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
15 March 1993

Abstract

We survey 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 so that they cannot (or at least are not supposed 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.

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 next generation database applications such as software development and CAD/CAM environments, even though for these applications we 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 require the preregistration of database accesses needed to plan transaction schedules *a priori*, and automatic ‘‘redo’’ of activities is often infeasible.
- *Cooperation among concurrent 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 so-called ‘‘checkout’’ model, in which users copy objects from the shared repository to private workspaces 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 of checkout systems are tempted to communicate through subversive operations outside system control. This chapter surveys proposed approaches to all three problems, with particular emphasis on supporting cooperation. We are primarily concerned with approaches to 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 we would like to note that many of their ideas were foreshadowed in the 1970’s [7, 8].

We assume the reader is familiar with the conventional atomic transaction model [5] and its main implementation mechanisms, such as two-phase locking [13], multi-version timestamp ordering [37], optimistic validation [29], multi-granularity locking [15], and nested transactions [32]. We start with a 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. 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. The chapter ends with a table summarizing all the models surveyed, as well as some others omitted from discussion due to lack of space.

2. Motivating Example

Two programmers, Alice and Bob, are working on the same software project. The program consists 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 a traditional concurrency control scheme based on two-phase locking was used, Bob and Alice would not be able to access module Z at the same time. They would be allowed to concurrently lock modules Y and X, respectively, since they work in isolation on these modules. These users need to work *cooperatively* on module Z, however, 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 a number of obvious ‘‘work-arounds’’. But none of them 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 these approaches are typically used in practice, although they do not encompass cooperation in the sense of the following section.

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 [34].

3.1. Version and Configuration Management

3.1.1. Checkout Model

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 any time. The *checkout/checkin* approach (also called reserve/deposit) has been implemented by widely-used version control tools like SCCS [38] and RCS [41]. 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 explicitly checking out (reserving) the object. The reservation makes a copy of the indicated version and gives the owner of the reservation exclusive access to the copy so that he 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 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 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 [18]).

Considering our example, Bob would checkout module X and Alice would checkout module Y; 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 in read-only mode the most recently checked in version of Z and simply release her reservation when done with her task, 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 — 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 forgets 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.

ORION-2 [27] solves the second problem, via a distributed database incorporating private databases into which objects are checked out from the shared database and a sophisticated query facility that distinguishes between private and shared databases. There is no access to objects outside of system control. However, ORION-2'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. Versions of 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*. Configurations are supported by numerous version control and configuration management tools for software development (e.g., Adele [12], DSEE [30], SMS [39]). A survey by Katz [26] gives a comprehensive overview of version and configuration systems oriented towards CAD/CAM environments.

Conversational transactions [31] support a shared database/private database structure similar to

ORION-2. However, the conversational transactions model treats all checkout/checkin operations on the shared database and all manipulation operations within a private database as actions within a long transaction, and require that every object reserved as part of that transaction be deposited before it commits. Thus all the new versions created during the same long transaction implicitly constitute a new configuration. However, knowledge of the configuration is not retained by the shared database once all of its components have been deposited.

Walpole *et al.* introduced a concurrency control concept called *domain relative addressing* [42], which supports versions of configurations. Domain relative addressing extends Reed's notion of time relative addressing (multi-version concurrency control) [37]. 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 transaction reads only those versions in its input domain and writes new versions generating an output domain [43]. Even when multiple transactions employ the same input domains concurrently, they produce distinct output domains.

```
        T_Alice         T_Bob
  |
  |                     read(X)
  |     read(Y)         read(Z)
  |     write(Y)        write(Z)
  |                     write(X)
  |     read(Z)         write(X)
  |     write(Y)
  V
 Time
```

Figure 3-1: Domain Relative Addressing Schedule

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

3.2.1. Enforcing Consistency

Smile [19] is a multi-user software development environment for C programming. As in ORION-2, Smile maintains private workspaces 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 conversation transactions, Smile requires the entire contents of a private workspace 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 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 private workspace (called an experimental database) at any time. When the user requests checkin of his experimental database, Smile analyzes all its modules using the lint tool, 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 aborted. The user is expected to repair the problems and try again.

Considering our example, Bob starts a 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 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 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 the baseline modules 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 another user 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 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. 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 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. Adding Groups

The *group-oriented model* [28] categorizes 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.

Groups are isolated from each other, i.e., one user group cannot see the work of another user group until the contents of the relevant group database has been deposited into the public repository. GTs are thus serializable with 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 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 (typically the manager of a project) the ability to observe the progress of the project without impinging on that progress.

3.2.3. Multi-Level Consistency

Infuse is another multi-user environment for C programming. As in Smile, copies of modules are checked out into experimental databases representing work areas. Like the group-oriented model, Infuse's experimental databases are not necessarily private to a single user. However, 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 the 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 individual users. The top-level experimental database corresponds roughly to a top-level nested transaction.

Infuse [21] 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 system modules from the baseline (or reserved elsewhere in the hierarchy), but their functionality is sufficient only for unit and integration 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 the example of the previous section, 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* [23], against any subset of the experimental databases in the hierarchy. This permits a manager to keep tabs on the progress of the work, as in the group-oriented model. Infuse also provides a facility whereby users can define *workspaces* [20] 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 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 in this section 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, Sun's Network Software Environment (NSE) [16] supports a nested model that operates on a hierarchical database structure. Each node of this structure is called an *environment* (not to be confused with a software development 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. Sun refers to this approach as *copy/modify/merge* [1].

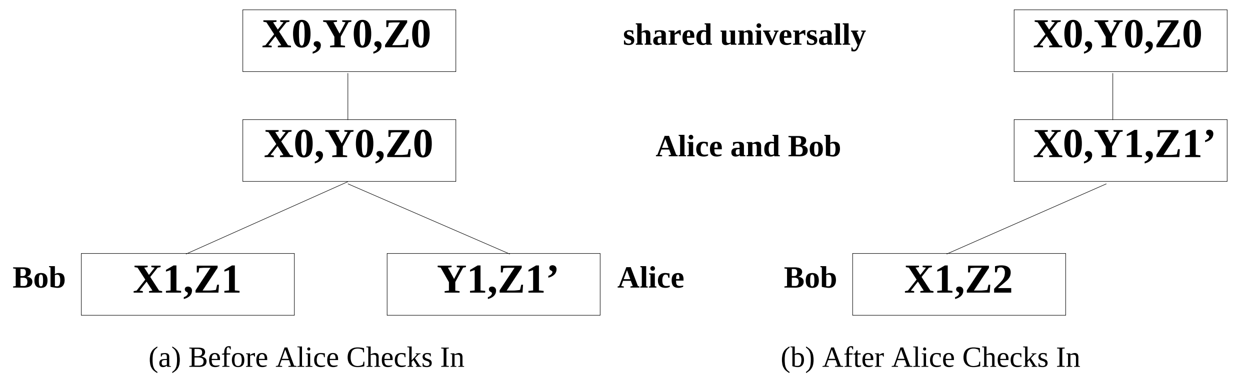

Figure 3-2: NSE Environments

Illustrating this approach, either Bob or Alice creates a top-level environment $ENV_{X,Y}$ to fix the files X and Y. Bob then creates his own child environment ENV_X to edit X. Z is added later as above, 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, we are just following the original 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 check 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 is illustrated in figure 3-2.

4. Modeling Coordination

Conversational transactions, domain relative addressing, Smile, the group-oriented model and Infuse all support an explicit atomic transaction model in the sense that a 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, ORION-2 and NSE all allow individual objects to be reserved and deposited independently, with no formal notion of maintaining consistency among multiple objects. Walter proposes a middle ground with the basic structure of nested transactions but a more flexible relationship between parent and child transactions [44].

In his model, the interface between a parent transaction and a child transaction can either be single-request, i.e., the parent in effect issues a query or update to the child and waits until the child returns the full result atomically, or *conversational*, i.e., the control alternates between the parent that generates a series of requests and the child that answers these requests individually. (Walter's conversational interface should not be confused with the conversational transactions of section 3.1.2.) 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. We could consider the basic checkout model to support a multiple-request interface with multiple outstanding requests, each request concerns only a single object, and a child transaction multi-tasks among computations towards answering these 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 a 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. Moss's classical nested transaction model [33], 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 systems of 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 [34]. The mechanisms described in the previous section 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 the previous section with other approaches concentrating on coordination.)

5.1. Cooperation Primitives

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 [10] provides a notification primitive that can be used in conjunction with other primitives (such as non-standard lock modes) 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* [45]. 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 transactions attempts to commit. Conflicts are resolved by instigating a ‘‘phone call’’ between the two parties with the assumption that they can interact directly to resolve the conflict.

These policies incorporate humans as part of the conflict resolution algorithm. On the one hand, this enhances concurrency in environment applications where many of the tasks are interactive. But on the other hand, it can also degrade consistency because humans cannot always be relied on to correctly resolve conflicts. It might be preferable for the environment to support 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 [25]. 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 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 [35]. 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*.

Time	T_{Alice}	T_{Bob}	T_{BobA}
|		begin	
|	begin	read(X)	
|	read(Y)	read(Z)	
|	write Y	write(Z)	
|		write(X)	
|	request read(Z)	corresponding notify(Z)	
|		split((Z),(X))	
|		commit(Z)	
|	actual read(Z)		write(X)
|	write(Y)		
|			commit(X)
|	commit(Y,Z)		
\/			

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 (note this may not be realistic, depending on the details of the scenario). 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 really in a final state but instead Alice intends to take over the work of editing it.

5.2. Group Transactions

Some of the mechanisms in this chapter were designed for CAD environments and others for software development environments. The differences among the corresponding extended transaction models reflect to some extent the differences in the structure of typical projects in the two application areas. It seems that CAD projects are more strictly organized than software development projects, with a more stringent division of tasks and with less sharing among tasks. CAD staff appear to be more content with the relative isolation of the checkout model and its extensions, while programmers — even those working on different projects — often demand shared access to common libraries and frequently view each others' partial results outside the checkout model.

5.2.1. Cooperating CAD Transactions

The *cooperating CAD transactions* model [2] is an exception. All the users involved in the same project participate in a single cooperating transaction, which represents the composite of all long transactions controlled by those users. Each of these long transactions is in turn made up of a sequence of conventional short-duration transactions, which are serialized normally. Thus, if a user invokes a short transaction (within his long transaction) that conflicts with another user's short transaction, one of them has to wait only for the duration of the short transaction. Cooperation is thus permitted through interleavings among the long transactions at the granularity of the short transactions.

Each cooperating transaction encapsulates a complete project. However, some of the tasks within a project can be ‘‘subcontracted’’ to another group instead of being carried out by members of the project. In this case, a special cooperating transaction called a *client/subcontractor* transaction is initiated. Each client/subcontractor transaction can invoke other client/subcontractor transactions leading to a hierarchy of such transactions. This structure is similar to Infuse's hierarchy of experimental databases and NSE's hierarchy of environments (section 3).

A cooperating transaction is a nested transaction that preserves certain consistency constraints specified as part of the transaction. Each subtransaction (itself a cooperating transaction) in turn preserves some constraints, not necessarily the same ones as its parent transaction. The only requirement is that subtransactions have weaker constraints than their ancestors but not vice versa. Thus, the consistency constraints defined at the top-level of a cooperating transaction imply all the constraints defined at lower levels.

To replace the conventional concept of a serializable schedule for a nested transaction, the cooperating CAD transactions model defines an *execution* of a cooperating transaction to be a total order of all the low-level database actions invoked within the composite cooperating transaction that is compatible with the partial orders imposed by the different levels of nested subtransactions. A *protocol* is a set of rules that restrict the set of admissible executions. If the set of rules was strict enough, it would permit only serializable executions and require semantic consistency enforcement as in Smile and Infuse. The rules can, however, allow arbitrary non-serializable executions as long as they fulfill the constraints.

5.2.2. Transaction Groups

The ObServer database system [17] replaces classical locks with a rich set of lock modes and communication modes that can be paired 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* [14] 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 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 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).

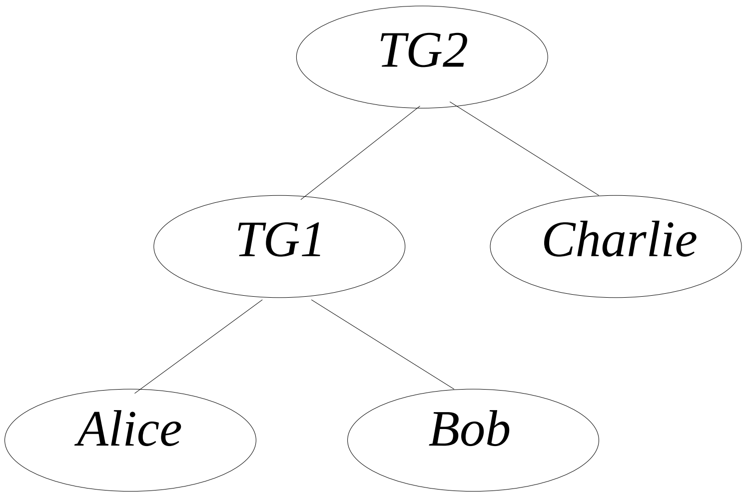

Figure 5-2: Transaction Groups

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 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 [22]. (Participation domains should not be confused with domain relative addressing.). 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) [24]. Those transactions that are not participants in a particular domain are considered *observers* with respect to that domain. The observer transactions must be serialized normally with respect to the transactions in the domain, and furthermore should not view any anomalous behavior due to the interactions among the participants.

Say a domain D is defined to respond to the modification request of section 2, and Alice and Bob start 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 that 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.

Now say that another user, Charlie, starts a transaction $T_{Charlie}$ in domain E, so it is an observer for domain D. Assume the sequence of events shown in figure 5-4. Charlie first modifies module Z. This by itself would be legal, 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

	T_{Alice}	T_{Bob}
		begin(D)
	begin(D)	read(X)
	read(Y)	read(Z)
	write(Y)	write(Z)
		write(X)
	read(Z)	write(X)
	write(Y)	
	commit(Y,Z)	
		read(Y)
		write(X)
		commit(X,Y,Z)
Time		

Figure 5-3: Participant Transaction Schedule

	T_{Alice}	T_{Bob}	Charlie
			begin(E)
		begin(D)	read(Z)
	begin(D)	read(X)	write(Z)
	read(Y)	read(Z)	
	write(Y)	write(Z)	
		write(X)	
	read(Z)	write(X)	
	write(Y)		
	commit(Y,Z)		
			read(Y)
Time			

Figure 5-4: Participation Conflict

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; 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. Summary

Mechanism	Reference	System	Consistency	Long	Interactive	Cooperative
Checkout	[41]	RCS	No	Yes	Yes	No
Distributed Checkout	[27]	ORION-2	No	Yes	Yes	No
Conversational Trans.	[31]	System R	Minimal	Yes	Yes	No
Domain Relative Addressing	[42]	Cosmos	Minimal	Yes	Limited	No
Single-level Consistency	[19]	Smile	Yes	Yes	Yes	No
Group-Oriented Trans.	[28]	N/A	Minimal	Yes	Yes	Limited
Multi-level Consistency	[21]	Infuse	Yes	Yes	Yes	Limited
Copy/Modify/ Merge	[1]	NSE	No	Yes	Yes	No
Backout and Commit Spheres	[44]	N/A	Possibly	Yes	Yes	Limited
Notification	[45]	Gordion	No	No	Yes	Limited
Dynamic Restructuring	[25]	N/A	Potentially	Yes	Yes	Limited
Epsilon Serializability	[36]	N/A	Yes	Possibly	No	Limited
Group Paradigm	[11]	N/A	Minimal	Yes	No	Limited
Cooperating CAD Trans.	[2]	N/A	Yes	Yes	Yes	Yes
Transaction Groups	[14]	ObServer	Yes	Yes	Limited	Yes
Visibility Domains	[9]	N/A	Potentially	Yes	Limited	Yes
Participant Trans.	[22]	N/A	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 and, in principle, could be used to implement diverse coordination mechanisms for experimentation purposes. 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.

The ACTA framework [6] meets the first criterion, but was never intended to be executable. Skarra [40] proposed a formalism, in the context of transaction groups, for recognizing required and prohibited patterns of transaction interactions using finite state. She concedes, though, that the implementation is unwieldy and unlikely to scale up. Barghouti [3] suggested rules for resolving transaction conflicts, with repair primitives intended to support (at least) transaction restructuring and participation domains. But his language is specific to the process modeling details of the intended target environment, Marvel [4], and development is still in progress. Thus cooperation modeling seems an important open problem for future research.

Acknowledgments

The author is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants from AT&T, BNR, Bull, DEC, IBM, Paramax and SRA, and by the New York State Center for Advanced Technology in Computers and Information Systems. This chapter is based in part on material gathered for Naser S. Barghouti and Gail E. Kaiser, Concurrency Control in Advanced Database Applications, *ACM Computing Surveys*, 23(3):269-317, September 1991.

References

[1] 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.

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

[3] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February, 1992. CUCS-001-92.

[4] Israel Z. Ben-Shaul, Gail E. Kaiser and George T. Heineman. An Architecture for Multi-User Software Development Environments. In Herbert Weber (editor), *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149-158. Tyson's Corner VA, December, 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] Amr El Abbadi and Sam Toueg. The Group Paradigm for Concurrency Control Protocols. *IEEE Transactions on Knowledge and Data Engineering* 1(3):376-386, September, 1989.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] Raymond Lorie and Wilfred Plouffe. Complex Objects and Their Use in Design Transactions. In *Engineering Design Applications, Annual Meeting Database Week*, pages 115-121. IEEE Computer Society Press, San Jose CA, May, 1983.

[32] 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.

[33] J. Eliot B. Moss. *Information Systems*: *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge MA, 1985. PhD Thesis, MIT LCS TR-260, April 1981.

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

[35] 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.

[36] Calton Pu, Wenwey Hseush, Gail E. Kaiser, Kun-Lung Wu and Philip S. Yu. Distributed Divergence Control for Epsilon Serializability. In *13th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Pittsburgh PA, May, 1993. In press.

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

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

[39] 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.

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

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

[42] 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 *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[43] 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.

[44] 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.

[45] 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 .