Split-Transactions for Open-Ended Activities

Calton Pu and *Gail E. Kaiser*

Department of Computer Science
Columbia University

Norman Hutchinson

Department of Computer Science
University of Arizona

Abstract

Open-ended activities such as CAD/CAM, VLSI layout and software development require consistent concurrent access and fault tolerance associated with database transactions, but their uncertain duration, uncertain developments during execution and long interactions with other concurrent activities break traditional transaction atomicity boundaries. We propose split-transaction as a new database operation that solves the above problems by permitting transactions to commit data that will not change. Thus an open-ended activity can release the committed data and serialize interactions with other concurrent activities through the committed data.

1 Introduction

Transactions provide atomicity in two senses: concurrency atomicity (consistent concurrent access) and reliability atomicity (fault-tolerance) in databases. However, for several reasons these atomicity properties restrict the applicability of the transaction concept in *open-ended* activities, such as CAD/CAM projects, VLSI design and software development in programming environments, 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),
- uncertain developments (actions unforeseeable at the beginning), and
- interaction with other concurrent activities.

Each one of these characteristics introduces difficulties when constrained by traditional transaction atomicity properties. For example, 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 at abort. Second, long transactions become concurrency bottlenecks because updated resources are retained for the duration of the transaction. 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.

One representative model of CAD transactions [1], developed by Bancilhon, Kim, and Korth, tries to solve these difficulties with an extended transaction model tailored for CAD activities. In this model, the CAD environment is divided into six hierarchical conceptual levels: project transactions, cooperating transactions, clients/subcontractors, short-duration transactions, sequences of database operations, and system operations. Each level is composed of a set, a sequence, a hierarchy, or a directed-acyclic graph of components of the immediately lower level. Despite this sophisticated activity structure, aborting a transaction at any level implies aborting all the enclosed activities. In addition, interaction with other concurrent activities is limited to the

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

sibling transactions that are enclosed in the same parent transaction.

Unpredictable developments in open-ended activities reveal two serious problems intrinsic to traditional transactions. First, sometimes initial activities modify data early in the transaction, which will remain unchanged until commit. For instance, suppose we are developing two modules to be released together in a transaction, F and G. Module F has been coded and tested before module G is finished. In this case, transaction abort due to crashes will cause the loss of module F. Second, some (or all) of the later activities may become independent of the previous activities. In the above example, module F may have been modified so it does not use module G any more. In this case, module F remains locked unnecessarily until transaction commits. In both cases, even though the original plan might commit F and G atomically, the developments lead us to want to commit module F, while continuing with module G.

To solve the difficulties outlined above, we introduce the notion of *split-transaction*, which divides an ongoing transaction into two serializable transactions. In particular, resources read and updated by the original transaction are divided among the resulting transactions into two or more sets. Thereafter, each one of the resulting transactions may proceed independently with its own data. We use split-transaction mainly to commit one of the new transactions and release useful results from the original transaction. The other new transaction continues.

Split-transaction brings three major advantages into transaction-oriented open-ended activities:

1. adaptive recovery, committing resources that will not change,
2. added concurrency, releasing the committed resources, and
3. serializable access to resources by all activities.

The inverse operation of split-transaction, called join-transaction for historical reasons (as explained later), can combine results together and release them atomically. Using split-transactions and join-transactions we can exploit the three advantages enumerated above to support open-ended activities.

With split-transactions, we do not intend to settle the debate of whether serializability is too restrictive for open-ended activities. Nevertheless, powerful serializable access is useful even in a system that admits non-serializable operations. Therefore, like a significant portion of previous work [1,4], we restrict our discussion to serializable access.

Another non-goal of this paper is the philosophical discussion of whether in principle an "atomic" transaction can be split completely. We only remind the gentle reader that in Physics atoms were split years ago.

The rest of this paper is organized as follows. In section 2 we define terms and the concepts of split-transaction and join-transaction. In section 3 we apply split-transactions to several open-ended activities to demonstrate its usefulness. In section 4 we integrate consistent database access by long and short transactions using split-transactions. In section 5 we discuss the implementation of split-transaction to show it is practical. In section 6 we compare split-transaction to other work on transactions for design activities. Section 7 concludes the paper.

2 Split-Transaction and Join-Transaction

2.1 Definitions and Assumptions

Our database is a set of *objects*, which are accessed by transactions. The results in this paper apply to transaction processing, independent of the data model and schemas supported by the database. Since we assume a simple database model with the basic read and write (update) operations, we believe our results can be explored further in *object bases* (also called persistent objects). The recent body of work on object bases [17], inspired by object-oriented languages [6], emphasizes navigation through complex structures as well as the exploitation of semantics of high-level operations to in-

crease reliability and concurrency. In this aspect, our approach is orthogonal to the object base work.

Each object is a sequence of *versions*; each version is produced by an update operation. Even though we make this standard assumption about the versions, we do not impose any particular concurrency control methods on the database. The operations that the database manager supports are:

- Begin-Transaction,
- Commit-Transaction,
- Abort-Transaction,

The split-transaction operation on transaction T produces the new transactions A and B. The inverse operation, join-transaction, finishes executing the transaction T (like a commit-transaction) and makes its results part of the target transaction S. The syntax of the split-transaction and join-transaction is:

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

For semantic simplicity, even though T may not have completed all its activities, we assume T is in a quiescent state (similar to abort or commit) when it is split. In other words, the resources will not change during the split-transaction. **AReadSet**, **AWriteSet**, **BReadSet**, and **BWriteSet** are sets of objects accessed in A and B. **AProcedure** and **BProcedure** are the starting points of code in which A and respectively B will continue to execute.

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. Because of the semantics of split (explained below), the n-way split-transaction may be seen as syntactic sugar for the successive two-way split.

The split-transaction operation divides all the activities completed in T into two subsets, A and B. The object sets associated with A (respectively, B) are the objects accessed by A (B). From the definition, the union of object sets in A and B must equal the set of objects accessed by T so far. Without loss of generality, we assume that A precedes B in the following discussion on the intersection of the object sets in A and B:

1. **AWriteSet** $\cap$ **BWriteSet** = **BWriteLast**
2. **AReadSet** $\cap$ **BWriteSet** = $\emptyset$
3. **BReadSet** $\cap$ **AwriteSet** = **ShareSet**

In property 1, objects in **BWriteLast** are updated 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 the following transaction.

The above properties 1, 2, and 3 serialize A before B. Since the database system 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 object 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 object in the ShareSet must remain unchanged in A after split-transaction. Otherwise, B would be using uncommitted data. Also in the serial case, if A aborts at some time, B must be aborted since otherwise B would be relying on aborted data.

The join-transaction is conceptually simpler than split-transaction. Transaction T joins the target transaction S to commit (or abort) the results of T atomically with S. T can be thought as a subtransaction of S. Alternatively, T and S are sibling transactions under the same supertransaction. An example of the use of join-transaction is to group the results from several designers for a release. After the atomic commit of

their results they can immediately split again, regaining their own transactions to proceed independently, if the information exchanged between the designers during the supertransaction is serializable.

We have used the abstract term "computation" to denote the activities in both A and B. The nature of computation depends on the transaction model, and it will be described in detail in the following sections.

2.2 Sequential Transaction Model

In this section, we describe computations involving the split-transaction operation for a simple transaction model. In the next section we extend it to the general case. In this simple model, a transaction is a (linear) sequence of database operations.

For the independent case, A and B are subsequences of the transaction; objects accessed by A have not been accessed by B, and vice versa. For the serial case, A has not accessed any objects updated by B, all values read by B from A's objects will not change until after A's commit, and if A aborts then B must abort.

Because of typical uncertainties in open-ended activities, we want to give to the transaction programmer (for example, the designer using a CAD database) the ability to specify the readset and writeset of both A and B. Once the split-transaction operation is called, the database manager checks the external operations on objects using an optimistic concurrency control algorithm, either centralized [9] or distributed [2], 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 2.1, which ensure serializability.

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 programmer-specified split is not serializable, the database manager can modify the set A or B and calculate some serializable splits, using closure algorithms, as suggestions to the programmer. In practice, the programmer 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. 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.

The abstract description of split-transaction may not give us an immediate intuition of its usefulness. So let us consider a few examples in which split-transaction yields a desirable result. For example, let us divide T into two consecutive subsequences, A and B. We assume the division is such that property 1 holds. If A and B are independent, they can commit independently. Otherwise, since all object accesses in A precede all accesses in B, they trivially satisfy our property 2. Also, after B has started A did nothing, so A can commit at any time and then B can commit. This case is so important that we call A a *prefix* of T relative to B. A prefix can be split and committed, saving its results and releasing them to the world outside T.

A slightly more general case is when the operations in A and B interleave, but for each object, A's accesses constitute a prefix of T relative to B. Assuming that between the interleaved operations all information exchange is from A's blocks to B's blocks, we can rearrange the blocks to make A a prefix of B. This case may happen when modifications to a group of objects must be committed atomically for an interim release, but further work continues on the same objects. This case also motivates the more general model of transactions with concurrent operations that we shall describe in the next section.

In the sequential model, join-transaction is also simple. For a transaction T to join another transaction S, T finishes its operations with the join-transaction call, instead of commit-transaction. T's results will be incor-

porated into S and committed/aborted atomically with S. Whether S makes use of the objects inherited from T or not is up to the open-ended activity in progress in S.

2.3 Nested and Concurrent Transaction Model

The sequential model of transactions has been extended to include parallelism and nesting [11,13]. In the concurrent model, a transaction is a set of database operations. Typically, nested transactions are introduced to serialize object access within a transaction. A *subtransaction* is a subset of operations that form an atomic unit with respect to the enclosing transaction (parent) and the sibling subtransactions. For example, in design activities, parts of the design may be carried out by different agents (tools or personnel using tools). The nested transactions model has been extended to include *supertransactions* [12]. A supertransaction encloses two transactions and makes them appear atomic to transactions outside the supertransaction. One application of the supertransaction is to implement the join-transaction operation.

The same properties of split-transaction described in Section 2.1, relating the object 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 objects. For the serial case, A and B are subsets of operations such that for each resource, all updates 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. In contrast, nondeterminism in the set of operations makes the syntactic checking of split-transaction even more difficult. We will consider first 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 each 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 object access.

3 Applications

3.1 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, we can split-transaction after any editor 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.

Although "undo" and "checkpoint" is probably the best model for editing of a single text file, split-transaction becomes more useful when the user is editing several files that depend on each other for consistency, to coordinate the checkpointing of multiple files.

3.2 CAD/CAM/CASE Databases

We mentioned in the introduction that open-ended activities are characterized by uncertain duration, uncertain developments, and interaction with other concurrent activities. One example of uncertain duration is that a design activity in computer-aided design, manufacturing or software engineering (CAD/CAM/CASE) may take arbitrarily long — it may even be set aside by the designer 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. Split-transaction would allow partial results to be published by committing transaction A while the unfinished part stays under wraps in transaction B.

An example of uncertain developments arises in CASE when a programmer sets out to fix a bug. He normally goes through a cycle where he examines the program's operation 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 doesn't 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. Split-transaction 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 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 locked until commit — even if they turn out irrelevant to the bug(s)! The former problem can be avoided by publishing 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, that he is faced with a complicated merge.

Although traditional transactions are clearly inappropriate, CAD/CAM/CASE still needs 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's 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 designer to see —to correct his specifications— even though the full set of error messages was not generated due to a crash. This applies to CASE as well, where the "layout" tool becomes a long system generation job (e.g., using `make` [3]), where the appropriate split-transaction points are the distinct command lines. However, an individual linking or compilation job can run very long, and reasonable split-transaction 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.

3.3 Programming Environments

The class of systems called "programming environments" includes single-user CASE tools as a subset. Here we are concerned with another subset of programming environments: those that support cooperation among multiple programmers working together on the same large software system. The same issues come up in office automation systems in regards to report production and other computer-supported cooperative

work.

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 used an atomic transaction, her manager would have to foresee the usefulness of the hash-table package and therefore enclose it in a separate transaction.

For a more complex example, consider 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 [15]. This works well using the split-transaction: Bob and Alice simultaneously modify their specifications; the transactions are then split making the new specifications public; finally, Bob and Alice 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 split 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 [8]. 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 when another manager is designated to share part of the responsibilities. If the project can be separated into two serializable parts, then split-transaction would define the responsibilities clearly.

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 uncertain 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 programs.

4 Integrating Long and Short Transactions

Besides their direct use in applications mentioned in the previous section, split-transaction and join-transaction can also help in the design of the database system to support open-ended activities. Open-ended activities involve both long and short (traditional) transactions and, in any case, are likely to operate simultaneously on the same database as other applications involving only short transactions. If an open-ended activity has access to several resources, which are required by a pending

short transaction, the activity must not unduly hold up the short transaction.

To solve this problem with split-transactions, there are two possibilities:

- to continue the open-ended activity until the next opportunity for a split-transaction,
- or to rollback the open-ended activity to the previous point in its history where a split-transaction would be consistent.

In either case the resources are committed at the split, the short transaction makes its brief access to the resources, and the open-activity reacquires the necessary subset of these resources to continue its operation. In comparison, atomic long transactions cannot be subdivided, so short transactions have to wait for the long transactions to either commit or abort. There are no other alternatives.

Our first case is appropriate when the future time of the next opportunity for split-transaction can be reasonably predicted and the short transaction can afford to wait this long. The former requires application-specific capabilities for the relevant "real-time" predictions, while the latter is realistic only when the short transaction is part of a batch job rather than a user query session. This first case permits the least disruption of the user carrying out the open-ended activity, since the short transaction can take place during the user's "think time".

We believe the second case will be more common. Here, a history must be maintained for each long transaction and the time to analyze this history to find the most recent point for the split-transaction must also be very short, the same order of magnitude as traditional concurrency control overhead. Further, it is desirable although not strictly necessary to be able to automatically redo the part of the transaction that was rolled back, rather than requiring the user to repeat some operations. The analysis may or may not be application-specific; a non-application-specific analysis would consider only the standard reads and updates.

One possibility is to automatically checkpoint at all updates, so such analysis does not hold up the short transaction but instead only cuts down the efficiency of the long transaction (which is probably as it should be). This also has the advantage of minimal work lost in crashes.

Unfortunately, frequent checkpoints leads to a proliferation of versions of the updated resources. Many of these versions will never be seen by any transactions except the open-ended activity and such "obsolete versions" may be garbage collected automatically at intervals [5], perhaps when the database load is low or the storage media is nearly full. Alternatively, if a checkpoint-induced version has not been seen nor will be seen by any past or current transaction, it can be overwritten by the next checkpoint-induced version.

We have developed a framework for application-specific behavior in the second case, where the history of the long transaction is analyzed to select the best point for split-transaction. Our framework extends the notion of atomic transactions for abstract data types [18]. The most significant difference is we treat relatively large tools that manipulate the internal representation of multiple data types, rather than being restricted to individual operations on a single data type. Most existing tools used for open-ended activities fall into the former category. For example, a compiler reads a source program data structure and produces an error message data structure and/or an object code data structure. A debugger reads the object code produced by the compiler and generates traces.

The framework is based on a two-dimensional matrix, where the axes are the data types and the tools that update the data types. The values in the matrix are boolean, where "true" means it is possible to reproduce the object automatically by invoking the tool on appropriate other objects and "false" means this is not possible. In general, false entries correspond to interactive tools such as drawing utilities and true entries to non-interactive tools such as routers.

This matrix is used when there is a short transaction that needs to update an object of a particular data type and an ongoing long transaction is holding the object. The history of the long transaction is examined for the most recent update operation performed by any tool

on that object where the corresponding matrix entry is "false". It is safe to split-transaction after this update, but not before, since updates after the split point will have to be redone after the short transaction commits. For example, it is ok for a short transaction to interrupt a bug-fixing transaction after the programmer has finished his last edit on a source code file, since subsequent operations on that file such as pretty-printing, style-checking and compilation can be re-run automatically with minimal disruption but the actual edits should not be automatically invalidated by the system.

Note that if a tool with a false entry is currently in operation, all work done so far during that tool invocation — or since the last checkpoint — will have to be repeated later. The case where the short transaction only needs to read an object is much simpler — the roll-back of currently executing tools is handled just like a crash in the previous section on fault tolerance. Fortunately, we expect that most short transactions that access the same objects as long transactions to be in fact read-only.

5 Implementation Issues

In the first place, we note that the split-transaction is an operation that is independent of any particular concurrency control or crash recovery methods. Regardless of the implementation technique chosen in a database system, as long as it guarantees serializability and reliability atomicity, split-transaction and join-transaction can be added to the database system.

The general algorithm for split-transaction contains three steps. Frist, we create a new unique transaction id (TID), say for B. Second, we add B's TID to the reader's list of all the objects in `BReadSet`. Third, we reassign B's TID to be the writer of all the objects in `BWriteSet`. Specifically, for two-phase locking, we just add or assign B's TID to the appropriate lock owner list. For timestamp-based methods, we update the timestamps for the objects with B's TID. For optimistic certification methods, we just create the new sets for B. Transaction A retains T's original transaction id. Threads of control in A and B are transferred to AProcedure and BProcedure, respectively.

From the concurrency control point of view, the transfer is relatively straightforward. We will consider the three major techniques: two-phase locking, timestamps and optimistic concurrency control. For two-phase locking, the database system simply changes the ownership of the locks in BReadSet and BWriteSet to the new transaction. For timestamps, the database system needs to assign a new timestamp to B, preferably immediately after A's timestamp. Multidimensional timestamps [10] is one of the techniques that provides this capability. For optimistic concurrency control, since synchronization is done only at commit, the only thing to be done at split-transaction is to move the workspaces of BWriteSet to the new transaction.

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), we simply make 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 objects from the old transaction to the new transaction. During the log processing for recovery, the transition 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.

The algorithms to verify the three properties of section 2.1 are exactly the same of the certification algorithms of optimistic concurrency control methods. Given the history of data access for both A and B, we build the access dependency graphs and check for conflicts. For rigorous serializability, we need to perform data flow analysis on the internal access to shared variables.

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. Since long-transactions have low throughput by definition, the additional overhead will not become a bottleneck.

Another important concern is the restricted case of cascaded aborts introduced by the serial case (Section

2.1). If A aborts then B must abort, since B has read objects 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 database 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.

The implementation of join-transaction is also straightforward. Say that the database system is trying to join-transaction T and S. It can add TReadSet to SReadSet and TWriteSet to SWriteSet. If S is aware of T's joining, then S can start accessing objects read and written by T. This general algorithm works for all the concurrency control methods and crash recovery techniques mentioned above for split-transaction.

6 Comparison

The development of tools and concepts for open-ended activities, including CAD/CAM and programming environments, motivated work in both the database community and the object-oriented programming languages community. Both areas have embraced the term "object-oriented data bases" or object bases as the standard support for what we call open-ended activities. Our work is orthogonal to the object base work since we concentrate on a new way to maintain serializable access to a database instead of complex object structures and exploitation of sophisticated operation semantics.

For several reasons split-transaction is orthogonal to the hierarchical model proposed by Bancilhon et al. [1] (summarized in section 1). First, in their model abort and commit of each level imply the abort and commit of all constituents. In contrast, split-transaction introduces the possibility of committing partial results. Second, their model allows data sharing at each level, but not beyond. Split-transaction may commit partial results and share data broadly thereafter. Finally, we emphasize that split-transactions maintain the transaction serializability, as does their model.

Another representative work on long transactions is Sagas by Garcia-Molina and Salem at Princeton [4]. Sagas are long transactions that consist of a sequence of relatively independent steps, where each step does not have to observe the same consistent database state. Therefore, the results from the previous steps may be "committed" and released to the rest of the concurrent transactions. If interrupted, a saga may try to proceed by executing the missing steps. There is no difficulty with such a forward recovery. However, in some cases a saga may be unable to proceed with forward recovery and it has to abort with backward recovery, in which case client-supplied compensation transactions are required.

Split-transaction is also orthogonal to sagas. Sagas are composed of normal, short-duration transactions that form the steps. Split-transaction is an operation internal to any kind of atomic transaction, including the short-duration ones. More technically, sagas may observe intermediate results of other sagas, while split-transaction produces atomic transactions by definition.

Altruistic locking [14] is an extension to two-phase locking that permits early release of locks without sacrificing serializability. Short transactions are allowed to execute in the wake of a long transaction that altruistically releases locks that it will not use again. In comparison, split-transaction is not an extension of any particular method, and applies to both crash recovery and concurrency control.

To the best of our knowledge, the earliest use of the term *split* and *join* of transactions was in a technical report by Jessop et al. [7]. However, the emphasis of that work was on the Eden Transactional File System and they described these operations very informally; the split corresponded roughly to our independent case.

7 Conclusion

We have informally introduced the class of open-ended activities, which include CAD/CAM design and program development in CASEs. These activities have become an important application area for which the next generation of databases are aiming (e.g. POSTGRES [16]).

To provide consistent database access in open-ended activities, we have introduced the notions of split-transaction and join-transaction. Split-transaction divides the objects in an atomic transaction into two sets of objects contained in two serializable transactions. We have defined the syntax of these operations, their semantics, and the properties that they should satisfy. We also outlined the algorithms that implement these operations and gave examples of safe split-transactions.

We have demonstrated the use of split-transactions in several real-world problems in the class of open-ended activities, such as editing, computer aided design, and program development. The split-transaction solved problems in single-user design activities and multiple-user coordination.

In summary, traditional transaction access to databases provides concurrency atomicity and reliability atomicity, both very desirable properties for open-ended activities. However, the uncertain duration, uncertain developments, and interaction with others in open-ended activities are formidable challenges to the traditional transactions. With split-transactions, we hope to take advantages of transaction atomicity for consistency and reliability at the same time as we increase flexibility in the saving of completed work, early release of results to the world, and consistent interaction with concurrent activities.

Acknowledgements

One of the referees made comments that improved the presentation of the paper considerably. Pu is supported in part by a grant from DEC and in part by the Center of Advanced Technology. Kaiser is supported in part by grants from AT&T, IBM, Siemens and Sun, in part by the Center of Advanced Technology and by the Center for Telecommunications Research, and in part by a DEC Faculty Award.

References

[1] F. Bancilhon, W. Kim, and H.F. Korth.
A model of CAD transactions.
In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pages 25–33, Stockholm, August 1985.

[2] S. Ceri and S. Owicki.
On the use of optimistic methods for concurrency control in distributed databases.
In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, Lawrence Berkeley Laboratory, University of California, Berkeley, February 1982.

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

[4] H. Garcia-Molina and K. Salem.
Sagas.
In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 249–259, May 1987.

[5] A. Nico Habermann.
Automatic deletion of obsolete information.
The Journal of Systems and Software, 5(2):145–154, May 1985.

[6] Norman Heyrowitz, editor.
Object-Oriented Programming Languages, Systems, and Applications, 1987 Conference Proceedings, Orlando, October 1987.

[7] 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.

[8] 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.

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

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

[11] J.E.B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, Massachusetts Institute of Technology, April 1981.

[12] C. Pu.
From nested transactions to supertransactions.
1987.
Submitted for publication.

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

[14] K. Salem and H. Garcia-Molina.
Altruistic Locking: A Strategy for Coping with Long Lived Transactions.
Technical Report CS-TR-087-87, Department of Computer Science, Princeton University, April 1987.

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

[16] M. Stonebraker and L. Rowe.
The design of POSTGRES.
In *Proceedings of 1985 SIGMOD International Conference on Management of Data*, pages 374–387, ACM/SIGMOD, 1985.

[17] S.M. Thatte, editor.
Report on the Object-Oriented Database Workshop: Implementation Aspects, Orlando, October 1987.
Held in conjunction with OOPSLA'87.

[18] W.E. Weihl.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, March 1984.
Tech.Report MIT/LCS/TR-314.