Dynamic Restructuring of Transactions

Gail E. Kaiser and *Calton Pu*
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
kaiser@cs.columbia.edu, 212-854-3856
calton@cs.columbia.edu, 212-854-8110
Fax: 212-666-0140

August 21, 1991

Abstract

Open-ended activities are characterized by uncertain duration, unpredictable developments, and interactions with other concurrent activities. Like other database applications, they require consistent concurrent access and fault-tolerance, but their unconventional characteristics are incompatible with the conventional database mechanisms of concurrency and failure atomicity. We present the split-transaction and join-transaction operations for *restructuring* in-progress transactions, as an approach to consistent concurrent access and fault-tolerance for open-ended activities. Split-transaction divides an on-going transaction into two or more transactions that are serializable with respect to each other and all other transactions, and each of the new transactions is later committed or aborted independently of the others. Join-transaction merges two or more transactions that are serializable with respect to each other into a single transaction as if they had always been part of the same transaction, and all their work is now committed or aborted together. Split-transaction is useful for committing some work early or dividing on-going work among several co-workers. Join-transaction allows to hand over results to a co-worker to integrate into his or her own ongoing task. The transaction manager enforces that the new transaction(s) will in fact be serializable, and does not permit the split or join otherwise.

1 Introduction

Database transactions provide atomicity in two senses: concurrency atomicity, for consistent concurrent access, and failure atomicity, for fault-tolerance. Unfortunately, these atomicity properties restrict the applicability of the transaction concept in *open-ended activities*, such as those that arise in design environments and office automation, even though for these activities we still want consistent concurrent access and fault-tolerance.

Open-ended activities are characterized by:

- *Uncertain duration* – from hours to months;
- *Unpredictable developments* – actions unforeseeable at the beginning; and
- *Interaction with other concurrent activities* – for concerted efforts among database users.

Each one of these characteristics introduces difficulties when constrained by traditional transaction atomicity properties.

Uncertain duration results in *long transactions*[1] with two specific problems. First, work done within a long transaction is vulnerable to crashes because of transaction rollback during recovery. Second, long transactions become concurrency bottlenecks if resources are retained for the duration of the transaction; if resources are released early, this introduces the possibility of cascaded aborts, which are particularly problematic for long transactions. Interaction with other concurrent activities is even more difficult, since resource access from all other transactions must be serialized either before or after the transaction.

Unpredictable developments in open-ended activities reveal two characteristics intrinsic to traditional transactions. First, sometimes initial activities modify data early in the transaction, which will remain unchanged until commit. Second, some (or all) of the later activities may become independent of the previous activities. In both cases, it may be desirable for the results of the earlier activities to be made available to other concurrent activities before the end of the transaction, but without introducing the possibility of cascaded aborts. But interaction among ongoing concurrent activities is intentionally prohibited by the concurrency atomicity property of transactions.

To solve these difficulties, we introduce *transaction restructuring operations*, which dynamically modify the set of concurrent transactions while they are in progress. Our operations

[1]Informally defined as those transactions that last for at least the same time magnitude as the mean time between failures of the computer system on which they run.

preserve serializability among all committed transactions, although the committed set of transactions may not bear any simple relationship to the set of transactions that was initiated. Another way to say this is the concurrency atomicity is violated, but serializability (equivalence to some serial schedule) is not.

The *split-transaction* operation divides an ongoing transaction into two serializable transactions. In particular, resources read and written by the original transaction are divided among the two resulting transactions. Thereafter, each one of the new transactions may proceed independently with its own resources. A special case is to use split-transaction to immediately commit one of the new transactions, for early release of useful results from the original transaction. The other new transaction continues. In the general case, both new transactions continue, and may later commit or abort entirely independently of each other as if they had always been distinct, serializable transactions. The new transactions are thus **not** subtransactions of the original, but instead effectively replace the original transaction.

The inverse operation of split-transaction, called *join-transaction*, can combine results together and release them atomically. In particular, two ongoing serializable transactions are merged into one, as if they had always been a single transaction. Split-transaction and join-transaction can be combined in any formation, so direct transfer of resources from one transaction to another can be achieved by a split where one of the new transactions immediately joins some previously ongoing transaction.

Dynamic restructuring operations thus bring three major advantages to transaction-oriented open-ended activities:

1. Adaptive recovery, committing resources that will not change;
2. Added concurrency, releasing the committed resources or transferring ownership of uncommitted resources; and
3. Serializable access to resources by all activities.

We do not intend to address the debate of whether serializability is too restrictive for open-ended activities [32, 36]. Powerful serializable access is useful even in a system that admits non-serializable operations, and therefore we restrict our discussion to serializable access.

It can be argued that split-transaction and join-transaction do not truly preserve serializability, since the set of committed and aborted transactions may not be the same as the set of originally initiated transactions. Our claim is not that the originally initiated transactions are necessarily serializable, as this would be meaningless in our context where some of the originally initiated transactions are subsumed into other transactions. Instead, our transaction restructuring operations ensure that the operations finally attributed to the committed transactions result in a schedule equivalent to a serial schedule.

The rest of this paper is organized as follows. We first elaborate in section 2 on the requirements of open-ended activities. We then define split-transaction and join-transaction, first assuming conventional programmed transactions in section 3 and then extending in section 4 to our context of user-controlled transactions whose operations are selected by the users as they go along. We give several example scenarios in section 5 to demonstrate that our transaction restructuring operations fulfill the requirements of open-ended activities. We discuss some approaches to implementing dynamic restructuring operations in section 6. In section 7 we compare to other work on extended transaction models that address some or all of the requirements of open-ended activities, and then conclude by summarizing our contributions.

2 Requirements

Numerous researchers have studied the nature of concurrent behavior in open-ended activities, and have arrived at new requirements for consistent concurrent access. These requirements are based on three underlying characteristics of open-ended activities, which we term uncertain duration, unpredictable developments, and interaction with other concurrent activities. We discuss these characteristics and mention a few approaches to concurrency control here. In section 7 we compare our transaction restructuring approach to many of the other extended transaction models developed to fulfill some or all of these requirements.

Uncertain duration: Operations on data in design environments (such as compiling source code or circuit layout) are often long-lived. If these operations are embedded in transactions, these transactions, unlike traditional ones, will also be long-lived. Long transactions need different support than traditional short transactions. In particular, blocking a transaction until another commits is rarely acceptable for long transactions. The problem of long transactions has also been addressed in traditional data processing applications, bank audit transactions, for example, where most proposed solutions rely on *a priori* static analysis of the transaction code [30, 37].

Consider the following example scenario. Say that a programmer Bob edits and recompiles module F, and then works for several days on other modules. Another programmer Alice would like to test her own changes to module G, but has to wait until the Bob commits his work, in order to be able to read module F's object code to build the system executable. The length of Bob's transaction may effectively prevent Alice from carrying out productive work for an unacceptably long period of time.

Unpredictable developments: In order to support user-controlled tasks that are nondeterministic and interactive in nature, the transaction manager should provide the user with the ability to start a transaction, interactively execute operations within it, dynamically restructure it, if needed, and commit or abort it at any time. The nondeterministic nature of transactions implies that the transaction manager will not be able to determine whether or not the execution of a transaction might violate database consistency, except by actually executing it and enforcing consistency along the way. Thus mechanisms based on static analysis are not applicable. Optimistic concurrency control [26] also should not be used, since this might lead to situations in which the user might have invested many hours running a transaction, only to find out later when he wants to commit his work that some of the operations he performed within the transaction violated some consistency constraints. The user would definitely oppose deleting all of his work (by rolling back the transaction). He might, however, be able to perform some operations explicitly to restore consistency, such as manually merging multiple updates to data [1]. Thus, there is a need to support user control over transaction management as well as transaction execution.

For instance, suppose a programmer Charlie is developing two modules, F and G, to be released together in a transaction. Module F has been coded and tested before module G is finished. In this case, transaction abort due to a system failure will cause the loss of module F. Now consider the possibility that module F may have been modified so it does not use module G any more. In this case, module F remains unavailable unnecessarily until transaction commits. In both cases, even though the original plan might have been to commit F and G atomically, the developments lead Charlie to want to commit module F, while continuing with module G.

Interaction with other concurrent activities: Cooperation among users to develop project components has significant implications on consistent concurrent access. In multi-user design environments and office automation systems, several users might have to exchange data (i.e., share it collectively) in order to be able to carry out their work. The activities of two or more users working on shared data may not be serializable. They may pass the shared data items back and forth in a way that cannot be accomplished by a serial schedule. Also, two users might be modifying two parts of the same data item concurrently, with the intent of integrating these parts to create a new version of the item. In this case, they might need to look at each others' work to make sure that they are not modifying the two parts in a way that would make their integration difficult. This kind of sharing and exchanging data was termed *synergistic interaction* by Yeh *et al.* [44]. To insist on concurrency atomicity in multi-user environments might thus decrease concurrency or, more significantly, actually disallow desirable forms of cooperation among users.

The first example problem above could be solved by maintaining multiple versions of modules [23]. Then Alice could use the object code for an old version of F together with the new version of G in order to build the system executable. But this does not permit the situation where the two programmers Bob and Alice are participating in a concerted effort, where the new version of G is intended to use the new version of F, and in fact will not even compile together with any old versions of F. In this case, Alice is in fact forced to wait, even if Bob has completed all his changes to F and is working on other parts of the system. If the independence of F from Bob's other changes had been known when the work began, he could have initiated a separate transaction to contain only his work on F. But it may be that Bob's situation is like Charlie's, where F and his other modules became independent of each other only because of unforeseen modifications during the transaction. Thus there seems to be no way for Bob and Alice to interact and exchange partial results while their transactions are in progress.

The goal of our transaction restructuring model is to satisfy all three requirements for practical cases of open-ended activities.

3 Programmed Transactions

For ease of exposition, we introduce the split-transaction and join-transaction operations in the artificial context where transactions are programmed in advance and statements invoking the split-transaction and join-transactions operation appear directly in the code. This context is unrealistic, since if the need for restructuring transactions was known when the code was written, then the transaction code itself would have been restructured without the need for dynamic restructuring operations. (Perhaps dynamic restructuring might be a useful facility for conditional execution within programmed transactions, but we do not address this.) Then in the following section we present split-transaction and join-transaction in the intended context of open-ended activities with unpredictable developments, where it becomes known only while a set of transactions are in progress that they should be restructured in some manner. Although the possibility of restructuring was anticipated at the start, by definition, the specific restructuring needed could not be foreseen.

3.1 Definitions

The operations supported by the transaction manager are:

- Begin-Transaction,
- Commit-Transaction,
- Abort-Transaction,
- Read-Data,
- Write-Data,
- Split-Transaction,
- Join-Transaction.

Begin, Commit, Abort, Read and Write have their usual meaning. We do not distinguish between actually reading or writing a data item as opposed to reserving the item in some way, since any differences depends on the specific implementation mechanisms employed. These issues are postponed until section 6. In general, we assume all other operations not on this list that appear in the transaction code have the normal semantics of the underlying programming language and do not impact transaction management.

The Split operation on some transaction T produces two new transactions, A and B, and dissolves the original T. The syntax of split-transaction is as shown below. It is not necessary to mention T explicitly among the arguments, since split-transaction may only be executed within the body of some transaction – which is T, by definition.

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

`AReadSet`, `AWriteSet`, `BReadSet`, and `BWriteSet` are sets of data items accessed in A and B. `AProcedure` and `BProcedure` are the starting points of code at which A and B, respectively, will begin execution.

For simplicity of presentation, we describe only the two-way split-transaction. To split a transaction n ways, we can apply the two-way split $n-1$ times in succession. An n-way split-transaction operation may be seen as syntactic sugar for the successive two-way splits.

The Split operation divides all the operations completed in T into two subsets, A and B. The data sets associated with A (respectively, B) are the data items accessed by A (B). From the definition, the union of data sets in A and B must equal the set of data items accessed by T prior to the execution of the split-transaction statement. Without loss of generality, we assume that A precedes B in the following discussion on the intersections of the data sets for A and B:

1. $\texttt{AWriteSet} \cap \texttt{BWriteSet} \subseteq \texttt{BWriteLast}$

2. $\texttt{AReadSet} \cap \texttt{BWriteSet} = \emptyset$

3. $\texttt{BReadSet} \cap \texttt{AWriteSet} = \texttt{ShareSet}$

Property 1 states that data items in `BWriteLast` are written last by B. This property says that A should not clobber B's output, but B is allowed to write over A's output. Property 2 says that A can be serialized before B, since A has not seen any of the results produced by B. Property 3 says that B may see the results from A; in other words, A is the preceding transaction and B is the following transaction.

The above properties 1, 2, and 3 serialize A before B. Since the transaction manager guarantees the serializability of T, if A and B can be serialized with each other then they can be serialized with respect to all other transactions.

If both `ShareSet` and `BWriteLast` are empty, we call this the *independent* case, in which there is no data access conflict between A and B, so they can be serialized in either order. Since A and B are independent, they can both continue without additional restrictions.

If either `ShareSet` or `BWriteLast` is not empty, we call this the *serial* case, in which B follows A because of data access dependencies. In the serial case, each data item in the `ShareSet` must remain unchanged in A after the split. Otherwise, B would be using uncommitted data. In the serial case where `ShareSet` is non-empty, if A aborts at some time, B must also be aborted since otherwise B would be relying on aborted data. B need not be aborted, however, if `BWriteLast` is non-empty but `ShareSet` is empty.

One special case of the serial case is when we divide T into two consecutive subsequences, A and B. We then call A a *prefix* of T relative to B [7]. A prefix can be split and committed, saving its results and releasing its resources.

The more general case of the serial case is when the operations in A and B interleave, but for each data item, A's accesses constitute a prefix of T relative to B. If all information exchange between the interleaved operations is from A's blocks to B's blocks, we can rearrange the blocks to make A a prefix of T with respect to B.

When we discuss user-controlled transaction, we will concentrate on the independent case. As will be shown, although the prefix special case is very useful, the general serial case is inappropriate for open-ended activities.

The `AProcedure` and `BProcedure` given in the split-transaction statement may indicate procedure calls, with arguments, or represent statement labels (i.e., targets of goto statements). The subsequent code executed by A and B are by definition disjoint portions of the original code associated with T. A and B might incidentally invoke the same subroutines; what is intended is that the original computation of T is now divided between A and B.

Neither the previous notion of dividing operations completed in T between A and B nor this notion of "disjoint" portions of code is precisely defined. Since this description of the Split operation within programmed transactions is only to aid intuition when we later discuss user-controlled transactions, and not intended as a practical model, the lack of a formal semantics is not an issue.

The inverse Join operation on some transaction T dissolves T, making its results part of the target transaction S. The syntax of the join-transaction operation is:

```
Join-Transaction(S:TID)
```

Again, it is not necessary to mention T as a parameter, since join-transaction may only be executed within the body of this transaction T.

The join-transaction operation is conceptually simpler than split-transaction. Transaction T joins the target transaction S to commit (or abort) the results of T atomically with S. The sets of data items read and written by T, respectively, are merged into the sets of data items read and written thus far by S. It is not necessary to indicate a point in the code, since by definition join-transaction must be the last statement executed in T; the concurrent (with respect to T) execution of S is interrupted at some safe point, the join-transaction is accomplished, and then S continues where it left off. S may or may not make use of the resources inherited from T, depending on its code, but in any case both the data sets from T as well as those acquired directly by S are committed or aborted atomically as part of a single transaction.

As previously, it is unnecessary to be precise regarding the notion of "safe"; this kind of issue, however, is treated more carefully when we later examine user-controlled transactions.

3.2 Nested Transactions

The previous section assumed the simple transaction model where each transaction is composed of a *sequence* of operations that can be abstracted into Reads and Writes [12], starting with

Begin and ending with either Commit or Abort. This sequential model has been extended to include parallelism. Rather than a general concurrent model, where the transaction constitutes a partially ordered set of database operations, some form of *nesting* is employed [31, 34].

In the nested transactions model, there is a set of top-level transactions, which may be executed concurrently with respect to each other. A top-level transaction consists of a sequence of steps, where each step is either a primitive operation (Read or Write) or the initiation of a *subtransaction*. A subtransaction may execute concurrently with further steps in the top-level transaction as well as its sibling subtransactions. The subtransactions may themselves be nested.

The nested transactions model has been extended to include *supertransactions* [35]. A supertransaction encloses two or more independent transactions and makes them appear atomic to transactions outside the supertransaction.

The same properties of split-transaction described in section 3.1, relating the data sets in A and B, hold for the concurrent model. For the independent case, A and B are subsets of the transaction's operations that access disjoint sets of data items. For the serial case, A and B are subsets of operations such that for each resource, all writes in A precede the first read in B.

The join-transaction operation remains simple in the concurrent transaction model, since the operation can be invoked only at the end of the transaction, when all concurrency has ceased. We will first consider the case corresponding to prefix in the sequential model.

To model the prefix, we describe the set of operations as a directed-acyclic graph (DAG). An arrow represents a dependency between operations. There are no cycles since such a computation would be impossible to carry out. An isolated node in the graph represents an operation independent of other operations on the database. A prefix of computations in the DAG is a subset of nodes such that each node is either a source node (a node without any incoming arrows), or connected to a source node through a path entirely in the subset. In other words, the extended prefix contains only operations that are independent of the rest of the computation. One simple example of this extended definition of prefix is a subset of connected components in the DAG, which corresponds to the independent case. Since the two subsets are disconnected, there is no dependency between them, so they can be serialized either way. The serial case corresponds to a cut in the DAG, with the source part as transaction A and the sink part transaction B. Since A does not depend on B, A can be serialized before B.

For the general case, we really need the log of operations in T for the data flow analysis. Since the log records data access in a serial order, the same algorithms from the sequential model suffice. The same observation applies to the external data access.

It is important to clarify that dynamic restructuring of transactions is not the same as either nested transactions or supertransactions. It is possible to solve Charlie's problem posed in section 2, of losing F when his long transaction aborts, by treating the work on F and the work on G as two subtransactions of Charlie's top-level transaction, and making subtransactions rather than top-level transactions the unit of failure atomicity (recovery) [43].

It is also possible to extend the notion of nested supertransactions to allow interaction between Bob and Alice by treating their work on F, G, and the other modules as all transactions of the same supertransaction, and in fact such transaction grouping is the basis of many approaches to open-ended activities [39, 14].

But both nested transactions and transaction groups presume that the structure of the set of (sub)transactions is known in advance, when the code is written. In contrast, we require support for unpredictable developments, where the durations and particularly the possible interactions are determined while the transactions are in progress rather than in advance before they start. Thus programmed transactions containing split-transaction and join-transaction really do not make much sense; the real application is to what we call *user-controlled transactions*.

4 User-Controlled Transactions

We now present split-transaction and join-transaction in the intended context of open-ended activities with unpredictable developments, where users determine what database operations to execute as they go along.

The user commands supported by the transaction manager should include:

- Begin-Transaction,
- Commit-Transaction,
- Abort-Transaction,
- Split-Transaction,
- Split-Commit-Transaction,
- Join-Transaction
- Accept-Join-Transaction
- Suspend-Transaction,

- Resume-Transaction.

The Begin, Commit and Abort commands operate as expected. The Abort command would be selected by the user to intentionally rollback to before her transaction began, probably to return to a known consistent state after modifying the data into a hopeless mess. (It might be desirable to save the modifications in some way, so the user could retrieve partial work, but we do not address this.)

Instead of Read and Write, the other user commands are specific to the application, e.g., edit and compile for software development. We can abstract these user commands into those that invoke tools that only read, those that read and write, and those that only write. Some commands, such as compile, will invoke tools that read one or more data items (the source code and include files) and write one or more other data items (the object code and symbol table). Thus without loss of generality, we continue in this section to refer to Read and Write operations and Read and Write data sets.

The Split command during some transaction T produces two new transactions, A and B, and dissolves the original T. The user provides six arguments, as shown below.

```
Split-Transaction(
    AReadSet, AWriteSet, AUser,
    BReadSet, BWriteSet, BUser)
```

`AReadSet`, `AWriteSet`, `BReadSet`, and `BWriteSet` are sets of data items accessed or reserved in A and B. `AUser` and `BUser` indicate the users who should take control of A and B, respectively.

The Split command divides all the user commands previously completed in T into two subsets, A and B. The data sets associated with A and B are the data items accessed by the commands in A and B, respectively. The transaction manager permits the Split command only if the following hold:

1. $\mathtt{AWriteSet} \cap \mathtt{BWriteSet} = \emptyset$
2. $\mathtt{AReadSet} \cap \mathtt{BWriteSet} = \emptyset$
3. $\mathtt{BReadSet} \cap \mathtt{AWriteSet} = \emptyset$

This corresponds to the independent case of the previous section, in which there is no data access conflict between A and B, so they can be serialized in either order.

The general serial case discussed in the previous section is not desirable for user-controlled transactions, with unpredictable developments. In that case, execution of A and B would depend on B being serialized after A, unreasonably restricting the commands that could be performed later on in A. Further, if A aborts then B must abort, unreasonably throwing away BUser's work because of some problem detected by AUser – or a crash of AUser's workstation.

Only the prefix special case of the serial case seems to makes sense for user-controlled transactions. To enforce that the newly split A transaction immediately commits, a distinct command is provided. The Split-Commit command takes only five arguments:

```
Split-Transaction(
    AReadSet, AWriteSet,
    BReadSet, BWriteSet, BUser)
```

Split-Commit divides all the user commands previously completed in T into two subsets, A and B. The data sets associated with A and B are the data items accessed by the commands in A and B, respectively. Then A immediately commits, while B may be taken over by BUser.

The following restrictions are placed:

1. $\mathtt{AWriteSet} \cap \mathtt{BWriteSet} \subseteq \mathtt{BWriteLast}$

2. $\mathtt{AReadSet} \cap \mathtt{BWriteSet} = \emptyset$

3. $\mathtt{BReadSet} \cap \mathtt{AWriteSet} = \mathtt{ShareSet}$

If A can immediately commit, as a subaction of the Split-Commit command, then `ShareSet` and `BWriteLast` need not be empty. Notice that the commit of A should not release any resources in `ShareSet` or `BWriteLast`, since these must continue to be held by B.

For Split or the B transaction of Split-Commit, a user must explicitly take control of a newly split transaction, using the new Resume command. It does not make much sense for a user to suddenly find that her commands are part of some transaction split by another user. Further, it is reasonable that in some cases that AUser and BUser will be the same person, often the same as the person who selected the Split command. But of course this user's commands cannot be part of both A and B, then they would not be split.

The Suspend command provides a means for giving up control of a transaction, in order to Resume another, or to execute commands outside any transaction. (From the transaction managers' point of view, each individual command executed outside any transaction is itself a transaction, and must be serializable with respect to all others.)

The notion of taking control over a transaction requires transaction *naming*, so that a user can indicate which particular transaction to Resume. One apparent way to avoid introducing user-visible transaction names is to restrict each user to exactly one transaction at a time. Then Resume by definition would take control over the user's personal transaction, and would fail if there had not been either a previous Suspend by this user or a previous Split assigning her one of the newly split transactions. Note that this disallows the case discussed above where AUser and BUser are the same person. This approach does not seem very practical, so we need transaction names.

A transaction's name might be assigned by the user as an argument to the Begin command. Newly split transactions would be named by additional arguments to the Split and Split-Commit commands. Perhaps a catalog of active and suspended transactions could be maintained by the design environment, with each associated with a human-intelligible description of the task being undertaken in that transaction. Further discussion of user interface issues is outside the scope of this paper; see [20].

The Join command on some transaction T dissolves T, making its results part of the target transaction S. The user provides one argument, the name of the ongoing transaction to join.

```
Join-Transaction(S:TNAME)
```

Again, transaction T joins the target transaction S to commit (or abort) the results of T atomically with S. The sets of data items read and written by T, respectively, are merged into the sets of data items read and written thus far by S.

But the user controlling S should agree to accept responsibility for T's changes, prior to completing the merge. Thus this user must execute the Accept-Join command in transaction S, with the argument T.

```
Accept-Join-Transaction(T:TNAME)
```

The Accept-Join command might be executed before or after the original user requests the Join command, but the join is not consummated until both users have agreed. The environment might provide additional support, for example, to notify one user that another user had asked to join her transaction.

5 Applications

5.1 Editing

One of the simplest open-ended activities is the modification of a text file, known as editing. The fundamental problem in enclosing an editing session within a transaction is that if the machine crashes in the middle of the session, all the typing effort would have been lost due to abort rollback. Typical text editors provide the explicit-rollback aspect of transactions with an "undo" facility and the fault-tolerance aspect by periodically saving consistent intermediate results on disk (sometimes called a "checkpoint"). In the editing of a single text file, a consistent state is defined by the completion of all previous editor commands.

To provide the same functionality of saving intermediate results of a single text file, the editor can split to commit after any command. At that point, the consistent version is split into A's writeset and the current state (which happens to be the same) into B's readset. Committing A saves the consistent version on disk, and B continues the editing transaction, as if it has started from the version A has just written to disk.

Note that in this example, the programmed transaction model works, since the points to execute split-commit-transaction are preplanned (e.g., after every N commands). Although conventional "undo" and "checkpoint" is probably a better model for editing of a single text file, transaction restructuring operations become more useful when the user is editing several files that depend on each other for consistency, to coordinate the checkpointing of multiple files.

5.2 Design Environments

We mentioned in the introduction that open-ended activities are characterized by uncertain duration, unpredictable developments, and interaction with other concurrent activities. One example of uncertain duration is that an operation in design environments may take arbitrarily long — it may even be set aside by the user for arbitrary periods of time, without committing, while he works on some higher priority task. The problem with traditional transactions is that no information can be released while the design remains in the drawer. The transaction restructuring operations would allow partial results to be published by committing transaction A while the unfinished part stays under wraps in transaction B.

Another example of unpredictable developments arises when a programmer sets out to fix a bug. He normally goes through a cycle where he examines the program's execution in a debugger, reads certain source files that he guesses are related to the bug, modifies some subset of these files, compiles and links, runs test cases, discovers that the patch does not work, backs up to the previous versions of some or all of these files, reads other source files (perhaps overlapping with the first set), modifies some subset, etc., until he's finally satisfied with the results and commits all changes. Transaction restructuring allows the programmer to back up by aborting transaction A while keeping the changes he's currently happy with in the ongoing transaction B. In the case of fixing multiple bugs, he may split to commit some subset of the changes along the way to allow other programmers to take advantage of the code already fixed.

In a traditional transaction, in contrast, the solutions become public only when all the changes are committed. Note also that all source files read would remain unavailable until commit — even if they turn out irrelevant to the bug(s)! The former problem can be avoided by publish each part of the change as soon as it is made, but this might lead to a catastrophic cascading rollback if the programmer decided his solution was incorrect and aborted the transaction. The latter problem can be avoided by using optimistic concurrency control, but then the programmer might discover when it came time to commit (1) that his changes were invalidated by activities of other programmers or (2) if multiple versions are used, then he is faced with a complicated merge.

Although traditional transactions are clearly inappropriate, design environments still need the fault-tolerance associated with transactions. Consider a VLSI layout tool that has been given a specification and is now busy laying out. Such tools often run for several hours, so it is desirable to save intermediate results if (1) it is possible for the tool to continue from the saved results, or (2) the intermediate results include error messages that it would be useful for the human users to see —to correct his specifications— even though the full set of error messages was not generated due to a crash. This applies to make [13] as well, where the appropriate split points are the distinct command lines. However, an individual linking or compilation job can run very long, and reasonable split points are not so obvious — and in fact only (2) applies to most instances of these tools. Atomic transaction would rollback everything in the case of a crash.

5.3 Multi-User Design Environments

The class of systems called "design environments" includes both single-user design tools and multi-user environments that support cooperation among multiple users working together on the same large project. The same issues come up in office automation systems and other computer-supported cooperative work [17].

A simple example of cooperation among multiple users is when a programmer has finished a hash-table package that she is going to use in some larger program. Somebody else discovers her package and wants to use it. Even though this has not been planned beforehand, she can split the transaction with the hash-table in transaction A, to be committed and released, while continuing her own work in transaction B. If she had only atomic transactions available, her manager would have had to foresee the usefulness of the hash-table package and therefore assign it as a separate task, and thus a separate transaction.

For a more complex example, consider again a coordinated change to modules F and G, where programmer Bob is responsible for F and programmer Alice for G. Bob needs to use the new version of G and Alice needs to use the new version F. When we treat Bob's work and Alice's work as a pair of traditional transactions, we arrive at a deadlock.

However, most modern programming languages provide language constructs for separating the specification portion of a module from its implementation, and enforce the constraint that one module can depend only on the specification part of another and can in no way depend on the details hidden in another module's implementation part [38]. This works well using transaction restructuring operations: Bob and Alice can modify their specifications at the same time; the transactions are then split making the new specifications public; finally, Bob and Alice can modify their implementations using the new specifications. If we had used simple nested transactions, only the immediately enclosing transaction would be able to see the specification, instead of the public.

This simple division at the specification/implementation level does not work well for changes involving large numbers of programmers — say, more than twenty — because too many programmers could be held up waiting for all the others to finish their specifications so the transactions can split. This problem can be solved by dividing the programmers into groups whose modules most closely depend on each other [21]. Only programmers within the same group see the new version of a module, while others initially use the old version; only the members of the same group must split their transactions at the same time. A further set of splits is made later when the groups coordinate to make their new versions public to all the other groups so they can complete the necessary modifications.

Another reason to split an ongoing transaction is management re-organization. For instance, one project transaction may have to be split into two to adjust workload. If the project can be separated into two serializable parts, then a split operation would define the responsibilities clearly. In such cases, it may be desirable for a managerial user to be able to execute transaction restructuring operations externally, to affect in-progress transactions controlled by subordinate users [22].

Join-transactions are useful when two open-ended activities turn out to relate to each other through some consistency constraints, *after* the activities have started. For example, modules from two programmers may need to be bundled together in an emergency release for the unexpected development of a demo for a potential customer. If the programmers knew they were bound by the same consistency constraints at the beginning, they could have created two subtransactions nested in a larger one, which maintains the consistency constraints. However, due to unpredictable developments in open-ended transactions, the flexibility to join originally separate transactions is desirable. In our example, the demo may require a set of modules that work with each other and some *ad hoc* utility programs that show off those modules. Join-transaction can group a particular subset of modules that have been tested together and release them to the utility programmers.

6 Implementation Issues

When the split-transaction operation is called, the transaction manager checks the external operations on data using an optimistic concurrency control algorithm [26] to certify the serializability between A and B. This certification sweeps the external operations recorded on the log or versions to verify the three properties enumerated in section 3.1, for programmed transactions, or in section 4 for user-controlled transactions, to ensure serializability. Note that optimistic concurrency control is employed only to detect conflicts between A and B *before* rather than *after* they have been split, so there is no possibility of rollback of the long transaction — the worst that can happen is the split is disallowed.

Version-based recovery algorithms record read accesses as well as write accesses. But typical log-based database systems register only the write accesses. For the above automatic checking algorithm to work, we need to record the long-transaction read accesses on the log to verify the three properties. In case the user-specified split is not serializable, the transaction manager might modify the set A or B and calculate some serializable splits, using closure algorithms, as suggestions to the user. In practice, the user may request a split and choose the appropriate split from a menu.

We should recognize that even though external accesses conform to the three properties, rigorously speaking the algorithm in general does not guarantee serializability between A and B. The reason is that as part of the same original transaction, A and B may have exchanged information through shared variables, e.g., clipping from one editor buffer and pasting into another. To guarantee serializability in the rigorous sense, we need to record all data access (as in an undo log for an editing session) in the long-transaction and perform a data flow analysis on the internal operations in T.

We note that even though we have introduced the need for logging of read access and possibly shared variable access, this kind of overhead is imposed only on long-transactions, and is not needed for short transactions executing in the same database. Since long-transactions have low throughput by definition, the additional overhead will not become a bottleneck.

A pessimistic scheme is necessary in the rest of the implementation, to avoid undesired roll-back of long transactions. Otherwise, our transaction restructuring operations are independent of any particular concurrency control or failure recovery methods. Any pessimistic implementation technique chosen in a database system, as long as it guarantees serializability, can be employed in conjunction with split-transaction and join-transaction. We will consider the two major techniques: two-phase locking and timestamps.

The general algorithm for split-transaction contains three steps after the transaction manager has validated that the operation is valid. First, the transaction manager creates a new unique transaction identifier (TID), say for B. Second, it adds B's TID to the reader's list of all the data items in `BReadSet`. Third, it reassigns B's TID to be the writer of all the data items in `BWriteSet`. Specifically, for two-phase locking, the transaction manager just adds or assigns B's TID to the appropriate lock owner list. For timestamp-based methods, it updates the timestamps for the data with B's TID. Transaction A retains T's original transaction identifier. Threads of control in A and B are transferred to AProcedure and BProcedure, respectively, when implementing programmed transactions.

From the concurrency control point of view, the transfer is relatively straightforward. For two-phase locking, the transaction manager simply changes the ownership of the locks in BReadSet and BWriteSet to the new transaction. For timestamps, it needs to assign a new timestamp to B, preferably immediately after A's timestamp. Multidimensional timestamps [28] is one of the techniques that provides this capability.

From the crash recovery point of view, the transfer is also simple. For recovery systems based on versions (which is what we have assumed in this discussion), the system simply makes B the creator of the versions in BWriteSet. For systems based on logging, the recovery manager needs to write a special "transaction record" to the log, marking the transition of BWriteSet data from the old transaction to the new transaction. During the log processing for recovery, the transaction record should be read during the backward pass, and then the log records on BWriteSet written by the old transaction will be translated correctly as B's records.

Another important concern is the restricted case of cascaded aborts introduced by the general serial case for programmed transactions (section 3.1). If A aborts then B must abort, since B has read data written by A. Fortunately, this restricted case is much easier to handle than general cascaded aborts. Specifically, the database system does not have to maintain the dependencies among all transactions. It is sufficient to remember only the dependency between transactions that have split, given to the transaction manager explicitly by the split-transaction operation. Transactions that have not split are protected by the usual concurrency control mechanism. Consequently, if the "native" concurrency control prevents cascaded aborts (e.g. strict two-phase locking), then the cascaded aborts are restricted exclusively to transactions that have split. Recall that cascaded aborts are explicitly prevented by the properties required for user-controlled transactions (section 4).

The implementation of join-transaction is straightforward. Say that the transaction manager is trying to join-transaction T and S. Since a pessimistic concurrency control scheme is used, it is guaranteed that T and S are already serializable with respect to each other, so there is no need to check. The transaction manager then adds TReadSet to SReadSet and TWriteSet to SWriteSet. If S is aware of T's joining, then S can start accessing data read and written by T. This is unlikely to be the case for programmed transactions, but works correctly if S would have tried to access T's data later on in its code anyway, since there are now no conflicts. This general algorithm works for all the concurrency control methods and crash recovery techniques mentioned above for split-transaction.

7 Comparison to Related Work

There has been a flurry of research to develop new approaches to transaction management that meet the requirements of open-ended activities given in section 2. One of the authors has developed a survey of extended transaction models proposed in the literature and/or evident in existing systems [6]. The survey concentrates on concurrency control issues, and does not directly address recovery.

Figure 1 summarizes the main models covered in that survey in terms of whether they have been implemented, and their capabilities for supporting uncertain duration, unpredictable developments, and interaction with other concurrent activities. None of the existing models, including our own, provides a fully satisfactory solution to all three problems — but it may be possible to reach such a solution by combining some of the mechanisms that have complementary strengths, for example, participation domains and transaction restructuring. A limited form of participation domains is currently being implemented for the multi-user Marvel environment [5, 4], and we are investigating the integration of participation domains with transaction restructuring operations.

It is interesting to note that many of the ideas implemented in the mechanisms surveyed were actually discussed earlier in other contexts. For instance, some of the ideas related to

Mechanism	System	Duration	Unpredictable	Interaction
Altruistic Locking[37]	N/A	Yes	No	Limited
Snapshot Validation[33]	N/A	Yes	No	Limited
Order-Preserving Transactions[45]	DASDBS	Yes	No	Limited
Entity-State Transactions[27]	N/A	Yes	No	Limited
Semantic Atomicity[15]	N/A	Yes	No	Limited
Multilevel Atomicity[30]	N/A	Yes	No	Yes
Sagas[16]	N/A	Yes	No	Limited
Conflict-Based Serializability[25]	N/A	Yes	No	Limited
Checkout[41]	RCS	No	No	Limited
Conversational Transactions[29]	System R	Limited	Limited	No
Multilevel Coordination[21]	Infuse	Yes	Yes	Limited
Domain Relative Addressing[42]	Cosmos	Yes	No	Limited
Copy/Modify/Merge[1]	NSE	Yes	Yes	Limited
Multiple Commit Points[43]	N/A	Yes	Yes	Limited
Interactive Notification[11]	Gordion	No	Limited	Limited
Visibility Domains[10]	N/A	Yes	Limited	Yes
Group-Oriented CAD Transactions[24]	N/A	Yes	Limited	Yes
Cooperating CAD Transactions[3]	Orion	Yes	Limited	Yes
Transaction Groups[14]	ObServer II	Limited	Limited	Yes
Participation Domains[19]	N/A	Yes	Limited	Yes
Transaction Restructuring	N/A	Yes	Yes	Limited

Table 1: Comparison Table of Mechanisms

multilevel transactions, long transactions and cooperative transactions were discussed by Davies in 1973 [9]!

Our work was influenced by the brief mention of the terms *split* and *join* in conjunction with transactions in a technical report by Jessop *et al.* [18] discussing the Eden Transactional File System.

Chrysanthis and Ramamritham have defined the ACTA formalism for specifying the semantics of various transaction models in terms of commit-dependencies and abort-dependencies among transactions [8]. They have used this formalism to model several extended transaction schemes, including our transaction restructuring operations. They also tested the reasoning capabilities of their formalism by combining our model with conventional nested transactions, to verify that the result retained the properties of the two original schemes.

8 Conclusions

We have informally introduced the class of open-ended activities, which arise in design environments, office automation systems and other advanced database applications. These activities have become an important application area for which the next generation of databases are aiming [40, 2]. The common requirements of these activities include the ability to support consistent concurrent access and fault-tolerance when the activities are of uncertain duration, may include unpredictable developments, and often require interaction with other concurrent activities.

To solve these problems, we have introduced the split-transaction and join-transaction operations for dynamically restructuring ongoing transactions. Split-transaction divides the data in a transaction into two sets of data items contained in two serializable transactions, while join-transaction merges the data in two transactions into a single set containing in one serializable transaction. We have described the syntax of these operations, their semantics, and the properties that they should satisfy. We also outlined the algorithms to implement these operations. We have demonstrated the safe use of our transaction restructuring operations for several real-world problems.

The traditional transaction guarantees consistent concurrent access and fault-tolerance of databases by enforcing concurrency atomicity and failure atomicity, both very desirable properties for conventional data processing, but problematic for open-ended activities. The uncertain duration, unpredictable developments, and interaction with others in open-ended activities are formidable challenges to the traditional transactions. With dynamic transaction restructuring, we maintain the desirable properties of transaction serializability at the same time as we increase flexibility.

Acknowledgments

We first described the split-transaction and join-transaction operations in Calton Pu, Gail E. Kaiser and Norman Hutchinson, Split-Transactions for Open-Ended Activities, *Fourteenth International Conference on Very Large Data Bases*, August 1988, pp. 26-37. This paper expands on our previous ideas, benefitting from discussions with Bob Balzer, Naser Barghouti, Panos Chrysanthis, Dan Duchamp, Bill Harrison, Scott Hudson, Eliot Moss, Dewayne Perry, Ken Salem and Stan Zdonik. We would particularly like to thank Wenwey Hseush for his corrections on a previous version of this paper. We would also like to thank the anonymous reviewers for their comments.

Kaiser is supported by National Science Foundation grants CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC and SRA, by the New York State Center for Advanced Technology on Computer and Information Systems and by the NSF Engineering Research Center for Telecommunications Research. Pu is supported by National Science Foundation grant CDA-8820759, by grants from AT&T, DEC and Sun, and the New York State Center for Advanced Technology on Computer and Information Systems.

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, Pittsburgh PA, May 1989.

[2] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. *The Object-Oriented Database System Manifesto.* Elsevier Science, 1990.

[3] Francois Bancilhon, Won Kim, and Henry Korth. A model of CAD transactions. In *11th International Conference on Very Large Databases*, pages 25–33, Stockholm, Sweden, August 1985.

[4] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments.* PhD thesis, Columbia University, 199x.

[5] Naser S. Barghouti and Gail E. Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990.

[6] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 1991. In press. Available as Columbia University Department of Computer Science CUCS-425-89, revised March 1991.

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

[8] 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.

[9] Charles T. Davies, Jr. Recovery semantics for a DB/DC system. In *28th ACM National Conference*, pages 136–141, Atlanta GA, August 1973.

[10] 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.

[11] 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.

[12] 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.

[13] S.I. Feldman. Make — a program for maintaining computer programs. *Software — Practice & Experience*, 9(4):255–265, April 1979.

[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] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

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

[17] Irene Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings.* Morgan Kaufmann, San Mateo CA, 1988.

[18] W.H. Jessop, J.D. Noe, D.M. Jacobson, J-L. Baer, and C. Pu. An introduction to the Eden transactional file system. Technical Report 82-02-05, Department of Computer Science, University of Washington, February 1982.

[19] Gail E. Kaiser. A flexible transaction model for software engineering. In *6th International Conference on Data Engineering*, pages 560–567, Los Angeles CA, February 1990. IEEE Computer Society.

[20] Gail E. Kaiser. Interfacing cooperative transactions to software development environments. *Office Knowledge Engineering*, 4(1):56–78, February 1991. Invited paper.

[21] Gail E. Kaiser and Dewayne E. Perry. Workspaces and experimental databases: Automated support for software maintenance and evolution. In *Conference on Software Maintenance*, pages 108–114, Austin TX, September 1987. IEEE Computer Society Press.

[22] Gail E. Kaiser and Dewayne E. Perry. Making progress in cooperative transaction models. *Data Engineering*, 14(1), March 1991. Invited paper.

[23] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[24] 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. ACM.

[25] Henry F. Korth and Gregory D. Speegle. Formal model of correctness without serializability. In *SIGMOD International Conference on the Management of Data*, pages 379–386, Chicago IL, June 1988. ACM Press.

[26] H. T. Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[27] Ali R. Kutay and Charles M. Eastman. Transaction management in engineering databases. In *Engineering Design Applications, Annual Meeting Database Week*, pages 73–80, San Jose CA, May 1983. IEEE Computer Society Press.

[28] P.J. Leu and B. Bhargava. Multidimensional timestamp protocols for concurrency control. *IEEE Transactions on Software Engineering*, SE-13(12):1238–1253, December 1987.

[29] Raymond Lorie and Wilfred Plouffe. Complex objects and their use in design transactions. In *Engineering Design Applications, Annual Meeting Database Week*, pages 115–121, San Jose CA, May 1983. IEEE Computer Society.

[30] Nancy A. Lynch. Multilevel atomicity — a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[31] J. Eliot B. Moss. Nested transactions and reliable distributed computing. In *2nd Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, Pittsburgh PA, July 1982. IEEE Computer Society.

[32] Erich Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record*, 18(1):17–26, March 1989.

[33] U. Pradel, G. Schlageter, and R. Unland. Redesign of optimistic methods: Improving performance and availability. In *International Conference on Data Engineering*, pages 466–473, Los Angeles CA, February 1986.

[34] Calton Pu. *Replication and Nested Transactions in the Eden Distributed System*. PhD thesis, Department of Computer Science, University of Washington, 1986.

[35] Calton Pu. Supertransactions. In *2nd Workshop on Large Grained Parallelism*, Hidden Valley PA, October 1987.

[36] Lawrence A. Rowe. Report on the 1989 Software CAD Databases Workshop. In Gerhard Ritter, editor, *11th World Computer Conference IFIP Congress '89*, pages 719–725, San Francisco CA, August 1989. Elsevier Science Publishers B.V.

[37] Kenneth Salem, Hector Garcia-Molina, and Rafael Alonso. Altruistic locking: A strategy for coping with long lived transactions. In *2nd International Workshop on High Performance Transaction Systems*, Pacific Grove CA, September 1987.

[38] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, 1(4):10–26, October 1984.

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

[40] Michael Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, and David Beech. Third-generation database system manifesto. *SIGMOD Record*, 19(3):31–44, September 1990.

[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] 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, Singapore, August 1984. Morgan Kaufmann.

[44] 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.

[45] Gerhard Weikum and Hans-Jorg Schek. Architectural issues of transaction management in multi-level systems. In Umeshwar Dayal, G. Schlageter, and Lim Huat Seng, editors, *10th International Conference on Very Large Data Bases*, pages 454–465, Singapore, August 1984. Morgan Kaufmann.