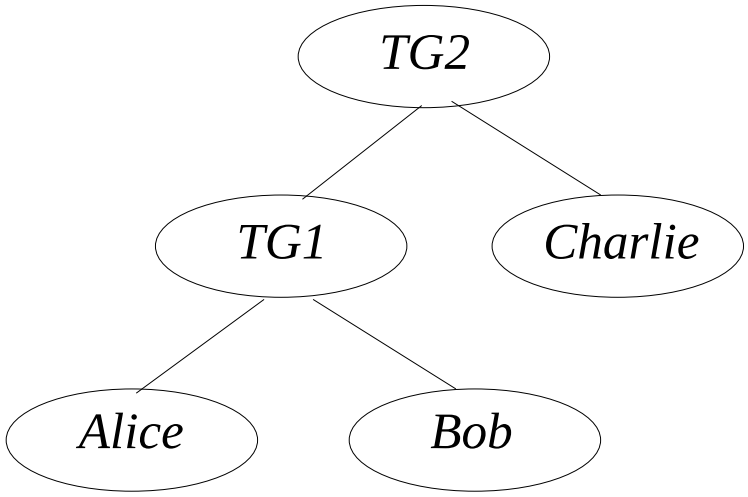

Figure 5-2: Transaction Groups

the modules match the documentation. Two transaction groups are defined, TG1 and TG2. TG2 has $T_{Charlie}$ and TG1 as its members, and TG1 includes T_{Bob} and T_{Alice}. The output protocol of TG1 states that changes made by the transactions within TG1 are committed with respect to TG2 only when all the transactions of TG1 have either committed or aborted. The input protocol of TG1 accepts lock modes that allow T_{Alice} and T_{Bob} to cooperate (e.g., see each others' partial results) while isolation is maintained within TG2 (to prevent $T_{Charlie}$ from accessing the partial results of the transactions in TG1).

5.2.3. Participant Transactions

The transaction groups model determines cooperation opportunities in terms of non-standard lock modes and how the remaining lock conflicts are handled. A related approach is to define each (long) transaction as a *participant* in a specified *domain*, where participant transactions in the same domain need not appear to have been performed in some serial order with respect to each other [Kaiser 90]. (Participation domains should not be confused the domains of domain relative addressing, section 3.1.2.). A domain would typically represent the set of transactions controlled by the users collaborating on a common task. However, unlike transaction groups, there is no implication that all the transactions in a domain commit together, or even that all of them commit (some may abort) [Kaiser 91]. Those transactions that are not participants in a particular domain are considered *observers* with respect to that domain. Observer transactions must be serialized with respect to the transactions in the domain, and furthermore should not view any anomalous behavior due to the interactions among participants.

Say a domain D is defined to respond to the modification request of section 2, and Alice and Bob start long transactions T_{Alice} and T_{Bob} that participate in D. The schedule shown in Figure 5-3 is not serializable according to any of the conventional concurrency control mechanisms. T_{Alice} reads the updates T_{Bob} made to module Z, which are written but are not yet committed by T_{Bob}, modifies parts of module Y, and then commits. T_{Bob} reads the changed Y (e.g., to test his own code in the context of the entire program) and continues to modify X after T_{Alice} has committed. Since T_{Alice} and T_{Bob} participate in the same domain D, the schedule is legal according to the participant transactions model.

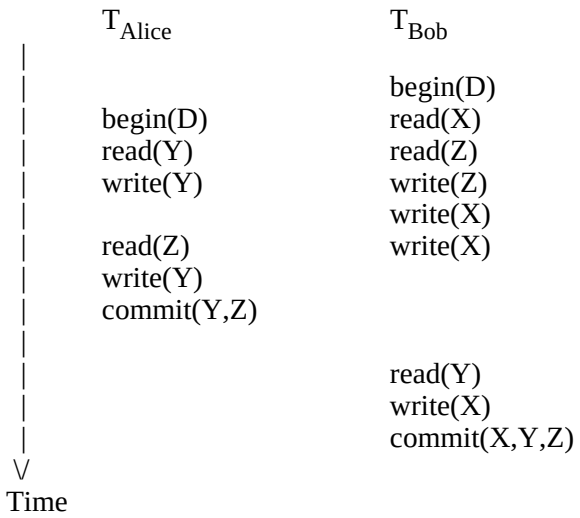

Figure 5-3: Participant Transaction Schedule

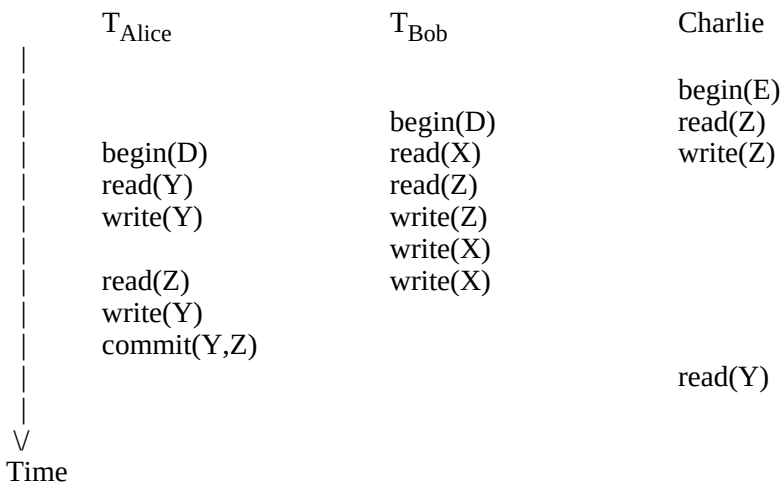

Figure 5-4: Participation Conflict

Now say that our third user, Charlie, starts a long transaction $T_{Charlie}$ in domain E, so it is an observer for domain D. Assume the sequence of events shown in figure 5-4, where Charlie does not work on documentation, but instead modifies library module Z. The transaction schedule is legal to this point, since $T_{Charlie}$ thus far could be serialized before T_{Bob} (but not after, since Bob reads Charlie's updates to Z). But then $T_{Charlie}$ attempts to read module Y, which has been modified and committed by T_{Alice}. This would be illegal even though T_{Alice} has already committed! The reason is that T_{Alice} cannot be serialized before $T_{Charlie}$, and thus before T_{Bob}, because T_{Alice} reads the uncommitted changes to module Z written by T_{Bob}. In fact, T_{Alice} cannot be serialized either before or after T_{Bob}. This would not be a problem if it was unnecessary to serialize T_{Alice} with any transactions outside the shared domain D; the serializability

of transactions within a participation domain need be enforced only with respect to what can actually be observed by transactions that are not participants in the domain.

6. Research Directions

Mechanism	System	Consistency	Long	Interactive	Cooperative
Classical Transactions	numerous	Yes	No	No	No
Checkout	RCS	No	Yes	Yes	No
Distributed Checkout	Itasca	Possible	Yes	Yes	No
Domain Relative Addressing	Cosmos	Minimal	Yes	Limited	No
Single-level Consistency	Smile	Yes	Yes	Yes	No
Multi-level Consistency	Infuse	Yes	Yes	Yes	Limited
Copy/Modify/ Merge	NSE	No	Yes	Yes	No
Backout and Commit Spheres	N/A	Possible	Yes	Yes	Unclear
Notification	Gordion	No	Yes	Yes	Limited
Dynamic Restructuring	in progress	Yes	Yes	Yes	Limited
Group-Oriented Trans.	unknown	Minimal	Yes	Yes	Limited
Transaction Groups	ObServer	Possible	Yes	Limited	Yes
Participant Trans.	in progress	Minimal	Yes	Yes	Yes

Figure 6-1: Proposed Concurrency Control Approaches

We saw in section 4 a formalism that intends to model the class of nested transaction models. Since there are several cooperative transaction models but no consensus on which one is best, or even that there is one approach that is universally superior for all environment applications, it would be desirable to develop an analogous formalism for cooperation. Such a formalism should provide a relatively small number of primitives that could be combined in different ways to specify the characteristics of a wide range of extended transaction models. Further, the formalism should be *executable* in the sense that it would be possible to prototype constructed models, albeit not necessarily in the most efficient manner. For example, [Salem 93] proposed a toolkit that seems feasible for implementing a range of extended transaction models, but concerned only with long duration and to some extent interactive control, without the possibility of cooperation.

The ACTA framework [Chrysanthis and Ramamritham 90] meets the first criterion, but was never intended to be executable. [Skarra 91] proposed a formalism, as a sample input/output protocol for

transaction groups, for recognizing required and prohibited patterns of transaction interactions using finite state automata. She concedes, though, that the implementation is unwieldy and unlikely to scale up. [Barghouti 92] suggested condition/action rules for recognizing and resolving transaction conflicts. A proof-of-concept implementation was included in the Marvel 3.1 software development environment, which also supports user-defined lock modes [Ben-Shaul *et al.* 93]. [Heineman 93] is currently developing a formalism which provides a superset of the repair primitives used in the actions to support (at least) dynamic restructuring and participant transactions; it remains to be seen whether this approach will be practical. Thus cooperation modeling remains an important open problem for future research.

Acknowledgments

The author is supported by grants from the National Science Foundation, Andersen Consulting, AT&T Foundation, Bull HN Information Systems and IBM Canada Ltd, and by the New York State Center for Advanced Technology in Computers and Information Systems. This chapter is based in part on material gathered by Naser Barghouti for [Barghouti and Kaiser 91].

References

[Adams *et al.* 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. IEEE Computer Society Press, Pittsburgh PA, May, 1989.

[Barghouti 92] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February, 1992.

[Barghouti and Kaiser 91]
Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys* 23(3):269-317, September, 1991.

[Ben-Shaul *et al.* 93]
Israel Z. Ben-Shaul, Gail E. Kaiser and George T. Heineman. An Architecture for Multi-User Software Development Environments. *Computing Systems, The Journal of the USENIX Association* 6(2):65-103, Spring, 1993.

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

[Chrysanthis and Ramamritham 90]
Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In Hector Garcia-Molina and H.V. Jagadish (editors), *1990 ACM SIGMOD International Conference on Management of Data*, pages 194-203. Atlantic City NJ, May, 1990. Special issue of *SIGMOD Record*, 19(2), June 1990.

[Davies 73] Charles T. Davies, Jr. Recovery Semantics for A DB/DC System. In *28th ACM National Conference*, pages 136-141. Atlanta GA, August, 1973.

[Davies 78] C.T. Davies, Jr. Data processing spheres of control. *IBM Systems Journal* 17(2):179-198, 1978.

[Ege and Ellis 87] Aral Ege and Clarence A. Ellis. Design and implementation of Gordion, an Object Base Management System. In *3rd International Conference on Data Engineering*, pages 226-234. Los Angeles CA, February, 1987.

[Estublier *et al.* 84]
J. Estublier, S. Ghoul and S. Krakowiak. Preliminary Experience with a Configuration Control System for Modular Programs. In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 149-156. Pittsburgh PA, April, 1984. Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[Eswaran *et al.* 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.

[Fernandez and Zdonik 89]
Mary F. Fernandez and Stanley B. Zdonik. Transaction Groups: A Model for Controlling Cooperative Work. In *3rd International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128-138. Queensland, Australia, January, 1989.

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

[Gray *et al.* 75] J. Gray, R. Lorie and G. Putzolu. Granularity of Locks and Degrees of Consistency in a Shared Database. In *International Conference on Very Large Data Bases*, pages 428-451. Morgan Kaufmann, 1975.

[Heineman 93] George T. Heineman. *A Transaction Manager Component for Cooperative Transaction Models*. Technical Report CUCS-017-93, Columbia University Department of Computer Science, July, 1993.

[Honda 88] M. Honda. Support for Parallel Development in the Sun Network Software Environment. In *2nd International Workshop on Computer-Aided Software Engineering*, pages 5-5 - 5-7. 1988.

[Hornick and Zdonik 87]
Mark F. Hornick and Stanley B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Automation Systems* 5(1):70-95, January, 1987.

[Horwitz *et al.* 89]Susan Horwitz, Jan Prins and Thomas Reps. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems* 11(3):345-387, July, 1989.

[Johnson 78] S.C. Johnson. Lint, a C Program Checker. *Unix Programmer's Manual.* AT&T Bell Laboratories, 1978.

[Kaiser 90] Gail E. Kaiser. A Flexible Transaction Model for Software Engineering. In *6th International Conference on Data Engineering*, pages 560-567. IEEE Computer Society Press, Los Angeles CA, February, 1990.

[Kaiser 91] Gail E. Kaiser. Interfacing Cooperative Transactions to Software Development Environments. *Office Knowledge Engineering* 4(1):56-78, February, 1991.

[Kaiser *et al.* 89] Gail E. Kaiser, Dewayne E. Perry and William M. Schell. Infuse: Fusing Integration Test Management with Change Management. In *COMPSAC 89 The 13th Annual International Computer Software & Applications Conference*, pages 552-558. IEEE Computer Society Press, Orlando FL, September, 1989.

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

[Kaiser and Perry 87]
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. IEEE Computer Society Press, Austin TX, September, 1987.

[Kaiser and Perry 91]
Gail E. Kaiser and Dewayne E. Perry. Making Progress in Cooperative Transaction Models. *Data Engineering* 14(1):19-23, March, 1991.

[Kaiser and Pu 92]
Gail E. Kaiser and Calton Pu. Dynamic Restructuring of Transactions. In Ahmed K. Elmagarmid (editor), *Database Transaction Models for Advanced Applications*, pages 265-295. Morgan Kaufmann, San Mateo CA, 1992.

[Katz 90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys* 22(4):375-408, December, 1990.

[Kim *et al.* 91] Won Kim, Nat Ballou, Jorge F. Garz and Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems* 9(1):31-51, January, 1991.

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

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

[Leblang and Chase 87]
David B. Leblang and Robert P. Chase, Jr. Parallel Software Configuration Management in a Network Environment. *IEEE Software* 4(6):28-35, November, 1987.

[Moss 82] J. Eliot B. Moss. Nested Transactions and Reliable Distributed Computing. In *2nd Symposium on Reliability in Distributed Software and Database Systems*, pages 33-39. IEEE Computer Society Press, Pittsburgh PA, July, 1982.

[Perry and Kaiser 91]
Dewayne E. Perry and Gail E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering* 17(3):283-295, March, 1991.

[Pu *et al.* 88] Calton Pu, Gail E. Kaiser and Norman Hutchinson. Split-Transactions for Open-Ended Activities. In Francois Bancilhon and David J. Dewitt (editor), *14th International Conference on Very Large Data Bases*, pages 26-37. Morgan Kaufman, Los Angeles CA, August, 1988.

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

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

[Salem 93] Kenneth Salem. *Impleemnting Extended Transaction Models Using Transaction Groups*. Technical Report CS-TR-3051, University of Maryland Department of Computer Science, 1993.

[Schwanke *et al.* 89]
R.W. Schwanke, E.S. Cohen, R. Gluecker, W.M. Hasling, D.A. Soni and M.E. Wagner. Configuration Management in BiiN SMS. In *11th International Conference on Software Engineering*, pages 383-393. IEEE Computer Society Press, Pittsburgh PA, May, 1989.

[Skarra 91] Andrea H. Skarra. Localized Correctness Specifications for Cooperating Transactions in an Object-Oriented Database. *Office Knowledge Engineering* 4(1):79-106, February, 1991.

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

[Tichy 86] Walter F. Tichy. Smart Recompilation. *ACM Transactions on Programming Languages and Systems* 8(3):273-291, July, 1986.

[Walpole *et al.* 88a]
J. Walpole, G.S. Blair, J. Malik and J.R. Nicol. A Unifying Model for Consistent Distributed Software Development Environments. In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 183-190. Boston MA, November, 1988. Special issue of *Software Engineering Notes*, 13(5), November 1988.

[Walpole *et al.* 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. IEEE Computer Society Press, San Jose CA, June, 1988.

[Walter 84] B. Walter. Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications. In Umeshwar Dayal, G. Schlageter and Lim Huat Seng (editors), *10th International Conference on Very Large Data Bases*, pages 161-171. Morgan Kaufmann, Singapore, August, 1984.

[Yeh *et al.* 87] Show-way Yeh, Clarence Ellis, Aral Ege and Henry Korth. *Performance Analysis of Two Concurrency Control Schemas for Design Environments*. Technical Report STP-036-87, MCC, June, 1987 .